Грант С. Ингерсолл
Томас С. Мортон
Эндрю Л. Фэррис

Обработка неструктурированных текстов

Поиск, организация и манипулирование

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
MANNING

Грант С. Ингерсолл, Томас С. Мортон, Эндрю Л. Фэррис

ОБРАБОТКА НЕСТРУКТУРИРОВАННЫХ ТЕКСТОВ

Поиск, организация и манипулирование

Taming Text

HOW TO FIND, ORGANIZE, AND MANIPULATE IT

GRANT S. INGERSOLL
THOMAS S. MORTON
ANDREW L. FARRIS

MANNING
SHELTER ISLAND

Обработка неструктурированных текстов

ПОИСК, ОРГАНИЗАЦИЯ И МАНИПУЛИРОВАНИЕ

ГРАНТ С. ИНГЕРСОЛЛ
ТОМАС С. МОРТОН
ЭНДРЮ Л. ФЭРРИС

ДМК издательство
Москва, 2015

УДК 004.738.52
ББК 32.972.53
И59

И59 Грант С. Ингерсолл, Томас С. Мортон, Эндрю Л. Фэррис
Обработка неструктурированных текстов. Поиск, организация и манипулирование. / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2015. – 414 с.: ил.

ISBN 978-5-97060-144-0

В книге описаны инструменты и методы обработки неструктурированных текстов. Прочитав ее, вы научитесь пользоваться полнотекстовым поиском, распознавать имена собственные, производить кластеризацию, пометку, извлечение информации и автореферирование. Знакомство с фундаментальными принципами сопровождается изучением реальных применений.

Издание предназначено для читателей без подготовки в области математической статистики и обработки естественных языков. Примеры написаны на Java, но сами идеи могут быть реализованы на любом языке программирования.

УДК 004.738.52
ББК 32.972.53

ISBN 978-1-93398-838-2 (англ.)
ISBN 978-5-97060-144-0 (рус.)

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Во времена, когда спрос на высококачественные средства обработки текста растет экспоненциально, трудно назвать хотя бы одну отрасль экономики, которая не зависела бы от той или иной текстовой информации. А в связи с развитием веб-экономики эта зависимость только усиливается. И вместе с ней быстро возрастает потребность в талантливых технических специалистах. Вот в таких условиях выходит на свет отличная, практически ориентированная книга «Обработка неструктурированных текстов», в которой вы найдете проверенные на реальном опыте рекомендации и инструкции.

Грант Ингерсолл и Дрю Фэррис, два блистательных и в высшей степени квалифицированных инженера-программиста, с которыми я работала много лет, и Тим Мортон, внесший немалый вклад в обработку естественного языка (natural language processing, NLP), предлагают прагматическое руководство тем, кто хотел бы войти в избранный круг специалистов по обработке текстов,

Грант, Дрю и Том выбрали подход, который я называю «обучение на практике ради практики», и сумели сорвать покров тайны с действительно очень сложных процессов. Для этого они не пошли по длинному пути – теоретическому семестровому курсу по NLP, а сосредоточились на существующих инструментах, реализованных до конца примерах и хорошо протестированном коде.

Для инженера-программиста этих основ будет достаточно, чтобы открыть дверь в мир примеров и упоминаемых проектов с открытым исходным кодом. И гораздо быстрее, чем вам кажется, вы превратитесь в настоящего эксперта, готового к решению реальных задач.

Лиз Лидди
Декан, ISchool,
Сиракузский Университет

ВСТУПЛЕНИЕ

Жизнь полна удивительных сюрпризов, и некоторые из них оказали определяющее влияние на мою карьеру. Было это в конце 1990 годов, я тогда был молодым программистом, занимался моделированием распределения электромагнитных полей и случайно наткнулся на предложение места разработчика в небольшой компании в городе Сиракузы, штат Нью-Йорк, которая называлась TextWise. Прочитав описание работы, я подумал, что едва ли подойду, но решил все-таки попробовать и отправил резюме. Непонятно почему, меня взяли, и так началась моя карьера в области обработки естественных языков. Кто бы мог подумать, что спустя столько лет я так и буду заниматься поиском и NLP и даже напишу книгу на эту тему.

А тогда моей первой задачей стало участие в разработке межъязыковой информационно-поисковой системы (CLIR), которая позволяла пользователю вводить запросы на английском языке, а находить и автоматически переводить документы на французском, испанском или китайском. Оглядываясь назад, я понимаю, что в первой же системе, над которой я работал, встретились все те трудные проблемы работы с текстом, которые я впоследствии так полюбил: поиск, классификация, извлечение информации, машинный перевод, а также специфические правила конкретных языков, способные свести с ума любого, кто изучает грамматику. После этого проекта я работал над самыми разными системами поиска и обработки естественных языков – от классификаторов на основе правил до вопросно-ответных систем. Позже, в 2004 году, уже на новой работе в Центре обработки естественных языков я столкнулся с Apache Lucene, поисковой системой с открытым исходным кодом, которая в те дни являлась стандартом де факто. И снова мне пришлось разрабатывать CLIR-систему, только теперь для английского и арабского языков. Поскольку мне потребовались кое-какие функции, которых в Lucene не было, я начал писать дополнения и исправлять ошибки. Спустя какое-то время я стал отправлять плоды своих трудов в репозиторий исходного кода. И пошло-поехало. Я пристрастился к проектам с открытым исходным

кодом, начав с системы машинного обучения Apache Mahout вместе с Изабель Дрост (Isabel Drost) и Карлом Веттином (Karl Wettin), а потом стал сооснователем компании Lucid Imagination, которая специализировалась на задачах поиска и анализа текстов с применением Apache Lucene и Solr.

Описав полный круг, я пришел к выводу, что поиск и NLP принадлежат к числу вопросов, определяющих предмет информатики, поскольку требуют изощренных подходов как к структурам данных, так и к алгоритмам решения задач. Добавьте сюда требования к масштабируемости, необходимой для обработки гигантских объемов данных, порождаемых пользователями веб вообще и социальных сетей в частности, – и вот вам мечта любого разработчика. Эта книга призвана заполнить пустующую на тот момент рыночную нишу – текст, написанный инженерами для инженеров и посвященный, прежде всего, использованию существующих, проверенных практикой библиотек с открытым исходным кодом для решения трудных задач обработки текста. Надеюсь, что она поможет вам в повседневной работе, а также откроет мир текстовых данных – богатейшую возможность для изучения нового.

Грант Ингерсолл

Я подпал под очарование искусственного интеллекта на втором курсе вуза, а на старших курсах решил остаться в аспирантуре и сосредоточиться на обработке естественных языков. В Пенсильванском университете я очень много узнал об обработке текстов, машинном обучении, а также об алгоритмах и структурах данных вообще. У меня также была возможность работать с некоторыми из лучших специалистов в области обработки естественных языков и набираться у них ума-разума.

В аспирантуре я занимался различными NLP-системами и принимал участие в ряде финансируемых агентством DARPA исследований по кореференции, свертыванию и порождению ответов на вопросы. В ходе этой работы я познакомился с системой Lucene и движением за ПО с открытым исходным кодом в целом. Я также обратил внимание на пробел в открытых программах обработки текстов, заполнение которого могло бы обеспечить эффективную сквозную обработку. Работая над диссертацией, я активно участвовал в проекте OpenNLP, а также продолжал изучать NLP-системы, разрабатывая систему автоматизированной оценки сочинений и кратких ответов в службе образовательного тестирования (Educational Testing Services).

Тесное сотрудничество с разработчиками ПО с открытым исходным кодом научило меня коллективной работе и позволило усовершенствоваться в своей профессии. Сейчас я работаю в компании Comcast Corporation с командами программистов, которые применяют многие описанные в этой книге приемы и инструменты. Надеюсь, эта книга станет мостом между напряженно ищущими исследователями типа тех, у кого я учился в аспирантуре, и инженерами-практиками, цель которых – использовать обработку текстов для решения реальных задач в интересах обычных людей.

Томас Мортон

Я, как и Грант, получил первое представление об информационном поиске и обработке естественных языков под руководством д-ра Элизабет Лидди, Вуджина Пайка (Woojin Paik) и прочих сотрудников компании TextWise в середине 1990-х годов. В то время TextWise была в стадии превращения из исследовательской группы в новообразованную компанию, специализирующуюся на разработке приложений на основе полученных результатов в области обработки текста. Я работал в компании много лет и все это время занимался самообразованием, открывал для себя что-то новое и общался с выдающимися людьми, которые, не убоявшись трудностей, решили научить машины понимать различные аспекты человеческого языка.

Лично я подхожу к проблеме анализа текста, прежде всего, с точки зрения разработчика программного обеспечения. Мне повезло работать с блестящими учеными и превращать их идеи из экспериментов сначала в функционирующие прототипы, а затем и в массивно масштабируемые системы. По ходу дела у меня была возможность плотно заниматься тем, что теперь принято называть наукой о данных, и я глубоко и навсегда полюбил исследование больших наборов данных и методы извлечения информации из них.

Невозможно переоценить то огромное влияние, которое открытое ПО оказало на мою карьеру. Наличие под рукой исходного кода как подспорья в исследованиях, – невероятно эффективный способ изучения новых методов и подходов к анализу текста и к разработке ПО вообще. Я приветствую всякого, кто приложил усилия, чтобы поделиться своими знаниями и опытом с другими людьми, страстно желающими сотрудничать и учиться. И особо я хочу поблагодарить отличных ребят из фонда Apache Software Foundation, неустанно взращивающих динамичную экосистему, которая способствует раз-

работке открытого ПО и помогает организовывать процессы и сплачивать людей, это ПО поддерживающих.

Инструменты и методы, описанные в этой книге, своими корнями уходят глубоко в сообщество разработчиков ПО с открытым исходным кодом. Lucene, Solr, Mahout и OpenNLP – все эти проекты выросли под опекой Apache. В этой книге мы лишь скользнули по поверхности того, что умеют эти инструменты. Нашей целью было продемонстрировать базовые концепции, лежащие в основе обработки текстов, и заложить прочный фундамент под будущие исследования в этой области.

Успехов в кодировании!

Дрю Фэррис

БЛАГОДАРНОСТИ

Работа над книгой заняла немало времени, и в ней принимало участие множество людей, которым мы с радостью выражаем свою признательность. Мы благодарны:

- пользователям и разработчикам Apache Solr, Lucene, Mahout, OpenNLP и других инструментов, упоминаемых в этой книге;
- издательству Manning Publications, не бросившему нас, а в особенности Дугласу Пандику (Douglas Pundick), Карен Тегтмейер (Karen Tegtmeyer) и Брайану Бейсу (Marjan Bace);
- Джеффу Блейелю (Jeff Bleiel), нашему выпускающему редактору, который всю дорогу подгонял нас, невзирая на наш безумный график, всегда имел наготове похвалу и сумел превратить разработчиков в авторов;
- наших рецензентов за вопросы, комментарии и критические замечания, благодаря которым книга стала лучше: Адама Тейси (Adam Tacy), Амоса Бэннистера (Amos Bannister), Клинта Ховарта (Clint Howarth), Костантино Чербо (Costantino Cerbo), Давида Вайсса (Dawid Weiss), Дениса Куриленко (Denis Kurilenko), Дуга Уоррена (Doug Warren), Фрэнка Джаниа (Frank Jania), Ганна Бирнера (Gann Bierner), Джеймса Хатуэя (James Hatheway), Джеймса Уоррена (James Warren), Джейсона Ренни (Jason Rennie), Джеффри Коупленда (Jeffrey Copeland), Джоша Рида (Josh Reed), Жульена Ниоша (Julien Nioche), Кита Кима (Keith Kim), Маниша Катьяла (Manish Katyal), Маргрит Бруггеман (Margriet Bruggeman), Массимо Перга (Massimo Perga), Никандера Бруггемана (Nikander Bruggeman), Филиппа К. Дженерта (Philipp K. Janert), Рика Вагнера (Rick Wagner), Роби Сена (Robi Sen), Саншета Диге (Sanchet Dighe), Шимона Чойнацки (Szymon Chojnacki), Тима Поттера (Tim Potter), Вайджаната Рао (Vaijanath Rao) и Джеффа Годлшрафе (Jeff Goldschrafe);
- наших соавторов, которые внесли вклад в отдельные разделы книги: Дж. Нила Рихтера (J. Neal Richter), Маниша Катьяла,

Роба Зинкова (Rob Zinkov), Шимона Чойнацки, Тима Поттера и Вайджаната Рао;
- Стивена Роуи (Steven Rowe) за скрупулезное техническое редактирование, а также за многие часы, которые он отдал написанию приложений для обработки текста в компаниях TextWise, CNLP, а также в проекте Lucene;
- д-ра Лиз Лидди за то, что она ввела Дрю и Гранта в мир анализа текстов и познакомила с таящимися в нем удивительными возможностями, а также за написание предисловия к этой книге;
- всех читателей предварительной версии книги за терпение и отзывы;
- а прежде всего, наши семьи, друзей и коллег за ободрение, моральную поддержку и понимание на протяжении всего периода, когда мы были вынуждены отдавать часть своей жизни работе над книгой.

Грант Ингерсолл

Благодарю всех своих коллег по компаниям TextWise и CNLP, которые столь много рассказали мне об анализе текстов; г-на Урдала, который привил мне интерес к математике, и г-жу Реймонд, благодаря которой я стал хорошим студентом и человеком; своих родителей, Флойда и Делорес, и детей, Джеки и Уильяма (всегда вас люблю); свою жену Робин, которая мирилась с моими занятиями допоздна и упущенными выходными – спасибо, что ты сумела вытерпеть все это!

Том Мортон

Благодарю своих соавторов за тяжкий труд и дружеские отношения; свою жену Туи и дочь Хлою за терпение, поддержку и отданное мне время; своих родственников, Мортонов и Транов, за постоянное ободрение; коллег из Пенсильванского университета и компании Comcast за поддержку и сотрудничество, а особенно На-Рай Хан (Na-Rae Han), Джейсона Балбриджа (Jason Baldridge), Ганна Бирнера (Gann Bierner) и Марту Палмер (Martha Palmer); Йорна Котманна (Jцrn Kottmann) за неустанную работу над проектом OpenNLP.

Дрю Фэррис

Благодарю Гранта за привлечение меня к этому и многим другим интересным проектам; своих коллег, прошлых и настоящих, от которых я многому научился и с которыми делил и делю страсть к ана-

лизу текстов, машинному обучению и разработке удивительных программ; свою жену Кристин и детей, Фебу, Одри и Оуэна, за терпение и поддержку, невзирая на то, что я тратил столько времени на это и другие технические предприятия – в ущерб им; всей моей семье за ободрение и проявленный интерес, а особенно маму, которой не суждено посмотреть на эту книгу в завершенном виде.

ОБ ЭТОЙ КНИГЕ

«Обработка неструктурированных текстов» – это книга о создании программных приложений, ценность которых состоит, главным образом, в использовании и манипулировании содержимым обычных письменных текстов. Это не теоретический трактат по поиску, обработке естественного языка и машинному обучению, хотя все эти вопросы обсуждаются довольно подробно. Мы стремились избегать специальной терминологии и сложной математики, а сосредоточиться на концепциях и примерах, необходимых современным программистам, архитекторам и пользователям для реализации интеллектуальных приложений обработки текста нового поколения. Кроме того, наша твердая позиция – демонстрировать примеры из реальной практики с помощью бесплатных, широко распространенных инструментов с открытым исходным кодом – Apache Solr, Mahout, OpenNLP и других.

Предполагаемая аудитория

Будет ли эта книга полезна вам? Возможно. Мы ориентировались на программистов-практиков, не имеющих солидной теоретической подготовки в проблемах поиска, обработки естественного языка и машинного обучения. На самом деле, книга рассчитана на людей, с которыми мы встречались во многих компаниях: команду разработчиков, которой поручено добавить поиск и другие средства в уже существующее приложение при том, что мало кто из них (а то и вовсе никто) не имеет опыта работы с текстом. Им необходимо хорошее введение в суть предмета, не отягощенное ненужными деталями.

Часто мы отсылаем читателя к легко доступным источникам типа википедии и фундаментальным научным статьям, тем самым подготавливая стартовую площадку для, кто хотел бы изучить предмет более подробно. И еще – хотя большинство инструментов и примеров написаны на Java, сами идеи легко переносятся на многие другие языки программирования, поэтому пишущие на Ruby, Python или еще каком-то языке тоже получат пользу от чтения этой книги.

Эта книга определенно не подойдет тем, кому интересны объяснения математических основ описываемых систем, или тем, кто жаждет академической строгости изложения, хотя, как нам кажется, она пригодится студентам, когда перед ними встанет задача реализовать идеи, почерпнутые из лекций и других книг академической направленности.

Не рассчитана эта книга и на опытных специалистов-практиков, которые за свою карьеру написали не одно приложение для обработки текстов, хотя и они смогут найти в ней подсказку-другую о том, как работать с описываемыми пакетами с открытым исходным кодом. Не раз мы слышали от практиков, что эта книга очень помогает, когда нужно быстро обучить новых членов команды концепциям, относящимся к созданию приложений для обработки текстов.

В общем и целом, мы надеемся, что эта книга станет актуальным пособием для современного программиста, пособием, которого всем нам так не хватало, когда мы только начинали свою карьеру в области обработки текста.

Структура книги

В главе 1 объясняется, почему задача обработки текста важна и в чем ее трудность. Мы дадим предварительный обзор вопросно-ответной фактографической системы, подготовив сцену для «приручения» текста с применением открытых библиотек.

В главе 2 описываются основные элементы обработки текста: лексический анализ, разбиение на блоки, грамматический разбор и частеречная разметка. Затем мы поговорим о том, как извлекать текст из файлов в распространенных форматах с помощью проекта Apache Tika с открытым исходным кодом.

Глава 3 посвящена теории поиска и основам векторной модели. Мы познакомимся с поисковым сервером Apache Solr и покажем, как с его помощью индексировать документы. Вы узнаете о количественных и качественных оценках работы поисковой системы.

В главе 4 рассматривается неточный поиск в строке с помощью префиксов и *n*-грамм. Мы рассмотрим две характеристики близости строк – меру Жаккарда и расстояние Джаро-Винклера – и объясним, как с помощью Solr находить и ранжировать соответствия.

В главе 5 представлены основные концепции распознавания именованных сущностей. Мы покажем, как находить именованные сущности с помощью проекта OpenNLP, и обсудим некоторые аспекты его функционирования. Мы также рассмотрим вопрос о на-

стройке OpenNLP на распознавание сущностей новой предметной области.

Глава 6 посвящена кластеризации текста. Из нее вы узнаете об основах стандартных алгоритмов кластеризации текстов и увидите, как кластеризация может повысить качество приложения. Мы также объясним, как производить кластеризацию целых наборов документов с помощью Apache Mahout и как кластеризовать результаты поиска с помощью Carrot[2].

В главе 7 обсуждаются основы классификации, категоризации и грамматической разметки. Мы покажем, как категоризация применяется в приложениях для обработки текста и как можно построить, обучить и использовать классификатор с помощью открытых инструментов. Мы также воспользуемся реализацией алгоритма наивной байесовской фильтрации в проекте Mahout для построения категоризатора документов.

Графические выделения и загрузка исходного кода

В этой книге много примеров кода. Код выделяется моноширинным шрифтом, чтобы было проще отличить его от обычного текста. Элементы программы, например имена методов, классов и т. д., также выделяются моноширинным шрифтом.

Многие листинги сопровождаются аннотациями, иллюстрирующими основные идеи, и пронумерованными маркерами, на которые даются ссылки в последующих пояснениях.

Многие приведенные в книге примеры довольно близки к тем, что можно найти в сети. Но для краткости мы иногда удаляли некоторые части, например комментарии, чтобы код поместился на странице.

Исходный код к этой книге можно скачать с сайта издательства по адресу www.manning.com/TamingText.

Автор в сети

Приобретение книги «Обработка неструктурированных текстов» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/TamingText. Там же написано, как зайти на форум после регис-

трации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору какие-нибудь хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

ОБ ИЛЛЮСТРАЦИИ НА ОБЛОЖКЕ

Рисунок на обложке книги называется «Le Marchand» – купец, или лавочник. Он взят из изданного в 19-ом веке во Франции четырехтомного собрания местных костюмов, опубликованного Сильвеном Марешалем. Все иллюстрации в нем превосходно нарисованы и раскрашены вручную. Богатое разнообразие собрания Марешаля показывает, как сильно города и области различались в культурном отношении каких-то 200 лет назад. Отделенные друг от друга, люди говорили на разных языках и диалектах. Встретив человека на улице города или в деревне, по его одежде легко было определить, откуда он родом и чем занимается.

Манера одеваться с тех пор сильно изменилась, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов, что уж говорить о деревеньках или городках, разделенных несколькими километрами. Мы обменяли культурное разнообразие на иное устройство личной жизни – основанное на многостороннем и стремительном технологическом развитии.

Во времена, когда одну книгу по компьютерам трудно отличить от другой, издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из собрания Марешаля.

ГЛАВА 1. Готовимся к приручению текста

В этой главе:

- Почему так важна задача обработки текста.
- В чем сложность обработки текста.
- Подготовка к использованию библиотек с открытым исходным кодом для обработки текста.

Раз вы читаете эту книгу, то, наверное, вы программист или, по крайней мере, подвизаетесь в области информационных технологий. Вы без особых проблем работаете с электронной почтой, системами мгновенного обмена сообщениями, Google, YouTube, Facebook, Twitter, блогами и большинством других технологий, определяющих облик нашей цифровой эпохи. Но поздравив себя с собственным техническим мастерством, задумайтесь о своих пользователях. Зачастую они испытывают муки от одного лишь объема получаемой электронной почты. Они изо всех тщатся как-то организовать данные, чтобы не утонуть в них. И, возможно, они не знают и знать не хотят о всяких там RSS или JSON, не говоря уже о поисковых системах, байесовских классификаторах или нейронных сетях. Они хотят получать ответы на свои вопросы, не просеивая многие страницы результатов. Они хотят, чтобы их почта была организована и упорядочена по важности, но при этом не желают тратить на это собственное время. И вообще, пользователям нужны инструменты, которые позволяют сосредоточиться на своей жизни и работе, а не на том, как они устроены. Они хотят контролировать – приручить – дикого зверя, каким является *текст*. Но что значит «приручить текст»? Об этом мы и будем гово-

рить далее, а пока скажем, что приручение текста подразумевает три основных вещи:

- умение находить ответы и подкрепляющее их содержимое, удовлетворяющие информационные потребности;
- умение организовывать текст (ставить пометки, делать извлечения, составлять реферат) и манипулировать им почти или совсем без вмешательства пользователя;
- умение решать обе вышеуказанные задачи в условиях постоянного роста объема входной информации.

Это подводит нас к основной цели настоящей книги: предоставить вам, программисту, средства и практические рекомендации для создания приложений, которые позволят людям лучше справляться с приливной волной коммуникаций, грозящей затопить их жизнь. Другая цель этой книги – показать, как использовать существующие, бесплатные, высококачественные библиотеки и инструменты с открытым исходным кодом.

Но прежде чем вплотную заняться декларированными целями, отступим на шаг и посмотрим, из чего состоит обработка текста и почему эта задача так трудна, а также обратимся к некоторым примерам, на которых будем иллюстрировать материал в последующих главах. Конкретно, мы дадим предварительный обзор приложения, которое будет построено к концу книги: фактографической вопросно-ответной системы. Имея это в виду, приведем обоснования для обработки текста, рассмотрев размер и форму информационного мира, в котором мы живем.

1.1. Почему так важна задача обработки текста

Забавы ради попробуйте представить день, в течение которого вы не прочли ни единого слова. Да-да, именно так: целый день без новостей, вывесок, веб-сайтов и даже телевизора.

Думаете, получилось бы? Вряд ли, разве что вы целый день проспите. А теперь подумайте, что предшествовало чтению всего этого добра: годы учебы в школе, практическое освоение в ходе общения с родителями, учителями и сверстниками, бесконечные диктанты, уроки грамматики, изложения, не говоря уже о сотнях тысяч долларов, которые тратятся на обучение одного человека в колледже. А теперь остановитесь и подумайте, сколько всего вы действительно читаете за день.

Для начала попробуйте ответить на следующие вопросы.

- Сколько вы сегодня получили сообщений по электронной почте (личных и рабочих, включая спам)?
- Сколько из них вы прочли?
- На сколько ответили немедленно? В течение часа? Дня? Недели?
- Как вы находили старое почтовое сообщение?
- Сколько блогов вы сегодня прочли?
- Сколько новостных сайтов вы посетили?
- Общались ли вы с друзьями или коллегами в чатах, Twitter или Facebook?
- Сколько раз вы искали информацию с помощью Яндекса, Google, Yahoo! или Bing?
- Сколько электронных документов вы прочли на своем компьютере? В каком формате они были (Word, PDF, простой текст)?
- Как часто вы искали что-то локально (на собственной машине или в корпоративной сети Интранет)?
- Сколько нового содержимого вы создали в виде почтовых сообщений, отчетов и т. д.?

И наконец, главный вопрос: сколько времени вы на все это потратили? Если вы типичный информационный работник, то, скорее всего, к вам относятся следующие данные, опубликованные корпорацией IDC (International Data Corporation) по результатам исследований, проведенных в 2009 году (Feldman [2009]):

> Работа с электронной почтой отнимает в среднем 13 часов в неделю... Но электронная почта уже не является единственным средством коммуникации. Социальные сети, системы мгновенного обмена сообщениями, Yammer, Twitter, Facebook и LinkedIn – вот новые каналы общения, которые могут красть рабочее время у информационного работника. В этом году среднее время, затраченное на поиск информации, составило 8,8 часов в неделю, т. е. в расчете на одного работника 14 209 долларов в год. На анализ информации уходит еще 8,1 часов, что обходится организации в 13 078 долларов ежегодно. Таким образом, эти две задачи являются очевидными кандидатами на автоматизацию. Раз уж работники тратят треть времени на поиск информации и еще четверть на ее анализ, то эта деятельность должна быть сделана максимально продуктивной.

В этом исследовании даже не учитывается, сколько времени те же самые работники тратят на создание нового содержимого в свобод-

ное от работы время. На самом деле, по оценке компании eMarketer, средний пользователь Интернета проводит в сети 18 часов в неделю; для сравнения отмечается, что на просмотр телевизора, который по-прежнему занимает доминирующее место среди развлечений, уходит 30 часов в неделю.

Так что писаное слово – будь то чтение электронной почты, поиск в Google, чтение книг или просмотр Facebook – присутствует в нашей жизни повсеместно.

Мы познакомились с той частью картины создания содержимого, которая связана с отдельными лицами. А как обстоит дело с содержимым, порождаемым коллективно? По данным IDC (за 2011 год), в 2011 году в мире было создано *1,9 зетабайт* цифровой информации, а «к 2020 году мир произведет в 50 раз больше [этой величины]». Естественно, такие прогнозы зачастую оказываются заниженными, поскольку мы не можем предсказать очередную возникшую ниоткуда тенденцию, из-за которой объем содержимого превысит ожидания.

Пусть даже добрая часть этих данных приходится на сигнальные данные, изображения, аудио и видео, все равно нужно обеспечить возможность находить их в сети, и сейчас для этой цели применяются такие подходы, как составление аналитических отчетов, добавление ключевых слов-меток и текстовых описаний, или преобразование аудиоданных в текст с помощью средств распознавания речи или ручного добавления титров. Иными словами, любая добавленная нами структурная информация оказывается текстом, который помогает понять смысл содержимого и сделать его общедоступным. Как видите, объем содержимого ошеломляет сам по себе, и это вдобавок к тому, что, как мы убедимся в следующем разделе, обработка текста является сложной задачей даже в небольшом масштабе. А пока полезно было бы подумать о том, что должны уметь идеальные приложения или инструменты, дабы обуздать наступающий на нас шквал текстовой информации. Для многих это умение быстро и эффективно отвечать на вопросы, а не просто выдавать перечень возможных ответов, который мы потом должны просматривать вручную. Более того, само задание вопроса не должно сопровождаться громоздкими ритуалами; мы хотим просто писать или произносить слова, не затрудняя себя всякими кавычками, операторами И/ИЛИ и другими вещами, которые упрощают работу машине, но усложняют человеку.

И хотя мы знаем, что наш мир не идеален, один из многообещающих подходов к приручению текста, популяризируемый программой

IBM Watson для игры в Jeopardy![1] и приложением Siri от компании Apple, – это вопросно-ответная система, способная обрабатывать вопросы на естественных языках, например английском, и возвращать *настоящие* ответы, а не просто список *возможных* ответов. В этой книге мы собираемся заложить фундамент для построения такой системы. Для этого подумаем, как она могла бы выглядеть, а затем рассмотрим простой код, который умеет находить и извлекать из текста полезную информацию; впоследствии этот код найдет применение в нашей вопросно-ответной системе. И завершим мы эту главу мыслями о том, почему построение подобной системы, равно как и других приложений для обработки естественного языка, оказывается такой трудной задачей. А, кроме того, опишем, как в последующих главах будет возводиться здание вопросно-ответной системы, а заодно и других систем для работы с текстом.

1.2. Предварительный обзор фактографической вопросно-ответной системы

С точки зрения этой книги, вопросно-ответная система (ВО-система) должна обрабатывать набор документов, который теоретически мог бы содержать ответы на интересующие пользователя вопросы. Например, источником ответов может быть википедия или подборка научно-исследовательских статей. Иными словами, предлагаемая нами ВО-система основана на выявлении и анализе текста, из которого можно было бы получить ответ, опираясь на то, что она видела в прошлом. Она не сможет выводить ответ из многих разнообразных источников. Например, если спросить систему «Кто является дядей Боба?» и если в наборе имеется документ, содержащий фразы «Отцом Боба является Ола. Братом Ола является Пауль.», то система не сможет сделать вывод, что дядей Боба является Пауль. Но если имеется фраза, в которой прямо утверждается, что «дядей Боба является Пауль», то мы ожидаем, что система сможет ответить на вопрос. Сказанное не означает, что первая задача неразрешима, просто она выходит за рамки этой книги.

Схема простой ВО-системы, описанной выше, приведена на рис. 1.1.

Разумеется, на этой простой схеме не показаны многочисленные детали и не отражена подача на вход документов, однако основные

[1] Российский аналог – «Своя игра». – *Прим. перев.*

компоненты обработки вопросов пользователей все же представлены. Во-первых, чтобы разобрать вопрос и определить, что спрашивается, обычно требуется такая базовая функциональность, как выделение слов, а равно способность понять, ответ какого типа подходит для данного вопроса. Например, ответом на вопрос «Кто является дядей Боба?», вероятно, должно быть имя человека, а на вопрос «Где находится Буффало?» – название места. Во-вторых, для поиска потенциальных ответов обычно нужно уметь быстро находить фразы, предложения или фрагменты, содержащие потенциальные ответы, не заставляя систему разбирать большие куски текста.

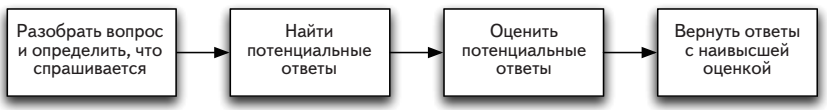

Рис. 1.1. Порядок обработки информации для получения ответов на вопросы, задаваемые простой ВО-системе

Для оценки также нужны многие базовые функции, например выделение слов, а также более глубокое понимание того, содержит ли потенциальный ответ необходимые компоненты, например упоминание человека или места. Кое-что из вышеупомянутого кажется тривиальным, если принять во внимание, с какой легкостью это делает – а точнее *думает*, что делает, – человек, однако при ближайшем рассмотрении все оказывается совсем не так просто. Памятуя об этом, рассмотрим пример обработки блока текста для поиска фрагментов и выявления разных интересных вещей, например, имен.

1.2.1. Здравствуй, доктор Франкенштейн

В свете обсуждения нашей вопросно-ответной системы и трех основных задач при работе с текстом рассмотрим пример простой операции обработки текста. Естественно, нам понадобится какой-нибудь текст. Для этой цели мы решили взять классический роман Мэри Шелли «Франкенштейн». Почему именно он? Помимо того, что он нравится авторам с литературной точки зрения, это первая книга, на которую мы наткнулись на сайте проекта Гутенберг (http://www.gutenberg.org/). К тому же, это простой текст с приличным форматированием (что для текстов, встречающихся нам в повседневной жизни, – большая редкость). Дополнительное преимущество – отсутствие копирайта и возможность бесплатно загрузить со страницы http://www.gutenberg.org/cache/epub/84/pg84.txt.

Итак, текст у нас есть. Теперь решим несколько задач, которые снова и снова возникают в приложениях для обработки текста:

- найти в тексте заданную пользователем строку и вернуть релевантный фрагмент (в данном примере абзац);
- разбить фрагмент на предложения;
- выделить из текста «интересные» вещи, например, имена людей.

Для решения этих зада мы воспользуемся двумя написанными на Java библиотеками, Apache Lucene и Apache OpenNLP, а также кодом в прилагаемом к этой книге файле com.tamingtext.frankenstein. Frankenstein, который можно скачать с сайта GitHub по адресу at http://www.github.com/tamingtext/book. В файле https://github.com/tamingtext/book/blob/master/README приведены инструкции по сборке программы из исходного кода.

В листинге ниже приведен код, управляющий процессом.

Листинг 1.1. Управляющий код для программы Frankenstein

```
Frankenstein frankenstein = new Frankenstein();
frankenstein.init();
frankenstein.index();   <-- Обеспечить возможность поиска по содержимому
String query = null;
while (true) {
 query = getQuery();    <-- Получить запрос от пользователя
 if (query != null) {
  Results results = frankenstein.search(query);   <-- Выполнить поиск
  frankenstein.examineResults(results);   <-- Разобрать результат и показать интересные элементы
  displayResults(results);
 } else {
  break;
 }
}
```

Управляющая программа первым делом индексирует содержимое. Индексирование – это процесс, в результате которого по содержимому становится возможно производить поиск. Мы применяем для построения индекса Lucene. Подробнее это будет описано ниже в главе, посвященной поиску. А пока считайте, что индекс дает быстрый способ поиска слов в блоке текста. Далее мы входим в цикл, где предлагаем пользователю ввести запрос, выполняем поиск и обрабатываем найденные результаты. В данном примере каждый абзац рассматривается как поисковая единица. Это означает, что при выполнении поиска мы сможем точно узнать, какие абзацы книги отвечают запросу.

Получив абзацы, мы переходим к использованию библиотеки

OpenNLP, которая позволяет разбить каждый абзац на предложения и попытаться выделить в предложениях имена людей. Мы отложим изучение деталей реализации каждого метода до рассмотрения соответствующих концепций в последующих главах. А пока запустим программу, сформулируем запрос и посмотрим на результаты.

Для запуска программы откройте окно терминала, перейдите в каталог, содержащий распакованный исходный код, и введите команду `bin/frankenstein.sh` в UNIX/Mac или `bin/frankenstein.cmd` в Windows. Должны появиться такие строки:

```
Initializing Frankenstein
Indexing Frankenstein
Processed 7254 lines. Paragraphs: 722
Type your query. Hit Enter to process the query \
(the empty string will exit the program):
>
```

Теперь можно ввести запрос, например `"three months"`. Ниже показана часть списка результатов. Квадратные скобки [...] в разных местах добавлены нами в целях форматирования.

```
>"three months"
Searching for: "three months"
Found 4 total hits.
-----------------------------------
Match: [0] Paragraph: 418
Lines: 4249-4255
     "'Do you consider,' said his companion to him, ...
    ----- Sentences ----
       [0] "'Do you consider,' said his companion to him, ...
       [1] I do not wish to take any unfair advantage, ...
-----------------------------------
Match: [1] Paragraph: 583
Lines: 5796-5807
     The season of the assizes approached. ...
     ----- Sentences ----
...    [2] Mr. Kirwin charged himself with every care ...
       >>>> Names
           Kirwin
...    [4] ... that I was on the Orkney Islands ...
       >>>> Locations
           Orkney Islands
-----------------------------------
Match: [2] Paragraph: 203
Lines: 2167-2186
     Six years had elapsed, passed in a dream but for one indelible trac
e, ...
     ----- Sentences ----
```

```
...     [4] ... and represented Caroline Beaufort in an ...
        >>>> Names
             Caroline Beaufort
...     [7] While I was thus engaged, Ernest entered: ...
"Welcome, my dearest Victor," said he. "Ah!
        >>>> Names
             Ah

        [8] I wish you had come three months ago, and then you
would ha ve found us all joyous and delighted.
        >>>> Dates
             three months ago

        [9] ... who seems sinking under his misfortune; and your
pers uasions will induce poor Elizabeth to cease her ...
        >>>> Names
             Elizabeth
...
```

Здесь показаны результаты выборки первых абзацев, в которых встречается фраза «three months» (всего таких абзацев четыре), содержащие избранные предложения из каждого абзаца, а также списки встретившихся имен, дат и географических названий. В данном случае мы видим примеры распознавания предложений, а равно выделения имен, дат и географических названий. Внимательный читатель заметит несколько мест, в которых эта простая система заведомо ошиблась. Например, система полагает, что *Ah* – имя, а *Ernest* – нет. Она также не смогла разбить на предложения текст, заканчивающийся фразой «... said he. "Ah!». Быть может, наша система не знает, как правильно обрабатывать восклицательный знак, или в тексте встречается какое-то странное форматирование.

Пока что оставим в стороне вопрос о причине таких ошибок. Если вы продолжите эксперименты с запросами, то, вероятно, обнаружите самые разные ситуации: хорошие, плохие и откровенно никуда не годные. Но этот пример послужит удачным переходом к следующему разделу, в котором мы коснемся некоторых трудностей, присущих обработке текста. На нем же мы будем демонстрировать различные изучаемые в этой книге подходы.

1.3. Понять смысл текста трудно

Предположим, что Робин и Джо беседуют, и Джо говорит: «The bank on the left is solid, but the one on the right is crumbling». О чем они разговаривают? Находятся ли они на Уолл-стрит и смотрят на

здания двух финансовых учреждений или плывут по Миссисипи на байдарке и ищут, где пристать к берегу?[2] Если предположить первое, то слова *solid* (устойчивый) и *crumbling* (на грани банкротства), скорее всего, относятся к финансовому положению банков, а если второе, то к качеству почвы на берегу реки. А если заменить имена персонажей на *Гек* и *Том* – мальчиков из «Приключений Тома Сойера»? Тогда вы, наверное, не усомнитесь, что речь идет о береге реки, а не о банке, правда? Как видим, контекст тоже важен. Часто бывает, что понять смысл отрывка текста можно, только располагая дополнительной информацией из окружающего контекста и привлекая собственный опыт. Неоднозначность высказывания Джо – лишь простенький пример сложностей, связанных с пониманием смысла текста.

Видя правильно написанные, связные предложения и абзацы, умудренный человек может посмотреть в словаре значения слов и, добавив свой опыт и знание контекста, понять смысл текста или диалога. Образованный взрослый человек может (с большей или меньшей точностью) разобрать предложение, выявить связи между его членами и понять смысл почти мгновенно. И, как в примере с Робином и Джо, люди почти всегда осознают, что в предложении, абзаце или документе в целом что-то находится не на своем месте или вообще отсутствует. Люди также пользуются в диалоге невербальными средствами, мгновенно изменяя тональность и эмоциональную окраску для передачи своих мыслей по различным предметам, от погоды до политики. Мы принимаем эти навыки как нечто само собой разумеющееся, но следует помнить, что они оттачивались годами устного общения, образования и оценки реакции со стороны собеседников, не говоря уже о наследственной памяти, полученной нами от предков.

В то же время компьютеры вообще и отрасли информационного поиска и обработки естественных языков (NLP) в частности еще сравнительно молоды. Компьютеры должны уметь обрабатывать язык на многих уровнях, чтобы хотя бы приблизиться к «пониманию» содержимого, как это делают люди. (Детальное обсуждение многочисленных факторов, которые нужно учитывать в NLP, см. в работе Liddy [2001].) Полное понимание представляется для компьютера непосильной задачей, но даже самые простые вещи могут оказаться чрезвычайно трудными, принимая во внимание объем доступных текстов и разнообразие ситуаций.

---

[2] Одно из значений слова *bank* – банк, другое – берег реки. Crumbling может означать как «близкий к банкротству», так и «осыпающийся». – *Прим. перев.*

Не без причины говорят «числа не лгут», а не «текст не лжет»; текст предстает в самых разных формах и смыслах и регулярно вводит в заблуждение даже самых умных людей. Разработчик приложений для обработки текста сталкивается с различными проблемами технического и нетехнического характера. В табл. 1.1 представлены некоторые возникающие проблемы в порядке возрастания сложности.

Таблица 1.1. При обработке текста возникают проблемы на разных уровнях, от кодировки символов до выведения смысла в контексте окружающего мира

Уровень	Проблемы
Символ	– Кодировки, например: ASCII, Shift-JIS, Big 5, Latin-1, UTF-8, UTF-16. – Регистр (верхний и нижний), знаки препинания, диакритические знаки и числа в разных приложениях должны обрабатываться по-разному.
Слова и морфемы[1]	– Разбиение на слова. Сравнительно просто для английского и других языков, в которых слова разделяются пробелами; гораздо труднее для языков типа китайского и японского. – Определение частей речи. – Выявление синонимов, они полезны при поиске. – Стемминг: процесс выделения основы слова. Например, основой слова «words» является «word». – Аббревиатуры, акронимы и правописание играют важную роль для понимания смысла слов.
Несколько слов и предложение	– Распознавание групп слов; примерами могут служить *quick red fox* (быстрая рыжая лиса), *hockey legend Bobby Orr* (легенда хоккея Бобби Орр) и *big brown shoe* (большой коричневый ботинок). – Грамматический разбор: определение подлежащего и сказуемого, а также других связей между частями предложения часто дает полезную информацию о словах и их взаимосвязях. – Определение границ предложений в английском языке – хорошо изученная, но все же не до конца решенная проблема. – Разрешение кореференции: «Джейсон любит собак, но он никогда не купит ее». Здесь «он» – кореференция для «Джейсон». Необходимость в разрешении кореференции может возникать и для нескольких предложений. – У слов часто бывает несколько значений; для выбора правильного нужно учитывать контекст предложения. Этот процесс называется разрешением лексической неоднозначности, и хорошо реализовать его трудно. – Совместное использование определений слов и их взаимосвязей для выведения смысла предложения.

Таблица 1.1. (окончание)

Уровень	Проблемы
Несколько предложений и абзац	На этом уровне обработка усложняется, ее цель – глубже понять намерения автора. В алгоритмах автореферирования часто требуется выявлять, какие предложения наиболее важны.
Документ	Как и на уровне абзаца, для понимания смысла документа часто необходимы знания, выходящие за рамки самого документа. Автор нередко ожидает, что у читателя имеется определенная подготовка или навыки чтения. Например, читатель, никогда не пользовавшийся компьютером и не писавший программ, не поймет большую часть этой книги, а в большинстве газет предполагается умение читать на уровне хотя бы шестого класса.
Несколько документов и корпус	На этом уровне требуется быстро находить интересующую информацию, а также группировать взаимосвязанные документы и составлять их рефераты. Особенно полезны приложения, которые умеют агрегировать и организовывать факты и мнения и выявлять связи между ними.

[1] Морфемой называется мельчайшая значимая единица языка. Примерами морфем являются префиксы и суффиксы.

Помимо вышеуказанных проблем, при работе с текстом важную роль играет человеческий фактор. Из-за различия культур, языков и интерпретаций одного и того же текста даже у самого лучшего инженера может возникнуть вопрос, а что собственно требуется сделать. Попытка экстраполировать на весь набор документов подход, примененный к немногим выборочным файлам, зачастую оказывается неприемлемым решением. Другая крайность – ручной анализ и аннотирование больших наборов документов – может оказаться чрезмерно затратным и длительным процессом. Но даже не сомневайтесь – текст можно приручить, и помощь придет.

1.4. Прирученный текст

А теперь, увидев, с какими проблемами придется столкнуться, воспряньте духом: для решения этих и многих других задач существует множество инструментов – как коммерческих, так и с открытым исходным кодом (см. http://www.opensource.org). Одна из самых замечательных особенностей того мира, в который мы собираемся отправиться, – его вечно изменяющаяся и вечно совершенствующаяся природа. Задачи, к которым 10 лет назад нельзя было подступиться из-за ограниченности ресурсов, теперь благополучно решаются благодаря появлению улучшенных алгоритмов, более быстрых процессоров, бо-

лее дешевой памяти и дисков, а также программ, которые позволяют без труда превратить многочисленные компьютеры в один виртуальный процессор. Сейчас особенно велико количество инструментов с открытым исходным кодом, которые могут послужить фундаментом для новых идей и новых приложений.

Цель настоящей книги – подвергнуть эти инструменты испытанию реальным миром и познакомить вас с проблематикой обработки естественных языков и информационного поиска. Мы не в состоянии охватить все аспекты того и другого и не собираемся обсуждать исследования, ведущиеся на переднем крае науки, ну разве что в самом конце книги. Мы сосредоточимся на тех областях, которые с высокой вероятностью окажутся наиболее важны в вашей практике приручения текста.

Ограничившись такими темами, как поиск, *распознавание сущностей* (нахождение имен людей, географических названий и т. п.), группировка, разметка, кластеризация и автореферирование, мы сможем построить практические приложения, которые помогают человеку легко и быстро находить и понимать смысл важных частей текста.

Мы не хотим ломать кайф от восторга, испытываемого при виде прирученного текста, но все же должны отметить, что в работе с текстом нет идеальных решений. Очень часто два человека, которые видят один и тот же результат, имеют разные мнения о его правильности, и совершенно неочевидно, что нужно поправить, чтобы удовлетворить обоих. Скажем больше – устранение одной проблемы может повлечь за собой появление новых. Как обычно, для достижения высококачественных результатов необходимы тестирование и анализ. Вообще, самые лучшие системы получаются, когда в контур управления включается человек; тогда они могут обучаться, учитывая реакцию пользователей, как умные люди учатся на собственных и на чужих ошибках. Но обратная связь с пользователями не обязана быть явной. Сохранение последовательности переходов по ссылкам, анализ журналов и других видов поведения пользователей может дать ценные сведения о том, как пользователи работают с приложением. Ниже мы приводим несколько общих рекомендаций о том, как улучшить приложение и сохранить душевное равновесие.

- Надо знать своих пользователей. Интересны ли им определенные структуры, например таблицы или списки, или достаточно просто собрать все слова, встречающиеся в документе? Готовы ли они предоставить вам больше информации, чтобы получить более качественные результаты, или на первом месте

стоит простота? Готовы ли они подольше подождать более качественные результаты или хотят видеть наилучшую гипотезу немедленно?

- Надо знать свое содержимое. Какие используются форматы файлов (HTML, Microsoft Word, PDF, простой текст)? Какие структуры и особенности важны? Встречаются ли в тексте жаргонизмы, аббревиатуры или различные способы выразить одну и ту же мысль? Относится ли текст преимущественно к одной области или охватывает несколько тем?
- Тестируйте, тестируйте и еще раз тестируйте. Потратьте какое-то время (но не слишком много) на измерение качества результатов и затрат на их получение. Практикуйтесь в искусстве компромисса. В любом нетривиальном приложении для обработки текста необходимо искать компромиссы в части качества результатов и масштабирования. Знания о пользователях и содержимом часто помогают найти оптимальное сочетание качества и производительности, удовлетворяющее большинство людей.
- Иногда достаточно и приближения к точному результату. Постарайтесь сообщить пользователю уровень достоверности ответа, чтобы он мог принять обоснованное решение.
- При прочих равных условиях отдавайте предпочтение более простому подходу. Кстати, вы удивитесь, насколько хорошие результаты могут давать простые решения.

И еще – хотя работать с иностранными языками интересно, в этой книге мы придерживаемся английского. Но уверяем вас, что многие описанные решения применимы и к другим языкам при наличии соответствующих ресурсов.

Следует также отметить, что трудность стоящих перед вами задач может варьироваться в широких пределах: от относительно простых до трудных настолько, что вместо поиска решения вполне можно подбрасывать монетку. Например, в английском и других европейских языках алгоритмы разбиения на лексемы и частеречной разметки работают неплохо, тогда как инструменты машинного перевода, анализа тональности и выведения смысла текста гораздо труднее и зачастую без наложения тех или иных ограничений дают неудовлетворительные результаты.

Наконец, обработка текста – все равно что катание на американских горках. Существуют вершины, где приложение просто не может ошибиться, и впадины, где оно никогда не дает правильных результа-

тов. Нужно смириться с тем фактом, что ни один из подходов, обсуждаемых в этой книге, равно как и в области NLP вообще, не является окончательным решением задачи. Так что вы можете копнуть глубже и вписать свое имя на скрижали. Итак, давайте приступим и заложим фундамент для некоторых идей, а в последующих главах вернемся и определим контекст, который позволит нам перейти от поиска в чудесный мир обработки естественных языков.

1.5. Текст и интеллектуальные приложения: поиск и не только

Уже много лет, как поиск правит миром. Без Google, Yahoo! и им подобных Интернет без сомнения даже отдаленно не был бы таким, каким мы его знаем. И при этом появление хороших инструментов поиска с открытым исходным кодом, в частности Apache Solr и Apache Lucene, наряду с мириадами роботов-индексаторов и методов распределенной обработки превратило поиск в предмет широкого потребления, по крайней мере, на персональных компьютерах и в корпоративных сетях, где не нужны гигантские центры обработки данных. В то же время люди ждут все большего от поисковых систем. Мы хотим получать более качественные результаты за меньшее время, введя всего одно-два ключевых слова. Еще мы хотим, чтобы по имеющимся у нас данным можно было легко искать и чтобы их можно было организовывать.

Далее, в условиях конкурентного давления корпорации вынуждены постоянно совершенствовать свои продукты. Всякий раз как крупный игрок типа Google или Amazon делает очередной шаг, улучшающий доступ к информации, планка поднимается для всех. Пять, десять, пятнадцать лет назад достаточно было добавить возможность поиска данных; теперь наличие поиска само собой подразумевается, а определяющие правила игры компании применяют сложные алгоритмы, в которых используется машинное обучение и глубокий статистический анализ для работы с такими объемами данных, на понимание которых у человека ушли бы годы. Такова эволюция интеллектуальных приложений. Все больше компаний осваивают машинное обучение и углубленный анализ текста в хорошо изученных областях, чтобы наделить свои приложения дополнительным «интеллектом».

Внедрение машинного обучения и методов NLP обусловлено тем, что практические приложения имеют дело с огромными объемами данных, а не напыщенными, хотя и не лишенными смысла, возгласа-

ми о машинах, которые «понимают» человека или каким-то образом проходят тест Тьюринга (см. http://ru.wikipedia.org/wiki/Тест_Тьюринга). Компании акцентируют внимание на следующих вопросах: нахождение и извлечение из текста важных признаков; агрегирование информации, в том числе пользовательских переходов по ссылкам, оценок и отзывов; группировка и автореферирование сходного содержимого и, наконец, отображение всего вышеозначенного так, чтобы пользователь мог более эффективно производить поиск по содержимому, что в конечном итоге увеличит объемы продаж или будет способствовать достижению какой-то иной цели. Ведь не можете же вы купить то, что не нашли, правда?

Все это прекрасно, но с чего же начать? Начнем с основ поиска (глава 3), а затем рассмотрим способы автоматической организации содержимого с применением концепций, знакомых по повседневной деятельности. Только делать это мы будем не вручную, а поручим машине (немного помогая ей по мере необходимости). Памятуя о поставленной цели, мы в следующих разделах выделим три класса идей, касающихся поиска и организации содержимого, и приведем пример, который свяжет различные концепции воедино. Этот пример будет подробно исследоваться на протяжении всей книги.

1.5.1. Поиск и сопоставление

Поиск – это отправная точка для большинства наших усилий по приручению текста, включая и предложенную вопросно-ответную систему, которая нуждается как в индексировании входных данных, так и в отыскании потенциальных фрагментов, отвечающих на вопрос пользователя. Даже в тех случаях, когда приходится применять методы, выходящие за рамки поиска, все равно сначала, как правило, нужно найти подмножество текста или набора документов, к которому эти методы применяются.

В главе 3 «Поиск» мы обсудим, как индексировать документы и обеспечить возможность поиска по ним, а также как выбрать документы, отвечающие запросу. Мы также изучим, каким образом документы ранжируются поисковой системой, и воспользуемся этой информацией для повышения качества возвращаемых результатов. Наконец, мы рассмотрим фасетный поиск, который позволяет уточнить результаты, ограничив их предопределенной категорией. Эти вопросы будут проиллюстрированы примерами на основе Apache Solr и Apache Lucene.

Познакомившись с методами поиска, вы быстро придете к выводу, что поиск хорош лишь настолько, насколько хороши данные, по которым он производится. Если слова и словосочетания, которые ищут пользователи, отсутствуют в индексе, то не стоит ожидать релевантных результатов. В главе 4 «Неточное совпадение со строками» мы рассмотрим приемы, позволяющие рекомендовать запросы, исходя из доступного содержимого, за счет проверки орфографии запроса. Мы также увидим, как аналогичную технику можно применить к задачам связывания записей в базе данных, выйдя тем самым за пределы простого соединения таблиц. Эти методы часто используются не только в составе поиска, но и для более сложных вещей, в частности определения того, принадлежат ли два профиля одному и тому же пользователю; такая задача возникает, когда две компании сливаются, и требуется объединить списки их клиентов.

1.5.2. Извлечение информации

Поиск помогает находить документы, содержащие нужную вам информацию, но часто требуется выделить более мелкие информационные единицы. Например, умение находить имена собственные в большом массиве текста может оказаться весьма полезным для прослеживания преступной деятельности или отыскания связей между людьми, которые, возможно, никогда не встречались. Для этого нам понадобится изучить методы выделения и классификации небольших участков текста, обычно длиной всего несколько слов.

В главе 3 «Основы приручения текста» мы познакомимся с методами распознавания слов, образующих лингвистическую единицу, например словосочетание, которые можно использовать для идентификации тех слов в документе или запросе, которые имеет смысл сгруппировать. В главе 5 «Распознавание имен людей, географических названий и других сущностей» мы узнаем, как выделять имена собственные и словосочетания, содержащие числа, и относить их к той или иной семантической категории – человек, место или дата – не зависящей от лингвистического использования. Эта возможность окажется принципиальной при построении вопросно-ответной системы в главе 8. Для решения обеих задач мы будем пользоваться средствами пакета OpenNLP и познакомимся с тем, как работать со встроенными в него моделями и как создавать новые модели, точнее отвечающие данным. В отличие от задач поиска и сопоставления, для построения этих моделей сначала содержимое аннотируется вруч-

ную, а затем к нему применяются методы статистического машинного обучения.

1.5.3. Группировка информации

Дополнением к извлечению информации из текста служит обогащение текста новой информацией путем группировки данных или добавления меток. Представьте, к примеру, насколько проще было бы работать с электронной почтой, если бы она была автоматически размечена и рассортирована по важности, чтобы вы могли без труда находить все похожие сообщения. Тогда можно было уделить первостепенное внимание письмам, требующим немедленной реакции, а также находить вспомогательные материалы для отправляемых писем.

Широко распространенный подход к решению этой задачи – группировка текста по категориям. Оказывается, что методы, применяемые для извлечения информации, можно применить и к группировке текста или документов. Такие группы часто можно использовать как фасеты в поисковом индексе, как дополнительные ключевые слова или как альтернативный способ информационной навигации, доступный пользователю. Даже в тех случаях, когда пользователи явно указывают категории в виде меток, применение вышеупомянутых методов позволяет рекомендовать метки, использованные в прошлом. В главе 7 «Классификация, категоризация и пометка» мы покажем, как строятся модели для классификации документов и как эти модели применяются к новым документам для улучшения результатов обработки текста.

После того как вы нашли то, что искали, и извлекли нужную информацию, может выясниться, что хорошенького оказалось слишком много. В главе 6 «Кластеризация» мы рассмотрим, как группировать схожую информацию. Ту же технику можно использовать для выявления и – при необходимости – устранения избыточной информации. Она же годится для группировки схожих документов, чтобы пользователь мог видеть сразу все материалы по теме и их релевантность без необходимости прочитывать каждый документ.

1.5.4. Интеллектуальное приложение

В предпоследней главе «Пример вопросно-ответной системы» мы соберем вместе подходы, описанные в предшествующих главах, и построим интеллектуальное приложение. Точнее, наша фактографи-

ческая система сможет искать в тексте ответы на вопросы о любопытных фактах. Например, при наличии подходящего текста она сможет ответить на вопрос «Кто является президентом США?». В системе будут применяться методы, описанные в главе 3 «Поиск», для выделения текста, который может содержать ответ на вопрос. Для нахождения фрагментов текста, которые часто являются ответами на вопросы о фактах, мы будем применять методы из главы 5 «Распознавание имен людей, географических названий и других сущностей». Материал из главы 2 «Основы приручения текста» и главы 7 «Классификация, категоризация и пометка» будут использованы для анализа заданного вопроса и определения типа искомой информации. Наконец, для ранжирования ответов мы применим методы, описанные в главе 3.

1.6. Резюме

Обработка текста – это большая, иногда ошеломляюще большая, задача, которая дополнительно осложняется наличием разных языков, диалектов и интерпретаций. В качестве текста может выступать изящная проза, написанная великим писателем, или никуда не годное сочинение, лишенное стиля и предмета. Но какова бы ни была форма, тексты присутствуют повсеместно, и ни люди, ни программы не могут от них отмахнуться. По счастью, существует достаточно инструментов – коммерческих и с открытым исходным кодом, которые помогают извлечь из текста смысл. Пока они не совершенны, но становятся все лучше и лучше. Мы поговорили о том, почему обработка текста так важна и в то же время трудна. Мы также обсудили роль текста в интеллектуальном Интернете, рассказали, какие темы собираемся рассмотреть и вкратце остановились на том, что нужно для построения простой вопросно-ответной системы. В следующей главе мы займемся реальным делом: заложим основы анализа текста и остановимся на вопросе извлечения собственно текста из файлов в различных используемых сегодня форматах.

1.7. Ресурсы

"Americans Spend Half of Their Spare Time Online." 2007. Media-Screen LLC. http://www.media-screen.com/press050707.html.

Feldman, Susan. 2009. "Hidden Costs of Information Work: A Progress Report." International Data Corporation.

Gantz, John F. and Reinsel, David. 2011. "Extracting Value from Chaos." International Data Corporation. http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

Liddy, Elizabeth. 2001. "Natural Language Processing." *Encyclopedia of Library and Information Science, 2nd Ed.* NY. Marcel Decker, Inc.

"Trends in Consumers' Time Spent with Media." 2010. eMarketer. http://www.emarketer.com/Article.aspx?R=1008138.

ГЛАВА 2. Основы приручения текста

В этой главе:

- Элементы обработки текста: лексический анализ, разбиение на блоки, грамматический разбор и частеречная разметка.
- Выделение текста из файлов стандартных форматов с помощью проекта с открытым исходным кодом Apache Tika.

Прежде чем приступать к тяжелым процессам приручения текста, нужно слегка разогреться. Начнем с того, что освежим в памяти школьный курс английского языка, вспомнив о лексическом анализе, стемминге, частях речи, словосочетаниях и сложных предложениях. Все эти вопросы важны с точки зрения качества результатов, и это станет ясно, когда мы начнем создавать приложения. Например, такое простое на первый взгляд действие, как разбиение на слова, в китайском языке оказывается весьма трудным. Даже в английском лексический анализ усложняется наличием знаков препинания. Аналогично выделение частей речи и словосочетаний может представлять сложности из-за присущей языку неоднозначности. Мы продолжим обсуждение основ лингвистики, рассмотрев вопрос об извлечении текста из файлов различных форматов, встречающихся на практике. Во многих книгах и статьях вопросом об извлечении содержимого пренебрегают, считая, что у пользователя есть текст, готовый к обработке, но мы считаем важным исследовать возникающие здесь проблемы по нескольким причинам.

- Зачастую извлечь текст из файла в недокументированном формате затруднительно. Даже коммерческие программы нередко ошибаются при извлечении содержимого.
- На практике вы потратите заметное время, изучая различные форматы файлов и средства извлечения текста и пытаясь разобраться, что правильно. Реальные данные редко предстают в виде наборов простых строк. Они странно отформатированы, в них временами встречаются символы, которых не должно быть. Есть и другие проблемы, заставляющие рвать на себе волосы.
- Результаты обработки не могут быть лучше исходных данных. Старая поговорка «мусор на входе, мусор на выходе» в данном случае справедлива, как и во всех остальных. В конце этой главы, после того как мы вспомним правила английского языка и извлечем содержимое, будут рассмотрены некоторые основополагающие элементы, упрощающие написание приложений и библиотек. А теперь без долгих предисловий посмотрим, как выделять слова и составлять из них разные полезные конструкции: предложения, именные и другие группы и, возможно, полные деревья разбора.

2.1. Основы лингвистики

Не тоскуете по школьным денькам? По урокам английского языка, разбору предложений, определению подлежащего и сказуемого, обособленных причастных оборотах? Ну что ж, вам повезло, поскольку для анализа текста придется вспомнить основы английского, изучавшиеся в средней школе, и даже больше. Но шутки в сторону, в следующих разделах мы заложим фундамент, на котором будут возводиться наши приложения, для чего рассмотрим типичные проблемы, которые необходимо решать при анализе текста. Это даст нам возможность составить единый словарь, наличие которого упростит дальнейшие объяснения, и заставит думать о возможностях и функции языка и о том, как учитывать их в приложении. Например, при построении вопросно-ответной системы в главе 8 нам нужно будет разбивать строки на слова, а затем определять, какую роль каждое слово играет в предложении (какой частью речи оно является), и устанавливать связи между словами посредством таких конструкций, как именные группы и сложные предложения. Имея эту информацию, мы сможем разобрать вопрос типа «Кто является дядей Боба?» и понять, что от-

ветом должно быть имя собственное (которое состоит из слов, помеченных как существительные) и что оно должно встречаться в том же предложении, в каком встречаются слова *дядя* и *Боб* (и предпочтительно именно в таком порядке)[1]. И хотя нам это кажется само собой разумеющимся, компьютеру необходимо сказать, чтобы он принимал во внимание эти атрибуты. Есть приложения, в которых требуются все указанные элементы, но много и таких, которым достаточно одного или двух. В некоторых приложениях их использование декларируется явно, в других – нет. В конечном итоге, чем больше вы знаете об устройстве языка, тем лучше подготовлены к анализу компромиссов, неизбежных в любой системе анализа текста.

В первом разделе мы опишем различные категории слов и словосочетаний и посмотрим, как из слов составляются предложения. В этом кратком введении в *синтаксис* мы уделим основное внимание темам, которые встретятся далее в книге. Во втором разделе мы займемся самими словами, эта область лингвистики называется *морфологией*. Хотя в этой книге морфология не будет рассматриваться явно, краткое введение поможет понять, какие здесь имеются подходы. И еще – хотя понятия синтаксиса и морфологии применимы ко всем разговорным, или естественным языкам, в наших примерах мы ограничимся только английским.

2.1.1. Категории слов

Существует несколько лексических категорий слов, они называются частями речи. К ним относятся существительные, глаголы, прилагательные, определяющие слова, предлоги и союзы. Хотя эти категории наверняка встречались вам в разных контекстах, возможно, вы не видели их все сразу и не помните, что они означают. Краткое знакомство с этими понятиями окажется полезно в последующих главах, когда мы будем изучать методы, в которых категории используются непосредственно или, по крайней мере, учитываются. В табл. 2.1 перечислены основные лексические категории с определениями и примерами. А далее мы сообщим дополнительные сведения об этих категориях верхнего уровня.

---

[1] Все изложение в книге ориентировано на английский язык, поэтому в переводе оставлены ссылки на английскую грамматику. Тем не менее, такие вещи, как вопрос «Кто является дядей Боба?», для удобства читателя переведены. Однако не следует забывать, что перевод меняет структуру предложения. Так в оригинале слово *Боб* предшествует слову *дядя* (uncle), поэтому и в англоязычном предложении они должны встречаться в порядке *Bob*, затем *uncle*. – *Прим. перев.*

Таблица 2.1. Определения и примеры самых известных лексических категорий[2]

Лексическая категория	Определение	Пример (курсивом)
Имя прилагательное	Слово или словосочетание, обозначающее характеристику предмета, добавляется или грамматически связано с существительным, которое модифицирует или описывает.	The *quick red* fox jumped over the *lazy brown* dogs.
Наречие	Слово или словосочетание, которое модифицирует или дополняет прилагательное, глагол, другое наречие или группу слов и обозначает связь с местом, временем, обстоятельствами, образом действия, причиной, степенью и т. д.	The dogs *lazily* ran down the field after the fox.
Союз	Слово, соединяющее два слова, словосочетания или части сложного предложения.	The quick red fox *and* the silver coyote jumped over the lazy brown dogs.
Определяющее слово	Слово-модификатор, которое определяет характер существительного или именной группы, например: *a*, *the*, *every*.	*The* quick red fox jumped over *the* lazy brown dogs.
Имя существительное	Слово, применяемое для обозначения класса людей, мест или предметов, а также для именования конкретного представителя всего вышеперечисленного.	The quick red *fox* jumped over the lazy brown *dogs*.
Предлог	Слово, которое управляет и обычно предшествует существительному или местоимению и выражает связь с другим словом или частью сложного предложения.	The quick red fox jumped *over* the lazy brown dogs.
Глагол	Слово, обозначающее действие, состояние или происшествие и образующее главную часть сказуемого предложения, например: *hear (слышать)*, *become (становиться)* и *happen (происходить)*.	The quick red fox *jumped* over the lazy brown dogs.

Выделение этих лексических категорий основано на синтаксическом использовании, а не на смысловом значении, но поскольку

[2] Определения взяты из словаря New Oxford American Dictionary, 2-е издание.

некоторые семантические понятия обычно выражаются в виде конкретной синтаксической конструкции, то часто в основу определения категорий кладут семантические ассоциации. Например, имя существительное нередко определяют как человека, место или предмет, а глагол – как действие, но ведь есть же существительное *разрушение*, а глагол *be* (быть) употребляется в конструкциях типа «Judy is 12 years old» (Джуди 12 лет). Ни то, ни другое словоупотребление не соответствуют типичным семантическим связям, ассоциирующимся с существительными и глаголами.

У этих категорий верхнего уровня есть более узкие подкатегории, и некоторые из них мы будем использовать в этой книге. Так, говоря об именах существительных, выделяют нарицательные, собственные и местоимения. Нарицательные существительные описывают классы объектов, например *город*, *океан* или *человек*, а имена собственные – конкретные объекты, обычно они начинаются с заглавной буквы, например: *Лондон*, *Джон*, *Эйфелева башня*. Местоимения – это существительные, ссылающиеся на другие объекты, обычно упоминавшиеся ранее. Примерами местоимений служат слова *он*, *она*, *оно*. Подкатегории имеются и у многих других лексических категорий, есть даже дополнительные подкатегории имен существительных, но уже сказанного достаточно для целей этой книги. Дополнительные сведения можно найти в любом хорошем справочнике по грамматике и языку, которых хватает в Интернете и особенно в википедии. Можно также почитать печатные учебники, например «The Chicago Manual of Style» или послушать подкасты, например, на сайте Grammar Girl (http://grammar.quickanddirtytips.com/).

2.1.2. Словосочетания и части предложения

Большинству лексических категорий слов, перечисленных в предыдущем разделе, соответствуют конструкции, которые могут состоять из нескольких слов, – словосочетания. В словосочетании выделяется по меньшей мере одно корневое слово конкретного типа, но могут присутствовать также слова и словосочетания других типов. Например, именная группа *the happy girl* состоит из определяющего слова (the), прилагательного (happy), а его корнем является нарицательное существительное *girl*. В табл. 2.2 приведены примеры словосочетаний.

Таблица 2.2. Примеры распространенных синтаксических групп (фразовых категорий)

Тип словосочетания	Пример (*курсивом*)	Примечание
Адъективная группа	The *unusually red* fox jumped over the *exceptionally lazy* dogs	Наречия *unusually* (необычно) и *exceptionally* (исключительно) модифицируют прилагательные *red* (рыжая) и *lazy* (ленивая), образуя адъективные группы
Наречная группа	The dogs *almost always* ran down the field after the fox	Наречие *almost* (почти) модифицирует наречие *always* (всегда), образуя наречную группу
Союзная группа	The quick red fox *as well as* the silver coyote jumped over the lazy brown dogs	Это довольно редкий случай, но, как видите, словосочетание *as well as* (так же как) выполняет ту же функцию, что союз *and* (и)
Именная группа	*The quick red fox* jumped over *the lazy brown dogs*	Имя существительное *fox* (лиса) и его модификаторы *the* (артикль), *quick* (быстрая) и *red* (рыжая) образуют именную группу, равно как существительное *dogs* (собаки) и его модификаторы *the*, *lazy* и *brown* (коричневые)
Предложные	The quick red fox jumped *over the lazy brown dogs*	Предлог *over* (через) и именная группа *the lazy brown dogs* образуют предложную группу, которая модифицирует глагол *jumped* (перепрыгнула)
Глагольная группа	The quick red fox *jumped over the lazy brown dogs*	Глагол *jumped* и модифицирующая его предложная группа *over the lazy brown dogs* образуют глагольную группу

Словосочетания могут комбинироваться и образовывать части предложения – минимальные единицы, из которых строится предложение. Часть предложения содержит как минимум именную группу (подлежащее) и глагольную группу (сказуемое), которая часто состоит из глагола и еще одной именной группы. Фраза *The fox jumped the fence* представляет собой часть предложения, состоящую из именной группы *The fox* (*лиса*, подлежащее) и глагольной группы *jumped the fence* (*перепрыгнула через забор*), которая, в свою очередь, состоит из именной группы *fence* (дополнение) и глагола *jumped*. В предложении могут присутствовать и группы других типов для выражения

иных связей. Таким образом, мы видим, что любое предложение можно разложить на множество частей, части – на множество групп, а группы – на слова, являющиеся теми или иными частями речи. Задача определения частей речи, групп, частей предложения и их взаимосвязей называется *грамматическим анализом*, или *разбором*. Вполне возможно, вы проводили такой анализ, если когда-нибудь строили диаграмму разбора предложения. Ниже в этой главе мы рассмотрим программы, решающие эту задачу.

2.1.3. Морфология

Морфология изучает внутреннюю структуру слова. В большинстве языков слово состоит из *лексемы*, или начальной формы и различных аффиксов (префиксов и суффиксов), которые видоизменяют слово в зависимости от его употребления. В английском языке применяются преимущественно суффиксы, а правила их использования зависят от лексической категории слова.

Имена существительные – нарицательные и собственные – в английском изменяются по числу, существуют две разные формы: единственное и множественное число. В единственном числе существительное представлено в начальной форме, а во множественном к нему обычно добавляется суффикс *s*. Хотя у нарицательных и собственных имен существительных есть всего две формы, местоимения изменяются по числу, лицу, падежу и роду. Впрочем, местоимения – это замкнутый класс слов, в котором есть всего 34 различных формы, поэтому обычно проще их перечислить, чем создавать модель морфологической структуры. Существительные, производные от других лексических категорий, также содержат суффиксы, соответствующие

Таблица 2.3. Примеры морфологии отглагольных существительных

Суффикс	Пример	Глагол
-ation	nomination	Nominate
-ee	appointee	appoint
-ure	closure	close
-al	refusal	Refuse
-er	runner	Run
-ment	advertisement	advertise

Таблица 2.4. Примеры морфологии отадъективных существительных

Суффикс	Пример	Прилагательное
-dom	freedom	free
-hood	likelihood	likely
-ist	realist	real
-th	warmth	Warm
-ness	happiness	Happy

преобразованию. Так, у существительных, производных от глаголов или прилагательных, могут быть суффиксы, перечисленные в табл. 2.3 и 2.4.

У глаголов более сложная морфология; всего возможно восемь форм грамматического изменения, но у правильных глаголов их только четыре. Некоторые неправильные глаголы употребляются с суффиксом *en* при использовании в качестве причастия прошедшего времени вместо обычного для этой формы окончания *ed*. Эти формы показаны в табл. 2.5. Остальные три формы лексически выделяются только ради немногих неправильных глаголов, у них нет общего суффикса.

Таблица 2.5. Примеры морфологии правильного глагола и типичные окончания неправильных глаголов в форме причастия прошедшего времени

Суффикс	Пример	Производная форма
нет	look	Начальная форма
-ing	looking	Герундиальная форма
-s	looks	Форма третьего лица единственного числа
-ed	looked	Форма прошедшего времени
-en	taken	Форма причастия прошедшего времени

Производные формы прилагательных и наречий употребляются для обозначения сравнения, существует сравнительная и превосходная степень. К прилагательному *tall* (высокий) в сравнительной степени добавляется суффикс *-er* (*taller*), а в превосходной – суффикс *-est* (*tallest*). Аналогично наречие *near* (близко) в сравнительной степени приобретает суффикс *-er* (*nearer*), а в превосходной – суффикс *-est* (*nearest*).

Изложенных базовых сведений о связях между словами и о структуре самих слов достаточно, чтобы приступить к работе с программным обеспечением, в котором все это используется для приручения текста.

2.2. Популярные инструменты для обработки текста

Разобравшись с основами синтаксиса и семантики языка, давайте познакомимся с некоторыми инструментами, которые помогают распознавать эти и другие важные элементы в цифровом тексте. Одни из них используются постоянно, другие – от случая к случаю. Мы начнем с манипуляций со строками, а затем перейдем к более сложным вещам, в частности, к полному разбору предложения. Вообще говоря,

базовые вещи находят применение каждодневно, а такие сложные, как полные анализаторы языка, – только в специфических приложениях.

2.2.1. Инструменты для манипуляций со строками

Библиотеки для работы со строками, массивами символов и другими представлениями текста лежат в основе большинства программ обработки текста. Для большинства языков программирования имеются библиотеки для таких базовых операций, как конкатенация строк, разбиение строки, поиск подстроки и различные методы сравнения двух строк. Дополнительные возможности дает библиотека регулярных выражений, например `java.util.regex` в случае Java (подробное изложение регулярных выражений имеется в книге Джеффри Фридла «Регулярные выражения»). Также неплохо свободно владеть классами `String`, `Character`, `StringBuilder` и пакетом `java.text`. С помощью этих средств легко проводить поверхностный анализ текста, например, узнать, начинается ли слово с заглавной буквы, вычислить длину слова, определить, содержит ли оно цифры или небуквенные символы. Кроме того, полезно знать, как разбирать даты и числа. В главе 4 мы более подробно рассмотрим алгоритмы для работы со строками. А пока остановимся на лингвистически значимых атрибутах текста.

2.2.2. Лексемы и лексический анализ

Почти всегда за извлечением содержимого из файла следует его разбиение на мелкие, пригодные для работы блоки текста, называемые *лексемами*. Часто лексема соответствует одному слову, но, как мы вскоре увидим, что именно понимать под «мелким, пригодным для работы блоком», зависит от приложения. Если говорить об английском языке, то наиболее распространенный подход к лексическому анализу – разбиение строки в местах вхождения пробелов, перехода на другую строку и других символов-разделителей. Вот пример простейшего лексического анализатора: `String[] result = input.split("\\s+");`. Здесь входная строка (объект типа `String`) разбивается на массив строк с помощью регулярного выражения `\s+` (обратите внимание, что в Java символ обратной косой черты необходимо экранировать), то есть по символам-разделителям. В большинстве случае этот подход годится, но применение его к предложению

```
I can't believe that the Carolina Hurricanes won the
2005-2006 Stanley Cup.

(Не могу поверить, что Каролина Харрикейнз выиграли
кубок Стэнли 2005-2006.)
```

порождает лексемы, показанные в табл. 2.6. Вы будете правы, считая, что точка в конце слова *Cup* не вполне уместна.

Таблица 2.6. Результат разбиения предложения по пробелам

I	can't	believe	that	Carolina	Hurricanes	won	the	2005-2006	Stanley	Cup.

Хотя в некоторых случаях разбиения по пробелам достаточно, в большинстве приложений необходимо правильно обрабатывать знаки препинания, акронимы, адреса электронной почты, URL-адреса и числа – для повышения качества результатов. Кроме того, у разных приложений часто бывают разные требования к лексическому анализу. Например, класс `StandardTokenizer` из поисковой библиотеки Apache Solr/Lucene (рассматривается в главе 3) учитывает такие типичные элементы, как знаки препинания и акронимы. Если пропустить показанное выше предложение через `StandardTokenizer`, то завершающая точка будет опущена; получающиеся лексемы показаны в табл. 2.7.

Таблица 2.7. Результат разбиения предложения с помощью класса Solr `StandardTokenizer`

I	can't	believe	that	Carolina	Hurricanes	won	the	2005	2006	Stanley	Cup

Быть может, вы подумали: «Но ведь я не хотел совсем избавляться от точки, нужно было только не присоединять ее к *Cup*»? Разумное возражение, которое лишь подчеркивает, насколько важно обдумывать, что хочет конкретное приложение от лексического анализа. Поскольку поисковым приложениям типа Lucene точка в конце предложения обычно неинтересна, в данном случае вполне можно не включать ее в состав лексем.

В других приложениях может понадобиться другой лексический анализатор. Обработка того же предложения с помощью класса `english.Tokenizer` из проекта OpenNLP дает результаты, показанные в табл. 2.8.

Таблица 2.8. Результат разбиения предложения с помощью класса OpenNLP `english.Tokenizer`

I	ca	n't	believe	that	Carolina	Hurricanes	won	the	2005-2006	Stanley	Cup	.

Обратите внимание, что пунктуация сохранена, а сокращение *can't* разбито на две части. В случае OpenNLP лексический анализ – шаг, предшествующий грамматическому анализу. В такого рода приложениях знаки препинания помогают определить границы частей предложения, а у слов *can* и *not* разные грамматические роли.

Если взглянуть на другой вид обработки в пакете OpenNLP – для распознавания именованных сущностей (см. главу 5), то мы увидим еще один тип лексического анализа. Здесь строки разбиваются по типам лексем: буквенные, числовые, пробелы и прочие. Для того же самого предложения класс `SimpleTokenizer` дает результат, показанный в табл. 2.9.

Таблица 2.9. Результат разбиения предложения с помощью класса OpenNLP `SimpleTokenizer`

I	can	'	t	Believe	that	Carolina	Hurricanes	won	the	2005	-	2006	Stanley	Cup	.

Как видим, здесь диапазон дат разбит на две части. Это позволяет компоненту распознавания сущностей рассматривать начальную и конечную дату по отдельности – как того и требует данное приложение.

Ранее мы видели, что на решения лексического анализатора влияет то, какая операция выполняется. Подходящая текстовая единица зависит от видов обработки, которой подвергается текст. К счастью, средства лексического анализа, предоставляемые большинством библиотек, позволяют сохранять все необходимое для последующих шагов обработки. Или же пользователю дается возможность написать собственный анализатор.

Перечислим другие распространенные методы, применяемые на уровне лексем.

- *Изменение регистра* – перевод в нижний регистр всех лексем, которые могут быть полезны при поиске.
- *Удаление стоп-слов* – фильтрация часто встречающихся слов, например: *the*, *and*, *a*. Ценность таких слов для приложений, не анализирующих структуру предложения, обычно невелика (заметьте – мы не говорим, что они не представляют *никакой* ценности).
- *Расширение* – добавление в поток лексем синонимов или расшифровка акронимов и аббревиатур; это дает приложению возможность рассматривать альтернативные формы полученного от пользователя вопроса.

- *Частеречная разметка* – сохранение вместе с лексемой информации о том, какой частью речи она является. Подробнее рассматривается в следующем разделе.
- *Стемминг* – приведение слова к начальной форме, например, переход от *dogs* к *dog*. Подробнее рассматривается в разделе 2.2.4.

Мы не будем останавливаться на простых случаях – удалении стоп-слов, расширении запроса и изменении регистра, поскольку эти операции обычно сводятся к просмотру словаря `Map` или вызову методов объекта `String`. В следующих двух разделах мы более подробно поговорим о частеречной разметке и стемминге, а уже потом перейдем на уровень предложения.

2.2.3. Частеречная разметка

Знание того, какой частью речи является слово – существительным, глаголом или прилагательным – обычно способствует повышению качества результатов последующей обработки. Например, наличие этой информации помогает выявить существенные ключевые слова, встречающиеся в документе (см., например, Mihalcea [2004]), или искать конкретные варианты словоупотребления (например, *Will* – это имя собственное, а *will* – модальный глагол, как во фразе «you will regret that»). Существует много готовых и допускающих обучение частеречных разметчиков с открытым исходным кодом. Один такой разметчик из проекта OpenNLP, основанный на понятии максимальной энтропии (Maximum Entropy Tagger), имеется на сайте http://opennlp.apache.org/. Не расстраивайтесь, если не знаете, что такое *максимальная энтропия*; речь идет просто об использовании статистических данных для определения того, какой частью речи данное слово является с максимальной вероятностью. В англоязычном частеречном разметчике из проекта OpenNLP для разметки слов применяются метки, определенные в проекте Penn Treebank Project (http://www.cis.upenn.edu/~treebank). Наиболее употребительные из них приведены в табл. 2.10. Для многих меток имеются связанные метки, которые служат для пометки многочисленных форм слова, например в настоящем и прошедшем времени или в единственном и множественном числе. Полный список смотрите на странице http://repository.upenn.edu/cgi/viewcontent.cgi?article=1603&context=cis_reports.

Таблица 2.10. Определения и примеры часто встречающихся частей речи

Часть речи	Метки из проекта Penn Treebank	Примеры
Прилагательное, прилагательное в превосходной степени, прилагательное в сравнительной степени	JJ, JJS, JJR	nice, nicest, nicer
Наречие, наречие в превосходной степени, наречие в сравнительной степени	RB, RBR, RBS	early, earliest, earlier
Определяющее слово	DT	a/the
Существительное, существительное во множественном числе, имя собственное, имя собственное во множественном числе	NN, NNS, NNP, NNPS	house, houses, London, Teamsters
Личное местоимение, притяжательное местоимение	PRP, PRP$	he/she/it/himself, his/her/its
Глагол в неопределенной форме, глагол в прошедшем времени, причастие прошедшего времени, глагол в третьем лице единственного числа, глагол в любом лице, кроме третьего единственного числа, герундий или причастие настоящего времени	VB, VBD, VBN, VBZ, VBP, VBG	be, was, been, is, am, being

Познакомившись с некоторыми частеречными метками, с которыми мы еще столкнемся далее, посмотрим, как работает разметчик OpenNLP. В нем используется статистическая модель, построенная по результатам обследования *корпуса*, или набора уже размеченных документов. В этой модели хранятся данные для вычисления вероятности того, что слово является той или иной частью речи. По счастью, вам создавать свою модель не нужно (хотя это можно было бы сделать), она уже готова. В листинге ниже показано, как загрузить модель и связанную с ней информацию и как запустить частеречный разметчик.

```
File posModelFile = new File(                          <-- Указать путь к модели частеречной разметки
    getModelDir(), "en-pos-maxent.bin");
FileInputStream posModelStream = new FileInputStream(posModelFile);
POSModel model = new POSModel(posModelStream);         Провести лексический анализ предложения и выделить слова

POSTaggerME tagger = new POSTaggerME(model);
String[] words = SimpleTokenizer.INSTANCE.tokenize(
```

```
    "The quick, red fox jumped over the lazy, brown dogs.");
String[] result = tagger.tag(words);
for (int i=0 ; i < words.length; i++) {
  System.err.print(words[i] + "/" + result[i] + " ");
}
System.err.println("n");
```

Передать разбитое на лексемы предложение для разметки

Эта программа печатает такой результат:

```
The/DT quick/JJ ,/, red/JJ fox/NN jumped/VBD over/IN the/DT
lazy/JJ ,/, brown/JJ dogs/NNS ./.
```

При беглом взгляде все выглядит разумно: *dogs* и *fox* – существительные; *quick*, *red*, *lazy* и *brown* – прилагательные, *jumped* – глагол. Пока что это все, что нам нужно знать о частеречной разметке, хотя мы еще не раз будем возвращаться к ней.

2.2.4. Стемминг

Представьте, что ваша работа состоит в прочитывании всех газет, выходящих в вашей стране (надеемся, вы запаслись кофе!), в поисках статей о банках. Вы раздобыли поисковую систему, загрузили в нее все газеты и начали искать слова bank, banks, banking, banker, banked и т. д. Но времени мало, и у вас мелькает мысль: «А хорошо бы, если бы я мог просто набрать слово *bank*, а система нашла бы все его разновидности». В этот момент вы поняли, в чем заключается мощь стемминга (и синонимии, но это отдельная тема). *Стеммингом* называется процесс приведения слова к более простой начальной форме, которая необязательно является сама по себе словом. Стемминг часто применяется в приложениях для обработки текста, в частности поисковых системах, потому что обычно пользователь ожидает, что в ответ на запрос *bank* будут найдены все документы, касающиеся банков. В большинстве случае пользователю не нужно точное совпадение с ключевыми словами, если только он не сообщил об этом явно.

Существует много подходов к стеммингу, у каждого свои цели. Некоторые агрессивно приводят слова к минимально возможной начальной форме, другие попроще и довольствуются такими простыми действиями, как удаление суффиксов *s* и *ing*. Например, при поиске почти всегда приходится искать классический компромисс между количеством и качеством. Агрессивный стемминг обычно дает больше результатов худшего качества, а упрощенный обеспечивает более высокое качество, рискуя пропустить некоторые полезные результаты. Стемминг может порождать проблемы, когда слова с разным

значением приводятся к одной начальной форме, теряя при этом свой смысл, и наоборот – схожие по смыслу слова не приводятся к одной форме (см. Krovetz [1993]).

Как выбирать стеммер и когда его следует использовать? Как и в большинстве приложений NLP, всё зависит от обстоятельств. Прогон тестов, эмпирические суждения, метод проб и ошибок – в конечном итоге только так можно получить практически оптимальные ответы на эти вопросы. Так что лучший совет, который мы можем дать, – программируйте на основе интерфейсов, чтобы один стеммер можно было легко подменить другим. А потом начните с простого стеммера и попытайтесь собрать отзывы тестировщиков или пользователей. Если пользователи считают, что в результатах поисках представлены не все вариации слова, то можно перейти к более агрессивному стеммеру.

Теперь, когда мы познакомились с некоторыми причинами использования стемминга, рассмотрим стеммеры Snowball (http://snowball.tartarus.org/), которые разработал д-р Мартин Портер (Martin Porter) со своими сотрудниками. Помимо либеральных условий лицензирования, стеммеры Snowball обладают тем преимуществом, что поддерживают различные подходы и языки, в том числе (перечень не полный):

- Porter и Porter2 (называемые также EnglishStemmer);
- Lovins;
- испанский;
- французский;
- русский;
- шведский.

В листинге ниже мы создаем английский стеммер (иногда называется Porter2), а затем в цикле перебираем массив лексем.

Листинг 2.2. Использование английского стеммера Snowball

```
EnglishStemmer english = new EnglishStemmer();  Готовим лексемы для стемминга
String[] test = {"bank", "banks", "banking", "banker", "banked",
                 "bankers"};
String[] gold = {"bank", "bank", "bank", "banker", "bank", "banker"};  Определяем ожидаемые результаты
for (int i = 0; i < test.length; i++) {
  english.setCurrent(test[i]);   Говорим английскому стеммеру, какую лексему обрабатывать
  english.stem();  <-- Выполняем стемминг
  System.out.println("English: " + english.getCurrent());
  assertTrue(english.getCurrent() + " is not equal to " + gold[i],
             english.getCurrent().equals(gold[i]) == true);
```

В этом автономном тесте мы ожидаем, что результатами стемминга будут слова "bank", "bank", "bank", "banker", "bank", "banker". Отметим, что, согласно правилам английского стеммера, *banker* (и *bankers*) не приводятся к *bank*. Это не ошибка, просто так работает алгоритм. Для анализа газет не очень удобно, но все же лучше, чем прогонять через поисковую систему все варианты слова *bank*. Без сомнения, рано или поздно тестировщик или пользователь пожалуются, что слово *X* не находится или его основа выделена неправильно. Если вы согласны, что указанный недочет следует исправить, то самое лучшее решение – завести список защищенных слов, которые не подвергаются стеммингу, а не пытаться подправить сам алгоритм (если только вы не контролируете код стеммера).

2.2.5. Распознавание предложений

Допустим, вам требуется найти все вхождения фразы *Super Bowl Champion Minnesota Vikings*[3] (Победитель суперкубка Миннесота вайкингс) в новостях, и встретился такой текст:

> *Last week the National Football League crowned a new Super Bowl Champion. Minnesota Vikings fans will take little solace in the fact that they lost to the eventual champion in the playoffs.*
>
> *(На прошлой неделе Национальная футбольная лига короновала нового победителя суперкубка. Слабым утешением для болельщиков Миннесота вайкингс остается тот факт, что их команда в плейофф проиграла чемпиону.)*

Лексический анализ этих предложений с помощью класса `StandardTokenizer`, использованного в разделе 2.2.2, дает следующие лексемы:

```
..."a", "new", "Super", "Bowl", "Champion", "Minnesota",
"Vikings", "fans", "will", ...
```

Если нас интересуют только такие вхождения лексем *Super*, *Bowl*, *Champion*, *Minnesota*, *Vikings*, в которых они находятся рядом, то это фразовое совпадение. Однако в данном случае такого совпадения нет из-за точки между словами *Champion* и *Minnesota*.

Определение границ предложения может уменьшить количество ошибочных фразовых совпадений, а также дает механизм для выявления структурных связей между словами и словосочетаниями, рав-

[3] Увы, увы... Но помечтать-то можно?

но как и между предложениями. Выявив такие связи, мы затем можем попытаться найти в тексте осмысленные фрагменты информации. В Java имеется готовый класс `BreakIterator`, который выделяет предложения, но часто необходимо дополнительное программирование для обработки особых случаев. Вот как выглядит бесхитростное использование `BreakIterator`:

```
BreakIterator sentIterator =
                BreakIterator.getSentenceInstance(Locale.US);
String testString =
        "This is a sentence. It has fruits, vegetables, etc. " +
        "but does not have meat. Mr. Smith went to Washington.";
sentIterator.setText(testString);
int start = sentIterator.first();
int end = -1;
List<String> sentences = new ArrayList<String>();
while ((end = sentIterator.next()) != BreakIterator.DONE) {
  String sentence = testString.substring(start, end);
  start = end;
  sentences.add(sentence);
  System.out.println("Предложение: " + sentence);
}
```

В результате прогона этого примера получаем:

```
Предложение: This is a sentence.
Предложение: It has fruits, vegetables, etc. but does not have meat.
Предложение: Mr.
Предложение: Smith went to Washington.
```

Хотя `BreakIterator` правильно обработал встретившееся слово *etc.* (и т. д.), он запутался с сокращением *Mr.* (Г-н). Чтобы исправить эту ошибку, необходимо добавить код для обработки особых случаев: аббревиатур, кавычек и других возможных вариантов ложного конца предложения. А еще лучше воспользоваться более надежной программой распознавания предложений, например, той, что имеется в проекте OpenNLP и продемонстрирована ниже.

Листинг 2.3. Распознаватель предложений из проекта OpenNLP

```
//... Настраиваем модели
File modelFile = new File(modelDir, "en-sent.bin");
InputStream modelStream = new FileInputStream(modelFile);
SentenceModel model = new SentenceModel(modelStream);
SentenceDetector detector =
new SentenceDetectorME(model);
String testString =
```

Создаем объект SentenceDetector с моделью en-sent.bin

```
    "This is a sentence. It has fruits, vegetables," +
    " etc. but does not have meat. Mr. Smith went to Washington.";
String[] result = detector.sentDetect(testString);
for (int i = 0; i < result.length; i++) {
    System.out.println("Предложение: " + result[i]);
}
```

Вызываем распознаватель

Прогон этой программы дает правильный результат:

```
Предложение: This is a sentence.
Предложение: It has fruits, vegetables, etc. but does not have meat.
Предложение: Mr. Smith went to Washington.
```

2.2.6. Грамматика и грамматический анализ

Один из самых сложных элементов – грамматический анализ, или разбор предложения, результатом которого является осмысленное структурированное дерево связей, – является и одним из самых полезных. Например, выделение именных и глагольных групп и их связей между собой может помочь при определении подлежащего, а равно предпринимаемых им действий. Разбор предложения затрудняется свойственной всем естественным языкам неоднозначностью. Например, на рис. 2.1 показано дерево разбора следующего предложения:

```
The Minnesota Twins, the 1991 World Series Champions, are currently
in third place.

(Миннесота Твинс, чемпион мировой серии 1991 года, сейчас
занимает третье место.)
```

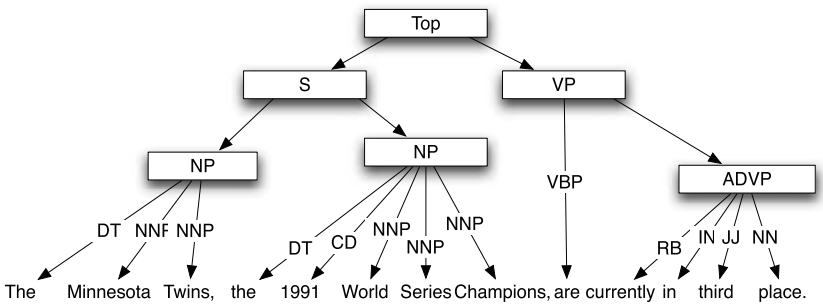

Рис. 2.1. Пример разбора предложения с помощью грамматического анализатора OpenNLP

Это дерева разбора было создано классом грамматического анализатора `Parser` из проекта OpenNLP. В нем применяется синтаксис, заимствованный из проекта Penn Treebank, который мы уже упоминали в разделе 2.2.3. Ниже показан код разбора:

```
File parserFile = new File(modelDir, "en-parser-chunking.bin");
FileInputStream parserStream = new FileInputStream(parserFile);
ParserModel model = new ParserModel(parserStream);

Parser parser = ParserFactory.create(
      model,
      20, // размер луча
      0.95); // процент продвижения

Parse[] results = ParserTool.parseLine(
        "The Minnesota Twins , the 1991 World Series " +
        "Champions , are currently in third place .",
        parser, 3);
for (int i = 0; i < results.length; i++) {
  results[i].show();
}
```

Здесь главный шаг – вызов метода `parseLine()`, который принимает подлежащее разбору предложение и максимальное число возможных вариантов разбора. В результате прогона этого примера получается такой результат (из типографических соображений часть вариантов опущена), в котором каждая строка соответствует одному варианту разбора:

```
(TOP (S (NP (NP (DT The) (NNP Minnesota) (NNS Twins)) (, ,) (NP (DT the)
...
(TOP (S (NP (NP (DT The) (NNP Minnesota) (NNPS Twins)) (, ,) (NP (DT the)
...
(TOP (S (NP (NP (DT The) (NNP Minnesota) (NNP Twins)) (, ,) (NP (DT the)
...
```

Пристальное изучение результатов показывает тонкости разбора, прежде всего, в разметке слова *Twins*. В первом варианте *Twins* помечено как имя существительное во множественном числе (NNS), во втором – как имя собственное во множественном числе, а в третьем – как имя собственное в единственном числе. От выбора варианта зависит качество результата, потому что если слово *Twins* трактуется как имя существительное нарицательное (первый случай), то инструмент, пытающийся извлечь именные группы с собственными именами (которые часто очень важны в тексте), вообще пропустит словосочетание *Minnesota Twins*. В спортивной статье это, может быть, и не

критично, но представьте, что вы пишете приложение, задача которого – находить людей, замешанных в преступлении, путем анализа статей, где обсуждаются ключевые участники преступных групп.

Здесь мы показали пример полного разбора, однако полные (или глубокие) грамматические анализаторы нужны не всегда. Во многих приложениях достаточно *поверхностного разбора*. В этом случае выделяются важные фрагменты предложения, например именные и глагольные группы, но связывать их или определять уточненные структуры необязательно. Например, в примере с Minnesota Twins выше поверхностный разбор вернул бы только *Minnesota Twins* и *1991 World Series Champions* или какие-то варианты этих частей предложения.

Грамматический анализ – содержательная и сложная область, где ведутся активные исследования, большинство которых выходят за рамки этой книги. В нашей вопросно-ответной системе разбор применяется, чтобы правильно определить структуру предложения и вычленить ответ. По большей части, мы будем рассматривать разбор как черный ящик, которым можно воспользоваться при необходимости, но копаться в его внутренностях не станем.

2.2.7. Моделирование последовательности

Представленные до сих пор конструкции дают средства выделения поверхностных особенностей, а также лингвистически мотивированных атрибутов текста. Теперь мы рассмотрим моделирование текста в виде последовательности слов или символов. Распространенный способ моделирования последовательности состоит в том, чтобы исследовать каждый ее элемент, а также предшествующий и последующий элементы. Насколько велик или мал размер учитываемого объемлющего контекста, зависит от приложения, но в общем случае можно представлять себе контекстное окно, размер которого может задаваться программой. Например, последовательности символов могут быть полезны для обнаружения совпадений при поиске в данных, прошедших процесс OCR (оптическое распознавание символов). Они могут оказаться полезны и при поиске фраз или при работе с языками, в которых слова не разделяются пробелами.

Например, если используется окно размером 5 вокруг среднего моделируемого элемента, то рассматриваются два предшествующих ему в последовательности и два следующих за ним. Вместе со средним элементом мы рассматриваем не более пяти соседних элементов последовательности – это и есть размер окна. Можно считать, что при обработке каждого элемента окно сдвигается вдоль входной последо-

вательности. Чтобы для каждого элемента последовательности размер окна был одинаков, часто в начало и в конец последовательности добавляют граничные слова (см. табл. 2.11).

Таблица 2.11. Пример окна *n*-граммы размером 5 в позиции 6 предложения

-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bos	bos	I	ca	n't	believe	that <-	the -	Carolina окно	Hurricanes -	won ->	the	2005-2006	Stanley	Cup	.	eos	eos

Эту схему часто называют *моделированием n-грамм*. Идея в том, что размер окна равен *n*, а слова внутри окна образуют *n-граммы*. В табл. 2.12 приведены примеры нескольких *n*-грамм разных размеров для нашего предложения.

Таблица 2.12. *N*-граммы разных размеров

Униграммы	believe	Stanley	the	Carolina
Биграммы	believe, that	Stanley, Cup	the, Carolina	
Триграммы	believe, that, the	2005-2006, Stanley, Cup		
4-граммы	can't, believe, that, the	2005-2006, Stanley, Cup,.		
5-граммы	that, the, Carolina, Hurricanes, won			

Та же идея применима к моделированию символов, если каждый символ рассматривать как элемент в окне *n*-граммы. Такой подход к использованию *n*-грамм совместно с символами будет применен в главе 4.

N-граммы легко поддаются декомпозиции, а, значит, удобны для статистического моделирования и оценивания. В нашем примере для моделирования контекста, окружающего слово *Carolina*, взята 5-грамма. Маловероятно, что мы увидим точно такую последовательность слов, и трудно высказать какие-то предположения о вероятности ее появления. Однако мы можем оценить вероятность, основываясь на триграмме *the Carolina Hurricanes* и даже на биграммах *the,Carolina* и *Carolina,Hurricanes*. Такой способ оценивания с ослаблением, когда больший контекст используется в сочетании с меньшими, легче поддающимися оцениванию, часто применяется для статистического моделирования неоднозначности текста.

В большинстве видов обработки текста применяется комбинация поверхностных характеристик строки, лингвистических единиц и моделирования последовательности или *n*-грамм. Например, в типичном частеречном разметчике лексический и грамматический анализ предложения используются, чтобы определить, какие элементы размечаются, манипуляции со строками – для моделирования префиксов и суффиксов и моделирование последовательности – чтобы запомнить слова, находящиеся до и после размечаемого слова. И хотя эти приемы не позволяют уловить смысл, или семантику текста, они на удивление эффективны. Этот подход, сочетающий все три типа обработки, будет использован в главе 5.

Теперь, получив представление об основах обработки текста, включая манипуляции со строками, лингвистическую обработку и моделирование последовательностей с помощью *n*-грамм, мы можем обратиться к одному из вопросов, с которыми разработчик сталкивается в первую очередь: как выделить текст из файла в том или ином распространенном формате, например HTML или Adobe PDF.

2.3. Предобработка и выделение содержимого из файлов в распространенных форматах

В этом разделе мы покажем, как извлекается текст из файлов в различных форматах. Мы обсудим важность предобработки и познакомимся с открытым каркасом извлечения содержимого и метаданных из таких форматов, как HTML и Adobe PDF.

2.3.1. Важность предобработки

Допустим, требуется написать приложение, которое будет искать введенные пользователем ключевые слова во всех хранящихся в компьютере файлах. Для этого нужно решить, каким образом сделать эти файлы пригодными для поиска. Чтобы составить представление о том, что предстоит сделать, вы знакомитесь с файлами на своем жестком диске и быстро обнаруживаете, что файлов многие тысячи, а то и миллионы, причем существует множество разных типов. Одни файлы представлены в формате Microsoft Word, другие – в формате Adobe Portable Document Format (PDF), третьи – в текстовых форматах типа HTML и XML. Есть и файлы в форматах конкретных фирм,

о структуре которых вы понятия не имеете. Будучи толковым программистом, вы понимаете, что справиться с этим разнобоем можно только одним способом – приведя все файлы к единому внутреннему формату. Процедура преобразования файлов разных типов к общему текстовому представлению называется *предобработкой*. В ходе предобработки можно также добавлять или модифицировать текст, чтобы сделать его пригодным для использования в библиотеке. Например, некоторые библиотеки ожидают, что в тексте уже выделены предложения. В общем, в состав предобработки можно включать любые действия, предшествующие подаче данных на вход библиотеки или приложения, которое будет эти данные использовать. В качестве одного из важнейших примеров предобработки мы рассмотрим извлечение содержимого из файлов в различных форматах.

Существует много программ с открытым исходным кодом для извлечения текста из файлов разных типов (чуть ниже мы их рассмотрим), но это, пожалуй, та область, где стоит потратить деньги на покупку библиотеки конвертеров форматов, которую можно было бы подключить к приложению. Производители коммерческих конвертеров платят таким компаниям, как Microsoft и Adobe, лицензионные отчисления, что дает им доступ к документации и библиотекам, которого авторы конвертеров с открытым исходным кодом лишены. Кроме того, вы получаете техническую поддержку и сопровождение. Несправедливо, конечно, но никто не говорил, что жизнь справедлива! Однако прежде чем платить за коммерческую программу, получите ознакомительную версию и протестируйте ее на своих документах. Мы протестировали, по меньшей мере одну, хорошо известную коммерческую библиотеку на тех же PDF-файлах (наиболее сложных с точки зрения извлечения текста), что и открытую библиотеку. Оказалось, что открытая библиотека не уступает, а то и лучше коммерческой. Учитывая стоимость коммерческой библиотеки, выбор в данном случае не составил труда. Но у вас может быть другое содержимое, и решение, возможно, окажется иным.

Поскольку эта книга посвящена инструментам обработки текста с открытым исходным кодом, было бы неправильно пройти мимо имеющихся средств предобработки. В табл. 2.13 перечислены некоторые употребительные файловые форматы и библиотеки – поставляемые в дистрибутиве Java или открытые – для извлечения текста из них. Мы не можем охватить все файловые форматы, поэтому сосредоточимся на тех, с которыми вам, скорее всего, придется столкнуться в своих приложениях. Но при любом типе файла и любой библиотеке – ком-

мерческой или открытой – в большинстве приложений внутреннее представление данных должно быть простым текстом. Это позволит использовать базовые библиотеки для работы со строками, поставляемые вместе с Java, Perl и большинством других современных языков программирования. Поскольку библиотек и подходов к извлечению текста много, лучше всего либо разработать собственный каркас для отображения файлового формата на содержимое, либо воспользоваться существующим.

Таблица 2.13. Часто встречающиеся файловые форматы

Файловый формат	Тип MIME	Библиотека с открытым исходным кодом	Примечания
Текст	plain/text	Встроенная	
Microsoft Office (Word, PowerPoint, Excel)	application/msword, application/vnd.msexcel и т. д.	1. Apache POI 2. Open Office 3. textmining.org	textmining.org годится только для MS Word
Adobe Portable Document Format (PDF)	application/pdf	PDFBox	Если PDF-файл содержит графику, то для извлечения текста необходимо сначала подвергнуть его оптическому распознаванию символов.
Rich Text Format (RTF)	application/rtf	Встроенный в Java класс RTFEditorKit	
HTML	text/html	1. Jtidy 2. CyberNeko 3. Многие другие	
XML	text/xml	Существует много библиотек для работы с XML (очень популярна Apache Xerces)	В большинстве приложений необходимо разбирать файл с помощью анализатора SAX, а не путем построения модели DOM – чтобы не создавать дублирующие структуры данных
Почта	Неприменимо	Java Mail API, экспорт почтового сообщения в файл, mstor	Ситуация зависит от почтовых клиента и сервера

Таблица 2.13. (окончание)

Файловый формат	Тип MIME	Библиотека с открытым исходным кодом	Примечания
Базы данных	Неприменимо	JDBC, Hibernate, другие библиотеки, экспорт базы	

По счастью, существует несколько проектов, предлагающих каркас для предобработки. Они обертывают многие библиотеки, перечисленные в табл. 2.13. При таком подходе в приложении можно использовать единый унифицированный интерфейс ко всем библиотекам. Один такой открытый проект называется Apache Tika (http://tika.apache.org/), с ним мы и познакомимся далее.

2.3.2. Извлечение содержимого с помощью Apache Tika

Tika – каркас, предназначенный для извлечения содержимого из различных источников, в том числе Microsoft Word, Adobe PDF, простой текст и многие другие. Tika не только обертывает многочисленные библиотеки извлечения текста, но и предоставляет средства распознавания типа MIME, т. е. с помощью Tika можно автоматически определить тип содержимого, а затем разобрать его с помощью подходящей библиотеки.

Если вы не найдете нужного формата в Tika, то не отчаивайтесь – нередко в сети можно найти библиотеку или приложение для работы с этим форматом, которую можно подключить к каркасу (и внести свой вклад в проект!) или использовать автономно. Даже при работе с таким каркасом, как Tika, рекомендуется создавать интерфейсы, обертывающие процесс извлечения, поскольку очень может статься, что впоследствии возникнет необходимость добавить отсутствующий файловый формат, и новый код хорошо бы чисто интегрировать со своей кодовой базой.

На архитектурном уровне Tika работает похоже на SAX-анализаторы (Simple API for XML, см. http://www.saxproject.org/) для разбора XML-документов. Tika извлекает данные из исходного файла (в формате PDF, Word и т. д.) и генерирует события, которые могут быть обработаны приложением. Это тот же самый механизм обратных вызовов, что в интерфейсе `ContentHandler` из SAX, поэтому он не вызовет трудностей у человека, работавшего с SAX в других проек-

тах. Для взаимодействия с Tika нужно всего лишь создать экземпляр одного из классов, реализующих интерфейс `Parser` с единственным методом:

```
void parse(InputStream stream, ContentHandler handler,
           Metadata metadata, ParseContext parseContext)
throws IOException, SAXException, TikaException;
```

Методу `parse` нужно всего лишь передать содержимое в виде объекта `InputStream`, и генерируемые события будут обрабатываться реализацией `ContentHandler`, предоставленной приложением. Метаданные, описывающие содержимое, будет помещены в объект `Metadata`, который, по существу, является обычной хеш-таблицей.

В состав Tika входит несколько готовых реализаций интерфейса `Parser`, по одной для каждого поддерживаемого типа MIME плюс класс `AutoDetectParser`, который умеет автоматически определять тип MIME. Кроме того, Tika поставляется с несколькими реализациями интерфейса `ContentHandler`, соответствующими типичным сценариям извлечения, например, извлечения одного лишь главного тела файла.

Познакомившись с положенной в основу Tika идеологией и базовым интерфейсом, мы можем рассмотреть примеры извлечения текста из файлов разных форматов. Начнем с простого случая – извлечения текста из HTML-файла – а затем проиллюстрируем разбор PDF-файлов.

Итак, допустим, что требуется разобрать такой простенький HTML-файл:

```
<html>
 <head>
  <title>Best Pizza Joints in America</title>
 </head>
 <body>
  <p>The best pizza place in the US is
     <a href="http://antoniospizzas.com/">Antonio's Pizza</a>.
  </p>
  <p>It is located in Amherst, MA.</p>
 </body>
</html>
```

Глядя на этот пример, вы, скорее всего, захотите извлечь заглавие, тело и, возможно, ссылки. Все это Tika позволяет сделать без труда.

Листинг 2.4. Извлечение текста из HTML-файла с помощью Tika

```
InputStream input = new ByteArrayInputStream(
                    html.getBytes(Charset.forName("UTF-8")));
ContentHandler text = new BodyContentHandler();
LinkContentHandler links = new LinkContentHandler();
ContentHandler handler = new TeeContentHandler(links, text);
Metadata metadata = new Metadata();
Parser parser = new HtmlParser();
ParseContext context = new ParseContext();
parser.parse(input, handler, metadata, context);
System.out.println("Title: " + metadata.get(Metadata.TITLE));
System.out.println("Body: " + text.toString());
System.out.println("Links: " + links.getLinks());
```

ContentHandler, который знает о ссылках

ContentHandler, который извлекает текст внутри тега body

Объединяем два ContentHandler'а в один

Объект, в котором сохраняются метаданные

На входе HTML-файл, конструируем соответствующий ему анализатор

Производим разбор

В результате обработки приведенного выше HTML-файла этой программой будет напечатано:

```
Title: The Big Brown Shoe
Body: The best pizza place in the US is Antonio's Pizza.
It is located in Amherst, MA.
Links: [<a href="http://antoniospizzas.com/">Antonio's Pizza</a>]
```

Код для разбора HTML-файла состоит из двух частей: конструирование объектов `ContentHandler` и `Metadata`, за которым следует создание и выполнение объекта `Parser`, т. е. собственно разбор файла. В примере выше мы воспользовались для разбора содержимого классом `HTMLParser`, но в большинстве случаев предпочтительно использовать имеющийся в Tika класс `AutoDetectParser`, как показано в следующем примере разбора PDF-файла.

Листинг 2.5. Использование класса `AutoDetectParser` для определения типа и извлечения содержимого

```
InputStream input = new FileInputStream(
            new File("src/test/resources/pdfBox-sample.pdf"));
ContentHandler textHandler = new BodyContentHandler();
Metadata metadata = new Metadata();
Parser parser = new AutoDetectParser();
ParseContext context = new ParseContext();
parser.parse(input, textHandler, metadata, context);
```

Создаем объект InputStream для чтения содержимого

В объекте Metadata будут храниться метаданные, описывающие содержимое, в частности, автор и название

Объект AutoDetectParser автоматически определяет тип MIME документа при вызове его метода parse. Поскольку мы заранее знаем, что это PDF-файл, можно было бы вместо этого создать объект PDFParser

Производим разбор

```
System.out.println("Title: " + metadata.get(Metadata.TITLE));
System.out.println("Body: " + textHandler.toString());
```

Получаем название из объекта Metadata

Распечатываем тело, извлеченное объектом ContentHandler

В последнем примере на вход был подан PDF-файл в виде объекта `InputStream`. Далее мы сконструировали объект одного из входящих в состав Tika классов, реализующих интерфейс `ContentHandler`, и объект класса `Metadata` для хранения дополнительной информации, например, автора и количества страниц в документе. Наконец, мы создали объект `Parser`, разобрали документ и распечатали сведения о нем. Как видите, все просто.

То, что Tika позволяет так легко работать с различными файловыми форматами, уже неплохо, но хорошие новости этим не исчерпываются: Tika интегрирован в пакет Apache Solr благодаря расширению Solr Content Extraction Library (оно же *Solr Cell*). В главе 3 мы покажем, как просто отправить документ любого типа системе Solr, которая индексирует его и сделает доступным для поиска. Но даже если вы не используете Solr для поиска, можно настроить ее в качестве сервера извлечения содержимого, который будет возвращать извлеченный из документа текст, не индексируя его.

2.4. Резюме

В этой главе мы рассмотрели основные элементы естественного языка: морфологию, грамматику, синтаксис и семантику, а также некоторые инструменты для работы с ними. Затем мы остановились на неизменно возникающей задаче предобработки файлов с целью извлечения полезного содержимого. И хотя эти кусочки головоломки под названием «Обработка текста» не слишком увлекательны, они почти всегда необходимы. Многие не обращают внимания на важность извлечения текста из файлов в фирменных форматах, стремясь поскорее перейти к анализу текста, однако не следует забывать, что извлечение содержимого в виде полезного текста часто оказывается трудоемкой задачей, которую трудно реализовать правильно. Точно так же понимание основных понятий языка в объеме, описанном в этой главе, необходимо для понимания дальнейшего материала, равно как и других книг на эту тему. Памятуя об этом, мы теперь сделаем первые шаги по пути нахождения и организации содержимого и займемся тем, что часто образует фундамент систем анализа текста: поиском.

2.5. Ресурсы

Hull, David A. 1966. Stemming Algorithms: A Case Study for Detailed Evaluation. *Journal of the American Society of Information Science*, volume 47, number 1.

Krovetz, R. 1993 "Viewing Morphology as an Inference Process." Proceedings of the Sixteenth Annual International (ACM) (SIGIR) Conference on Research and Development in Information Retrieval.

Левенштейн В.И. «Двоичные коды с исправлением выпадений, вставок и замещений символов», Доклады АН СССР, 163, 4, 1965, 845-848

Mihalcea, Rada, and Tarau, Paul. 2004, July. "TextRank: Bringing Order into Texts." In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2004), Barcelona, Spain.

"Parsing." Wikipedia. http://en.wikipedia.org/wiki/Parsing.

Winkler, William E., and Thibaudeau, Yves. 1991. "An Application of the Fellegi-Sunter Model of Record Linkage to the 1990 U.S. Decennial Census." *Statistical Research Report Series RR91/09*, U.S. Bureau of the Census, Washington, D.C.

ГЛАВА 3. Поиск

В этой главе:

- Теория поиска и основы векторной модели.
- Настройка Apache Solr.
- Обеспечение возможности поиска по содержимому.
- Создание запросов для Solr.
- Производительность поиска.

Поиск как функциональная возможность приложения или само приложение не нуждается во вводных словах. Это часть нашей жизни – мы ищем информацию в Интернете или на своем рабочем столе, ищем друзей в Facebook, ищем слова в тексте. Для разработчика поиск нередко является важнейшей функцией многих приложений, но особенно тех, что работают с данными, когда пользователю необходимо просеивать огромные объемы текста. Отметим еще, что поиск часто входит составной частью в готовые решения, как, например, в программу Apple Spotlight для поиска на локальном диске, или в аппаратные устройства типа Google Search Appliance.

Принимая во внимание вездесущность поиска и доступность готовых решений, возникает естественный вопрос: зачем создавать собственную систему поиска, применяя инструменты с открытым исходным кодом? Тому есть несколько основательных причин:

- *гибкость* – вы можете контролировать многие, а то и все аспекты процесса;
- *стоимость разработки* – даже купив коммерческое решение, вы все равно должны будете интегрировать его со своей программой, а это большая работа;
- *кто лучше вас знает особенности ваших данных?* – в большинстве готовых решений делаются определенные предположения о

характере содержимого, которые в вашем случае могут не отвечать действительности;
- *цена* – отсутствие лицензионных платежей. Сказанного достаточно.

Помимо этих причин, нельзя не отметить высочайшее качество инструментов поиска с открытым исходным кодом. Такие поисковые системы, как Apache Lucene, Apache Solr и другие, стабильны, масштабируемы и готовы к работе более, чем все прочие обсуждаемые в этой книге, – просто потому что используются во множестве мест. В этой книге мы построим на базе Apache Solr систему, которая сможет быстро осуществлять поиск по вашему содержимому и под вашим контролем. Начнем с изучения некоторых концепций, используемых во многих поисковых системах, в том числе Lucene и Solr. Затем мы покажем, как установить и настроить Solr, после чего перейдем к индексированию и поиску по содержимому с помощью Solr. В заключение мы приведем ряд рекомендаций и приемов повышения производительности поиска – как вообще, так и для Solr в частности.

Для начала познакомимся с идеей, которая быстро становится стандартной функцией в Интернет-магазинах типа Amazon.com и eBay: фасетным поиском. В последующих разделах, основанных на этом простом для понимания, но реальном примере, мы подумаем, как задействовать эти средства, реализованные на сайте Amazon и других, в своих собственных приложениях. К концу главы вы не только будете понимать базовые концепции поиска, но сможете настроить и запустить настоящую поисковую систем и будете знать, как заставить ее работать быстро.

3.1. Пример фасетного поиска: Amazon.com

Все мы с этим знакомы. Вы заходите на сайт магазина и не можете точно подобрать ключевые слова, чтобы найти именно то, что нужно, не занимаясь беспорядочным перебором сотен результатов. По крайней мере, это характерно для сайтов, не имеющих функции фасетного поиска. Предположим, к примеру, что вы ищете новый экономичный LCD-телевизор с диагональю 50 дюймов, получивший хорошие отзывы. Набираете *LCD TV* и получаете в ответ что-то наподобие показанного на рис. 3.1. Естественно, на первой странице результатов

того, что вам нужно, нет. Да и странно было бы ожидать иного при таком общем запросе. Вы начинаете уточнять поиск. В системах без фасетного поиска для этого приходится добавлять в запрос ключевые слова, например: *50 inch LCD TV* или *Sony 50 inch LCD TV*. Но не зная точно, что нужно, а имея лишь смутные представления, вы начинаете исследовать предоставляемые Amazon фасеты. *Фасеты* – это категории, выведенные из результатов поиска, которые полезны для сужения области поиска. В случае Amazon (рис. 3.1) они отображаются слева под заголовком Department и могут принимать такие значения, как Electronics (4,550) или Baby (3). Почти всегда фасет сопровождается количеством элементов в соответствующей категории, чтобы пользователю было проще решить, что для него интересно.

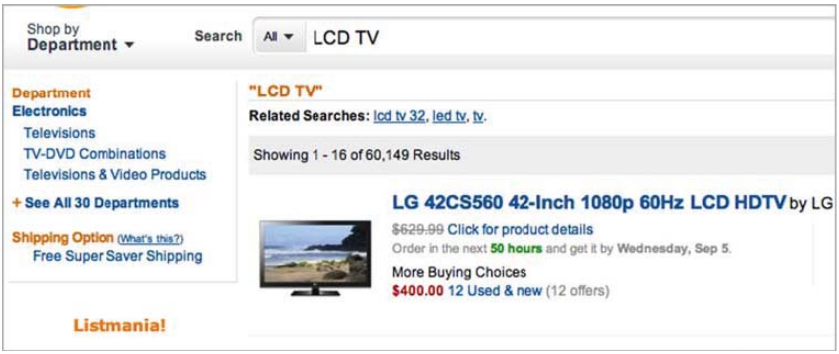

Рис. 3.1. Фрагмент результатов поиска по запросу «LCD TV» на сайте Amazon.com. Снимок экрана сделан 2.09.2012

Но вернемся к нашему примеру. Мы знаем, что телевизор – электронный прибор, поэтому щелкаем по фасету Electronics и получаем фасеты, показанные на рис. 3.2. Обратите внимание, что фасеты изменились и теперь отражают новые результаты поиска. Перечисленные категории соотносятся как с поисковыми словами, так и с ранее выбранными фасетами. Более того, гарантируется, что в каждой показанной категории что-то есть, потому что сами категории выбраны, исходя из найденных результатов.

Наконец, указав дополнительные фасеты, например: Portable Audio & Video, 4 stars and up[1] и ценовой диапазон от 50 до 100 долларов, вы можете дойти до страницы с обозримым набором результатов (рис. 3.3). Вот теперь легко выбрать из того, что есть.

[1] Не ниже 4 звезд по стандарту энергопотребления Energy Star. – *Прим. перев.*

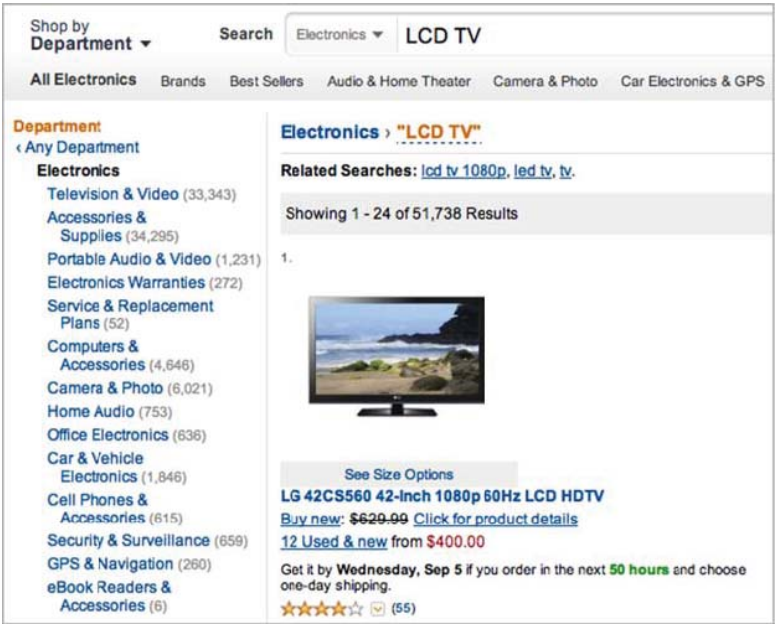

Рис. 3.2. Фасеты для запроса «LCD TV» после выбора фасета Electronics. Снимок экрана сделан 2.09.2012

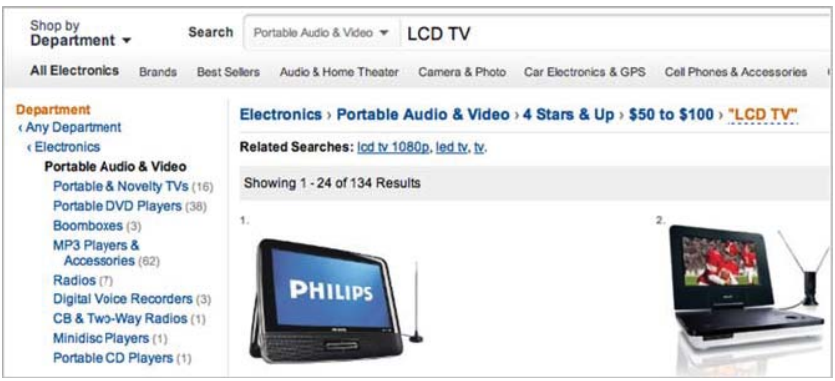

Рис. 3.3. Результаты поиска по запросу «LCD TV», суженные выбором нескольких фасетов. Снимок экрана сделан 2.09.2012

На примере Amazon мы показали, что фасетный поиск – мощное средство для сайтов, на которых имеется сочетание структурированных (метаданные) и неструктурированных (текстовых) данных.

К таковым относятся, в частности, Интернет-магазины, библиотеки, научные классификаторы. Для вычисления фасетов необходимо исследовать метаданные, ассоциированные с результатами поиска, а затем выполнить группировку и подсчет. (По счастью, Solr все это уже умеет делать.) Но даже если оставить фасеты в стороне, очевидно, что если пользователь не найдет чего-то, то он не сможет это что-то купить или узнать, какие у него есть возможности. Поэтому повышение качества поиска – это не просто экзотическое упражнение в программировании; оно может стать решающим условием успеха вашего предприятия. Чтобы понять, как добавить и усовершенствовать средства поиска, отступим на шаг назад и познакомимся с базовыми концепциями поиска.

3.2. Введение в концепции поиска

Перед тем как приступить к изучению концепций поиска, давайте начнем с чистого листа и забудем все, что когда-либо знали о поиске в веб (по крайней мере, на время). Забудьте о Google. Забудьте о Яндексе и Bing. Забудьте об алгоритме PageRank (если знаете, что это такое), о центрах обработки данных и о тысячах процессоров, прочесывающих потаенные закоулки Интернета в поисках каждого байта, который можно было бы включить в поисковую систему. Счистим эту шелуху, и нашему взгляду откроются основополагающие концепции поиска. На самом базовом уровне поисковая система состоит из четырех частей.

1. *Индексирование* – файлы, сайты и записи базы данных обрабатываются, так чтобы обеспечить возможность поиска в них. Далее мы будем называть индексированные файлы документами.
2. *Ввод данных пользователем* – чтобы обозначить свои информационные потребности, пользователю необходим какой-то интерфейс.
3. *Ранжирование* – поисковая система сравнивает запрос со всеми найденными в индексе документами и ранжирует документы в зависимости от того, насколько точно они соответствуют запросу.
4. *Отображение результатов* – награда пользователю: окончательные результаты отображаются в пользовательском интер-

фейсе, будь то командная строка, окно браузера или экран мобильного телефона.

В следующих разделах мы рассмотрим каждый из этих процессов более подробно.

3.2.1. Индексирование содержимого

Какими бы замечательными ни были механизм ввода запроса и алгоритмы ранжирования, если вы не имеете ясного представления о структуре и типе содержимого в вашем наборе данных, а, значит, поисковая система не понимает, что в документе важно, а что нет, то никакие математические ухищрения не позволят улучшить результат. Например, если во всех документах имеется атрибут «название» и вы знаете, что совпадение с названием часто является наиболее информативным, то можете настроить поисковую систему, так чтобы она повышала вес документов с подходящими названиями, перемещая их на более высокое место в результатах. Аналогично, если в данных встречается много дат и чисел или имен людей и словосочетаний, то, возможно, придется проделать дополнительную работу по правильному индексированию содержимого. С другой стороны, вообразите, насколько негодной оказалась бы поисковая система, если бы она индексировала все HTML-теги на новостном сайте, не отличая их от собственно содержимого. Понятно, что это надуманный пример, но он показывает, как важно подходить к созданию и совершенствованию приложений итеративно, отвечая на нужды пользователей. Первая задача любого человека, приступающего к реализации поиска, – получить представление об индексируемом содержимом. Это исследование должно охватывать как типичную структуру документа, так и его фактическое наполнение.

Составив предварительное представление о содержимом, можно приступать к процессу, обеспечивающему возможность поиска, – *индексированию*. Чтобы в документе можно было искать, процесс индексирования должен проанализировать его содержимое. Обычно анализ документа состоит из разбиения его на лексемы и, возможно, видоизменения каждой лексемы для приведения ее к нормализованному виду, называемому *термом*. К числу видоизменений, порождающих терм из лексемы, относятся стемминг, преобразование в нижний регистр или вообще удаление. Обычно за принятие решения о том, какие изменения применять, отвечает приложение. Одни приложения не вносят никаких изменений, тогда как другие изменяют

лексемы очень сильно. В некоторых случаях поисковая система мало что может предложить в плане управления анализом. Хотя поначалу может показаться, что отсутствие управления упрощает работу, впоследствии это, скорее всего, аукнется – когда выяснится, что результаты не дотягивают до требуемого стандарта качества. В табл. 3.1 приведены некоторые распространенные подходы к преобразованию лексем в термы для целей индексирования.

Таблица 3.1. Типичные методы анализа

Метод	Описание
Лексический анализ	Процесс разбиения строки на лексемы, подлежащие индексированию. Важно правильно и единообразно обрабатывать знаки препинания, числа и другие символы. Например, слово *microprocessor* можно разбить на несколько лексем (micro, processor и microprocessor), чтобы было больше шансов найти ответ на запрос пользователя с учетом вариантов слова.
Приведение к нижнему регистру	Все слова преобразуются в нижний регистр, чтобы было проще производить поиск без учета регистра.
Стемминг	Убирает суффиксы слов и т. п. Описан в главе 1.
Удаление стоп-слов	Удаляет такие расхожие слова, как *the*, *and* и *a*, которые часто встречаются в большинстве документов. Поначалу это делалось для экономии места в индексе, но некоторые современные поисковые системы перестали удалять стоп-слова, поскольку они могут быть полезны при выполнении расширенных видов поиска.
Расширение синонимами	Для каждой лексемы производится ее поиск в тезаурусе, и найденные синонимы добавляются в индекс. Часто это делается для поисковых, а не индексных термов, поскольку в этом случае обновленный список синонимов динамически учитывается на этапе запроса, так что переиндексирование не нужно.

После извлечения из документа термы обычно сохраняются в структуре данных, называемой *инвертированным индексом*. Эта структура оптимизирована для быстрого поиска документов, содержащих некий терм. Когда пользователь вводит поисковый терм, система быстро находит все содержащие его документы. На рис. 3.4 показаны ссылки со слов в инвертированном индексе (который часто называют *словником*) на документы, в которых они встречаются. Во многих поисковых системах в индексе хранятся не только ссылки между термами и документами, но и позиции каждого терма в документе. Это упрощает выполнение фразовых и других расширенных

запросов, в которых информация о позициях нужна для вычисления того, насколько близко два или более термов расположены друг к другу.

Помимо сохранения термов, в процессе индексирования часто вычисляется и сохраняется информация о важности одних термов относительно других в том же документе. Это вычисление играет главную роль в переходе от модели простого булева совпадения (существует терм в документе или нет) к модели ранжирования, позволяющей придавать более релевантным документам больший вес (и более высокое место в списке результатов), чем менее релевантным. Нетрудно догадаться, что умение ранжировать документы по релевантности – огромный шаг вперед в ситуации, когда объем данных очень велик, поскольку это дает пользователю возможность сосредоточиться только на самых релевантных документах. Более того, вычисляя как можно больше показателей на этапе индексирования, поисковая система способна уменьшить время поиска и ранжирования. Пока этих сведений о деталях индексирования довольно. Посмотрим теперь, что хотят от индекса пользователи поисковой системы.

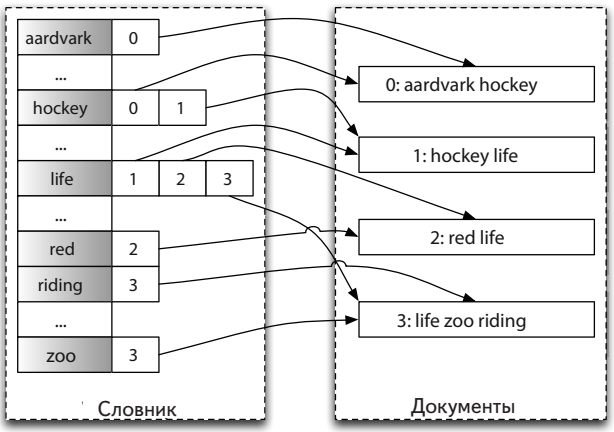

Рис. 3.4. Структура данных, называемая инвертированным индексом, сопоставляет термам документы, в которых они встречаются, что позволяет поисковой системе быстро производить поиск по заданным термам. Слева показан пример словника, а справа – документы, в которых эти слова встречаются. В инвертированном индексе хранится информация о том, в каких документах встречается каждый терм.

3.2.2. Ввод запроса пользователем

Обычно поисковые системы предлагают пользовательский интерфейс, позволяющий ввести различные данные: ключевые слова, типы документов, язык, диапазон дат и т. п. А в ответ возвращают ранжированный список документов, наиболее релевантных запросу. Многие поисковые системы в Интернете ограничиваются только вводом ключевых слов, как показано на рис. 3.5, но и они предлагают механизм расширенного поиска, показанный (только частично) на рис. 3.6.

Рис. 3.5. Сайт http://search.yahoo.com предлагает простой интерфейс для ввода поискового запроса

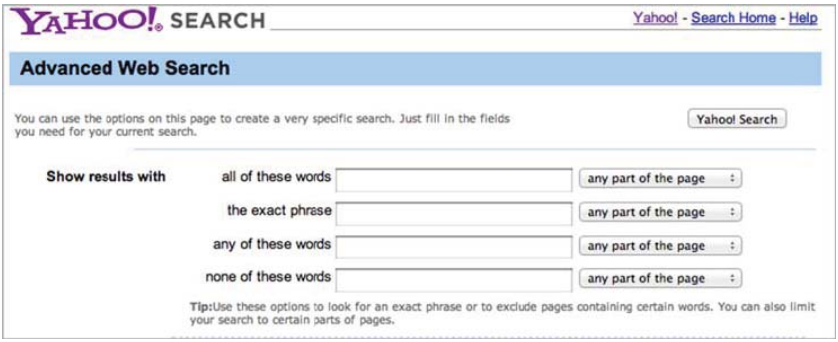

Рис. 3.6. На сайте http://search.yahoo.com есть и расширенный интерфейс поиска (доступен по ссылке More), который позволяет точнее задать условия

Принимая решение о механизмах ввода, важно учитывать, какой контингент будет пользоваться системой и насколько удобны для него те или иные механизмы. Пользователи Интернета привыкли к простому поиску по ключевым словам, тогда как более умудренным и опытным пользователям нужны средства для получения более точных результатов. Во многих случаях имеет смысл предложить как простой интерфейс для поиска по словам, так и более развитый для опытных пользователей.

С годами средства поисковых систем в части формулирования запросов неуклонно расширялись. Пользователи получили возмож-

ность вводить сложные запросы со словосочетаниями, метасимволами, регулярными выражениями и даже на естественном языке. Кроме того, во многих универсальных поисковых системах используются различные операторы, например AND, OR, NOT, кавычки для фразовых запросов и т. д., что позволяет сужать множество результатов. Коммерческие системы идут еще дальше и предлагают специальные операторы, предназначенные для прямой работы с внутренними структурами данных, файловыми форматами и, в конечном счете, для расширения возможностей запросов. В табл. 3.2 перечислены наиболее распространенные типы запросов и применяемые операторы. Поисковые системы иногда неявно создают подобные запросы, а пользователь даже не подозревает об этом. Например, системы, принимающие запросы на естественном языке (пользователь вводит в качестве запроса целое предложение или даже абзац), зачастую автоматически выделяют словосочетания на этапе обработки запроса и передают исполняющей поисковой системе фразовые запросы. С другой стороны, расширенный поисковый интерфейс Google Canada предлагает простые текстовые поля, с помощью которых пользователь может создавать сложные фразовые и булевы запросы (рис. 3.7), даже не вводя кавычек или специальных операторов. Более того, многие поисковые системы выполняют несколько внутренних запросов для каждого запроса, введенного пользователем, а затем собирают все полученные результаты для представления пользователю. При таком подходе система может использовать несколько разных стратегий для нахождения наилучших результатов.

Таблица 3.2. Наиболее распространенные типы запросов и операторы

Тип запроса и операторы	Описание	Пример
По ключевым словам	Каждый терм – самостоятельный поиск в индексе.	dog programming baseball
Фразовый	Термы должны находиться в документе рядом или, по крайней мере, отстоять друг друга не далее, чем на заданное пользователем расстояние. Обычно для обозначения начала и конца фразы (словосочетания) используются двойные кавычки.	«President of the United States» «Manning Publications» «Minnesota Wild Hockey» «big, brown, shoe»

Таблица 3.2. (продолжение)

Тип запроса и операторы	Описание	Пример
С булевыми операторами	Операторы AND, OR и NOT часто применяются для соединения двух и более слов. AND означает, что в документе должны встречаться оба терма, OR – что должен встречаться хотя бы один терм, NOT – что следующий далее терм не должен встречаться в документе. Для задания порядка вычислений употребляются круглые скобки. Во многих поисковых системах неявно используется оператор AND или OR, если ничего иного не указано.	franks AND beans boxers OR briefs ((“Abraham Lincoln” AND “Civil War”) NOT (“Gettysburg Address”))
Метасимволы и регулярные выражения	Поисковые термы могут содержать как метасимволы (? и *), так и полноценные регулярные выражения. На обработку таких запросов обычно тратится больше ресурсов процессора, чем на более простые запросы.	bank? – соответствует любому слову, начинающемуся строкой *bank*, после которого идет один произвольный символ: banks. bank* – соответствует любому слову, начинающемуся строкой *bank*, после которой может идти любое количество символов: banks, banker. aa.*k – соответствует слову, начинающемуся строкой *aa*, после которого идет произвольная последовательность символов (но не менее одного), а затем символ *k*: aardvark.
Структурированный	Структурированные запросы учитывают структуру документа, в котором производится поиск. Типичные структурные элементы документа: название, дата публикации, автор, URL-адрес, оценки пользователей и т. д.	Диапазон дат – найти все документы, опубликованные в указанном временном промежутке. Найти документы конкретных авторов. Ограничить поиск одним или несколькими доменными адресами.

Таблица 3.2. (окончание)

Тип запроса и операторы	Описание	Пример
Похожие документы	Имея один или несколько уже найденных документов, найти другие, похожие на них. Иногда это называется *обратной связью по релевантности* или *еще такие же*.	Google предлагает для большинства результатов ссылку «Похожие». Если щелкнуть по ней, то по выбранному документу будет автоматически сгенерирован запрос и произведен поиск в индексе.
Направляемый поиск	Направляемый, или фасетный поиск – набирающий популярность механизм, который дает пользователям рекомендации для уточнения результатов путем указания заведомо непустых категорий.	На сайте Amazon.com фасетный поиск применяется для задания ограничений по цене, производителю и другим параметрам. Фасетные счетчики показывают, сколько элементов находится в каждой категории.

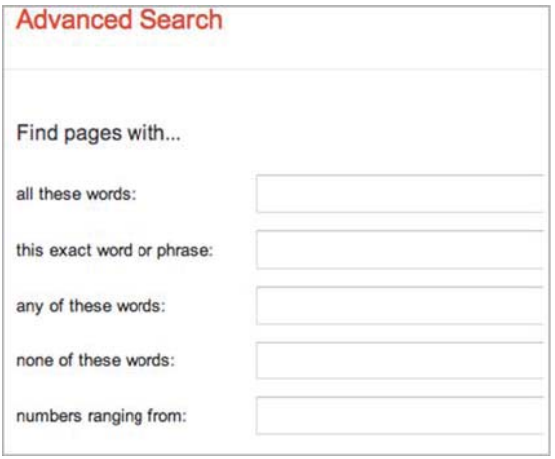

Рис. 3.7. Расширенный интерфейс поиска на сайте Google Canada автоматически строит сложные фразовые и булевы запросы, не заставляя пользователя вводить зарезервированные слова типа AND, OR, NOT или кавычки для задания словосочетаний

Некоторые поисковые системы заходят настолько далеко, что пытаются классифицировать запрос по виду и в зависимости от него выбирают различные параметры оценивания. Например, в Интернет-магазинах типичны два вида запросов: по известному товару и по категории либо ключевому слову. Поиск *по известному товару* производится, когда пользователь точно (или почти точно) знает название товара и хочет только выяснить, в каком месте магазина этот товар находится. Например, поиск *Sony Bravia 53-inch LCD TV* – это

поиск по известному товару, поскольку для него, скорее всего, найдется только одно-два соответствия. Поиск по категории гораздо более общий и часто включает небольшое количество ключевых слов: *televisions* или *piano music* (фортепьянная музыка). В случае поиска по известному товару отсутствие указанного товара среди первых нескольких результатов считается ошибкой системы. В случае поиска по категории больше свободы в выборе того, что возвращать, поскольку термы зачастую довольно общие.

После передачи поисковой системе лексемы запроса обычно подвергаются такому же анализу, как индексные лексемы, т. е. проходят через те же преобразования от лексем к термам. Например, если перед помещением в индекс лексемы подвергались стеммингу, то точно так же следует поступить с лексемами запроса. Многие поисковые системы также производят расширение запроса синонимами. *Расширение синонимами* – это вид анализа, при котором каждая лексема ищется в определенном пользователем тезаурусе. Если поиск оказался успешным, то в список лексем добавляются новые лексемы, соответствующие синонимам. Например, если исходный запрос состоял только из терма *bank*, то в ходе анализа в него незаметно для пользователя можно добавить лексемы *financial institution* (финансовое учреждение) и *credit union* (кредитный союз). Расширение синонимами можно производить и во время индексирования, но это часто приводит к значительному увеличению размера индекса и требует переиндексирования содержимого в случае изменения списка синонимов.

Разобравшись с основами построения пользовательского интерфейса поиска, обсудим модель векторного пространства и посмотрим, благодаря чему поиск вообще работает. Это позволит нам более осмысленно выбирать компромиссы между различными подходами к поиску и принимать обоснованные решения относительно того, какой подход лучше отвечает потребностям пользователя.

3.2.3. Ранжирование документов с помощью векторной модели

Хотя информационный поиск – достаточно зрелая область по сравнению с другими обсуждаемыми в этой книге темами, это не означает, что определился один-единственный идеальный способ нахождения информации. Есть много путей моделирования задачи поиска, и у каждого свои плюсы и минусы. Мы остановимся на векторной модели (vector space model, VSM), поскольку именно она применяется

в выбранных нами поисковых библиотеках и является одним из самых популярных методов ранжирования документов относительно запроса. Если вы хотите почитать о других моделях, в том числе вероятностной, обратитесь к разделу 3.7.4, книге Baeza-Yates, Ribeiro-Neto «Modern Information Retrieval» (Baeza-Yates [2011]) или книге Grossman, Frieder «Information Retrieval: Algorithms and Heuristics» (2-е издание) (Grossman [1998]).

Краткое знакомство с векторной моделью

Векторная модель, впервые предложенная в 1975 году (Salton 1975) – это алгебраическая модель, в которой термы, встречающиеся в документе, отображаются на *n*-мерное линейное пространство. Слова-то какие ученые, но что за ними стоит? Представьте, что имеется набор документов на очень ограниченном языке, состоящем только из двух слов: *hockey (хоккей)* и *cycling* (велосипед). Теперь изобразим эти документы на двумерном графике, где по горизонтальной оси отложено *cycling*, а по вертикальной – *hockey*. Тогда документ, в котором присутствуют оба слова, можно изобразить стрелкой (вектором, или вектором термов), наклоненной под углом 45 градусов к каждой оси (рис. 3.8).

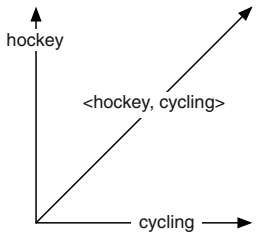

Рис. 3.8. Пример векторной модели для документа из двух слов: *hockey* и *cycling*

Наглядно представить двухмерное пространство совсем просто, но и экстраполировать это представление на большее число измерений тоже не намного труднее. Таким образом, мы можем представить все множество документов в виде векторов в *n*-мерном линейном пространстве. При этом каждому слову, встречающемуся хотя бы в одном документе, соответствует измерение. Пусть, например, в нашем наборе имеются два документа с таким содержимым:

- Документ 1: The Carolina Hurricanes won the Stanley Cup;
- Документ 2: The Minnesota Twins won the World Series.

Эти документы можно представить в векторном пространстве, пронумеровав уникальные слова. Так, слову *the* можно поставить в соответствие число 1, слову *carolina* – 2, слову *hurricanes* – 3 и т. д. Эти числа соответствуют осям векторного пространства. Легко видеть, что если у двух документов есть общее слово, то они пересекаются по соответствующей оси. На рис. 3.9 эта идея иллюстрируется на примере двух документов в 10-мерном векторном пространстве;

присутствие некоторого слова в документе означает, что в соответствующей координате вектора хранится значение 1.

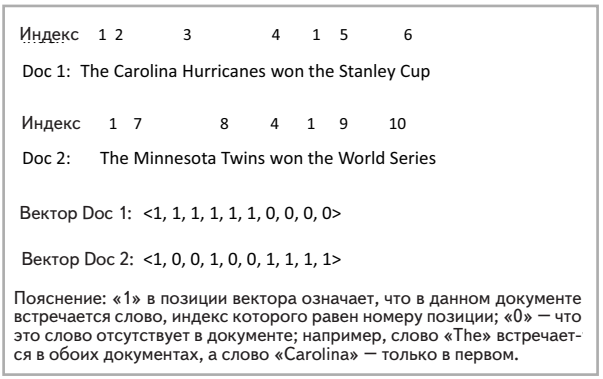

Рис. 3.9. Два документа, представленные векторами в 10-мерном пространстве

Реальные поисковые системы работают с очень большим числом измерений (*n* часто больше миллиона), поэтому описанную простую модель необходимо модифицировать с учетом требований к хранению и качеству. Что касается хранения, то в системе запоминается только наличие терма, но не его отсутствие – отсюда и инвертированный индекс. Это позволяет не хранить кучу нулей, поскольку в большинстве документов встречается лишь малая часть всех слов. Теперь о качестве: многие поисковые системы хранят не просто 1, обозначающую наличие слова, но и некий вес, описывающий важность данного терма относительно всех остальных. В математике это называется взвешенным вектором. Теперь вы начинаете понимать, что если сравнить термы запроса с термами и их весами в документах, где встречаются термы запроса, то можно будет вывести формулу, показывающую, насколько документ релевантен запросу. Чуть ниже мы вернемся к этому вопросу.

Существует много схем назначения весов, но наиболее распространена так называемая модель *TF-IDF* (*term frequency-inverse document frequency* – частота термов-обратная частота документа). Смысл ее в том, что термы, которые часто встречаются в данном документе (TF) по сравнению с частотой употребления во всех документах набора (IDF), являются более важными, чем термы, часто встречающиеся во многих документах. Можно считать, что TF и IDF – это инь и ян поиска, уравновешивающие друг друга. Например, слово *the* часто встречается в большинстве англоязычных текстов и, следовательно, у

него очень высокая частота документа (а IDF, соответственно, мала), поэтому его вклад в общий вес при оценке документа крайне низок. Мы вовсе не хотим сказать, что слово *the* и другие часто встречающиеся слова (их также называют *стоп-словами*) бесполезны для поиска; напротив, они весьма полезны во фразовых запросах и других более сложных применениях, которые выходят за рамки данного обсуждения. На противоположном конце спектра находятся слова, которые многократно встречаются в данном документе (у них высокая TF), но редко в остальных документах набора. Это ценные слова, которые вносят заметный вклад в общий вес документа относительно запроса, содержащего данный терм. Вернемся к примеру с двумя документами: слово *the* дважды встречается в первом документе и дважды во втором. Общая частота его вхождения во все документы равна 4, а это значит, что вес слова *the* в первом документе равен $2/4 = 1/2 = 0{,}5$. Слово *Carolina* один раз встречается только в первом документе, а значит, и во всем наборе, поэтому его вес равен $1/1 = 1$. Применяя похожие рассуждения ко всем термам, мы получим полный набор взвешенных векторов. Документ 1 в этом примере будет представлен таким вектором:

```
<0.5, 1, 1, 0.5, 1, 1, 0, 0, 0, 0>
```

Теперь естественно возникает следующий вопрос: как при таком представлении документов в векторном пространстве модель VSM осуществляет сопоставление запросов с документами? Прежде всего, заметим, что запросы можно отобразить на то же самое векторное пространство, что и документы. Далее, отметим, что если совместить начальные точки вектора запроса и вектора документа, то между ними образуется угол. Вспомним школьную тригонометрию: косинус этого угла – число от –1 до 1, и его можно взять в качестве ранга документа относительно запроса. Легко видеть, что если угол между двумя векторами равен 0, то мы имеем полное совпадение. Поскольку косинус нуля равен 1, выбранная мера ранжирования согласуется с этим пониманием. Эта идея наглядно изображена на рис. 3.10, где Θ обозначает угол между двумя векторами: документа (d_j) и запроса (q). Кортеж, ассо-

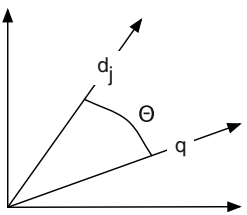

Рис. 3.10. Сравнение запроса *q* с *j*-ым документом набора в векторной модели

циированный с каждым вектором под рисунком, представляет веса, взятые при его создании (как в примере с двумя документами выше). Проделав это для всех документов в наборе, мы получим ранжированный список результатов, который можно вернуть пользователю. На практике поисковые системы обычно не оценивают все документы, а ограничиваются лишь теми, в которых встречается один или несколько термов из запроса. Кроме того, в большинстве поисковых систем к чистой оценке согласно модели VSM примешиваются другие параметры, в частности длина документа и средняя длина всех документов в наборе. Применяются также алгоритмы, которые могут назначить одному документу больший вес, чем другому, и даже приписать разные веса разделам одного документа.

Естественно, авторы поисковых систем разработали эффективные механизмы вычисления этих оценок с использованием векторной модели, позволяющие производить поиск среди миллионов (и даже миллиардов) документов за доли секунды на стандартном оборудовании. Хитрость в том, чтобы обеспечить релевантность документов, возвращаемых пользователю. Скорость поиска и релевантность будут обсуждаться в разделе 3.6. А пока предположим, что все это у нас есть, и поговорим об отображении результатов.

3.2.4. Отображение результатов

Лично для нас вопрос об отображении результатов обычно стоит на последнем месте. В конце концов, что плохого в том, чтобы просто показать первые десять упорядоченных результатов и предоставить способ для перехода к следующей порции? Будем откровенны: ничего отталкивающего в таком подходе нет, если пользователи довольны; простота – благородная цель проектировщика и хорошо бы, чтобы об этом помнило больше людей. Однако дополнительное время, потраченное на изучение оптимальных способов отображения результатов, окупится повышенным качеством взаимодействия с пользователем. Но берегитесь: хитроумное представление результатов не всегда дает самую полезную информацию, пусть даже выглядит замечательно, поэтому думайте об удобстве пользователей и о том, насколько комфортно будет ощущать себя целевая аудитория при работе с предложенным интерфейсом. Перечислим некоторые вопросы, которые стоит задать себе при проектировании отображения результатов.

- Какие части документа следует отображать? Обычно название, если оно есть, не вызывает сомнения. А как насчет реферата или доступа к исходному содержимому?

- Нужно ли выделять в результатах слова из запроса?
- Как относиться к дубликатам или почти дубликатам?
- Какие навигационные ссылки или меню вы можете предложить, чтобы пользователю было удобнее работать?
- Что делать, если пользователю не понравятся результаты или он захочет расширить или сузить область поиска?

На эти вопросы можете ответить только вы, и ответы будут зависеть от приложения. Мы рекомендуем ориентироваться на то, что целевая аудитория, скорее всего, привыкла видеть и, как минимум, предоставить необходимые для этого средства отображения. Затем можно поэкспериментировать с альтернативами, для этого следует выбрать какое-то подмножество пользователей, дать им возможность перейти на альтернативное отображение и понаблюдать за реакцией.

Путем анализа типичных запросов и поведения пользователей вы можете принять решения не только о том, как отображать ранжированный список результатов, но и о том, какую информацию показывать, чтобы пользователь быстрее находил то, что ему нужно. Памятуя об этом, проанализируем некоторые способы организации результатов.

На рис. 3.11 показан результат поиска по слову *Apple* в Google. Обратите внимание, что к первому результату Google добавляет ссылки на самые популярные страницы сайта компании Apple, биржевые котировки ее акций, адреса магазинов (даже карту), связанные с ней лица и другие данные. Далее (на рисунке не показаны) расположены ссылки и связанные поисковые запросы.

Еще отметим, что на рис. 3.11 нет ни одного результата о яблоке как фрукте, например, сортах Грэнни Смит или Гала (ниже на странице встречается один, но Google решила отдать большую часть места на экране одноименной компании). Понятно, что, учитывая популярность Apple Inc., она должна иметь высокий ранг, но что, если бы мы смогли разбить результаты на группы взаимосвязанных элементов? Именно такое желание стоит за набирающим популярность фасетным поиском и тесно связанным с ним понятием кластеризации (см. главу 6). Оба эти метода позволяют объединять результаты поиска в группы, основываясь на атрибутах документов. В случае фасетного поиска документы обычно разносятся по заранее выбранным категориям, а в случае кластеризации схожесть документов определяется динамически, исходя из возвращенных результатов. В обоих случаях пользователь может уточнить или ограничить множество отображаемых результатов, перейдя в заведомо непустую категорию.

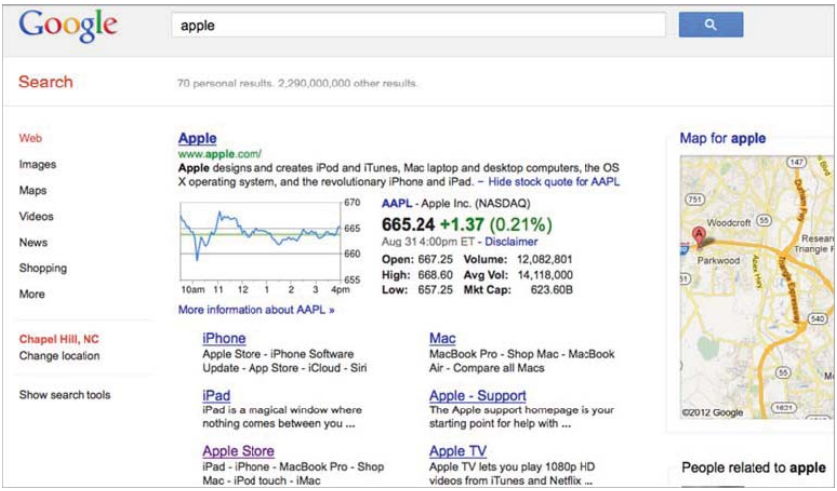

Рис. 3.11. При отображении результатов поиска Google предлагает ряд возможностей в дополнение к простому ранжированному списку

Кластеризация результатов поиска также может улучшить их отображение. Например, на сайте Carrot Search (http://www.carrotsearch.com) предлагается механизм кластеризации результатов, полученных от многих поисковых систем (перейдите по ссылке «Live Demo» на главной странице). На рис. 3.12 показаны результаты выполнения точно такого же запроса по слову Apple. Слева Carrot Search отображает кластеры, выведенные на основе результатов. При таком способе отображения легко сузить поиск определенными кластерами, например, искать «apple pie» (яблочный пирог) вместо Apple Inc., как показано на рис. 3.12. В главе 6 мы узнаем, как воспользоваться механизмом кластеризации Carrot для отображения собственных результатов.

Прежде чем переходить к Apache Solr, скажем, что существует немало отличных книг по информационному поиску, и это не считая сайтов, групп по интересам, целых сообществ и научных статей в академических журналах. Поэтому, если вы ищете какую-то специальную поисковую систему или просто хотите узнать больше, начинайте искать по запросу *information retrieval* (информационный поиск). Кроме того, Ассоциация вычислительной техники (Association for Computing Machinery, ACM) поддерживает великолепную группу по интересам SIGIR, которая ежегодно проводит конференции, где самые блестящие умы в отрасли делятся своими знаниями.

Но, наверное, хватит высокоуровневых концепций, да? Давайте уже посмотрим, как включить в приложение реальную поисковую систему, Apache Solr.

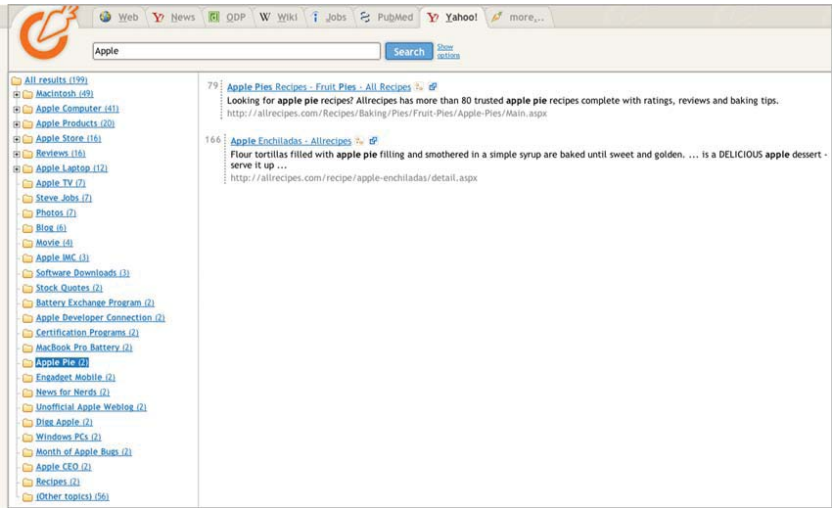

Рис. 3.12. Кластеризация – полезный способ показа результатов в случае, когда поисковые термы неоднозначны. Так, Apple означает и компанию Apple Inc., и фрукт (а также многое другое)

3.3. Введение в поисковый сервер Apache Solr

Apache Solr (http://lucene.apache.org/solr) – это высокопроизводительный, потокобезопасный поисковый сервер промышленного качества, построенный на базе Apache Lucene. Сервер Solr был создан компанией CNET, которая в начале 2006 года безвозмездно передала его фонду Apache Software Foundation. С тех пор он не переставал развиваться – в него добавлено много новых возможностей, а вокруг сформировалось большое и активное сообщество, члены которого разрабатывают новые функции, исправляют ошибки и повышают производительность. Перечислим некоторые функции, которые ставят Solr на одно из первых мест среди поисковых серверов.

- Простые, основанные на HTTP протоколы индексирования и поиска, а также наличие клиентов, написанных на Java, PHP, Ruby и других языках.

- Развитые механизмы кэширования и репликации с целью повышения производительности.
- Простота настройки.
- Фасетный поиск.
- Выделение поисковых термов в найденных результатах.
- Средства администрирования, протоколирования и отладки, позволяющие не гадать, что происходит в работающем сервере Solr.
- Распределенный поиск.
- Проверка орфографии.
- Извлечение содержимого с помощью Apache Tika.
- Качественная документация.

Одна из лучших черт Solr – использование Apache Lucene. Как и вокруг Solr, вокруг Lucene образовалось активное сообщество, и у нее сложилась репутация надежной, как скала, поисковой библиотеки (Solr, напротив, является поисковым сервером), выдающей высококачественные результаты с великолепной производительностью. Apache Lucene, первоначально написанная Дугом Каттингом (Doug Cutting), превратилась в быструю и мощную библиотеку для полнотекстового поиска. Тем, кто хочет узнать о Lucene побольше, рекомендуем книгу Erik Hatcher, Otis Gospodnetić, Mike McCandless «Lucene In Action» (второе издание). В ней прекрасно описано внутреннее устройство Lucene, и многое применимо также к Solr.

А теперь, когда вы знаете о том, что Solr может предложить, займемся настройкой и использованием этого сервера. Имейте в виду, что многое из того, что при работе с Lucene приходится программировать вручную, в Solr задается в конфигурационном файле.

3.3.1. Первый запуск Solr

Полный исходный код Apache Solr вместе с примерами включен в код, сопровождающий эту книгу. Можно также скачать дистрибутив с сайта Solr http://lucene.apache.org/solr, для этого щелкните по ссылке Download на начальной странице и следуйте инструкциям. Для работы Solr необходима версия Java JDK 1.6 или выше. Вместе с сервером поставляется контейнер сервлетов Jetty, но он должен работать с большинством современных контейнеров, в том числе Apache Tomcat. В этой книге мы будем пользоваться версией, сопровождающей книгу, но вы можете взять и более позднюю, если таковая имеется; в таком случае приведенные ниже инструкции по работе, возмож-

но, придется немного модифицировать. Официальная документация приведена на сайте Solr. Скачав дистрибутив, прилагаемый к книге (каталог tamingText-src), запустите демонстрационное приложение, выполнив следующие команды:

1. `cd apache-solr/example`
2. `java -jar start.jar` – эта команда должна запустить контейнер Jetty, развернув внутри него Solr как веб-приложение, прослушивающее порт 8983.
3. Перейдя в браузере по адресу http://localhost:8983/solr/, вы должны увидеть окно, показанное на рис. 3.13. Если приветственный экран не появился, смотрите инструкции по поиску неисправностей на сайте Solr.

Рис. 3.13. Появление приветственного экрана означает, что приложение Solr запустилось правильно

4. В другом окне команд перейдите в каталог `example`, как на шаге 1.
5. В этом окне введите команду `cd exampledocs`.
6. Отправьте Solr тестовые документы, находящиеся в этом каталоге, выполнив Java-приложение post.jar: `java -jar post.jar *.xml`.
7. Зайдите в окно браузера с приветственным экраном Solr и, щелкнув по ссылке, перейдите на экран администрирования Solr, показанный на рис. 3.14.
8. Наберите в поле ввода какой-нибудь запрос, например Solr, нажмите кнопку для его отправки и полюбуйтесь на результаты.

Вот и все, что необходимо для запуска Solr. Но чтобы настроить Solr для конкретного приложения, необходимо подготовить схему Solr (schema.xml) и, возможно, изменить конфигурационный файл Solr (solrconfig.xml). Ниже в этой книге мы остановимся только на некоторых частях этих файлов, а полные примеры конфигурации можно найти в прилагаемом коде в каталоге `apache-solr/example/solr/conf`. Кроме того, на сайте Solr (http://lucene.apache.org/solr) имеется много ресурсов, учебных пособий и статей о том, как настраивать и использовать Solr для решения различных задач.

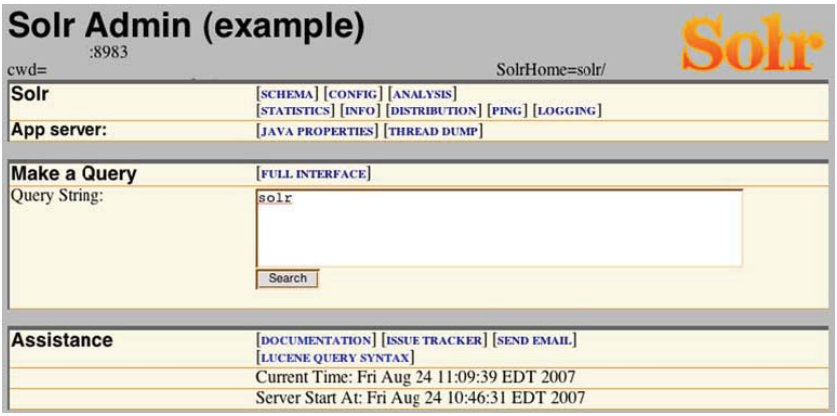

Рис. 3.14. Административный интерфейс Solar

3.3.2. Основные концепции Solr

Поскольку Solr – поисковая служба, размещенная в Интернете, то для выполнения большинства операций клиентское приложение должно посылать серверу Solr HTTP-запросы типа GET и POST. Благодаря такой гибкости с Solr могут работать самые разные приложения, а не только написанные на Java. На самом деле, в дистрибутиве Solr имеются клиенты для Java, Ruby, Python и PHP, а также описание стандартных XML-ответов, которые легко может обработать любое приложение.

Получив запрос от клиента, Solr разбирает URL-адрес и передает запрос подходящему обработчику – объекту `SolrRequestHandler`. Этот объект занимается обработкой параметров запроса, выполнением вычислений и сборкой ответа – объекта `SolrQueryResponse`. После того как ответ создан, объект, реализующий интерфейс `QueryResponseWriter`, сериализует его и отправляет клиенту. Solr поддерживает различные форматы ответа, в том числе XML и JSON, а также такие, которые легко обработать в программах на Ruby или PHP. Если имеющихся форматов не хватает, то можно без труда подключить пользовательские классы, реализующие `QueryResponseWriter`.

В части обработки содержимого Solr разделяет терминологию и средства индексирования и поиска с Lucene. В Solr (и Lucene) индекс состоит из одного или нескольких документов (объектов `Document`). Каждый документ содержит одно или несколько полей (объектов `Field`). Поле состоит из имени, содержимого и метаданных, которые

сообщают Solr/Lucene, как обрабатывать содержимое. Метаданные описаны в табл. 3.3.

Таблица 3.3. Параметры и атрибуты поля Solr

Имя	Описание
`indexed`	По индексированным полям можно проводить поиск и сортировку. Для них можно также запустить процесс анализа Solr с целью модифицировать содержимое для улучшения или изменения результатов.
`stored`	Содержимое хранимых полей сохраняется в индексе. Это полезно для извлечения и выделения содержимого для целей отображения, но для самого поиска такие поля не нужны.
`boost`	Задав повышающий коэффициент, можно увеличить вес данного поля относительно остальных. Например, полю названия обычно назначают больший вес, чем обычному содержимому, потому что совпадение с заголовком часто означает большую релевантность.
`multiValued`	Позволяет несколько раз добавлять в документ одно и то же поле.
`omitNorms`	По существу, подавляет учет длины поля (количество лексем в нем) при оценивании. Применяется, чтобы сэкономить дисковое пространство, когда поле не дает вклад в ранг результата поиска.

Solr несколько более ограничителен, чем Lucene, в том смысле, что накладывает определение схемы (схема пишется на XML и хранится в файле schema.xml) поверх структуры `Field`, что открывает возможность для гораздо более строгой типизации полей за счет объявления объектов `FieldType`. Таким образом, можно указать, что поле является датой, целым числом или обычной строкой, а также сообщить об атрибутах поля. Например, поле типа дата (`dateFieldType`) объявляется в виде:

```
<fieldType name="date" class="solr_DateField"
           sortMissingLast="true" omitNorms="true"/>
```

Solr также позволяет потребовать, чтобы с каждым документом было ассоциировано уникальное поле.

Если поле индексировано, то Solr может подвергнуть его процессу анализа, трансформирующему содержимое поля. Так, Solr предоставляет средства для стемминга, удаления стоп-слов и иных способов видоизменения индексируемых лексем, которые мы обсуждали в разделе 3.2.1. Этот процесс контролируется с помощью класса Lucene

Analyzer. В состав анализатора входят необязательный фильтр символов (CharFilter), обязательный лексический анализатор (Tokenizer), а также нуль или более фильтров лексем (TokenFilter). Объект CharFilter можно использовать для удаления содержимого с сохранением правильной информации о смещении (например, вырезать HTML-теги), что бывает необходимо, например, для подсветки. Отметим, что в большинстве случаев фильтр символов не нужен. Лексический анализатор Tokenizer порождает лексемы (объекты Token), которые как правило соответствуют подлежащим индексированию словам. Далее объект TokenFilter принимает лексемы от лексического анализатора и может дополнительно модифицировать или удалить их до передачи Lucene на индексирование. Например, включенный в Solr анализатор WhitespaceTokenizer разбивает текст на слова по символам-пустышкам, а фильтр StopFilter удаляет часто встречающиеся слова.

В качестве примера обогащенного типа поля FieldType демонстрационная схема, включенная в дистрибутив Solr (apache-solr/example/solr/conf/schema.xml) содержит показанное ниже объявление textFieldType (отметим, что оно эволюционирует, так что ваша версия может отличаться):

```
<fieldType name="text" class="solr.TextField"
           positionIncrementGap="100">
 <analyzer type="index">
   <tokenizer
     class="solr.WhitespaceTokenizerFactory"/>
   <filter class="solr.StopFilterFactory" ignoreCase="true"
           words="stopwords.txt"/>
   <filter class="solr.WordDelimiterFilterFactory"
       generateWordParts="1" generateNumberParts="1" catenateWords="1"
       catenateNumbers="1" catenateAll="0"
       splitOnCaseChange="1"/>
   <filter class="solr.LowerCaseFilterFactory"/>
   <filter class="solr.EnglishPorterFilterFactory"
       protected="protwords.txt"/>
```

Этот анализатор применяется только при индексировании документа, на что указывает атрибут type="index". Если на этапах индексирования и обработки запроса применяется в точности один и тот же подход, то нужно было бы объявить всего один анализатор, а атрибут type опустить

Считать, что лексемы разделены пробелами. *Примечание*: все объекты Tokenizer и TokenFilter обернуты фабриками, которые порождают экземпляр нужного анализатору класса

Удалять часто встречающиеся стоп-слова. Список стоп-слов по умолчанию содержится в файле stopwords.txt в конфигурационном каталоге Solr

Разбивать слова, содержащие буквы разных регистров, цифры и т. д. Например, iPod превращается в iPod, i и Pod

Производить стемминг с помощью стеммера д-ра Мартина Портера. См. http://snowball.tartarus.org

```
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
 </analyzer>
 <analyzer type="query">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.SynonymFilterFactory"
     synonyms="synonyms.txt" ignoreCase="true" expand="true"/>
  <filter class="solr.StopFilterFactory" ignoreCase="true"
     words="stopwords.txt"/>
  <filter class="solr.WordDelimiterFilterFactory"
     generateWordParts="1"
     generateNumberParts="1" catenateWords="0"
     catenateNumbers="0" catenateAll="0"
     splitOnCaseChange="1"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.EnglishPorterFilterFactory"
     protected="protwords.txt"/>
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
 </analyzer>
</fieldType>
```

Этот анализатор применяется только при обработке запроса, на что указывает атрибут type="query". Он отличается от индексного анализатора тем, что добавляет расширение синонимами, но в остальном лексемы выглядят точно так же, что очень важно для сравнения

Расширять любой терм запроса, встречающийся в файле synonyms.txt (находится в конфигурационном каталоге Solr)

Из этого примера должно быть ясно, что приложение может применять различные подходы к анализу – достаточно просто объявить типы и порядок используемых лексических анализаторов и фильтров лексем.

Имея эти два объявления `FieldType`, можно объявить несколько полей, например:

```
<field name="date" type="date" indexed="true" stored="true"
       multiValued="true"/>
<field name="title" type="text" indexed="true"
       stored="true"/>
<field name="generator" type="string" indexed="true"
       stored="true" multiValued="true"/>
<field name="pageCount" type="sint"
       indexed="true" stored="true"/>
```

Индексированное и хранимое поле даты, которое можно использовать для поиска и сортировки документов по дате

Название – текстовое индексированное и хранимое поле. Текстовые поля разбиваются на лексемы и подвергаются анализу с помощью анализатора, объявленного в схеме Solr

Поле generator строковое, Solr индексирует и хранит его точно в том виде, в котором оно было передано

В поле pageCount хранится количество страниц документа, это целочисленное, допускающее сортировку поле, т. е. человеку оно невидимо, но оптимизировано для целей сортировки

Проектирование схемы Solr

Как при проектировании любой системы анализа текста, необходимо тщательно продумать, каким образом пользователь будет получать доступ к поиску. Синтаксис схемы Solr дает богатые возможности: от развитых инструментов лексического анализа и стемминга до средств проверки правописания и настраиваемой сортировки. Все эти возможности описываются в файле schema.xml и конфигурационном файле Solr. Примите за правило, приступая к новому проекту Solr, начинать с примеров схемы и конфигурационного файла, входящих в дистрибутив, и смотреть, что стоит оттуда удалить, а что оставить. Такой подход отлично работает, потому что эти файлы хорошо документированы – все концепции объясняются очень подробно.

В схеме Solr можно выделить три секции:

- объявления типов полей;
- объявления полей;
- прочие объявления.

Объявление типа поля говорит Solr о том, как интерпретировать содержимое поля. Типы, определенные в этих объявлениях, затем можно использовать в секции объявления полей. Но сам факт объявления типа еще не означает, что вы обязаны этот тип использовать. В текущей версии Solr имеется много типов, самые употребительные из них `IntField`, `FloatField`, `StrField`, `DateField` и `TextField`. Помимо них, в приложении можно реализовать собственный тип `FieldType`, расширив тем самым возможности Solr.

В секции объявления полей происходит самое интересное. Здесь приложение точно объявляет, как собирается индексировать и хранить документы внутри Solr: задается имя каждого поля, его тип и прочие метаданные, благодаря которым Solr знает, как производить индексирование и обрабатывать поисковые запросы.

Наконец, в секции «Прочее» находятся разные разности, например, имя поля, содержащего уникальный ключ документа, или поля, в котором производится поиск по умолчанию. Дополнительные объявления сообщают Solr о необходимости скопировать содержимое одного поля в другое. Тиражируя поле, Solr может проанализировать одно и то же содержимое несколькими способами, что расширяет возможности поиска. Например, часто имеет смысл предоставить пользователю возможность искать с учетом регистра. Для этого достаточно создать элемент `<copyField>`, который копирует содержимое одного поля в другое, настроенное так, чтобы регистр сохранялся.

При проектировании схемы часто возникает искушение индексировать все подряд, нужное и ненужное, а затем генерировать гигантские запросы, которые ищут во всех полях. Хотя Solr это позволяет, лучше все же задуматься о том, какие поля нужно хранить и индексировать, а какие не стоит, потому что соответствующая информация есть в другом месте. Зачастую самые простые запросы можно обработать, создав поле «all», включающее содержимое всех остальных поисковых полей. При такой стратегии можно произвести быстрый поиск по содержимому, не создавая запросов с участием нескольких полей. Ну а когда возникает необходимость уточнить поиск, можно подключить и отдельные поля.

В Solr процесс анализа легко настраивается и часто не требует вообще никакого программирования. Только в особых случаях, когда имеющихся в Lucene и Solr готовых классов, реализующих интерфейсы `CharFilter`, `Tokenizer` и `TokenFilter`, недостаточно (а разработчики со всего мира написали множество модулей анализа для Lucene и Solr), возникает необходимость в создании нового кода.

Теперь, разобравшись с основами структурирования содержимого в Solr, мы готовы углубиться в тайны добавления содержимого, по которому впоследствии сможем искать. В следующем разделе описано, как формировать документы и отправлять их в Solr для индексирования. А уже потом мы поговорим о поиске по этому содержимому.

3.4. Индексирование содержимого с помощью Apache Solr

В Solr есть несколько способов индексирования, например: отправка сообщений в формате XML или JSON, CSV-файлов или файлов стандартных офисных типов MIME, выборка данных из базы с помощью команд SQL или из RSS-лент. Здесь мы рассмотрим только индексирование XML-сообщений и файлов стандартных офисных типов, а за сведениями о других возможностях отсылаем читателя к документации по Solr. Точнее, если вам интересно индексирование CSV-файлов, загляните на страницу http://wiki.apache.org/solr/UpdateCSV. А если вы хотите побольше узнать об индексировании данных из базы или из RSS-лент, приглашаем познакомиться с обработчиком импорта данных по адресу http://wiki.apache.org/solr/DataImportHandler.

Прежде чем начинать разговор об индексировании XML, следует отметить, что в Solr есть четыре операции, имеющие отношение к индексированию.

- *Добавление/обновление* – позволяет добавить или обновить документ в Solr. Новые или измененные документы становятся доступны для поиска только после фиксации.
- *Фиксация* – говорит Solr, что все изменения, произведенные после последней фиксации, должны быть сделаны доступными для поиска.
- *Удаление* – позволяет удалить документы по идентификатору или по запросу.
- *Оптимизация* – реорганизует внутренние структуры Lucene для повышения производительности поиска. Если уж производить оптимизацию, то лучше делать это после завершения индексирования. В большинстве случаев об оптимизации можно не думать.

3.4.1. Индексирование данных в формате XML

Один из способов индексирования в Solr состоит в том, чтобы построить XML-сообщение, содержащее предварительно обработанное содержимое, и отправить его по протоколу HTTP в виде запроса типа POST. Такое XML-сообщение может выглядеть примерно следующим образом:

```
<add>
 <doc>
   <field name="id">solr</field>
   <field name="name" boost="1.2">
      Solr, the Enterprise Search Server
   </field>
   <field name="mimeType">text/xml</field>
   <field name="creator">Apache Software Foundation</field>
   <field name="creator">Yonik Seeley</field>
   <field name="description">An enterprise-ready, Lucene-based search
      server. Features include search, faceting, hit highlighting,
      replication and much, much more</field>
 </doc>
</add>
```

Здесь ясно прослеживается простая структура: внешний элемент `<add>` содержит один или несколько элементов `<doc>`. В каждом документе описаны его поля и, возможно, повышающий коэффициент.

Такое сообщение можно отправить методом POST в Solr прямо из любого веб-браузера или иного HTTP-клиента. Дополнительные сведения об использовании XML в Solr см. на странице вики http://wiki.apache.org/solr/UpdateXmlMessages.

По счастью, вместе с Solr поставляется простая клиентская библиотека SolrJ, которая берет на себя всю работу, связанную с конструированием XML-сообщений. В листинге ниже демонстрируется использование SolrJ для добавления документов в Solr.

Листинг 3.1. Пример использования клиентской библиотеки SolrJ

```
SolrServer solr = new CommonsHttpSolrServer(
                  new URL("http://localhost:" + port + "/solr"));

SolrInputDocument doc = new SolrInputDocument();

doc.addField("id", "http://tortoisehare5k.tamingtext.com");

doc.addField("mimeType", "text/plain");

doc.addField("title",
             "Tortoise beats Hare! Hare wants rematch.", 5);
Date now = new Date();

doc.addField("date",
             DateUtil.getThreadLocalDateFormat().format(now));
doc.addField("description", description);

doc.addField("categories_t", "Fairy Tale, Sports");

solr.add(doc);

solr.commit();
```

Создаем соединение с сервером Solr по протоколу HTTP

Схема, заданная для этого экземпляра Solr, подразумевает обязательное наличие уникального поля с именем id

Добавляем в документ поле title, содержащее название, и говорим, что оно в 5 раз важнее всех остальных полей

Даты в Solr должны форматироваться специальным образом

Механизм динамических полей позволяет добавлять в Solr заранее неизвестные поля. Суффикс _t означает, что Solr должен рассматривать это поле как текстовое

Отправляем созданный документ в Solr. Библиотека SolrJ сама сформирует правильное XML-сообщение и отправит его в Solr с помощью библиотеки Apache Jakarta Commons HTTPClient

После того как все документы добавлены и могут быть сделаны доступными для поиска, отправляем Solr команду фиксации

Для индексирования содержимого необходимо отправить Solr команду `add` для каждого документа `SolrInputDocument`. Отметим,

что в одной команде `add` может быть перечислено несколько документов, нужно только воспользоваться перегруженным методом объекта `SolrServer`, который принимает коллекцию (объект типа `Collection`). Это даже рекомендуется для повышения производительности. Если вы думаете, что накладные расходы, связанные с HTTP, негативно отразятся на производительности индексирования, успокойтесь – на практике объем работы по управлению соединениями, как правило, невелик по сравнению с затратами на индексирование.

Вот, собственно, вы и познакомились с основами индексирования данных в формате XML. Теперь рассмотрим индексирование файлов стандартных форматов.

3.4.2. Извлечение и индексирование содержимого с помощью Solr и Apache Tika

Для того чтобы извлечь содержимое из документа и проиндексировать его, необходимо воспользоваться некоторыми идеями, изложенными в этой главе и в главе 1. Самое главное – необходимо спроектировать схему Solr, так чтобы она отражала структуру информации, извлекаемой из содержимого, по которому вы собираетесь искать. Поскольку для извлечения содержимого мы будем использовать библиотеку Tika, то приложение должно установить соответствие между полями схемы и метаданными и содержимым, возвращаемым Tika.

Для просмотра полной схемы Solr откройте файл solr/conf/schema.xml в своем любимом редакторе. К сказанному выше о проектировании схемы можно лишь добавить, что мы старались правильно сопоставить типы Tika и типы полей Solr. Например, счетчик страниц – целое число, поэтому мы сделали его целым в схеме Solr. Для этого и других полей мы итеративно просматривали выборочное множество документов и извлекаемую из них информацию. Чтобы это сделать, мы воспользовались тем фактом, что интеграция Solr с Tika устроена так, что позволяет извлекать содержимое, не индексируя его. С помощью программы `curl` (имеется по умолчанию в большинстве систем на базе *NIX, а версию для Windows можно скачать со страницы http://curl.haxx.se/download.html) мы можем посылать Solr файлы и другие HTTP-запросы. Если вы еще не запустили сервер Solr, сделайте это, как описано в главе 1. После запуска Solr можно индексировать содержимое. В данном случае мы посылаем Solr команду с требованием извлечь содержимое тестового файла:

```
curl "http://localhost:8983/solr/update/extract?&extractOnly=true" \
-F "myfile=@src/test/resources/sample-word.doc"
```

Параметром команды является тестовый документ Word, расположенный в каталоге src/test/resources в коде, прилагаемом к книге, но можно взять любой файл в формате Word или PDF. Чтобы команда `curl` нашла документ, ее нужно запускать, находясь в каталоге на верхнем уровне дерева распакованного кода. В ответ будет получено извлеченное из файла содержимое и метаданные. Выглядит это следующим образом (часть содержимого для краткости опущена):

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
 <lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">8</int>
 </lst>
 <str name="sample-word.doc">&lt;?xml version="1.0"
      encoding="UTF-8"?&gt;
  &lt;html xmlns="http://www.w3.org/1999/xhtml"&gt;
  &lt;head&gt;
  &lt;title&gt;This is a sample word document&lt;/title&gt;
  &lt;/head&gt;
  &lt;body&gt;
  &lt;p&gt;This is a sample word document.&amp;#xd;
  &lt;/p&gt;
  &lt;/body&gt;
  &lt;/html&gt;
 </str>
 <lst name="sample-word.doc_metadata">
  <arr name="Revision-Number">
   <str>1</str>
  </arr>
  <arr name="stream_source_info">
   <str>myfile</str>
  </arr>
  <arr name="Last-Author">
   <str>Grant Ingersoll</str>
  </arr>
  <arr name="Page-Count">
   <str>1</str>
  </arr>
  <arr name="Application-Name">
   <str>Microsoft Word 11.3.5</str>
  </arr>
  <arr name="Author">
   <str>Grant Ingersoll</str>
  </arr>
  <arr name="Edit-Time">
```

```
    <str>600000000</str>
   </arr>
   <arr name="Creation-Date">
    <str>Mon Jul 02 21:50:00 EDT 2007</str>
   </arr>
   <arr name="title">
    <str>This is a sample word document</str>
   </arr>
   <arr name="Content-Type">
    <str>application/msword</str>
   </arr>
   <arr name="Last-Save-Date">
    <str>Mon Jul 02 21:51:00 EDT 2007</str>
   </arr>
  </lst>
 </response>
```

Из полученного результата видно, что возвращает Tika, и это позволяет правильно спланировать схему.

После того как схема определена, можно начинать процесс индексирования. Для этого документы посылаются обработчику `ExtractingRequestHandler`, который предоставляет инфраструктуру, необходимую Tika для извлечения содержимого. Чтобы воспользоваться классом `ExtractingRequestHandler`, необходимо настроить его в конфигурационном файле Solr. В данном случае файл `solrconfig.xml` будет выглядеть следующим образом:

```
<requestHandler name="/update/extract"
    class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
 <lst name="defaults">
  <str name="fmap.Last-Modified">last_modified</str>
  <str name="fmap.Page-Count">pageCount</str>
  <str name="fmap.Author">creator</str>
  <str name="fmap.Creation-Date">created</str>
  <str name="fmap.Last-Save-Date">last_modified</str>
  <str name="fmap.Word-Count">last_modified</str>
  <str name="fmap.Application-Name">generator</str>
  <str name="fmap.Content-Type">mimeType</str>
  <!-- Всему остальному сопоставляется ignored -->
  <bool name="uprefix">ignored_</bool>
 </lst>
</requestHandler>
```

Это файл уже находится в коде, сопровождающем книгу, так что к индексированию содержимого все готово. Вам нужно только послать Solr какие-нибудь документы. Мы снова воспользуемся программой `curl`, но в реальном приложении лучше бы иметь робот-обходчик или какой-нибудь код, получающий документы из хранилища (сис-

темы управления содержимым, базы данных и т. д.) и передающий их Solr с помощью клиентской библиотеки типа SolrJ.

Для демонстрации индексирования с помощью `curl` достаточно модифицировать показанную выше команду извлечения содержимого, убрав параметр `extract.only` и добавив несколько других:

```
curl "http://localhost:8983/solr/update/extract? \
   literal.id=sample-word.doc&defaultField=fullText&commit=true" \
   -F "myfile=@src/test/resources/sample-word.doc"
```

Помимо указания загружаемого файла (флаг `-F`), следует отметить еще два момента:

- файл отправляется на URL-адрес /update/extract, а не просто /update. Для Solr это означает, что мы хотим использовать обработчик `ExtractingRequestHandler`, сконфигурированный выше.
- в данном случае передаются следующие параметры:
  - `literal.id=sample-word.doc`: добавить литеральное значение `sample-word.doc` в качестве поля `id` документа. Иными словами, это уникальный идентификатор.
  - `defaultField=fullText`: если возможно, Solr автоматически пытается сопоставить имена полей в извлеченном содержимом с именами полей Solr. Если совпадений не найдено, то с помощью этого параметра задается имя поля по умолчанию, в которое индексируется содержимое. В данном случае все неопознанное содержимое попадет в поле `fullText`.
  - `commit=true`: это означает, что Solr должен сразу же фиксировать новые документы в индексе, чтобы они стали доступны для поиска.

На примере этой команды видны основные возможности обработчика `ExtractingRequestHandler`. О дополнительных его функциях читайте на странице http://wiki.apache.org/solr/ExtractingRequestHandler.

Выполнение команд индексирования для вашего набора файлов должно дать похожие результаты – разумеется, с поправкой на содержимое ваших каталогов. Имея этот простой индекс, мы теперь можем посмотреть, какие результаты получаются при запросе к нему с помощью встроенного в Solr инструмента администрирования.

> **Совет.** Программа Luke – ваш лучший друг во всем, что касается исследования структуры и содержимого индекса Lucene (и Solr). Эта

программа, написанная Анджеем Бялецки (Andrzej Bialecki), полезна, когда нужно понять, как именно проиндексированы термы и какие документы хранятся в индексе. Она показывает и другие метаданные, например, 50 термов с наибольшей частотой в указанном поле. Программу Luke можно бесплатно скачать по адресу http://code.google.com/p/luke/.

Итак, у нас есть непустой индекс и можно, наконец, пожать плоды своих трудов: посмотреть, как работает поиск в Solr. В следующем разделе мы воспользуемся клиентом SolrJ для отправки поисковых запросов и заодно выясним, как выглядят результаты.

3.5. Поиск по содержимому в Apache Solr

Как и индексирование, поиск производится путем отправки Solr HTTP-запросов, описывающих, что нужно пользователю. В Solr имеется весьма развитый язык запросов, который позволяет задавать термы, фразы, метасимволы и ряд других опций. И количество опций в каждой новой версии только растет. Но не пугайтесь – на сайте Solr (http://lucene.apache.org/solr) все новые функции отлично документированы, так что вы всегда можете найти информацию о самых последних возможностях.

Чтобы понять, как работает поиск в Solr, отступим на шаг назад и посмотрим, как Solr обрабатывает запросы. В разделе 3.3.2 говорилось, что Solr анализирует поступающий запрос и передает его тому или иному объекту, реализующему интерфейс `SolrRequestHandler`. По счастью, в Solr есть много полезных реализаций этого интерфейса, так что писать собственный обработчик вам не придется. В табл. 3.4 перечислены некоторые из наиболее употребительных обработчиков с описанием их функциональности.

Класс `RequestHandler` умеет обрабатывать любые запросы – обобщенным образом, – тогда как за обработку запросов на поиск отвечает производный от него класс `SearchHandler`. Класс `SearchHandler` состоит из одного или нескольких компонентов `SearchComponent`. Эти компоненты вкупе с анализатором запроса делают большую часть работы, связанной с поиском. В составе Solr имеется много реализаций `SearchComponent`, а также несколько разных анализаторов запроса. При желании вы можете добавить и собственный компонент или анализатор. Чтобы лучше разобраться в этом вопросе и в других параметрах Solr, приглядимся к ним пристальнее.

Таблица 3.4. Наиболее употребительные реализации `SolrRequestHandler`

Имя	Описание	Пример запроса
`StandardRequestHandler`	Как нетрудно догадаться, класс `StandardRequestHandler` – реализация `SolrRequestHandler` по умолчанию. Он предоставляет средства для задания термов запроса, полей, в которых нужно искать, количества возвращаемых результатов, необходимость фасетного поиска, подсветки и показа релевантности.	`&q=description%3Awin+OR+description%3Aall &rows=10` Запрос на поиск термов *all* и *win* в поле description. Возвращать не более 10 строк.
`MoreLikeThisHandler`	Возвращает документы, похожие на данный.	`&q=lazy&rows=10&qt=%2Fmlt&qf=title^3+des cription^10`
`LukeRequestHandler`	Этот обработчик назван в честь замечательной программы Luke для исследования индекса Lucene/Solr. Luke обладает простым, но мощным графическим интерфейсом, который позволяет изучить все детали структуры и содержимого существующего индекса. `LukeRequestHandler` во многом повторяет функциональность Luke, возвращая в ответах на запросы сведения об индексе, которые приложение может показать пользователю.	`&show=schema` Возвращает сведения о схеме текущего индекса (имена полей, состояние хранилища и индексирования и т. д.)

3.5.1. Параметры запроса к Solr

Solr предлагает развитый синтаксис для задания входных параметров и способов обработки для получения результата. В табл. 3.5 перечислено семь входных параметров с описаниями и примерами значений. Поскольку Solr постоянно совершенствуется, полный и актуальный список параметров смотрите на официальном сайте.

Таблица 3.5. Общеупотребительные входные параметры Solr

Параметр	Описание	По умолчанию	Поддерживается обработчиками	Пример
q	Сам запрос. Синтаксис зависит от используемого подкласса SolrRequestHandler. Так, StandardRequestHandler поддерживает синтаксис, описанный на странице http://wiki.apache.org/solr/SolrQuerySyntax.	Нет	StandardRequestHandler DisMaxRequestHandler MoreLikeThisHandler SpellCheckerRequestHandler	q=title:rabbit AND description:"Bugs Bunny" q=jobs:java OR programmer
sort	По какому полю сортировать результаты.	score	Большинство подклассов SolrRequestHandler	q=ipod&sort=price desc q=ipod&sort=price desc,date asc
start	Номер начального результата в порции.	0	Большинство подклассов SolrRequestHandler	q=ipod&start=11 q=ipod&start=1001
rows	Количество возвращаемых результатов в одной порции.	10	Большинство подклассов SolrRequestHandler	q=ipod&rows=25
fq	Задает класс FilterQuery, используемый для ограничения результатов. Это позволяет, например, возвращать только документы в определенном диапазоне дат или только такие, в названии которых встречается буква A. Фильтры полезны только при повторном выполнении ранее обработанного запроса с дополнительными ограничениями.	Нет	Большинство подклассов SolrRequestHandler	q=title:ipod&fq=manufacturer:apple

Таблица 3.5. (окончание)

Параметр	Описание	По умолчанию	Поддерживается обработчиками	Пример
`facet`	Требование предоставить фасетную информацию для данного запроса.	Нет	Большинство подклассов `SolrRequestHandler`	`q=ipod&facet=true`
`facet.field`	Поле, по которому предоставлять фасетную информацию. По его содержимому строится множество фасетов.	Нет	Большинство подклассов `SolrRequestHandler`	`q=ipod&facet=true&facet.field=price&facet.field=manufacturer`

Естественно, при наличии столь разнообразных возможностей – а в табл. 3.5 перечислено далеко не все, на сайте описано гораздо больше – возникает трудный вопрос: что включить в приложение? Ключ к ответу – знать своих пользователей и понимать, что они хотят от поиска. Вообще говоря, такие вещи, как подсветка и похожие документы, пусть и очень привлекательные, способны замедлить поиск из-за необходимости дополнительной обработки (особенно если используется то и другое вместе). С другой стороны, подсветка полезна, когда пользователь хочет быстро сориентироваться в контексте найденного совпадения. Далее мы рассмотрим, как обращаться к Solr из программы, потому что именно это является основным способом интеграции Solr с приложением.

> **Примечание.** Обсуждение параметров запроса наверняка оставило вас в недоумении: а что мы получим в обмен на все старания? Точнее, как выглядят результаты и как их обрабатывать? Для этого Solr предоставляет подключаемые обработчики результатов, производные от класса `QueryResponseWriter`. Как и в случае `SolrRequestHandler`, таких подклассов несколько, и какой-нибудь да удовлетворит ваши потребности. Самый общеупотребительный (и подразумеваемый по умолчанию) – `XMLResponseWriter`, который сериализует результаты в формате XML, строит множество фасетов, добавляет подсветку (и выполняет все прочие функции). Среди других реализаций упомянем `JSONResponseWriter`, `PHPResponseWriter`, `PHPSerializedResponseWriter`, `PythonResponseWriter`, `RubyResponseWriter` и `XSLTResponseWriter`. Надеемся, что имена классов не требуют пояснений, а, если это не так, к вашим услугам сайт Solr. Кроме того, если вам нужен интерфейс к унаследованной системе или вы хотите выводить результаты в своем собственном двоичном формате, то не так уж

сложно самостоятельно написать подкласс `QueryResponseWriter`, и в исходном коде Solr нет недостатка в примерах.

Программный доступ к Solr

До сих пор мы описывали различные параметры запросов к Solr, но как насчет реального кода для выполнения поиска?

Solr предлагает также несколько более продвинутые, но все же широко используемые средства формирования запросов. Например, анализатор `DismaxQParser` поддерживает более простой синтаксис запросов, чем `LuceneQParser`, и отдает предпочтение документам, в котором группы встречаются в различных полях. На примере кода в листинге ниже демонстрируется, как вызывать обработчик `RequestHandler`, в котором для разбора запроса применяется анализатор `DismaxQParser`.

Листинг 3.2. Пример кода отправки запроса Solr

```
queryParams.setQuery("lazy");
queryParams.setParam("defType", "dismax");
```

Просим Solr использовать анализатор запросов DisMax (в файле solrconfig.xml он называется dismax)

```
queryParams.set("qf", "title^3 description^10");
```

Анализатор DisMax производит поиск по полям, указанным в параметре qf и соответственно увеличивает вес термов

```
System.out.println("Query: " + queryParams);
response = solr.query(queryParams);
assertTrue("response is null and it shouldn't be", response != null);
documentList = response.getResults();
assertTrue("documentList Size: " + documentList.size() +
  " is not: " + 2, documentList.size() == 2);
```

Еще одна распространенная техника – дать пользователю возможность легко и быстро искать документы, похожие на уже найденный. Этот процесс часто называют «Найти похожие» или «Еще такие же» (More Like This). В Solr встроены средства поиска похожих документов, их только нужно сконфигурировать. Например, в файле solrconfig.xml можно следующим образом задать обработчик запросов `/mlt`:

```
<requestHandler name="/mlt" class="solr_MoreLikeThisHandler">
 <lst name="defaults">
   <str name="mlt.fl">title,name,description,fullText</str>
 </lst>
</requestHandler>
```

Здесь мы говорим, что обработчик `MoreLikeThisHandler` должен использовать для генерации нового запроса поля `title`, `name`, `description` и `fullText`. Если пользователь попросит найти похожие документы, то Solr возьмет указанный документ-образец, проанализирует термы в заданных полях, определит, какие из них наиболее важны, и составит новый запрос. Этот новый запрос будет отправлен серверу, а его результаты возвращены клиенту. Ниже показано, как воспользоваться новым обработчиком запросов.

Листинг 3.3. Пример поиска похожих документов

```
queryParams = new SolrQuery();
queryParams.setQueryType("/mlt");
queryParams.setQuery("description:number");
queryParams.set("mlt.match.offset", "0");
queryParams.setRows(1);
queryParams.set("mlt.fl", "description, title");
response = solr.query(queryParams);
assertTrue("response is null and it shouldn't be", response != null);
SolrDocumentList results =
     (SolrDocumentList) response.getResponse().get("match");
assertTrue("results Size: " + results.size() + " is not: " + 1,
     results.size() == 1);
```

Задать поиск похожих документов

Указать документ, используемый как образец

Указать, по какому полю генерировать запрос

Разумеется, одна из самых популярных функций Solr – встроенная поддержка фасетного поиска, ее мы рассмотрим в следующем разделе.

3.5.2. Построение фасетов по извлеченному содержимому

Выше, в разделе 3.4.2, мы проиндексировали файлы MS Word с помощью Solr и Tika. В этом разделе мы добавили еще немного своих собственных данных, так что у вас могут получиться другие результаты. Но теперь у вас есть более глубокое понимание того, как в реализации поиска используется класс `SolrRequestHandler`, и вы можете воспользоваться простым административным интерфейсом Solr для выполнения различных операций поиска. В предыдущем примере сервер Solr прослушивал порт 8983 на локальной машине. Если в браузере перейти по адресу http://localhost:8983/solr/admin/form.jsp, то должна появиться страница, показанная на рис. 3.15.

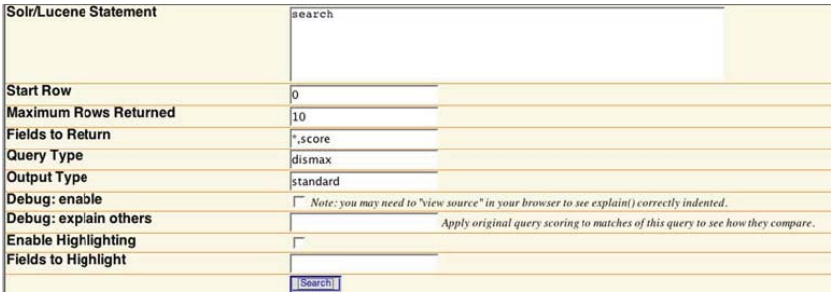

Рис. 3.15. Интерфейс задания запроса к Solr

В этом примере мы воспользовались обработчиком `DisMaxSolrRequestHandler`, который определен в конфигурационном файле solrconfig.xml следующим образом:

```
<requestHandler name="dismax" class="solr_DisMaxRequestHandler" >
 <lst name="defaults">
   <str name="echoParams">explicit</str>
   <float name="tie">0.01</float>
   <str name="qf">
     name title^5.0 description keyword fullText all^0.1
   </str>
   <str name="fl">
     name,title,description,keyword,fullText
   </str>
   <!-- Фасеты -->
   <str name="facet">on</str>
   <str name="facet.mincount">1</str>
   <str name="facet.field">mimeType</str>
   <str name="f.categories.facet.sort">true</str>
   <str name="f.categories.facet.limit">20</str>
   <str name="facet.field">creator</str>
   <str name="q.alt">*:*</str>
   <!-- пример конфигурации подсветки, чтобы разрешить для
         определенного запроса, нужно задать hl=true -->
   <str name="hl.fl">name,title,fullText</str>
   <!-- для этого поля не нужно фрагментирование, только подсветка -->
   <str name="f.name.hl.fragsize">0</str>
   <!-- просим Solr вернуть самое поле, если не найдено ни
         одного поискового терма -->
   <str name="f.name.hl.alternateField">name</str>
   <str name="f.text.hl.fragmenter">regex</str> <!-- определено
                                                    ниже -->
 </lst>
</requestHandler>
```

Чтобы упростить формулирование запросов, мы задали значения по умолчанию для различных параметров обработчика dismax, определяющие, по каким полям искать и что возвращать, а также описали, для каких полей строить фасеты и производить подсветку. Отправив запрос Solr, мы получим в ответ XML-документ, содержащий результаты, а также информацию о фасетных категориях.

Листинг 3.4. Пример результатов поиска, возвращенных Solr

```
<response>

<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">4</int>
 <lst name="params">
   <str name="explainOther"/>
   <str name="fl">*,score</str>
   <str name="indent">on</str>
   <str name="start">0</str>
   <str name="q">search</str>
   <str name="hl.fl"/>
   <str name="wt">standard</str>
   <str name="qt">dismax</str>
   <str name="version">2.2</str>
   <str name="rows">10</str>
   </lst>
</lst>
<result name="response" numFound="2" start="0"
        maxScore="0.12060823">
 <doc>
   <float name="score">0.12060823</float>
   <arr name="creator">...</arr>
   <arr name="creatorText">...</arr>
   <str name="description">An enterprise-ready, Lucene-based
         search server. Features include search,
         faceting, hit highlighting, replication and much,
         much more</str>
   <str name="id">solr</str>
   <str name="mimeType">text/xml</str>
   <str name="name">Solr, the Enterprise Search Server</str>
 </doc>

 <doc>
   <float name="score">0.034772884</float>
   <arr name="creator">...</arr>
   <arr name="creatorText">...</arr>
   <str name="description">A Java-based search engine library
         focused on high-performance, quality results.</str>
```

В секции <responseHeader> возвращаются метаданные, описывающие входные параметры и результаты поиска

В секции <result> приведена информация о документах, найденных с учетом запроса, конфигурации и входных параметров

```
    <str name="id">lucene</str>
    <str name="mimeType">text/xml</str>
    <str name="name">Lucene</str>
  </doc>
 </result>
 <lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="mimeType">
      <int name="text/xml">2</int>
    </lst>
    <lst name="creator">
      <int name="Apache Software Foundation">2</int>
      <int name="Doug Cutting">1</int>
      <int name="Yonik Seeley">1</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
 </lst>
 </response>
```

В списке facet_fields приведены сведения о фасетах, выведенных из результатов поиска. В данном примере фасет mimeType говорит, что четыре из девяти результатов имели тип MIME image/tiff, еще четыре – тип text/plain, а один – тип image/png

В листинге 3.4 приведен сокращенный (часть данных о полях намеренно опущена ради экономии места) список результатов для тестового запроса, в состав которого входит также информация о фасетах. Хотя мы задали в конфигурации dismax многочисленные значения по умолчанию, в параметрах запроса в URL эти значения можно переопределить: возвращать больше результатов в одной порции, подсвечивать найденные термы или искать по другим полям.

Сказанного достаточно для начала работы с Solr. На сайте Solr вы найдете дополнительные сведения по всем рассмотренным, а также многим другим вопросам, например, о кэшировании, репликации и администрировании. Мы же вернемся назад и поговорим о некоторых проблемах производительности – как общих, так и характерных только для Solr, и посмотрим, как они могут повлиять на вашу реализацию поиска.

3.6. Факторы, влияющие на производительность поиска

На самом верхнем уровне производительность поиска – это мера того, насколько хорошо поисковая система порождает результаты. В ней можно выделить две стороны: количественную и качественную. Количественная производительность характеризует быстроту возврата результатов (сколько результатов система может вернуть

в течение заданного времени), а качественная – релевантность результатов запросу. Часто (но не всегда) это два разнонаправленных фактора, что заставляет разработчиков постоянно искать компромиссы между повышением быстродействия и улучшением качества результатов. В этом разделе мы рассмотрим многие рекомендации и приемы, которые используются в современных поисковых системах для повышения качественных и количественных показателей. Затем мы остановимся на некоторых специфичных для Solr способах улучшения производительности. Но прежде скажем несколько слов о том, как оценивать быстродействие и релевантность, поскольку без этого вы не будете знать, принесли ваши усилия успех или нет.

3.6.1. Оценка качественных показателей

Для пользователя поисковой системы нет большего разочарования, чем получить в ответ на свой запрос список результатов, имеющих лишь отдаленное отношение к делу, а то и вообще никакого. Вслед за этим почти всегда начинаются попытки добиться желаемого за счет добавления или удаления каких-то ключевых слов с последующим просмотром первых 10 результатов. Часто кажется, что стоит лишь подобрать нужную комбинацию ключевых слов – и вот оно, счастье! А иногда создается впечатление, что лучше плюнуть на все это.

С другой стороны, производители поисковых систем неустанно ищут компромиссы между качеством результатов, скоростью и простой использования. Поскольку запрос по самой своей природе – это неполное выражение информационных потребностей пользователя, то в поисковых системах применяют сложные алгоритмы, пытающиеся заполнить разрыв между неполнотой запроса и истинным желанием пользователя.

Между пользователями и производителями находится нечеткое нечто, именуемое *релевантностью*. Релевантность – это мера соответствия множества результатов запросу пользователя. А нечеткой мы называем ее, потому что нет двух пользователей, которые бы во всех случаях согласились, что найдено именно то, что нужно. И хотя любое суждение о релевантности субъективно, многие исследователи пытались выработать систематический подход к ее определению. Некоторые даже дошли до того, что организовали конференции, предложив стандартные наборы документов, запросы и инструменты оценивания. Участники прогоняют запросы на данном наборе и отправляют свои результаты в конференцию, где результаты обобщаются, оцениваются, а группы ранжируются по достижениям. Прадедушка всех та-

ких конференций – *TREC* (Text REtrieval Conference – конференция по текстовому поиску) проводится ежегодно под патронажем Национального института стандартов и технологии США (NIST).

В основе многих таких конференций лежат две метрики, позволяющие решить, какая поисковая система дает лучшие результаты. Первая – *точность* – измеряет количество релевантных документов в списке возвращенных системой результатов. Часто точность рассматривается в контексте количества возвращенных результатов. Например, точность при 10 (обозначается P@10) – это количество релевантных документов в списке первых десяти результатов. Поскольку большинство пользователей смотрят только на первые 10 результатов или на первую страницу, то наиболее полезна именно точность при 10. Вторая метрика – *полнота* – это отношение числа найденных релевантных документов к общему числу релевантных документов в наборе. Отметим, что для достижения идеальной полноты нужно было бы возвращать все документы набора для каждого запроса, как бы глупо это ни звучало. Во многих ситуациях можно найти компромисс между точностью и полнотой. Например, нередко точность можно повысить, потребовав, чтобы в документе присутствовали многие, а то и все термы запроса. Но если набор документов невелик, то это означало бы, что в ответ на некоторые запросы будет возвращаться пустое множество результатов. Аналогично полноту можно увеличить, добавляя в запрос все известные синонимы для каждого поискового терма. К сожалению, из-за неоднозначности многих слов в состав результатов могут попасть документы с различными значениями одного и того же слова, что снижает точность.

Так как все-таки оценить качество своей системы? Прежде всего, помните, что у любого метода есть сильные и слабые стороны. Почти наверняка вам придется применить несколько методов. В табл. 3.6 приведен перечень некоторых методов оценивания; несомненно, для конкретных систем возможны и другие методы.

Несистематический метод, метод фокус-группы, TREC и – в меньшей степени – анализ журналов и A/B-тестирование подвержены одному и тому же риску – получить систему, оптимизированную для тестовых данных, но работающую не столь хорошо в условиях реальной эксплуатации. Например, если вы пользуетесь документами, запросами и оценками (называемыми *суждениями о релевантности*) TREC, то, возможно, ваша система будет прекрасно работать с запросами в духе TREC (а они обладают определенной спецификой), но куда хуже – с теми запросами, которые будут предъявлять реальные пользователи.

Таблица 3.6. Распространенные методы оценивания

Метод	Описание	Затраты	Компромиссы
Несистематический	Разработчики, ОТК и другие заинтересованные стороны неформально оценивают систему и сообщают, что работает, а что нет.	Сначала низкие, но в долгосрочной перспективе могут резко возрасти.	Не поддаются формальному воспроизведению, если не ведется журнал запросов. Сложно понять, как изменения влияют на другие части системы. Охват может оказаться слишком узким и потому неэффективным. Есть риск, что оптимизация будет производиться для очень специфического множества запросов. Как минимум, все тестировщики должны пользоваться одним и тем же набором документов.
Фокус-группа	Группу пользователей приглашают поработать с системой. Ведутся журналы запросов, выбранных документов и т. д. Пользователей могут попросить явно указать релевантные и нерелевантные документы. Для принятия решений о качества поиска используется собранная статистика.	Зависит от числа участников и стоимости подготовки системы оценивания.	Может быть полезно в зависимости от числа участников. Может также служить для сбора мнений об удобстве использования. Если сохранять журналы, то можно организовать воспроизводимые тесты.
TREC и другие оценочные конференции	В TREC имеется несколько групп для оценивания различных информационно-поисковых систем, в том числе ориентированных на веб, блоги и юридические документы.	Получение данных платное; формальное участие (отправка результатов) может оказаться затратным по времени и финансам. Вопросы и оценки бесплатны, при наличии данных могут запускаться в автономном режиме по мере необходимости.	Хороши для сравнения с другими системами и получения общего представления о качестве, но данные могут оказаться нерепрезентативными для вашего набора, а потому и результаты будет не слишком полезны.

Таблица 3.6. (продолжение)

Метод	Описание	Затраты	Компромиссы
Анализ журнала запросов	Журналы записываются на производственной системе, из них выбираются 50 наиболее частых запросов и 10-20 случайно выбранных. Затем аналитики оценивают 5 или 10 первых результатов для каждого запроса, присваивая оценку: релевантно, частично релевантно, совсем не релевантно, никуда не годно. Результаты коррелируются и анализируются на предмет неоптимальных запросов. Цель состоит в том, чтобы максимизировать число релевантных результатов, минимизировать число нерелевантных и полностью устранить никуда не годные. Кроме того, анализируются операции поиска, не вернувшие ни одного результата на предмет возможных улучшений. В ходе дополнительного анализа можно посмотреть, какие результаты пользователи выбирали (анализ переходов по ссылкам) и сколько времени пользователь находился на каждой странице. Предполагается, что чем дольше пользователь задерживается на изучении результатов, тем они релевантнее его потребностям.	Зависит от размера журналов и количества использованных запросов.	Оптимально для тестирования ваших данных пользователями. Необходимо тщательно планировать, что помещать в журнал и как и когда забирать журнал для анализа. Лучше делать на этапе бета-тестирования и постоянно в производственной системе. На ранних этапах процесса есть риск получить результаты низкого качества.

Таблица 3.6. (окончание)

Метод	Описание	Затраты	Компромиссы
A/B-тестирование	Аналогично анализу журналов и тестированию фокус-группой, но разным пользователям могут предъявляться разные результаты. Например, 80 % пользователей будут получать результаты, основанные на одном подходе, а 20 % – на другом. Производится анализ и сравнение журналов, чтобы понять, какой подход предпочитали пользователи из разных групп, задававшие одинаковые запросы. Этот метод можно применять к любой части системы: от индексирования до отображения результатов. Нужно только четко документировать различия между группами A и B.	Требует развертывания и поддержки двух производственных систем. Кроме того, зависит от числа запросов и размера журналов.	Хорошо приспособлен для тестирования силами реальных пользователей. Лучше проводить при умеренной нагрузке в ограниченный период времени. Иными словами, Интернету-магазину не стоит запускать такое тестирование за три недели до Рождества!

С практической точки зрения, для большинства приложений, рассчитанных на широкую аудиторию, необходимо провести как минимум несистематическое тестирование и анализ журнала запросов. При наличии финансирования фокус-группы и оценки в стиле TREC могут дать дополнительные данные для анализа. Анализ журналов и A/B-тестирование дают наиболее надежные и полезные результаты, именно их предпочитают авторы.

Естественно, настройку релевантности не следует рассматривать как однократную задачу – сделал и забыл. Но и зацикливаться на качестве одного какого-то результата, если только это не один из самых часто задаваемых вопросов, тоже не стоит. Не нужно также слишком долго размышлять, почему вдруг этот результат оказался на четвертом месте, а тот – на пятом. Это имеет смысл разве что в случае, когда какой-то заказчик заплатил за то, чтобы определенный документ находился на определенном месте. В конце концов, если какой-то результат должен быть на первом месте для некоторого запроса, то за-

шейте это прямо в код – и дело с концом. Пытаться настроить систему так, чтобы этот документ оказался первым естественным образом, без каких бы то ни было уловок, – только искать приключений на свою голову; к тому же, при этом другие операции поиска, скорее всего, будут работать криво. Тот, кто знает, когда брать кувалду, а когда отвертку, уже наполовину выиграл битву за довольного пользователя.

3.6.2. Оценка количественных показателей

Есть немало метрик для оценки того, насколько хорошо система работает с точки зрения количественных показателей. Перечислим наиболее полезные из них.

- *Пропускная способность запросов* – сколько запросов система может обработать в единицу времени. Обычно измеряется в запросах в секунду (QPS). Чем выше, тем лучше.
- *Популярность и время обработки* – графически показывает среднее время, затрачиваемое на обработку одного запроса, в зависимости от частоты этого запроса. Это особенно полезно, когда нужно узнать, на что направить усилия по совершенствованию системы. Например, если самый популярный запрос одновременно и самый медленный, то определенно имеется проблема в системе, но если долго работают только редко задаваемые запросы, то тратить время на детальное исследование проблемы не имеет смысла.
- *Среднее время запроса* – сколько времени в среднем обрабатывается запрос? Сюда же относится статистика распределения запросов по времени обработки. Чем меньше, тем лучше.
- *Статистика кэширования* – во многих системах результаты и документы кэшируются, и полезно знать, как часто имеют место попадания и промахи кэша. Если число попаданий устойчиво мало, то, быть может, система будет работать быстрее, если вообще отключить кэширование.
- *Размер индекса* – измеряет эффективность алгоритма сжатия индекса. Иногда индекс можно сжать до 20 % от первоначального размера. Чем меньше, тем лучше, но диски нынче дешевы, так что особо не усердствуйте.
- *Пропускная способность индексирования* – количество документов, индексируемых в единицу времени. Обычно измеряется в документах в секунду (DPS). Чем выше, тем лучше.

- *Число документов в индексе* – если размер индекса увеличивается сверх меры, то индекс имеет смысл сделать распределенным.
- *Число уникальных термов* – метрика очень высокого уровня, дающая поверхностное представление о размере индекса.

При оценке количественных показателей работы системы следует также учитывать стандартные характеристики: потребляемое процессорное время, оперативную память, дисковое пространство и ресурсы ввода/вывода. Главное, что эти метрики следует отслеживать во времени. Многие системы (в том числе Solr) предоставляют административные средства, которые позволяют легко следить за тем, что происходит.

Получив представление о том, как работает система, мы можем перейти к вопросу о различных приемах, позволяющих улучшить ее производительность. В следующем разделе мы рассмотрим как аппаратные возможности, так и компромиссы между действиями на этапе поиска и индексирования.

3.7. Повышение производительности поиска

С первых дней появлений поисковых систем исследователи и практики занимались их оптимизацией, имея в виду различные цели. Одним нужна большая релевантность, другим – лучшее сжатие, третьим – максимальная пропускная способность запросов. Сегодня мы хотим иметь все это и еще кое-что, но как этого достичь? Надеемся, что, прочитав следующие разделы, вы получите целый спектр опций, о которых стоит поразмыслить – и испытать на практике.

Но сначала предупреждение: настройка поисковой системы может занять очень много времени и не дать видимого улучшения. Перед тем как что-то делать, рекомендуем проверить и перепроверить, что усовершенствования действительно нужны, а для этого вести мониторинг качественных и количественных показателей работы. Кроме того, приведенные ниже советы годятся не для всех ситуаций. К некоторым поисковым системам они вообще неприменимы или требуют глубокого знания внутреннего устройства системы. Наконец, прежде чем тратить время на настройку, убедитесь, что проблема действительно в поисковой системе, а не в приложении.

Ну а теперь, когда все метрики на столе и все предупреждения высказаны, посмотрим, что можно сделать для повышения произво-

дительности поиска. Некоторые проблемы возникают на этапе индексирования, другие – на этапе собственно поиска. Одни проблемы затрагивают только качественные или только количественные показатели, другие – то и другое одновременно. Начнем с оборудования, а потом перейдем к программным решениям таким, как улучшение анализа и написание более качественных запросов.

3.7.1. Совершенствование на уровне оборудования

Одно из самых простых и рентабельных решений, годящееся для всех поисковых систем, – модернизация оборудования. Как правило, поисковые системы весьма положительно реагируют на добавление оперативной памяти и обожают, когда есть много процессорных ресурсов. Кроме того, если система велика и не помещается целиком в памяти, то улучшение подсистемы ввода-вывода воздастся сторицей. Если речь идет о запросах, то особый интерес представляют твердотельные накопители (SSD), у которых время подвода головки гораздо ниже, зато время записи может быть больше.

Но наращивание мощности одной машины имеет пределы. Рано или поздно – когда именно, зависит от объема данных, оборудования и количества пользователей – наступает момент, когда рабочую нагрузку необходимо распределить между несколькими машинами. Вообще говоря, это можно сделать двумя способами.

1. *Репликация* – индекс, помещающийся на одной машине, копируется на несколько машин, между которыми осуществляется балансировка нагрузки. Так часто делают в Интернет-магазинах, когда индекс не особенно велик, зато количество запросов очень значительно.
2. *Распределение/секционирование* – один индекс распределяется между несколькими узлами, или секциями на основе хеш-кода или какого-то другого механизма. Главный узел посылает входящие запросы каждой секции, а затем собирает результаты.

В качестве альтернативы расщеплению индекса по хеш-коду часто можно логически разделить индекс и запросы по нескольким узлам. Например, для многоязычного поиска бывает полезно (правда, не всегда) строить по одному индексу на каждый язык, так что один узел будет обслуживать англоязычные запросы и документы, а другой – испаноязычные.

Разумеется, установка более совершенного оборудования может дать выигрыш, но только в плане ускорения, с точки зрения повышения релевантности так ничего не добиться. Имеет смысл познакомиться с некоторыми проверенными временем рекомендациями и приемами ускорения и повышения качества поиска, пусть даже этот путь и труднее.

3.7.2. Повышение качества анализа

Все поисковые системы, как с открытым, так и с закрытым исходным кодом, должны иметь механизм преобразования входного текста в набор индексируемых лексем. Например, в Solr это делается в процессе `Analyzer`, который разбивает входной поток `InputStream` на начальное множество лексем, которое затем модифицируется (хотя это и не обязательно). Именно на этапе анализа зачастую определяется качество индексирования с точки зрения скорости и релевантности поиска. В табл. 3.7 еще раз перечислены некоторые распространенные методы анализа, описанные выше в этой главе, а также ряд новых, и добавлены примечания о том, как они могут повысить, а иногда и понизить производительность.

Таблица 3.7. Распространенные методы анализа с целью повышения производительности

Метод	Описание	Преимущества	Недостатки
Удаление стоп-слов	Перед индексированием удаляет такие расхожие слова, как *the*, *and* и *a*, чтобы уменьшить размер индекса.	Ускорение индексирования, уменьшение размера индекса.	Теряется информация. Лучше проиндексировать стоп-слова и обрабатывать их на этапе запроса. Стоп-слова бывают полезны при обработке фразовых запросов.
Стемминг	Стеммер анализирует лексемы и, возможно, приводит их к начальной форме. Например, слово *banks* приводится к *bank*.	Повышает полноту.	Теряется информация, что может воспрепятствовать поиску по точному совпадению. Решение: завести два поля с лексемами до и после стемминга.

Таблица 3.7. (окончание)

Метод	Описание	Преимущества	Недостатки
Расширение синонимами	Для каждой лексемы добавляется 0 или более синонимов. Обычно производится на этапе обработки запроса.	Повышает полноту за счет нахождения документов, которые не содержат поисковые термы, но тем не менее релевантны запросу.	Из-за неоднозначности синонимов могут быть найдены документы, не относящиеся к делу.
Приведение к нижнему регистру	Все лексемы преобразуются в нижний регистр.	Пользователи часто не обращают внимания на регистр при вводе запросов; перевод в нижний регистр на этапах индексирования и обработки запроса позволяет найти больше документов.	Препятствует поиску совпадений с учетом регистра. Во многих системах хранится два поля: для точного и неточного поиска.
Внешние знания в качестве полезной нагрузки	Производится обращение в внешнему источнику, который дает информацию о важности лексемы. Эта информация затем кодируется в индексе как полезная нагрузка, ассоциированная с лексемой. Примеры: насыщенность шрифта, анализ ссылок, частеречная разметка.	Обычно позволяет придать дополнительный смысл конкретной лексеме и тем повысить качество поиска. Например, анализ ссылок – основа алгоритма Google PageRank (Brin 1998).	Может существенно замедлить процесс анализа и увеличить размер индекса.

Еще один полезный, но реже используемый метод, способный найти результаты в трудных ситуациях, – это *анализ n-грамм*, представляющий собой вариант моделирования последовательности; мы обсуждали его выше. *N-граммой* называется последовательность из одного или нескольких соседних символов лексемы. Например, в слове *example* можно выделить следующие символьные 1-граммы (униграммы): *e*, *x*, *a*, *m*, *p*, *l*, *e* и такие биграммы: *ex*, *xa*, *am*, *mp*, *pl*, *le*. Аналогично лексемные *n*-граммы порождают псевдофразы. Например, во фразе *President of the United States* есть такие биграммы: *President of*, *of the*, *the United*, *United States*. В чем польза *n*-грамм? Они удобны, когда требуется найти приблизительное совпадение или когда данные пло-

хого качества, как, например, после оптического распознавания. В Lucene компонент проверки правописания пользуется *n*-граммами для порождения потенциальных соответствий, которые затем ранжируются. При поиске в текстах на китайском и аналогичных языках, когда алгоритмически трудно выделить лексемы, *n*-граммы применяются для создания множественных лексем. При таком подходе не требуется никаких знаний о языке (Nie 2000). *N*-граммы, состоящие из слов, полезны при поиске с учетом стоп-слов. Например, предположим, что документ содержит такие два предложения:

- John Doe is the new Elbonian president (Джон Доу – новый элбонский президент).
- The United States has sent an ambassador to the country (Соединенные Штаты направили посла в страну).

После анализа с удалением стоп-слов этот документ мог бы быть преобразован в такую последовательность лексем: *john*, *doe*, *new*, *elbonian*, *president*, *united*, *states*, *sent*, *ambassador*, *country*. Если затем поступит запрос *President of the United States*, то в результате анализа он будет преобразован в форму *president united states*; при обработке этого запроса документ, касающийся элбонского президента, будет найден, потому что в нем встречаются все три лексемы. Но если на этапе индексирования стоп-слова сохраняются, но используются только для порождения фразовых *n*-грамм при обработке запроса, то можно снизить вероятность ложного совпадения путем генерации запроса вида *President of the* и *the United States* или какого-то другого, зависящего от типа порожденных *n*-грамм. Тогда шансы найти только предложения, содержащие точную последовательность слов, повышаются. В некоторых случаях полезно порождать *n*-граммы разной длины. Если в том же примере можно выделять *n*-граммы вплоть до длины 5, то можно было бы найти точное соответствие фразе *President of the United States*. Разумеется, эта техника не идеальна и имеет собственные проблемы, но все же она позволяет воспользоваться информацией, содержащейся в стоп-словах, которая иначе была бы потеряна.

Помимо методов, описанных в табл. 3.7, и *n*-грамм, у каждого приложения могут быть свои специфические потребности. И тут мы воочию видим одно из основных достоинств программного обеспечения с открытым исходным кодом: код всегда можно дополнить. Помните, что чем полнее анализ, тем медленнее индексирование. Поскольку индексирование производится не в реальном времени, возможно, имеет смысл усложнить анализ, если таким образом удается достичь измеримого выигрыша. Однако общее правило – начинать с простого

и добавлять новые функции, только если они решают какую-то конкретную проблему.

Покончив с анализом, мы теперь можем перейти к вопросу о повышении качества обработки запросов.

3.7.3. Повышение качества обработки запросов

Что касается поиска, то есть много способов повысить его скорость и точность. В большинстве случаев трудность получения хороших результатов обусловлена недостаточно полно выраженными информационными потребностями пользователя. С этим ничего не поделаешь. И простой интерфейс Google, поощряющий запросы с одним-двумя ключевыми словами, высоко поднял планку не только для поисковых систем масштаба веб, но и для менее крупных. И если большие системы поиска в Интернете располагают теми же исходными данными, что и Google, то системы поменьше обычно лишены доступа к объемным журналам запросов, к внутренним структурам документов типа HTML-ссылок и к другим механизмам обратной связи, которые могут дать ценную информацию о том, что важно пользователю. Прежде чем тратить время на построение чего-то очень сложного, попробуйте обратить внимание на два ключевых фактора, которые могут улучшить качество результатов.

1. *Обучение пользователей* – иногда пользователям нужно показать, насколько лучших результатов можно добиться, если следовать нескольким простым рекомендациям, касающимся синтаксиса: фразовый поиск и т. п.
2. *Внешние знания* – есть ли в одном документе нечто такое, что делает его более важным, чем другой? Быть может, он написан генеральным директором или 99 из 100 человек пометили его как полезный или прибыльность описанного в нем продукта в пять раз больше, чем для всех сравнимых с ним. Что бы это ни было, подумайте, как включить эту информацию в систему и сделать ее доступной при поиске. Если система этого не позволяет, значит, настало время сменить систему!

Помимо обучения пользователей и использования априорных знаний о документах, есть много других вещей, которые позволяют повысить скорость и точность поиска. Прежде всего, в большинстве ситуаций поисковые термы следует объединять связкой AND, а не OR. Например, если пользователь ввел запрос `Jumping Jack Flash`, то в

предположении, что вы не пытаетесь искать фразу целиком, его следует преобразовать к виду *Jumping AND Jack AND Flash*, а не *Jumping OR Jack OR Flash*. Наличие AND означает, что должны найтись все термы. Это почти наверняка повысит точность, но может снизить полноту. К тому же, такой запрос будет обрабатываться быстрее, т. к. уменьшается количество подлежащих оцениванию документов. При использовании AND может оказаться, что не найдено ничего, но тогда можно обратиться к резервному варианту с OR, если в этом есть необходимость. Пожалуй, единственный случай, когда использование AND может дать недостаточно полезных результатов для простых запросов, – очень малый размер набора (грубо говоря, менее 200 000 документов).

> **Примечание.** Мы не хотим сказать, что все поисковые системы подразумевают именно связку AND, но именно так поступает Solr, и мы решили придерживаться этого соглашения ради простоты объяснений.

Еще одна полезная техника, способная дать большой выигрыш в точности, – распознавать фразы или автоматически индуцировать их с помощью *n*-грамм, состоящих из лексем. В первом случае запрос анализируется на предмет обнаружения в нем фраз, и, если таковые найдены, то преобразуется во внутренний системный формат фразового запроса. Например, если пользователь ввел запрос `Wayne Gretzky career stats` (Статистика карьеры Уэйна Грецки), то хороший распознаватель фраз поймет, что Wayne Gretzky – фраза, и сгенерирует запрос `"Wayne Gretzky" career stats` или даже `"Wayne Gretzky career stats"`. Во многих поисковых системах поддерживается также коэффициент *гибкости* (slop factor) при задании фраз. Он говорит, как далеко могут отстоять друг от друга слова (сколько других слов может быть между ними), не переставая при этом считаться фразой. Часто этот механизм дополняется другим: чем слова ближе, тем выше оценка.

Что касается скорости выполнения запроса, то чем меньше в нем поисковых термов, тем обычно быстрее завершается поиск. Кроме того, часто встречающиеся слова замедляют обработку, потому что системе приходится оценивать большое количество документов, чтобы отобрать релевантные. Поэтому на этапе выполнения запроса имеет смысл убирать из него стоп-слова. Очевидно, что если бы пользователи избегали неоднозначных и часто встречающихся слов, то запросы обрабатывались бы быстрее, но рассчитывать на это не

приходится, если только аудитория не состоит преимущественно из опытных пользователей.

Наконец, для повышения качества (но не скорости) поиска часто применяется техника *обратной связи по релевантности*, когда на один запрос пользователя посылается по меньшей мере два запроса системе. Какие-то результаты первого запроса вручную или автоматически помечаются как релевантные, а затем важные термы из них используются для формирования нового запроса. В случае ручной формы обратной связи пользователь каким-то образом (помечая флажком или щелкая по ссылке) показывает, какие документы релевантны, после чего системе автоматически посылается новый запрос. В случае автоматической обратной связи релевантными считаются первые 5 или 10 документов, на основе которых и формируется новый запрос. В обоих случаях имеется также возможность пометить нерелевантные документы, тогда встречающиеся в них термы либо удаляются из запроса, либо получают меньший вес. Нередко термам нового запроса, перешедшим из старого, присваивается другой вес, причем с помощью дополнительно задаваемых параметров можно регулировать, какой вес присвоить термам из исходного запроса, а какой – вновь добавленным. Скажем, можно сделать так, чтобы новые термы весили вдвое больше исходных. Рассмотрим на простом примере, как может выглядеть процесс обратной связи. Предположим, что в наборе имеются четыре документа, описанных в табл. 3.8.

Таблица 3.8. Пример набора документов

Идентификатор документа	Термы
0	minnesota, vikings, dome, football, sports, minneapolis, st. paul
1	dome, twins, baseball, sports
2	gophers, football, university
3	wild, st. paul, hockey, sports

Предположим также, что пользователь хочет узнать, в какие спортивные игры играют в штате Миннесота, и для этого вводит запрос `minnesota AND sports`. Вполне разумный запрос, и тем не менее в этом, казалось бы, тривиальном примере возвращается только документ 0. Если же воспользоваться автоматической обратной связью по релевантности и для расширения запроса взять первый результат, то система создаст новый запрос `(minnesota AND sports) OR (vikings OR dome OR football OR minneapolis OR st. paul)*2`. И уже этот запрос вернет все документы (как удобно!). Понятно, что обратная связь по релевантности редко работает настолько хорошо,

но обычно все же помогает, особенно если пользователь готов высказывать свои суждения. Подробнее об обратной связи по релевантности можно прочитать в книгах «Modern Information Retrieval» (Baeza-Yates [2011]) или «Information Retrieval: Algorithms and Heuristics» (Grossman [1998]). А мы пойдем дальше и рассмотрим альтернативные модели оценивания, чтобы получить представление о других подходах к поиску.

3.7.4. Альтернативные модели оценивания

Выше мы рассматривали векторную модель оценивания (VSM), а точнее, модель, принятую в Lucene, но было бы непростительным упущением не сказать, что существует много других способов оценивания в рамках модели VSM, равно как и совершенно других моделей, некоторые из которых перечислены в табл. 3.9. Эти альтернативы по большей части реализованы в исследовательских системах или защищены патентами, так что для пользователей ПО с открытым исходным кодом недоступны или непрактичны. Но кое-какие реализованы под капотом Lucene и могут настраиваться в Solr. На самом деле, большая часть средств оценивания Lucene будут подключаемыми модулями в версии Lucene и Solr 4.0 (или уже являются таковыми – в зависимости от того, когда вы читаете эту книгу). Поскольку мы работаем с более ранней версией, то эти модели рассматривать не будем. Но один из способов повысить качество обработки запросов – перейти на другую модель оценивания.

Таблица 3.9. Альтернативные методы и модели оценивания

Название	Описание
Языковое моделирование	Альтернативная вероятностная модель, которая ставит проблематику информационного поиска с ног на голову. Что-то вроде телевизионной «Своей игры», только дается ответ, а игроки должны придумать для него вопрос. Измеряется не степень соответствия документа запросу, а вероятность того, что к данному документу мог бы быть предъявлен данный запрос.
Латентно-семантическое индексирование	Матричный подход, при котором производится попытка моделировать документы на концептуальном уровне с помощью сингулярного разложения матрицы. Защищен патентом, поэтому для пользователей ПО с открытым исходным кодом малоинтересен.

Таблица 3.9. (окончание)

Название	Описание
Нейронные сети и другие методы машинного обучения	Функция поиска обучается на обучающем наборе с обратной связью от пользователя.
Альтернативные схемы взвешивания	Различные исследователи предлагали варианты схем взвешивания, применяемые во многих моделях. В некоторых из них в основу оценивания положена длина и средняя длина документа (Okapi BM25, опорная нормировка длины документа и другие). Основная идея заключается в том, что в длинных документах значение TF для ключевого слова с высокой вероятностью будет больше, чем в коротких, но преимущества от повторения термов уменьшаются с ростом длины документа. Отметим, что коэффициент нормировки длины нелинеен.
Вероятностные модели и родственные методы	Используется статистический анализ для определения вероятности того, что данные документ соответствует запросу. Родственными методами являются сети вывода и языковое моделирование.

Наконец, отметим, что ведутся активные исследования по улучшению качества обработки запросов – с применением разных моделей и разных формулировок запросов. Много хороших и интересных работ публикуется в вестнике специальной группы ACM по информационному поиску (SIGIR) на ежегодной конференции. Следите за этими публикациями, и вы будете в курсе последних достижений в области улучшения качества результатов. А теперь перейдем к конкретным применяемым в Solr методам, которые вы можете включить в свою систему.

3.7.5. Способы повышения производительности Solr

Хотя Solr прекрасно работает безо всякой настройки, есть целый ряд рекомендаций, которым нужно следовать, чтобы сервер Solr мог показать все, на что способен. Если говорить о производительности, то проблему можно разбить на две части: производительность индексирования и поиска. Прежде чем что-то делать, отметим, что простейший способ повысить производительность – установить последнюю версию программы. Сообщество активно работает, так что усовершенствования появляются постоянно.

Повышение производительности индексирования

У производительности индексирования есть три стороны:

- проектирование схемы;
- конфигурирование;
- способ отправки содержимого.

Правильный подход к проектированию схемы, как уже было сказано ранее, подразумевает определение того, какие поля необходимы, как их следует анализировать и нужно их ли их хранить. Поиск по многим полям обычно медленнее, чем по одному. И извлечение документов с большим число хранимых полей также производится дольше, чем в случае, когда хранимых полей мало. Кроме того, сложный процесс анализа негативно влияет на производительность индексирования вследствие затрат времени на нетривиальный лексический анализ и фильтрацию лексем. В зависимости от пользовательской аудитории зачастую имеет смысл пожертвовать качеством в пользу быстродействия.

Конфигурационный файл Solr предлагает много уровней управления производительностью; из них упомянем параметры, сообщающие Lucene (библиотеке, лежащей в основе поиска), как создавать и записывать файлы, в которых хранится индекс. Эти параметры задаются в разделе `<indexDefaults>` файла solrconfig.xml. Ниже приведен пример.

```
<useCompoundFile>false</useCompoundFile>
```

Использование формата составного файла Lucene позволяет сэкономить на дескрипторах файлов ценой замедления поиска и индексирования

```
<mergeFactor>10</mergeFactor>
```

Параметр mergeFactor определяет, как часто Lucene объединяет внутренние файлы. Если значение невелико (< 10), то потребляется меньше памяти ценой замедления индексирования. В большинстве случаев значение по умолчанию можно оставить, но, возможно, вы захотите поэкспериментировать

```
<maxBufferedDocs>1000</maxBufferedDocs>
```

Параметр maxBufferedDocs определяет, сколько документов хранить во внутреннем буфере перед сбросом на диск. Чем больше значение, тем больше потребляется памяти и тем быстрее индексирование

```
<maxMergeDocs>2147483647</maxMergeDocs>
```

Параметр maxMergeDocs задает максимальное число документов, объединяемых Lucene. Небольшие значения (<10 000) лучше подходят для систем, которые часто обновляются, большие – для пакетного индексирования и ускорения поиска

```
<maxFieldLength>10000</maxFieldLength
```

Параметр maxFieldLength задает максимальное число индексируемых лексем из одного поля. Увеличьте это значение (и размер кучи Java), если ожидаете появления документов с большим количеством лексем

Наконец, производительность индексирования существенно зависит от того, как приложение отправляет документы Solr. Рекомендуется отправлять по несколько документов за раз в POST-запросах по протоколу HTTP. Дополнительный рост возможен за счет отправки документов из нескольких потоков, что повышает пропускную способность и минимизирует накладные расходы на HTTP. Сервер Solr берет на себя заботы о синхронизации, так что вы можете не тревожиться за правильность индексирования своих данных. Ниже рассмотрены различные вопросы, относящиеся к производительности Solr.

Производительность поиска

У производительности поиска есть несколько аспектов, и по каждому направлению предлагаются различные уровни производительности (см. табл. 3.10).

Таблица 3.10. Направления повышения производительности поиска

Направление	Описание
Тип запроса	Solr поддерживает развитый язык запросов, допускающий как простые термы, так и запросы с метасимволами и по диапазону. Чем сложнее запрос, тем медленнее он выполняется.
Размер	Производительность зависит как от размера запроса (количества частей), так и от размера индекса. Чем больше индекс, тем больше термов приходится просматривать (время обычно зависит от числа документов сублинейно). Чем больше термов в запросе, тем больше документов и полей приходится проверять. Чем больше полей, тем больше содержимого приходится просматривать, если поиск производится по всем полям.
Анализ	Как и при индексировании, сложные процессы анализа требуют больше времени, чем простые, но обычно разница пренебрежимо мала, если только речь не идет о действительно длинном запросе или расширении запроса многочисленными синонимами.
Стратегии кэширования и прогрева кэша	В Solr имеются развитые механизмы кэширования запросов, документов и прочих важных структур. Более того, сервер может автоматически заполнить некоторые кэши еще до того, как произведенные в индексе изменения станут доступны для поиска. О кэшировании см. комментарии в файле solrconfig.xml. Анализ журнала запросов и административный интерфейс Solr могут помочь в определении того, полезно кэширование при поиске или нет. Если бесполезно (много промахов кэша), то его лучше отключить.

Таблица 3.10. (окончание)

Направление	Описание
Репликация	С большим потоком запросов можно справиться путем репликации индексов Solr на несколько серверов с балансируемой нагрузкой. Solr предлагает комплект инструментов для синхронизации индексов на разных серверах.
Распределенный поиск	Большие индексы можно разместить на нескольких машинах (секционировать). Главный узел рассылает входящие запросы всем секциям, а затем собирает полученные от них результаты. В сочетании с репликацией это позволяет строить большие отказоустойчивые системы.

Как почти всегда с оптимизацией, стратегия, работающая для одного приложения, может не подойти другому. Приведенные выше рекомендации – это лишь общие эвристические правила использования Solr, но узнать, что оптимально именно в вашем приложении, можно только с помощью прагматичного тестирования и профилирования на собственных данных и серверах. Далее мы рассмотрим некоторые альтернативы Solr для Java и других языков.

3.8. Альтернативные поисковые системы

Одно из величайших достоинств ПО с открытым исходным кодом – тот факт, что любой человек может состряпать проект и сделать его доступным всем желающим (естественно, это одновременно и недостаток, потому что трудно разобраться, что хорошо, а что плохо). В своем проекте вы можете воспользоваться любой из целого ряда открытых поисковых библиотек, причем у всех них разные проектные установки. Одни стремятся к максимальной скорости, другие нацелены на проверку новых теорий поиска и потому более академичны.

Хотя авторы этой книги много работали с Solr и Lucene и пристрастны в отношении этих решений, в табл. 3.11 описаны некоторые альтернативные подходы и реализации для других языков.

Есть и много других поисковых систем, но перечисленные в табл. 3.11 – представительная подборка инструментов для разных языков и с разными лицензиями.

Таблица 3.11. Альтернативные поисковые системы

Направление	URL	Особенности	Лицензия
Apache Lucene и варианты	http://lucene.apache.org/	Низкоуровневая библиотека, требующая значительного опыта, но зато и обеспечивающая большую гибкость, контроль над потреблением памяти и т. д. Существуют реализации Lucene API для .NET, Python (PyLucene) и Ruby (Ferret), каждая из которых в той или иной мере совместима с Lucene.	Apache Software License (ASL)
Apache Nutch	http://lucene.apache.org/nutch/	Полноценный робот-обходчик, индексатор и поисковая система, построенная на базе Apache Hadoop и Lucene Java.	Apache Software License (ASL)
ElasticSearch	http://elasticsearch.com	Поисковый сервер на базе Lucene.	Apache Software License (ASL)
Minion	https://minion.dev.java.net/	Поисковая система с открытым исходным кодом от Sun Microsystems.	GPL v2.0
Sphinx	http://www.sphinxsearch.com/	Поисковая система, ориентированная в первую очередь на индексирование содержимого, хранящегося в базе данных, поддерживающей язык SQL.	GNU Public License v2 (GPL)
Lemur	http://www.lemurproject.org/	Вместо векторной модели используется альтернативная модель ранжирования – языковое моделирование.	BSD
MG4J, Managing Gigabytes for Java	http://mg4j.dsi.unimi.it/	Поисковая система, основанная на идеях великолепной книги «Managing Gigabytes» (Witten [1999]). По замыслу создателей, является быстрой и масштабируемой. Предлагает также различные алгоритмы ранжирования.	GPL

Таблица 3.11. (окончание)

Направление	URL	Особенности	Лицензия
Zettair	http://www.seg.rmit.edu.au/zettair/	Задумана компактной и быстрой, позволяет индексировать HTML-документы и наборы данных TREC, а затем производить в них поиск.	BSD

3.9. Резюме

Поиск по собственному содержимому – богатая и сложная тема. Предоставление удобного доступа к содержимому – первый шаг на пути обретения контроля над данными, пронизывающими всю вашу жизнь. К тому же, поиск сегодня – один из обязательных компонентов в любом приложении, ориентированном на пользователя. Amazon, Google, Yahoo! и другие компании убедительно продемонстрировали, какие возможности несет хороший поиск равно пользователям и компаниям. Теперь дело за вами – воспользуйтесь рассмотренными в этой главе идеями, чтобы сделать свое приложение функционально более насыщенным. Конкретно, вы теперь должны понимать основы индексирования и поиска, знать, как настроить и использовать Apache Solr, и разбираться в некоторых проблемах поиска (вообще в и Solr в частности) в реальных приложениях. Утвердившись на этом плацдарме, мы далее рассмотрим методы работы с текстом в ситуациях, когда результаты не всегда предсказуемы. Мы имеем в виду так называемое *неточное сравнение строк*.

3.10. Ресурсы

Baeza-Yates, Ricardo and Ribiero-Neto, Berthier. 2011 *Modern Information Retrieval: The Concepts and Technology Behind Search, Second Edition*. Addison-Wesley.

Brin, Sergey, and Lawrence Page. 1998. “The Anatomy of a Large-Scale Hypertextual Web Search Engine.” http://infolab.stanford.edu/~backrub/google.html.

Grossman, David A. and Frieder, Ophir. 1998. *Information Retrieval: Algorithms and Heuristics*. Springer.

Nie, Jian-yun; Gao, Jiangfeng; Zhang, Jian; Zhou, Ming. 2000. “On the Use of Words and N-grams for Chinese Information Retrieval.”

Fifth International Workshop on Information Retrieval with Asian Languages, IRAL2000, Hong Kong, pp 141- 148.

Salton, G; Wong, A; Yang, C. S. 1975. "A Vector Space Model for Automatic Indexing." *Communications of the ACM*, Vol 18, Number 11. Cornell University. http://www.cs.uiuc.edu/class/fa05/cs511/Spring05/other_papers/p613-salton.pdf.

"Vector space model." Wikipedia. http://en.wikipedia.org/wiki/Vector_space_model.

Witten, Ian; Moffatt, Alistair; Bell, Timothy. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann, New York.

ГЛАВА 4. Неточное сравнение строк

В этой главе:

- Поиск неточно совпадающих строк с помощью префиксов и *n*-грамм.
- Использование префиксного поиска для упреждающего ввода запроса.
- Применение *n*-грамм для проверки правильности написания запроса.
- Методы сравнения строк в применении к задаче сравнения записей.

Одна из самых сложных сторон работы с текстом – приблизительность многих решаемых задач. Идет ли речь о релевантности результатов поиска или кластеризации похожих результатов, очень трудно определить, что такое релевантность или похожесть, так чтобы это было одновременно точно и интуитивно понятно. В жизни мы сталкиваемся с этим явлением постоянно, даже не задумываясь о причудах языка. Например, услышав такое описание новой музыкальной группы: «Они, как Radiohead, только другие», вы просто кивнете, приняв первую пришедшую в голову интерпретацию и не рассматривая великое множество других – потенциально тоже правильных.

Мы точно помним, как в поиске Google появилась функция «Возможно, вы имели в виду» (Did you mean?) (см. рис. 4.1, в настоящее время Google пишет «Показаны результаты по запросу», когда есть высокая степень уверенности, что в запросе допущена орфографическая ошибка). Хотя эта функция была, скорее всего, ориентирована на людей, делающих опечатки в запросах, другие – те, кто (как бы ска-

зать помягче?) были не вполне в ладах с орфографией, – испытали настоящий всплеск продуктивности. Мало того что таким образом повышалась продуктивность поиска, так еще и под рукой оказалось средство искать слова, которых либо нет в словарях, либо настолько сложных, что невозможно было даже выдвинуть разумную гипотезу об их написании. Ныне мы можем быстро написать «joie de vivre[1] кодирования по ночам» вместо пресного «кодирование – это вещь», поскольку проверить, как пишется это заимствованное из французского выражение теперь очень просто.

Рис. 4.1. Пример рекомендации «Did You Mean» (Возможно, вы имели в виду) в поиске Google. Снимок экрана сделан 1.02.2009

Функции типа «возможно, вы имели в виду», или, как их иногда называют, *проверка орфографии запроса* нуждаются в неточном сравнении. Если говорить конкретно, то необходимо сгенерировать несколько потенциальных рекомендаций для введенного запроса, ранжировать их и определить, нужно ли вообще показывать какую-нибудь рекомендацию пользователю. *Неточное сравнение строк* (похожее на сопоставление с регулярным выражением, только другое), или просто *неточное сравнение* – это процесс поиска похожих, но не обязательно в точности совпадающих строк. Напротив, говоря о *сравнении строк*, мы имеем в виду точное совпадение. Степень похожести обычно определяется в терминах расстояния, оценки или вероятности. Например, в смысле редакционного расстояния (расстояния Левенштейна), которое мы опишем ниже, слова *there* и *here* находятся на расстоянии 1. Наверное, вы догадываетесь, что понимается под этой конкретной оценкой схожести, но пока мы не будем уточнять, а вернемся к этой теме позже.

Проверка орфографии – лишь один пример неточного сравнения строк. Другая типичная ситуация возникает, когда две компании сливаются и должны объединить списки своих клиентов. Еще пример –

[1] Радость жизни (*франц.*)

государство или авиакомпания проверяет полетный лист на предмет наличия в нем потенциальных преступников. Такого рода применения называются *связыванием записей* или *разрешением сущностей*; задача состоит в том, чтобы сравнить один список с другим и попытаться определить, верно ли, что два похожих имени на самом деле относятся к одному человеку. Простого сравнения имен для получения полного ответа недостаточно, но это, тем не менее, важный шаг. Для исследования этих примеров мы в этой главе опишем несколько подходов к неточному сравнению строк и упомянем об их реализации в виде открытого исходного кода. Вы узнаете, как сравнивать слова, короткие фразы и имена между собой, находить похожие и ранжировать по степени похожести. Мы также обсудим применение этих методов в конкретных приложениях и увидим, как с помощью Solr и сравнительно короткого пользовательского кода на Java строятся такие приложения. Наконец, объединив все изложенное, мы напишем некоторые типичные приложения с неточным сравнением строк.

4.1. Различные подходы к неточному сравнению строк

О том, как сравниваются строки, программисты редко задумываются. В большинстве языков программирования для этого есть готовые средства. Ну а если озаботиться этим вопросом, то нетрудно представить себе реализацию функции, которая сравнивает соответственные символы в каждой строке и возвращает true, только если все они совпадают.

Но неточное сравнение строк сразу же поднимает ряд вопросов, ответы на которые отнюдь не очевидны. Например:

- Сколько символов должны совпадать?
- Что, если символы одинаковы, но следуют в разном порядке?
- Что, если имеются лишние символы?
- Верно ли, что одни символы в каком-то смысле важнее других?

В различных подходах к неточному сравнению на эти вопросы отвечают по-разному. В одних упор делается на множество общих символов, как основную меру схожести строк. В других более четко моделируется порядок следования символов, а в третьих рассматривается сразу несколько символов. Мы отведем каждому подходу свой раздел. В первом разделе «Меры, основанные на множестве общих

символов» мы рассмотрим меру Жаккара и некоторые вариации этой идеи, в том числе расстояние Джаро-Винклера. В следующем разделе, «Меры, основанные на редакционном расстоянии», речь пойдет о подходе, ставящем во главу углу порядок символов, и его вариантах. И наконец, учет одновременно нескольких символов станет темой раздела «Редакционное расстояние на основе *n*-грамм».

Но прежде чем приступать к обсуждению различий, вы должны свыкнуться с мыслью о том, что, в отличие от сопоставления с регулярными выражениями, результатом алгоритма неточного сравнения почти никогда не является булевское значение. А возвращают такие алгоритмы число, показывающее степень подобия строк. По общепринятому соглашению это вещественное число от 0 до 1, причем 1 означает, что строки совпадают полностью. Далее мы увидим, как можно вычислять эти значения, и начнем с подхода, основанного на пересечении множеств символов.

4.1.1. Меры, основанные на множестве общих символов

Один из подходов к неточному сравнению строк основан на том, сколько у них общих символов. Интуитивно кажется очевидным, что строки, в которых много общих символов, больше похожи друг на друга, чем строки, в которых общих символов мало или нет вовсе. В этом разделе мы рассмотрим два подхода на основе этой идеи. Первый – мера Жаккара и ее варианты. Затем мы поговорим о расстоянии Джаро-Винклера.

Мера Жаккара

Мера Жаккара, или *коэффициент сходства* – это один из способов формализовать интуитивное представление о том, что строки с большим числом общих символов похожи. В контексте сравнения строк эта величина вычисляется как процентная доля уникальных символов, общих для двух строк, относительно общего числа уникальных символов в обеих строках. Точнее, если A – множество символов первой строки, а B – множество символов второй строки, то мера Жаккара выражается следующей формулой:

$$\frac{|A \cap B|}{|A \cup B|}$$

В случае меры Жаккара все символы трактуются одинаково, не уменьшается вес часто встречающихся общих символов и не увеличивается вес редких символов. Ниже приведен код вычисления меры Жаккара.

Листинг 4.1. Вычисление меры Жаккара

```
public float jaccard(char[] s, char[] t) {
 int intersection = 0;
 int union = s.length+t.length;
 boolean[] sdup = new boolean[s.length];
 union -= findDuplicates(s,sdup);
 boolean[] tdup = new boolean[t.length];
 union -= findDuplicates(t,tdup);
 for (int si=0;si<s.length;si++) {
  if (!sdup[si]) {
    for (int ti=0;ti<t.length;ti++) {
      if (!tdup[ti]) {
        if (s[si] == t[ti]) {
          intersection++;
          break;
        }
      }
    }
  }
 }
 union-=intersection;
 return (float) intersection/union;
}

private int findDuplicates(char[] s, boolean[] sdup) {
 int ndup =0;
 for (int si=0;si<s.length;si++) {
  if (sdup[si]) {
    ndup++;
  }
  else {
    for (int si2=si+1;si2<s.length;si2++) {
      if (!sdup[si2]) {
        sdup[si2] = s[si] == s[si2];
      }
    }
  }
 }
 return ndup;
}
```

Ищем дубликаты и вычитаем из объединения

Пропускаем дубликаты

Находим пересечение

Возвращаем меру Жаккара

В листинге 4.1 мы сначала вычисляем мощность объединения (знаменатель), для чего вычитаем количество повторяющихся сим-

волов из общего числа символов в двух строках. Затем вычисляется количество символов, общих для обеих строк (числитель). И в конце возвращается итоговая оценка.

Хорошо известное обобщение меры Жаккара основано на идее приписывания символам различных весов в зависимости от частоты вхождения. Так устроена мера TF–IDF, рассмотренная в главе 3. При этом естественно было бы положить в основу меры схожести функцию косинуса, как в случае поиска, но она возвращает значение от –1 до 1. Чтобы нормировать эту меру и возвращать значение от 0 до 1, можно модифицировать ее следующим образом:

$$\frac{A \cdot B}{\|A\|^2 + \|B\|^2 - A \cdot B}$$

Это выражение, называемое *коэффициентом Танимото*, совпадает с мерой Жаккара, когда всем символам приписан одинаковый вес.

Поскольку в Solr результаты поиска ранжируются с помощью косинусоидальной функции, которая дает оценки, похожие на получающиеся при вычислении коэффициента Танимото, то простейший способ реализовать такой тип оценивания состоит в том, чтобы проиндексировать словарь или иначе устроенный набор символов, считая каждый терм документом, а каждый символ – отдельной лексемой в смысле Solr. Для этого нужно задать образец для лексического анализатора, как показано ниже:

```
<fieldType name="characterDelimited" class="solr.TextField">
 <analyzer>
   <tokenizer class="solr.PatternTokenizerFactory" pattern="."
              group="0" />
 </analyzer>
</fieldType>
```

Если теперь обратиться к этому полю с запросом, содержащим один терм, то Solr произведет косинусоидальное ранжирование термов на основе частоты появления символов в словах из словаря. Если вы хотите поэкспериментировать с простой версией такого индексирования термов, то взгляните на метод `cosine` класса `com.tamingtext.fuzzy.OverlapMeasures`. На практике он дает разумные результаты, но испытывает затруднения при выборе между результатами с похожими оценками. Кроме того, при таком подходе к оцениванию игнорируется позиция символа, хотя она могла бы помочь при выборе рекомендации. Для решения этой проблемы рассмотрим другую меру, в которой позиция учитывается.

Расстояние Джаро-Винклера

Один из недостатков подхода, основанного на множестве общих символов, заключается в том, что в нем не учитывается порядок символов. Например, если в рассмотренных выше мерах символ в начале строки соответствует символу в конце строки, то считается, что такое совпадение ничем не хуже, чем совпадение символов в близких частях строк. Крайнее проявление этого эффекта – тот факт, что две точно совпадающие строки оказываются настолько же похожими, как исходная и инвертированная строка. Расстояние Джаро-Винклера – это попытка эвристически разрешить эту проблему тремя разными способами. Первый состоит в том, что мы ограничиваем сравнение окном символов второй строки, вычисляемым на основе длины большей из двух строк. Второй – в том, что учитывается также количество перестановок, т. е. символов, встречающихся в разном порядке. Наконец, к оценке прибавляется поправка, основанная на длине самого длинного общего префикса двух строк. В Интернете есть много обсуждений этого подхода, а для тех, кто хочет воспользоваться этим расстоянием, имеется эффективная реализация в классе Lucene `org.apache.lucene.search.spell.JaroWinklerDistance`.

Основной недостаток подходов на базе множества общих символов в том, что они плохо моделируют порядок следования символов. В следующем разделе мы рассмотрим так называемое редакционное расстояние, в котором порядок символов моделируется более формально. Обычно учет порядка требует большего объема вычислений, чем одно лишь нахождение общих символов. Мы поговорим также о способах эффективного вычисления таких мер.

4.1.2. Редакционные расстояния

Еще один подход к определению степени схожести двух строк – вычисление *редакционного расстояния* между ними. Редакционным расстоянием между двумя строками называется количество операций редактирования, необходимых для превращения одной строки в другую. Операции бывают разными, но обычно включают вставку, удаление и замену.

- Операция вставки добавляет символ в первую строку, чтобы сделать ее более похожей на вторую.
- Операция удаления удаляет символ.
- Операция замены заменяет один символ в первой строке каким-то символом из второй строки.

Редакционное расстояние равно общему числу операций вставки, удаления и замены, необходимых для преобразования первой строки во вторую. Например, для преобразования строки *tamming test* в *taming text* требуется одно удаление буквы *m* и одна замена буквы *s* буквой *x*, так что редакционное расстояние равно 2. Этот простой вид редакционного расстояния, в котором разрешены только операции вставки, удаления и замены и каждой из них присваивается вес 1, называется *расстоянием Левенштейна*.

Вычисление редакционного расстояния

Существует много последовательностей операций, преобразующих одну строку в другую, но нас обычно интересует преобразование с наименьшим числом операций. Вычисление минимальной последовательности поначалу может показаться вычислительно сложным, однако же его можно выполнить, произведя всего $n \times m$ сравнений, где n – длина одной строки, а m – длина другой. Алгоритм решения этой задачи – классический пример динамического программирования, когда задача сводится к определению оптимальной операции редактирования при данных смещениях от начала обеих сравниваемых строк. В листинге 4.2 приведен код алгоритма на Java. Отметим, что мы выбрали прямолинейный подход к вычислению расстояния Левенштейна; существуют более эффективные реализации, требующие меньше памяти. Оставляем исследование этого вопроса в качестве упражнения для читателя.

Листинг 4.2. Вычисление редакционного расстояния

```
public int levenshteinDistance(char s[], char t[]) {
 int m = s.length;
 int n = t.length;
 int d[][] = new int[m+1][n+1];
 for (int i=0;i<=m;i++)
  d[i][0] = i;
 for (int j=0;j<=n;j++)
  d[0][j] = j;
 for (int j=1;j<=n;j++) {
  for (int i=1;i<=m;i++) {
   if (s[i-1] == t[j-1]) {
    d[i][j] = d[i-1][j-1];
   } else {
    d[i][j] = Math.min(Math.min(
       d[i-1][j] + 1,
       d[i][j-1] + 1),
       d[i-1][j-1] + 1);
```

Выделяем память для матрицы расстояний

Инициализируем верхнюю границу расстояния

Стоимость такая же, как для предыдущего совпадения

Стоимость равна 1 для вставки, удаления или замены

```
      }
    }
  }
  return d[m][n];
}
```

В табл. 4.1 показано, как выглядит матрица расстояний для строк *taming text* и *tamming test*. В каждой ячейке находится минимальная стоимость редактирования одной строки для получения другой. Например, мы видим стоимость удаления буквы *m* в ячейке на пересечении строки 3 и столбца 4 и стоимость замены в ячейке на пересечении строки 10 и столбца 11. Минимальное расстояние редактирования всегда оказывается в правом нижнем углу матрицы расстояний.

Таблица 4.1. Матрица для вычисления редакционного расстояния

		t	a	m	m	i	n	g		t	e	s	t
	0	1	2	3	4	5	6	7	8	9	10	11	12
t	1	0	1	2	3	4	5	6	7	7	8	9	10
a	2	1	0	1	2	3	4	5	6	7	8	9	10
m	3	2	1	0	1	2	3	4	5	6	7	8	9
i	4	3	2	1	1	1	2	3	4	5	6	7	8
n	5	4	3	2	2	2	1	2	3	4	5	6	7
g	6	5	4	3	3	3	2	1	2	3	4	5	6
	7	6	5	4	4	4	3	2	1	2	3	4	5
t	8	6	6	5	5	5	4	3	2	1	2	3	4
e	9	7	7	6	6	6	5	4	3	2	1	2	3
x	10	8	8	7	7	7	6	5	4	3	2	2	3
t	11	8	9	8	8	8	7	6	5	3	3	3	2

Научившись вычислять редакционное расстояние, посмотрим, как оно используется.

Нормировка редакционного расстояния

В большинстве приложений, где используется редакционное расстояние, требуется задать пороговую величину расстояния, чтобы исключить преобразования, включающие слишком много операций. При выполнении такого вычисления мы довольно быстро сталкива-

емся со следующей проблемой. Интуитивно кажется, что редакционное расстояние 2 для строки длиной 4 гораздо больше, чем такое же расстояние для строки длиной 10. Кроме того, приходится ранжировать довольно большое число возможных вариантов, и это число зависит от редакционного расстояния. При этом не нужно забывать, что длина получающихся после преобразования строк может различаться. Чтобы сравнивать редакционные расстояния для строк разной длины, полезно нормировать расстояние на длину строки.

Для нормировки с целью получения числа от 0 до 1 мы вычитаем расстояние из длины большей из двух строк и делим его на эту длину. В рассмотренном выше примере более длинной является строка *tamming test*, поэтому нормированное значение равно (12 – 2) / 12 = 0,833. Если корректируется короткая строка, например *tammin* с целью получения *taming*, то нормированное расстояние равно (6 – 2) / 6 = 0,666. В шаге нормировки находит отражение интуитивное представление о том, что две операции редактирования во втором примере составляют более существенное изменение, чем две операции редактирования в первом примере. Этот процесс также упрощает назначение пороговых значений, потому что нормированные редакционные расстояния можно сравнивать для строк разной длины.

Взвешивание операций редактирования

В различных приложениях операциям редактирования, используемым для вычисления редакционного расстояния, назначаются веса. В таких случаях редакционное расстояние равно сумме весов операций, задействованных в преобразовании одной строки в другую. Как мы видели на примере расстояния Левенштейна, в простейшем случае всем операциям назначается вес 1. Назначать операциям разные веса полезно, когда не все операции равновероятны. Например, при исправлении орфографических ошибок, замена одной гласной буквы другой – тоже гласной – более вероятна, чем замена одной согласной на другую. Назначение весов операциям в зависимости от их операндов может уловить такие различия.

Распространенный вариант расстояния Левенштейна называется расстоянием Дамерау-Левенштейна. В нем допускается дополнительная операция – перестановка соседних букв. Можно считать это альтернативной схемой взвешивания, когда перестановке соседних букв назначается вес 1, а не вес 2, как было бы, если бы перестановка была реализована с помощью операций удаления и вставки.

В Интернете можно найти много материалов по расстоянию Левенштейна, в том числе формальный анализ и доказательство правильности представленного алгоритма. Для тех, кто собирается воспользоваться этой мерой схожести, в Lucene имеется оптимизированная реализация в классе `org.apache.lucene.search.spell.LevenshteinDistance`. В ней выделяется память всего для двух строк матрицы расстояний, поскольку для вычисления следующей строки необходима только предыдущая. Кроме того, производится описанная выше нормировка по длине.

4.1.3. N-граммное редакционное расстояние

В рассмотренных до сих пор вариантах редакционного расстояния каждая операция манипулировала только одиночными символами. Понятие редакционного расстояния можно обобщить, разрешив действия сразу над несколькими символами. Такое расстояние называется *n-граммным*. В *n*-граммном редакционном расстоянии использована идея расстояния Левенштейна, только символом является *n*-грамма. В табл. 4.2 показана матрица расстояний, вычисленная для рассмотренного выше примера с применением *n*-грамм длины 2 (биграмм).

Таблица 4.2. Матрица для вычисления *n*-граммного редакционного расстояния

		ta	am	mm	mi	in	ng	g_	_t	te	es	st
	0	1	2	3	4	5	6	7	8	9	10	11
ta	1	0	1	2	3	4	5	6	7	8	9	9
am	2	1	0	1	2	3	4	5	6	7	8	9
mi	3	2	1	1	1	2	3	4	5	6	7	8
in	4	3	2	2	2	1	2	3	4	5	6	7
ng	5	4	3	3	3	2	1	2	3	4	5	6
g_	6	5	4	4	4	3	2	1	2	3	4	5
_t	7	6	5	5	5	4	3	2	1	2	3	4
te	8	7	6	6	6	5	4	3	2	1	2	3
ex	9	8	7	7	7	6	5	4	3	2	2	2
xt	10	9	8	8	8	7	6	5	4	3	2	3

Смысл подхода на основе *n*-грамм в том, что вставки и удаления, в которых не участвуют сдвоенные буквы, штрафуются более серьезно, чем в униграммных методах, где все замены штрафуются одинаково.

Улучшения метода *n*-граммного редакционного расстояния

Подход на основе *n*-грамм обычно дополняется несколькими улучшениями. Первое связано с наблюдением, что начальный символ участвует только в одной *n*-грамме, тогда как промежуточные символы часто участвуют во всех *n*-граммах. Во многих приложениях совпадение начальных символов важнее, чем промежуточных. Подход, в котором этот факт используется, называется *аффиксацией*. Его суть состоит в добавлении $n-1$ символов (где n – длина *n*-граммы) в начало строки. В результате первый символ участвует в таком же количестве *n*-грамм, что и промежуточные. Кроме того, слова, которые не начинаются теми же $n-1$ символами, штрафуются за несовпадение префикса. Такую же процедуру можно применить в конце строки, если совпадение концов строк считается важным.

Второе улучшение – установить поощрение для *n*-грамм, у которых есть общие символы. Можно было бы вычислить расстояние Левенштейна между двумя *n*-граммами и нормировать его, разделив на длину *n*-граммы, так чтобы поощрение за частичное совпадение оставалось в диапазоне между 0 и 1. Вместо расстояния Левенштейна можно с тем же успехом применить простое позиционное совпадение. При таком подходе мы подсчитываем количество одинаковых символов в двух *n*-граммах, которые к тому же находятся в одной и той же позиции. Если длина *n*-граммы больше 2, то эту величину вычислить проще. В случае же биграмм она эквивалентна расстоянию Левенштейна. В табл. 4.3 приведена матрица расстояний, вычисленная с учетом аффиксации и поощрения за частичное совпадение.

В Lucene имеется реализация *n*-граммного редакционного расстояния с аффиксацией и нормировкой по длине. Она находится в классе `org.apache.lucene.search.spell.NgramDistance`.

В этом разделе мы рассмотрели различные подходы к определению схожести двух строк путем вычисления того или иного расстояния между ними. Мы обсудили меры на основе множества общих символов – меру Жаккара и ее обобщения, в которых для назначения весов символам используются частоты. Мы говорили также о расстоянии Джаро-Винклера, в котором анализируются общие символы в окне, скользящем вдоль строки. Затем мы перешли к редакционному рас-

стоянию и изучили его простейший вариант – расстояние Левенштейна. Мы обсудили также улучшения этой идеи – нормировку по длине и назначение различных весов операциям редактирования. И наконец, мы обобщили эту меру на несколько букв. До сих пор мы предполагали, что имеются две строки, которые нужно сравнить, и сосредоточились на том, как выполнить такое сравнение. В следующем разделе мы увидим, как находить строки, похожие на введенную строку, не сравнивая ее со всеми возможными кандидатами.

Таблица 4.3. Матрица для вычисления *n*-граммного редакционного расстояния с улучшениями

		0t	ta	am	mm	mi	in	ng	g_	_t	te	ex	xt
	0	1	2	3	4	5	6	7	8	9	10	11	12
0t	1	0	1	2	3	4	5	6	7	8	9	10	11
ta	2	1	0	1	2	4	5	6	7	8	8.5	9.5	10.5
am	3	2	1	0	1.5	3	4	5	6	7	8	9	10
mi	4	3	2	1	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8
in	5	4	3	2	1.5	1.5	1.5	2.5	3.5	4.5	5.5	6.5	7
ng	6	5	4	3	2.5	2.5	2.5	1.5	2.5	3.5	4.5	5.5	6
g_	7	6	5	4	3.5	3.5	3.5	2.5	1.5	2.5	3.5	4.5	5
_t	8	6.5	6	5	4.5	4.5	4.5	3.5	2.5	1.5	2.5	3.5	4
te	9	7.5	6.5	6	5.5	5.5	5.5	4.5	3.5	2.5	1.5	2.5	3
ex	10	8.5	7.5	7	6.5	6.5	6.5	5.5	4.5	3.5	2.5	2	3
xt	11	9.0	8.5	8	7.5	7.5	7.5	6.5	5.5	4.5	3.5	3	2.5

4.2. Нахождение строк, неточно совпадающих с данной

Умение вычислять меру сходства двух строк полезно, но только, если в наличии обе строки. Во многих приложениях, где сравниваются строки, имеется только одна из строк, подаваемых на вход функциям неточного сравнения, описанным в предыдущем разделе. Например, в программе проверке орфографии обычно имеется слово, отсутствующее в словаре, которое предположительно написано неправильно. Если бы у нас был список рекомендаций, то можно было с помощью какой-нибудь из рассмотренных функций, ранжировать его и пред-

ложить пользователю несколько лучших вариантов. Теоретически можно вычислить меру сходства данного слова со всеми словами в словаре. Но это требует непомерных вычислительных ресурсов (а, стало быть, работает медленно), и большинство сравниваемых слов будут совсем непохожи на имеющееся. На практике необходим быстрый способ построения небольшого списка вероятных кандидатов, с которыми уже можно производить вычислительно дорогостоящие сравнения. В этом разделе мы опишем два подхода к построению такого списка – префиксное сравнение и сравнение *n*-грамм – а также их эффективные реализации.

4.2.1. Использование префиксного сравнения в Solr

Один из способов быстро найти множество строк, похожих на заданную, называется *префиксным сравнением*. Эта операция возвращает множество строк, имеющих общий префикс с рассматриваемой. Например, чтобы исправить написание слова *tamming*, можно рассмотреть слова с префиксом *tam*; тогда вместо 100 000 слов в словаре нужно будет исследовать только 35 – различные формы семи слов: tam, tamale, tamarind, tambourine, tame, tamp, tampon. С вычислительной точки зрения такое сокращение очень существенно, а поскольку у строк общий префикс, гарантируется, что они действительно похожи.

Solr предоставляет один способ префиксного сравнения. В процессе добавления документа в индекс Solr мы можем вычислить все префиксы заданной длины и сохранить их как термы, с которыми Solr будет сравнивать запрос. На этапе обработки запроса ту же самую операцию можно применить к запросу, и в результате будет возвращен список термов с общими префиксами. Поскольку это довольно распространенная операция, в Solr имеется реализация в классе `EdgeNGramTokenFilter` (полное имя класса `org.apache.lucene.analysis.ngram.EdgeNGramTokenFilter`). В результате при индексировании терма *taming* в индекс будут помещены также термы *ta*, *tam*, *tami* и *tamin*. В главе 3 мы видели, что для этого достаточно задать в файле schema.xml тип поля, применяемый на этапах индексирования и обработки запроса. Конкретный пример показан в листинге ниже.

Листинг 4.3. Задание типа поля для префиксного сравнения в Solr

```
<fieldtype name="qprefix" stored="false" indexed="true"
           class="solr.TextField">
```

```
 <analyzer>
   <tokenizer class="solr.WhitespaceTokenizerFactory"/>
   <filter class="solr.LowerCaseFilterFactory"/>
   <filter class="solr.EdgeNGramFilterFactory"
           side="front" minGramSize="2" maxGramSize="3"/>
 </analyzer>
</fieldtype>
```

Здесь для порождения префиксов мы используем включенный в Solr класс `EdgeNGramFilterFactory` в сочетании с анализатором, разбивающим текст на лексемы по пробелам, и фильтром, преобразующим лексемы в нижний регистр.

Этот фильтр используется на этапах индексирования и обработки запроса. На этапе обработки запроса он генерирует префиксы, которые должны сравниваться с префиксами, сгенерированными на этапе индексирования. Одно из типичных применений этой возможности – реализация упреждающего ввода запроса, или *автозаполнение*. Префиксное сравнение полезно в данном случае, потому что его семантика интуитивно очевидна, и пользователь, которому предложен список слов с тем же префиксом, что у набираемого запроса, понимает, почему эти слова оказались в списке и как ввод следующего символа изменит содержимое списка. Мы еще вернемся к автозаполнению ниже в этой главе, а в следующем разделе рассмотрим структуры данных, необходимые для эффективного хранения префиксов в памяти; в некоторых приложениях они могут понадобиться.

4.2.2. Использование префиксных деревьев для префиксного сравнения

Хотя Solr и можно использовать для выполнения префиксного сравнения, в некоторых случаях устанавливать соединение с сервером Solr для каждого проверяемого префикса непрактично. И тогда желательно выполнять префиксные запросы прямо в памяти, для чего нужна подходящая структура данных. Такая структура есть и называется *префиксным деревом* (или *trie*-деревом).

Что такое префиксное дерево?

Префиксное дерево – это структура данных, в которой строки хранятся разложенными на символы. Цепочка символов образует путь в дереве, заканчивающийся узлом, который однозначно определяет слово; не исключено, что такая цепочка будет содержать все символы слова. В префиксном дереве на рис. 4.2 видно, что большинство

слов представлены цепочками, содержащими лишь часть входящих в слово символов, поскольку никаких других слов с таким же префиксом в дереве нет. Однако слово *tamp* является префиксом другого слова, *tampon*, поэтому к соответствующему узлу ведет цепочка из всех символов слова.

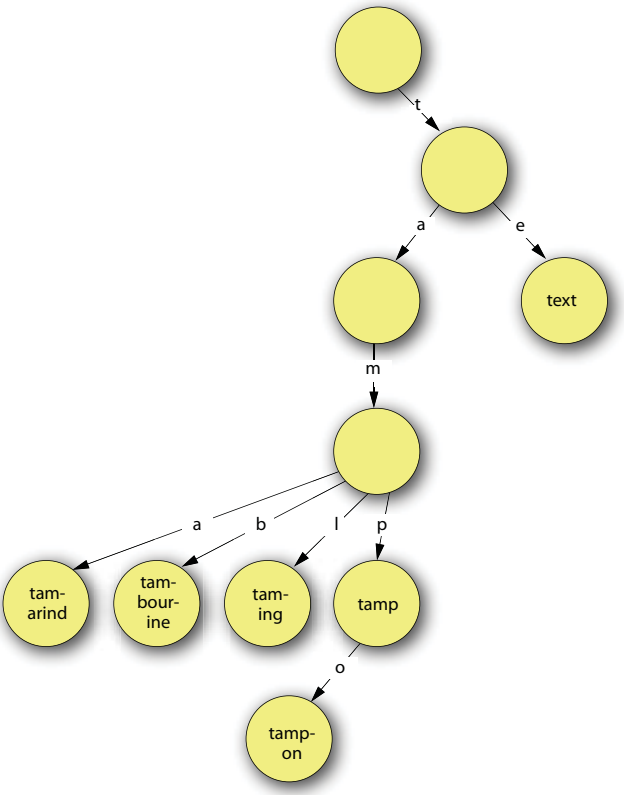

Рис. 4.2. Префиксное дерево для слов *tamarind*, *tambourine*, *taming*, *tamp*, *tampon*, *text*

Реализация префиксного дерева

Простая реализация префиксного дерева во многом похожа на реализацию любой древовидной структуры. В данном случае число потомков каждого узла, как правило, равно количеству символов в алфавите, над которым построено дерево. В листинге ниже показана реализация для строк, состоящих только из строчных букв *a*–*z*.

Листинг 4.4. Построение узла префиксного дерева

```
private boolean isWord;
private TrieNode[] children;
private String suffix;

public TrieNode(boolean word, String suffix) {
 this.isWord = word;
 if (suffix == null) children = new TrieNode[26];
 this.suffix = suffix;
}
```

Этот префикс однозначно определяет слово?

Остаток слова, если префикс однозначно его определяет

Инициализировать потомков, соответствующих каждой букве

Префиксное дерево должно поддерживать операции добавления и поиска слов. Поскольку это древовидная структура, то обычно применяют рекурсивное решение – удаляют первый символ и продолжают спуск, используя оставшиеся символы. Из соображений производительности явное расщепление строки не производится, а просто хранится целое число, показывающее место расщепления. И хотя нужно рассмотреть несколько случаев, рекурсивный подход позволяет написать сравнительно короткий код, показанный в следующем листинге.

Листинг 4.5. Добавление слова в префиксное дерево

```
public boolean addWord(String word) {
 return addWord(word.toLowerCase(),0);
}

private boolean addWord(String word, int index) {
 if (index == word.length()) {
   if (isWord) {
     return false;
   }
   else {
     isWord = true;
     return true;
   }
 }
 if (suffix != null) {
   if (suffix.equals(word.substring(index))) {
     return false;
   }
   String tmp = suffix;
   this.suffix = null;
   children = new TrieNode[26];
   addWord(tmp,0);
 }
```

Проверяем, достигнут ли конец слова

Слово уже существует, возвращаем false

Помечаем, что префикс представляет полное слово

Проверяем, есть ли у этого узла суффикс

Слово уже существует, возвращаем false

Отщепляем суффикс

```
int ci = word.charAt(index)-(int)'a';
TrieNode child = children[ci];
if (child == null) {
  if (word.length() == index -1) {
    children[ci] = new TrieNode(true,null);
  }
  else {
    children[ci] = new TrieNode(false,word.substring(index+1));
  }
  return true;
}
return child.addWord(word, index+1);
}
```

Префикс создает новое слово

Префикс и суффикс создают новое слово

Рекурсивно обрабатываем следующий символ

Для поиска слова производится спуск по дереву до узла, представляющего запрошенный префикс. Для этого проверяется очередной символ префикса и производится переход к дочернему узлу по ветви, соответствующей этому символу. При обнаружении узла, содержащего префикс, выполняется поиск в глубину для сбора всех слов с данным префиксом. Если узла с префиксом не существует, то будет возвращено не более одного слова, причем одно слово возвращается, когда имеется узел, представляющий полное слово, которое в точности совпадает с указанным префиксом. Реализация этого алгоритма показана ниже.

Листинг 4.6. Поиск слов в префиксном дереве

```
public String[] getWords(String prefix, int numWords) {
 List<String> words = new ArrayList<String>(numWords);
 TrieNode prefixRoot = this;
 for (int i=0;i<prefix.length();i++) {
  if (prefixRoot.suffix == null) {
    int ci = prefix.charAt(i)-(int)'a';
    prefixRoot = prefixRoot.children[ci];
    if (prefixRoot == null) {
      break;
    }
  }
  else {
    if (prefixRoot.suffix.startsWith(prefix.substring(i))) {
      words.add(prefix.substring(0,i)+prefixRoot.suffix);
    }
    prefixRoot = null;
    break;
  }
 }
 if (prefixRoot != null) {
```

Спускаемся по дереву, пока еще есть символы в префиксе

Обрабатываем случай, когда префикс не был отщеплен

```
    prefixRoot.collectWords(words,numWords,prefix);
  }
  return words.toArray(new String[words.size()]);
}

private void collectWords(List<String> words,
                          int numWords, String prefix) {
  if (this.isWord()) {
    words.add(prefix);
    if (words.size() == numWords) return;
  }
  if (suffix != null) {
    words.add(prefix+suffix);
    return;
  }
  for (int ci=0;ci<children.length;ci++) {
    String nextPrefix = prefix+(char) (ci+(int)'a');
    if (children[ci] != null) {
      children[ci].collectWords(words, numWords, nextPrefix);
      if (words.size() == numWords) return;
    }
  }
}
```

Собираем все слова, представленные потомками узла, содержащего префикс

Эта реализация префиксного дерева эффективна для добавления и поиска слов, но в каждом узле, не содержащем суффикса, хранится массив, рассчитанный на все символы алфавита. Это позволяет очень эффективно искать символ, но в действительности используется лишь малая часть элементов массива. Существуют другие реализации, например, префиксное дерево с двумя массивами, которые снижают потребление памяти, необходимой для хранения переходов, но увеличивают время вставки нового слова. Во многих приложениях префиксных деревьев, например для поиска в словаре, дерево строится по относительно статическим данным, и тогда этот подход выгоден, потому что дополнительные затраты приходится нести всего один раз. Обсуждение этого алгоритма см. в работе «An efficient digital search algorithm by using a double-array structure» (Aoe [1989]).

Префиксные деревья в Solr

В Solr 3.4 поддерживается основанная на идее префиксного дерева реализация числовых полей, которая существенно повышает производительность запросов по диапазону. Чтобы ей воспользоваться, нужно указать, что поле реализовано классом `TrieField`, как показано ниже.

Листинг 4.7. Использование типа TrieField в Solr

```
<fieldType name="tint" class="solr.TrieField" type="integer"
  omitNorms="true" positionIncrementGap="0" indexed="true"
  stored="false" />
```

В отличие от показанной выше реализации префиксного дерева, версия Solr устроена по принципу префиксных лексем, рассмотренному ранее. И хотя тип TrieField используется в Solr для числовых полей, чтобы понять, как он работает, сначала рассмотрим пример со строками. Если бы мы захотели выполнить запрос по диапазону между строкой *tami* и строкой *tamp*, то могли бы ограничить поиск, взяв общий префикс *tam*, а затем проверить, входит ли в диапазон каждый возвращенный документ. Как мы видели, это существенно ограничивает количество потенциальных кандидатов, проверяемых на соответствие запросу. Можно еще сузить область поиска, введя понятие инкрементирования префикса – *tamj*, *tamk*, *taml* ... *tamp* – а затем производить поиск документов, удовлетворяющих этим префиксам. Отметим, что любое множество результатов, соответствующее одному из инкрементированных префиксов, предшествующих *tamp*, включает только документы, удовлетворяющие запросу по диапазону.

Чтобы определить, какие документы подходят, нужно рассмотреть только те документы, которые соответствуют граничным префиксам, и сравнить их с заданными в запросе границами диапазона. Величина инкремента относительно количества документов, соответствующих префиксу, определяет, сколько необходимо сравнений для вычисления множества документов, отвечающих запросу по диапазону. В данном случае мы использовали инкремент в один символ на строках длиной четыре символа, но возможны и другие инкременты. Можно представить себе поиск в целочисленном диапазоне, например `[314 – 345]`, организованный как поиск чисел с префиксами 31, 32, 33, 34. В классе `solr.TrieField` аналогичный подход используется для вычисления диапазонов целых чисел и чисел с плавающей точкой, но только в двоичном представлении, а не в десятичном, как здесь.

В этом разделе мы рассмотрели представления префиксного дерева, позволяющие эффективно вставлять и искать префиксы. Мы показали простую реализацию и обсудили, как в Solr те же идеи применяются для повышения производительности запросов по диапазону. В следующем разделе мы обратимся к более мощным методам сравнения строк, которым не ограничиваются символами только в начале слова.

4.2.3. Сравнение с помощью n-грамм

Префиксный поиск – штука мощная, но не лишенная ограничений. Одно из них заключается в том, что любой похожий терм, рекомендуемый таким способом, должен иметь общий префикс с введенным словом. Если начальный символ слова или терма введен неправильно, то с помощью префиксного поиска невозможно будет дать подходящие рекомендации. И такие случаи хоть и редко, но встречаются. Поэтому рассмотрим другой метод, который справляется с подобными ситуациями.

В предыдущем разделе мы видели, как с помощью префиксов ограничить множество строк, являющихся кандидатами на сравнение со строкой, введенной пользователем. Мы также видели, что чем длиннее префикс, тем меньше предлагается рекомендаций, но при этом возрастает риск, что хороший терм будет пропущен из-за ошибки, допущенной в префиксе. Идею префиксов можно развить, заметив, что префикс длины n – это первая n-грамма строки. Рассматривая также вторую, третью и последующие n-граммы, мы можем обобщить понятия префикса на все позиции в строке.

Сравнение с помощью n-грамм позволяет ограничить потенциальные совпадения такими, у которых есть одна или более n-грамм, общих со строкой запроса. В примере со словом *tamming* можно было бы рассмотреть не только строки с префиксом *tam*, но и строки, содержащие другие триграммы: *amm*, *mmi*, *min*, *ing*. В словаре из 100 000 слов лишь десятая часть слов содержит какую-нибудь из этих триграмм. И хотя сокращение не больше, чем при использовании одного лишь префикса, теперь мы можем исправлять и другие ошибки в исходном тексте, в том числе в первом символе. Подход на основе n-грамм позволяет очень просто ранжировать найденные совпадения: чем больше совпавших n-грамм, тем лучше рекомендация. В нашем примере 19 слов содержат 4 из 5 n-грамм, а 74 слова (с учетом этих 19) содержат 3 n-граммы. Ранжируя результаты поиска по n-граммам, мы можем принять решение, что будем рассматривать только фиксированное число рекомендаций с достаточно высокой уверенностью в совпадении, причем первым в списке будет слово с наименьшим редакционным расстоянием.

Поиск с помощью n-грамм в Solr

Solr поддерживает не только префиксный поиск, но и поиск с помощью n-грамм. Для этого предназначен класс `org.apache.lucene.`

`analysis.ngram.NGramTokenFilter` и связанный с ним фабричный класс `org.apache.solr.analysis.NGramFilterFactory`.

Но поиск с помощью *n*-грамм не учитывает позиционную информацию. Ценность *n*-граммы, расположенной в начале строки, но совпавшей с n-граммой в конце другой строки, никак не уменьшается. Один из способов преодолеть это ограничение – воспользоваться аффиксацией строк для запоминания позиционной информации о начале или конце строки. Эта техника уже рассматривалась в разделе 4.1.3.

В этом разделе мы обсудили методы быстрого поиска неточно совпадающих строк. В сочетании со средствами вычисления редакционного расстояния это позволяет находить и ранжировать неточные совпадения. А в следующем разделе мы рассмотрим три приложения, в которых все эти инструменты используются совместно.

4.3. Использование неточного сравнения строк в приложениях

В этом разделе мы разработаем три приложения, в которых применяются средства неточного сравнения строк. Точнее, мы реализуем автозаполнение поискового запроса, проверку орфографии запроса и сравнение записей. *Автозаполнение*, или *упреждающий ввод* – это механизм, позволяющий предлагать пользователю примеры хранящихся в индексе термов, чтобы он мог выбрать слово из списка и не нажимать лишние клавиши. Заодно появляется дополнительное преимущество – гарантия того, что слово написано без ошибок. Проверка орфографии запроса, которая часто оформляется на сайтах типа Google в виде секции «Возможно, вы имели в виду», – это просто способ предложить пользователям альтернативное написание слова, которое даст более качественные результаты. Обратите внимание – *альтернативное* необязательно означает правильно написанное. В некоторых случаях очень многие слова, попавшие в индекс, написаны неправильно (например, в сетевых форумах), и результат поиска получится лучше, если предложить пользователю слово с ошибкой. Наконец, *сопоставление записей* (иногда употребляют термины *связывание записей* или *разрешение сущностей*) – это процесс определения того, относятся ли две разных записи к одному и тому же объекту. Например, при объединении двух баз данных пользователей цель сравнения записей – решить, верно ли, что Боб Смит в одной записи и Боб Смит в другой записи – одно и то же лицо. Мы остановимся на

этих трех примерах, поскольку они демонстрируют распространенные применения неточного сравнения строк во многих современных приложениях, работающих с текстом.

4.3.1. Добавления механизма автозаполнения к поиску

Типичная функция многих приложений – автоматическое завершение ввода текста. Например, многие интегрированные среды разработки (IDE) автоматически завершают имена переменных при вводе программы. В поисковых приложениях автозаполнение часто подключается, когда пользователь начинает вводить запрос, и по мере того, как пользователь печатает, ему предлагаются вероятные варианты завершения. Эта функция не только экономит время, позволяя не вводить запрос целиком, но и улучшает результаты поиска, так как предлагаются лишь такие варианты запросов, которым заведомо удовлетворяет хотя бы один документ в индексе. Предложение вариантов, исходя из вводимого запроса, можно считать префиксным запросом. Точнее говоря, вы хотите получить результаты с таким же префиксом, как в запросе. И Solr в этом поможет.

Индексирование префиксов в Solr

Первым делом нужно разрешить Solr создавать префиксные запросы. В случае неполных запросов можно снова воспользоваться классом `EdgeNGramFilterFactory` для вычисления префиксов и добавления их в множество генерируемых лексем. Как и раньше, мы можем определить в файле schema.xml соответствующий тип поля и применить его к полю, в котором будут храниться префиксы (см. листинг 4.8). При этом разрешается задать максимальную длину *n*-граммы – чем длиннее, тем больше вариантов будет предложено пользователю. Это, конечно, увеличит размер необходимых структур данных, но они все же останутся обозримыми.

Листинг 4.8. Задание типа поля для автозаполнения в Solr

```
<fieldtype name="prefix" stored="false" indexed="true"
           class="solr.TextField">                          <-- Задаем анализаторы для индексирования
 <analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.EdgeNGramFilterFactory"               <-- Используем граничные n-граммы (префиксы)
          minGramSize="2" maxGramSize="10"/>
```

```
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldtype>
```

Убираем фильтр граничных *n*-грамм; запросы уже трактуются как префиксы

Этот тип поля позволяет обрабатывать запросы с несколькими словами и допускает совпадения, начинающиеся на границе слова. Отметим, что применять к запросу класс `EdgeNGramFilterFactory` необязательно, потому что запрос и так уже является префиксом. Чтобы воспользоваться этим типом поля, нам необходимо указать поле данного типа и проиндексировать документы для его заполнения. Предположим, к примеру, что мы индексируем словарные статьи в поле `word`. Тогда в файл schema.xml нужно будет добавить поле показанного выше типа:

```
<fields>
 <!-- прочие поля -->
 <field name="wordPrefix" type="prefix" />
</fields>
<!-- прочие атрибуты схемы -->
<copyField source="word" dest="wordPrefix"/>
```

Теперь для документов с полем `word` префиксы на этапе индексирования будут сохраняться в поле `wordPrefix`. Проиндексировав префиксы, покажем, как по ним выполняются запросы.

Получение результатов префиксного поиска в Solr

Используя это поле и тип поля, мы теперь можем производить префиксный поиск в Solr и возвращать список слов. Для этого нужно отправить такой запрос:

```
http://localhost:8983/solr/select?q=wordPrefix:tam&fl=word
```

Чтобы это работало и при вводе запроса из браузера, нужно написать JavaScript-код, который будет отправлять такие запросы по мере того, как пользователь печатает, и отображать лучшие результаты. В этом коде необходимо обрабатывать события нажатия клавиш в поле поиска, периодически посылать запросы серверу, который возвращает список расширений запроса, и отображать полученные результаты. К счастью, все это хорошо известные действия, и существует немало JavaScript-библиотек, в которых они уже реализованы.

Одна из наиболее популярных – script.aculo.us (jQuery тоже поддерживает эту функциональность). Для ее использования вы должны уметь задавать префиксный запрос и возвращать расширения запроса в виде неупорядоченного HTML-списка. Было бы несложно написать сервлет, который взаимодействует с Solr и нужным образом форматирует результаты, но мы воспользуемся этой возможностью, чтобы продемонстрировать, как Solr позволяет настраивать формат запроса и ответа.

Для настройки формата ответа необходимо написать класс, реализующий интерфейс `QueryResponseWriter`, и указать, что он предназначен для формирования ответа на запросы об автозаполнении. Такие классы пишутся легко, и в Solr есть несколько примеров, которые можно взять за образец. Ниже приведен код для нашего простого случая.

Листинг 4.9. Генератор ответов Solr на запросы об автозаполнении

```
public class TypeAheadResponseWriter implements QueryResponseWriter {

 private Set<String> fields;

 @Override
 public String getContentType(SolrQueryRequest req,
                              SolrQueryResponse solrQueryResponse) {
  return "text/html;charset=UTF-8";
 }

 public void init(NamedList n) {
  fields = new HashSet<String>();
  fields.add("word");
 }

 @Override
 public void write(Writer w, SolrQueryRequest req,
                   SolrQueryResponse rsp) throws IOException {
  SolrIndexSearcher searcher = req.getSearcher();
  NamedList nl = rsp.getValues();
  int sz = nl.size();
  for (int li = 0; li < sz; li++) {
   Object val = nl.getVal(li);
   if (val instanceof DocList) {
    DocList dl = (DocList) val;
    DocIterator iterator = dl.iterator();
    w.append("<ul>n");
    while (iterator.hasNext()) {
     int id = iterator.nextDoc();
     Document doc = searcher.doc(id, fields);
     String name = doc.get("word");
```

Определяем поле, отображаемое генератором ответа

Находим список документов

Извлекаем документ с указанным полем

```
        w.append("<li>" + name + "</li>n");
      }
      w.append("</ul>n");
    }
  }
 }
}
```

JAR-файл, содержащий этой класс, следует поместить в библиотечный каталог Solr, обычно solr/lib, чтобы сервер мог найти его во время выполнения. Кроме того, необходимо сообщить Solr о новом генераторе ответов и создать оконечную точку, которая будет использоваться его по умолчанию. Делается это в файле solrconfig.xml, как показано ниже.

Листинг 4.10. Описание генератора ответов и обработчика запросов в Solr

```
<queryResponseWriter name="tah"                    <-- Описываем свой генератор ответов
 class="com.tamingtext.fuzzy.TypeAheadResponseWriter"/>
<requestHandler name="/typeahead"
                class="solr.SearchHandler">        <-- Задаем обработчик запросов в виде URL-адреса
 <lst name="defaults">
  <str name="wt">tah</str>                         <-- Задаем генератор ответов по умолчанию
  <str name="defType">dismax</str>
  <str name="qf"> wordPrefix^1.0 </str>            <-- Задаем поисковое поле
 </lst>
</requestHandler>
```

Новый обработчик запросов позволяет указывать префиксы и форматирует ответ в виде, ожидаемом script.aculo.us. Теперь вы можете отправить Solr следующий запрос, не утруждая себя заданием типа ответа и поискового поля в параметрах.

Листинг 4.11. URL-адрес Solr для отправки префиксного запроса соответствующему обработчику

```
http://localhost:8983/solr/type-ahead?q=tam
```

Динамическое заполнение поискового поля

Ниже показано, как поместить данные в автозаполняемое поле поиска.

Листинг 4.12. HTML и JavaScript-код для автозаполнения поискового поля

```
<html>
 <head>
  <script src="./prototype.js" type="text/javascript">
```

```
  </script>
  <script src="./scriptaculous.js?load=effects,controls"
    type="text/javascript">
  </script>                                    <— Импортируем script.aculo.us

  <link rel="stylesheet" type="text/css"
    href="autocomplete.css" />
</head>
<body>
  <input type="text" id="search"               <— Задаем поле ввода
    name="autocomplete_parameter"/>
  <div id="update" class="autocomplete"/>      <— Определяем div для размещения ответов на запрос об автозаполнения

  <script type="text/javascript">
    new Ajax.Autocompleter('search','update',  <— Создаем объект автозаполнения
      '/solr/type-ahead',
      {paramName: "q",method:"get"});
  </script>
</body>
</html>
```

На рис. 4.3 показано, как выглядит результат работы этого JavaScript-кода в браузере.

Из этого примера вы узнали, как использовать префиксный поиск для включения автозаполнения в свое приложение. Вы добавили в Solr тип поля и поле этого типа для хранения префиксов и применили описанные в нем фильтры лексем для генерации префиксов указанного размера. Благодаря гибкости Solr вы смогли также настроить обработку запроса и ответа, интегрировав ее с популярной JavaScript-библиотекой, в данном случае script.aculo.us. В следующем примере мы продолжим использовать инструменты неточного сравнения для создания приложений.

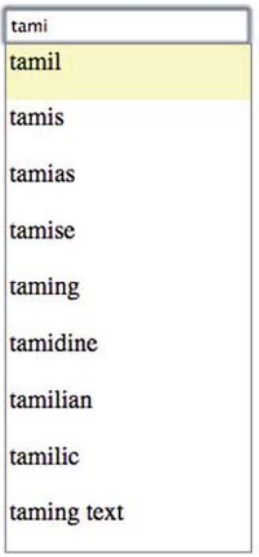

Рис. 4.3. Предложенные варианты автозаполнения для префикса *taml*

4.3.2. Проверка орфографии запроса

Добавление проверки орфографии запроса, или функции «Возможно, вы имели в виду» помогает пользователям сайта отличить не-

правильно записанные запросы от тех, которые не возвращают ни одного результата. Зная это, пользователь сможет уточнять запросы более осмысленно, не испытывая разочарования. В этом разделе мы увидим, как ранжировать варианты правильного написания с вероятностной точки зрения, а затем покажем, как воспользоваться рассмотренными выше инструментами и приемами для приближенного вычисления вероятностей. Это легко сделать с помощью Solr, SolrJ и библиотек, входящих в состав Lucene. Хотя получающееся средство проверки орфографии – не верх совершенства, основу вы получите, а дальше можете улучшать как этот инструмент с учетом особенностей своего приложения, так и сам код проверки орфографии в Lucene. Наконец, мы опишем, как воспользоваться компонентом проверки орфографии, поставляемым вместе с Solr, и как компонентная архитектура Solr упрощает интеграцию с пользовательским компонентом проверки орфографии.

Краткое описание нашего подхода

Задача выбора лучшего исправления для неправильно написанного слова обычно формализуется как задача максимизации вероятности правильного написания и исправления. Для этого нужно вычислить произведение двух вероятностей, $p(s \mid w) \times p(w)$, где s – вариант правильного написания, а w – слово. Первый сомножитель – это вероятность определенного написания слова при условии самого слова, а второй – вероятность самого слова. Поскольку трудно оценить $p(s \mid w)$ без большого количества исправленных данных, которые были проверены человеком, то в качестве разумной аппроксимации принимается редакционное расстояние. Нормированное редакционное расстояние для варианта написания s и слова w можно рассматривать как оценку $p(s \mid w)$. Вероятность предложения слова в качестве рекомендации ($p(w)$) обычно легче оценить, чем вероятность конкретного варианта написания. Во многих случаях можно получить приемлемые результаты, оценив этот сомножитель грубо или вообще отбросив. Именно так и поступает большинство программ проверки орфографии, предлагая пользователю выбрать вариант из списка. В качестве одного из обоснований такого подхода можно привести тот факт, что подавляющее большинство орфографических ошибок (от 80 до 95 %) – это ошибки в одной букве. Редакционное расстояние в этом случае дает удовлетворительное ранжирование рекомендаций.

Такой подход не срабатывает, если нуждающийся в исправлении вариант написания сам является правильным словом, если количество разумных рекомендаций очень велико или если существует только

одна рекомендация. В таких случаях возможность повлиять на ранг рекомендации, принимая во внимание ее правдоподобие, может резко повысить качество работы. Сделать это можно на основе анализа частоты слов в журналах запросов или индексированном тексте. В случае словосочетаний для оценки последовательности слов можно применить *n*-граммные модели, но их обсуждение выходит за рамки этой книги.

Познакомившись с теоретическими основами исправления орфографических ошибок, посмотрим, как это реализуется на практике. Наш подход будет таким:

- построить множество потенциальных вариантов исправления;
- ранжировать потенциальные варианты;
- показать только рекомендации с рангом, превышающим пороговое значение.

Выше мы видели, что сравнение на основе *n*-грамм – хороший способ построения потенциальных соответствий. Отбор термов, которые содержат много *n*-грамм, совпадающих с тем написанием, которое мы пытаемся исправить, даст разумный список кандидатов. Затем для каждого терма можно вычислить редакционное расстояние и по нему произвести ранжирование. И наконец, нужно отбросить варианты, для которых редакционное расстояние превышает пороговую величину, чтобы не давать рекомендации в случае, когда хороших вариантов не найдено. Для определения пороговой величины обычно требуется поэкспериментировать.

Реализация функции «Возможно, вы имели в виду» в Solr

Описанный выше подход можно реализовать с помощью Solr и SolrJ. Сначала необходимо определить в файле schema.xml тип поля для хранения *n*-грамм, само поле и источник данных для него. Это показано в примере ниже.

Листинг 4.13. Изменения схемы, необходимые для поддержки сравнения с помощью *n*-грамм

```
<fieldtype name="ngram" stored="false" indexed="true"
           class="solr.TextField">
 <analyzer>
  <tokenizer class="solr.KeywordTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.NGramFilterFactory"
          minGramSize="2" maxGramSize="10"/>
```

```
 </analyzer>
</fieldtype>

<!-- прочие типы -->
<field name="wordNGram" type="ngram" />
<!-- прочие поля -->
<copyField source="word" dest="wordNGram"/>
```

Далее нужно опросить это поле и вычислить редакционное расстояние между каждым из полученных результатов и конкретным вариантом написания. Соответствующий код приведен в листинге ниже.

Листинг 4.14. Java-код для получения от Solr возможных вариантов исправления и их ранжирования

```
public class SpellCorrector {

 private SolrServer solr;
 private SolrQuery query;
 private StringDistance sd;
 private float threshold;

 public SpellCorrector(StringDistance sd, float threshold)
     throws MalformedURLException {
  solr = new CommonsHttpSolrServer(
          new URL("http://localhost:8983/solr"));
  query = new SolrQuery();
  query.setFields("word");
  query.setRows(50);
  this.sd = sd;
  this.threshold = threshold;
 }

 public String topSuggestion(String spelling)
     throws SolrServerException {
  query.setQuery("wordNGram:"+spelling);
  QueryResponse response = solr.query(query);
  SolrDocumentList dl = response.getResults();
  Iterator<SolrDocument> di = dl.iterator();
  float maxDistance = 0;
  String suggestion = null;
  while (di.hasNext()) {
   SolrDocument doc = di.next();
   String word = (String) doc.getFieldValue("word");
   float distance = sd.getDistance(word, spelling);
   if (distance > maxDistance) {
    maxDistance = distance;
    suggestion = word;
```

Максимальное число рассматриваемых *n*-граммных соответствий

Опрашиваем поле, содержащее *n*-граммы

Вычисляем редакционное расстояние

Оставляем лучшую рекомендацию

```
      }
    }
    if (maxDistance > threshold) {
      return suggestion;
    }
    return null;
  }
}
```

Сравниваем с порогом; если больше, не даем никаких рекомендаций

Объект, передаваемый конструктору класса `SpellCorrector`, должен реализовывать интерфейс `org.apache.lucene.search.spell.StringDistance` и возвращать 1, если строки совпадают, и 0, если они максимально различаются. В составе Lucene имеется несколько реализаций этого интерфейса, в том числе расстояние Левенштейна, расстояние Джаро-Винклера и *n*-граммное расстояние.

Описанный нами подход эквивалентен игнорированию сомножителя, соответствующего вероятности слова в упомянутой выше модели (или трактовке его как константы). Оценить вероятность слова, можно, подсчитав, сколько раз это слово встречается во всем наборе документов или в журналах запросов и поделив эту величину на общее количество слов. Полученную таким образом оценку можно будет использовать при ранжировании наряду с редакционным расстоянием. Если вы хотите, чтобы эта вероятность оказывала влияние на возвращаемые Solr рекомендации, то это можно сделать с помощью механизма повышающих коэффициентов документов.

Повышающий коэффициент увеличивает релевантность документа; он задается на этапе индексирования и обычно больше 1. Внутренняя релевантность документа с повышающим коэффициентом 2 в два раза выше, чем у документа без повышающего коэффициента. Поскольку все потенциальные рекомендации по исправлению орфографических ошибок в Solr моделируются как документы, то вероятность $p(w)$ можно смоделировать как внутреннюю релевантность документа. Определение повышающих коэффициентов для конкретной предметной области требует экспериментирования. После того как коэффициенты определены, они будут оказывать влияние на полученные методом *n*-грамм результаты, возвращаемые в ответ на запрос о потенциальных рекомендациях.

Использование встроенного в Solr компонента проверки орфографии

Поняв, как с помощью Solr реализовать исправление орфографических ошибок, посмотрим на встроенные в Solr механизмы проверки

орфографии. Этот механизм реализован внутри Lucene и интегрирован в Solr. Общий подход аналогичен описанному выше. В Solr к нему можно обратиться как к компоненту поиска, который добавляется в обработчик запросов. Компонент поиска для проверки орфографии определяется следующим образом.

Листинг 4.15. Определение компонента поиска для проверки орфографии в Solr

```
<searchComponent name="spell_component"
 class="org.apache.solr.handler.component.SpellCheckComponent">
 <lst name="spellchecker">
   <str name="name">default</str>
   <str name="field">word</str>                    <-- Здесь хранятся потенциальные рекомендации
   <str name="distanceMeasure">                    <-- Какое расстояние использовать
     org.apache.lucene.search.spell.LevensteinDistance
   </str>
   <str name="spellcheckIndexDir">./spell</str>
   <str name="accuracy">0.5</str>                  <-- Пороговая величина для отбрасывания рекомендаций
 </lst>
</searchComponent>
```

Этот компонент можно добавить в обработчик запросов, поместив его в качестве аргумента после определенных для обработчика умолчаний.

Листинг 4.16. Добавление компонента поиска для проверки орфографии в обработчик запросов

```
<requestHandler ...
 <lst name="defaults">
   ...
 </lst>
 <arr name="last-components">
   <str>spell_component</str>
 </arr>
</requestHandler>
```

Запросы к компоненту проверки орфографии производятся одновременно с обычными запросами, а рекомендации возвращаются вместе с обычными результатами поиска. Это позволяет давать рекомендацию и извлекать результаты в одном запросе к Solr. В тех случаях, когда для исправления орфографических ошибок необходим иной лексический анализатор, нежели указан в обработчике запросов (так обстоит дело с обработчиком DisMax), запрос на проверку орфографии можно поместить в отдельный параметр `spellcheck.q`.

Как уже отмечалось, в компоненте проверки орфографии, реализованном в Lucene и предоставляемом Solr, применяется подход, аналогичный тому, который мы реализовали с помощью SolrJ. Есть и небольшие различия: дополнительный повышающий коэффициент для рекомендаций с совпадающим префиксом и возможность предлагать только слова, частота которых выше, чем у поискового терма, указанного в запросе. Вы можете также написать собственный компонент проверки орфографии и включить его в процедуру сборки Solr. Для этого нужно расширить абстрактный класс `org.apache.solr.spelling.SolrSpellChecker` и реализовать метод `getSuggestions`, а также методы построения и перезагрузки компонента. Как мы уже говорили, этот класс нужно поместить в JAR-файл, который кладется в каталог solr/lib, где сервер Solr сможет его найти. Затем компонент следует описать в конфигурационном файле, как это было проделано для класса `SpellCheckComponent`.

В этом разделе вы видели, как путем сочетания методов нечеткого сравнения с помощью *n*-грамм с методами вычисления редакционного расстояния можно выполнять проверку орфографии. Далее мы воспользуемся этим и другими методами для нечеткого сравнения разнородных полей при решении задачи о сопоставлении записей.

4.3.3. Сопоставление записей

Наш последний пример на тему неточного сравнения строк не настолько функционально богат, как предыдущие, но является основой для многих интересных приложений. По сути своей, сопоставление записей – пример объединения данных. Если существует два источника данных, содержащих записи об одной и той же реальной сущности, и если мы умеем сопоставлять данные в этих записях, то сможем объединить непересекающуюся информацию из разных источников. Зачастую такое объединение позволяет извлечь из данных такую информацию, какую не давал ни один источник в отдельности. Иногда эта задача решается в лоб, но обычно требует неточного сравнения.

Краткое описание нашего подхода

Наш подход к сопоставлению записей на основе неточного сравнения таков:

- найти потенциальных кандидатов;
- ранжировать кандидатов;
- оценить результаты и выбрать единственного кандидата.

Это похоже на подход к исправлению орфографических ошибок, где кандидаты определялись с помощью сравнения *n*-грамм, ранжирование производилось путем вычисления редакционного расстояния и в конце выполнялось сравнение с пороговой величиной. В данном случае добавляется еще одно существенное условие: должен быть только один кандидат с рангом, превышающим порог. Это предотвращает успешное сопоставление в ситуации, когда кандидатов несколько и алгоритм ранжирования неспособен четко разделить их.

В качестве предметной области мы возьмем фильмы, выпущенные на экраны кинотеатров. Здесь есть много источников информации, в частности: база данных фильмов в Интернете (Internet Movie Database, IMDb), NetFlix, программы телепередач и телевидения по заказу, аренда фильмов на сайтах iTunes и Amazon, каталоги DVD-дисков. В этом примере мы будем сопоставлять записи о фильмах из IMDb и службы Tribune Media Service (TMS). TMS предоставляет программы телепередач для Tivo[2] ив Интернете на сайте http://tvlistings.zap2it.com.

Нахождение потенциальных кандидатов с помощью Solr

Наша первая задача – отобрать множество потенциальных кандидатов, для которых впоследствии можно было бы провести более детальное сопоставление. Как и в случае проверки орфографии, для этой цели можно воспользоваться сравнением с помощью *n*-грамм. Мы применим такое сравнение к полю, где шансы обнаружить соответствие максимальны, а информативность наибольшая. Для фильма это, очевидно, будет название. В отличие от проверки орфографии, где совершенно понятно, с каким набором данных сравнивать *n*-граммы (с потенциальными исправлениями), здесь исходным материалом для построения *n*-грамм может быть любой источник данных. И хотя у приложения могут быть причины предпочесть один источник другому, обычно *n*-граммы применяют к тому, в котором записей больше. Связано это с тем, что записи можно прогонять через алгоритм сопоставления несколько раз, и построение *n*-грамм вряд ли изменится. Когда вы вносите в алгоритм улучшения, на обработку меньшего набора записей уходит меньше времени. Кроме того, индексирование часто выполняется быстрее, чем сопоставление записей.

Вернемся к нашему набору данных. Каждый фильм в базе IMDb представлен XML-документом. Типичная запись выглядит следующим образом.

[2] Служба показа программы телепередач на экране телевизора. – *Прим. перев.*

Листинг 4.17. Пример записи в наборе данных IMDb

```
<doc>
 <field name="id">34369</field>
 <field name="imdb">tt0083658</field>
 <field name="title">Blade Runner</field>
 <field name="year">1982</field>
 <field name="cast">Harrison Ford</field>
 <field name="cast">Sean Young</field>
 <!-- Еще много актеров -->
</doc>
```

Ниже показаны относящиеся к делу строки файла schema.xml.

Листинг 4.18. Модификация схемы Solr для задачи сопоставления записей

```
<field name="title" type="ngram"
  indexed="true" stored="true"/>
<field name="year" type="integer
  indexed="true" stored="true"/>
<field name="imdb" type="string"
  indexed="false" stored="true"/>
<field name="cast" type="string" indexed="true"
  multiValued="true" stored="true"/>
```

Поле для хранения *n*-грамм, которое раньше использовалось для проверки орфографии

Многозначное поле cast

Для запроса и получения кандидатов в SolrJ можно написать такой код.

Листинг 4.19. Получение от Solr потенциальных кандидатов для сопоставления записей

```
private SolrServer solr;
private SolrQuery query;

public Iterator<SolrDocument> getCandidates(String title)
       throws SolrServerException {
 String etitle = escape(title);
 query.setQuery("title:""+etitle+""");
 QueryResponse response = solr.query(query);
 SolrDocumentList dl = response.getResults();
 return dl.iterator();
}
```

Экранированное название

Название заключено в кавычки, чтобы предотвратить разбиение на лексемы

Название необходимо экранировать, чтобы специальные символы и строки, например AND, + или !, не интерпретировались как операторы запроса.

Ранжирование потенциальных кандидатов

Получив множество потенциальных кандидатов, мы должны их каким-то образом проранжировать. При проверке орфографии мы использовали с этой целью редакционное расстояние. Для названия фильма это, может быть, и неплохо, но для оптимального сопоставления записей лучше задействовать данные из нескольких полей. Будем рассматривать следующие поля в качестве объектов сопоставления, а заодно опишем, как оценивать каждое поле и всю запись в целом.

- *Название* – здесь редакционное расстояние, пожалуй, будет наилучшей мерой близости. Поскольку названия больше походят на имена, то мы возьмем расстояние Джаро-Винклера, а не Левенштейна и не *n*-граммное.
- *Актеры* – имена актеров, конечно, можно сравнивать с помощью редакционного расстояния, но, по существу, они являются торговой маркой и потому имеют стандартное написание (никто не скажет *Томас Круз* вместо *Том Круз*). А раз так, то почти во всех случаях подойдет точное сравнение с небольшим нормированием. Мы будем измерять перекрытие по актерам процентной долей точно совпавших имен актеров. Поскольку разные источники данных часто указывают различное число актеров, в качестве знаменателя обычно берут наименьшее число.
- *Дата выпуска* – эта дата тоже полезна для различения фильмов с одинаковым названием. Возможны небольшие отличия, потому что в качестве даты одни указывают год запуска фильма в производство, другие – год выхода на экран, третьи – год выпуска DVD. Разность между годами, указанными в источниках, – хорошая мера близости, которую можно нормировать, взяв обратный ранг (reciprocal rank). Это означает, что фильмы с одним и тем же годом выпуска получают оценку 1, а фильмы с годом выпуска, отличающимся на 1, – оценку 1/2. Различия можно также линейно уменьшать, если удается установить порог отсечения, после которого следует считать, что оценка равна 0 или иной константе.

Далее требуется объединить эти члены. Для этого можно вычислить их взвешенную сумму. Так как каждый член нормирован на 1 (принимает значение от 0 до 1), то если выбрать веса, равные в сумме 1, то взвешенная сумма членов тоже будет находиться в диа-

пазоне между 0 и 1. В нашем случае мы заранее припишем вес 1/2 названию, а оставшуюся половину распределим между двумя другими членами. Реализация показана в листинге 4.20. Можно было бы также взять набор уже сопоставленных записей и определить веса эмпирически. Процесс сопоставления сводится к запросу у Solr записей по названию фильма и последующему ранжированию кандидатов.

Листинг 4.20. Получение от Solr потенциальных кандидатов для сопоставления записей

```
private StringDistance sd = new JaroWinklerDistance();

private float score(String title1, int year1, Set<String> cast1,
                   String title2, int year2, Set<String> cast2) {
                                  Для названий используем расстояние Джаро-Винклера
 float titleScore = sd.getDistance(title1.toLowerCase(),
                                   title2.toLowerCase());
                                  Для годов используем обратный ранг
 float yearScore = (float) 1/(Math.abs(year1-year2)+1);
                                  Для актеров используем процент перекрытия
 float castScore = (float) intersectionSize(cast1,cast2)/
       Math.min(cast1.size(),cast2.size());
 return (titleScore*.5f) + (yearScore*0.2f) + (castScore*0.3f);
}                                                  Объединяем все оценки в одну

private int intersectionSize(Set<String> cast1,
                             Set<String> cast2) {   Вычисляем пересечение, используя точное сравненин строк
 int size = 0;
 for (String actor : cast1)
   if (cast2.contains(actor)) size++;
 return size;
}
```

Оценка результатов

Проиллюстрируем этот подход на нескольких примерах. В таблицах 4.4 и 4.5 показаны преимущества объединения нескольких источников данных с последующим сопоставлением.

Таблица 4.4. Демонстрация важности объединения нескольких наборов данных

Идентификатор	Название	Год	Актеры
MV000000170000	Nighthawks	1981	Sylvester Stallone, Billy Dee Williams, ...

Таблица 4.5. Демонстрация важности объединения нескольких наборов данных

Ранг	Взвешенная сумма	Ид	Название	Год	Актеры
0.55	(0.5*1.00) + (0.2*0.25) + (0.3*0.00)	tt0077993	Nighthawks	1978	Ken Robertson, Tony Westrope, ...
0.24	(0.5*0.43) + (0.2*0.12) + (0.3*0.00)	tt0097487	Hawks	1988	Timothy Dalton, Anthony Edwards, ...
0.96	(0.5*0.98) + (0.2*1.00) + (0.3*0.88)	tt0082817	Night Hawks	1981	Sylvester Stallone, Billy Dee Williams, ...

Кандидаты перечислены в том порядке, в котором вернул Solr, исходя из совпадения одних только названий. Как видим, объединение данных позволяет не только присвоить правильной записи более высокий ранг, чем другим кандидатам, но и исключить прочих кандидатов, так что вы можете быть уверены в правильности записи с наивысшим рангом. В общем случае, применение *n*-грамм для выборки кандидатов и редакционного расстояния Джаро-Винклера позволяет справляться с различными вариантами представления данных: знаками препинания, числами, подзаголовками и даже орфографическими ошибками. Ниже приведены примеры такого вариативного написания:

- *Willy Wonka and the Chocolate Factory* и *Willy Wonka & the Chocolate Factory*;
- *Return of the Secaucus 7* и *Return of the Secaucus Seven*;
- *Godspell* и *Godspell: A Musical Based on the Gospel According to St. Matthew*;
- *Desert Trail* и *The Desert Trail*.

Применив этот подход к 1000 фильмам в базе TMS, мы смогли сопоставить 884 с базой IMDb. Все сопоставились правильно, так что точность оказалась равна 100 %, а полнота – 88,4 % в предположении, что каждому фильму действительно есть соответствие. Отсюда также следует, что в ходе настройки алгоритма имело бы смысл разрешить более слабые условия сопоставления, поскольку при данном алгоритме и весах не произошло ни одной ошибки. И хотя цель этого упражнения не в том, чтобы оптимизировать сопоставление фильмов, анализ случаев, в которых алгоритм не смог найти соответствие, полезен, поскольку вскрывает факторы, которые можно было бы учесть при разработке алгоритма сопоставления записей в других предметных

областях. В табл. 4.6 приведено несколько фильмов, которые не удалось сопоставить.

Таблица 4.6. Анализ отказов сопоставления

Название/год в базе TMS	Название/год в базе IMDb	Описание
*M*A*S*H* (1970)	*MASH* (1970)	Причина в том, что сравнение *n*-грамм не прошло, т. к. все *n*-граммы одного из названий содержат звездочки, а ни в одной *n*-грамме другого названия этого символа нет.
9 to 5 (1980)	*Nine to Five* (1980)	Здесь для сопоставления названий необходимо приведение числительных к единому формату.
The Day the World Ended (1956)	*Day the World Ended* (1955)	В данном случае алгоритм не смог компенсировать наличие определяющего слова в начале из-за расхождений в других полях, конкретно дате выпуска. Проблему можно было бы решить удалением начальных артиклей *The*, *An* и *A*.
Quest for Fire (1981)	*La guerre du feu* (1981)	Иногда не помогает никакая нормализация. Этот случай должен быть разрешен редактором или же необходимо ввести в рассмотрение альтернативные названия.
Smokey and the Bandit (1977)	*Smokey and the Bandit* (1977)	В данном случае лучшая оценка была правильной, но у этого фильма имеется сиквел, *Smokey and the Bandit II*, который тоже получил оценку выше пороговой, поэтому сопоставление было отменено.
The Voyage of the Yes (1972)	*The Voyage of the Yes* (1972)	В данном случае один источник данных (TMS) классифицировал запись как фильм, а другой – как телевизионное шоу. Поэтому она вообще не рассматривалась при сопоставлении.

Эти примеры показывают, что даже при использовании самых лучших алгоритмов для успешного сопоставления необходима нормализация данных. Алгоритм много выиграл бы от таких видов нормализации, как приведение числительных к единому формату, устранение определяющих слов и рассмотрение альтернативных названий. Для создания действительно эффективных алгоритмов необходимо потратить немало усилий и позаботиться о том, чтобы на вход программы сопоставления попадали именно те данные, которые вас интересуют.

В этом разделе мы рассмотрели применение различных методов сравнения строк в ряде приложений. Для поддержки автозаполнения мы воспользовались префиксным поиском в Solr. Мы показали, что сравнение *n*-грамм в сочетании с редакционным расстоянием можно использовать для предложения альтернативных вариантов написания. И наконец, мы применили сравнение *n*-грамм, редакционное расстояние и точное сравнение для сопоставления записей о фильмах.

4.4. Резюме

Мы начали эту главу вопросом о том, что такое похожие строки – насколько неточным может быть неточное сравнение? Затем мы рассмотрели несколько подходов к неточному сравнению, дав тем самым формальное определение степени сходства строк. Среди них были меры, основанные только на сравнении одиночных символов, например, мера Жаккара; учитывающие порядок символов, например, редакционное расстояние; а также окна символов – мера Джаро-Винклера и *n*-граммное редакционное расстояние. Мы также показали, как с помощью сравнения префиксов и *n*-грамм можно эффективно отобрать кандидатов, для которых затем производятся более ресурсоемкие вычисления редакционного расстояния. Наконец, мы разработали приложения, в которых применяются все эти методы, используя Solr в качестве платформы. В следующей главе мы перейдем от сравнения строк к выделению информации, содержащейся в строках и документах.

4.5. Ресурсы

Aoe, Jun-ichi. 1989. «An efficient digital search algorithm by using a double-array structure». IEEE Transactions on Software Engineering, 15, no.9:1066–1077.

ГЛАВА 5. Распознавание имен людей, географических названий и других сущностей

В этой главе:

- Основные концепции распознавания именованных сущностей.
- Использование OpenNLP для поиска именованных сущностей.
- Вопросы производительности OpenNLP.

Имена людей, названия мест и различные объекты – имена существительные – играют важную роль в языке, являясь в предложении подлежащими, а зачастую и дополнениями. Поэтому при обработке текста бывает полезно попытаться выявить имена существительные и использовать это знание в приложении. Эта задача, которую обычно называют *идентификацией сущностей* или *распознаванием именованных сущностей* (named-entity recognition, NER), часто решается грамматическим анализатором или модулем разбиения на блоки, как мы видели в главе 2. И хотя анализатор очень хорош для понимания смысла предложения, в приложениях обработки текста нередко полезнее сосредоточить внимание на существительных, обозначающих конкретные экземпляры объектов, в частности, на именах собственных. Такие существительные называют *именованными сущностями*. К тому же, полный разбор предложения – ресурсоемкая задача, тогда как нахождение имен собственных необязательно потребляет много ресурсов.

Во многих ситуациях полезно не ограничиваться только именами людей и географическими названиями, но распознавать также числа и даты или время, например, июль 2007 или $50.35. С точки зрения приложения

для обработки текста, имена собственные кишмя кишат, но в то же время конкретные экземпляры имени собственного могут встречаться крайне редко. Например, возьмите любую новость (особенно не относящуюся к какой-нибудь знаменитости или высокопоставленному официальному лицу). Сколько в ее тексте имен собственных? Сколько людей, о которых вы никогда не слыхали? Сколько из них еще будут на слуху через шесть месяцев? Где и когда происходили описываемые события?

Понятно, что из контекста статьи видно, что некоторая последовательность слов является именем собственным, и, вероятно, имеются другие признаки, например, большая буква в начале, обращения типа г-н или г-жа, но как все формализовать, чтобы приложение смогло распознавать такие сущности? В этой главе мы сначала немного поговорим о принципах, лежащих в основе распознавания именованных сущностей, а затем посмотрим, как проект Apache OpenNLP позволяет научить приложение этому трюку. Мы остановимся также на вопросах производительности и расскажем о том, как настроить модель для конкретной предметной области. Но сначала рассмотрим, что можно делать с именованными сущностями.

Идентификация имен людей, названий организаций, мест и других именованных сущностей позволяет уяснить характер сущности и предпринять соответствующие действия. Например, располагая этой информацией, мы можем предложить дополнительные сведения о сущностях, рекомендовать сопутствующие материалы и в конечном итоге повысить интерес к своему сайту. Во многих крупных компаниях или организациях идентификация именованных сущностей возложена на специальных редакторов. В результате человек начинает читать одну статью, видит интересную ссылку, переходит к другой статье, а, посмотрев на часы, обнаруживает, что уже целый час бродит по сайту. Например, на рис. 5.1 показан сайт Yahoo!, где подчеркнута именованная сущность ссылка *Sarah Palin*, при щелчке по которой открывается всплывающее окно с дополнительной информацией об этой женщине, которая была кандидатом на должность вице-президента в избирательной кампании 2008 года. В нижней части этого окна владельцы сайта даже разместили рекламные объявления, относящиеся к именованной сущности, в надежде увеличить доход. Такое привлечение внимания пользователя бесценно для сайта (особенно получающего прибыль от демонстрации рекламы), это побуждает пользователя возвращаться на сайт снова и снова. Понятно, что решение этой задачи силами редактора – дело трудоемкое, и компании ищут способы автоматизировать процесс идентификации именованных сущностей полностью или частично.

Рис. 5.1. Фрагмент статьи на сайте Yahoo! News и результат щелчка по именованной сущности. Сара Палин (Sarah Palin) отмечена для включения в сводку главных новостей (Y! News Shortcut). Снимок экрана сделан 21.09.2008

К тому же, в отличие от ключевых слов, меток и прочих семантических представлений о содержании статьи, идея статьи на ту же тему, основанная на наличии или отсутствии некоторой сущности, – это очевидная связь, которая интуитивно понятна пользователю (в предположении, что вы правильно сопоставили записи, так что они относятся к одним и тем же сущностям, – см. главу 4).

В этой главе мы изучим, как можно автоматически идентифицировать имена и названия в тексте. Мы оценим точность и характеристики производительности одной популярной программы с открытым исходным кодом, решающей эту задачу. Это поможет решить, в каких случаях применять эту технологию. Мы также рассмотрим вопрос о настройке моделей с учетом особенностей конкретных данных. Даже если вы не собираетесь предоставлять информацию о распознанных сущностях непосредственно пользователям, все равно она может быть полезна для построения данных, привлекающих пользователей, например, о духе сайта или о 10 самых популярных личностях, упоминаемых на нем. Ну а теперь перейдем к деталям.

5.1. Различные подходы к распознаванию именованных сущностей

При решении задачи о распознавании именованных сущностей (NER) нас интересует выделение некоторых или всех упоминаний людей, мест, организаций, моментов времени и чисел (строго говоря,

не все вышеперечисленные сущности являются именами собственными, но для краткости мы будем относиться к ним как к таковым). Задача NER сводится к поиску ответов на вопросы где, когда, кто, как часто и сколько. Например, в предложении

The Minnesota Twins won the 1991 World Series,

система NER может распознать именованные сущности *Minnesota Twins*, *1991* и *World Series* (а, быть может, словосочетание *1991 World Series* будет распознано как одна именованная сущность). На практике не у всех систем одинаковые требования к выделению именованных сущностей. Например, маркетологам могут быть интересны названия производимых компанией товаров, чтобы знать, какие люди о них говорят. А историк, занимающийся реконструкцией событий по свидетельствам сотен очевидцев, хочет знать не только, какие люди были участниками события, но и точное время и местонахождение свидетеля в момент события.

5.1.1. Применение правил для распознавания имен и названий

Один из подходов к NER – использовать для идентификации сочетание списков и регулярных выражений. В этом случае формализуются некоторые базовые правила, касающиеся применения прописных букв и чисел, которые затем комбинируются с различными списками, например, распространенных имен и фамилий и хорошо известных мест (не говоря уже о днях недели, названиях месяцев и т. д.). Все это применяется к фрагменту текста. Такой подход был популярен, когда исследования в области распознавания именованных сущностей только начинались, но сейчас утратил позиции из-за сложностей сопровождения системы, а именно:

- поддержание списков в актуальном состояния – трудоемкая задача, а результату не хватает гибкости;
- переход на другие языки или предметные области может потребовать повторения значительной части проделанной работы;
- многие имена собственные имеют и другие значения (например, Will или Hope[1]). Иначе говоря, борьба с неоднозначностью – трудное дело;

[1] Соответственно «желание, завещание» и «надежда». – *Прим. перев.*

- многие названия образуются путем соединения других названий, например, Scottish Exhibition and Conference Center[2], и не всегда понятно, где кончается название;
- имена людей и названия мест часто совпадают, например: Washington (штат и Джордж) или Cicero[3] (античный оратор, город в штате Нью-Йорк или иное место);
- с помощью правил, основанных на регулярных выражениях, трудно моделировать зависимости между встречающимися в документе именами.

Следует отметить, что подходы на основе правил могут вполне прилично работать в хорошо изученных предметных областях, так что полностью отбрасывать их не стоит. В такой области, как сбор результатов измерений длины, где сами предметы встречаются редко, а количество единиц измерения ограничено, этот подход, скорее всего, будет жизнеспособен. Существует много полезных ресурсов, где описывается, как организовать такой процесс для различных типов сущностей. Базовые правила и некоторые ресурсы общего характера представлены во Всемирной книге фактов ЦРУ (https://www.cia.gov/library/publications/the-world-factbook/index.html) и в википедии (http://www.wikipedia.org). Доступны также словари имен собственных (для многих есть сетевые версии) и ресурсы, отражающие специфику конкретной предметной области, например база данных кинофильмов в Интернете. Все они могут быть эффективно использованы для достижения результатов приемлемого качества с минимумом трудозатрат.

5.1.2. Применение статистических классификаторов для распознавания имен и названий

Очень желательно иметь не столь хрупкий подход, который легко обобщался бы на другие предметные области и языки и не требовал создания больших списков (географических справочников), нуждающихся в сопровождении. Эта идея воплощена в статистических классификаторах. Типичный классификатор анализирует каждое слово в предложении и решает, является оно началом именованной сущности, продолжением уже начатой именованной сущности или вообще не

---

[2] Шотландский центр выставок и конференций. – *Прим. перев.*

[3] Цицерон. – *Прим. перев.*

является частью имени или названия. Объединяя эти предсказания, мы можем использовать классификатор для идентификации слов, составляющих имя или название.

Хотя большинство подходов к распознаванию имен на основе классификаторов так или иначе связаны с разметкой, существуют варианты распознавания сущностей различных типов. Например, можно воспользоваться разметкой или даже регулярными выражениями, чтобы выделить текст, содержащий имя или название любого вида, а на втором проходе разделить сущности по типам. Другой подход – одновременно распознавать сущности всех типов, предсказывая не только, где сущность начинается или продолжается, но и тип этой сущности. Еще один подход – построить отдельный классификатор для каждого типа сущностей и объединять их результаты для каждого предложения. Именно этот последний подход и применяется в программе, с которой мы будем работать в этой главе. Но ближе к концу мы рассмотрим и другие варианты и обсудим имеющиеся компромиссы.

Вне зависимости от того, какой подход к классификации применяется, классификатор необходимо обучить идентификации сущностей на наборе аннотированных человеком текстов. Перечислим некоторые преимущества такого подхода.

- Списки можно рассматривать как признаки, сделав, следовательно, лишь одним из источников информации.
- Для перехода на другой язык или предметную область нужны лишь минимальные изменения.
- Проще моделировать контекст в пределах одного предложения или документа.
- Если потребуется включить дополнительные тексты или новые признаки, классификатор можно будет переобучить.

Основной недостаток этого подхода – необходимость иметь аннотированные человеком данные. Если программист может написать набор правил и сразу же посмотреть на результат их применения к набору текстов, то для сколько-нибудь приличного функционирования классификатора его нужно обучить на тексте, содержащем примерно 30 000 слов. Впрочем, процесс аннотирования, хотя и утомительный, не требует специальных знаний и навыков составления правил. А полученные аннотации – ресурс, который можно наращивать и использовать многократно. При достаточном объеме обучающих данных классификатор по качеству распознавания приближается к человеку при том, что и человек решает задачу идентификации имен и назва-

ний не идеально. Хорошие NER-системы способны правильно распознавать более 90 % представленных в эксперименте образцов. На реальных данных показатели ниже, но большинство систем все же демонстрируют вполне пристойное для практического использования качество. Кроме того, хорошая система должна легко настраиваться и при необходимости обучаться распознаванию имен собственных. В идеале система должна также поддерживать инкрементное обновление путем предъявления новых примеров или контрпримеров. Памятуя об этих требованиях, мы в следующем разделе познакомимся со средствами распознавания именованных сущностей, предоставляемыми проектом OpenNLP.

5.2. Основы распознавания сущностей в OpenNLP

Проект OpenNLP, упоминавшийся в главе 2 и доступный для скачивания с сайта http://opennlp.apache.org, представляет собой комплект инструментов для решения многих типичных задач обработки естественных языков, в том числе частеречной разметки, грамматического анализа и, что особенно ценно в данной главе, распознавания именованных сущностей. Все эти инструменты распространяются по лицензии Apache Software License (ASL). Первоначально они были разработаны Томасом Мортоном и другими, но теперь являются проектом фонда Apache Software Foundation, как и Solr, и поддерживаются сообществом пользователей и разработчиков. Хотя для распознавания именованных сущностей есть несколько программных систем, большинство из них либо не поставляются в открытом виде, либо являются исследовательскими проектами, распространяемыми только по научно-исследовательским лицензиям или на условиях лицензии GPL, что для многих компаний неприемлемо. OpenNLP поставляется вместе с набором моделей, адаптированных для некоторых типичных сущностей. Проект активно развивается и хорошо поддерживается. По указанным причинам, а также потому, что мы сами хорошо знакомы с этим программным обеспечением, именно его мы будем рассматривать в качестве примера системы для распознавания именованных сущностей.

В состав OpenNLP входит несколько готовых моделей, настроенных на распознавание имен собственных и числительных и семантическую классификацию их по семи категориям. Ниже приведены как сами категории, так и примеры для каждой из них.

- Люди – Bill Clinton, Mr. Clinton, President Clinton.
- Географические названия – Alabama, Montgomery, Guam.
- Организации – Microsoft Corp., Internal Revenue Service, IRS, Congress.
- Даты – Sept. 3, Saturday (суббота), Easter (Пасха).
- Время – 6 minutes 20 seconds, 4:04 a.m., afternoon (полдень).
- Проценты – 10 percent, 45,5 percent, 37,5%.
- Денежные величины – $90 000, $35 billion, one euro, 36 pesos.

Пользователь может выбрать любое подмножество этих категорий в зависимости от требований конкретного приложения.

Далее в этом разделе мы изучим, как OpenNLP применяется для выделения сущностей вышеупомянутых категорий в тексте, а затем познакомимся с некоторыми инструментами, помогающими понять, что же было выделено. И закончим этот раздел обсуждением того, как воспользоваться вычисленными OpenNLP оценками для определения вероятности того, что распознавание выполнено правильно.

5.2.1. Нахождение имен и названий с помощью OpenNLP

Для начала посмотрим, как с помощью OpenNLP распознать имена людей. Нам придется написать несколько строк кода на Java.

Листинг 5.1. Распознавание имен с помощью OpenNLP[4]

```
String[] sentences = {
 "Former first lady Nancy Reagan was taken to a " +
     "suburban Los Angeles " +
 "hospital "as a precaution" Sunday after a " +
     "fall at her home, an " +
 "aide said. ",

 "The 86-year-old Reagan will remain overnight for " +
 "observation at a hospital in Santa Monica, California, " +
    "said Joanne " +
 "Drake, chief of staff for the Reagan Foundation."};

NameFinderME finder = new NameFinderME(
 new TokenNameFinderModel(new FileInputStream(getPersonModel())
```

Инициализировать новую модель для идентификации имен людей, читая сжатый двоичный файл en-ner-person.bin

[4] Бывшую первую леди Нэнси Рейган доставили в больницу в пригороде Лос-Анджелеса «на всякий случай» в воскресенье, после того как она упала у себя в доме, – по словам домработницы. 86-летняя Рейган останется на ночь для обследования в больнице города Санта-Моника в Калифорнии – сказала Джоан Дрейк, глава секретариата фонда Рейгана. – *Прим. перев.*

```
)
  );
  Tokenizer tokenizer = SimpleTokenizer.INSTANCE;
  for (int si = 0; si < sentences.length; si++) {
   String[] tokens = tokenizer.tokenize(sentences[si]);
   Span[] names = finder.find(tokens);
   displayNames(names, tokens);
  }

  finder.clearAdaptiveData();
```

Инициализировать лексический анализатор, который разбивает предложения на отдельные слова и символы

Представить предложение в виде массива лексем

Распознать имена в предложении и вернуть индексы соответствующих им лексем

Очистить структуры данных, в которых хранятся слова документа и информация о том, какие из них являются именами людей

Здесь мы сначала создаем документ, содержащий два предложения, а затем инициализируем объект класса `NameFinderME` и лексический анализатор для работы с этим документом. Конструктору класса `NameFinderME` передается модель для идентификации одного вида именованных сущностей (в данном случае имен людей), которая читается из файла модели, входящего в состав OpenNLP. Затем каждое предложение разбивается на лексемы и из него выделяются и показываются пользователю имена. После обработки всех предложений вызывается метод `clearAdaptiveData()`. Он говорит объекту `NameFinderME`, что все прочитанные до сих пор данные – как исходные, так и созданные в процессе обработки – можно очистить. По умолчанию класс `NameFinderME` хранит информацию о том, было ли некоторое слово распознано как часть имени раньше, поскольку это может служить основанием и дальше рассматривать его как часть имени. Обращение к методу `clearAdaptiveData()` очищает этот кэш.

В этом примере показано, что объект `NameFinderME` обрабатывает по одному предложению за раз. Хотя явно это и не оговаривается, такое решение препятствует ошибочному распознаванию имен, пересекающих границу предложения. В общем случае, лучше, чтобы распознаватель имен обрабатывал минимально возможные единицы текста, не приводящие к расщеплению имени. Связано это с тем, что реализация OpenNLP в действительности рассматривает три альтернативных набора имен для каждой обрабатываемой единицы текста. Если обрабатывается весь документ, то мы получаем три варианта для документа в целом, а если одно предложение – то по три варианта для каждого предложения.

На вход методу `NameFinderME.find()` подается последовательность лексем. Это означает, что каждое обрабатываемое предложение сначала нужно подвергнуть лексическому анализу. В данном случае

мы воспользовались анализатором, предлагаемым OpenNLP, который выделяет лексемы, исходя из классов символов. Поскольку от лексического анализатора зависит, как метод `find()` видит предложение, при распознавании имен важно применять ту же процедуру разбиения на лексемы, которая применялась при обучении модели. В разделе 5.5 мы обсудим обучение новых моделей распознавания именованных сущностей с другими лексическими анализаторами.

5.2.2. Интерпретация имен, распознанных OpenNLP

Метод `NameFinderME.find()` возвращает массив интервалов (span), определяющих местоположение имен, распознанных во входном предложении. В типе `Span` хранится индекс первой лексемы имени (его возвращает метод `getStart()`) и индекс лексемы, непосредственно следующей за последней лексемой имени (его возвращает метод `getEnd()`). В данном случае интервалы используются для представления смещений лексем от начала массива, но в OpenNLP этот тип данных применяется и для представления смещений символов. Показанный ниже код печатает все распознанные имена в виде последовательности лексем.

Листинг 5.2. Вывод имен с помощью OpenNLP

```
private void displayNames(Span[] names, String[] tokens) {
 for (int si = 0; si < names.length; si++) {
  StringBuilder cb = new StringBuilder();
  for (int ti = names[si].getStart();
       ti < names[si].getEnd(); ti++) {
    cb.append(tokens[ti]).append(" ");
  }
  System.out.println(cb.substring(0, cb.length() - 1));
  System.out.println("ttype: " + names[si].getType());
 }
}
```

Обходим имена

Обходим лексемы, составляющие имена

Удаляем лишние концевые пробелы и печатаем

OpenNLP предоставляет также вспомогательный метод для преобразования интервала в строку, представляющую имя, — `Span.spansToStrings()`. Его применение показано в листинге 5.3.

Чтобы увидеть имя безотносительно лексем, можно сопоставить имени смещения составляющих его символов, как показано ниже. В данном случае нас не интересуют строковые представления лексем, но с помощью метода `tokenizePos()` мы просим анализатор вернуть

описывающие смещения символов интервалы для каждой лексемы. Это позволяет определить местоположение имени в исходном предложении.

Листинг 5.3. Вывод имен с помощью интервалов

```
for (int si = 0; si < sentences.length; si++) {
 Span[] tokenSpans = tokenizer.tokenizePos(sentences[si]);

String[] tokens = Span.spansToStrings(tokenSpans, sentences[si]);
 Span[] names = finder.find(tokens);

for (int ni = 0; ni < names.length; ni++) {
  Span startSpan = tokenSpans[names[ni].getStart()];
  int nameStart = startSpan.getStart();

  Span endSpan = tokenSpans[names[ni].getEnd() - 1];
  int nameEnd = endSpan.getEnd();

  String name = sentences[si].substring(nameStart, nameEnd);
  System.out.println(name);
```

Обходим предложения

Разбиваем на лексемы; возвращаем смещения символов (интервалы)

Преобразуем интервалы в строки

Распознаем имена; возвращаем смещения соответствующих лексем

Вычисляем индекс первого символа имени

Вычисляем индекс символа, следующего за последним символом имени

Формируем строку, представляющую имя

5.2.3. Фильтрация имен на основе вероятности

В OpenNLP используется вероятностная модель, позволяющая определить вероятность правильного распознавания имени. Это особенно полезно тогда, когда требуется отбросить некоторые возвращенные распознавателем имена как ошибочные. И хотя не существует способа автоматически решить, какие имена распознаны ошибочно, в общем случае можно считать, что имена, которым модель присвоила низкие вероятности, скорее всего, сомнительны. Чтобы получить вероятность для конкретного имени, вызовите метод `NameFinderME.getProbs()` после обработки предложения, как показано в листинге ниже. В возвращенном массиве вероятности соответствуют по индексам именам, которыми они представлены во входном массиве интервалов.

Листинг 5.4. Вычисление вероятностей имен

```
for (int si = 0; si < sentences.length; si++) {

String[] tokens = tokenizer.tokenize(sentences[si]);
```

Обходим каждое предложение

Разбиваем предложение на лексемы

```
Span[] names = finder.find(tokens);
double[] spanProbs = finder.probs(names);
}
```

Распознаем имена, возвращаем смещения лексем

Возвращаем вероятности, ассоциированные с каждым именем

Для фильтрации имен нужно затем определить специфическую для приложения пороговую вероятность, ниже которой имя исключается из рассмотрения.

В этом разделе мы видели, как распознавать имена одного типа с помощью OpenNLP, как интерпретировать структуры данных, которые в OpenNLP используются для указания положения имен в тексте, и как определять, какие имена, скорее всего, точны. В следующем разделе мы рассмотрим, как распознавать разнородные имена, и узнаем, как OpenNLP на самом деле определяет наличие или отсутствие имени в тексте.

5.3. Подробнее о распознавании сущностей в OpenNLP

Освоив основы, давайте рассмотрим более сложные случаи, которые вполне могут встретиться при построении реальных систем. В разделе 5.1 мы видели, что к идентификации именованных сущностей есть несколько походов. У разобранных выше примеров есть существенное ограничение: они умеют работать только с именованными сущностями одного вида. В этом разделе мы посмотрим, как с помощью OpenNLP можно распознавать разнородные сущности в одном предложении, и выясним, какая информация необходима OpenNLP для идентификации типа сущности.

5.3.1. Распознавание разнородных сущностей в OpenNLP

В OpenNLP для каждого типа сущностей существует независимая модель, позволяющая распознавать сущности этого типа. Преимущества такого решения в том, что вы можете использовать только те модели, которые нужны приложению, и добавлять собственные модели для новых сущностей. Кроме того, различные модели могут распознавать сущности в перекрывающихся участках текста, как показано ниже[5]:

[5] Майкл Вик, бывший нападающий команды Атланта Фальконс, отбывает 23-месячный срок заключения в тюрьме строгого режима в Ливенворте, штат Канзас.

```
<person> Michael Vick </person>, the former <organization> <location>
Atlanta </location> Falcons </organization> quarterback, is serving a
23-month sentence at maximum-security prison in <location> Leavenworth
</location>, <location> Kansas </location>.
```

Здесь слово *Atlanta* помечено и как географическое название, и как часть организации.

Недостаток же состоит в том, что приходится объединять результаты всех моделей. В этом разделе мы рассмотрим некоторые способы борьбы с этой проблемой. Наличие отдельной модели для каждого типа сущностей оказывает влияние на качество работы и на обучение. Об этих последствиях мы будем говорить в разделах 5.4 и 5.5.

Поскольку все модели независимы, использование нескольких моделей сводится просто к обработке одного и того же предложения каждой из них, после чего следует объединить результаты – а это уже не так просто. В листинге 5.5 мы объединяем имена и названия, полученные от трех моделей. Для этой цели мы создали вспомогательный класс `Annotation`, в котором будем хранить интервал текста, занятый сущностью, а также ее вероятность и тип.

Листинг 5.5. Прогон нескольких моделей для одного текста

```
String[] sentences = {
  "Former first lady Nancy Reagan was taken to a " +
     "suburban Los Angeles " +
  "hospital "as a precaution" Sunday after a fall at " +
     "her home, an " +
  "aide said. ",
  "The 86-year-old Reagan will remain overnight for " +
  "observation at a hospital in Santa Monica, California, " +
     "said Joanne " +
  "Drake, chief of staff for the Reagan Foundation."};
NameFinderME[] finders = new NameFinderME[3];
String[] names = {"person", "location", "date"};
for (int mi = 0; mi < names.length; mi++) {
```

Инициализируем новую модель для распознавания имен людей, географических названий и дат, для чего читаем двоичные сжатые представления моделей из файлов en-ner-person.bin, en-ner-location.bin, en-ner-date.bin

```
  finders[mi] = new NameFinderME(new TokenNameFinderModel(
    new FileInputStream(
      new File(modelDir, "en-ner-" + names[mi] + ".bin")
    )));
}
```

Получаем ссылку на лексический анализатор, который разбивает предложение на слова и символы

```
Tokenizer tokenizer = SimpleTokenizer.INSTANCE;
```

```
for (int si = 0; si < sentences.length; si++) {
 List<Annotation> allAnnotations = new ArrayList<Annotation>();
 String[] tokens = tokenizer.tokenize(sentences[si]);
 for (int fi = 0; fi < finders.length; fi++) {
  Span[] spans = finders[fi].find(tokens);
  double[] probs = finders[fi].probs(spans);
  for (int ni = 0; ni < spans.length; ni++) {
   allAnnotations.add(
    new Annotation(names[fi], spans[ni], probs[ni])
   );
  }
 }
removeConflicts(allAnnotations);
```

Обходим каждое предложение

Преобразуем предложение в массив лексем

Перебираем все распознаватели сущностей (имен, названий и дат)

Распознаем сущности в предложении и возвращаем смещения соответствующих им лексем

Получаем вероятности правильного распознавания

Собираем воедино сущности, идентифицированные каждым распознавателем

Разрешаем перекрывающиеся сущности, отдавая предпочтение более вероятным

Объединение результатов всех трех моделей вызывает трудности, только когда сущности перекрываются. Что означает перекрытие в конкретном случае, зависит от приложения. Чтобы объединить результаты, мы должны ответить на следующие вопросы.

- Разрешено ли разным моделям распознавать один и тот же отрезок текста, как именованную сущность? Обычно нет.
- Разрешено ли более короткому имени встречаться в более длинном? Обычно да.
- Могут ли имена перекрываться, но при этом содержать еще различный дополнительный текст? Обычно нет.
- Какой критерий применять для разрешения конфликта имен? Обычно вероятностный.

Ниже приведена реализация, в которой подразумеваются указанные выше умолчания: имена должны располагаться в разных отрезках и могут перекрываться, но в таком случае одно имя должно быть частью другого.

Листинг 5.6. Разрешение конфликта имен

```
private void removeConflicts(List<Annotation> allAnnotations) {
 java.util.Collections.sort(allAnnotations);
 List<Annotation> stack = new ArrayList<Annotation>();
 stack.add(allAnnotations.get(0));
 for (int ai = 1; ai < allAnnotations.size(); ai++) {
  Annotation curr = (Annotation) allAnnotations.get(ai);
```

Инициализируем стек для запоминания ранее встретившихся имен

Сортируем имена сначала по возрастанию начального индекса интервала, а затем по убыванию конечного индекса

Обходим имена

```
    boolean deleteCurr = false;
    for (int ki = stack.size() - 1; ki >= 0; ki--) {
      Annotation prev = (Annotation) stack.get(ki);
      if (prev.getSpan().equals(curr.getSpan())) {
        if (prev.getProb() > curr.getProb()) {
          deleteCurr = true;
          break;
        } else {
          allAnnotations.remove(stack.remove(ki));
          ai--;
        }
      } else if (prev.getSpan().intersects(curr.getSpan())) {
        if (prev.getProb() > curr.getProb()) {
          deleteCurr = true;
          break;
        } else {
          allAnnotations.remove(stack.remove(ki));
          ai--;
        }
      } else if (prev.getSpan().contains(curr.getSpan())) {
        break;
      } else {
        stack.remove(ki);
      }
    }
    if (deleteCurr) {
      allAnnotations.remove(ai);
      ai--;
      deleteCurr = false;
    } else {
      stack.add(curr);
    }
  }
}
```

Обходим все элементы стека

Проверяем, совпадает ли интервал одного имени с интервалом другого и, если да, удаляем менее вероятное имя

Обновляем индекс имени после удаления, чтобы компенсировать предложение a++ в конце цикла

Проверяем, пересекается ли интервал одного имени с интервалом другого и, если да, удаляем менее вероятное имя

Обновляем индекс имени после удаления, чтобы компенсировать предложение a++ в конце цикла

Проверяем, находится ли интервал одного имени полностью внутри интервала другого и, если да, выходим из цикла

Проверяем, находится ли интервал одного имени полностью внутри интервала другого и, если да, выходим из цикла

Проверяем, расположен ли интервал одного имени после интервал другого имени, и, если да, удаляем предыдущее имя из стека

Вычислительная сложность этого алгоритма объединения линейно зависит от длины предложения, но, поскольку мы допускаем появление одного имени внутри другого, необходим второй цикл для обработки стека вложенных имен. Размер этого стека не может превысить количество используемых типов именованных сущностей, поэтому затрачиваемое на его выполнение время можно считать постоянным. Итак, мы рассмотрели принципиальные основы использования нескольких моделей и продемонстрировали пример. Теперь обратимся к техническим деталям: как в OpenNLP распознаются имена.

5.3.2. Под капотом: как в OpenNLP распознаются имена

Если спросить обывателя «Как работает телевизор?», в ответ можно услышать: «Направляешь на него пульт и нажимаешь красную кнопку...». До сих пор мы говорили о том, как использовать средства распознавания именованных сущностей, имеющиеся в OpenNLP, т. е. давали примерно такой же ответ. В этом разделе мы увидим, как на самом деле решается задача распознавания имен. Эти знания пригодятся нам в разделах 5.4 и 5.5 при обсуждении вопросов качества работы и настройки.

В OpenNLP распознавание имен рассматривается как задача разметки – по аналогии с той, что будет обсуждаться в главе 7. Требуется пометить каждую лексему одной из трех меток:

- *start* – эта лексема начинает новое имя;
- *continue* – эта лексема является продолжением существующего имени;
- *other* – эта лексема не является частью имени.

В табл. 5.1. показано, как выглядит такая разметка для типичного предложения.

Таблица 5.1. Предложение, размеченное для распознавания именованных сущностей

0	1	2	3	4	5	6	7	8	9	10	11
“	It	is	a	familiar	story	,	“	Jason	Willaford	said	.
other	other	other	other	other	other	other	other	start	continue	other	other

Если соединить метки start со следующими за ними метками continue, то из классифицированных лексем получится множество интервалов – объектов `Span`. В OpenNLP используется пакет статистического моделирования, который строит модель, определяющую, как расставлять метки. В этой модели имеется набор *признаков*, зашитых в код, с помощью которых предсказывается наиболее вероятный результат. Признаки подобраны так, чтобы выделять имена собственные, различные виды числовых строк с учетом окружающего контекста и ранее принятых решений о разметке. Для распознавания именованных сущностей в OpenNLP используются следующие признаки.

1. Размечаемая в данный момент лексема.
2. Предыдущая лексема.

3. Лексема на 2 позиции левее текущей.
4. Последующая лексема.
5. Лексема на 2 позиции правее текущей.
6. Класс размечаемой лексемы.
7. Класс предыдущей лексемы.
8. Класс лексемы на 2 позиции левее текущей.
9. Класс последующей лексемы.
10. Класс лексемы на 2 позиции правее текущей.
11. Размечаемая лексема и ее класс.
12. Предыдущая лексема и ее класс.
13. Лексема на 2 позиции левее текущей и ее класс.
14. Последующая лексема и ее класс.
15. Лексема на 2 позиции правее текущей и ее класс.
16. Предсказанный результат для предыдущей лексемы или null.
17. Предсказанный результат для лексемы 2 позиции левее текущей или null.
18. Размечаемая и предыдущая лексема.
19. Размечаемая и последующая лексема.
20. Классы размечаемой и предыдущей лексем.
21. Классы размечаемой и последующей лексем.
22. Результат, ранее присвоенный такой же строке, как размечаемая лексема, или null.

Многие из этих признаков основаны на самой размечаемой лексеме и ее ближайших соседях, но некоторые – на классе лексемы. Класс лексемы определяется ее базовыми характеристиками, например, состоит ли она исключительно из строчных букв.

Признаки позволяют принять решение о том, какие лексемы образуют именованную сущность и каков тип этой сущности, на основе встретившихся слов, классов этих слов и решений, принятых ранее при анализе данного предложения или всего документа. Сами слова важны и аналогичны спискам в подходе на основе правил. Если в обучающих данных некоторое слово достаточно много раз аннотировано как сущность, то, используя признак 1, классификатор может просто запомнить это слово. Признак 6, как и признак 1, анализирует только само размечаемое слово, но интересуется не словом как таковым, а классом лексемы.

Ниже перечислены классы лексем для распознавания именованных сущностей.

1. Лексема содержит только строчные буквы.
2. Лексема содержит две цифры.

3. Лексема содержит четыре цифры.
4. Лексема содержит число и букву.
5. Лексема содержит число и дефис.
6. Лексема содержит число и знак обратной косой черты.
7. Лексема содержит число и запятую.
8. Лексема содержит число и точку.
9. Лексема содержит число.
10. Лексема содержит одну прописную букву.
11. Лексема содержит только прописные буквы.
12. Начальные буквы лексемы прописные.
13. Прочее.

Перечисленные выше классы лексем предназначены для предсказания типов некоторых сущностей. Например, класс лексемы 3 позволяет предложить, что это год, например, 1984. Классы лексем 5 и 6 также соответствуют типичным форматам дат, а классы 7 и 8, скорее, характерны для денежных величин. Признаки 2–15 позволяют учесть контекст, в котором встречается слово, поэтому неоднозначное слово типа *Washington* с большей вероятностью будет распознано как географическое название в контексте «in Washington»[6] и как имя человека в контексте «Washington said»[7]. Благодаря признакам 16 и 17, модель может учесть тот факт, что метки continue следуют за метками start, а признак 18 позволяет модели заключать, что если слово ранее было размечено как часть имени человека, то и последующие вхождения этого слова тоже могут быть частью имени. И хотя ни один из признаков сам по себе не дает исчерпывающего предсказания, их комбинация с эмпирически подобранными весами обычно дает возможность распознать именованные сущности в OpenNLP.

Признаки нацелены на улавливание той информации, которая необходима для распознавания поддерживаемых OpenNLP именованных сущностей. Может статься, что вашему приложению этих сущностей хватит. Поскольку автоматизированные методы распознавания имен никогда не будут совершенными (как, впрочем, и ручные), возникает вопрос, а достаточно ли они хороши для целей вашего приложения? В следующем разделе мы рассмотрим, насколько качественно модели, поставляемые в составе OpenNLP, распознают сущности и какова их производительность. Эти характеристики помогут решить, можно ли использовать это ПО «как есть» и в какого рода приложениях. Возможно, вам потребуется распознавать и другие сущности. В таком

[6] в Вашингтоне.

[7] Вашингтон говорил.

случае вышеперечисленных признаков может не хватить для улавливания той информации, которая необходима для их распознавания.

5.4. Качество работы OpenNLP

Мы рассмотрим три аспекта распознавания именованных сущностей. Во-первых, это качество лингвистических аннотаций, а конкретно сущностей, идентифицируемых компонентами OpenNLP. Как уже было сказано, лингвистический анализ не бывает совершенным, но это не должно останавливать нас от изучения того, насколько хорошо система имитирует человека. Имея эту информацию, мы можем оценить, достаточно ли точны предлагаемые модели для нужд вашего приложения или же необходимо дополнительное обучение или настройка.

Во-вторых, нас интересует быстродействие. Мы обсудим некоторые оптимизации, которые OpenNLP применяет для повышения эффективности, и оценим скорость применения различных моделей к тексту. И наконец, мы рассмотрим вопрос о том, сколько памяти необходимо для работы распознавателя имен. Как уже отмечалось, модели распознавания именованных сущностей независимы, так что вы можете использовать только те, что вам нужны. Отчасти это объясняется высокой потребностью моделей в памяти. Мы дадим точные оценки потребления памяти, а также покажем, как можно существенно снизить объем памяти, необходимой для загрузки следующей модели. Эти сведения пригодятся при решении о том, применима ли эта технология к конкретному приложению и, если да, то в каком режиме: оперативном или пакетном.

5.4.1. Качество результатов

Лингвистическое качество системы распознавания именованных сущностей зависит от данных, на которых она была обучена. В OpenNLP для обучения моделей применяются данные, подготовленные для задачи MUC-7 (http://www-nlpir.nist.gov/related_projects/muc/proceedings/muc_7_toc.html). Этот набор данных часто применяется в исследованиях по системам распознавания именованных сущностей. В этой задаче имеются конкретные критерии того, что считать именем человека, названием организации, географическим названием и другими сущностями. На первый взгляд, это достаточно четкие категории, но в реальных текстах встречаются случаи, когда выбор неочевиден и для разрешения проблемы требуются инструкции. Так,

предметы, имеющие собственные имена, например Space Shuttle Discovery (космический челнок Дискавери), как правило, не классифицируются, но вот аэропорты, согласно инструкциям, считаются географическими названиями. В процессе обучения, направленном на распознавание таких категорий, важно не столько конкретное решение в пограничных случаях, сколько единообразное аннотирование, чтобы модель могла их выучить.

Для оценки моделей, входящих в состав OpenNLP, мы использовали обучающий и тестовый наборы, подготовленные для MUC-7. Модели OpenNLP не обучались на данных, присутствующих в тестовом наборе, поэтому можно проверить, как они работают на ранее не предъявлявшемся тексте. Результаты оценки показаны в табл. 5.2.

Таблица 5.2. Оценка качества аннотаций, полученных с помощью OpenNLP

Набор данных	Точность	Полнота	F-мера
обучающий muc7	87	90	88,61
тестовый muc7	94	75	83,481

Точность показывает, как часто распознанное системой имя оказывалось правильным. *Полнота* – это мера, показывающая, какая доля предъявленных имен вообще была распознана. *F-мера* – это взвешенное среднее гармоническое точности и полноты. В данном случае при вычислении F-меры были взяты одинаковые веса. Как видно из таблицы, система смогла распознать не менее 75 % сущностей и ошиблась примерно в 10 % случаев.

5.4.2. Производительность

Второй аспект качества работы – это производительность. В разделе 5.1.2 мы видели, что для распознавания имен модель помечает каждое слово в предложении меткой start, continue или other. Это означает, что требуется принимать решения для каждой лексемы и каждого типа именованной сущности. И вообще-то нужно еще умножить на три, потому что для каждой обрабатываемой единицы текста рассматривается до трех альтернативных наборов имен. По мере добавления в систему новых моделей затраты могут стать неприемлемо высокими.

OpenNLP снижает затраты на обработку двумя способами. Во-первых, вычисленные вероятности кэшируются: если признаки, выработанные для предсказания результата, одинаковы для всех трех альтернативных наборов имен, то распределение вероятностей вычисляется

только один раз и затем кэшируется. Во-вторых, кэшируется и сам процесс выработки признаков. Так как одни и те же признаки используются во всех моделях, а предложения обрабатываются по одному за раз, то признаки уровня предложения, которые не зависят от предыдущих решений, принятых моделью, вычисляются только один раз и кэшируются. В результате получается, что скорость обработки при добавлении моделей возрастает не линейно, а медленнее, хотя, конечно, чем меньше моделей, тем быстрее. Это наглядно видно из графика производительности, показанного на рис. 5.2.

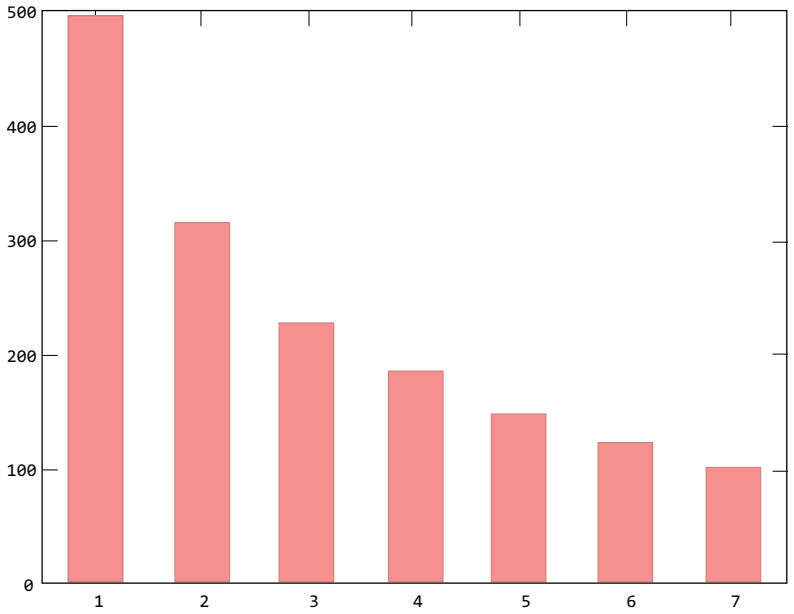

Рис. 5.2. Зависимость количества предложений, обрабатываемых распознавателем имен в секунду, от числа распознающих моделей

В разделе 5.5 мы опишем альтернативную модель, которая частично жертвует гибкостью, зато распознает все сущности так же быстро, как сущность какого-то одного вида.

5.4.3. Потребление памяти в OpenNLP

В этом разделе мы рассмотрим, какие требования система распознавания именованных сущностей OpenNLP предъявляет к памяти. Выше мы видели, что работа одновременно с несколькими моделями

обходится недаром. Процесс, в котором используется только модель имен людей, потребляет примерно 68 МБ. Часть этой памяти занята кодом и виртуальной машиной Java, но около 54 МБ приходится на модель. В состав модели входят признаки, выходные результаты и значения параметров, но большая часть отведена под хранение лексем, составляющих признаки. Связано это с тем, что наши признаки лексические, так что каждое слово, встречающееся в обучающих данных чаще некоторого порога, будет храниться в модели в виде признака. Поскольку лексические признаки потенциально могут комбинироваться с другими лексическими и нелексическими признаками, общее число хранящихся в модели признаков может оказаться весьма большим. Для загрузки всех моделей требуется примерно 400 МБ памяти.

В предыдущем разделе мы упомянули, что все модели именованных сущностей пользуются одними и теми же признаками. Если они обучены на одних и тех же данных, то будут содержать одинаковые признаки, а разными будут только параметры признаков, поскольку для одних именованных сущностей более важны одни факторы, а для других – другие. И даже если модели обучены на разных данных, все равно между признаками имеется существенное перекрытие, поскольку множества слов в двух фрагментах текста, как правило, сильно пересекаются. Если бы удалось каким-то образом организовать общую память для всех признаков, то при использовании нескольких моделей можно было бы ожидать заметного снижения общего потребления памяти. В Java это можно сделать, в частности, с помощью метода `String.intern()`. Этот метод возвращает каноническое представление строки и применяется для реализации пула строк. С его помощью можно гарантировать, что все экземпляры одной строки ссылаются на один и тот же объект в памяти.

В исходном коде к настоящей книге имеется класс чтения модели, который пользуется методом `String.intern()` для достижения указанного эффекта. В примере ниже показано, как использовать этот класс при работе с несколькими моделями именованных сущностей.

Листинг 5.7. Использование пула строк для сокращения потребления памяти при распознавании именованных сущностей

```
String[] names = {"person","location","date"};
NameFinderME[] finders = new NameFinderME[names.length];
for (int mi = 0; mi < names.length; mi++) {
 finders[mi] = new NameFinderME(
```

Инициализируем распознаватели имен людей, географических названий и дат

```
    new PooledTokenNameFinderModel(
      new FileInputStream(
        new File(modelDir, "en-ner-" + names[mi] + ".bin"))));
  }
```

Используем модель с пулом строк для сокращения потребления памяти

При таком подходе все семь моделей можно загрузить примерно в 225 МБ памяти, сэкономив 175 МБ. И поскольку отображение признаков на общее представление производится во время загрузки модели, эффективность применения модели к тексту не страдает.

Разобравшись с качеством, быстродействием и потреблением памяти OpenNLP, давайте поднимемся на уровень выше и посмотрим, как можно настроить OpenNLP на свою предметную область. В следующем разделе мы объясним, что значит обучить собственную модель, и рассмотрим другие настройки, которые могут понадобиться при развертывании приложения.

5.5. Настройка OpenNLP для распознавания сущностей в новой предметной области

Во многих ситуациях готовых моделей, входящих в состав OpenNLP, достаточно для приложения или предметной области. Но бывает так, что требуется обучить собственную модель. В этом разделе мы опишем, как обучать модели в OpenNLP и как изменить признаки для предсказания сущностей, а также расскажем о другом способе использования OpenNLP для распознавания имен, который в некоторых случаях имеет преимущества.

5.5.1. Зачем и как обучают модель

Для обучения собственной модели есть немало причин. Например, требуется распознавать новый тип сущностей, скажем, транспортные средства. Или нужно распознавать имена людей или другие сущности, уже поддерживаемые OpenNLP, но ваша предметная область настолько своеобразна, что модели, поставляемые в составе OpenNLP, не годятся. Или, быть может, у вас особый случай, когда определение имени человека отличается от принятого в модели OpenNLP. Кроме того, если вы хотите распознавать новый тип сущности или работаете в нестандартной предметной области, то, возможно, модели понадобятся новые признаки. Наконец, может случиться, что лексический

анализатор, используемый в OpenNLP, не подходит к предметной области или не согласуется с последующей обработкой, и тогда необходимо обучить модель с другим анализатором.

При обучении новой модели труднее всего найти или создать обучающие данные в объеме, достаточном для статистического моделирования. Конечно, есть несколько общедоступных наборов данных, но коль скоро вы создаете модель для сущности нового типа, то почти наверняка придется аннотировать собственный текст. И хотя аннотирование отнимает много времени, для него не требуется специальных навыков, и зачастую это обходится дешевле, чем нанять программиста для построения набора правил. Аннотированный набор данных можно повторно использовать для других типов моделей, так что при усовершенствовании моделирования (быть может, за счет идентификации с признаками, имеющими большую предсказательную силу) общее качество системы распознавания именованных сущностей можно повысить без дополнительных затрат на аннотирование. Типичный обучающий набор данных для распознавания именованных сущностей включает не менее 10–15 тысяч слов. Кроме того, необходимо иметь дополнительные аннотированные данные для оценки успешности обучения и гарантий того, что после изменения системы качество ее работы повысилось.

Если ваша цель – улучшить работу модели, поставляемой вместе с OpenNLP, в новой предметной области, то имеются и другие возможности. Хотя в состав OpenNLP входит набор моделей, данные, на которых эти модели обучались, не поставляются из-за лицензионных ограничений. У этой проблемы есть три решения.

- *Пользоваться только данными из предметной области* – при наличии достаточных данных это, скорее всего, позволит получить более точную модель, поскольку она «заточена» под предметную область.
- *Построить отдельную модель и объединить результаты* – это похоже на то, что мы делали выше в этом разделе, когда комбинировали аннотации разных типов, только в данном случае оба классификатора предсказывают один и тот же класс. Если точность обоих классификаторов высока, то их объединение должно повысить и полноту.
- *Использовать результаты модели OpenNLP в качестве обучающих данных* – этот метод позволяет увеличить объем пригодных для использования обучающих данных. Лучше всего он работает в сочетании с правкой со стороны человека. Модели

OpenNLP обучены на текстах с лент новостей, поэтому оптимальный результат получается при применении их к аналогичному тексту.

Каковы бы ни были причины для настройки OpenNLP, следующие далее разделы помогут вам понять, как подойти к этой процедуре.

5.5.2. Обучение модели OpenNLP

Итак, вы решили, когда и почему хотите обучить новую модель распознавания именованных сущностей, подготовили аннотированные данные, и теперь самое время взглянуть, как производится обучение. OpenNLP предоставляет код обучения в методе `NameFinderME.main()`, который поддерживает различные параметры, в том числе задание кодировки символов. В листинге ниже приведен сокращенный и немного реорганизованный вариант этого кода.

Листинг 5.8. Обучение модели распознавания именованных сущностей в OpenNLP

```
File inFile = new File(baseDir,"person.train");
NameSampleDataStream nss = new NameSampleDataStream(
 new PlainTextByLineStream(
   new java.io.FileReader(inFile)));

int iterations = 100;
int cutoff = 5;
TokenNameFinderModel model = NameFinderME.train(
 "en", // язык
 "person", // тип сущности
 nss,
 (AdaptiveFeatureGenerator) null,
 Collections.<String,Object>emptyMap(),
 iterations,
 cutoff);

File outFile = new File(destDir, "person-custom.bin");
FileOutputStream outFileStream = new FileOutputStream(outFile);
model.serialize(outFileStream);
```

Создаем поток образцов сущностей, основываясь на аннотированных данных

Обучаем модель

Сохраняем модель в файле

В первых двух строчках задается файл с обучающими данными и создается объект типа `NameSampleStream`, который предоставляет простой интерфейс, позволяющий обойти последовательность объектов `NameSample`. Класс `NameSample` определяет простую структуру данных для хранения интервалов и лексем, составляющих именованные сущности. Класс `NameSampleDataStream` реализует интерфейс `NameSampleStream` и производит разбор, предполагая, что в каждой

строке находится одно предложение. Каждая строка состоит из лексем, разделенных пробелами, а имена выделены метками <START> и <END>, также обрамленными с обеих сторон пробелами:

```
"It is a familiar story " , <START> Jason Willaford <END> said .
```

Этот формат поддержан изначально, но несложно поддержать и другие, написав класс для их разбора, который реализовывал бы интерфейс `NameSampleStream`.

Процедура обучения принимает несколько параметров. Первые два задают язык и тип порождаемой модели. Следующий – объект `NameSampleDataStream`, из которого мы будем получать образцы `NameSample`, которые преобразуются в поток событий, используемых для обучения модели. В разделе 5.1.2 мы видели, что OpenNLP интерпретирует каждое имя как цепочку меток start/continue/other, исходя из набора контекстных признаков.

Следующие два параметра метода `train` присутствуют только потому, что их требует интерфейс. Задание `null` вместо объекта `AdaptiveFeatureGenerator` заставляет `NameFinderME` использовать подразумеваемый по умолчанию набор генераторов признаков, который эффективно распознает именованные сущности. А пустой словарь ресурсов мы задаем, потому что не включаем в генерируемую модель никаких дополнительных ресурсов.

Последние два параметра метода `train` – число итераций и порог отсечения признака. Порог говорит, сколько раз (как минимум) должен встретиться признак, чтобы быть включенным в модель. По умолчанию предполагается, что признак, встречающийся менее пяти раз, в модель не включается. Это необходимо для ограничения размера модели и снижения потенциального шума, поскольку модель не может точно оценить значения параметров для признаков, которые встречаются слишком редко. Если задать чрезмерно малое значение этого параметра, то модель будет плохо работать на данных, которые раньше не видела. Однако если набор данных мал, то при таком пороге отсечения модель может неправильно классифицировать пример, который видела в обучающих данных при условии, что слова из этого примера встречались менее пяти раз, а его контекст не обладает большой предсказательной силой.

Последние две строки – запись модели на диск. Имя файла подсказывает, что модель следует записать в двоичном сжатом формате. И хотя пакет `opennlp.maxent` поддерживает и другие форматы, код, применяющий модель к новому тексту, ожидает именно этот.

5.5.3. Изменение входных данных для модели

Мы уже обсудили несколько причин для обучения своей модели и показали, как это делается. Среди причин упоминалось изменение входных данных для процедуры построения модели вкупе с приобретением или самостоятельным созданием набора аннотированных данных. Сначала мы поговорим об изменении процесса лексического анализа.

Для этого нужно предпринять два шага. Во-первых, обучающий и тестовый текст для процедуры обучения должны быть преобразованы в формат с разделенными пробелами лексемами – как в примере с именем Jason Willaford из предыдущего раздела. Способ преобразования может быть любым, но мы рекомендуем использовать тот же код, что на втором шаге процесса: применить модель к ранее не предъявлявшемуся тексту. Чтобы это сделать, нужно идентифицировать новые лексемы и передать их методу `NameFinderME.find`. На этом шаге придется написать свою реализацию интерфейса `Tokenizer` по аналогии с классом `opennlp.tools.tokenize.SimpleTokenizer`, с которым мы работали в разделе 5.2. Поскольку для расширения этого класса нужно только разбить строку и вернуть массив строк, то мы не станем приводить тривиальный код, а перейдем лучше к вопросу об изменении признаков.

Еще одно изменение, которое мы рассмотрим, касается признаков, используемых моделью для распознавания имени. Как и в случае лексического анализа, изменения признаков не вызывает особых сложностей, потому что метод `train` класса `NameFinderME` можно настроить, передав объект `AggregatedFeatureGenerator`, содержащий коллекцию генераторов признаков.

Листинг 5.9. Генерация нестандартных признаков для распознавания именованных сущностей в OpenNLP

```
AggregatedFeatureGenerator featureGenerators =
 new AggregatedFeatureGenerator(
  new WindowFeatureGenerator(
   new TokenFeatureGenerator(), 2, 2),
  new WindowFeatureGenerator(
```

Создаем агрегированный генератор признаков, содержащий 3 определенных ниже генератора

Создаем генератор признаков, соответствующий лексемам в окне шириной 5 лексем (по 2 слева и справа от текущей)

```
    new TokenClassFeatureGenerator(), 2, 2),
  new PreviousMapFeatureGenerator()
);
```

Создаем генератор признаков, соответствующий классам лексем в окне шириной 5 лексем (по 2 слева и справа от текущей)

Создаем генератор признаков, который определяет, как эта лексема размечалась раньше

В OpenNLP имеется много реализаций интерфейса `AdaptiveFeatureGenerator`, так что выбирать есть из чего. Ну а при необходимости можно написать свой генератор. Перечислим некоторые из имеющихся классов и объясним, что они делают.

- `CharacterNgramFeatureGenerator` – для генерации признаков, описывающих лексемы, использует символьные *n*-грамм.
- `DictionaryFeatureGenerator` – генерирует признаки, если лексемы встречаются в словаре.
- `PreviousMapFeatureGenerator` – генерирует признаки, сообщающие о результате, ассоциированном с ранее встречавшимся словом.
- `TokenFeatureGenerator` – генерирует признак, содержащий саму лексему.
- `TokenClassFeatureGenerator` – генерирует признаки, отражающие различные аспекты лексемы: класс символов (цифры/буквы/знаки препинания), длину лексемы, информацию о строчных и прописных буквах.
- `TokenPatternFeatureGenerator` – разбивает лексемы на подлексемы, исходя из классов символов, и генерирует классовые признаки для каждой подлексемы и комбинаций подлексем.
- `WindowFeatureGenerator` – генерирует признаки для данного генератора `AdaptiveFeatureGenerator` в окне лексем (например, одна слева и одна справа).

В процедуре обучения необходимо изменить обращение к методу `NameFinderME.train`, передав ему коллекцию генераторов признаков, как показано ниже.

Листинг 5.10. Обучение модели распознавания именованных сущностей с нестандартными признаками

```
File inFile = new File(baseDir,"person.train");
NameSampleDataStream nss = new NameSampleDataStream(
 new PlainTextByLineStream(
  new java.io.FileReader(inFile)));

int iterations = 100;
```

Создаем поток образцов

```
int cutoff = 5;
TokenNameFinderModel model = NameFinderME.train(
 "en", // язык
 "person", // тип сущности
 nss,
 featureGenerators,
 Collections.<String,Object>emptyMap(),
 iterations,
 cutoff);

File outFile = new File(destDir,"person-custom2.bin");
FileOutputStream outFileStream = new FileOutputStream(outFile);
model.serialize(outFileStream);
```

Обучаем модель с нестандартным генератором признаков

Сохраняем модель в файле

Аналогично для тестирования нужно модифицировать конструктор класса `NameFinderME`, включив в него коллекцию генераторов.

Листинг 5.11. Использование модели распознавания именованных сущностей с нестандартными признаками

```
NameFinderME finder = new NameFinderME(
 new TokenNameFinderModel(
   new FileInputStream(
     new File(destDir, "person-custom2.bin")
)), featureGenerators, NameFinderME.DEFAULT_BEAM_SIZE);
```

Поняв, как изменять входные данные механизма обучения OpenNLP, вы теперь сможете моделировать новые типы именованных сущностей и собирать дополнительную информацию о распознанных сущностях. Это даст возможность распространить вашу программу на широкий спектр типов сущностей и предметных областей. Но в некоторых случаях компромиссы, на которые идет OpenNLP во имя гибкости моделирования различных сущностей, обходятся слишком дорого с точки зрения потребления памяти и быстродействия. В следующем разделе мы рассмотрим еще некоторые способы настройки распознавания сущностей в OpenNLP, позволяющие повысить производительность.

5.5.4. Другой способ моделирования имен

Мы уже говорили о том, что в OpenNLP имеется отдельная модель для каждого типа именованных сущностей. Идея в том, чтобы пользователь мог выбирать только те модели, которые ему нужны. В этом разделе мы рассмотрим альтернативный способ моделирования име-

нованных сущностей, который не обладает подобной гибкостью, зато имеет другие преимущества. Выше мы видели, что имена моделируются с помощью сопоставления каждой лексеме одной из трех меток: start, continue, other. А в новой модели результат будет включать также тип распознаваемой сущности. При таком подходе результат распознавания имеет вид: person-start, person-continue, date-start, date-continue, other и т. д. в зависимости от того, какие типы должна предсказывать модель. В табл. 5.3 показано, как было размечено предложение.

Таблица 5.3. Предложение, размеченное с помощью альтернативной модели распознавания именованных сущностей

0	1	2	3	4	5	6	7	8	9
Britney	Spears	was	reunited	briefly	with	her	sons	Saturday	.
person-start	person-continue	other	other	other	other	other	other	date-start	other

У этого подхода есть несколько преимуществ над заведением отдельной модели для каждого типа именованных сущностей, но также ограничения и недостатки. Сначала перечислим преимущества.

- *Потенциальный выигрыш в производительности на этапе выполнения* – т. к. модель всего одна, признаки необходимо вычислять только один раз для всех категорий. И набор предсказаний для этой модели тоже всего один. Возможно, при обработке предложения придется увеличить число альтернативных наборов, потому что возрастает количество потенциальных результатов, но маловероятно, что потребуется три для каждого вычисляемого типа сущности.
- *Потенциальная экономия памяти* – в память нужно загружать только один набор признаков. Да и параметров меньше, потому что метка *other* общая для всех типов сущностей.
- *Объединение сущностей* – при наличии всего одной модели процесс объединения сущностей излишен.

Теперь о недостатках.

- *Отсутствие перекрытия сущностей* – поскольку с каждой лексемой сопоставляется только одна метка, то появление вложенных сущностей невозможно. Обычно это проблема решается путем разметки только той именованной сущности в обучающих данных, которая занимает максимальный интервал.

- *Непроизводительное расходование памяти и процессорного времени* – поскольку модель всего одна, нет возможности воспользоваться только теми ее частями, которые действительно необходимы. Это может повлечь за собой бессмысленное растранжиривание памяти или замедление работы в случае, когда интересуют всего одна-две категории.
- *Обучающие данные* – невозможно воспользоваться обучающими данными, аннотированными не для всех категорий. Добавление новой категории в модель потребует повторного аннотирования всех обучающих данных.

Как всегда при выборе компромиссов, приемлемость этого подхода зависит от потребностей приложения. Не исключено, что в вашем приложении оптимальным будет сочетание обеих моделей. Описанные в разделе 5.3.1 методы комбинирования моделей применимы и к комбинированию такой модели с моделями для распознавания одного типа сущностей.

Чтобы построить модель такого вида, нужно внести два изменения. Во-первых, в обучающих данных должны быть все аннотации, а не только аннотации для одного какого-то типа сущностей. Это можно сделать, изменив формат обучающих данных. Ниже приведен пример формата с поддержкой нескольких типов меток.

```
<START:person> Britney Spears <END> was reunited with her sons
<START:date>Saturday <END>.
```

Имея обучающие данные в таком виде, мы сможем воспользоваться предоставляемым OpenNLP классом `NameSampleDataStream` точно так же, как это было сделано в разделе 5.5.2. В следующем листинге показано, как порождается такая модель.

Листинг 5.12. Обучение модели с сущностями разных типов

```
String taggedSent =
  "<START:person> Britney Spears <END> was reunited " +
  "with her sons <START:date> Saturday <END> ";
ObjectStream<NameSample> nss = new NameSampleDataStream(
new PlainTextByLineStream(new StringReader(taggedSent)));

TokenNameFinderModel model = NameFinderME.train(
"en",
"default" ,
nss,
(AdaptiveFeatureGenerator) null,
```

```
 Collections.<String,Object>emptyMap(),
 70 , 1 );

File outFile = new File(destDir,"multi-custom.bin");
FileOutputStream outFileStream = new FileOutputStream(outFile);
model.serialize(outFileStream);

NameFinderME nameFinder = new NameFinderME(model);

String[] tokens =
   (" Britney Spears was reunited with her sons Saturday .")
   .split("\s+");
Span[] names = nameFinder.find(tokens);
displayNames(names, tokens);
```

В этой модели метки start и continue связаны с типом сущности. Тип записывается в начале метки. Последовательности меток person-start и person-continue порождают интервалы лексем, относящихся к именам людей, а метки date-start и date-continue определяют даты. Для получения типа служит метод `getType()` интервала, порождаемого в ходе обработки текста этой моделью.

5.6. Резюме

Распознавание и классификация имен собственных в тексте может стать богатым источником информации для приложений, занимающихся обработкой текста. Мы обсудили, как воспользоваться проектом OpenNLP для распознавания имен, а также привели некоторые показатели, характеризующие качество его работы, объем потребляемой памяти и скорость обработки. Мы рассмотрели также вопрос об обучении собственной модели и объяснили, зачем это может понадобиться. Наконец, мы описали, как эта задача решается в OpenNLP, и рассказали о настройке моделей, в том числе об альтернативном способе моделирования именованных сущностей. Эти знания помогут вам задействовать высокопроизводительные механизмы распознавания именованных сущностей в своих приложениях для обработки текста. Кроме того, вы получили первоначальное представление об использовании систем классификации для обработки текста. Позже мы еще вернемся к теме классификации и категоризации на более детальном уровне. А в следующей главе речь пойдет об автоматической группировке похожих объектов, например целых документов или результатов поиска, с помощью техники *кластеризации*. В отличие от классификации, кластеризация обычно производится без участия

человека, т. е. не предполагает предварительного обучения модели, а выделяет группы, основываясь на той или иной мере схожести.

5.7. Ресурсы

Mikheev, Andrei; Moens, Marc; Glover, Claire. 1999. “Named Entity Recognition without Gazetteers.” Proceedings of EACL ’99. HCRC Language Technology Group, University of Edinburgh. http://acl.ldc.upenn.edu/E/E99/E99-1001.pdf.

Wakao, Takahiro; Gaizauskas, Robert; Wilks, Yorick. 1996. “Evaluation of an algorithm for the recognition and classification of proper names.” Department of Computer Science, University of Sheffield. http://acl.ldc.upenn.edu/C/C96/C96-1071.pdf.

Zhou, GuoDong; Su, Jian. 2002. “Named Entity Recognition using an HMM-based Chunk Tagger.” Proceedings of the Association for Computational Linguistics (ACL), Philadelphia, July 2002. Laboratories for Information Technology, Singapore. http://acl.ldc.upenn.edu/acl2002/MAIN/pdfs/Main036.pdf.

ГЛАВА 6. Кластеризация текста

В этой главе:

- Основные концепции распространенных алгоритмов кластеризации текста.
- Примеры использования кластеризации для улучшения приложений обработки текста.
- Кластеризация слов для выделения представляющих интерес тем.
- Кластеризация целых наборов документов с помощью Apache Mahout и кластеризация результатов поиска с помощью Carrot^2.

Как часто, бродя по Интернету, вам доводилось щелкать по статье с интересным названием лишь для того, чтобы убедиться, что она мало чем отличается от только что прочитанной? А, быть может, ваша работа состоит в том, чтобы реферировать для начальства новости за день, но у вас не хватает времени внимательно прочитывать все относящиеся к делу статьи, а нужно только краткое резюме и основные моменты. Или пользователи вашей системы постоянно вводят неоднозначные или слишком общие запросы, так что в результатах поиска оказываются слишком разнородные документы, и вы хотели бы сгруппировать их по темам, чтобы не заставлять пользователя просеивать не нужные ему результаты. Хорошо бы иметь инструмент, который автоматически группирует похожие объекты и снабжает результаты обобщающими заголовками; это позволило бы сориентироваться в текстах большого объема или результатах поиска, не читая все содержимое.

В этой главе мы посмотрим, как решаются подобные задачи с помощью метода машинного обучения, который называется *кластериза-*

цией. Кластеризация производится без участия человека (в частности, не требуется аннотировать обучающий текст), а ее результатом является автоматическое распределение содержимого по «корзинам» и даже «наклеивание» на каждую корзину этикетки, сообщающей, что в ней находится.

Познакомившись с основными концепциями, мы затем перейдем к кластеризации результатов поиска с помощью проекта Carrot². Далее мы рассмотрим, как проект Apache Mahout применяется для кластеризации больших наборов документов. На самом деле, и в Carrot², и в Mahout реализовано несколько подходов к кластеризации, каждый со своими плюсами и минусами. Мы также обсудим, как применить метод *латентного размещения Дирихле* (он, кстати, тоже включен в Apache Mahout) для кластеризации на уровне слов с целью выделения тем в документе (эту задачу иногда называют *тематическим моделированием*). Рассматривая примеры, мы покажем, как все это можно надстроить над тем, что мы проделали раньше с помощью Apache Solr, чтобы обогатить способы доступа к информации еще и результатами кластеризации. И завершим мы эту главу разделом о показателях функционирования – количественных (насколько быстро?) и качественных (насколько хорошо?). Начнем с примера приложения, с которым многие из вас, наверное, уже знакомы, хотя и не подозревали, что оно основано на кластеризации: Google News.

6.1. Кластеризация документов в Google News

В условиях 24-часового новостного цикла, когда бесчисленные информагентства выкладывают свои версии событий, Google News позволяет читателям быстро ознакомиться с большинством материалов на указанную тему, опубликованных в определенный период времени, для чего группирует похожие статье вместе. Например, на рис. 6.1 видно, что под заголовком «Vikings begin Favre era on the road in Cleveland»[1] указано, что на эту тему есть еще 2181 статей. И хотя неизвестно, какие алгоритмы кластеризации Google применяет для реализации этой функции, в опубликованной документации четко говорится, что кластеризация используется (Google News [2011]):

> *В нашей технологии группировки принимаются во внимание различные факторы, в том числе заголовки, текст и время публикации.*

[1] Викинги начинают эру Фавра, отправляясь в Кливленд.

Мы применяем различные алгоритмы кластеризации для выявления тесно связанных материалов. Эти материалы отображаются на сайте Google News, давая посетителям возможность ознакомиться с новостями, видео, изображениями и другой информацией.

Возможности, которые таит такое умение выполнять группировку в гигантском масштабе, очевидны любому имеющему подключение к Интернету. И хотя для группирки новостей в режиме, близком к реальному времени, одного лишь применения алгоритма кластеризации к содержимому недостаточно, наличие масштабируемых реализаций, как в Apache Mahout, необходимо, чтобы сдвинуться с мертвой точки.

В таких задачах, как кластеризация новостей, приложение должно уметь быстро обрабатывать большое количество документов, определять репрезентативные документы или метки для отображения и иметь стратегию работы с вновь поступающими документами. Хороший алгоритм кластеризации – это еще не все, но нас будет интересовать только вопрос о том, как кластеризация помогает решать без участия человека эти и другие задачи, позволяющие выделять и обрабатывать информацию.

Рис. 6.1. Пример кластеризации документов в Google News. Снимок экрана сделан 13.09.2009

6.2. Основы кластеризации

По сути дела, кластеризация – это группировка похожих неразмеченных документов на основе некоторой меры схожести. Задача состоит в том, чтобы поместить все похожие документы из имеющегося набора в один и тот же кластер и при этом гарантировать, что непохожие документы попадают в разные кластеры. Но прежде чем излагать основы кластеризации, сформулируем, что мы ожидаем от нее. Кластеризация часто бывает полезна, но это не панацея. Оценка качества кластеризации зависит от ожиданий пользователя. Если пользовате-

ли настроены на идеальный результат, то они будут разочарованы. Если же они ожидают чего-то, что в большинстве случаев поможет сориентироваться в большом объеме данных и готовы смириться с некоторым количеством ложноположительных результатов, то, вероятно, будут довольны. С точки зрения проектировщика, приложение необходимо тщательно тестировать, чтобы найти подходящий баланс между быстродействием и качеством результатов. Помните, что ваша конечная цель – способствовать выявлению информации и творческому взаимодействию с материалом, а не во что бы то ни стало найти все похожие объекты.

Итак, мы дали определение и напутствия, а теперь расскажем, что собираемся рассмотреть в следующих разделах.

- различные типы текстов, к которым можно применить кластеризацию;
- как выбирать алгоритм кластеризации;
- как определять схожесть;
- подходы к идентификации меток;
- как оценивать результаты кластеризации.

6.2.1. Три типа текстов, поддающихся кластеризации

Кластеризацию можно применять к различным аспектам текста, в том числе словам в документе, самим документам или результатам поиска. Кластеризация полезна не только для текста, но и для группировки пользователей или данных от нескольких датчиков, но это выходит за рамки настоящей книги. Мы же сконцентрируем внимание на трех типах кластеризации: документы, результаты поиска и слова по тематике.

В случае кластеризации документов акцент ставится на документе как едином целом, как в примере с Google News. Обычно такая кластеризация производится пакетной задачей, результатом которой является список документов и вектор центроида. Поскольку задача кластеризации документов все равно решается не в режиме реального времени, часто имеет смысл потратить дополнительное (в пределах разумного) время на получение более качественных результатов. Описание кластера обычно генерируется путем анализа наиболее важных термов (определяемых с помощью какого-то механизма взвешивания, например TF-IDF) в документах, ближайших к центроиду. Обычно необходима некоторая предобработка, например, удаление стоп-слов

и стемминг, но есть алгоритмы, для которых это необязательно. При исследовании различных подходов можно поэкспериментировать и с другими стандартными способами обработки текстов, например, выделением словосочетаний или использованием *n*-грамм. Дополнительные сведения о кластеризации документов можно почерпнуть из книги «An Introduction to Information Retrieval» (Manning [2008]).

В случае кластеризации результатов поиска следует произвести поиск в ответ на введенный пользователем запрос и сгруппировать его результаты. Этот вид кластеризации особенно эффективен, когда пользователь вводит общие или неоднозначные термы (например, *apple*) или когда набор данных изначально разбит на отчетливые категории. Для кластеризации результатов поиска характерны следующие особенности:

- кластеризация по коротким текстовым фрагментам (название и, возможно, небольшая часть основного текста, где встретились поисковые термы);
- алгоритмы разрабатываются так, чтобы они могли работать с небольшим набором результатов, причем максимально быстро;
- меткам придается большее значение, поскольку пользователи будут трактовать их как фасеты, принимая решение, в какую сторону двигаться по полученным результатам дальше.

Как и при кластеризации документов, зачастую производится предобработка, но поскольку метки обычно важнее, имеет смысл потратить время на выявление часто встречающихся словосочетаний. Обзор по теме кластеризации результатов поиска см. в работе «A Survey of Web Clustering Engines» (Carpineto [2009]).

Кластеризация слов по тематике, называемая также *тематическим моделированием*, – эффективный способ быстро находить темы, затрагиваемые в большом наборе документов. Этот подход основан на предположении о том, что в документах часто рассматривается несколько тем и что слова, относящиеся к одной теме, часто расположены рядом. Производя кластеризацию слов, мы можем быстро понять, какие именно слова встречаются рядом и какие документы ассоциированы с этими словами. (В некотором смысле при этом осуществляется также кластеризация документов.) В табл. 6.1 показан результат работы алгоритма тематического моделирования – кластеры слов (см. раздел 6.6).

Первое, на что здесь стоит обратить внимание, – отсутствие названий у самих тем. Придумывание названия темы – задача человека,

создающего набор тем. С другой стороны, вы даже не знаете, какие документы относятся к каждой теме. Так о чем беспокоиться? Генерирование тем для набора документов – еще один способ помочь пользователям разобраться в нем и выявить интересную информацию о наборе, не читая все документы целиком. Кроме того, недавно появились исследования, посвященные более качественному определению характеристик тем с помощью словосочетаний (см. Blei [2009]).

Таблица 6.1. Пример разбиения слов по темам

Тема 0	Тема 1
win saturday time game know nation u more after two take over back has from texa first day man offici 2 high one sinc some sunday	yesterday game work new last over more most year than two from state after been would us polic peopl team run were open five american

Для получения дополнительных сведений о тематическом моделировании см. библиографию в конце главы и статью http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation, которая содержит ссылки на научные работы по этой теме.

Получив представление о том, какие типы текстов мы собираемся подвергать кластеризации, рассмотрим факторы, влияющие на выбор алгоритма.

6.2.2. Выбор алгоритма кластеризации

Существуют различные алгоритмы кластеризации, их полное рассмотрение выходит за рамки этой книги. На момент написания этой книги в проекте Apache Mahout имелись реализации метода K-средних (демонстрируется ниже), нечеткого метода K-средних, сдвига среднего, Дирихле, метода купольной кластеризации (canopy clustering), спектрального метода и латентного размещения Дирихле. Без сомнения, к моменту выхода книги из печати их будет больше. Но вместо того чтобы копаться в деталях реализации каждого метода, поговорим о некоторых чертах, общих для всех алгоритмов кластеризации, чтобы лучше понять, какие критерии определяют выбор алгоритма.

При обсуждении алгоритмов кластеризации нужно учесть многочисленные нюансы – только так удастся узнать, что лучше подходит для конкретного приложения. Традиционно считается, что одна из важнейших особенностей алгоритма – иерархический он или плоский. Иерархический алгоритм строит – снизу вверх или сверху вниз – иерархию связанных документов, разбивая множество на все

более и более мелкие подмножества. Плоские алгоритмы обычно работают гораздо быстрее, потому что не обязаны связывать кластеры между собой. Имейте также в виду, что некоторые плоские алгоритмы можно преобразовать в иерархические.

В табл. 6.2 приведены и другие факторы, играющие роль при выборе алгоритма кластеризации.

Таблица 6.2. Пример разбиения слов по темам

Характеристика	Описание
Членство в кластере (жесткое или мягкое)	Жесткое – каждый документ принадлежит одному и только одному кластеру. Мягкое – документ может принадлежать одновременно нескольким кластерам, причем зачастую он является членом кластера с некоторой вероятностью.
Возможность обновления	Можно ли обновлять кластеры при появлении новых документов или необходимо выполнить вычисление заново.
Вероятностный подход	Понимание подоплеки поможет оценить достоинства и недостатки такого подхода.
Быстродействие	Время работы большинства плоских алгоритмов кластеризации линейно зависит от количества документов, тогда как у многих иерархических алгоритмов зависимость нелинейная.
Качество	Иерархические алгоритмы часто точнее плоских, но за счет более медленной работы. Подробнее о количественных оценках см. раздел 6.2.5.
Возможность обратной связи	Может ли алгоритм адаптироваться и совершенствоваться в ответ на действия пользователя? Например, если пользователь пометит документ как не соответствующий кластеру, сможет ли алгоритм исключить его? Изменятся ли при этом другие кластеры?
Количество кластеров	Некоторые алгоритмы требуют, чтобы приложение заранее установило количество кластеров, другие решают, сколько кластеров необходимо, самостоятельно. Если алгоритм требует априорного задания, то будьте готовы экспериментировать для получения качественных результатов.

У некоторых алгоритмов имеются особенности, которые следует учитывать при выборе подхода, но в целом табл. 6.2 позволяет сориентироваться. Но будьте готовы потратить некоторое время на сравнение разных подходов, чтобы понять, какой оптимален для ваших данных.

6.2.3. Определение сходства

Во многих алгоритмах кластеризации применяется понятие сходства для определения того, принадлежит ли документ некоторому кластеру. Мера сходства реализуется как расстояние между документами. Чтобы идея расстояния заработала, в большинстве систем документы представляются как векторы (почти всегда разреженные, т. е. большинство элементов нулевые), в которых каждый элемент – вес конкретного терма в этом документе. Приложение вольно выбирать любое весовое значение, но обычно применяется тот или иной вариант TF-IDF. Если вам все это что-то напоминает, то и неудивительно, поскольку подходы к назначению весов документам при кластеризации и при поиске аналогичны. Если хотите освежить память, обратитесь к разделу 3.2.3.

На практике векторы документов почти всегда сначала нормируются с помощью p-нормы ($p >= 0$), так что очень короткие и очень длинные документы не оказывают негативного влияния на результат. Нормирование – это просто деление вектора на его длину, в результате чего концы всех векторов оказываются на поверхности единичного шара (в случае 2-нормы это окружность радиуса 1). Наиболее употребительные и, как следствие, наиболее знакомые читателю нормы – 1-норма (манхэттенское расстояние) и 2-норма (евклидово расстояние). Далее в этой главе для нормирования векторов будет применяться евклидово расстояние. Дополнительные сведения о p-нормах см. в статье http://en.wikipedia.org/wiki/Norm_(mathematics).

Коль скоро векторы введены в рассмотрение, расстояние между документами можно измерять как расстояние между векторами. Существует много способов измерения расстояния, но мы остановимся на самых распространенных.

- *Евклидово расстояние* – испытанное временем расстояние «прямой видимости» между двумя точками. Варианты – квадрат евклидова расстояния (чтобы сэкономить на извлечении квадратного корня) и назначение весовых коэффициентов отдельным элементам вектора.
- *Манхэттенское расстояние* – еще его называют *расстоянием городских кварталов*, потому что это расстояние, которое проехало бы такси в городе, разбитом на кварталы взаимно перпендикулярными улицами, как в районе Манхэттена в Нью-Йорке. Иногда отдельные элементы расстояния снабжаются весами.

- *Косинусоидальное расстояние*, равное косинусу угла, образованного векторами; следовательно, расстояние между двумя похожими документами (угол равен 0) равно 1. См. раздел 3.2.3.

В разделе о проекте Apache Mahout ниже будет показано, что способ вычисления расстояния часто является передаваемым параметром, что позволяет экспериментировать с различными метриками. Но в любом случае следует использовать расстояние, согласованное со способом нормирования вектора. Например, если использовалась 2-норма, то лучше брать евклидово или косинусоидальное расстояние. Впрочем, несмотря на теоретическую неправильность, некоторые алгоритмы неплохо работают и без такого согласования.

Для вероятностных подходов вопрос о сходстве – это, по существу, вопрос о вероятности того, что данный документ попадает в некоторый кластер. Часто в них применяется более сложная модель связи документов, основанная на статистическом распределении и других свойствах. Кроме того, можно показать, что некоторые алгоритмы, основанные на расстоянии (например, К средних), являются вероятностными.

6.2.4. Пометка результатов

Поскольку кластеризация используется реальными пользователями в качестве инструмента извлечения информации, то выбор хороших меток или достаточно репрезентативных документов зачастую не менее важен, чем нахождение самих кластеров. Не имея хороших меток и репрезентативных документов, пользователи не смогут интерактивно работать с кластерами для поиска полезных документов.

Выбрать из кластера репрезентативные документы можно несколькими способами. Самое простое – выбрать документы случайным образом, предложив пользователю широкую выборку результатов, что в принципе может привести к новым открытиям. Однако, если выбранные документы находятся далеко от центра, то пользователь может не понять, чему посвящен кластер. Чтобы исправить положение, при выборе документов можно руководствоваться их расстоянием до центроида кластера или вероятностью членства в кластере. В этом случае высоки шансы, что документы окажутся хорошими индикаторами назначения кластера, но способность к неожиданным открытиям, свойственная случайной выборке, отчасти теряется. Поэтому иногда применяют комбинированный подход, когда часть документов выбирается случайно, а часть – на основе близости или вероятности.

Выбрать хорошие метки, или темы труднее, чем репрезентативные документы. Для решения этой задачи разработаны различные подходы, со своими плюсами и минусами. В некоторых приложениях можно использовать простые методы, близкие к фасетному поиску (см. главу 3), позволяющие представить частоту меток, общих для кластера, но в большинстве случаев лучше искать и отображать важные слова и словосочетания, встречающиеся в кластере. Разумеется, что именно считать важным, – тема для исследования. Одно из простых решений – использовать веса в векторе (скажем, нашего доброго знакомого TF-IDF; см. раздел 3.2.3) и возвращать список термов, отсортированный по весу. С помощью *n*-грамм это решение можно обобщить и возвращать список словосочетаний (технически это словосочетания, но их качество может оказаться невысоким) на основе их весов в наборе (кластере). Другой распространенный подход – произвести концептуальное (тематическое) моделирование с помощью методов, в которых используется сингулярное разложение, например: латентный семантический анализ (см. Deerwester [1990]) или латентное размещение Дирихле (см. Blei [2003], демонстрируется ниже в этой главе). Еще один полезный подход – вычислить логарифмическое отношение правдоподобия (LLR; см. Dunning [1993]) термов, попавших и не попавших в кластер. Математические основания всех подходов, кроме TF-IDF (который мы уже обсуждали) выходят за рамки этой книги. Но рассказывая об использовании описываемых в этой главе инструментов (Carrot[2] и Apache Mahout), мы продемонстрируем все эти подходы, реализованные неявно, как часть самого алгоритма, или явно в отдельном инструменте. Так или иначе, метки, каким бы способом они ни были получены, весьма полезны для оценки качества кластеров, что является темой следующего раздела.

6.2.5. Как оценивать результаты кластеризации

Как всегда при обработке текста, экспериментирование и оценка результатов кластеризации должны быть не менее важными этапами разработки приложения, чем проектирование его архитектуры или способов развертывания. Как и в случае поиска, распознавания именованных сущностей и других рассматриваемых в этой книге тем, кластеризацию можно оценивать по-разному. Первым делом большинство разработчиков применяют *тест на смех*, или, как еще говорят, *проверку на вшивость*. Покажутся ли построенные кластеры

разумными человеку, который знает, как должен выглядеть хороший результат? Хотя вы нигде не прочтете о проверке на вшивость, она, тем не менее, является неоценимой частью процесса и обычно позволяет отловить «тупые» ошибки вроде неправильных или отсутствующих параметров. Беда в том, что невозможно повторить реакцию человека по запросу воспроизводимым образом, не говоря уже о том, что это лишь субъективное мнение. Кроме того, эта реакция во многом зависит от процесса генерации меток, которые могут неточно отражать состав кластера. Следующий этап тестирования – собрать несколько человек и попросить их оценить результаты. И неважно, кто эти люди: сотрудники отдела контроля качества или выборка из целевой аудитории – корреляция мнений небольшой группы лиц может дать ценные сведения. Цена –потраченное время и вознаграждение участникам, а дополнительные недостатки – субъективные ошибки и невоспроизводимость по запросу. Но если такой тест провести несколько раз, то можно вывести золотой стандарт, что и является нашей следующей целью.

Золотой стандарт – это набор кластеров, созданный одним или несколькими лицами и считающийся идеальным решением задачи кластеризации. Созданный набор может затем сравниваться с результатами кластеризации при различных подходах. Создавать золотой стандарт не всегда практично – он неустойчив (меняется при изменении или добавлении документов и т. д.), а для больших наборов данных это может оказаться недопустимо дорого. Если вы знаете, что набор документов не будет сильно изменяться, и располагаете временем, то, возможно, создание золотого стандарта, хотя бы для части набора, и окупится. Существует полуавтоматизированный подход – прогнать один или несколько алгоритмов кластеризации и попросить одного или нескольких человек вручную подправить обнаруженные кластеры. После применения экспертных оценок можно воспользоваться различными формулами (чистота, нормированная взаимная информация, индекс Ранда и F-мера) для приведения результатов к единому показателю, показывающему качество кластеризации. Мы не станем выписывать эти формулы здесь, а отошлем любознательных читателей к разделу 16.3 книги «An Introduction to Information Retrieval» (Manning [2008]), где всем им уделено достаточно внимания.

Наконец, существуют математические инструменты для оценки кластеров. Все они представляют собой различные эвристики и не требуют вмешательства человека. Лучше использовать их не по отде-

льности, а как полезные совокупные индикаторы качества кластеризации. Первый шаг состоит в том, чтобы удалить некоторое случайное подмножество исходных данных и снова выполнить кластеризацию. По завершении следует вычислить и запомнить процентную долю документов, попавших в каждый кластер. Затем удаленные данные возвращаются, кластеризация прогоняется еще раз и процентные далее рассчитываются заново. В предположении, что возвращенные данные распределены случайно, можно ожидать, что оба процентных распределения будут примерно одинаковы. Если в кластере A было 50 % документов при первом прогоне, то можно предположить (хотя это и не гарантируется), что и при втором прогоне сохранится доля 50 %.

В теории информации (первоначальные сведения см. в статье http://en.wikipedia.org/wiki/Information_theory) есть несколько полезных мер, пригодных для оценки качества кластеризации. Прежде всего, упомянем *энтропию*. Это мера неопределенности случайной переменной. На практике она измеряет информацию, содержащуюся в кластере. В случае кластеризации текста можно взять энтропию за основу и вычислить перплексивность, которая показывает, насколько хорошо членство в кластере может предсказать, какие слова используются.

О кластеризации можно рассказывать долго. Интересующимся предлагаем почитать книгу «An Introduction to Information Retrieval» (Manning [2008]). В частности, в главах 16 и 17 затронутые здесь концепции обсуждаются более детально. В работе Каттинга с соавторами «Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections» (Cutting [1992]) также имеется немало информации о том, как использовать кластеризацию для выделения информации. Мы же продолжим и рассмотрим несколько реализаций кластеризации, в том числе одну для результатов поиска, а другую для набора документов.

6.3. Подготовка к созданию простого приложения кластеризации

Далее мы проиллюстрируем рассмотренные выше концепции на примере содержимого нескольких новостных сайтов, полученного с помощью их лент в формате RSS или Atom. Для этой цели мы

настроили простой сервер Solr Home (схему, конфигурационный файл и т. д.), разместив все файлы в каталоге solr-clustering. Этот сервер будет переваривать новостные ленты нескольких газет и информагентств. Он автоматически импортирует и индексирует данные с помощью обработчика Data Import Handler, входящего в состав Solr. Имея новостные данные, мы сможем продемонстрировать различные библиотеки кластеризации.

Из главы 3 вы знаете, что основной интерес для нашего приложения представляют три файла: schema.xml, rss-data-config.xml и часть solrconfig.xml, в которой описывается обработчик Data Import Handler. Что касается схемы и конфигурации RSS, то мы изучили содержимое различных новостных лент и отобразили его на несколько общих полей, которые затем проиндексировали. Мы также сохраняем векторы термов по причинам, которые станут ясны в разделе 6.5.1.

Подробные сведения о конфигурировании Data Import Handler (DIH) можно найти на вики-сайте Solr по адресу http://wiki.apache.org/solr/DataImportHandler. Чтобы запустить Solr с теми настройками кластеризации, которые есть в прилагаемом к книге исходном коде, зайдите в корневой каталог дистрибутива и выполните следующие команды:

- `cd apache-solr/example`;
- `./bin/start-solr.sh solr-clustering`;
- выполните команду импорта данных: http://localhost:8983/solr/dataimport?command=full-import;
- проверьте состояние импорта: http://localhost:8983/solr/dataimport?command=status.

После этого мы можем приступить к демонстрации кластеризации на базе только что проиндексированных новостных данных. Сначала воспользуемся Carrot² для кластеризации результатов поиска, а затем применим Apache Mahout для кластеризации набора документов.

6.4. Кластеризация результатов поиска с помощью Carrot²

Carrot² – это библиотека кластеризации результатов поиска с открытым исходным кодом, распространяемая по лицензии, похожей на BSD. Скачать ее можно с сайта http://project.carrot2.org/. Она спроектирована специально для высокопроизводительной обработки типичных результатов поиска (заголовок и небольшой фрагмент

текста). Библиотека поддерживает различные поисковые API, в том числе Google, Yahoo!, Lucene и Solr (в качестве клиента), а также умеет производить кластеризацию документов в формате XML или создаваемых из программы. Кроме того, Carrot² интегрирована в серверную часть проекта Solr, и мы продемонстрируем это ниже. В Carrot² есть две реализации кластеризации: STC (кластеризация с помощью суффиксных деревьев) и Lingo.

Метод STC был впервые предложен для кластеризации результатов поиска в веб Замиром и Этциони в работе «Web document clustering: a feasibility demonstration» (Zamir [1998]). Алгоритм основан на структуре данных (суффиксном дереве), которая позволяет эффективно (за линейное время) находить общие подстроки. Именно это свойство важно для быстрого определения меток кластеров. Подробнее о суффиксных деревьях смотрите статью http://en.wikipedia.org/wiki/Suffix_tree.

Алгоритм Lingo предложен Станиславом Осиньски (Stanisław Osiński) и Давидом Вайссом (Dawid Weiss) (создателями проекта Carrot²). Не вдаваясь в детали, можно сказать, что в Lingo используется сингулярное разложение (см. http://en.wikipedia.org/wiki/Singular_value_decomposition) для обнаружения хороших кластеров и выявления словосочетаний для присваивания хороших меток кластерам.

В состав Carrot² входит также пользовательский интерфейс, позволяющий экспериментировать с данными, и серверная реализация, поддерживающая доступ в стиле REST, чтобы упростить взаимодействие с Carrot² из различных языков программирования. Наконец, при желании вы можете добавить в каркас свой собственный алгоритм кластеризации, лишь бы он был согласован со строго определенным API. Дополнительные сведения о Carrot² см. в руководстве по адресу http://download.carrot2.org/head/manual/.

Далее в этом разделе мы займемся тем, как использовать API для кластеризации источника данных, а затем посмотрим, как Carrot² интегрирован с Solr. И закончим этот раздел обзором качества и быстродействия обоих алгоритмов.

6.4.1. Использование Carrot² API

Архитектурно Carrot² представляет собой конвейер. Содержимое поступает из источника документов и передается одному или нескольким компонентам, которые модифицируют и кластеризуют исходные данные, выводя на другом конце кластеры. Если говорить

о конкретных классах, то конвейер состоит из одного или нескольких классов, реализующих интерфейс `IProcessingComponent`, которыми управляет класс, реализующий интерфейс `IController`. Контроллер отвечает за инициализацию компонентов и их вызов в правильном порядке и с нужными входными данными. Реализации `IProcessingComponent` – это классы, обрабатывающие различные источники документов (`GoogleDocumentSource`, `YahooDocumentSource`, `LuceneDocumentSource`), а также собственно реализации кластеризации: `STCClusteringAlgorithm` и `LingoClusteringAlgorithm`.

Естественно, используются и такие вещи, как лексический анализ и стемминг. Что до контроллера, то имеются две реализации: `SimpleController` и `CachingController`. Класс `SimpleController` легко настраивается и предназначен для одноразового использования, тогда как `CachingController` спроектирован для работы в производственной среде, где можно ожидать частого повторения запросов, и, следовательно, имеет смысл кэшировать результаты.

Чтобы увидеть Carrot[2] в действии, приведем пример кода для кластеризации простых документов. Первым делом необходимо эти документы создать. В нашем случае документ будет содержать три элемента: заголовок, краткий текст (реферат) и URL-адрес. Имея набор таких документов, нетрудно выполнить их кластеризацию, как показано в следующем листинге.

Листинг 6.1. Простой пример использования Carrot[2]

```
//... документы подготовлены в другом месте
final Controller controller =
       ControllerFactory.createSimple(); <— Создаем IController
documents = new ArrayList<Document>();
for (int i = 0; i < titles.length; i++) {
 Document doc = new Document(titles[i], snippets[i],
        "file://foo_" + i + ".txt");
 documents.add(doc);
}
final ProcessingResult result = controller.process(documents,
       "red fox",
       LingoClusteringAlgorithm.class); <— Cluster documents
displayResults(result); <— Распечатываем кластеры
```

Выполнение этой программы дает такой результат:

```
Cluster: Lamb
Mary Loses Little Lamb. Wolf At Large.
March Comes in like a Lamb
```

```
Cluster: Lazy Brown Dogs
Red Fox jumps over Lazy Brown Dogs
Lazy Brown Dogs Promise Revenge on Red Fox
```

И хотя документы для этого примера, очевидно, подготовлены искусственно (код их создания см. в файле Carrot2ExampleTest.java), из этого примера наглядно видно, как просто работать с Carrot² API. Но из соображений производительности многие приложения не захотят ограничиваться таким простым случаем, а предпочтут прибегнуть к классу `CachingController`. Как следует из названия, `CachingController` кэширует результаты для повышения производительности. Кроме того, автор приложения может использовать данные и из других источников (например, Google или Yahoo!) либо написать свою реализацию интерфейса `IDataSource` для представления содержимого документов. И наконец, многие компоненты обладают разнообразными атрибутами, настройка которых позволяет оптимизировать быстродействие и качество, но эту тему мы отложим до раздела 6.7.2.

Получив общее представление о том, как реализуется кластеризация с помощью Carrot², мы можем перейти к вопросу об интеграции с Solr.

6.4.2. Кластеризация результатов поиска Solr с помощью Carrot²

Начиная с версии 1.4, Apache Solr полностью поддерживает кластеризацию результатов поиска с помощью Carrot², в том числе задание в файле solrconfig.xml, всех атрибутов компонентов и все алгоритмы кластеризации. Разумеется, Carrot² осуществляет кластеризацию результатов поиска Solr, давая возможность приложению определить, какие поля содержат заголовок, фрагмент текста и URL-адрес. На самом деле, все это уже настроено в исходном коде, прилагаемом к книге, и находится в каталоге solr-clustering.

Конфигурирование Solr для использования Carrot² в качестве компонента кластеризации включает три шага. Во-первых, компонент реализован в виде `SearchComponent`, а, значит, может подключаться к обработчику запросов Solr (`RequestHandler`). Ниже показан фрагмент XML, относящийся к конфигурированию этого компонента.

```
<searchComponent
    class="org.apache.solr.handler.clustering.ClusteringComponent"
    name="cluster">
 <lst name="engine">
   <str name="name">default</str>
   <str name="carrot.algorithm"><lineArrow/>
```

```
    org.carrot2.clustering.lingo.LingoClusteringAlgorithm</str>
  </lst>
  <lst name="engine">
    <str name="name">stc</str>
    <str name="carrot.algorithm"><lineArrow/>
      org.carrot2.clustering.stc.STCClusteringAlgorithm</str>
  </lst>
</searchComponent>
```

В элементе `<searchComponent>` мы указываем, что настраивается компонент `ClusteringComponent`, а затем сообщаем Carrot[2], какие использовать алгоритмы кластеризации. В данном случае заданы оба алгоритма: Lingo и STC. Следующий шаг – подключить `SearchComponent` к `RequestHandler`:

```
<requestHandler name="standard"
     class="solr.StandardRequestHandler" default="true">
  <!-- значения параметров запроса по умолчанию -->
  <!-- ... -->
  <arr name="last-components">
    <str>cluster</str>
  </arr>
</requestHandler>
```

Наконец, часто бывает полезно настроить некоторые разумные умолчания, чтобы не передавать все параметры в строке запроса. В нашем примере мы поступили так:

```
<requestHandler name="standard"
     class="solr.StandardRequestHandler" default="true">
  <!-- значения параметров запроса по умолчанию -->
  <lst name="defaults">
    <!-- ... -->
    <!-- Кластеризация -->
    <!--<bool name="clustering">true</bool>-->
    <str name="clustering.engine">default</str>
    <bool name="clustering.results">true</bool>
    <!-- Поле заголовка -->
    <str name="carrot.title">title</str>
    <!-- Поле, по которому производить кластеризацию -->
    <str name="carrot.snippet">desc</str>
    <str name="carrot.url">link</str>
    <!-- создавать рефераты -->
    <bool name="carrot.produceSummary">true</bool>
    <!-- не создавать подкластеры -->
    <bool name="carrot.outputSubClusters">false</bool>
  </lst>
</requestHandler>
```

При настройке параметров по умолчанию мы объявили, что Solr должен использовать подсистему кластеризации, подразумеваемую по умолчанию (Lingo), и что Carrot² должна рассматривать поле title как заголовок, поле description – как фрагмент текста, а поле link – как URL-адрес. Наконец, мы просим Carrot² создавать рефераты, но не порождать подкластеры (для справки – подкластеры создаются путем выполнения кластеризации в пределах одного кластера).

И это вся настройка! В предположении, что Solr запускался, как показано в листинге 6.3, для получения от Solr кластеров в результатах поиска достаточно добавить в URL параметр `&clustering=true`, например: http://localhost:8983/solr/select/?q=*:*&clustering=true&rows=100. Выполнение этой команды заставит Solr извлечь 100 документов из индекса и произвести для них кластеризацию. На рис. 6.2 показано, как выглядит результат этого запроса.

Рис. 6.2. Результат выполнения команды кластеризации в Solr

Внизу мы сознательно оставили выпадающий из общего ряда результат *R Reuters sportsNews 4*, чтобы продемонстрировать необходи-

мость тщательной настройки Carrot² посредством различных имеющихся атрибутов. Эту тему мы обсудим в разделе 6.7.2. Полный перечень всех параметров компонента кластеризации в Solr см. на странице http://wiki.apache.org/solr/ClusteringComponent.

Поняв, как кластеризуются результаты поиска, пойдем дальше и обратимся к кластеризации целых наборов документов с помощью Apache Mahout. К Carrot² мы еще вернемся, когда будем говорить о производительности.

6.5. Кластеризация наборов документов с помощью Apache Mahout

Apache Mahout – это проект фонда Apache Software Foundation, цель которого – разработка комплекта библиотек машинного обучения, изначально спроектированных для масштабирования на очень большой объем входных данных. На момент написания этой книги он содержит алгоритмы классификации, кластеризации, коллаборативной фильтрации, эволюционного программирования и другие, а также полезные утилиты для решения задач машинного обучения, в том числе действий с матрицами и работы с примитивами Java (реализации интерфейсов Map, List и Set для хранения чисел типа int, double и т. п.). Во многих случаях Mahout опирается на каркас Apache Hadoop (http://hadoop.apache.org) (посредством модели программирования MapReduce и распределенной файловой системы *HDFS*) для разработки масштабируемых алгоритмов. В этой главе мы в основном будем говорить об использовании Mahout для кластеризации, а в главе 7 рассмотрим также классификацию. О других частях Mahout можно узнать на сайте http://mahout.apache.org/ и из книги «Mahout in Action» (см. http://manning.com/owen/). Перед чтением этого раздела скачайте Mahout 0.6 по адресу http://archive.apache.org/dist/mahout/0.6/mahout-distribution-0.6.tar.gz и распакуйте архив в какой-нибудь каталог, который мы, начиная с этого места, будем называть $MAHOUT_HOME. Затем перейдите в этот каталог и выполните команду `mvn install -DskipTests` (при желании можете прогнать тесты, но это долго!).

А теперь, не откладывая в долгий ящик, посвятим следующие три раздела вопросу о том, как подготовить данные и произвести их кластеризацию с помощью имеющейся в Apache Mahout реализации алгоритма K-средних.

Apache Hadoop – желтый слоник[2] с большими способностями к вычислениям

Hadoop – это реализация идей, выдвинутых Google (см. Dean [2004]). Первоначально он был частью подпроекта Lucene под названием Nutch, но затем отпочковался в отдельный проект фонда Apache Software Foundation. Основная идея – соединить распределенную файловую систему (у Google она называется *GFS*, а в Hadoop – *HDFS*) с моделью программирования (MapReduce), которая позволяет разработчикам, мало знакомым с параллельными и распределенными системами, писать масштабируемые и отказоустойчивые программы, работающие на очень больших компьютерных кластерах.

И хотя не любое приложение приспособлено к модели MapReduce, для многих приложений обработки текста этот подход вполне годится. Дополнительные сведения об Apache Hadoop см. в книгах Tom White «Hadoop: The Definitive Guide»[3] (http://oreilly.com/catalog/9780596521981) и Chuck Lam «Hadoop in Action»[4] (http://manning.com/lam/).

6.5.1. Подготовка данных к кластеризации

Mahout предполагает, что данные представлены в формате `org.apache.mahout.matrix.Vector`. Под вектором (`Vector`) в Mahout понимается просто кортеж чисел с плавающей точкой, например: <0.5, 1.9, 100.5>. Говоря более общо, вектор, или, как еще говорят, *вектор признаков* – это структура данных, часто применяемая в машинном обучении для представления свойств документа или иных данных в системе. Векторы нередко бывают разреженными, хотя это, конечно, зависит от данных. В случае приложений обработки текста векторы разрежены, потому что в наборе документов в целом количество различных слов очень велико, но в каждом конкретном документе – относительно мало. По счастью, у разреженности имеются преимущества, когда речь заходит о задачах машинного обучения. Естественно, в составе Mahout есть реализации, расширяющие класс `Vector` для представления как разреженных, так и плотных векторов. Они называются соответственно `org.apache.mahout.matrix.SparseVector` и `org.apache.mahout.matrix.DenseVector`. Перед запуском приложения необходимо на небольшой выборке данных понять, разреженные они или плотные, и выбрать подходящее пред-

[2] По словам автора Hadoop, Дуга Каттинга, Хадупом называл своего плюшевого слоника его маленький сын. – *Прим. перев.*

[3] Том Уайт «Hadoop. Подробное руководство», Питер, 2013.

[4] Чак Лэм «Hadoop в действии», ДМК Пресс, 2012.

ставление. И обязательно проверяйте оба подхода на подмножествах данных, чтобы понять, какой работает лучше.

В Mahout есть несколько способов создания векторов для кластеризации.

- *Программный* – написать код, который создает экземпляр `Vector` и сохраняет его в подходящем месте.
- *Из индекса Apache Lucene* – преобразовать индекс Apache Lucene в множество векторов.
- *Формат Weka ARFF* – Weka – это посвященный машинному обучению проект университета Вайкато (Новая Зеландия), в котором определен формат ARFF. Дополнительные сведения см. на странице http://cwiki.apache.org/MAHOUT/creating-vectors-from-wekas-arff-format.html. Проект Weka описан в книге Witten, Frank «Data Mining: Practical Machine Learning Tools and Techniques (3-е издание)» (http://www.cs.waikato.ac.nz/~ml/weka/book.html).

Поскольку в этой книге проект Weka не используется, мы опустим формат ARFF, а остановимся на первых двух способах создания векторов для Mahout.

Программное создание векторов

Создать векторы из программы несложно. Как это делается, показано в примере ниже.

Листинг 6.2. Создание векторов в Mahout

```
double[] vals = new double[]{0.3, 1.8, 200.228};
Vector dense = new DenseVector(vals);
assertTrue(dense.size() == 3);
Vector sparseSame = new SequentialAccessSparseVector(3);

Vector sparse = new SequentialAccessSparseVector(3000);
for (int i = 0; i < vals.length; i++) {
 sparseSame.set(i, vals[i]);
 sparse.set(i, vals[i]);
}
assertFalse(dense.equals(sparse));

assertEquals(dense, sparseSame);
assertFalse(sparse.equals(sparseSame));
```

Создаем DenseVector с тремя значениями

Создаем SparseVector мощностью 3

Создаем SparseVector мощностью 3000

Присваиваем значения первым трем элементам разреженного вектора

Плотный и разреженный векторы не равны, потому что у них разная мощность

Плотный и разреженный векторы равны, потому что у них одинаковая мощность и элементы

Векторы создаются программно, если данные читаются из базы или другого источника, не поддерживаемого Mahout. Построенный вектор следует записать в понятном Mahout формате. Все алгоритмы кластеризации в Mahout ожидают получить на входе один или несколько файлов в формате файла последовательности Hadoop (SequenceFile). Mahout предоставляет класс `org.apache.mahout.utils.vectors.io.SequenceFileVectorWriter` для сериализации векторов в подходящем формате. В листинге ниже показано, как это делается.

Листинг 6.3. Создание векторов в Mahout

```
File tmpDir = new File(System.getProperty("java.io.tmpdir"));
File tmpLoc = new File(tmpDir, "sfvwt");
tmpLoc.mkdirs();
File tmpFile = File.createTempFile("sfvwt", ".dat", tmpLoc);

Path path = new Path(tmpFile.getAbsolutePath());
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
SequenceFile.Writer seqWriter = SequenceFile.createWriter(fs, conf,
     path, LongWritable.class, VectorWritable.class);
VectorWriter vecWriter = new SequenceFileVectorWriter(seqWriter);
List<Vector> vectors = new ArrayList<Vector>();
vectors.add(sparse);
vectors.add(sparseSame);
vecWriter.write(vectors);
vecWriter.close();
```

Создаем конфигурацию Hadoop

Создаем объект SequenceFile.Writer, на который возлагается задача физической записи векторов в файл в системе HDFS

Объект VectorWriter обрабатывает векторы, вызывая методы стоящего за ним объект SequenceFile.Writer

Производим запись в файлы

Mahout умеет также записывать векторы в формате JSON, но это делается только для чтения человеком, потому что сериализация в такой формат и обратная десериализация во время выполнения требуют больше времени, что значительно замедляет работу алгоритмов кластеризации. Поскольку мы работаем с сервером Solr, а тот опирается на Apache Lucene, то в следующем разделе рассмотрим создание векторов из индекса Lucene, что куда интереснее.

Создание векторов из индекса Apache Lucene

Один или несколько индексов Lucene – отличный источник создания векторов в предположении, что в схеме поле, из которого создаются векторы, было определено с атрибутом `termVector="true"`, как показано ниже:

```
<field name="description" type="text"
       indexed="true" stored="true"
       termVector="true"/>
```

Имея такой индекс, мы можем воспользоваться входящими в состав Mahout утилитами для работы с Lucene, чтобы преобразовать индекс в файл последовательности, содержащий векторы. Сделать это можно из командной строки, выполнив программу `org.apache.mahout.utils.vector.lucene.Driver`. У этой программы много аргументов, в табл. 6.3 перечислены наиболее употребительные.

Таблица 6.3. Аргументы преобразования индексов Lucene

Аргумент	Описание	Обязателен
`--dir <Path>`	Определяет местоположение индекса Lucene	Да
`--output <Path>`	Путь к результирующему файлу последовательности в файловой системе.	Да
`--field <String>`	Имя поля Lucene, используемого в качестве источника данных.	Да
`--idField <String>`	Имя поля Lucene, содержащего уникальный идентификатор документа. Может служить для пометки вектора.	Нет
`--weight [tf|tfidf]`	Тип веса, применяемого для представления термов в модели. TF – только частота терма; TF-IDF – частота терма и обратная частота документа.	Нет
`--dictOut <Path>`	Путь к файлу, в который записывается соответствие между термами и их позицией в векторе.	Да
`--norm [INF|-1|A double >= 0]`	Способ нормирования вектора. См. http://en.wikipedia.org/wiki/Lp_norm.	Нет

Чтобы воспользоваться этим в контексте сервера Solr, мы можем указать драйверу путь к каталогу, содержащему индекс Lucene, и задать входные параметры – драйвер сделает все остальное. Для демонстрации предположим, что индекс Lucene находится в каталоге <Solr Home>/data/index и что он был создан, как показано выше.

Тогда для генерации векторов нужно запустить драйвер, как показано в следующем листинге.

Листинг 6.4. Пример создания векторов из индекса Lucene

```
<MAHOUT_HOME>/bin/mahout lucene.vector
       --dir <PATH>/solr-clustering/data/index
       --output /tmp/solr-clust-n2/part-out.vec --field description
       --idField id --dictOut /tmp/solr-clust-n2/dictionary.txt --norm 2
```

Здесь драйвер читает индекс Lucene, получает из него необходимую информацию о документах и создает файл part-out.dat (слово *part* важно для связки Mahout/Hadoop). Создается также файл dictionary.txt, в котором хранится соответствие между термами индекса и позициями в созданном векторе. Это важно для последующего воссоздания векторов для целей отображения. Наконец, мы указали 2-норму, чтобы при кластеризации можно было использовать класс `CosineDistanceMeasure`, входящий в состав Mahout. Теперь, имея векторы, мы можем произвести кластеризацию с помощью включенной в Mahout реализации метода K-средних.

6.5.2. Кластеризация методом K-средних

Существует много способов кластеризации, как в области машинного обучения в широком смысле, так и в составе Mahout. Так, на момент написания этой книги в Mahout были включены реализации следующих методов кластеризации:

- купольная (Canopy);
- сдвига среднего (Mean-Shift);
- Дирихле (Dirichlet);
- спектральная (Spectral);
- K-средних (K-Means) и нечеткая K-средних (Fuzzy K-Means).

Из них метод K-средних известен наиболее широко. Это простой и прямолинейный подход к кластеризации, который часто дает хорошие результаты и при том относительно быстро. Идея в том, чтобы итеративно помещать документы в один из k кластеров, исходя из расстояния между документом и центроидом этого кластера (метрика, на основе которой вычисляется расстояние, задается пользователем). В конце итерации положение центроида пересчитывается. Процесс останавливается, когда центроиды почти перестают сдвигаться или после заранее заданного числа итераций, поскольку сходимость метода K-средних не гарантируется. Для запуска

алгоритма задаются некоторые начальные положения центроидов или они выбираются случайно из входного набора данных. У метода K-средних есть недостатки. Главный из них состоит в том, что необходимо заранее выбрать *k*; понятно, что при разных *k* получаются разные результаты. Далее, результат может сильно зависеть от начального выбора центроидов, поэтому нужно прогнать алгоритм несколько раз с разными значениями и сравнить результаты. И наконец, как и в большинстве других методов, разумно будет поиграть с параметрами и посмотреть, когда алгоритм работает лучше всего на ваших конкретных данных.

Для кластеризации методом K-средних в Mahout нужно просто выполнить программу `org.apache.mahout.clustering.kmeans.KMeansDriver` с подходящими параметрами. Благодаря Hadoop это можно делать в автономном или распределенном режиме (на кластере Hadoop). В этой книге мы пользуемся автономным режимом, но распределенный отличается не сильно.

Прежде чем описывать параметры программы `KMeansDriver`, сразу покажем, как произвести кластеризацию, на примере созданного ранее файла векторов. В листинге ниже приведена командная строка для запуска `KMeansDriver`.

Листинг 6.5. Пример использования командной утилиты KMeansDriver

```
<$MAHOUT_HOME>/bin/mahout kmeans \
     --input /tmp/solr-clust-n2/part-out.vec \
     --clusters /tmp/solr-clust-n2/out/clusters -k 10 \
     --output /tmp/solr-clust-n2/out/ --distanceMeasure \
     org.apache.mahout.common.distance.CosineDistanceMeasure \
     --convergenceDelta 0.001 --overwrite --maxIter 50 --clustering
```

Большинство параметров не нуждается в объяснениях, мы остановимся лишь на шести самых важных для алгоритма K-средних.

- `--k` – значение *k*. Задает число возвращаемых кластеров.
- `--distanceMeasure` – задает метрику для вычисления расстояния от документа до центроида. В данном случае мы использовали косинусоидальное расстояние (вспомните, как работает Lucene/Solr). В Mahout есть и другие метрики в пакете `org.apache.mahout.common.distance`.
- `--convergenceDelta` – определяет порог, по достижении которого считается, что алгоритм сошелся и можно завершать кластеризацию. По умолчанию 0,5. Выбранное нами значение 0,001 совершенно произвольно. Пользователям рекомендует-

ся подобрать такое значение, при котором имеет место приемлемый компромисс между временем и качеством работы.

- `--clusters` – путь к файлу, содержащему начальные центроиды кластеров. Если параметр `--k` явно не задан, то этот файл должен содержать *k* векторов (файл должен быть сериализован, как показано в листинге 6.3). Если параметр `--k` задан, то будет случайным образом выбрано *k* векторов из входных данных.
- `--maxIter` – задает максимальное число итераций, после которых программа завершается, даже если алгоритм не сошелся.
- `--clustering` – отвести дополнительное время на вывод членов кластеров. Если этот параметр опущен, то определяются только центроиды кластеров.

После запуска показанной выше команды по экрану пробегут различные информационные сообщения, среди которых, надеемся, не будет сообщений об ошибках и исключениях. По завершении в выходном каталоге появится несколько подкаталогов, содержащих результаты каждой итерации (с именем clusters-*X*, где *X* – номер итерации), а также входные кластеры (в нашем случае они были сгенерированы случайно) и точки, соответствующие результирующему кластеру, полученному на последней итерации.

Результатом также являются файлы последовательностей Hadoop, не предназначенные для чтения человеком. Но в составе Mahout есть несколько утилит для просмотра результатов кластеризации. Наиболее полезная из них `org.apache.mahout.utils.clustering.ClusterDumper`, но стоит присмотреться также к `org.apache.mahout.utils.ClusterLabels`, `org.apache.mahout.utils.SequenceFileDumper` и `org.apache.mahout.utils.vectors.VectorDumper`. Здесь мы остановимся на `ClusterDumper`. Как следует из названия, эта утилита предназначена для вывода созданных кластеров в окно консоли или в файл в понятном человеку формате. Например, чтобы увидеть результаты работы показанной ранее программы `KMeansDriver`, выполните такую команду:

```
<MAHOUT_HOME>/bin/mahout clusterdump \
       --seqFileDir /tmp/solr-clust-n2/out/clusters-2 \
       --dictionary /tmp/solr-clust-n2/dictionary.txt --substring 100 \
       --pointsDir /tmp/solr-clust-n2/out/points/
```

В этом примере мы указываем, где находятся каталоги, содержащие кластеры (`--seqFileDir`), словарь (`--dictionary`) и исходные точки (`--pointsDir`). Мы также просим оставить при печати вектора

центра кластера только 100 символов (`--substring`), чтобы результат был обозримее. После выполнения этой команды для результатов кластеризации индекса, созданного по новостям от 5 июля 2010 года, получилась такая распечатка:

```
:C-129069: [0:0.002, 00:0.000, 000:0.002, 001:0.000, 0011:0.000, \
002:0.000, 0022:0.000, 003:0.000, 00
Top Terms:
  time                              =>0.027667414950403202
  a                                 => 0.02749764550779345
  second                            => 0.01952658941437323
  cup                               =>0.018764212101531803
  world                             =>0.018431212697043415
  won                               =>0.017260178342226474
  his                               => 0.01582891691616071
  team                              =>0.015548434499094444
  first                             =>0.014986381107308856
  final                             =>0.014441638909228182
:C-129183: [0:0.001, 00:0.000, 000:0.003, 00000000235:0.000, \
001:0.000, 002:0.000, 01:0.000, 010:0.00
Top Terms:
  a                                 => 0.05480601091954865
  year                              =>0.029166628670521253
  after                             =>0.027443270009727756
  his                               =>0.027223628226736487
  polic                             => 0.02445617250281346
  he                                =>0.023918227316575336
  old                               => 0.02345876269515748
  yearold                           =>0.020744182153039508
  man                               =>0.018830109266458044
  said                              =>0.018101838778995336
...
```

В данном случае `ClusterDumper` выводит идентификатор вектора центроида и несколько попавших в кластер термов с указанием частоты. Внимательное изучение первых термов показывает, что среди них много хороших, но встречаются и плохие, в частности несколько стоп-слов (a, his, said и т. д.). Пока мы закроем на это глаза, но вернемся в разделе 6.7.

Хотя простая распечатка кластеров тоже бывает полезна, в большинстве приложений необходимы короткие метки, резюмирующие содержимое кластеров (см. раздел 6.2.4 выше). Входящий в Mahout класс `ClusterLabels` является средством генерации меток по индексу Lucene (Solr), его можно использовать для получения списка слова, наилучшим образом описывающих кластер. Чтобы применить программу `ClusterLabels` к полученному нами результату кластери-

зации, выполните следующую команду, находясь в том же каталоге, откуда запускали предыдущие команды.

```
<MAHOUT_HOME>/bin/mahout \
org.apache.mahout.utils.vectors.lucene.ClusterLabels \
--dir /Volumes/Content/grantingersoll/data/solr-clustering/data/index/\
--field desc-clustering --idField id \
--seqFileDir /tmp/solr-clust-n2/out/clusters-2 \
--pointsDir /tmp/solr-clust-n2/out/clusteredPoints/ \
--minClusterSize 5 --maxLabels 10
```

Здесь мы задали многие из тех параметров, которые использовались при извлечении содержимого из индекса, в частности, местоположение индекса и интересующие поля. Мы добавили также информацию о том, где находятся кластеры и точки. Параметр `minClusterSize` – это минимальное количество документов в кластере, при котором возможно вычислить метки. Это полезно при кластеризации очень больших наборов, когда получаются большие кластеры, поскольку приложение может игнорировать мелкие кластеры, считая их аномалиями. Параметр `maxLabels` задает максимальное количество меток для одного кластера. При выполнении этой команды для созданных ранее данных получаем (распечатка сокращена):

```
Top labels for Cluster 129069 containing 15306 vectors
Term         LLR                 In-ClusterDF     Out-ClusterDF
team         8060.366745727311              3611         2768
cup          6755.711004478377              2193         645
world        4056.4488459853746             2323         2553
reuter       3615.368447394372              1589         1058
season       3225.423844734556              2112         2768
olymp        2999.597569386533              1382         1004
championship 1953.5632186210423             963          781
player       1881.6121935029223             1289         1735
coach        1868.9364836380992             1441         2238
u            1545.0658127206843             35           7101

Top labels for Cluster 129183 containing 12789 vectors
Term         LLR                 In-ClusterDF     Out-ClusterDF
polic        13440.84178933248              3379         550
yearold      9383.680822917435              2435         427
old          8992.130047334154              2798         1145
man          6717.213290851054              2315         1251
kill         5406.968016825078              1921         1098
year         4424.897345832258              4020         10379
charg        3423.4684087312926             1479         1289
arrest       2924.1845144664694             1015         512
murder       2706.5352747719735             735          138
death        2507.451017449319              1016         755
...
```

Столбцы распечатки содержат следующие данные.

- *Term* – сама метка.
- *LLR (логарифмическое отношение правдоподобия)* – используется для оценки качества терма на основе различных статистических данных об индексе Lucene. Подробнее о LLR см. http://en.wikipedia.org/wiki/Likelihood-ratio_test.
- *In-ClusterDF* – количество вошедших в кластер документов, в которых данный терм встречается. Для вычисления LLR используется эта величина и Out-ClusterDF.
- *Out-ClusterDF* – количество не вошедших в кластер документов, в которых данный терм встречается.

Как и в случае первых термов, напечатанных программой `ClusterDumper`, при ближайшем рассмотрении оказывается, что имеются как хорошие термы (не обращаем внимания на то, что они подвергнуты стеммингу), так и бесполезные. Следует отметить, что большинство термов дают правильное общее представление о том, чему посвящены документы, попавшие в кластер. В разделе 6.7 мы поговорим о том, как улучшить результат. А пока посмотрим, как можно воспользоваться некоторыми возможностями Mahout, чтобы выявить темы, исходя из результатов кластеризации слов в документах.

6.6. Тематическое моделирование с помощью Apache Mahout

В Mahout имеются не только средства для кластеризации документов, но и для тематического моделирования. В применении к тексту эти средства можно представлять себе как кластеризацию на уровне слов. Единственная реализация тематического моделирования в Mahout основана на алгоритме латентного размещения Дирихле (LDA). Алгоритм LDA (Deerwester [1990]) – это

> ... порождающая вероятностная модель для наборов дискретных данных, в частности, корпусов текстов. LDA – это трехуровневая иерархическая байесовская модель, в которой элементы набора моделируются как конечная смесь множества исходных тем. Каждая тема, в свою очередь, моделируется как бесконечная смесь множества исходных вероятностей тем.

Говоря простым языком, алгоритм LDA преобразует кластеры слов в темы, исходя из предположения, что исходные документы посвящены некоторому числу тем, но неизвестно, какой документ к какой теме относится, а также, как называется тема. На первый взгляд, идея представляется бесполезной, однако знание о тематических словах, ассоциированных с имеющимся набором, весьма ценно. Например, в сочетании с Solr эти слова можно было бы использовать для наделения поискового приложения дополнительными средствами обнаружения информации. Или же их можно применить для составления краткого реферата большого набора документов. Тематические термы полезны и в других задачах, например, классификации и коллаборативной фильтрации (подробнее об этих применениях см. Deerwester [1990]). Ну а мы пока посмотрим, как запустить реализацию LDA в Mahout.

Чтобы воспользоваться LDA в Mahout, нам понадобятся какие-нибудь векторы. Как мы уже говорили, создать векторы можно из индекса Lucene. Но для LDA потребуется одно небольшое изменение: в качестве весов нужно использовать только частоты термов (TF), а не подразумеваемые по умолчанию величины TF-IDF; это объясняется способом вычисления внутренних статистик в алгоритме. Команда может выглядеть следующим образом:

```
<MAHOUT_HOME>/bin/mahout lucene.vector \
--dir <PATH TO INDEX>/solr-clustering/data/index/ \
--output /tmp/lda-solr-clust/part-out.vec \
--field desc-clustering --idField id \
--dictOut /tmp/lda-solr-clust/dictionary.txt \
--norm 2 --weight TF
```

Этот пример отличается от приведенного выше только другими путями для выходных файлов и значением (TF) параметра `--weight`.

Имея векторы, мы далее можем запустить алгоритм LDA:

```
<MAHOUT_HOME>/bin/mahout lda --input /tmp/lda-solr-clust/part-out.vec \
--output /tmp/lda-solr-clust/output --numTopics 30 --numWords 61812
```

Назначение большинства параметров очевидно, но некоторые заслуживают пояснений. Во-первых, мы выделили приложению дополнительную память[5]. Алгоритм LDA требователен к памяти, так что это не повредит. Во-вторых, мы просим программу LDADriver выделить

[5] В приведенной команде такого параметра нет. Автор, видимо, что-то забыл. – *Прим. перев.*

30 тем (`--numTopics`) в переданных векторах. Как и алгоритм K-средних, LDA требует, чтобы желательное количество целей было задано заранее. Это означает, что придется методом проб и ошибок определить, сколько тем нужно вашему приложению. Мы выбрали 30 – не претендуя на научный подход, а просто посмотрев, какие результаты получаются при 10 и 20 темах. И наконец, параметр `--numWords` – это количество слов во всех векторах. Если пользоваться тем способом создания векторов, который описан выше, то количество слов легко получить из первой строки файла dictionary.txt. После завершения работы LDA в выходном каталоге появится несколько подкаталогов с именами *state-**, например: state-1, state-2 и т. д. Сколько именно, зависит от входных данных и параметров. В каталоге с наибольшим номером будет находиться конечный результат.

Разумеется, после прогона LDA хотелось бы посмотреть на результаты. По умолчанию LDA их не распечатывает. Но в Mahout имеется удобный инструмент для распечатки тем, который так и называется: `LDAPrintTopics`. Он принимает несколько обязательных и один факультативный параметр:

- `--input` – один из каталогов state, содержащих результаты прогона LDA. Это может быть любой каталог, не обязательно последний. Параметр обязателен.
- `--output` – каталог, в который записываются результаты. Параметр обязателен.
- `--dict` – каталог термов, использованных при создании векторов. Параметр обязателен.
- `--words` – сколько слов печатать в каждой теме. Параметр необязателен.

После запуска LDA с помощью показанной выше команды мы запустили `LDAPrintTopics`:

```
java -cp "*" \
    org.apache.mahout.clustering.lda.LDAPrintTopics \
    --input ./lda-solr-clust/output/state-118/ \
    --output lda-solr-clust/topics \
    --dict lda-solr-clust/dictionary.txt --words 20
```

В данном случае мы хотели получить первые 20 слов из каталога state-118 (последнего, кстати). В результате выполнения этой команды в выходном каталоге появилось 30 файлов, по одному для каждой темы. Файл для темы 22 выглядел так:

```
Topic 22
==========
yearold
old
cowboy
texa
14
second
year
manag
3414
quarter
opera
girl
philadelphia
eagl
arlington
which
dalla
34
counti
five
differ
1996
tri
wide
toni
regul
straight
stadium
romo
twitter
```

На основании этих слов можно сделать вывод, что тема, вероятно, посвящена победе Далласских ковбоев над Филадельфийскими орлами в плей-офф Национальной футбольной лиги, которая имела место за день до прогона этого примера. И, хотя некоторые слова выглядят аномалиями (opera, girl), большая их часть позволяет составить представление о теме. Поиск в индексе нескольких термов показывает, что действительно были статьи об этом событии, и в одной говорилось о появлении в Твиттере заявления принимающего Орлов Дешона Джексона о том, что Орлы разгромят Ковбоев (этим спортсменам, похоже, никакой урок не впрок). Вот и все, что нужно знать о выполнении алгоритма LDA из Apache Mahout. Далее мы поговорим о качестве кластеризации в Carrot[2] и Mahout.

6.7. Качество кластеризации

Как всегда бывает, стоит программисту хотя бы в общих чертах понять принцип работы приложения, он сразу начинает прикидывать, как бы использовать его в производственной системе. Для ответа на этот вопрос нужно познакомиться с качественными и количественными показателями функционирования. Начнем с вопроса об отборе и уменьшении числа признаков с целью повышения качества, а затем рассмотрим алгоритм отбора и его входные параметры в Carrot[2] и Apache Mahout. И напоследок выполним несколько тестов производительности с помощью облачной системы *EC2*, предоставляемой компанией Amazon (http://aws.amazon.com). Для тестирования на Amazon мы воспользовались помощью двух добровольцев, Тимоти Поттера (Timothy Potter) и Шимона Чойнацки (Szymon Chojnacki), попросив их обработать довольно большой набор сообщений электронной почты и посмотреть, как Mahout будет вести себя в кластере из нескольких машин.

6.7.1. Отбор и уменьшение числа признаков

Отбор и уменьшение числа признаков – это методика, призванная либо улучшить качество результатов, либо сократить объем обрабатываемых данных. Цель отбора признаков – выбрать хорошие признаки заранее, отыскав в ходе предобработки или передав на вход алгоритма. Уменьшение количества признаков преследует цель убрать в ходе автоматизированной обработки признаки, вклад которых невелик. Оба метода дают ряд преимуществ, в том числе:

- уменьшение размерности задачи до обозримой величины с точки зрения как объема вычислений, так и потребной памяти;
- повышение качества за счет устранения зашумленности данных, в частности стоп-словами;
- визуализация и постобработка – если признаков слишком много, то их трудно наглядно представить в пользовательском интерфейсе и обработать на последующих этапах.

Во многих отношениях вы уже знакомы с уменьшением количества признаков – благодаря работе, проделанной в главе 3. Там мы применяли различные приемы анализа, в том числе удаление стоп-

слов и стемминг, для уменьшения количества термов, среди которых производится поиск. Подобные приемы полезны и для улучшения результатов кластеризации. Более того, при кластеризации выгодно применять еще более агрессивные методы отбора признаков, поскольку количество подлежащих обработке документов зачастую очень велико, и любое априорное уменьшение объема способно дать существенную экономию.

Например, для примеров из этой главы мы взяли другой файл стоп-слов (файл stopwords-clustering.txt в прилагаемом к книге коде), в котором число стоп-слов больше, чем при поиске. При создании этого файла мы изучили список термов, чаще всего встречающихся в индексе, и выполнили несколько итераций кластеризации, чтобы решить, какие слова имеет смысл удалить.

К сожалению, этот подход не является общим и требует довольно много работы. Кроме того, он не обобщается на другие языки и даже на другие корпусы текстов. Для повышения переносимости приложения обычно пытают удалять термы, исходя из их весов (применяя TF-IDF и другие методы), а затем итеративно вычисляют некоторую меру и определяют, улучшилось качество кластеров или нет. В упоминаемых в библиографии работах (Dash [2000], Dash [2002] и Liu [2003]) можно найти различные подходы с обсуждением каждого. Можно также применить метод сингулярного разложения (SVD), включенный в Mahout, который позволяет значительно уменьшить размер входных данных. Отметим, однако, что имеющийся в Carrot[2] алгоритм Lingo изначально построен на основе SVD, так что для Carrot2 специально ничего делать не надо.

Сингулярное разложение – это общий метод уменьшения числа признаков (т. е. не ограничивается одной лишь кластеризацией), предназначенный для уменьшения размерности (в задачах кластеризации текста каждое уникальное слово обычно представляется элементом n-мерного вектора) исходного набора данных путем оставления «существенных» признаков (слов) и отбрасывания несущественных. Это чревато потерей информации и потому несколько рискованно, но в общем случае позволяет добиться значительной экономии памяти и процессорных ресурсов. Что касается понятия существенности, то алгоритм часто сравнивают с извлечением концепций из корпуса текстов, но это не гарантируется. Для любителей математики отметим, что SVD сводится к отысканию собственных векторов и других характеристик некоторой матрицы (в случае кластеризации документы представляются в виде матрицы). Детали можно прочитать в другом

месте; например, на странице https://cwiki.apache.org/confluence/display/MAHOUT/ Dimensional+Reduction имеются дополнительные сведения о реализации в Mahout, а также ссылки на другие пособия и объяснения алгоритма SVD.

Для выполнения сингулярного разложения в Mahout нам снова пригодится командная утилита `bin/mahout`. Процедура запуска SVD в Mahout состоит из двух шагов. На первом шаге производится разложение матрицы, а на втором – итоговые вычисления. Первый шаг – разложение матрицы кластеризации (построенной ранее) – производится такой командой:

```
<MAHOUT_HOME>/bin/mahout svd --input /tmp/solr-clust-n2/part-out.vec \
--tempDir /tmp/solr-clust-n2/svdTemp \
--output /tmp/solr-clust-n2/svdOut \
--rank 200 --numCols 65458 --numRows 130103
```

Здесь мы видим уже привычные параметры, например местоположение входных векторов (`--Dmapred.input.dir`) и путь к временному каталогу, используемому системой (`--tempDir`), а также некоторые параметры, специфичные для SVD:

- `--rank` – ранг выходной матрицы;
- `--numCols` – общее число столбцов в векторе. В данном случае это количество уникальных термов в корпусе, которое приведено в первой строке файла /tmp/solr-clust-n2/dictionary.txt;
- `--numRows` – общее число векторов в файле. Используется для задания размеров структур данных.

Ранг определяет, как будет выглядеть результат, и именно эту величину труднее всего подобрать. Вообще говоря, подбирается ранг методом проб и ошибок – начинаем с небольшого числа (скажем, 50) и постепенно увеличиваем его. По словам Джейка Мэнникса (Mannix [2010, July]), одного из разработчиков Mahout и автора первоначальной версии кода SVD в Mahout, для задач обработки текста подходит ранг в диапазоне от 200 до 400. Понятно, что для определения оптимального для ваших данных ранга потребуется несколько попыток.

После завершения основного алгоритма SVD необходимо выполнить задачу очистки, которая сформирует окончательный результат:

```
<MAHOUT_HOME>/bin/mahout cleansvd \
--eigenInput /tmp/solr-clust-n2/svdOut \
--corpusInput /tmp/solr-clust-n2/part-out.vec \
--output /tmp/solr-clust-n2/svdFinal --maxError 0.1 \
--minEigenvalue 10.0
```

На этом шаге главное – выбрать пороговую погрешность (`--maxError`) и минимальное собственное значение (`--minEigenValue`). В качестве последнего всегда безопасно выбирать 0, но можете попробовать указать большее число. Что касается максимальной погрешности, то метод проб и ошибок в сочетании с заданием результата алгоритма кластеризации позволят понять, какое значение оптимально (детали см. в работе Mannix [2010, August]).

Как видим, существует много способов отобрать признаки или уменьшить размерность задачи. Как всегда в таких случаях, для определения оптимального варианта в конкретной ситуации требуется экспериментирование. И наконец, если вы хотите использовать результаты SVD для кластеризации (а ведь все ради того и затевалось, правда?), то остается сделать еще один шаг. Необходимо умножить транспонированную исходную матрицу на транспонированный результат SVD. Это также позволяет сделать утилита `bin/mahout` при задании соответствующих параметров. Оставляем проверку этого утверждения в качестве упражнения. Мы же пойдем дальше и рассмотрим некоторые количественные показатели функционирования Carrot² и Apache Mahout.

6.7.2. Производительность и качество Carrot²

Для настройки производительности и качества работы Carrot² предоставляет массу параметров, и это помимо главного – выбора самого алгоритма. Мы вкратце рассмотрим вопросы производительности, но полное обсуждение всех параметров оставим руководству по Carrot².

Выбор алгоритма в Carrot²

Сразу скажем, что у алгоритмов STC и Lingo есть общая черта: любой документ может принадлежать нескольким кластерам. В остальном они идут к достижению цели разными путями. Вообще говоря, Lingo порождает более качественные метки, чем STC, но зато работает гораздо медленнее, о чем свидетельствует рис. 6.3.

Как видно из рисунка, Lingo намного медленнее STC, но при небольшом размере результата ради качества меток можно пойти на увеличение времени работы. Кроме того, имейте в виду, что Carrot² можно скомпоновать с матричными библиотеками, написанными на машинном языке, что ускорит разложение матриц в Lingo. В приложениях, где производительность более важна, мы рекомендуем на-

чинать с STC. Если же важнее качество, пробуйте сначала Lingo. В любом случае потратьте некоторое время на подбор атрибутов, оптимальных для ваших данных. Полный перечень атрибутов Carrot[2] приведен по адресу http://download.carrot2.org/head/manual/index.html#chapter.components.

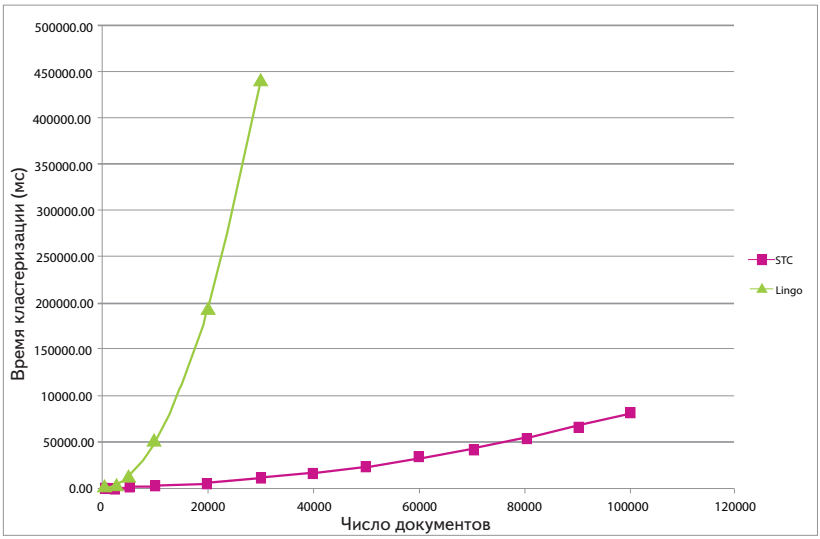

Рис. 6.3. Сравнение алгоритмов STC и Lingo для различного числа документов при запуске на данных Open Directory Project Data (http://www.dmoz.org)

6.7.3. Тесты производительности кластеризации в Mahout

Одна из самых сильных сторон Mahout – возможность распределять вычисления по нескольким компьютерам, чему он обязан Apache Hadoop. Для тестирования масштабируемости Mahout мы запускали алгоритм K-средних и другие алгоритмы кластеризации на нескольких экземплярах Amazon Elastic MapReduce (http://aws.amazon.com/elasticmapreduce/) и EC2, постепенно увеличивая количество экземпляров (машин).

Подготовка

Как было сказано в разделе 6.5.1, архивы электронной почты необходимо предварительно преобразовать в векторы Mahout. Подго-

товительные шаги можно выполнить на локальной машине, кластер Hadoop для этого не нужен. Входящий в дистрибутив Mahout скрипт prep_asf_mail_archives.sh (находится в каталоге utils/bin) делает следующее.

- Скачивает файлы из каталога s3://asf-mail-archives/ распаковывает их с помощью программы `tar`.
- Преобразует распакованные каталоги, содержащие почтовые архивы, сжатые с помощью gzip, в файлы последовательностей Hadoop, применяя специально написанную программу, основанную на утилите Mahout `seqdirectory` (см. файл `org.apache.mahout.text.SequenceFilesFromMailArchives` в исходном коде Mahout). В каждом файле находится много почтовых сообщений; мы отделяем их друг от друга и с помощью регулярных выражений извлекаем тему и текст. Все прочие почтовые заголовки пропускаются, потому что с точки зрения кластеризации они бесполезны. Каждое сообщение дописывается в конец файла последовательности с блочным сжатием. В результате получается 6 094 444 пары ключ-значение в 283 файлах, занимающих примерно 5,7 ГБ на диске.

Настройка HADOOP. Все описанные в этом разделе тесты выполнялись в версии Mahout 0.4 с применением Hadoop 0.20.2 на Amazon EC2. Точнее, мы работали с экземплярами EC2 типа `xlarge`, развернутыми с помощью скриптов contrib/ec2, включенных в дистрибутив Hadoop. Мы разместили на каждом узле три редуктора (mapred.reduce.tasks = n*3), выделив каждому дочернему процессу кучу размером не более 4 ГБ (mapred.child.java.opts = –Xmx4096M). Скрипты в каталоге contrib/ec2 выделяют дополнительный узел в роли NameNode, для которого мы не выделяли специального экземпляра, таким образом, в кластере из 4 узлов работает пять экземпляров EC2. Подробные инструкции по настройке кластера Hadoop для запуска Mahout имеются на вики-сайте Mahout по адресу https://cwiki.apache.org/confluence/display/MAHOUT/Use+an+Existing+Hadoop+AMI.

Векторизация содержимого

Файлы последовательностей необходимо преобразовать в разреженные векторы с помощью MapReduce-задачи Mahout `seq2sparse`. Мы остановились на разреженных векторах, потому что почтовые сообщения, как правило, короткие и в их совокупности встречается много уникальных термов. При использовании подразумеваемой по умолчанию конфигурации `seq2sparse` порождаются векторы с несколькими миллионами элементов, поскольку всякий уникальный

терм в корпусе документов представляется отдельным элементом в *n*-мерном векторе. Подвергать кластеризации векторы такой длины практически нереально, да и результат, скорее всего, оказался бы не особенно ценным из-за длинного хвоста уникальных термов в почтовом архиве.

Чтобы уменьшить число уникальных термов, мы разработали специальный анализатор Lucene, более агрессивный, чем подразумеваемый по умолчанию `StandardAnalyzer`. Конкретно, класс `MailArchivesClusteringAnalyzer` использует более широкий набор стоп-слов, исключает все лексемы, кроме алфавитно-цифровых, и применяет стемминг Портера. Мы задействовали также несколько способов уменьшения количества признаков, предлагаемых программой `seq2sparse`. Ниже показана команда, с помощью которой мы производили векторизацию:

```
bin/mahout seq2sparse \ --input
s3n://ACCESS_KEY:SECRET_KEY@asf-mail-archives/mahout-0.4/sequence-files/ \
--output /asf-mail-archives/mahout-0.4/vectors/ \
--weight tfidf \ --minSupport 500 \ --maxDFPercent 70 \
--norm 2 \ --numReducers 12 \ --maxNGramSize 1 \
--analyzerName org.apache.mahout.text.MailArchivesClusteringAnalyzer
```

Для получения входных данных мы использовали «родной» протокол Hadoop S3 Native (s3n), чтобы читать файлы последовательностей напрямую из файловой системы S3. Отметим, что в URI-адресе необходимо указывать ключ доступа и секретный ключ Amazon, только тогда Hadoop сможет получить доступ к корзине asf-mail-archives. Если вы не понимаете, что это такое, загляните на страницу, посвященную EC2, на вики-сайте Mahout.

> **Примечание автора.** Корзины asf-mail-archives больше нет из-за потенциальной возможности недобросовестного использования. Эта команда оставлена в тексте книги ради исторической точности, поскольку именно она применялась для получения показателей производительности Mahout 0.4, а также потому что у нас нет оплаченного доступа на Amazon, а прогон таких тестов обходится ой как недешево! Для получения аналогичных результатов вы можете воспользоваться более свежей версией открытых архивов почтовых данных, предоставляемой Amazon. Эти архивы находятся по адресу http://aws.amazon.com/datasets/7791434387204566.

Большинство параметров мы уже обсуждали, поэтому остановимся только на тех, которые важны для кластеризации.

- `--minSupport 500` – исключить термы, встречающиеся в корпусе документов менее 500 раз. Для корпусов меньшего размера значение 500 может оказаться слишком большим, из-за чего часть важных термов останется вне рассмотрения.
- `--maxDFPercent 70` – исключить термы, встречающиеся как минимум в 70 % документов. Это позволяет устранить термы, относящиеся к самой почтовой системе, которые прошли сквозь сито анализа текста.
- `--norm 2` – применять для нормирования векторов 2-норму, поскольку в качестве меры сходства на этапе кластеризации мы собираемся применять косинусоидальное расстояние.
- `--maxNGramSize 1` – рассматривать только одиночные термы.

При таких параметрах `seq2sparse` создала 6 077 604 вектора длиной 20 444 элементов каждый. На кластере из 4 узлов для этого понадобилось 40 минут. Количество векторов отличается от количества входных документов, потому что пустые векторы исключаются из результата `seq2sparse`. По завершении программ получившиеся векторы и словарные файлы копировались в открытую корзину S3, чтобы не создавать их заново при каждом прогоне задачи кластеризации.

Мы попробовали также сгенерировать биграммы (`--maxNGramSize=2`). К сожалению, при этом получились слишком большие векторы – примерно с 380 000 элементами. Кроме того, создание коллокаций термов заметно сказывается на скорости работы `seq2sparse`; на создание биграмм ушло 2 часа 10 минут, причем не менее половины этого времени было потрачено на вычисление коллокаций.

Результаты измерения производительности кластеризации методом K-средних

Чтобы начать кластеризацию, нам необходимо скопировать векторы из S3 в HDFS с помощью встроенной утилиты Hadoop `distcp`; как и раньше, для чтения из S3, мы используем протокол Hadoop S3 Native (s3n):

```
hadoop distcp -Dmapred.task.timeout=1800000 \
  s3n://ACCESS_KEY:SECRET_KEY@BUCKET/asf-mail-archives/mahout-0.4/
sparse-1-gram-stem/tfidf-vectors \
  /asf-mail-archives/mahout-0.4/tfidf-vectors
```

На это уйдет всего несколько минут, точное время зависит от размера кластера; при этом платить за передачу данных не придется, если кластер EC2 находится в регионе по умолчанию (в нашем случае

us-east-1). После того как данные будут скопированы в HDFS, запускаем кластеризацию методом K-средних с помощью следующей команды:

```
bin/mahout kmeans \ -i /asf-mail-archives/mahout-0.4/tfidf-vectors/ \
      -c /asf-mail-archives/mahout-0.4/initial-clusters/ \
      -o /asf-mail-archives/mahout-0.4/kmeans-clusters \
      --numClusters 60 --maxIter 10 \
      --distanceMeasure org.apache.mahout.common.distance.CosineD
istanceMeasure \
      --convergenceDelta 0.01
```

Эта задача сначала создает 60 случайно расположенных центроидов, применяя класс `RandomSeedGenerator`, на что уходит примерно 9 минут работы на одном главном сервере (эта задача не распределяется с помощью MapReduce). По завершении мы копируем начальные кластеры на S3, чтобы не создавать их каждый раз при прогоне тестов, коль скоро k не изменяется. В качестве `convergenceDelta` мы выбрали 0.01, чтобы во время тестирования было выполнено не менее 10 итераций; после 10 итераций 59 кластеров из 60 сошлись. Для показа первых десяти термов в каждом кластере мы воспользовались утилитой `clusterdump` (см. раздел 6.5.2).

Чтобы оценить масштабируемость реализации метода K-средних как задачи MapReduce, мы запускали кластеризацию в кластерах из 2, 4, 8 и 16 узлов с тремя редукторами в каждом узле. Во время работы средняя загрузка оставалась умеренной (<4), и ни один узел не начал выгружать виртуальную память в файл подкачки. График на рис. 6.4 показывает почти линейный рост производительности, на что мы и надеялись.

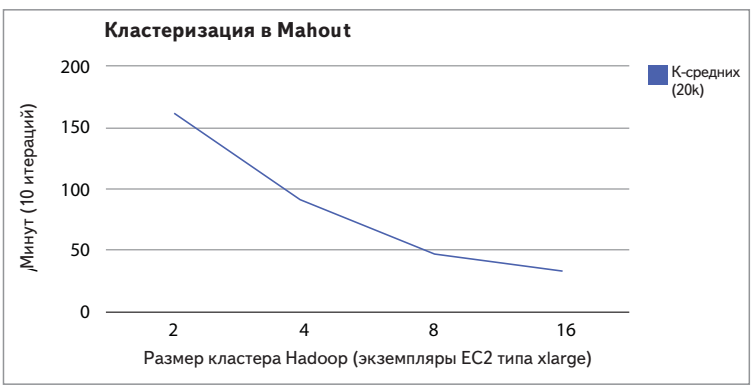

Рис. 6.4. График производительности метода K-средних при работе в облаке Amazon EC2 на кластере с числом узлов от 2 до 16

При каждом удвоении количества узлов мы наблюдаем почти двукратное уменьшение времени работы. Однако по мере увеличения числа узлов кривая медленно уплощается. Эта нелинейность объясняется тем, что некоторые узлы получают данные, требующие большего объема работы, а остальные вынуждены ждать, пока они закончат. Поэтому ресурсы недоиспользуются и чем больше в кластере узлов, тем чаще это будет происходить. Более того, при соблюдении двух условий можно предвидеть различие между выборками документов. Первое условие – векторы должны быть представлены разреженными структурами. Второе – у набора данных имеется длинный хвост признаков, что приводит к появлению больших векторов, на обработку которых уходит много времени. В нашем случае выполнены оба условия. Мы также пробовали выполнить ту же задачу на кластере из 4 узлов с большими экземплярами EC2 при двух редукторах на каждом узле. Мы ожидали, что в такой конфигурации задача закончится примерно за 120 минут, однако в действительности она заняла 137 минут и средняя загрузка системы не опускалась ниже 3.

Результаты измерения производительности других алгоритмов кластеризации в Mahout

Мы экспериментировали и с другими алгоритмами кластеризации, имеющимися в Mahout, в том числе нечетким алгоритмом K-средних, купольной кластеризацией и методом Дирихле. В общем и целом, мы не смогли получить неоспоримых результатов на имеющемся наборе данных. Так, одна итерация нечеткого алгоритма K-средних работает в среднем в 10 раз медленнее, чем одна итерация алгоритма K-средних. Однако считается, что нечеткий алгоритм сходится быстрее обычного, т. е. может потребоваться меньше итераций. Сравнение нечеткого алгоритма K-средних и двух вариантов обычного алгоритма K-средних изображено на рис. 6.5.

В ходе экспериментов мы использовали четыре сверхбольших экземпляра. Для выполнения всех 10 итераций нечеткого алгоритма K-средних с 60 кластерами и параметром сглаживания $m = 3$ потребовалось 14 часов (848 минут). Это примерно в 10 раз дольше, чем алгоритм K-средних, который занял всего 91 минуту. Мы заметили, что первая итерация в обоих алгоритмах неизменно выполнялась гораздо быстрее всех остальных. А на вторую, третью и последующие итерации затрачивалось примерно одинаковое время. Поэтому мы взяли характеристики второй итерации для оценки времени выполнения всех 10 итераций при различных k. При увеличении числа кластеров

в 10 раз можно ожидать пропорционального замедления. Точнее, мы оценили время работы при k = 600 в 725 минут. Это несколько меньше, чем 10×91, потому что увеличение k позволяет более эффективно амортизировать постоянные накладные расходы на обработку. Различие между первой и второй итерациями можно объяснить тем, что на первой итерации в качестве центроидов используются случайные векторы. А на последующих итерациях центроиды гораздо плотнее и требуют более длительных вычислений.

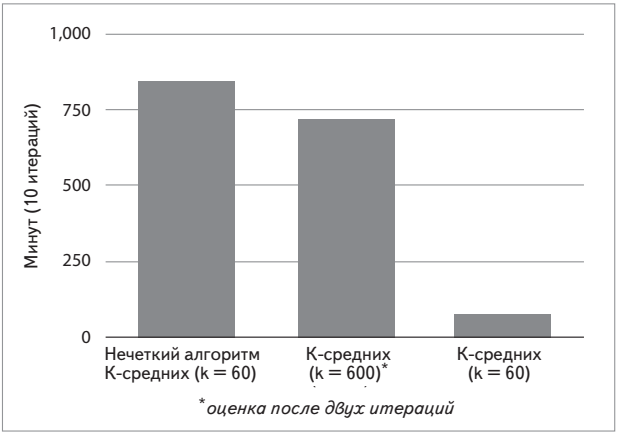

Рис. 6.5. Сравнение времени выполнения различных алгоритмов кластеризации

Купольный алгоритм Mahout (Canopy) потенциально полезен в качестве шага предобработки для определения числа кластеров в большом наборе данных. При использовании этого алгоритма пользователь должен определить всего два порога, которые влияют на расстояния между созданными кластерами. Мы обнаружили, что купольный алгоритм порождает нетривиальный набор кластеров при T1 = 0,15 и T2 = 0,9. Но время, потраченное на отыскание этих значений, похоже, не окупается ускорением других алгоритмов. Кроме того, имейте в виду, что на момент написания этой книги Mahout находился на уровне версии pre-1.0, и вполне вероятно, что с увеличением количества пользователей этого кода скорость возрастет.

При использовании алгоритма Дирихле мы столкнулись с трудностями, но благодаря помощи сообщества Mahout смогли завершить одну итерацию с параметрами alpha0 = 50 и modelDist = L1ModelDistribution. При большем значении alpha0 возрастает веро-

ятность выбрать новый кластер уже на первой итерации, что помогает распределить рабочую нагрузку на последующих итерациях. К сожалению, последующие итерации все же не завершились за разумное время, потому что на предыдущих итерациях слишком много данных было отнесено к небольшому числу кластеров.

Итоги тестов производительности и дальнейшие шаги

Приступая к тестированию, мы надеялись сравнить производительность различных имеющихся в Mahout алгоритмов кластеризации на большом наборе документов и дать рецепт крупномасштабной кластеризации с применением облака Amazon EC2. Мы выяснили, что реализация алгоритма K-средних линейно масштабируется при кластеризации миллионов документов примерно с 20 000 признаков. Для других способов кластеризации мы не смогли получить сравнимых результатов и пришли к единственному выводу – нужно продолжать эксперименты и одновременно совершенствовать код Mahout. Тем не менее, мы документировали наши опыты по настройке EC2 и Elastic MapReduce на вики-сайте Mahout, чтобы другие могли воспользоваться плодами нашей работы.

6.8. Благодарности

Авторы благодарят за помощь в написании этой главы Тэда Даннинга (Ted Dunning), Джейка Мэнникса (Jake Mannix), Станислава Осиньски (Stanisław Osiński) и Давида Вайсса (Dawid Weiss). Тестирование производительности Mahout на Amazon Elastic MapReduce и EC2 стало возможным благодаря финансовой поддержке в рамках программы тестирования проектов Apache с помощью веб-служб Amazon.

6.9. Резюме

Кластеризация является эффективным способом выявления информации в самых разных задачах: сокращение количества просматриваемых новостей, быстрое реферирование результатов поиска при неоднозначных поисковых термах или идентификация тем в больших наборах документов. В этой главе мы обсудили различные концепции, лежащие в основе кластеризации, в том числе некоторые факторы, определяющие выбор подхода. Затем мы рассмотрели практические примеры использования Carrot[2] и Apache Mahout для кластеризации

результатов поиска, документов и слов по темам. И закончили эту главу обсуждением приемов повышения производительности, в том числе алгоритма сингулярного разложения, включенного в Mahout.

6.10. Ресурсы

Blei, David; Lafferty, John. 2009. «Visualizing Topics with Multi-Word Expressions». http://arxiv.org/abs/0907.1013v1.

Blei, David; Ng, Andrew; Jordan, Michael. 2003. «Latent Dirichlet allocation». *Journal of Machine Learning Research*, 3:993–1022, January.

Carpineto, Claudio; Osiński, Stanisław; Romano, Giovanni; Weiss, Dawid. 2009. «A Survey of Web Clustering Engines». *ACM Computing Surveys*.

Crabtree, Daniel; Gao, Xiaoying; Andreae, Peter. 2005. «Standardized Evaluation Method for Web Clustering Results». The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05).

Cutting, Douglass; Karger, David; Pedersen, Jan; Tukey, John W. 1992. «Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections». Proceedings of the 15th Annual International ACM/SIGIR Conference.

Dash, Manoranjan; Choi, Kiseok; Scheuermann, Peter; Liu, Huan. 2002. «Feature Selection for Clustering – a filter solution». Second IEEE International Conference on Data Mining (ICDM'02).

Dash, Manoranjan, and Liu, Huan. 2000. «Feature Selection for Clustering». Proceedings of Fourth Pacific-Asia Conference on Knowledge Discovery and Data Mining.

Dean, Jeffrey; Ghemawat, Sanjay. 2004. «MapReduce: Simplified Data Processing on Large Clusters». OSDI'04: 6th Symposium on Operating Systems Design and Implementation. http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf.

Deerwester, Scott; Dumais, Susan; Landauer, Thomas; Furnas, George; Harshman, Richard. 1990. «Indexing by latent semantic analysis». *Journal of the American Society of Information Science*, 41(6):391–407.

Dunning, Ted. 1993. «Accurate methods for the statistics of surprise and coincidence». *Computational Linguistics*, 19(1).

Google News. 2011. http://www.google.com/support/news/bin/answer.py?answer= 40235&topic=8851.

Liu, Tao; Liu, Shengping; Chen, Zheng; Ma, Wei-Ying. 2003. «An evaluation on feature selection for text clustering». Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003).

Manning, Christopher; Raghavan, Prabhakar; Schütze, Hinrich. 2008. *An Introduction to Information Retrieval.* Cambridge University Press.

Mannix, Jake. 2010, July. «SVD Memory Reqs». http://mail-archives.apache.org/mod_mbox/mahout-user/201007.mbox/%3CAANLkTik-uHrN2d838dHfYwOhxHDQ3bhHkvCQvEIQCLT@mail.gmail.com%3E.

Mannix, Jake. 2010, August. «Understanding SVD CLI inputs». http://mail-archives.apache.org/mod_mbox/mahout-user/201008.mbox/%3CAANLkTi= ErpLuaWK7Z-2an786v5AsX3u5=adU2WJM5Ex7@mail.gmail.com%3E.

Steyvers, Mark, and Griffiths, Tom. 2007. «Probabilistic Topic Models». *Handbook of Latent Semantic Analysis.* http://citeseerx.ist.psu.edu/viewdoc/download?doi-0.=10.1.1.80.9625&rep=rep1&type=pdf.

Zamir, Oren, and Etzioni, Oren. 1998. «Web document clustering: a feasibility demonstration». Association of Computing Machinery Special Interest Group in Information Retrieval (SIGIR).

ГЛАВА 7. Классификация, категоризация и пометка

В этой главе:

- Основные концепции классификации, категоризации и пометки.
- Как категоризация применяется в приложениях обработки текста.
- Построение, обучение и оценка классификаторов с помощью программ с открытым исходным кодом.
- Интеграция категоризации с приложением.
- Построение системы рекомендования меток, обученной на помеченных данных.

Скорее всего, бродя по разным сайтам, вы встречали слово «метка» (тег, tag). Фотографии, видео, музыка, новости, статьи в блогах, твиты – все они часто сопровождаются словами или словосочетаниями, содержащими краткое описание предмета и ссылку на связанные объекты. Возможно, вам попадались облака меток: перечень набранных шрифтами разного размера слов, описывающих темы, жанры фильмов или музыкальные стили, предпочитаемые автором. Метки расплодились по всей сети и используются как средство навигации или для организации всего чего угодно – от новостей до закладок (рис. 7.1).

Метки – это данные, описывающие другие данные. Иначе их называют *метаданными*. Их можно применить к содержимому любого вида. Метки бывают как неструктурированными – простой список релевантных ключевых слов или имен пользователей, так и строго структурированными – рост, вес, цвет глаз.

Рис. 7.1. Метки в твите. Хэштеги, начинающиеся знаком #, служат для идентификации ключевых слов в твите, а метки, начинающиеся знаком @, ссылаются на других пользователей

Как создаются метки? Некоторые вводит человек. Автор или куратор присваивает описательные слова – иногда первое, что приходит на ум, а иногда тщательно отобранные из перечня одобренных слов и словосочетаний. Бывает и так, что метки присваивают пользователи сайта. Любой человек может пометить объект словами, которые, с его точки зрения, имеют смысл. Книга, песня или иное содержимое описываются так, как его воспринимают сотни и тысячи людей, при этом решающими оказываются мудрость или тупость толпы.

Методы машинного обучения позволяют генерировать метки автоматически или полуавтоматически, исходя из содержимого. Для анализа меток и предложения меток для еще не помеченного содержимого или альтернатив существующим меткам применяются алгоритмы. Такой вид автоматизированной пометки – это частный случай классификации.

Задача классификации формулируется просто: пусть дан объект, требуется отнести его к одной или нескольким предопределенным категориям. Для решения этой задачи необходимо принять во внимание свойства объекта. Чем он похож на другие объекты? Как сгруппировать объекты с общими свойствами и исключить отличающиеся от них?

Алгоритмы классификации обучаются на примерах с использованием данных, отнесенных к классам вручную или с помощью какого-то иным автоматизированного процесса. В ходе обучения алгоритм классификации определяет, какие свойства (или признаки) указывают на принадлежность объекта к данному классу. Обученный алгоритм может затем классифицировать данные, которых раньше не видел.

В главе 6 мы рассматривали другой класс обучаемых алгоритмов – *алгоритмы кластеризации*. Классификация и кластеризация – две стороны одной медали. Те и другие алгоритмы стремятся приписать объектам метки, исходя из признаков объектов. Но классификация отличается от кластеризации тем, что множество меток заранее опре-

делено и требуется наилучшим образом уложить объекты в эту схему. Такой подход называется *обучением с учителем* – метки, назначаемые алгоритмом классификации, основаны на внешних входных данных, например, заранее определенном человеком множестве меток. Кластеризация – пример обучения без учителя, когда предопределенного множества меток нет. Задача кластеризации – сгруппировать объекты, исходя из общих характеристик. Несмотря на это различие, оба вида алгоритмов – классификация и кластеризация – применяются для категоризации документов.

Категоризация документов заключается в том, чтобы приписать документу метку, обозначающую его категорию или тему. Это одно из применений алгоритмов классификации. В начале процесса категоризации имеется набор обучающих документов-примеров, каждый из которых отнесен к одной или нескольким категориям, или тематическим рубрикам. Алгоритм классификации строит модель, описывающую, как отдельные термы либо иные признаки документа, например длина или структура, соотносятся с рубриками. По завершении обучения модель можно использовать для назначения рубрик – иными словами, для категоризации нового документа.

Мы начнем эту главу с обзора классификации и категоризации и обсудим процесс обучения классификатора для использования в производственной системе. Затем мы рассмотрим несколько алгоритмов классификации и посмотрим, как они применяются для автоматической категоризации и пометки текстовых документов. Одни алгоритмы, например наивный байесовский классификатор и метод максимальной энтропии, основаны на статистических моделях, в других, например в методе k-ближайших соседей и в категоризаторе на основе TF-IDF, используется векторная модель, описанная в главе 3 при рассмотрении информационного поиска.

Эти алгоритмы будут представлены на ряде практических примеров с применением командных утилит и кода из таких открытых проектов, как OpenNLP, Apache Lucene, Solr и Mahout. В каждом случае мы пройдем все шаги создания классификатора. Мы изучим различные подходы к получению и подготовке обучающих данных, обучим классификаторы с помощью разных алгоритмов и посмотрим, как оценивать качество результатов. Наконец, мы покажем, как классификаторы интегрируются в производственную систему. Дочитав главу до конца, вы сможете модифицировать рассмотренные примеры под свои нужды и создать автоматический категоризатор и рекомендатель меток для собственных приложений.

7.1. Введение в классификацию и категоризацию

С вычислительной точки зрения, задача классификации заключается в назначении данным меток. Имея набор признаков, классификатор пытается присвоить метку объекту. Для этого классификатор пользуется знаниями, полученными при изучении примеров уже помеченных объектов. Эти примеры, или *обучающие данные* служат источником априорных знаний, на основе которых классификатор принимает решения об объектах, которых раньше не видел.

Категоризация – это частный случай классификации. Ее задача – отнести объект к некоторой категории. Другие алгоритмы классификации могут просто ответить да или нет, как, например, детектор мошеннических транзакций по кредитным картам. Алгоритмы же категоризации призваны поместить объект в одну из небольшого множества категорий, например, решить, является ли автомобиль седаном, купе, внедорожником или минивэном. Многие рассматриваемые в этой главе идеи относятся к классификации в целом, но некоторые более тесно связаны именно с категоризацией. Мы будем употреблять эти термины как синонимы.

Категоризация документов в том смысле, который является предметом настоящей главы, – это процесс отнесения к категориям текстовых документов. В примерах мы будем назначать документам категории, связанные с их тематикой, но возможны и другие виды категоризации, например анализ тональности, призванный определить, является ли отзыв о товаре позитивным или негативным, либо выявить эмоциональную окраску почтового сообщения или запроса клиента в службу поддержки.

Чтобы понять, как производится автоматическая классификация, подумайте, чем вертолет отличается от самолета. Скорее всего, никто явно не говорил вам, что «наличие горизонтально вращающихся лопастей» или «отсутствие неподвижных крыльев» – признаки, отличающие вертолет от самолета. Но увидев однажды примеры обоих летательных аппаратов, вы, без сомнения, в дальнейшем сможете отличить один от другого. Бессознательно вы сумели выделить признаки *вертолета*, использовать их для идентификации других вертолетов и понять, что аппарат, у которого на крыльях размещены реактивные двигатели, вертолетом не является. С другой стороны, увидев летательный аппарат с горизонтально расположенным винтом, вы

сразу скажете, что это вертолет. Алгоритмы классификации действуют по тому же принципу.

Из этого примера видна также важность отбора признаков для различения классов объектов. И вертолеты, и самолеты летают и перевозят людей. Эти признаки не помогут отличить вертолет от самолета, поэтому использовать их при обучении классификатора бессмысленно. Или допустим, что увиденный вами вертолет был желтым, а самолет – синим. Если вы никогда не видели никаких других самолетов и вертолетов, то могли бы решить, что все самолеты синие, а все вертолеты желтые. Но собственный опыт постижения мира подсказывает, что цвет – не то, что можно использовать для различения самолета и вертолета. Не обладающий такими знаниями алгоритм классификации, который учитывает цвет при принятии решений, ошибется. Это показывает, насколько важно обучать алгоритм на широком наборе примеров, охватывающих максимальное много признаков и их сочетаний.

Алгоритмы классификации обучаются на примерах, а обучающие данные разбиты на классы вручную или с помощью какого-то автоматизированного процесса. Анализируя связи между признаками и классами, алгоритм учится понимать, какие признаки существенны для правильного присвоения метки, а какие несут несущественную или уводящую в сторону информацию. Результатом процесса обучения является модель, которая впоследствии используется для классификации ранее не помеченных объектов. Классификатор выделяет признаки объектов, подлежащих классификации, и с помощью модели определяет наилучшие метки для каждого объекта. В зависимости от алгоритма классификации может порождаться одна или несколько меток, снабженных оценкой, или вероятностью, благодаря чему метки можно ранжировать.

Есть много типов алгоритмов классификации. Одной из отличительных особенностей алгоритма является порождаемый им результат. Существуют двоичные алгоритмы, которые дают ответ да или нет. Но есть и алгоритмы, выдающие несколько результатов, взятых из дискретного набора категорий, или результат из непрерывного диапазона, например, число с плавающей точкой – оценку или вероятность.

Двоичный классификатор сообщает, принадлежит ли рассматриваемый объект некоторому классу. Простейший пример такого рода – фильтр спама. Этот классификатор анализирует признаки, присутствующие в почтовом сообщении, и решает, спам это или не спам. В байесовских алгоритмах классификации, которые мы рас-

смотрим в разделе 7.4 и которые чаще всего применяются для распознавания спама, строится статистическая модель, позволяющая определить, является объект членом некоторого класса или нет. Еще один двоичный алгоритм классификации – *машина опорных векторов* (SVM) – пытается найти прямую или *n*-мерную плоскость, так называемую *гиперплоскость*, которая делит пространство признаков на две части: образцы, принадлежащие и не принадлежащие классу.

Иногда двоичные классификаторы комбинируются для выполнения многоклассовой классификации. Каждому классу сопоставляется один двоичный классификатор, и образец предъявляется каждому классификатору, чтобы определить, в какие классы он попадает. В зависимости от алгоритма результатом может быть один класс, в который образец попадает с наибольшей вероятностью, или несколько классов, в которые входит образец, причем каждому классу сопоставляется некоторая величина, характеризующая вероятность попадания в него объекта. Байесовский классификатор из проекта Mahout, рассматриваемый в этой главе, дает пример обучения нескольких двоичных классификаторов с целью назначения категорий по схеме многоклассовой классификации.

Множественные двоичные классификаторы иногда организуются в виде дерева. В таком случае документ, попавший в класс A, у которого есть дочерние классы B и C, будет оцениваться с помощью классификаторов, обученных для каждого класса B и C. Если он соответствует B, то будет оценен с помощью потомков B – E и F. Если документ отвергнут классификаторами E и F, то он останется в классе B, в противном случае будет отнесен к категории самого нижнего уровня, которой соответствует. Этот подход полезен, когда классы естественным образом образуют иерархию, например, в случае тематической рубрикации. Варианты иерархического подхода с успехом применяются в таких методах, как двоичные деревья решений и случайные леса.

Категоризатор документов по методу максимальной энтропии, который будет рассматриваться в разделе 7.5, – пример многоклассового алгоритма классификации. В нем в качестве признаков используются встречающиеся в документе слова, а в качестве категорий – предметные рубрики. В процессе обучения строится модель связей между словами и рубриками. Для еще не категоризованного документа с помощью модели определяются веса признаков, которые в конечном счете служат для выдачи результата, описывающего предмет документа.

В разделе 7.3 мы изучим алгоритмы категоризации документов, основанные на свойствах векторной модели, обсуждавшейся в главе 3. В них вычисляются расстояния между предъявленным новым документом и всеми ранее классифицированными и на этой основе производится его классификация. В этом контексте некатегоризованный документ становится запросом, по которому отбираются классифицированные документы или документы, репрезентативные для содержимого каждой категории. Мы рассмотрим этот подход, а также пример, в котором Lucene используется как механизм индексирования обучающих данных и поиска документов, удовлетворяющих такому запросу.

Существуют разнообразные алгоритмы классификации документов, и многие из них неплохо справляются с задачей категоризации или пометки документов. В этой главе мы представим несколько алгоритмов, которые легко реализовать с помощью библиотек из проектов с открытым исходным кодом. Приведенные примеры могут стать отправной точкой для дальнейших изысканий в области методов классификации. Многое из сказанного относится как к классификации, так и к другим задачам обучения с учителем независимо от принятого подхода или применяемого алгоритма. Рассматривая примеры, мы будем иметь в виду эти сквозные вопросы, в том числе: подготовка обучающих данных, идентификация признаков и оценка качества классификации. Каждый пример посвящен отдельному подходу к категоризации или пометке, но основывается на предыдущих примерах в этой главе и других местах книги.

7.2. Процесс классификации

Процесс разработки автоматического классификатора не зависит от используемого алгоритма. Он состоит из нескольких этапов (рис. 7.2): подготовка, обучение, тестирование и эксплуатация. Зачастую этот процесс приходится повторять несколько раз, автоматически или вручную, чтобы настроить поведение классификатора для получения оптимальных результатов. Суть настройки заключается в том, чтобы использовать результаты этапа тестирования для совершенствования обучения. Уже после внедрения классификатора в эксплуатацию бывает необходимо расширить его, включив случаи, которые не встречались в исходных обучающих данных.

На *этапе подготовки* готовятся данные для процедуры обучения. Выбирается множество меток, которые классификатор должен уметь

распознавать, способ идентификации признаков, используемых для обучения и те элементы набора данных, которые следует зарезервировать для тестирования. Затем данные должны быть преобразованы в формат, который понимает алгоритм обучения.

Подготовив данные, мы переходим к *этапу обучения*, на котором алгоритм обрабатывает все помеченные примеры и учится сопоставлять признаки с метками. Каждый признак ассоциирован с меткой, присвоенной документу, а алгоритм обучения моделирует эту связь между признаками и метками классов. Существуют многочисленные алгоритмические подходы к обучению моделей классификации, но в конечном итоге алгоритм должен научиться распознавать как признаки, которые важны для различения классов, так и те, которые мало что дают для этого. Часто алгоритм классификации принимает параметры, управляющие построением модели. Во многих случаях подходящие значения параметров сначала выбираются, исходя из некоторой обоснованной гипотезы, а затем итеративно уточняются.

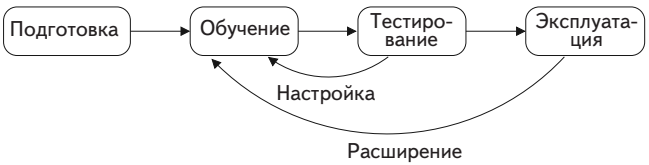

Рис. 7.2. Этапы процесса разработки автоматического классификатора: подготовка данных, обучение модели, тестирование модели, развертывание классификатора в производственной системе. В ходе настройки классификатор модифицируется с учетом полученных в ходе тестирования результатов. Расширение – это распространение классификатора на новые случаи уже после развертывания

На *этапе тестирования* алгоритм классификации оценивается путем предъявления дополнительных примеров – *тестовых данных*. В процессе оценки класс, которому действительно принадлежит пример, сравнивается с тем, который присвоил классификатор. Точность алгоритма обучения определяется путем подсчета правильных и неправильных решений. Некоторые алгоритмы выводят ход «рассуждений» на этапе тестирования, чтобы человек понимал, как интерпретируются обучающие данные. Эту информацию затем используют в качестве обратной связи для настройки параметров и методик, применяемых на этапах подготовки и обучения.

На протяжении жизни классификатора его обучение может производиться неоднократно. На стадии начальной разработки классифи-

катора этапы обучения и тестирования обычно повторяют несколько раз, настраивая процесс обучения и (хочется надеяться) получая все лучшие и лучшие результаты. Многие алгоритмы классификации, в частности метод максимальной энтропии, рассматриваемый в разделе 7.5, спроектированы так, чтобы автоматически повторять процесс обучения, пока алгоритм не сойдется к наилучшему ответу. Есть и другие подходы, когда итерации повторяются вручную. В некоторых случаях итеративное обучение распараллеливается, так что одновременно обучаются несколько вариантов классификатора. Затем выбирается и внедряется классификатор, который дает наилучшие результаты.

Уже после внедрения классификатора в эксплуатацию бывает необходимо переобучить его, расширив знания о предметной области. Часто так происходит, когда со временем вырабатывается словарь, играющий ключевую роль в различении классов. Взять, к примеру, классификатор, который распределяет отзывы о товаре по темам. После выпуска новой продукции классификатор должен увидеть примеры помеченных документов, содержащих названия изделий, иначе он не сможет определить, что слово *android*, скорее всего, относится к смартфону, а слово *ipad* – к мобильному вычислительному устройству.

Разобравшись с основными этапами процесса классификации, обсудим более подробно, какие проблемы приходится решать на каждом этапе.

7.2.1. Выбор схемы классификации

Алгоритмы классификации обучаются на примерах данных, которые были распределены по классам вручную или с помощью какого-то автоматизированного процесса. Классы или категории, назначаемые объектам, имеют имена и семантику, отдельные от классифицируемого объекта. Каждый класс существует в контексте других классов, и вся такая система называется *схемой классификации* или *категоризации*. Одни схемы классификации жесткие, т. е. объекту позволено попадать только в один класс. Другие – более гибкие, поскольку признают тот факт, что в реальном мире у объектов могут быть различные аспекты, или фасеты. Некоторые схемы классификации, например система классификации биологических организмов Линнея, по природе своей иерархичны. У других нет никакой особой структуры, кроме подразумеваемой лингвистическими связями, как, например, у простых ключевых слов-меток на сайте Flickr или Technorati. Схемы классификации существенно различаются по охвату. Они

могут охватывать широкую предметную область, как десятичная система Дьюи, применяемая в библиотечном деле, или какую-либо узкоспециализированную область, как, скажем, схема для описания технических средств для инвалидов.

Во многих контекстах схема классификации возникает и развивается вполне органично. На таком сайте социальных закладок, как delicious.com, метки определяются тем, как они используются для классификации веб-страниц. Пользователи сами выбирают слова, описывающие страницу, и это действие повторяется миллионы раз в день. Набор слов, применяемых в схеме классификации, постоянно эволюционирует, а их семантика имеет тенденцию непрерывно изменяться в зависимости от способа использования меток. Схема классификации возникает и развивается со временем снизу вверх в отличие от нисходящих подходов к классификации, когда пользователю предлагается заранее определенный, возможно, иерархически организованный набор тем.

Выбор схемы классификации для приложения – это вопрос оценки компромиссов. Восходящие схемы, основанные на метках, отдают предпочтение простоте в ущерб точности, но могут столкнуться с лингвистическими проблемами, когда у метки несколько смысловых значений или одна и та же концепция описывается разными словами. Или еще проще – один пользователь употребляет для аннотирования ресурса слово во множественном числе, а другой ищет по слову в единственном числе. У нисходящих схем на основе иерархической рубрикации таких проблем может не быть, они предлагают авторитетное представление пространства классов. Однако они испытывают трудности с адаптацией новых словников или смысловых значений.

7.2.2. Отбор признаков для категоризации

В главе 2 мы обсудили несколько подходов к предобработке текста для извлечения слов, которые будут использоваться на последующих этапах. Извлеченные из текста слова считаются *признаками* текста. В качестве таковых они используются в алгоритмах классификации для определения того, к какому классу или категории отнести документ, в котором они встречаются.

Простейший подход, называемый *мешок слов*, заключается в том, чтобы рассматривать документ просто как множество слов. Каждое встречающееся в документе слово считается признаком, и этим признакам присваиваются веса в соответствии с частотой вхождения. Для вычисления веса каждого слова используется представленная в

главе 3 схема взвешивания TF-IDF, так что важность встречающихся в документе слов зависит от того, насколько часто слово встречается во всем обучающем корпусе. Если корпус очень велик, то в качестве набора признаков для построения классификатора, возможно, придется взять какое-то подмножество термов. Исключая слова, которые встречаются очень часто или имеют низкую частоту IDF, мы можем обучить классификатор на самых важных словах во всем корпусе – тех, которые позволяют с наибольшей уверенностью разграничить категории. В главе 3 были также описаны другие схемы назначения весов, которые можно использовать для отбора признаков классификации.

Словосочетания также могут служить полезными признаками документов. Вместо того чтобы рассматривать каждое слово как признак, мы используем *n*-граммы для улавливания важных словосочетаний. Категория обычно содержит уникальные определяющие ее словосочетания, например: *title insurance* (страхование правового титула), *junk bond* (бросовая облигация) или *hard disk* (жесткий диск). Отбор всех вообще словосочетаний в корпусе может привести к комбинаторному взрыву признаков, поэтому алгоритмы строят так, чтобы выявить статистически значимые словосочетания, которые называются коллокациями, и отбросить те, что не имеют ценности.

Для построения классификаторов могут быть полезны и другие характеристики документа, помимо содержимого. Документы, входящие в корпус, могут обладать свойствами, способными повысить качество алгоритма категоризации. Например, часто представляют интерес авторы и источники. Тот факт, что документ опубликован в японской газете, позволяет предположить, что он скорее попадает в категорию «бизнес в Азии». Одни авторы часто пишут на спортивные темы, другие – о технологиях. Длина документа может служить фактором, отличающим статьи в научном журнале от почтового сообщения или твита.

Для отбора признаков из документа можно привлекать и дополнительные ресурсы. Такой лексический ресурс, как WordNet[1], можно использовать для расширения термов за счет синонимов или гиперонимов для основных признаков документа. Часто алгоритмически извлекаются именованные сущности (см. главу 5), например *Camden Yards*[2] или *Baltimore Orioles*[3], которые тоже добавляются в качестве

[1] Электронный тезаурус для английского языка. – *Прим. перев.*

[2] Бейсбольный стадион в пригороде Балтимора. – *Прим. перев.*

[3] Профессиональный бейсбольный клуб. – *Прим. перев.*

признаков, позволяющих отнести статью к категории «Спорт». Наконец, результаты алгоритмов кластеризации или других классификаторов могут подаваться на вход нового классификатора и использоваться для определения категорий.

При таком изобилии возможностей с чего же лучше начать? Можно продвинуться довольно далеко, применяя только подход «мешок слов» в сочетании со схемой назначения весов TF-IDF, основанной на стандартной векторной модели. В примерах из этой главы мы начнем именно с этого и по ходу дела изучим другие подходы к отбору признаков. Алгоритмы, конечно, играют важную роль в точности системы автоматической категоризации, но именно от выбора признаков зависит успех или провал всего дела, какой бы алгоритм ни применялся.

7.2.3. Важность обучающих данных

Точность классификатора определяется признаками, на которых он обучен, а также качеством и количеством предъявленных во время обучения примеров. Не имея достаточного количества примеров, классификатор не сможет установить связь между признаками и категориями, поэтому в процессе обучения будут сделаны неправильные предположения о связях. Такой классификатор не сможет провести различие между категориями или предположит, что признак соотносится с некоторой категорией, хотя на самом деле это не так. Например, вы интуитивно понимаете, что цвет не является определяющей характеристикой, позволяющей отличить вертолет от самолета, но если алгоритм классификации видел только желтые самолеты и не видел ни одного желтого вертолета, то он может «поверить», будто все летательные аппараты желтого цвета – самолеты. Наличие полного и сбалансированного набора обучающих данных, включающего как можно больше релевантных признаков, а также равномерно распределенные обучающие примеры, очень важно для создания точной модели.

Но откуда взять обучающие данные? Один из возможных подходов – вручную назначить данным классы. Новостные агентства, например Рейтерс, потратили немало времени и сил на ручную пометку циркулирующих материалов. Метки, вручную присвоенные веб-страницам миллионами пользователей сайта delicious.com, тоже можно рассматривать как источник аннотированных страниц.

Существует также возможность получить обучающие данные с помощью автоматизированных процессов. В разделе 7.4, говоря о байе-

совском классификаторе, реализованном в Mahout, мы покажем, как поиск по ключевым словам применяется для получения документов, относящихся к предметной области. С классом ассоциируется одно или несколько ключевых слов, а затем производится поиск документов, содержащих эти слова. Из найденных документов извлекаются признаки, которые и определяют природу категории. Этот процесс, именуемый *бутстрапингом*, способен порождать классификаторы, дающие довольно точные результаты.

В Интернете нет недостатка в данных, полезных для обучения классификатора. Википедия и сходные проекты, например Freebase, выкладывают в свободный доступ выгруженные данные[4], которые можно расценивать как гигантский корпус документов с уже проставленными категориями, метками и прочей полезной информацией.

Для исследований по машинному обучению также имеется немало тестовых наборов. Они полезны, когда требуется воспроизвести результаты исследований или сравнить разные подходы. Многие наборы разрешено использовать только в некоммерческих целях и давая ссылку на источник, тем не менее это отличный способ с ходу приступить к работе по изучению различных аспектов классификации. Заодно вы получаете эталон, сравнение с которым показывает, в правильном ли направлении вы движетесь.

Один из самых известных исследовательских тестовых наборов – набор RCV1-v2/LYRL2004 для тестирования качества категоризации текстов (см. Lewis [2004]). Он содержит более 800 000 материалов, вручную классифицированных информационным агентством Рейтерс. В сопроводительной записке к этому набору приводится полное его описание, излагается методика обучения и результаты, полученные с помощью нескольких хорошо известных подходов к классификации текстов. До выхода версии RCV2 в открытом доступе находился другой тестовый набор Рейтерс – Reuters-21578 – который также широко используется. И хотя в нем гораздо меньше файлов, он все еще ценен как эталонный набор новостных материалов вследствие значительного числа основанных на нем опубликованных работ. Еще один тестовый набор под названием 20 Newsgroups (доступен по адресу http://people.csail.mit.edu/jrennie/20Newsgroups/) насчитывает примерно 11 000 статей из 20 новостных групп в Интернете, охватывающих такие разные темы, как компьютеры, спорт, автомобили,

[4] Данные, выгруженные из Википедии, доступны по адресу http://en.wikipedia.org/wiki/Wikipedia:Database_download, из Freebase – по адресу http://wiki.freebase.com/wiki/Data_dumps.

политика и религия. Он полезен в качестве небольшого, но хорошо организованного корпуса документов для обучения и тестирования.

Stack Exchange, родительская компания Stack Overflow (http://www.stackoverflow.com) и многих других социальных вопросно-ответных сайтов, предоставляет выгруженные данные по лицензии Creative Commons (архив находится по адресу http://blog.stackoverflow.com/category/cc-wiki-dump/). Все заданные на сайте Stack Overflow вопросы помечены ключевыми словами, выбранными пользователями. Выгруженные данные содержат все метки и потому могут служить великолепным источником обучающих данных. В разделе 7.6.1 мы воспользуемся подмножеством этого набора для построения своего рекомендателя.

Если необходимых данных в выгруженном виде нет, но в Интернете они все же имеются, то нередко разрабатывают специализированный робот, который собирает данные, нужные для обучения классификатора. Некоторые крупные компании, например Amazon, даже предлагают интерфейс к веб-службам, позволяющий собирать данные с их сайта. Прекрасными отправными точками для сбора обучающих данных в Интернете могут служить каркасы построения роботов с открытым исходным кодом, например Nutch и Bixo. Но, ступая на этот путь, обращайте внимание на примечание о правах интеллектуальной собственности и условия обслуживания, опубликованные на сайте. Проверьте, что собираемые данные разрешено использовать для намеченной вами цели. Уважайте посещаемые сайты – не позволяйте роботу считывать страницы со слишком большой скоростью и ведите себя так, будто вы сами оплачиваете трафик выкачивания данных. Забирайте только то, что вам нужно, и ничего лишнего и не нагружайте сайт так, что его владельцам придется нести дополнительные расходы, а пользователям – наблюдать отказ от обслуживания. Будьте добропорядочным гражданином Сети; если сомневаетесь, обратитесь к владельцам сайта. Возможно, существуют способы получить данные, не раскрываемые широкой публике.

Если все сказанное выше не подходит и вы оказываетесь перед необходимостью аннотировать набор материалов вручную, не отчаивайтесь. Помимо привлечения к ручной пометке набора твитов своих друзей, родственников, знакомых игроков в бридж и случайных прохожих, вы можете прибегнуть к услугам Механического турка[5]

[5] Механическим турком (Mechanical Turk) назывался шахматный автомат, сконструированный в 18-м веке Вольфгангом фон Кемпеленом, внутри которого, как потом выяснилось, прятался человек. Так и веб-служба Amazon предназначена, чтобы помогать машинам справляться с несвойственными им задачами. – *Прим. перев.*

Amazon. Механический турок – это механизм, с помощью которого предприимчивый разработчик программ машинного обучения, может попросить людей со всего мира оказать помощь в решении задачи, требующей человеческого интеллекта (в терминологии Механического турка – HIT, или *human intelligence task*) за небольшую плату. На сайте Механического турка имеется достаточно информации о настройке и выполнении таких задач.

Использование экспертных оценок в качестве обучающих данных

Есть много способов организовать обратную связь от человека при оценке или обучении машинных алгоритмов. Экспертные оценки часто используются для установления релевантности документов в контексте информационного поиска, когда человек определяет, релевантны ли результаты запросу, и таким образом оценивает качество алгоритма поиска. Есть немало глубоких исследований, посвященных вопросам применения экспертных оценок; некоторые из них приведены в разделе «Ресурсы» в конце этой главы.

Но следует иметь в виду, что человек далек от совершенства. Собираясь прибегнуть к крупномасштабному сбору экспертных оценок, убедитесь, что потенциальные эксперты понимают смысл категорий или меток, способны высказывать непротиворечивые суждения и не изменяют своего мнения со временем. Проверить компетентность эксперта и ясность своей схемы классификации можно, попросив его вынести оценку документам, которые уже были классифицированы другими людьми или машиной. Для проверки непротиворечивости нужно время от времени предъявлять эксперту одни и те же документы для оценки.

По большей части алгоритмы классификации принимают дополнительные параметры, влияющие на вычисления, выполняемые в ходе обучения или классификации. У наивного байесовского классификатора параметров немного, тогда как машины опорных векторов настраиваются в широких пределах. Часто обучение классификатора требует нескольких итераций: обучение с начальными значениями параметров, оценка и небольшая корректировка значений для достижения наилучшего качества классификации.

7.2.4. Оценка качества классификатора

Для оценки обученного классификатора ему предлагается классифицировать уже помеченные документы, после чего результаты сравниваются. Качество классификатора измеряется в терминах его

способности порождать ту же метку, что была присвоена данным ранее. Процентная доля испытаний, в которых класс был проставлен правильно, называется *верностью* (accuracy) классификатора. Она дает общее представление о качестве, но необходимо пойти дальше и определить причины ошибок.

В качестве более детальных показателей работы классификаторов используются варианты точности и полноты, с которыми мы познакомились в главе 3. Эти показатели основаны на типах обнаруженных ошибок.

При испытании двоичного классификатора возможны четыре исхода, показанные в табл. 7.1. Два из них правильны, два – нет. Если и метка, и классификатор говорят, что утверждение истинно, или, наоборот, ложно, то исход правилен. Такие исходы называются истинно положительным и истинно отрицательным соответственно. Если же между классификатором и меткой имеется расхождение, то исход неправилен. Исход называет ложноположительным, если классификатор утверждает, что объект принадлежит классу, а на самом деле это не так. Исход называет ложноотрицательным, если классификатор утверждает, что объект не принадлежит классу, а на самом деле принадлежит. В статистике это называют ошибками *первого* и *второго рода* соответственно.

Таблица 7.1. Классификация исходов

	Попал в класс	**Не попал в класс**
Принадлежит классу	Истинно положительный	Ложноположительный (ошибка первого рода)
Не принадлежит классу	Ложноотрицательный (ошибка второго рода)	Истинно отрицательный

В контексте классификации точность вычисляется как отношение числа истинно положительных исходов к общему числу истинно положительных и ложноположительных исходов. А полнота – как отношение числа истинно положительных исходов к общему числу истинно положительных и ложноотрицательных исходов. Третий показатель – *специфичность*, или *коэффициент ложноотрицательных результатов* – равен отношению числа ложноотрицательных исходов к общему числу истинно отрицательных и ложноположительных исходов.

Одни приложения более чувствительно к ошибкам первого рода, другие – к ошибкам второго рода. При распознавании спама ложно-

положительные исходы обходятся дорого, потому что из-за них пользователь не получает нормальное сообщение; ложноотрицательные более приемлемы, т. к. обнаружение спама в почтовом ящике – событие хоть и неприятное, но легко излечиваемое нажатием клавиши **Del**. В зависимости от требований приложения предпочтение может отдаваться общей верности, точности, полноте или специфичности.

В случае многоклассовых классификаторов принято вычислять все эти показатели для каждого класса, который может назначить классификатор. Затем они агрегируются в единую характеристику классификатора – среднюю верность, точность или полноту.

Часто помимо количества истинно или ложноположительных исходов полезно понимать взаимодействие между двумя классами. Представление результатов в виде *матрицы неточностей* (confusion matrix) описывает природу ошибок, показывая, как документы с различными метками соотносились с классами. Если была допущена ошибка, в матрице приведена категория, к которой ошибочно был отнесен документ. Иногда большая часть ошибок связана с устойчивым перепутыванием двух конкретных классов. Это может свидетельствовать о проблемах в обучающих данных или стратегии отбора признаков.

Для вычисления этих показателей необходимо зарезервировать часть помеченных данных и не предъявлять их процессе обучения. Если имеется 200 новостей в категории «Футбол», то стоит 180 включить в состав обучающих данных, а 20 – придержать, чтобы затем удостовериться, что классификатор верно распознает документы на футбольную тему. Никогда не следует предъявлять тестовые данные на стадии обучения, иначе тесты будут давать обманчиво правильные результаты. Если вы увидите, что результаты слишком хороши, чтобы быть правдой, проверьте, не обучался ли классификатор на тестовых данных.

Существует несколько способов разбить данные на обучающие и тестовые. Начать вполне можно со случайной выборки. Если документы как-то датированы, например, имеют дату публикации, то, возможно, имеет смысл отсортировать их по дате и оставить для тестирования самые свежие документы. Тем самым вы сможете проверить, что вновь поступающие документы правильно классифицируются на основе признаков, найденных в старых документах.

Иногда полезно не ограничиваться разбиением всего набора данных только на обучающие и тестовые. Наш набор из 200 новостей можно было бы разделить на 10 групп по 20 документов в каждой.

А затем обучить несколько классификаторов на разных комбинациях групп. Например, один классификатор обучается на группах с 1 по 9 и тестируется на группе 10, второй обучается на группах со 2 по 10 и тестируется на группе 1, третий обучается на группах 1 и с 3 по 10, а тестируется на группе 2 и т. д. Верности всех тестов усредняются, и результат считается оценкой качества классификатора. Такой подход называется *k-кратной перекрестной проверкой*, где k – количество групп; он часто применяется в статистических методах классификации.

Есть и другие методики оценки качества классификатора. Метрика *площадь под кривой* (area under curve, AUC) полезна, когда набор обучающих документов не сбалансирован по категориям. *Логарифмическое отношение правдоподобия* (log-likelihood ratio) применяется при оценке статистических моделей для сравнения результатов нескольких прогонов обучения.

7.2.5. Внедрение классификатора в эксплуатацию

После того обученный классификатор начинает давать результат приемлемого качества, остается еще решить следующие вопросы.

1. Развертывание классификатора в производственной системе как составной части более крупного приложения.
2. Обновление эксплуатируемого классификатора.
3. Оценка верности модели классификации с течением времени.

Любой из рассматриваемых в этой главе классификаторов можно развернуть в качестве компонента объемлющей службы. Как правило, классификатор развертывается в виде постоянно работающего процесса. На этапе его инициализации модель загружается в память, и служба получает запросы на классификацию – по отдельности или в виде пачки документов. Именно так работает классификатор OpenNLP MaxEnt. Если модель велика и не помещается в память целиком, то часть ее должна храниться на диске. В Apache Lucene такая гибридная схема применяется для хранения индексов документов, поэтому она пригодна и для поддержки больших моделей. Байесовский классификатор в Mahout поддерживает несколько механизмов хранения данных – как в памяти, так и в распределенной базе данных HBase. Кроме того, он предоставляет API расширения, позволяющий реализовать хранилище данных, отвечающее вашей модели развертывания.

Уже после развертывания классификатора должна быть возможность обновлять используемую им модель по мере появления новой информации. Иногда удается обновлять модель оперативно, а иногда приходится строить новую модель автономно, а затем помещать ее на место старой. Классификатор, поддерживающий оперативное обновление, иногда можно расширять динамически, подкладывая ему новые словари. Структура индексов Lucene позволяет без труда добавлять или заменять документы, продолжая при этом обслуживать запросы. В других случаях, например, когда модель хранится в памяти, приложение, работающее с классификатором, следует проектировать так, чтобы вторую модель можно было загружать в память, пока активна первая. По завершении загрузки новая модель подменяет старую, а старая удаляется из памяти.

Оценка качества классификатора с течением времени подразумевает сбор дополнительных данных. Например, можно сохранять часть полученных классификатором за время эксплуатации запросов и вручную проставлять для них категорию. Мониторинг случаев неправильной работы классификатора – вещь полезная, но важно также сохранять достаточно большую выборку данных для оценки, а не только ошибки. Обращайте внимание на новые предметные области, термы или темы обсуждений. Но каковы бы ни были данные, процесс оценки «боевого» классификатора не отличается от оценки на этапе разработки. По-прежнему нужно иметь помеченные вручную тестовые документы и сравнивать метки, проставленные человеком и классификатором.

Часто бывает необходимо изменить схему классификации с учетом новых данных. Проектируя приложение, думайте, как это может повлиять на него. Если вы храните классифицированные документы, то присвоенные им категории могут устареть после изменении схемы. Возможно, придется заново классифицировать документы по новой схеме и для этого потребуется доступ к исходным материалам. Альтернатива – установить соответствие между старыми и новыми категориями, но если категории объединяются и разбиваются, то такое решение может оказаться непригодным.

Рассмотрев принципиальные основы обучения и развертывания, мы можем заняться реальным делом – обучить и протестировать некоторые алгоритмы категоризации. Далее мы изучим три алгоритма классификации и категоризации и разработаем систему рекомендования меток. Разбирая примеры, мы будем обращать внимание на важные аспекты подготовки данных, обучения, тестирования и развертывания.

7.3. Построение классификаторов документов с помощью Apache Lucene

Некоторые алгоритмы классификации называются *пространственными методами*. В них содержимое документа представляется в виде *вектора признаков* – точки в векторном пространстве (см. описание векторной модели в главе 3). А категория документа определяется путем измерения расстояния или угла между вектором термов подлежащего классификации документа и другими векторами, представляющими документы или категории. В этом разделе мы рассмотрим два пространственных алгоритма классификации: метод k-ближайших соседей и TF-IDF. В том и другом классифицируемый документ рассматривается как запрос, и в индексе Lucene выполняется поиск удовлетворяющих этому запросу документов. Для определения категории документа-запроса используются категории найденных документов. В алгоритме k-ближайших соседей просматривается индекс классифицированных документов, а в алгоритме TF-IDF – индекс, в котором каждый документ представляет одну из присваиваемых категорий. У обоих алгоритмов есть преимущества: простота реализации и производительность.

Векторная модель – это основа системы Lucene, оптимизированной так, что вычисления расстояний, необходимые в обоих алгоритмах, производятся быстро, что создает отличный фундамент для возведения нужной функциональности.

В этом разделе мы построим классификаторы документов, в которых Apache Lucene и алгоритмы k-ближайших соседей и TF-IDF применяются для отнесения документов к предметным областям. Мы обучим эти классификаторы на бесплатном тестовом корпусе документов и посмотрим, как оценить качество выдаваемых ими результатов. Поскольку это первый пример, то мы постараемся не усложнять дело, однако все введенные концепции найдут применение и в последующих примерах. Обратите также внимание, что в изложении примера мы точно следуем процессу классификации, описанному в разделе 7.2.

7.3.1. Классификация текстов с помощью Lucene

Lucene демонстрирует высочайшую эффективность при вычислении расстояний. Документы, похожие на предъявленный, возвращаются

через доли секунды, даже если в индексе миллионы документов. В качестве оценки Lucene дает число, обратное расстоянию между документами: чем выше оценка соответствия, тем ближе документы в векторном пространстве. В обоих алгоритмах для назначения категории используются документы, ближайшие к документу-запросу.

В алгоритме k-ближайших соседей (k-NN) документу назначается категория, исходя из категорий его соседей в векторном пространстве. Число *k* в названии алгоритма – один из его параметров: количество соседей, анализируемых на предмет определения наиболее подходящей категории. Если, к примеру, *k* равно 10, то рассматриваются 10 ближайших соседей документа.

В алгоритме TF-IDF создается по одному документу для каждой потенциальной категории. В данном разделе каждый такой документ-категория представляет собой простую конкатенацию всех документов, принадлежащих данной категории. Альтернативный подход – выбрать репрезентативные документы вручную. Алгоритм называется *TF-IDF*, потому что в качестве основы для принятия решений о классификации используется частота терма и обратная частота документа. Относительная важность терма зависит от того, в скольких категориях он встречается. Это же определяет отбор поисковых термов и вычисление расстояния между документом-запросом и категориями в индексе. Различия между алгоритмами k-ближайших соседей и TF-IDF показаны на рис. 7.3.

Реализации обоих алгоритмов содержат много общего кода. В той и другой строится индекс Lucene по обучающим данным. С точки зрения Lucene API, это сводится к созданию объекта `IndexWriter`, его настройки для анализа текста и создания объектов `Document`, подлежащих индексированию. Содержимое объектов `Document` зависит от алгоритма классификации. Как минимум, в любом документе должны быть поля для категории и содержимого. В первом хранятся метки категорий, а во втором – сам текст документа. Код чтения и разбора обучающих данных, добавления документов в индекс, классификации документов и оценки качества алгоритма в обеих реализациях одинаков.

Важная сторона этого процесса – преобразование классифицируемого документа в запрос к Lucene. Простое решение – включить в запрос все уникальные термы, встречающиеся в документе. Если документ короткий, то этого может быть достаточно, но для длинных документов возникают проблемы. Многие слова в таких документах бесполезны для определения категории. Удаление стоп-слов могло

бы сократить размер запросов, основанных на документе, но еще полезнее принимать во внимание содержимое индекса, в котором производится поиск. Не имеет смысла искать терм, которого нет в индексе, а поиск терма, встречающегося во всех документах в индексе, не даст ничего, но заметно увеличит время поиска.

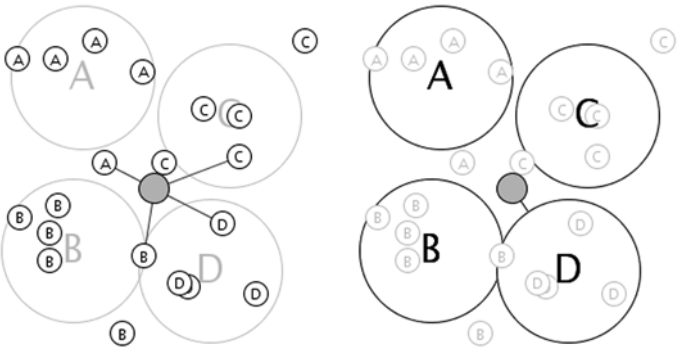

Рис. 7.3. Сравнение алгоритмов классификации по k-ближайшим соседям (k-NN) и по TF-IDF. Слева классифицируемый документ (серый кружок) соединен с 5 ближайшими соседями (k = 5). Два соседа принадлежат категории C и по одному – категориям A, B и D. В результате документ будет отнесен к категории C. Большие окружности на рисунке справа представляют документы-категории, используемые в алгоритме TF-IDF. Каждый такой документ образован конкатенацией исходных документов, принадлежащих соответствующей категории. Результат алгоритма TF-IDF показывает, что документ классифицирован по ближайшей категории, каковой является D, а не C

Для определения слов, наиболее подходящих для классификации, можно использовать такие метрики, как частота терма (TF) и частота документа (DF), которые уже вычислены в ходе построения индекса. Обратная частота документа используется, чтобы отфильтровать малозначимые слова. В результате получается список поисковых термов, отобранных из подлежащего классификации документа с учетом их относительной важности в индексе. И тогда можно не тратить время на поиск терма, от которого вообще не зависит, какой категории принадлежит документ.

По счастью, разработчики Lucene максимально упростили процедуру отбора для такого рода запросов. В Lucene имеется тип запросов `MoreLikeThisQuery`, который производит по индексу отбор термов из документов-запросов. Для рассматриваемых в этом разделе клас-

сификаторов мы воспользуемся запросами типа `MoreLikeThisQuery` для порождения объектов Lucene `Query` по входным данным.

В этом разделе мы обсудим реализацию категоризации, в которой Lucene API применяется для лексического анализа документа, генерации запроса типа `MoreLikeThis` и выполнения этого запроса к индексу Lucene с целью получения категорий входного документа. Индекс Lucene строится по отдельным обучающим документам в случае алгоритма k-ближайших соседей или документам-категориям в случае алгоритма TF-IDF. Поскольку код обоих алгоритмов во многом совпадает, мы поместим их в один пакет в виде вариантов единой реализации и назовем классификатором `MoreLikeThis`. В исходном коде к книге этот классификатор находится в пакете `com.tamingtext.classifier.mlt`.

7.3.2. Подготовка обучающих данных для классификатора MoreLikeThis

Мы обучим классификатор `MoreLikeThis` назначать тематические рубрики материалам из корпуса текстов 20 Newsgroups. В него входят сообщения, отправленные в 20 групп новостей в Интернете. Корпус разбит на обучающий и тестовый набор. Группы новостей охватывают различные тематики, причем некоторые из них, например `talk.politics.mideast` и `rec.autos`, однозначно различны, тогда как другие, например `comp.sys.ibm.pc.hardware` и `com.sys.mac.hardware`, потенциально имеют много общего. Обучающий набор содержит приблизительно 600 статей для каждой группы новостей, а тестовый – около 400 статей. Таким образом, мы имеем хороший набор, поскольку количество обучающих и тестовых примеров в каждой категории примерно одинаково. Из названия архивного файла видно, что разбиение данных на обучающие и тестовые произведено по дате, так что обучающие данные хронологически предшествуют тестовым.

Корпус 20 Newsgroups можно скачать по адресу http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz. Скачав и распаковав архив, вы получите два каталога: 20news-bydate-train и 20news-bydate-test. Каждый из них содержит по одному каталогу для каждой группы новостей. В каждом таком подкаталоге находится по одному файлу для каждого сообщения из группы новостей. Для обучения и тестирования классификатора нужно преобразовать эти файлы в подходящий формат. Для простоты мы будем во всех примерах из этой главы пользоваться форматом, принятым в Mahout.

Входные данные для обучения и тестирования мы создадим утилитой `PrepareTwentyNewsgroups` из Mahout. Выполните следующие команды:

```
$MAHOUT_HOME/bin/mahout \
 org.apache.mahout.classifier.bayes.PrepareTwentyNewsgroups \
 -p 20news-bydate-train \
 -o 20news-training-data \
 -a org.apache.lucene.analysis.WhitespaceAnalyzer \
 -c UTF-8

$MAHOUT_HOME/bin/mahout \
 org.apache.mahout.classifier.bayes.PrepareTwentyNewsgroups \
 -p 20news-bydate-test \
 -o 20news-test-data \
 -a org.apache.lucene.analysis.WhitespaceAnalyzer \
 -c UTF-8
```

> **Примечание.** В примерах из этой главы постоянно встречаются ссылки на переменные окружения $MAHOUT_HOME и $TT_HOME. MAHOUT_HOME должна указывать на начальный каталог развернутого дистрибутива Mahout 0.6, в котором имеется подкаталог *bin*, где находится скрипт *mahout*. Переменная TT_HOME должна указывать на начальный каталог, внутри которого находится исходный код, прилагаемый к книге. Его подкаталог *bin* содержит скрипт *tt*. Оба эти скрипта обертывают Java-программу для настройки окружения, в котором должны выполняться Java-классы, входящие в соответственные дистрибутивы. Подобные переменные окружения позволяют указать собственный рабочий каталог, где будут храниться сгенерированные данные, отделив его от кода Mahout и прилагаемого к книге.

Ко времени выхода книги из печати будет выпущена версия Mahout 0.7[6]. В ней байесовские классификаторы подверглись значительным изменениям, поэтому для выполнения примеров из этой книги пользуйтесь версией Mahout 0.6. Следите за сайтом автора книги, на котором будут выкладываться обновления кода, соответствующие версии Mahout 0.7 и более поздним.

Как видно, для простого лексического анализа входных данных мы используем класс `WhitespaceAnalyzer`. Позже – на этапах обучения и тестирования – с помощью входящего в Lucene анализатора `EnglishAnalyzer` будут выполнены стемминг и удаление стоп-слов, поэтому сейчас нам нужно просто разбить текст на лексемы по пробелам. Для других классификаторов, например байесовского классификатора из Mahout, стемминг и удаление стоп-слов лучше производить на этапе подготовки данных.

[6] Сейчас уже вышла версия 0.9. – *Прим. перев.*

Команды подготовки создают наборы файлов, содержащих обучающие и тестовые данные. Для каждой категории имеется один файл, состоящий из строк, причем каждая строка состоит из двух столбцов, разделенных знаком табуляции. В первом столбце находится имя группы новостей, а во втором – содержимое обучающего документа, из которого удалены все знаки табуляции и новой строки. Ниже приведен фрагмент одного файла из обучающего набора. В каждой строке мы видим заголовки сообщения и далее текст сообщения:

```
alt.atheism ... Alt.Atheism FAQ: Atheist Resources Summary: Books,
...
alt.atheism ... Re: There must be a creator! (Maybe) ...
alt.atheism ... Re: Americans and Evolution ...
...
comp.graphics ... CALL FOR PRESENTATIONS ...
comp.graphics ... Re: Text Recognition ...
comp.graphics ... Re: 16 million vs 65 ...
...
comp.os.ms-windows.misc ... CALL FOR PRESENTATIONS ...
comp.os.ms-windows.misc ... Re:color or Monochrome? ...
comp.os.ms-windows.misc ... Re: document of .RTF Organization: ...
```

Подготовив обучающий и тестовый наборы, мы можем приступить к обучению классификатора `MoreLikeThis`.

7.3.3. Обучение классификатора MoreLikeThis

Обучение классификатора `MoreLikeThis` запускается командой, ссылающейся на программы, прилагаемые к данной книге. Следующая команда генерирует модель с помощью алгоритма k-NN, рассмотренного в разделе 7.3.1:

```
$TT_HOME/bin/tt trainMlt \
-i 20news-training-data \
-o knn-index \
-ng 1 \
-type knn
```

Эта команда строит индекс Lucene по обучающим данным, подготовленным в предыдущем разделе. Построение индекса для алгоритма TF-IDF ничуть не сложнее, нужно только в аргументе `-type` указать значение `tfidf`.

Рассмотрим исходный код этих программ. В листинге 7.1 показан код создания индекса и настройки конвейера обработки текста для индексирования обучающих данных.

Листинг 7.1. Создание индекса Lucene

```
Directory directory                                    <-- ❶ Создаем каталог для индекса
 = FSDirectory.open(new File(pathname));
Analyzer analyzer                                      <-- ❷ Настраиваем анализатор
 = new EnglishAnalyzer(Version.LUCENE_36);

if (nGramSize > 1) {                                   <-- ❸ Настраиваем фильтр черепиц
 ShingleAnalyzerWrapper sw
  = new ShingleAnalyzerWrapper(analyzer,
     nGramSize,     // минимальный размер черепицы[7]
     nGramSize,     // максимальный размер черепицы
     "-",       // разделитель лексем
     true,      // выводить униграммы
     true);     // выводить униграммы, если нет черепиц
  analyzer = sw;
}

IndexWriterConfig config                               <-- ❹ Создаем IndexWriter
 = new IndexWriterConfig(Version.LUCENE_36, analyzer);
config.setOpenMode(OpenMode.CREATE);
IndexWriter writer = new IndexWriter(directory, config);
```

В точке ❶ мы создаем объект Lucene `Directory`, представляющий место на диске, в котором будет храниться индекс. Этот объект создается методом `FSDirectory.open()`, принимающим полный путь к каталогу. В точке ❷ создаем объект анализатора, который будет генерировать лексемы из входного текста.

Входящий в состав Lucene анализатор `EnglishAnalyzer` – неплохая отправная точка, которая обеспечивает разумный стемминг и удаление стоп-слов. В Lucene и во внешних библиотеках есть и другие анализаторы. Поэкспериментируйте с различными вариантами, чтобы получить лексемы, наиболее подходящие для вашего приложения. Например, возможно, вам нужен анализатор для другого языка или фильтр, который удаляет алфавитно-цифровые символы либо нормализует составные слова типа *Wi-Fi*. В стандартной конфигурации,

[7] Авторы Lucene так объясняют смысл слова shingle: «shingle – это просто n-грамма из соседних слов в противоположность *n*-грамме из символов. Мы выбрали это слово, чтобы отличить одно от другого в именах фильтров – ну и потому, что они, как и черепицы на кровле, накладываются друг на друга». Было бы очень обидно потерять такой красочный образ ради стремления к сухому академизму. И уж тем более негоже засорять русский язык «шинглами». – *Прим. перев.*

входящей в дистрибутив Solr, имеются примеры различных анализаторов и комбинаций (см. главу 3).

В этом примере мы дополняем результат, формируемый анализатором `EnglishAnalyzer`, *n*-граммами. Код в точке ❸ показывает, как использовать имеющийся в Lucene класс `ShingleAnalyzerWrapper` для порождения не только отдельных слов, но и *n*-грамм, если параметр `nGramSize` больше 1.

> **ShingleAnalyzer и *n*-граммы из слов.** Ранее в главе 4 мы сталкивались с *n*-граммами из символов. *N*-граммы, порождаемые анализатором ShingleAnalyzer, состоят из слов. Если обрабатывается текст *now is the time* и параметр `nGramSize` равен 2, то ShingleAnalyzer породит такие лексемы: *now-is*, *is-the* и *the-time* (в предположении, что стоп-слова не удаляются). Все такие лексемы будут рассматриваться как признаки, полезные для установления различий между категориями текста. В данном случае `ShingleFilter` применяется к результатам `EnglishAnalyzer`. Если из текста *now is the time* удалить стоп-слова *is* и *the*, то будут созданы *n*-граммы *now-_* и *_-time*. Знак подчеркивания отмечает место удаленного стоп-слова, чтобы предотвратить образование пар слов, которых в исходном тексте не было.

В команде для запуска обучения, показанной в начале этого раздела, использовалось подразумеваемое по умолчанию значение 1 параметра `nGramSize`. Чтобы его изменить, нужно задать желаемое значение после флага `-ng`, например, `-ng 2`.

Создав `EnglishAnalyzer` и, возможно, обернув его классом `ShingleAnalyzerWrapper`, мы готовы к построению индекса Lucene. В точке ❹ создаются объекты `IndexConfig` и `IndexWriter`, необходимые для индексирования обучающих данных.

Далее мы читаем обучающие данные из файлов и преобразуем их в объекты `Document`, которые затем добавляются в индекс. Начинать следует с создания объектов `Field`, в которых будет храниться информация, прочитанная из документа. В данном случае имеется три поля: уникальный идентификатор документа, категория документа и его содержимое, т. е. лексемы, порожденные анализатором, которые будут играть роль признаков для обучения. Ниже показано, как эти поля создаются.

Листинг 7.2. Инициализация полей документа

```
Field id = new Field("id", "", Field.Store.YES,
Field.Index.NOT_ANALYZED, Field.TermVector.NO);
Field categoryField = new Field("category", "", Field.Store.YES,
```

```
Field.Index.NOT_ANALYZED, Field.TermVector.NO);
Field contentField = new Field("content", "", Field.Store.NO,
Field.Index.ANALYZED, Field.TermVector.WITH_POSITIONS_OFFSETS);
```

Нам не нужно ни анализировать поля id и category, ни создавать для них векторы термов. Они просто хранятся в индексе для последующего использования. Поле content обрабатывается анализатором, созданным ранее в листинге 7.1, и создаваемые для него векторы термов включают позиции и смещения, чтобы запомнить порядок следования термов в исходном документе.

Программа обучения в цикле обходит все документы из входного файла и индексирует их, применяя различные методы в зависимости от алгоритма классификации. В листинге ниже показано, как индексируются документы в случае алгоритма k-ближайших соседей, когда каждый обучающий пример представлен одним документом в индексе.

Листинг 7.3. Индексирование обучающих документов для алгоритма классификации по k-ближайшим соседям

```
while ((line = in.readLine()) != null) {
 String[] parts = line.split("\t"); <-- ❶ Получаем компоненты документа
 if (parts.length != 2) continue;
 category = parts[0];
 categories.add(category);

 Document d = new Document();     <-- ❷ Строим документ
 id.setValue(category + "-" + lineCount++);
 categoryField.setValue(category);
 contentField.setValue(parts[1]);
 d.add(id);
 d.add(categoryField);
 d.add(contentField);

 writer.addDocument(d);  <-- ❸ Индексируем документ
}
```

В реализации алгоритма k-NN программа читает каждую строку, представляющую один исходный документ, и выделяет из нее категорию в точке ❶, а затем в точке ❷ создает документ, который добавляется в индекс Lucene в точке ❸. При таком подходе размер индекса пропорционален количеству документов в обучающем наборе.

В следующем листинге показано, как обучающие данные индексируются в случае алгоритма TF-IDF.

Листинг 7.4. Индексирование обучающих документов для алгоритма классификации TF-IDF

```
StringBuilder content = new StringBuilder();
String category = null;
while ((line = in.readLine()) != null) {
 String[] parts = line.split("\t");      <-- ❶ Получаем компоненты документа
 if (parts.length != 2) continue;
 category = parts[0];
 categories.add(category);
 content.append(parts[1]).append(" ");
 lineCount++;
}

in.close();

Document d = new Document();             <-- ❷ Строим документ
id.setValue(category + "-" + lineCount);
categoryField.setValue(category);
contentField.setValue(content.toString());
d.add(id);
d.add(categoryField);
d.add(contentField);

writer.addDocument(d);                   <-- ❸ Индексируем документ
```

В этом случае программа читает строку, представляющую один исходный документ, в точке ❶ и конкатенирует все содержимое в одну строку. Прочитав все документы одной категории, программа в точке ❷ создает документ Lucene и в точке ❸ добавляет его в индекс. Если для каждой категории размер текста велик, то этот алгоритм будет потреблять больше памяти, чем k-NN, потому что буфер, содержащий текст всех документов, принадлежащих категории, сначала строится в памяти, а только потом передается Lucene.

Построив индексы по обучающим данным, мы можем реализовать и протестировать алгоритмы классификации.

7.3.4. Классификация документов с помощью классификатора MoreLikeThis

Первое, что нужно сделать для классификации документов, – открыть индекс Lucene и настроить анализаторы для разбора подлежащего классификации текста. Важно, чтобы были созданы точно такие же классификаторы, как на этапе обучения, и чтобы они были сконфигурированы точно таким же образом, тогда поисковые термы

будут образованы так же, как термы, хранящиеся в индексе. Это означает, что необходимо использовать тот же самый список стоп-слов, тот же алгоритм стемминга и те же настройки *n*-грамм. Подготовив индекс и анализаторы, мы можем создать и сконфигурировать экземпляр класса `MoreLikeThis`. Ниже показано, как это делается.

Листинг 7.5. Настройка классификатора MoreLikeThis

```
Directory directory = FSDirectory.open(new File(modelPath));
                                                  Открываем индекс ❶
IndexReader indexReader = IndexReader.open(directory);
IndexSearcher indexSearcher = new IndexSearcher(indexReader);

Analyzer analyzer     ❷ Настраиваем анализатор
 = new EnglishAnalyzer(Version.LUCENE_36);

if (nGramSize > 1) {        ❸ Настраиваем n-граммы
 analyzer = new ShingleAnalyzerWrapper(analyzer, nGramSize, nGramSize);
}

MoreLikeThis moreLikeThis = new MoreLikeThis(indexReader);
moreLikeThis.setAnalyzer(analyzer);                          ❹
moreLikeThis.setFieldNames(new String[] {          Создаем объект
 "content"                                          MoreLikeThis
});
```

В точке ❶ мы создаем объект `Directory` и открываем `IndexReader` и `IndexSearcher`. Объект `IndexReader` будет извлекать термы для построения запроса, а затем, после выполнения запросов, извлекать содержимое документа. В точках ❷ и ❸ мы создаем анализатор `EnglishAnalyzer` и факультативно обертываем его объектом `ShingleAnalyzer`, чтобы порождать *n*-граммы, если это заказано. В точке ❹ мы создаем объект `MoreLikeThis`, передавая ему экземпляр `IndexReader`, затем устанавливаем в нем анализатор и настраиваем на использование поля content для выбора термов, участвующих в запросе. При построении запросов объект `MoreLikeThis` будет сравнивать частоты термов в документе-запросе и в индексе и решать, какие термы использовать. Термы, частота встречаемости которых в документе или в индексе ниже пороговой, а также те, которые встречаются индексе слишком часто, не рассматриваются как кандидаты на включение в запрос, т. к. мало что дают для его способности различать категории.

Создав объекты, необходимые для построения и выполнения запросов к индексу, мы можем приступить к поиску и назначению ка-

тегорий документам. В листинге ниже приведен метод извлечения документов, показывающий, как производится классификация.

Листинг 7.6. Классификация текста с помощью классификатора MoreLikeThis

```
Reader reader = new FileReader(inputPath); <-- ❶ Создаем запрос
Query query = moreLikeThis.like(reader);

TopDocs results
 = indexSearcher.search(query, maxResults); <-- ❷ Выполняем поиск

HashMap<String, CategoryHits> categoryHash
 = new HashMap<String, CategoryHits>();

for (ScoreDoc sd: results.scoreDocs) { <-- ❸ Собираем результаты
 Document d = indexReader.document(sd.doc);
 Fieldable f = d.getFieldable(categoryFieldName);
 String cat = f.stringValue();
 CategoryHits ch = categoryHash.get(cat);
 if (ch == null) {
   ch = new CategoryHits();
   ch.setLabel(cat);
   categoryHash.put(cat, ch);
 }
 ch.incrementScore(sd.score);
}

SortedSet<CategoryHits> sortedCats <-- ❹ Ранжируем категории
 = new TreeSet<CategoryHits>(CategoryHits.byScoreComparator());
sortedCats.addAll(categoryHash.values());

for (CategoryHits c: sortedCats) { <-- ❺ Отображаем категории
 System.out.println(
 c.getLabel() + "t" + c.getScore());
}
```

В точке ❶ мы создаем объект `Reader`, который будет читать содержимое классифицируемого документа. Этот объект передается методу `MoreLikeThis.like()`, который генерирует запрос к Lucene, основанный на ключевых термах в документе. Имея запрос, мы производим поиск в точке ❷ и получаем стандартный ответ Lucene – объект `TopDocs`, содержащий подходящие документы. В точке ❸ мы в цикле перебираем возвращенные документы, извлекаем категорию и сохраняем ее название и вес в объекте `CategoryHits`. Собрав все результаты, мы ранжируем их в точке ❹, где множество объектов `CategoryHits` сортируется по весу категории, а затем распечатыва-

ем в точке ❺. Категория с наибольшим весом назначается документу. Этот алгоритм ранжирования по весам, несмотря на примитивность, дает приемлемые результаты. Предлагаем вам исследовать другие подходы к ранжированию категорий и, сравнив результаты, решить, какой лучше всего отвечает вашим целям.

Выбор категории производится одинаково вне зависимости от способа построения индекса: с помощью алгоритма k-NN или TF-IDF. В случае k-NN в наборе результатов для каждой категории может оказаться один или несколько документов; в случае TF-IDF – ровно один. Окончательная оценка документа основана на оценках одного или нескольких документов, удовлетворяющих запросу.

Интеграция классификатора `MoreLikeThis` с производственной системой производится очень просто: нужно лишь один раз за все время жизни приложения выполнить инициализацию, а затем для каждого подлежащего классификации документа генерировать запрос типа `MoreLikeThis`, после чего извлекать и ранжировать возвращенные в результате поиска категории.

Все эти действия объединены в классе `MoreLikeThisCategorizer`, который включен в прилагаемый к книге код. Этот класс можно использовать в качестве отправной точки для развертывания производственного классификатора. В нем код, описанный в примере 7.6, организован немного иначе, но настройку и классификацию он все равно выполняет. В следующем разделе мы воспользуемся этим классом для оценки верности классификатора.

7.3.5. Тестирование классификатора MoreLikeThis

Для тестирования классификатора `MoreLikeThis` выполните следующую команду:

```
$TT_HOME/bin/tt testMlt \\
 -i category-mult-test-data \\
 -m knn-index \\
 -type knn \\
 -contf content -catf category
```

По завершении она напечатает две метрики, характеризующие качество выдаваемых результатов. Первая – количество и процентная доля правильно и неправильно классифицированных тестовых документов. Эта метрика характеризует верность классификатора (см. раздел 7.2.4). Вторая – таблица, представляющая *матрицу не-*

точностей, в которой описаны успешные и неуспешные классификации тестовых документов. Строки матрицы представляют метки, заранее назначенные тестовым документам, а столбцы – категории, присвоенные алгоритмом. Число на пересечении строки и столбца показывает, сколько документов должны были получить категорию, проставленную в данной строке, но получили категорию, проставленную в данном столбце. Правильно классифицированные документы представлены ячейками на главной диагонали матрицы. Ниже показан фрагмент матрицы неточностей, полностью ее можно увидеть, выполнив команду `testMlt`[8]:

```
=======================================================
Сводные данные
-------------------------------------------------------
Правильно классифицированные образцы     : 5381      71.4418%
Неправильно классифицированные образцы   : 2151      28.5582%
Всего классифицировано образцов          : 7532
=======================================================
Матрица неточностей
a   b   c   d   e   f   ... <--Классифицирован как
315 3   4   5   0   20  ... | 393 a   = rec.motorcycles
0   308 0   1   0   2   ... | 390 b   = comp.windows.x
0   0   320 4   1   0   ... | 372 c   = talk.politics.mideast
2   3   13  271 9   0   ... | 361 d   = talk.politics.guns
1   0   10  19  129 0   ... | 246 e   = talk.religion.misc
18  3   2   6   2   293 ... | 394 f   = rec.autos
...
Категория по умолчанию: unknown: 20
```

Матрицы неточностей применяются для оценки произвольных двоичных или многоклассовых классификаторов. В случае двоичного классификатора матрица имеет размерность 2×2, как в табл. 7.1. Если же классификатор обучен на N классах, то матрица неточностей будет иметь размерность $N \times N$.

В примере выше мы показали только результаты для первых шести категорий, присутствующих в корпусе 20 Newsgroups. Каждая строка представляет одну из категорий корпуса, и, глядя на нее, мы видим, сколько документов классификатор отнес к этой категории. Строка `a` соответствует категории `rec.motorcycles`. Из 393 тестовых документов в этой категории правильно классифицированы 315 (число в столбце `a`). В остальных столбцах показано, сколько документов, принадлежащих категории `rec.motorcycles`, классификатор поместил

[8] Для удобства читателя сообщения переведены, на самом деле они печатаются по-английски. – *Прим. перев.*

в другие категории. Так, мы видим, что 20 документов помещены в категорию `rec.autos`. Это и неудивительно, потому что предметные области, которым посвящены эти группы новостей, а, стало быть, и применяемая терминология схожи.

Из того, что на главной диагонали расположены самые большие числа, следует, что данный классификатор присваивает категории по большей части правильно. Но видно также, где он часто ошибается; например, 20 документов о мотоциклах были классифицированы как документы об автомобилях, а 18 документов об автомобилях классификатор счел относящимися к мотоциклам. Имеются также неточности в классификации документов из подгрупп группы talk: лишь 129 из 246 примеров в группе `talk.religion.misc` классифицированы правильно, а 19 отнесено к группе `talk.politics.guns`. В тех случаях, когда документы классифицированы неправильно, матрица неточностей показывает, куда они были отнесены и где, следовательно, обучающие данные неоднозначны.

В листинге 7.7 показано, как тестировался обученный классификатор `MoreLikeThis`. Мы читали файлы с тестовыми данными, классифицировали каждый документ и сравнивали получившийся результат с тем, который был присвоен документу вручную. Для обработки результатов сравнения мы воспользовались классом `ResultAnalyzer` из проекта Apache Mahout, который и напечатал описанные выше метрики.

Листинг 7.7. Оценка результатов классификатора MoreLikeThis

```
final ClassifierResult UNKNOWN = new ClassifierResult("unknown", 1.0);

ResultAnalyzer resultAnalyzer =                    <-- ❶ Создаем ResultAnalyzer
  new ResultAnalyzer(categorizer.getCategories(),
     UNKNOWN.getLabel());

for (File ff: inputFiles) {                        <-- ❷ Читаем тестовые данные
 BufferedReader in =
  new BufferedReader(
    new InputStreamReader(
      new FileInputStream(ff), "UTF-8"));
 while ((line = in.readLine()) != null) {
  String[] parts = line.split("\t");
  if (parts.length != 2) {
    continue;
  }

  CategoryHits[] hits =                            <-- ❸ Классифицируем
```

```
    categorizer.categorize(new StringReader(parts[1]));
  ClassifierResult result = hits.length > 0 ? hits[0] : UNKNOWN;
  resultAnalyzer.addInstance(parts[0], result);
}
 in.close();
}
System.out.println(resultAnalyzer.toString());
```

❹ Обрабатываем результаты

❺ Отображаем результаты

В точке ❶ мы создаем объект `ResultAnalyzer`, передавая конструктору список порожденных классификатором категорий и категорию по умолчанию `UNKNOWN` для документов, которые не удалось классифицировать. В точке ❷ очередная строка тестовых данных читается из файла и разбивается по знаку табуляции, заполняя `parts`. Элемент `parts[0]` будет содержать метку категории, а элемент `parts[1]` – текст документа. Документ классифицируется в точке ❸, где мы получаем ранжированный список категорий для него. Мы берем категорию с максимальным рангом и в точке ❹ передаем ее объекту `resultAnalyzer`. Если классификатор не вернул ни одной категории, то мы относим документ к классу `UNKNOWN`. После того как все тестовые данные обработаны, мы в точке ❺ выводим процентную долю правильных классификаций и матрицу неточностей.

7.3.6. Классификатор MoreLikeThis в производитственной системе

Мы изучили основные элементы классификатора документов на основе Lucene. Были рассмотрены взаимодействия с Lucene API, необходимые для обучения классификатора посредством построения индекса по уже классифицированным документам. Мы также видели, как можно классифицировать новый документ, преобразовав его в запрос к Lucene, и как оценить качество классификатора. Мы обсудили некоторые аспекты применения этих алгоритмов в производственной системе. В разделе 7.4.7 мы поговорим еще об одном сценарии развертывания. А тот, что был рассмотрен здесь, нетрудно модифицировать для «боевого» развертывания классификатора `MoreLikeThis`. API запросов к Lucene обладает высокой гибкостью и позволяет легко интегрировать подобные классификаторы и во многих других контекстах.

API индексирования в Lucene позволяет без труда изменить модель, применяемую для классификации. В случае алгоритма k-NN

для пополнения модели достаточно добавить новые классифицированные документы в индекс. Обучение можно производить инкрементно, ограничением является разве что размер индекса. В случае алгоритма TF-IDF нужно просто заменить существующий документ-категорию – включить обновленное содержимое категории в качестве нового документа, а старый удалить. Возможность таким образом добавлять новые обучающие данные в классификатор называется *оперативным обучением* и часто является желательным свойством алгоритмов классификации. Классификаторы, основанные на алгоритмах автономного обучения, так расширить нельзя, их приходится переучивать с нуля при каждом изменении данных, а это дорого, потому что требует времени и машинных ресурсов.

Познакомившись с процессом построения классификатора на основе Lucene, умеющего производить дистанционную классификацию, мы в разделе 7.4 повторим все заново и обучим наивный байесовский классификатор текстов, входящий в состав Apache Mahout. Помимо знакомства с алгоритмом статистической классификации, мы изучим, как можно использовать в качестве обучающих уже имеющиеся данные, например, собранные веб-роботом.

7.4. Обучение наивного байесовского классификатора в Apache Mahout

В главе 6 мы видели, как использовать Apache Mahout для группировки документов в кластеры на основе схожести тематики. В Mahout имеются также различные алгоритмы классификации, позволяющие назначать категории текстовым документам. Один из них – наивный байесовский классификатор. Этот алгоритм применяется для решения широкого спектра задач и может служить отличным введением в методы статистической классификации. Для отнесения к тому или иному классу такие алгоритмы строят модель, в основе которой лежит вероятность того, что признаки документа принадлежат данному классу.

В этом разделе мы воспользуемся включенной в Mahout реализацией наивного байесовского алгоритма для построения классификатора документов. Ранее в этой главе мы продемонстрировали, как обучить классификатор на корпусе текстов 20 Newsgroups. А сейчас мы создадим собственный корпус из данных, собранных в Интернете,

и с его помощью обучим классификатор. Для этой цели мы возьмем данные, которые собрали в главе, посвященной кластеризации. На их базе мы продемонстрируем итеративное обучение классификатора и представим стратегии реорганизации обучающих данных для повышения верности классификации. И наконец, мы покажем, как классификатор документов интегрируется с Solr, что позволяет автоматически присваивать категории документам уже на этапе индексирования. Начнем с обсуждения теоретических оснований алгоритма наивной байесовской классификации.

7.4.1. Наивная байесовская классификация текста

Наивная байесовская классификация – пример вероятностного алгоритма. Он принимает решения относительно класса входного документа, исходя из вероятностей, вычисленных на этапе обучения. В процессе обучения анализируются связи между словами, встречающимися в обучающих документах, и категориями, а также между категориями и всем обучающим набором. Вычисления на основе теоремы Байеса позволяют по имеющимся фактам рассчитать вероятность того, что набор слов (документ) принадлежит определенному классу.

Слово «наивный» связано с предположением о независимости слов, определяющих принадлежность к классу. Интуитивно мы понимаем, что в текстах на определенную тему слова вовсе не являются независимыми. Слово *рыба* скорее появится в документе со словом *вода*, чем в документе со словом *космос*. Поэтому вероятности, вычисленные наивным байесовским алгоритмом, не совсем правильны. Тем не менее, они полезны в качестве относительных показателей. С их помощью нельзя точно предсказать вероятность принадлежности документа к некоторому классу, но можно высказать предположение, что документ, содержащий слово *рыба*, с большей вероятностью относится к океанографии, чем к космическим полетам, – нужно лишь сравнить вероятности, приписанные слову *рыба* в каждой из этих категорий.

На этапе обучения наивного байесовского классификатора подсчитывается, сколько раз каждое слово появлялось в документе, принадлежащем классу, и эта величина делится на общее число слов в этом классе. Это называется *условной вероятностью* – в данном случае вероятностью появления слова в конкретной категории. Обычно она записывается в виде $P(Слово \mid Категория)$. Допустим, имеется

небольшой обучающий набор всего из трех документов в категории «Геометрия» и что в одном из документов встречается слово *угол*. Тогда вероятность того, что произвольный документ, отнесенный к категории *геометрия*, будет содержать слово *угол*, равна 0,33 или 33%.

Перемножив вероятности отдельных слов, мы получим вероятность принадлежности документа классу. Сама по себе она не очень полезна, но с помощью теоремы Байеса можно из этих вероятностей получить вероятность категории при условии документа, а это и есть основная задача классификации.

Теорема Байеса утверждает, что вероятность категории при условии документа равна вероятности документа при условии категории, умноженной на вероятность категории и поделенной на вероятность документа. Записывается это так:

```
P(Категория | Документ) = P(Документ | Категория) x P(Категория) /
P(Документ)
```

Мы показали, как вычислить вероятность документа при условии категории. Вероятность категории – это количество обучающих документов в этой категории, поделенное на общее количество обучающих документов. Вероятность документа в данной ситуации безразлична, т. к. является просто масштабным коэффициентом. Если положить *P*(*Документ*) = 1, то результаты, полученные по приведенной выше формуле, можно сравнивать для разных категорий. Чтобы определить, какой категории с наибольшей вероятностью принадлежит данный документ, нужно проделать это вычисление для каждой потенциально возможной категории; относительный ранг полученных результатов не зависит от значения *P*(*Документ*), лишь бы оно было больше нуля.

От этого объяснения можно отталкиваться, но это только часть общей картины. Реализация наивного байесовского алгоритма в Mahout содержит многочисленные усовершенствования, учитывающие некоторые уникальные особенности текстовых данных, из-за которых алгоритм перестает работать, в частности, описанную выше проблему зависимых термов. Описание усовершенствований можно найти на вики-сайте Mahout и в статье Rennie и др. «Tackling the Poor Assumptions of naive Bayes Text Classifiers» (Rennie [2003]).

7.4.2. Подготовка обучающих данных

Классификатор не может быть лучше данных, на которых обучался. Объем обучающих данных, способ их организации, отобранные при-

знаки – все это решающим образом сказывается на способности классификатора правильно классифицировать новые документы.

В этом разделе мы опишем, как готовить обучающие данные для байесовского классификатора из проекта Mahout. Мы продемонстрируем процесс извлечения данных из индекса Lucene и процесс бутстрапинга, который порождает обучающий набор с помощью атрибутов имеющихся данных. К концу этого примера вы будете понимать, как общее качество классификатора зависит от метода бутстрапинга.

В главе 6 мы описали, как настроить простое приложение кластеризации на основе Solr. Это приложение импортировало содержимое из различных RSS-лент и сохраняло его в индексе Lucene. Мы воспользуемся данными из этого индекса для построения обучающего набора. Если вы еще не собирали данные с помощью экземпляра Solr `Clustering`, то, следуя инструкциям в разделе 6.3, сделайте это сейчас – запустите Data Import Handler несколько раз на протяжении нескольких дней, чтобы построить достаточно представительный корпус обучающих документов. Имея данные, вы сможете обследовать индекс и решить, что можно использовать для обучения.

После того как в вашем распоряжении окажется заполненный индекс Lucene, нужно будет посмотреть, что там есть, и решить, как использовать его для обучения классификатора. Есть разные способы просмотра содержимого индекса, но, наверное, самый простой – воспользоваться программой Luke. Мы выясним, какие поля документа годятся в качестве источника категорий для схемы классификации. Мы определим набор категорий, а затем извлечем документы и запишем их в формате обучающих данных, понятном Mahout. В процессе обучения байесовского классификатора будут анализироваться слова, которые появляются в документах, отнесенных к определенной категории, и будет создана модель для вычисления наиболее вероятной категории документа, исходя из встречающихся в нем слов.

Последнюю версию Luke можно скачать с сайта http://code.google.com/p/luke/; вам нужен файл lukeall-version.jar, где `version` – номер текущей версии Luke. Затем запустите Luke командой `java -jar lukeall-version.jar`.

После запуска появится диалоговое окно просмотра файловой системы, в котором можно выбрать интересующий вас индекс. Выбрав индекс, нажмите кнопку **OK**, чтобы открыть его (параметры по умолчанию можно оставить).

Просматривая индекс с помощью Luke, вы обнаружите, что многие источники данных снабжают публикуемые документы категориями.

Категории могут быть как весьма общими, например Sports (Спорт), так и более узкими: Baseball (Бейсбол) или даже New York Yankees (Нью-Йорк Янкис). Этими сведениями мы и воспользуемся для организации обучающих данных. Задача состоит в том, чтобы построить список термов, с помощью которых статьи удастся сгруппировать в широкие категории для обучения классификатора. В списке ниже показаны первые 12 категорий, попавших в поле индекса `categoryFacet`, рядом с каждой указано количество документов в этой категории.

```
2081 Nation & World
923 Sports
398 Politics
356 Entertainment
295 sportsNews
158 MLB
128 Baseball
127 NFL
115 Movies
94 Sounders FC Blog
84 Medicine and Health
84 Golf
```

Сразу бросается в глаза, что в категории Nation & World (В стране и мире) больше всего документов – 2081, а дальше число документов быстро убывает, так что в двенадцатой по порядку категории Golf всего 84 статьи. Обратите также внимание на пересечение тематики рубрик Sports, Baseball и MLB (Главная лига бейсбола) и на различные представления одной и той же предметной области: Sports и sportsNews (Спортивные новости). Ваша задача – очистить данные, так чтобы их можно было эффективно использовать для обучения. И этот аспект этапа подготовки обучающих данных оказывает существенное влияние на верность классификации. Для демонстрации начнем с простой стратегии отбора обучающих документов, а затем рассмотрим более полную и сравним результаты.

Из списка найденных в индексе категорий видно, что некоторые полезные термы находятся в начале. Добавим еще несколько интересных категорий, которые помог найти Luke:

```
Nation
Sports
Politics
Entertainment
Movies
Internet
```

```
Music
Television
Arts
Business
Computer
Technology
```

Введите этот список в текстовом редакторе и сохраните в файле `trainingcategories.txt`. Имея список интересующих нас категорий, запустим утилиту `extractTrainingData`, указав этот список и индекс Lucene:

```
$TT_HOME/bin/tt extractTrainingData \
 --dir index \
 --categories training-categories.txt \
 --output category-bayes-data \
 --category-fields categoryFacet,source \
 --text-fields title,description \
 --use-term-vectors
```

Эта команда читает документы из индекса Lucene и ищет подходящие категории в полях categoryFacet и source. Обнаружив какую-нибудь категорию из списка перечисленных в файле training-categories.txt, программа извлекает термы из векторов, хранящихся в поле title и description. Эти термы записываются в файлы в каталоге category-bayes-data, по одному для каждой категории. Это обычный текстовый файл, который можно просмотреть в любом редакторе.

Просматривая созданные файлы, обратите внимание, что одна строка соответствует одному документу в индексе. В каждой строке есть два столбца, разделенных знаком табуляции. В первом столбце находится название категории, во втором – все встречающиеся в документе термы. Байесовский классификатор из проекта Mahout ожидает, что входные поля подвергнуты стеммингу, что вы и видите в обучающих данных. Аргумент `--use-term-vectors` команды `extractTrainingData` означает, что будет выводиться вектор термов, прошедших стемминг.

```
arts 6 a across design feast nut store world a browser can chosen ...
arts choic dealer it master old a a art auction current dealer ...
arts alan career comic dig his lay moor rest unearth up a a ...
business app bank citigroup data i iphon phone say store account ...
business 1 1500 500 cut job more plan tech unit 1 1500 2011 500 ...
business caus glee home new newhom sale up a against analyst ...
computer bug market sale what access address almost ani bug call ...
computer end forget mean web age crisi digit eras existenti face ...
computer mean medium onlin platon what 20 ad attract billion ...
```

По завершении работы утилита `extractTrainingData` печатает, сколько документов было найдено в каждой категории:

```
5417 sports
2162 nation
1777 politics
1735 technology
778 entertainment
611 business
241 arts
147 music
115 movies
80 computer
60 television
32 internet
```

Видно, что в одних категориях заметно больше документов, чем в других. Это может сказаться на верности классификатора. Некоторые алгоритмы классификации, в частности наивный байесовский, плохо реагируют на несбалансированность обучающих данных – вероятности признаков в категориях с большим числом примеров будут точнее, чем в более бедных категориях.

> **Бутстрапинг.** Этот процесс формирования набора обучающих документов с помощью простых правил называется *бутстрапингом*. В рассмотренном примере классификатору предъявляются документы, которым назначена одна из существующих категорий. Бутстрапинг часто необходим, потому что получить правильно помеченные данные обычно трудно. Во многих случаях данных для качественного обучения классификатора просто нет в достаточном количестве. А бывает и так, что данные поступают из разных источников, применяющих несовместимые схемы классификации. Бутстрапинг на основе ключевых слов позволяет сгруппировать документы в зависимости от наличия общих слов в их описании. Возможно, не все документы в данной категории отвечают этому правилу, но оно все же дает возможность сгенерировать достаточное число примеров для обучения классификатора. Существует множество разнообразных методик бутстрапинга. В одних для формирования начального набора категорий применяются короткие документы, в других – результаты работы других алгоритмов, например кластеризации, или даже классификаторов другого типа. Нередко различные методы бутстрапинга комбинируются, чтобы пополнить набор обучающих данных.

7.4.3. Резервирование тестовых данных

Теперь нужно зарезервировать часть обучающих данных для последующего тестирования. Нам нужно будет проверить, что катего-

рии, присвоенные тестовым документам обученным классификатором, совпадают с заранее известными. В состав кода, прилагаемого к книге, включена утилита для выполнения простого разбиения всего набора подготовленных данных на обучающие и тестовые – `SplitBayesInput`. При ее запуске мы указываем каталог, куда записывать результаты, а она создает в нем два подкаталога: с обучающими и с тестовыми данными. `SplitBayesInput` запускается следующим образом:

```
$TT_HOME/bin/tt splitInput \
 -i category-bayes-data \
 -tr category-training-data \
 -te category-test-data \
 -sp 10 -c UTF-8
```

В данном случае мы записываем в каталог тестовых данных 10 % документов из каждой категории, остальные – в каталог обучающих данных. Класс `SplitBayesInput` поддерживает несколько методов разбиения данных на обучающие и тестовые.

7.4.4. Обучение классификатора

Итак, обучающие данные с помощью утилиты `SplitBayesInput` подготовлены. Настало время засучить рукава и обучить свой первый классификатор. Если вы работаете в кластере Hadoop, скопируйте обучающие и тестовые данные в распределенную файловую систему Hadoop и выполните показанную ниже команду для построения модели классификатора. В противном случае данные будут читаться из текущего каталога вне зависимости от наличия флага `-source hdfs`:

```
$MAHOUT_HOME/bin/mahout trainclassifier \
-i category-training-data \
-o category-bayes-model \
-type bayes -ng 1 -source hdfs
```

Время обучения зависит от объема обучающих данных и от режима работы: локальный или распределенный в кластере Hadoop.

Если обучение завершится успешно, то модель записывается в выходной каталог, указанный в команде. В этом каталоге будет создано несколько файлов в формате файла последовательности Hadoop. Файл в таком формате содержит пары ключ-значение, именно так обычно выглядит результат работы каркаса MapReduce в Hadoop. Ключи и значения могут быть значениями примитивных типов или объектами Java, сериализованными средствами Hadoop. В состав

Apache Mahout входит несколько утилит для просмотра содержимого таких файлов:

```
$MAHOUT_HOME/bin/mahout seqdumper \
 -s category-bayes-model/trainer-tfIdf/trainer-tfIdf/part-00000 | less
```

Файлы в каталоге trainer-tfIdf содержат список всех признаков, который наивный байесовский классификатор будет использовать для классификации. Утилита `seqdumper` распечатывает их в следующем виде:

```
no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
Input Path: category-bayes-model/trainer-tfIdf/trainer-tfIdf/part-00000
Key class: class org.apache.mahout.common.StringTuple
Value Class: class org.apache.hadoop.io.DoubleWritable
Key: [__WT, arts, 000]: Value: 0.9278920383255315
Key: [__WT, arts, 1]: Value: 2.4908377174081773
...
Key: [__WT, arts, 97]: Value: 0.8524586871132804
Key: [__WT, arts, a]: Value: 9.251850977219403
Key: [__WT, arts, about]: Value: 4.324291341340667
...
Key: [__WT, business, beef]: Value: 0.5541230386115379
Key: [__WT, business, been]: Value: 7.833436391647611
Key: [__WT, business, beer]: Value: 0.6470763007419856
...
Key: [__WT, computer, design]: Value: 0.9422458820512981
Key: [__WT, computer, desktop]: Value: 1.1081452859525993
Key: [__WT, computer, destruct]: Value: 0.48045301391820133
Key: [__WT, computer, develop]: Value: 1.1518455320100698
...
```

Часто полезно просмотреть содержимое файла и понять, соответствуют ли признаки, включенные в обучающие данные, реально извлеченным. Распечатка может показать, что неправильно отфильтровываются стоп-слова, что странно работает стеммер или что не генерируются ожидаемые *n*-граммы. Полезно также проверить количество признаков, поскольку от этой величины зависит объем памяти, потребляемой байесовским классификатором.

7.4.5. Тестирование классификатора

После того как классификатор обучен, его качество следует проверить на ранее зарезервированных тестовых данных. Показанная ниже команда загружает созданную на предыдущем этапе модель в память и классифицирует все документы из тестового набора. Метка,

присвоенная документу классификатором, сравнивается с той, что присвоена вручную, и подводятся итоги.

```
$MAHOUT_HOME/bin/mahout testclassifier \
 -d category-test-data \
 -m category-bayes-model \
 -type bayes -source hdfs -ng 1 -method sequential
```

По завершении тестирования печатаются сводные показатели: процентные доли правильно и неправильно классифицированных документов и матрица неточностей. С ними мы познакомились в разделе 7.3.5.

```
=======================================================
Сводные данные
-------------------------------------------------------
Правильно классифицированные образцы      : 906        73.6585%
Неправильно классифицированные образцы    : 324        26.3415%
Всего классифицировано образцов           : 1230
=======================================================
Матрица неточностей
-------------------------------------------------------
a   b   c   d   e   f   g   h   i   j   k   l   ... <--Классифицирован как
0   0   0   0   5   0   0   0   1   0   3   2   | 11  a = movies
0   0   0   0   0   0   0   0   1   0   1   4   | 6   b = computer
0   0   0   0   0   0   0   0   0   0   1   2   | 3   c = internet
0   0   0   4   0   0   0   5   4   0   4   42  | 59  d = business
0   0   0   1   26  0   0   6   10  0   18  10  | 71  e = enter...
0   0   0   0   2   0   0   0   1   0   3   0   | 6   f = television
0   0   0   0   7   0   1   0   0   2   4   0   | 14  g = music
0   0   0   0   0   0   0   103 43  0   10  10  | 166 h = politics
0   0   0   0   1   0   0   25  145 0   16  10  | 197 i = nation
0   0   0   0   8   0   0   3   7   1   3   1   | 23  j = arts
1   0   0   0   1   0   0   1   7   0   493 4   | 507 k = sports
0   0   0   0   0   0   0   15  12  0   7   133 | 167 l = technology
Категория по умолчанию: unknown: 12
```

В данном случае матрицу неточностей можно использовать для настройки процесса бутстрапинга. Матрица показывает, что классификатор отлично справился с документами на спортивную тематику: 493 из 507 предъявленных документов о спорте были классифицированы правильно. Документы на технические темы (technology) были правильно классифицированы в 133 случаях из 167. С фильмами (movies) дело обстоит хуже: из 11 предъявленных документов с меткой movies ни одному не была присвоена правильная категория. Большая их часть попала в категорию entertainment (развлечения). Оно и понятно, потому что кино – один из видов развлечения, а в

обучающем наборе документов в категории entertainment (778) было куда больше, чем в категории movie (115). Это наглядная демонстрация эффекта несбалансированного обучающего набора и пересекающихся категорий. Развлечения напрочь забивают кино благодаря своему количеству, и в то же время мы видим, что документы, касающиеся развлечений, были неправильно отнесены к категориям nation (в стране), sports (спорт) и technology (техника), поскольку в этих категориях больше обучающих данных. Этот пример показывает, что верность классификации можно было бы повысить за счет более тщательного разделения тематик и более сбалансированного обучающего набора.

7.4.6. Усовершенствованный процесс бутстрапинга

В предыдущем примере мы определяли каждый класс документов только одним термом. Утилита `ExtractTrainingData` строила группы для каждого класса, отыскивая документы, содержащие соответствующий этому классу терм, в одном из полей categoryFacet или source. В результате получился классификатор, который путал классы из-за схожести тематик и несбалансированности обучающих данных для разных категорий. Чтобы решить эту проблему, мы будем определять классы документов с помощью нескольких термов. Это позволит свернуть все относящиеся к спорту категории в одну, а все относящиеся к развлечениям – в другую. Помимо объединения похожих категорий, этот подход позволяет включить в обучающий набор дополнительные документы из индекса Lucene.

Создайте файл training-categories-mult.txt, содержащий такие метки:

```
Sports MLB NFL MBA Golf Football Basketball Baseball
Politics
Entertainment Movies Music Television
Arts Theater Books
Business
Technology Internet Computer Science
Health
Travel
```

Первое слово в каждой строке станет названием категории, а следующие за ним слова будут использоваться для поиска документов. Если любой встречающийся в строке терм присутствует в поле categoryFacet или source документа, то этот документ включается в обучающий набор для данной категории. Например, любой документ,

содержащий строку *MLB* в поле categoryFacet, будет считаться принадлежащим категории Sports, документы, содержащие в этом поле терм *music*, – принадлежащими категории Entertainment, а документы, содержащие терм *Computer*, – категории Technology.

Снова запустим утилиту `ExtractTrainingData`:

```
$TT_HOME/bin/tt extractTrainingData \
 --dir index \
 --categories training-categories-mult.txt \
 --output category-mult-bayes-data \
 --category-fields categoryFacet,source \
 --text-fields title,description \
 --use-term-vectors
```

Результат записывается в каталог categories-mult-bayes-data, а на экране отображаются сводные данные о числе документов в каждой категории:

```
Category document counts:
5139 sports
1757 technology
1676 politics
988  entertainment
591  business
300  arts
173  health
12   travel
```

Скорее всего, вы не сможете обучить классификатор правильно классифицировать документы в категории travel (путешествия), потому что примеров слишком мало. Поэтому нужно либо подобрать еще примеры на эту тему, либо вообще отказаться от категории travel, но мы решили ее оставить для демонстрации результата.

Снова произведем разбиение, обучение и тестирование:

```
$TT_HOME/bin/tt splitInput \
 -i category-mult-bayes-data \
 -tr category-mult-training-data \
 -te category-mult-test-data \
 -sp 10 -c UTF-8

$MAHOUT_HOME/bin/mahout trainclassifier \
 -i category-mult-training-data \
 -o category-mult-bayes-model \
 -type bayes -source hdfs -ng 1

$MAHOUT_HOME/bin/mahout testclassifier \
 -d category-mult-test-data \
```

```
-m category-mult-bayes-model \
-type bayes -source hdfs -ng 1 \
-method sequential
```

Результат тестирования показывает, что новый классификатор стал лучше и теперь правильно назначает категории в 79,5 % случаев:

```
=======================================================
Сводные данные
-------------------------------------------------------
Правильно классифицированные образцы     : 846       79.5113%
Неправильно классифицированные образцы   : 218       20.4887%
Всего классифицировано образцов          : 1064
=======================================================
Матрица неточностей
-------------------------------------------------------
a   b   c   d   e   f   g   h   <--Классифицирован как
0   0   0   0   0   0   1   0   | 1   a = travel
0   3   0   0   3   0   5   43  | 59  b = business
0   0   2   1   7   1   2   4   | 17  c = health
0   1   0   57  12  1   19  9   | 99  d = entertainment
0   0   0   0   142 0   14  12  | 168 e = politics
0   0   0   17  3   3   4   3   | 30  f = arts
0   1   0   3   9   0   495 6   | 514 g = sports
0   1   0   1   23  0   7   144 | 176 h = technology
Категория по умолчанию: unknown: 8
```

Отсюда видно, что качество классификации улучшилось на 6 % – не так уж плохо. И хотя мы движемся в правильном направлении, матрица неточностей показывает, что кое-какие проблемы остались.

К счастью, в этом примере у нас имеются широкие возможности для получения обучающих данных и выбора схемы классификации. Прежде всего, понятно, что данных в категории travel явно не хватает, т. к. большая часть принадлежащих ей документов не классифицирована вовсе. Категории health (здоровье) и arts (искусство) страдают от той же проблемы, большая часть документов в них классифицирована неправильно. Тот факт, что большинство документов, относящихся к искусству, попали в категорию entertainment, наводит на мысль, что эти классы стоило бы объединить.

7.4.7. Интеграция байесовского классификатора Mahout с Solr

Обученный классификатор необходимо внедрить в эксплуатацию. В этом разделе мы покажем, как байесовский классификатор из проекта Mahout интегрируется с индексатором поисковой системы Solr в

качестве классификатора документов. По мере того как Solr загружает данные в индекс Lucene, мы прогоняем их через классификатор и генерируем значение поля category, которое будет использоваться как дополнительный поисковый терм или для фасетного отображения результатов.

Для этого мы создадим собственный класс `UpdateRequestProcessor`, который Solr будет вызывать при получении запроса на обновление индекса. На этапе инициализации этот процессор запросов загружает модель обученного байесовского классификатора и в ходе обработки анализирует и классифицирует содержимое полученного документа. Объект `UpdateProcessor` помещает метку категории в отдельное поле объекта `SolrDocument`, добавляемого в индекс Lucene.

Сначала необходимо добавить новую цепочку процессоров запросов на обновление (см. `org.apache.solr.update.processor.UpdateRequestProcessorChain`) в Solr, определив ее в файле solrconfig.xml. Эта цепочка определяет фабрики, используемые для создания объекта, который будет обрабатывать запросы на обновление. Класс `BayesUpdateRequestProcessorFactory` порождает объект, который назначает категорию, `RunUpdateProcessorFactory` добавляет категорию в конструируемый Solr индекс Lucene, а `LogUpdateProcessorFactory` собирает статистику обновлений и записывает ее в журналы Solr.

Листинг 7.8. Конфигурация цепочки процессоров запросов на обновление в файле solrconfig.xml

```
<updateRequestProcessorChain key="mahout" default="true">
 <processor class=
   "com.tamingtext.classifier.BayesUpdateRequestProcessorFactory">
 <str name="inputField">details</str>
 <str name="outputField">subject</str>
 <str name="model">src/test/resources/classifier/bayes-model</str>
 </processor>
 <processor class="solr.RunUpdateProcessorFactory"/>
 <processor class="solr.LogUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

Для конфигурирования `BayesUpdateRequestProcessorFactory` в параметре `inputField` указывается имя поля, содержащего подлежащий классификации текст, в параметре `outputField` – имя поля, в которое записывается метка класса, а в параметре `model` – путь к модели классификации. В необязательном параметре `defaultCategory` задается категория, которая записывается в документ, если его не

удалось классифицировать. Обычно это случается, когда во входном документе нет признаков, присутствующих в модели. Фабрика создается, когда Solr инициализирует свои подключаемые модули. На этом этапе проверяются параметры и в процессе инициализации объекта Mahout `Datastore` загружается модель. Создается объект алгоритма классификации и с учетом всех вышеперечисленных элементов инициализируется объект `ClassifierContext`.

В листинге ниже показано, как модель классификатора загружается в объект `InMemoryBayesDatastore`.

Листинг 7.9. Инициализация объекта ClassifierContext

```
BayesParameters p = new BayesParameters();
p.set("basePath", modelDir.getCanonicalPath());
Datastore ds = new InMemoryBayesDatastore(p);
Algorithm a = new BayesAlgorithm();
ClassifierContext ctx = new ClassifierContext(a,ds);
ctx.initialize();
```

Этот подход годится для небольших моделей, обученных на скромном числе признаков, но не подходит для моделей, не помещающихся в оперативной памяти. На этот случай Mahout предлагает альтернативное хранилище, читающее данные из базы HBase. Кроме того, сравнительно просто можно реализовать и другие хранилища.

Инициализированный объект `ClassifierContext` сохраняется в переменной-члене объекта `BayesUpdateRequestProcessorFactory` и внедряется в каждый экземпляр `BayesUpdateRequestProcessor`, создаваемый Solr при получении запроса на обновление. Запрос на обновление поступает в виде одного или нескольких объектов типа `SolrInputDocument`. С помощью Solr API очень просто извлечь содержимое любого поля документа, а затем выполнить предобработку и классификацию документа, применяя инициализированный ранее контекст классификатора. В листинге 7.10 показано, как производится предобработка с использованием анализатора Solr, который выполняет действия в соответствии с конфигурацией входных полей в схеме Solr и записывает результат в массив `String[]`, подаваемый на вход контекста классификатора Mahout. Анализатор Solr согласован с API класса Lucene `Analyzer`, поэтому показанный ниже код лексического анализа используется в любом контексте, работающем с анализаторами Lucene.

Листинг 7.10. Лексический анализ SolrInputDocument с помощью анализатора Solr

```
String input = (String) field.getValue();

ArrayList<String> tokenList = new ArrayList<String>();
TokenStream ts = analyzer.tokenStream(inputField, new StringReader(input));
while (ts.incrementToken()) {
 tokenList.add(ts.getAttribute(CharTermAttribute.class).toString());
}
String[] tokens = tokenList.toArray(new String[tokenList.size()]);
```

Коль скоро лексемы, необходимые классификатору, имеются, для получения результата достаточно вызвать метод `classifyDocument` объекта Mahout `ClassifierContext`. В листинге 7.11 показано, что эта операция возвращает объект `ClassifierResult`, содержащий метку присвоенного документу класса. Метод `classifyDocument` принимает также категорию по умолчанию, которая присваивается, если не удается классифицировать документ – например, если во входном документе и в модели нет общих слов. Если полученный результат отличен от `null` и не совпадает с категорией по умолчанию, представленной константой `NO_LABEL`, то метка записывается в поле документа `SolrInputDocument`.

Листинг 7.11. Классификация SolrInputDocument с помощью контекста ClassifierContext

```
SolrInputField field = doc.getField(inputField);
String[] tokens = tokenizeField(inputField, field);
ClassifierResult result = ctx.classifyDocument(tokens, defaultCategory);
if (result != null && result.getLabel() != NO_LABEL) {
 doc.addField(outputField, result.getLabel());
}
```

Недостаток этого подхода заключается в том, что результаты лексического анализа сохраняются в памяти, чтобы сделать их доступными классификатору. Возможно, в будущем Mahout научится читать поток лексем непосредственно. Второй недостаток – необходимость дважды подвергать поле лексическому анализу на этапе индексирования. Первый раз это происходит во время классификации, а второй – позже, при добавлении лексем в индекс Lucene.

Но если оставить эти проблемы в стороне, то описанный механизм эффективно классифицирует документы при добавлении их в индекс Solr, демонстрируя возможность использовать API байесов-

ского классификатора Mahout для классификации документов из программы. В своих проектах вы можете использовать любой из этих механизмов или оба сразу: для пометки документов во время индексирования или для автоматической классификации с помощью классификатора Mahout.

В этом разделе мы рассмотрели наивный байесовский алгоритм статистической классификации, который определяет вероятности слов при условии заданной категории, основываясь на наблюдениях, представленных обучающим набором данных. Далее с помощью теоремы Байеса из этой условной вероятности вычисляется вероятность категории при условии заданного набора слов в документе. В следующем разделе мы опишем другой статистический алгоритм, который также моделирует вероятность категории при условии набора слов, опуская этап предварительного определения инверсной условной вероятности.

Мы также обсудили способы использования обучающих данных, собранных из Интернета. Мы рассмотрели процесс бутстрапинга, поэкспериментировали с различными вариантами группировки обучающих документов и показали, как объем имеющихся обучающих данных отражается на верности классификатора. Мы продолжим наши исследования в разделе 7.5, где познакомимся с использованием именованных сущностей как способе пополнения обучающих данных.

7.5. Классификация документов с помощью OpenNLP

Алгоритм наивной байесовской классификации, рассмотренный в разделе 7.4, строил статистическую модель на основе связей между признаками и категориями, обнаруженными в обучающих данных. В этом разделе мы рассмотрим алгоритм максимальной энтропии из проекта OpenNLP, применяемый для классификации текстов. Этот алгоритм строит модель на основе той же информации, что и байесовский алгоритм, но исходит из других идей – использует алгоритм регрессии для определения того, как имеющиеся в документе признаки соотносятся с категориями. В процессе обучения производится итеративный анализ обучающего корпуса с целью найти веса всех признаков и построить математическую формулу, порождающую результаты, максимально близкие к наблюдаемым. В этом разделе мы дадим краткий обзор регрессионных моделей, чтобы объяснить принцип работы алгоритма и соотнести его идеи с нашей главной задачей – классификацией текстов.

Попутно мы познакомимся еще с одной полезной частью проекта OpenNLP – API распознавателя имен. Этот API был представлен в главе 5, когда мы рассказывали о его применении для выявления именованных сущностей: имен людей, названий мест и т. п. В этом разделе мы воспользуемся такими сущностями для повышения качества классификатора документов MaxEnt. В качестве признаков будут использоваться не только отдельные слова, встречающиеся в обучающих данных, но и словосочетания, которые OpenNLP распознает как именованные сущности, например *New York City*.

Распознаватель имен OpenNLP является также классификатором. Он обучен находить слова, который выглядят как именованные сущности, исходя из различных признаков. В дистрибутив OpenNLP входят модели, «заточенные» под извлечение именованных сущностей различных типов. Поэтому чтобы воспользоваться API, необязательно обучать собственный распознаватель, хотя эта возможность также остается открытой.

Итак, помимо классификации документов, мы будем использовать отдельную функцию OpenNLP – генерацию признаков, на основе которых принимаются решения о классификации. Здесь мы видим пример *совмещения* (piggybacking), когда один классификатор – документов – обучается на результатах другого классификатора – распознавателя именованных сущностей. Это распространенная практика. Вы можете также встретиться с ситуацией, когда классификатор применяется для определения границ предложений, слов или частей речи с целью генерации признаков для классификации документов. Нужно помнить, что качество классификатора, получающего данные, неразрывно связано с качеством классификатора, генерирующего признаки.

Прочитав этот раздел до конца, вы будете понимать, как работает алгоритм максимальной энтропии, а также терминологию, применяемую в его коде и ее связь с категоризацией документов. На примере мы покажем, как API категоризации документов и распознавателя имен работают совместно, и проделаем весь путь построения классификатора от обучения до оценки качества.

7.5.1. Регрессионные модели и классификация документов методом максимальной энтропии

Мультиномиальная логистическая регрессия, применяемая в классификаторе MaxEnt из проекта OpenNLP – одна из многих сущес-

твующих *регрессионных моделей*. Вообще говоря, назначение регрессионных моделей – определить связь между зависимой переменной и несколькими независимыми переменными. Любая регрессионная модель – это математическая функция, в которой каждому признаку приписан некоторый вес. Результат комбинирования весов со значениями признаков и последующего комбинирования взвешенных признаков и является предсказанием модели. Задача алгоритма регрессии – определить подходящие веса для каждого признака, исходя из наблюдаемых для обучающих данных предсказаний.

На рис. 7.4 показана тривиальная регрессионная модель, применяемая для предсказания быстродействия компьютерной программы. Эта модель связывает количество процессоров со временем работы программы. В ней всего одна независимая переменная, или признак – количество процессоров. Зависимой переменной является время работы. Каждая точка на графике представляет один элемент набора обучающих данных, которые были получены прямым измерением времени работы программы на машинах с различным числом процессоров. Всего у нас имеется пять обучающих примеров: от 1000 мс для одного ЦП до примерно 100 мс для 8 ЦП.

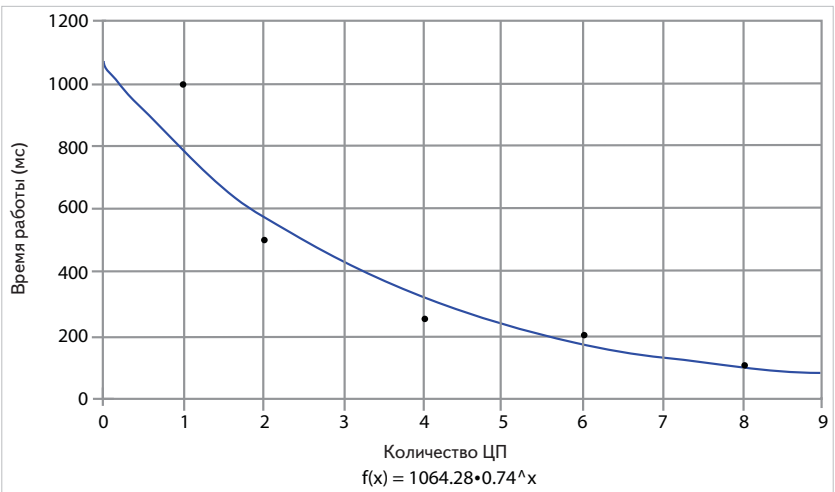

Рис. 7.4. Простая двумерная регрессионная модель. Точки соответствуют наблюдаемым данным. Линия представляет собой график функции, показанной под рисунком, которая позволяет производить экстраполяцию на ненаблюдавшиеся значения. Тот же принцип применим к классификации текстов, когда каждое слово считается отдельным измерением

Алгоритм регрессии пытается построить функцию, которая позволит предсказывать время работы программы на машинах с таким числом процессоров, для которого время не измерялось. Кривая на графике – это график функции, найденной алгоритмом регрессии. Из нее следует, что на машине с 3 процессорами программа будет работать примерно 450 мс, а на машине с 7 процессорами – 110 мс. Под графиком приведена формула, найденная алгоритмом регрессии для данного случая: возвести 0,74 в степень равную числу процессоров, и умножить результат на 1064,28. Значение, с помощью которого взвешивается независимая переменная (здесь оно равно 0,74), называется *параметром*; другая переменная, равная в данном случае 1064,28, называется *поправочным коэффициентом*. В регрессионной модели каждый признак сопровождается параметром, используемым для его взвешивания, а поправочный коэффициент один для всей формулы. Алгоритм регрессии определяет оптимальные значения всех параметров и поправочного коэффициента, так чтобы отклонение получающейся кривой от обучающих данных было минимальным. В этом тривиальном примере независимая переменная всего одна, но в типичных ситуациях, где применяются регрессионные модели, количество независимых переменных очень велико, и нужно найти значения всех параметров.

Существует много вариантов регрессионных моделей, в которых по-разному сочетаются независимые переменные, параметры и поправочные коэффициенты. В рассмотренной выше экспоненциальной модели каждый параметр возводится в степень, равную значению независимой переменной, и результат умножается на поправочный коэффициент. В линейной модели параметры умножаются на независимые переменные, а результаты складываются и умножаются на поправочный коэффициент. Разные регрессионные формулы приводят к кривым разной формы.

Если отвлечься от базовой структуры формулы, то алгоритмы регрессии отличаются способом поиска значений параметров. Отсюда и их названия, например, *градиентный спуск* или *итеративное шкалирование*. В каждом алгоритме применяется свой подход к поиску наилучших весов признаков, позволяющих получить ожидаемое предсказание.

Алгоритмы регрессии отличаются также порождаемыми результатами. Одни порождают непрерывную функцию, как в приведенном выше примере, другие двоичные результаты – да или нет. Алгоритм мультиномиальной логистической регрессии, применяемый в класси-

фикаторе с максимальной энтропией, порождает результат, отражающий последовательность дискретных предсказаний, например, набор категорий. Алгоритм максимальной энтропии ассоциирует с каждым возможным предсказанием вероятность; предсказание с наибольшей вероятностью становится меткой, назначаемой классифицируемым входным данным.

Регрессионные модели используются для предсказания всего чего угодно: от вероятности сердечного приступа при различных влияющих на здоровье факторах до цен на недвижимость с учетом местоположения, метража и количества спален. Как уже было сказано, сама модель – не более чем параметры, включенные в состав функции регрессии. Построение регрессионной модели сводится к подгонке наблюдаемых признаков под предсказания, воплощенные в обучающих данных, а ее применение для выдачи нового предсказания – к подстановке значений признаков и вычислению результата с использованием хранящихся в модели весов.

И какое же отношение все это имеет к классификации текстов? Результат регрессионной модели (зависимая переменная) соответствует результату классификатора: вырабатываемой им метке категории. Входными данными для регрессионной модели являются независимые переменные, которые в контексте классификации именуются признаками; таковыми являются аспекты текста, например, термы. Регрессионные модели, применяемые для классификации текстов, велики, потому что каждое уникальное слово в корпусе текстов трактуется как независимая переменная. Обучение усложняется разреженностью текста; в каждом обучающем примере имеется информация, относящаяся к сравнительно небольшой выборке всех независимых переменных, образующих функцию регрессии. Понятно, что определение подходящих весов при большом числе независимых переменных и так уже достаточно трудная задача, а тут еще приходится иметь дело с отсутствием полного набора данных.

Терминология, употребляемая для описания классификации в OpenNLP API и в исходном коде, более общая, чем та, что используется в этой главе применительно к классификации текстов. Чтобы понять, как выглядит классификация документов в OpenNLP, необходимо сначала разобраться в связях между этими предметными областями. В терминологии классификатора OpenNLP признаки, на которых производится обучение, называются *предикатами*. Предикаты встречаются в контекстах, причем любой предикат может встречаться в нескольких контекстах. Применительно к классифи-

кации документов предикаты – это слова или иные признаки документов. *Контексты* – это документы. Обучающий корпус состоит из множества контекстов. С каждым контекстом ассоциировано *предсказание*. Предсказания – это эквиваленты меток категорий, которые классификатор присваивает документам. Эти концепции можно также отобразить на язык регрессионных моделей. Предикат (признак, терм) – это независимая переменная. Всякая независимая переменная служит для прогнозирования значения зависимой переменной – предсказания или метки категории. Корпус обучающих данных состоит из контекстов, которые сопоставляют предсказания предикатам. Это сопоставление и есть наблюдения, на которых производится обучение и тестирование. На этапе обучения модели мы сравниваем наблюдаемые результаты с результатами, предсказанными моделью, и на этой основе решаем, как улучшить модель.

В процессе обучения каждый уникальный терм (предикат), присутствующий в корпусе, сравнивается с порожденным им предсказанием, и выполняется серия итераций, чтобы найти наилучший вес для каждого предиката, дающий желаемое предсказание. На каждой итерации улучшается способность уравнения регрессии порождать результаты, приближенные к обучающим данным.

7.5.2. Подготовка обучающих данных для классификатора документов на основе алгоритма максимальной энтропии

В примерах из этого раздела мы будем использовать те же обучающие данные, которые собирали для байесовского классификатора из проекта Mahout в разделе 7.4, но можете, если хотите, взять данные из корпуса 20 Newsgroups (см. раздел 7.3).

В отличие от байесовского классификатора, классификатор MaxEnt сам выполняет стемминг для обрабатываемого текста. Поэтому для извлечения обучающих данных из индекса Lucene понадобится немного другая команда, которая берет не уже прошедшие стемминг термы из вектора термов, а необработанный текст, хранящийся в полях индекса.

```
$TT_HOME/bin/tt extractTrainingData \
 --dir index \
 --categories training-categories.txt \
 --output category-maxent-data \
 --category-fields category,source \
 --text-fields title,description
```

Взглянув на обучающие данные, вы убедитесь, что термы не подвергались стеммингу, а слова не преобразованы в нижний регистр. Регистр важен для распознавания именованных сущностей классификатором MaxEnt:

```
arts 6 Stores Across the World Are a Feast for Design Nuts A few ...
arts For Old Masters, It's Dealers' Choice While auction houses ...
arts Alan Moore Digs Up 'Unearthing' and Lays His Comics Career ...
...
business Citigroup says iPhone banking app stored data Citigroup ...
business United Tech plans 1,500 more job cuts HARTFORD, Conn. - ...
business New-home sales up, but no cause for glee New-home sales ...
...
computer What's for Sale on the Bug Market? Almost any ...
computer The Web Means the End of Forgetting The digital age ...
computer The Medium: What 'Platonic' Means Online Craigslist ...
```

Далее мы выполняем команду `splitInput` для разбиения всего множества данных на обучающий и тестовый наборы:

```
$TT_HOME/bin/tt splitInput \
 -i category-maxent-data \
 -tr category-maxent-training-data \
 -te category-maxent-test-data \
 -sp 10 -c UTF-8
```

Подготовив оба набора, мы можем обучить классификатор MaxEnt.

7.5.3. Обучение классификатора документов на основе алгоритма максимальной энтропии

Классификатор документов входит в дистрибутив проекта OpenNLP, но для его использования придется изрядно попрограммировать. Далее описан код, необходимый для обучения и тестирования классификатора, построенного с помощью класса OpenNLP `DocumentCategorizer`.

Классы `TrainMaxent` и `TestMaxent` реализуют классификатор, запускаемый из командной строки. Для его обучения служит команда `trainMaxent`. Каталог с исходными данными, задаваемый с помощью флага `-i`, должен содержать обучающие данные в уже знакомом нам формате: по одному файлу на категорию и в каждой строке файла один документ. Классификатор документов MaxEnt ожидает получить текст в виде набора разделенных пробелами слов, не подвергав-

шихся стеммингу и преобразованию регистра. Это необходимо для распознавания именованных сущностей. Флаг -o задает имя файла, в который будет записана модель MaxEnt.

```
$TT_HOME/bin/tt trainMaxent \
 -i category-maxent-training-data \
 -o maxent-model
```

Рассмотрим внимательнее код обучения классификатора OpenNLP и то, как можно приспособить этот процесс под свои нужды.

Для обучения модели MaxEnt необходимо подготовить входной каталог и выходные файлы, создать источник необработанных данных и генераторы признаков, которые преобразуют обучающие данные в признаки, а затем передать все это методу train, который построит модель по обучающему набору. Весь процесс показан в листинге ниже.

Листинг 7.12. Обучение классификатора DocumentCategorizer

```
File[] inputFiles = FileUtil.buildFileList(new File(source));
File modelFile = new File(destination);

Tokenizer tokenizer = SimpleTokenizer.INSTANCE;
CategoryDataStream ds = new CategoryDataStream(inputFiles, tokenizer);

int cutoff = 5;
int iterations = 100;

NameFinderFeatureGenerator nffg
  = new NameFinderFeatureGenerator();
BagOfWordsFeatureGenerator bowfg
  = new BagOfWordsFeatureGenerator();

DoccatModel model = DocumentCategorizerME.train("en",
    ds, cutoff, iterations, nffg, bowfg);
model.serialize(new FileOutputStream(modelFile));
```

Создаем поток данных ❶

❷ Настраиваем генераторы признаков

❸ Обучаем классификатор

В точке ❶ мы инициализируем лексический анализатор `SimpleTokenizer` и поток `CategoryDataStream` для чтения лексем и меток категорий из файлов обучающих данных.

В точке ❷ мы создаем объекты `NameFinderFeatureGenerator` и `BagOfWordsFeatureGenerator`, которые порождают признаки: необработанные термы из документа и именованные сущности, выделенные распознавателем из проекта OpenNLP.

Создав поток данных и генераторы признаков, мы обучаем модель классификатора с помощью класса `DocumentCategorizerME`. В точ-

ке ❸ методу `train()` передается поток данных, генераторы признаков и параметры обучения, а после возврата из него обученная модель записывается на диск методом `serialize()`.

Поучительно более пристально присмотреться к лексическому анализу и генерации признаков, чтобы понять, как обучающие данные преобразуются в события, используемые для обучения классификатора. В следующем листинге показано, как объект `CategoryDatastream` порождает образцы `DocumentSample` по обучающим данным.

Листинг 7.13. Порождение образцов DocumentSample по обучающим данным

```
public DocumentSample read() {
 if (line == null && !hasNext()) {   <-- ❶ Читаем обучающие данные
  return null;
 }
 int split = line.indexOf('t');      <-- ❷ Извлекаем категорию
 if (split < 0)
  throw new RuntimeException("Invalid line in "
     + inputFiles[inputFilesIndex]);
 String category = line.substring(0,split);
 String document = line.substring(split+1);
 line = null; // mark line as consumed
 String[] tokens = tokenizer.tokenize(document);   <-- ❸ Разбиваем содержимое на лексемы
 return new DocumentSample(category, tokens);      <-- ❹ Создаем образец
}
```

В точке ❶ метод `read()` объекта `CategoryDataStream` получает строки из входных данных, вызывая метод `hasNext()`, который неявно читает несколько строк обучающих данных и поочередно предоставляет их в переменной `line`. По достижении конца обучающих данных переменной `line` присваивается значение `null`. После чтения очередной строки код в точке ❷ извлекает из нее категорию и содержимое документа. Затем содержимое документа разбивается на лексемы в точке ❸, в результате чего получается коллекция термов, которые будут использоваться в качестве признаков в процессе обучения. Наконец, в точке ❹ создается объект `DocumentSample` с категорией и лексемами, обнаруженными в предъявленном для обучения образце.

Внутри `DocumentCategorizerME` коллекция объектов `DocumentSample` передается генераторам признаков с помощью объекта `DocumentCategorizerEventStream`, порождающего поток событий, на котором обучается модель. Каждое событие состоит из

предсказания и контекста. Предсказания – это метки категорий, контексты – наборы слов, созданные в ходе лексического анализа содержимого документа.

Объекты событий `DocumentSample`, порожденные `CategoryDataStream`, преобразуются в признаки генераторами `NameFinderFeatureGenerator` и `BagOfWordsFeatureGenerator`. Последний является составной частью OpenNLP API, он возвращает найденные в примерах документов лексемы в виде набора признаков. Класс `NameFinderFeatureGenerator` пользуется API класса `NameFinder` для поиска именованных сущностей среди лексем и возвращает найденные сущности в виде признаков. Настройка класса OpenNLP `NameFinder` и загрузка различных моделей, применяемых для распознавания именованных сущностей, инкапсулирована в классе `NameFinderFactory`. В листинге ниже показано, как этот класс ищет и загружает модели.

Листинг 7.14. Загрузка моделей распознавателя имен

```
File modelFile;

File[] models                          <-- ❶ Ищем модели
  = findNameFinderModels(language, modelDirectory);

modelNames = new String[models.length];
finders = new NameFinderME[models.length];

for (int fi = 0; fi < models.length; fi++) {          Определяем имя модели
 modelFile = models[fi];                                        ❷
 modelNames[fi] = modelNameFromFile(language, modelFile);  <--
 log.info("Loading model {}", modelFile);
 InputStream modelStream = new FileInputStream(modelFile);
 TokenNameFinderModel model =                          <-- ❸ Читаем модель
    new PooledTokenNameFinderModel(modelStream);
 finders[fi] = new NameFinderME(model);
}
```

В точке ❶ метод `findNameFinderModels()` ищет в каталоге моделей файлы, содержащие модели. Затем найденные модели загружаются в массив, управляемый `NameFinderFactory`. В ходе загрузки метод `modelNameFromFile()` в точке ❷ преобразует имя файла в имя модели, удаляя путь и расширение имени. В точке ❸ объект `PooledTokenNameFinderModel` выполняет трудную работу по чтению и распаковке модели и размещению ее в памяти. Для каждой загруженной модели создается экземпляр класса `NameFinderME`. Моде-

ли хранятся в массиве, возвращенном методом `NameFinderFactory.getNameFinders()`[9].

Инициализировав объекты класса `NameFinder`, мы воспользуемся ими для выделения именованных сущностей из входных данных. Это делает показанный в следующем листинг метод из класса `NameFinderFeatureGenerator`. Входными данными для него являются образцы `DocumentSample`, порожденные `CategoryDataStream`.

Листинг 7.15. Использование класса NameFinderFeatureGenerator для генерации признаков

```
public Collection extractFeatures(String[] text) {
 NameFinderME[] finders = factory.getNameFinders();
 String[] modelNames    = factory.getModelNames();

 Collection<String> features = new ArrayList<String>();
 StringBuilder builder = new StringBuilder();

 for (int i=0; i < finders.length; i++) {
   Span[] spans = finders[i].find(text);
   String model = modelNames[i];

   for (int j=0; j < spans.length; j++) {
     int start = spans[j].getStart();
     int end = spans[j].getEnd();

     builder.setLength(0);
     builder.append(model).append("=");

     for (int k = start; k < end; k++ ) {
       builder.append(text[k]).append('_');
     }
     builder.setLength(builder.length()-1);
     features.add(builder.toString());
   }
 }
 return features;
}
```

❶ Получаем распознаватели имен

❷ Находим имена

❸ Выделяем имена

❹ Собираем имена

В точке ❶ мы получаем ссылки на объект `NameFinderME` и имена моделей, загруженных фабрикой `NameFinder`. Объект `NameFinderFactory` сохранил распознаватели и соответствующие им имена моделей в параллельных массивах. Каждая модель будет использоваться для распознавания соответствующего ей типа именованных сущностей: географических названий, имен людей, дат и моментов времени.

[9] Как видно из кода, массив объектов NameFinderME создается иначе. – *Прим. перев.*

В точке ❷ мы обрабатываем входные лексемы с помощью каждого из загруженных распознавателей `NameFinderME`, вызывая его метод `find`. Каждый интервал `Span` в массиве, возвращенном этим методом, описывает смещения начальной и конечной лексем именованной сущности в исходном тексте. В точке ❸ с помощью этих смещений генерируются строки, которые мы будем хранить в качестве признаков. Во всех случаях в начало строки дописывается имя модели, так что признаки имеют вид `location=New_York_City`.

Все сгенерированные признаки помещаются в список, который возвращается классификатору в точке ❹.

По ходу обучения на экран выводятся примерно такие сообщения:

```
Indexing events using cutoff of 5

Computing event counts... done. 10526 events
Indexing...
done.
Sorting and merging events... done. Reduced 10523 events to 9616.
Done indexing.
Incorporating indexed data for training...
done.
Number of Event Tokens: 9616
 Number of Outcomes: 12
 Number of Predicates: 11233
...done.
Computing model parameters...
Performing 100 iterations.
  1: .. loglikelihood=-26148.6726757207     0.0024707782951629764
  2: .. loglikelihood=-24970.114236056226   0.6394564287750641
  3: .. loglikelihood=-23914.53191047887    0.6485793024802813
...
 99: .. loglikelihood=-7724.766531988901    0.8826380309797586
100: .. loglikelihood=-7683.407561473442    0.8833982704551934
```

Еще до начала обучения классификатор документов должен разместить признаки, которые будет использовать для обучения, в индексе, где к ним можно будет быстро обратиться. Первые строки распечатки описывают результаты этого процесса. Каждый помеченный обучающий документ порождает событие, на этапе индексирования повторяющиеся события подсчитываются, и некоторые документы, не содержащие полезных признаков, могут быть отброшены.

Слово «cutoff» (отсечение) в распечатке обозначает минимальный порог частоты термов. Терм, который встречается в обучающем корпусе менее пяти раз, игнорируется. Документы, в которых все термы встречаются менее пяти раз, отбрасываются. Предика-

ты представляют термы, используемые для обучения; как видим, в данном случае корпус насчитывает 11 233 уникальных предиката. Сюда входят и однословные лексемы, порожденные генератором `BagOfWordsFeatureGenerator`, и именованные сущности, порожденные генератором `NameFinderFeatureGenerator`. В регрессионной модели каждый предикат является независимой переменной.

Из распечатки также видно, что по завершении процесса индексирования получилось 12 предсказаний и 9616 обучающих образцов (после исключения дубликатов).

После того как индексирование завершилось, начинается процесс обучения, отражаемый в распечатке. Всего произведено 100 итераций; на каждой итерации выполняется проход по всему набору обучающих данных с целью улучшить параметры модели и определить функцию регрессии. Логарифмическое отношение правдоподобия, вычисляемое на каждой итерации, служит для сравнения результатов модели с наблюдениями.

Логарифмическое отношение правдоподобия – это мера схожести двух моделей. Его значение – не абсолютный показатель, а относительная характеристика изменения модели от итерации к итерации. В идеале отношение правдоподобия должно приближаться к нулю. Это означает, что на каждом шаге модель дает результаты, все более близкие к наблюдавшимся в обучающих данных предсказаниям. Если логарифмическое отношение правдоподобия отдаляется от нуля, значит, модель ухудшается, что свидетельствует о потенциальных проблемах в обучающих данных. Нетрудно заметить, что логарифмическое отношение правдоподобия изменяется наиболее быстро на первых итерациях. Если вы заметите, что оно продолжает существенно варьировать после 100-й итерации, то можно попробовать обучение с большим числом итераций.

После 100-й итерации программа обучения записывает модель на диск, теперь ей можно пользоваться для классификации документов. В следующем разделе показаны вызовы API, с помощью которых модель используется в процессе тестирования.

7.5.4. Тестирование классификатора документов на основе алгоритма максимальной энтропии

К тестированию классификатора MaxEnt мы применим тот же подход, что в разделах 7.3.5 и 7.4.5: классифицируем сколько-то помечен-

ных документов и сравним изначальные и присвоенные классификатором метки. Класс `TestMaxent`, который это делает, вызывается следующей командой:

```
$TT_HOME/bin/tt testMaxent \
 -i category-maxent-test-data \
 -m maxent-model
```

Здесь мы используем тестовые данные, созданные утилитой `extractTrainingData`, и модель, порожденную командой `trainMaxent`. По завершении тестирования вы увидите уже знакомые процентные доли и матрицу неточностей.

На примере класса `TestMaxent` демонстрируется, как обученная модель загружается в память и используется для классификации документов. Код в листинге 7.16 загружает модель с диска и подготавливает конвейер лексического анализа для обработки документов. Обратите внимание на то, как этот код похож на тот, что использовался для обучения классификатора (листинг 7.12).

Листинг 7.16. Подготовка DocumentCategorizer

```
NameFinderFeatureGenerator nffg                    <-- ❶ Создаем генераторы признаков
  = new NameFinderFeatureGenerator();
BagOfWordsFeatureGenerator bowfg
  = new BagOfWordsFeatureGenerator();

InputStream modelStream =                          <-- ❷ Загружаем модель
    new FileInputStream(modelFile);
DoccatModel model = new DoccatModel(modelStream);
DocumentCategorizer categorizer                    <-- ❸ Создаем классификатор
  = new DocumentCategorizerME(model, nffg, bowfg);
Tokenizer tokenizer = SimpleTokenizer.INSTANCE;

int catCount = categorizer.getNumberOfCategories();
Collection<String> categories
  = new ArrayList<String>(catCount);
for (int i=0; i < catCount; i++) {
 categories.add(categorizer.getCategory(i));
}
                                                   <-- ❹ Подготавливаем анализатор результатов
ResultAnalyzer resultAnalyzer =
    new ResultAnalyzer(categories, "unknown");
runTest(inputFiles, categorizer, tokenizer, resultAnalyzer);   <-- ❺ Выполняем тестирование
```

Сначала в точке ❶ мы создаем генераторы признаков, а затем в точке ❷ загружаем модель с диска с помощью класса `DoccatModel`. Далее эта модель используется для создания экземпляра класса

DocumentCategorizer в точке ❸. Наконец, в точке ❹ мы инициализируем объект ResultAnalyzer, передавая его конструктору список категорий, полученных от классификатора с помощью модели. И в точке ❺ запускаем тестирование.

В следующем разделе мы рассмотрим код, с помощью которого классификатор документов на основе алгоритма максимальной энтропии интегрируется в производственную систему.

7.5.5. Классификатор документов на основе алгоритма максимальной энтропии в производственной системе

После того как модель загружена, а лексический анализатор, генераторы признаков, классификатор и анализатор результатов подготовлены, можно приступать к классификации документов. В листинге 7.17 показано, как прочитанные из файла документы обрабатываются, классифицируются и передаются экземпляру того же класса ResultAnalyzer, который мы использовали для оценки других классификаторов: MoreLikeThis в разделе 7.3.5 и байесовского в разделе 7.4.5.

Листинг 7.17. Классификация текста с помощью DocumentCategorizer

```
for (File ff: inputFiles) {
 BufferedReader in = new BufferedReader(new FileReader(ff));
 while ((line = in.readLine()) != null) {
   String[] parts = line.split("t");
   if (parts.length != 2) continue;

   String docText = parts[1];
   String[] tokens = tokenizer.tokenize(docText);

   double[] probs = categorizer.categorize(tokens);
   String label = categorizer.getBestCategory(probs);
   int bestIndex = categorizer.getIndex(label);
   double score = probs[bestIndex];

   ClassifierResult result = new ClassifierResult(label, score);
   resultAnalyzer.addInstance(parts[0], result);
 }
 in.close();
}

System.err.println(resultAnalyzer.toString());
```

❶ Предварительно обрабатываем текст

Классифицируем ❺

❹ Анализируем результаты

❺ Выводим результаты

Как вы помните, тестовые документы находятся во втором столбце в каждой строке входного файла. Сначала мы извлекаем текст документа, а затем с помощью лексического анализатора `SimpleTokenizer` разбиваем его на лексемы (❶). Эти лексемы далее передаются классификатору в точке ❷. Метод `categorize` генерирует признаки, вызывая объекты `BagOfWordsFeatureGenerator` и `NameFinderFeatureGenerator`, подготовленные в листинге 7.16, а затем комбинирует эти признаки с моделью, чтобы получить список возможных предсказаний, сопровождаемых вычисленными с помощью модели вероятностями. Каждое предсказание соответствует какой-то категории, и категория с наибольшей вероятностью назначается документу. В точке ❸ мы создаем объект `ClassifierResult`, передаваемый анализатору `ResultAnalyzer`. Обработав таким манером все документы, мы в точке ❹ печатаем сводку результатов.

Код, необходимый для интеграции классификатора `DocumentCategorizer` с производственной системой, не слишком отличается от описанного в этом разделе. Производственная система должна будет выполнить однократную настройку лексических анализаторов, генераторов признаков и классификатора точно так же, как в листинге 7.16, а затем заняться классификацией документов, как в листинге 7.17. Подумайте, как можно было бы адаптировать пример из раздела 7.4.7, взяв для интеграции с Solr не байесовский классификатор из проекта Mahout, а только что рассмотренный классификатор из проекта OpenNLP.

Представив целый ряд алгоритмов классификации документов по содержимому, давайте рассмотрим одно приложение таких алгоритмов – пометку содержимого – и то, как с помощью небольшого их изменения можно организовать содержимое. Для этого мы опишем механизм навигации по большому набору документов путем включения дополнительного тематического фасета в результаты поиска.

7.6. Построение рекомендателя меток с помощью Apache Solr

Прежде чем перейти к реализации автоматического разметчика содержимого, поговорим о том, как присвоение меток стало популярным механизмом поиска нужной информации и почему это так важно.

На заре Интернета сложилось два основных способа поиска информации. Поисковые системы индексировали веб и предоставляли простые интерфейсы поиска по ключевым словам. А сайты каталогов

классифицировали веб-страницы, составляя гигантские древовидные тематические рубрикаторы. Каждый подход был ориентирован на определенные способы поиска информации, у обоих были свои плюсы и минусы.

Индексы и пользовательские интерфейсы поисковых систем удовлетворяли потребности многих, трудности же возникали, когда пользователь не знал, как описать свои нужды с помощью ключевых слов. Поиск по ключевым словам осложнялся еще и тем, что многие концепции можно было описать разными термами, поэтому важная информация могла остаться незамеченной, если в документе и при поиске употреблялись неодинаковые слова. Таким образом, поиск по ключевым словам отчасти сводился к угадыванию правильных термов, а отчасти к внимательному просмотру результатов поиска, чтобы вытащить дополнительные термы, которые можно было бы включить в запрос.

Другие компании разрабатывали огромные рубрикаторы и помещали сайты в те или иные рубрики. Часто рубрикаторы бывали древовидными – чем дальше от корня, тем специфичнее становились уровни. От широких рубрик верхнего уровня, например «Искусство» или «Путешествия», ответвлялись такие узкоспециализированные, как «Театр Кабуки» или «Туризм в Токио». Такие схемы, получившие название *таксономия*, управлялись одной организацией и зачастую разрастались до такой степени, что конечный пользователь понять их уже не мог. По какой ветке идти, чтобы найти сайты, относящиеся к японской кухне? Из-за взрывного роста Интернета поддерживать актуальность стало невозможно. Каждый новый сайт классифицировался вручную людьми, знавшими таксономию вдоль и поперек. А пользователям нужно было быстро найти то, за чем они пришли, а не блуждать в дебрях рубрикатора неизвестно какой глубины.

Социальные метки, или закладки стали альтернативой большим, централизованно управляемым таксономиям. Вместо того чтобы укладывать содержимое в единственный иерархический рубрикатор, социальные закладки дают пользователям возможность самостоятельно организовывать содержимое. Например, если пользователь хочет запомнить какую-нибудь веб-страницу, он сопоставляет ей слова – метки – которые для него имеют смысл. Пользователи Твиттера вставляют в свои сообщения хэштеги, добавляя знак решетки (#) к ключевым словам, чтобы другие пользователи, интересующиеся той же темой, могли найти их сообщения.

В обоих случаях метки открыты для всех, и потому содержимое может найти любой человек, нужно только ввести определенную мет-

ку. Когда так ведут себя сотни и тысячи пользователей, постепенно начинает выкристаллизовываться некая схема, основанная на общем употреблении слов. Определение метки возникает из способа ее использования. Вместо того чтобы силой втискивать содержимое в прокрустово ложе жестко управляемой таксономии, социальные закладки улавливают общую точку зрения многих людей и таким образом определяют контекст информации.

Эту естественно образующуюся схему организации часто называют *фолксономией*, имея в виде социальный, создаваемый совместными усилиями многих людей (*коллаборативно*) вариант таксономии. При такой организации содержимое не обязано укладываться в строгую таксономию категорий, а представляется группой термов, каждый из которых можно рассматривать как полноценную категорию.

Поставить метку содержимого в веб несложно. Вы сами создаете или находите в Сети что-то интересное, нажимаете кнопку и ассоциируете с веб-страницей слова, которые помогут найти ее в будущем. По мере того как количество помеченных страниц растет, начинают проявляться любопытные вещи. Взгляните на облако меток на рис. 7.5, где показана совокупность всех меток, которые некий пользователь присвоил страницам на сайте delicious.com. Можете вы сходу сказать, что его интересует?

Рис. 7.5. Облако меток сайта Delicious.com, сгенерированное wordle.net. Размер шрифта отражает частоту встречаемости метки в наборе документов

Как доказывают сайты типа Delicious и Twitter, социальные метки пользуются плодами усилий миллионов людей, помечающих содержимое. Кто-то пометит страницу не теми словами, что наш взятый для примера пользователь, а выберет понятную себе фразеологию.

А кто-то воспользуется теми же словами, но для пометки других страниц. И хотя любой человек может пользоваться собственными метками для навигации по своему набору страниц, никто не мешает с помощью чужих меток искать и находить страницы на ту же тему или интересные по какой-то другой причине. Семантика меток зависит от того, как люди ими пользуются.

На рис. 7.6 мы видим рекомендованные метки. Откуда они взялись? Как были определены? Простой алгоритм рекомендования мог бы проанализировать метки, поставленные данным пользователем, и метки других пользователей, ассоциированные с теми же страницами, и на этой основе составить список. Есть и другие алгоритмы, которые рекомендуют метки, исходя из содержимого помечаемой страницы. Анализ текста помечаемой страницы и сравнение его со статистической моделью, основанной на метках других страниц с похожим содержимым, может дать очень точные рекомендации меток.

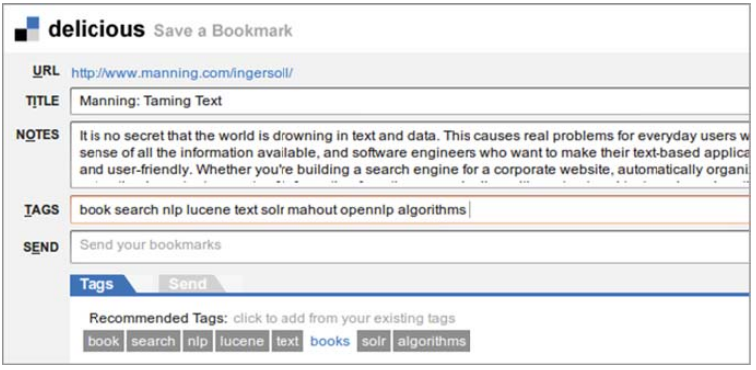

Рис. 7.6. Пометка веб-страницы на сайте delicious.com: пользователь сопоставляет странице несколько меток, с помощью которых впоследствии сможет ее найти. Метки описывают различные грани содержимого страницы

Любой из этих примеров можно развить и создать систему рекомендования меток с использованием Lucene. Подход, основанный на вычислении расстояния (см. раздел 7.3), предполагает, что каждый документ, или вектор термов помечен единственной категорией, и мы выбираем самую релевантную категорию из множества подходящих кандидатов. В примере байесовского классификатора мы видели, что зачастую существует много соответствующих документу категорий, которые могут быть как весьма общими, так и узкими и описывать различные грани (фасеты) его смысла.

А если не создавать ограниченный набор категорий, то можно ли воспользоваться уже имеющимися в наборе документов метками для генерации меток других документов?

Этим мы сейчас и займемся. Мы покажем, как с помощью алгоритма классификации по k-ближайшим соседям в сочетании с набором предварительно помеченных документов можно создать рекомендатель меток на базе Apache Solr. Как и в реализация алгоритма k-ближайших соседей в разделе 7.3, мы построим индекс по обучающим данным и воспользуемся запросом вида `MoreLikeThis` для сравнения документа с индексом. Рекомендуемые метки будут взяты из результатов поиска по этому запросу.

7.6.1. Подготовка обучающих данных для рекомендателя меток

Для построения рекомендателя меток мы воспользуемся набором данных с вопросно-ответного сайта Stack Overflow (http://www.stackoverflow.com) – «коллаборативно редактируемого вопросно-ответного сайта для программистов-профессионалов и любителей».

Сайт управляется компанией Stack Exchange, которая эксплуатирует подобные вопросно-ответные сайты, посвященные различным темам. Люди заходят на сайт, чтобы задать вопрос, дать ответ или оценить ответы, данные другими членами сообщества. Каждому вопросу его автор сопоставляет набор меток – ключевых слов. По состоянию на январь 2011 года набор данных, выгруженных с сайта Stack Overflow, содержал свыше 4 миллионов сообщений. Из них примерно миллион – это вопросы, сопровождаемые метками. Длина вопросов варьируется, но в большинстве из них есть текст и метки, полезные для обучения системы рекомендования, которую мы собираемся создать.

Помимо отличного источника обучающих данных, Stack Overflow и родственные ему сайты можно рассматривать как поучительный пример использования меток на узкоспециализированном сайте. На рис. 7.7 показана страница с описанием метки *java*. На ней приведены самые популярные запросы, снабженные этой меткой. Здесь же показана статистика, связанная с меткой, например, количество вопросов, снабженных меткой java, а также близкие метки со счетчиками. Иметь такое определение метки очень полезно. Оно предостерегает пользователей от использования ее в вопросах, касающихся Индонезии или сортов кофе.

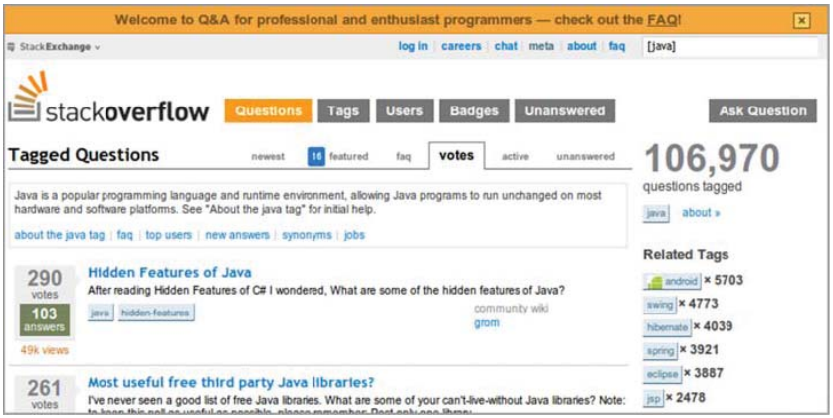

Рис. 7.7. Страница сайта stackoverflow.com с информацией о вопросах, помеченных словом java. На этом примере виден один из способов использования меток для навигации, а также в качестве полезного источника обучающих данных

Классификация и рекомендование

Термины *классификация* и *рекомендование* относятся к родственным семействам алгоритмов машинного обучения. Разница между ними в том, что классификация предлагает небольшое число возможных вариантов из контролируемого общего списка, тогда как результатом рекомендования является более обширное подмножество вариантов из потенциально бесконечного множества, например, каталога товаров или базы данных научных статей.

Вместе или вместо решений на основе содержимого многие программы решают, что рекомендовать, анализируя поведение пользователей, например, отслеживают, что вам нравится, а что не нравится или что вы покупаете или выбираете, и сравнивают ваше поведение с поведением других пользователей, пытаясь рекомендовать то, что и вам может понравиться. Такие компании, как Amazon и Netflix, применяют подобные алгоритмы, чтобы рекомендовать книги или фильмы.

В нашем примере рекомендатель меток будет основываться исключительно на анализе текста входной статьи и статей, уже имеющихся в системе. В этом смысле он больше походит на алгоритмы классификации и категоризации, а не на рекомендатели на основе поведения. Было бы интересно объединить оба подхода. Желающим узнать больше о рекомендателях, советуем книгу Owen, Anil, Dunning, Friedman «Mahout in Action», вышедшую в издательстве Manning (Owen [2010]).

Начнем с малого: определим обучающий набор, содержащий 100 000 сообщений, и тестовый набор на 1000 сообщений. С каждым

вопросом может быть связано несколько меток, поэтому текст вопроса будет использоваться как обучающие данные для ассоциированных с ним меток. Если с вопросом связаны метки *php*, *javascript* и *ajax*, то для каждой из них вопрос будет являться обучающим примером.

При оценке качества системы рекомендования на тестовых данных мы будем сравнивать метки, присвоенные сообщению системой и пользователями. Если хотя бы одна метка совпадает, то мы будем считать, что система сработала правильно.

Как и в предшествующих примерах из этой главы, для обучения рекомендателя меток мы проиндексируем обучающие документы с помощью Lucene. Мы воспользуемся Solr как платформой для быстрой загрузки и индексирования данных сайта Stack Overflow, применив программу Data Import Handler, а результаты рекомендателя будем предоставлять посредством веб-службы.

7.6.2. Подготовка обучающих данных

Для начала скачайте подмножество обучающих и тестовых данных Stack Overflow, которое мы выложили на сайте http://www.tamingtext.com. Если хотите, можете загрузить полный набор со страницы http://blog.stackoverflow.com/category/cc-wiki-dump/. Там имеется torrent-файл, который позволяет скачать данные с помощью любого клиента BitTorrent. В этом наборе имеются файлы для всех сайтов Stack Exchange, так что в клиенте укажите, что вас интересуют только данные Stack Overflow – файл Content\Stack Overflow 11-2010.7z в формате архива 7-Zip.

После распаковки архива появятся файлы badges.xml, comments.xml, posthistory.xml и posts.xml. Источником обучающих данных будет файл posts.xml, в котором нужные нам данные находятся внутри элемента `posts`. Мы вкратце опишем, как сгенерировать обучающий и тестовый наборы, а затем обсудим сам формат файла.

Для разбиения можно выполнить такую команду:

```
$TT_HOME/bin/tt extractStackOverflow \
 -i posts.xml \
 -te stackoverflow-test-posts.xml \
 -tr stackoverflow-training-posts.xml
```

По умолчанию `extractStackOverflow` берет первые 100 000 вопросов в качестве обучающих данных, а следующие 10 000 – в качестве тестовых. Можете поэкспериментировать – задайте больше или

меньше данных и посмотрите, как это отразится на времени и качестве обучения.

В файле posts.xml имеются элементы `row`. Нас из них интересуют только те, которые относятся к вопросам и снабжены метками. Каждый вопрос содержится в отдельном элементе `row`, но не каждый элемент `row` содержит вопрос. Утилита `extractStackOverflow` отфильтровывает все не относящиеся к вопросам элементы, глядя на атрибут `PostTypeId`. Если он равен 1, то это вопрос, все остальные элементы отбрасываются. Интерес представляют также атрибуты `Title`, `Tags` и `Body`. Их значения и будут нашими исходными обучающими данными. Кое-какие другие атрибуты тоже могут пригодиться, поэтому мы оставим и их.

Метки вопроса находятся в атрибуте `tags` элемента `row`, причем каждая метка заключена в угловые скобки. Так, для вопроса с тремя метками *javascript*, *c++* и *multithreaded programming* этот атрибут будет содержать строку `<javascript><c++><multithreaded programming>`. Для подготовки обучающих и тестовых данных нужно научиться разбирать этот формат.

7.6.3. Обучение рекомендателя меток на основе Solr

Мы обучим рекомендатель на данных сайта Stack Overflow с помощью обработчика импорта данных Solr. В коде, прилагаемом к книге, имеется пример конфигурации Solr для чтения файла с обучающими данными, извлечения необходимых полей из XML и преобразования меток в дискретные значения, сохраняемые в индексе. Представляющая интерес часть конфигурационного файла приведена ниже.

Листинг 7.18. Фрагмент файла dih-stackoverflow-config.xml

```
<entity name="post"
        processor="XPathEntityProcessor"
        forEach="/posts/row"
        url=".../stackoverflow-corpus/training-data.xml"
        transformer="DateFormatTransformer,HTMLStripTransformer,
         com.tamingtext.tagrecommender.StackOverflowTagTransformer">
    <field column="id" xpath="/posts/row/@Id"/>
    <field column="title" xpath="/posts/row/@Title"/>
    <field column="body" xpath="/posts/row/@Body"
           stripHTML="true"/>
    <field column="tags" xpath="/posts/row/@Tags"/>
```

Обработчик импорта данных пользуется процессором `XPathEntityProcessor` для выделения из обучающих данных отде-

льных документов Lucene, соответствующих каждому тегу <row> в объемлющем теге <posts>. Различные атрибуты тега row используются для заполнения полей индекса ID, title, body и tags. Из содержимого атрибута body удаляется вся HTML-разметка.

Класс StackOverflowTagTransformer – это простой специализированный преобразователь, который явно ищет атрибут tags и обрабатывает его, как описано выше. В результате получаются значения поля tag для каждого документа в индексе Solr. В следующем листинге приведен полный код этого класса.

Листинг 7.19. Преобразование данных в обработчике импорта Solr

```
public class StackOverflowTagTransformer {
 public Object transformRow(Map<String,Object> row) {
  List<String> tags = (List<String>) row.get("tags");
  if (tags != null) {
   Collection<String> outputTags =
            StackOverflowStream.parseTags(tags);
   row.put("tags", outputTags);
  }
  return row;
 }
}
```

Обработчик импорта передает данные по одной строке методу transformRow, а преобразователь вправе изменять содержимое столбцов разными способами. В данном случае столбец tags заменяется другим набором меток – результатом разбора исходного формата. Мы знаем, что значение, возвращенное вызовом row.get("tags"), можно рассматривать как List, потому что поле tags определено в схеме этого экземпляра Solr как многозначное.

Если вы заглянете в конфигурационный файл dih-stackoverflow-config.xml, то обнаружите, что обработчик импорта добавляет и ряд других полей из данных Stack Overflow. Поняв, как работает процесс индексирования, можно приступить к делу, т. е. запустить экземпляр Solr и загрузить в него документы. Для запуска сервера выполните такую команду:

```
$TT_HOME/bin/start-solr.sh solr-tagging
```

В процессе запуска Solr выводит на экран подробную информацию о загрузке всех компонентов и элементов конфигурации. Через пару секунд запуск должен завершиться сообщением:

```
Started SocketConnector@0.0.0.0:8983
```

Не закрывайте окно терминала, в которое Solr выводит журнал; оно еще может понадобиться для отладки проблем, возникающих при загрузке данных.

После того как Solr успешно запустится, нужно будет отредактировать конфигурацию обработчика импорта данных, указав файл, в котором будут храниться обучающие данные. Откройте файл $TT_HOME/apache-solr/solr-tagging/conf/dih-stackoverflow.properties и измените свойство URL с /path/to/stackoverflow-training-posts.xml на полный путь к файлу с обучающими данными в своей системе. Хотя свойство и называется `url`, его значением может быть путь к обычному файлу на диске. Сохраните файл dih-stackoverflow.properties и выйдите из редактора.

Зайдите в браузере на страницу http://localhost:8983/solr/admin/dataimport.jsp, щелкните по ссылке DIH-STACKOVERFLOW, и вы увидите консоль обработчика импорта данных. После запуска Solr вы изменяли конфигурацию обработчика, поэтому сейчас нужно нажать кнопку Reload-config в нижней части левого фрейма, чтобы ее перезагрузить. После этого можно приступать к загрузке обучающих данных. Нажмите кнопку Full-import, расположенную рядом с кнопкой Reload-config, – Solr начнет трудиться над загрузкой. Нажав кнопку Status, можно вывести текущее состояние в формате XML в правый фрейм. Спустя несколько минут появится такое сообщение:

```
Indexing completed. Added/Updated: 100000 documents.
Deleted 0 documents
(Индексирование завершено. Добавлено/обновлено: 100000 документов.
Удалено 0 документов)
```

Кроме того, на экране терминала, где открыт журнал Solr, появится сообщение:

```
Mar 11, 2011 8:52:39 PM org.apache.solr.update.processor.
LogUpdateProcessor
INFO: {add=[4, 6, 8, 9, 11, 13, 14, 16, ... (100000 adds)],optimize=} 05
```

Этап обучения завершен, и экземпляр Solr готов рекомендовать метки.

7.6.4. Создание рекомендаций меток

Экземпляр `solr-tagging` был настроен на использование обработчика Solr `MoreLikeThisHandler` для ответов на запросы. Как и рассмотренный выше классификатор `MoreLikeThis`, этот обработчик принимает на входе документ и с помощью индекса определя-

ет, какие термы запроса наиболее полезны для извлечения похожих документов. В следующем листинге показана конфигурация `MoreLikeThisHandler` в файле solrconfig.xml.

Листинг 7.20. Конфигурация MoreLikeThisHandler в файле solrconfig.xml

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
 <lst name="defaults">
   <str name="mlt.fl">title,body</str>
   <int name="mlt.mindf">3</int>
 </lst>
</requestHandler>
```

Принятый нами подход к рекомендованию меток аналогичен алгоритму классификации k-NN, описанному в разделе 7.3.6. Но подсчитываем мы не категории, назначенные документам, которые возвращены в ответ на запрос `MoreLikeThisRequest`, а метки, которые затем используются как рекомендации. Класс `TagRecommenderClient` отвечает за доставку входного документа Solr и постобработку результатов с целью агрегирования, оценки и ранжирования меток. В листинге ниже показано, как это выглядит на верхнем уровне.

Листинг 7.21. Применение TagRecommenderClient для генерации рекомендуемых меток

```
public TagRecommenderClient(String solrUrl)
        throws MalformedURLException {
 server = new HttpSolrServer(solrUrl);      <-- (1) Клиент Solr
}

public ScoreTag[] getTags(String content, int maxTags)
        throws SolrServerException {
 ModifiableSolrParams query = new ModifiableSolrParams();
 query.set("fq", "postTypeId:1")                    Параметры запроса (2)
   .set("start", 0)
   .set("rows", 10)
   .set("fl", "*,score")
   .set("mlt.interestingTerms", "details");

 MoreLikeThisRequest request      <-- (3) Создаем и выполняем запрос
   = new MoreLikeThisRequest(query, content);
 QueryResponse response = request.process(server);

 SolrDocumentList documents = response.getResults();
 ScoreTag[] rankedTags = rankTags(documents, maxTags);      (4)
 return rankedTags;                                 Собираем и ранжируем метки
}
```

Прежде всего необходимо установить соединение с Solr. В точке ❶ мы создаем экземпляр класса `HttpSolrServer`, который вопреки своему названию является клиентом Solr, использующим клиентскую библиотеку HTTP для отправки запросов серверу Solr. Конструктору в качестве параметра передается URL сервера.

После создания клиента мы строим запрос для получения документов, соответствующих нашему запросу. Метки из этих документов будут использоваться для выдачи рекомендаций. В данных с сайта Stack Overflow метками снабжаются только вопросы, поэтому, задавая параметры запроса в точке ❷, мы указываем фильтр, который оставляет только сообщения с `postTypeId`, равным 1. Отметим также, что запрашиваются первые 10 документов; имеет смысл поэкспериментировать с другими значениями, чтобы понять, при каком получаются наилучшие рекомендации.

Для отправки запроса Solr в точке ❸ используется написанный нами класс `MoreLikeThisRequest` вместо стандартного `QueryRequest`. Объект `MoreLikeThisRequest` отправляет потенциально длинный документ-запрос методом HTTP POST непосредственно обработчику Solr `/mlt`. Параметр конструктора `content` содержит текст, для которого мы хотели бы получить рекомендованные метки.

Получив от сервера результаты, мы должны извлечь и проранжировать метки и дать рекомендации. В точке ❹ мы отбираем для каждой метки счетчики и ранжируем метки. Класс `ScoreTag` используется для хранения полученных меток. Вместе с меткой хранится счетчик ее употреблений и оценка. Подробности видны из следующего листинга.

Листинг 7.22. Сбор и ранжирование меток

```
protected ScoreTag[] rankTags(SolrDocumentList documents,
                              int maxTags) {
 OpenObjectIntHashMap<String> counts =
     new OpenObjectIntHashMap<String>();
 int size = documents.size();          <-- ❶ Подсчитываем метки
 for (int i=0; i < size; i++) {
  Collection<Object> tags = documents.get(i).getFieldValues("tags");
  for (Object o: tags) {
   counts.adjustOrPutValue(o.toString(), 1, 1);
  }
 }                                                  Ранжируем метки ❷

 maxTags = maxTags > counts.size() ? counts.size() : maxTags;
 final ScoreTagQueue pq = new ScoreTagQueue(maxTags);
```

```
counts.forEachPair(new ObjectIntProcedure<String> () {
  @Override
  public boolean apply(String first, int second) {
    pq.insertWithOverflow(new ScoreTag(first, second));
    return true;
  }
});
ScoreTag[] rankedTags = new ScoreTag[maxTags];
int index = maxTags;
ScoreTag s;
int m = 0;
while (pq.size() > 0) {
  s = pq.pop();
  rankedTags[--index] = s;
  m += s.count;
}
for (ScoreTag t: rankedTags) {
  t.setScore(t.getCount() / (double) m);
}
return rankedTags;
}
```

❸ Собираем ранжированные метки

❹ Оцениваем метки

Перед ранжированием и оценкой меток нужно подсчитать, сколько раз каждая встречается. В точке ❶ мы перебираем все документы в наборе результатов, извлекаем метки из поля tags и вычисляем для каждой из них счетчик употреблений.

Имея счетчики для всех меток, можно проранжировать метки по значениям счетчиков. В точке ❷ мы помещаем метки в приоритетную очередь, в которой они хранятся в порядке убывания частоты употребления. Затем в точке ❸ мы извлекаем результаты из очереди и в точке ❹ вычисляем оценки меток. Оценка метки равна количеству ее появлений в наборе результатов, поделенному на общее число меток, встречающихся в этом наборе после отсечения редких. Таким образом, оценка лежит в диапазоне от 0 до 1, причем чем ближе она к 1, тем важнее метка в наборе результатов. Если оценки меток высоки, то их общее число сравнительно мало. И наоборот, если оценки низкие, значит, в наборе результатов встречается много разных меток. Это можно было бы использовать в качестве меры достоверности. Путем экспериментов можно подобрать такую пороговую оценку, что метки с более низкой оценкой не включаются в состав рекомендованных.

7.6.5. Оценивание рекомендателя меток

Процедура оценивания рекомендателя меток не сильно отличается от того, что мы делали ранее для классификатора. Часть данных Stack

Overflow, которая не использовалась для обучения, теперь становится тестовым набором. Для каждого вопроса из этого набора мы получаем рекомендованные метки и сравниваем их с теми, которые были присвоены вручную. Существенное различие заключается в том, что каждый обучающий документ сопровождается несколькими метками, и результат обработки запроса также содержит несколько меток. Поэтому при тестировании будут вычисляться два показателя. Первый – количество тестовых документов, для которых хотя бы одна метка рекомендована правильно. Процентная доля документов хотя бы с одной правильной меткой позволит понять, корректно ли работает классификатор. Второй показатель – количество случаев, когда правильно рекомендовано не менее 50 % меток. Например, если тестовый документ сопровождался четырьмя метками, а в наборе, вычисленном рекомендателем, присутствуют хотя бы две из них, то результат считается правильным. Это дает представление о том, насколько падает точность при предъявлении более строгих требований.

Помимо этих показателей, мы вычислим процент правильности для подмножества всех меток, встречающихся в обучающем и тестовом наборах. Мы найдем, какие метки встречаются в тестовом наборе чаще всего, и независимо вычислим для них точные показатели.

Посмотрим, как это делается. Во-первых, нужно извлечь данные Stack Overflow из XML-файлов, в которых они хранятся. Для этого нам понадобится класс `StackOverflowStream`. В нем для разбора документов применяется StAX API, в результате чего порождаются объекты `StackOverflowPost`, содержащие поля из каждого сообщения, в том числе и представляющие для нас интерес: title (заголовок), body (тело) и tags (метки). Большая часть кода класса `StackOverflowStream` – типичный разбор XML и обход коллекции сообщений, поэтому не будем его здесь приводить, а отошлем читателя к исходному коду, прилагаемому к книге.

Чтобы вычислить показатели отдельных меток, необходимо для начала отобрать их из данных Stack Overflow. Следующая команда выбирает 25 меток, встречающихся чаще всего в тестовых данных, отсекая те, что встречаются менее чем в 10 сообщениях:

```
$TT_HOME/bin/tt countStackOverflow \
 --inputFile stackoverflow-test-posts.xml \
 --outputFile stackoverflow-test-posts-counts.txt \
 --limit 25 --cutoff 10
```

В результате получается текстовый файл с тремя столбцами: ранг, счетчик и метка. Ниже приведен фрагмент этого файла, в котором наиболее популярной оказалась метка *c#*, встречающаяся в 1480 сообщениях. За ней следует *.net* (858 сообщений), *asp.net* (715 сообщений) и *java* (676 сообщений):

```
1    1480    c#
2    858     .net
3    715     asp.net
4    676     java
```

Полученные счетчики категорий можно подать на вход процесса тестирования. Следующая команда выполняет код класса `TestStackOverflow`. Он читает тестовый файл, извлекает нужные поля из XML-документа в формате Stack Overflow, с помощью `TagRecommenderClient` запрашивает набор меток у сервера Solr, а затем сравнивает метки, присутствующие в тестовых данных, с теми, что вычислил рекомендатель. В процессе работы вычисляются показатели, характеризующие качество рекомендателя.

```
$TT_HOME/bin/tt testStackOverflow \
 --inputFile stackoverflow-test-posts.xml \
 --countFile stackoverflow-test-posts-counts.txt \
 --outputFile stackoverflow-test-output.txt \
 --solrUrl http://localhost:8983/solr
```

Программа `testStackOverflow` печатает сведения после обработки каждых 100 сообщений. Это позволяет следить за тем, насколько хорошо рекомендатель присваивает метки. Ниже показано, что было напечатано после обработки 300 и 400 тестовых документов[10].

```
обработано 300 сообщений; одна метка правильна: 234, половина меток правильна: 151
   %одна правильна: 78, %половина правильна: 50.33
обработано 400 сообщений; одна метка правильна: 311, половина меток правильна: 204
   %одна правильна: 77.75, %половина правильна: 51
```

Здесь от 77 до 78 процентам документов была присвоена хотя бы одна правильная метка, и примерно в 50 % случаев правильны были не менее половины меток. Отметим, что с увеличением количества обработанных документов эти показатели стабилизируются. Это позволяет сделать вывод, что проводить тестирование более чем на 10 000 документов (хотя в Stack Overflow их гораздо больше) бессмысленно.

---

[10] Для удобства читателя сообщения переведены, на самом деле они печатаются по-английски. – *Прим. перев.*

По завершении работы `testStackOverflow` выводит еще показатели для отдельных меток в указанный файл. Это, с одной стороны, окончательные процентные доли документов, для которых правильно рекомендованы не менее одной и не менее половины меток, а, с другой стороны, точные данные о правильности для всех меток, представленных в файле счетчиков.

```
обработано 10000 сообщений; одна метка правильна: 8033, половина меток правильна: 5136
    %одна правильна: 80.33, %половина правильна: 53.16
-- метка       всего правильно %-правильных --
networking     48     12        25
nhibernate     70     48        68
visual-studio 152     84        55
deployment     48     19        39
```

Для каждой метки печатается, в скольких тестовых сообщениях она встретилась и сколько раз была рекомендована для соответствующего сообщения. Процентная доля правильных рекомендаций характеризует качество работы рекомендателя для данной конкретной метки. В примере выше рекомендатель плохо проявил себя на метке *networking*, но справился с меткой *nhibernate*. Экспериментируя с обучающими данными, можно проследить, как будут меняться эти величины для отдельных меток.

Нет никаких причин ограничиваться *X* наиболее часто встречающихся метками. Если вам интересно, как рекомендатель отработал на других метках, вручную включите их в файл счетчиков. Если эти метки встречаются в тестовом наборе, то в выходном файле появятся показатели для них.

7.7. Резюме

В этой главе мы изучили, как алгоритмы классификации применяются для автоматической категоризации и пометки текстовых документов. Попутно мы обсудили процесс создания автоматизированных классификаторов: подготовку входных данных, обучение для генерации модели, тестирование с целью оценки качества работы и способы развертывания в производственной системе. Мы поняли, насколько важно выбирать подходящую схему классификации и правильные признаки для обучения алгоритма, и рассмотрели методы получения тестовых данных: открытые ресурсы, бутстрапинг с использованием имеющегося набора категорий, экспертные оценки. Мы исследовали несколько оценочных показателей и обсудили применение верности, точности, полноты и матрицы неточностей для получения различ-

ных представлений о работе алгоритма классификации. Мы видели разные способы модификации параметров алгоритма для улучшения качества классификации и продемонстрировали, как все рассмотренные классификаторы интегрируются с Solr для предоставления службы категоризации в производственной системе.

В этой главе вы познакомились с основными идеями классификации и категоризации текстов и теперь можете самостоятельно изучать другие алгоритмы. Изучая имеющиеся исследовательские работы и программное обеспечение, вы обнаружите, что существует много подходов к решению задач классификации. Но имея представление о самых простых – по k-ближайшим соседям, наивная байесовская классификации и алгоритм максимальной энтропии – вы располагаете достаточными знаниями для исследования других способов классификации, категории и пометки. Применение того или иного алгоритма зависит от требований приложения. Упомянем еще два, которые стоит иметь в виду.

В проекте Mahout также имеется алгоритм логистической регрессии, в основу обучения которого положен метод стохастического градиентного спуска. В общем и целом, он напоминает классификатор на базе логистической регрессии из проекта OpenNLP, но интересен наличием разнообразных способов интерпретации признаков; предоставляются механизмы включения в единую модель признаков вида «похоже на число», «похоже на слово» и «похоже на текст». Тэд Даннинг (Ted Dunning), руководивший реализацией этого классификатора в Mahout, подробно описал его в книге «Mahout in Action» (2011), вышедшей в издательстве Manning.

Для классификации текстов также широко применяются машины опорных векторов (SVM). Существует немало исследований, посвященных различным подходам к моделированию текста с помощью SVM, и несколько реализаций таких систем классификации с открытым исходным кодом. В качестве примеров назовем SVMLIGHT (http://svmlight.joachims.org) Торстена Иоахимса (Thorsten Joachims) и LIBSVM (http://www.csie.ntu.edu.tw/~cjlin/libsvm) Чжи Чун Чана (Chih-Chung Chang) и Чжи Цзен Линя (Chih-Jen Lin). Обе эти библиотеки позволяют реализовать систему классификации текстов на разных языках.

Существуют бесчисленные вариации и комбинации этих и других методов. Кое-какие альтернативы мы обсудим далее в главе 9 «Неприрученный текст».

7.8. Ресурсы

Lewis, David; Yang, Yiming; Rose, Tony; Li, Fan. 2004. «RCV1: A New Benchmark Collection for Text Categorization Research». *Journal of Machine Learning Research*, 5:361-397. http://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf.

Owen, Sean; Anil, Robin; Dunning, Ted; Friedman, Ellen. 2010. *Mahout in Action*. Manning Publications.

Rennie, Jason; Shih, Lawrence; Teevan, Jaime; Karger, David. 2003. «Tackling the Poor Assumptions of Naive Bayes Text Classifiers». http://www.stanford.edu/ class/cs276/handouts/rennie.icml03.pdf.

ГЛАВА 8. Пример вопросно-ответной системы

В этой главе:

- Применение методов автоматической пометки документов.
- Использование меток документов и поддокументов в поиске.
- Пересчет рангов возвращенных Solr документов с учетом дополнительных критериев.
- Генерация возможных ответов на вопросы пользователей.

В предыдущих главах мы рассматривали различные технологии и подходы вне связи друг с другом. И хотя нам удавалось создавать полезные приложения с применением одной-двух технологий, чаще для выполнения поставленной задачи приходится сочетать несколько описанных выше инструментов. Например, поиск удачно сочетается с определением фасетных категорий (классификацией), равно как и с кластеризацией – когда нужно помочь пользователям в поиске новых материалов, имеющих отношение к их информационным потребностям. В этой главе мы построим вопросно-ответную систему, способную отвечать на вопросы, основанные на фактах и записанные на английском языке. Для этого будет использоваться поиск, распознавание именованных сущностей, сравнение строк и другие методы. И если другие главы мало зависят друг от друга, то в этой предполагается, что вы прочли все написанное выше, поэтому мы не будем объяснять основы Solr и других систем.

Прежде чем приступать к делу, оглянемся на пройденный путь. Изученный материал составляет концептуальную основу того, что мы собираемся сделать. В главе 1 мы обсудили важность текста для различных приложений и познакомились с основной терминологией, относящейся к поиску и обработке естественных языков, а также с проблемами, возникающими при построении таких систем. Многое из сказанного там, явно или неявно, будет использовано в этой главе, пусть даже мы не акцентируем на этом внимание.

В главе 2 мы занялись основами обработки текстов, включая такие вещи, как части речи, лексический и грамматический разбор. Возможно, это напомнило вам школьные уроки. Мы также уделили время вопросу о применении Apache Tika для извлечения содержимого из файлов разных форматов и преобразования в нужный формат. И хотя в этом примере мы не будем использовать Tika явно, предобработку информации для приведения ее к требуемому виду выполнить все же придется. Мы также будем широко пользоваться средствами лексического и грамматического анализа и частеречной разметки содержимого – это поможет нам отвечать на заданные вопросы.

В главе 3 мы познакомились с поиском и системой Apache Solr – мощной поисковой платформой, которая позволяет легко и быстро индексировать текст и находить документы в ответ на запросы. Сейчас мы вновь воспользуемся Solr как основой вопросно-ответной системы, а заодно рассмотрим некоторые продвинутые возможности Apache Lucene.

Глава 4 была посвящена неточному сравнению строк – технологии, полезной во многих повседневных приложениях обработки текста. В этой главе мы применим то, чему научились, для автоматического исправления орфографических ошибок. Пригодится нам и метод *n*-грамм. Некоторые приемы работы со строками находят применение на нижнем уровне Lucene, и мы легко могли бы встроить в нашу систему компонент проверки орфографии, но решили этого не делать.

В главе 5 мы использовали OpenNLP для обнаружения и классификации имен собственных. Здесь OpenNLP нам снова пригодится – для решения этой задачи, а также для выделения словосочетаний. Это полезно как для анализа вопроса, так и при обработке данных, на основе которых мы формируем ответы.

В главе 6 мы окунулись в мир кластеризации и показали, как можно автоматически группировать похожие документы без участия человека. И хотя в этой главе методам кластеризации не нашлось места, их можно было бы применить для сужения пространства поиска при

поиске ответов и для обнаружения «почти дубликатов» в самих результатах.

Наконец, в главе 7 мы показали, как классифицировать текст и автоматически ассоциировать ключевое слово или фолксономическую метку с новым текстом. Это пригодится нам для отнесения вопросов к той или иной категории.

Теперь, освежив в памяти все, что мы узнали, соберем это воедино и построим настоящее приложение. Наша цель – получить демонстрационную вопросно-ответную систему, на примере которой можно проиллюстрировать, как сочетаются рассмотренные выше детали. Эта система будет отвечать на фактографические вопросы, используя в качестве базы знаний википедию. Для решения этой задачи мы будем использовать Solr в качестве базовой системы не только из-за ее умения производить поиск, но и потому, что ее компонентная архитектура допускает расширение с минимумом усилий. На базовый каркас можно «навесить» средства анализа на этапе индексирования, а на этапе поиска – средства разбора запросов на естественном языке, ранжирования ответов и возврата результатов. Начнем с более пристального рассмотрения самой вопросно-ответной (QA) системы и некоторых ее применений.

8.1. Основы вопросно-ответной системы

Как следует из названия, QA-система предназначена для ответа на вопросы, заданные на естественном языке, например, «Кто является президентом США?». Такие системы позволяют пользователю не просматривать многочисленные страницы результатов поиска и даже не продираться сквозь бурелом фасетов. Например, система Watson DeepQA производства IBM (http://www.ibm.com/innovation/us/watson/) с помощью изощренной вопросно-ответной системы противостояла человеку в игре *Jeopardy!*[1] (http://www.jeopardy.com). И знаете – выиграла у двух величайших мастеров *Jeopardy!* всех времен! В этой системе для обработки ответов (напомним, в *Jeopardy!* ответ должен в форме вопроса[2]) использовалось очень много компьютеров,

[1] Русский аналог называется «Своя игра». – *Прим. перев.*

[2] Особенность официальной версии игры в том, что вопросы звучат в утвердительной форме, а ответы игроков даются в вопросительной форме. Например, вопрос: Она рассказывала сказки в книге «Тысяча и одна ночь». Ответ: Кто такая Шехерезада? (См. http://ru.wikipedia.org/wiki/Jeopardy!). – *Прим. перев.*

к услугам которых был огромный массив мировых знаний, а также вспомогательные системы для выработки игровой стратегии (выбор вопросов, ставки и т. д.; см. рис. 8.1).

Рис. 8.1. Аватар системы Watson, который отображался на экране во время участия в игре Jeopardy! со стороны IBM

Отметим, что автоматизированную QA-систему не следует путать с популярными нынче системами задавания вопросов сообществу типа Yahoo! Answers, ChaCha или Ответы@mail.ru, пусть даже некоторые применяемые в таких системах технологии (идентификация похожих вопросов, к примеру) полезны и при построении автоматизированных систем. Во многих отношениях вопросно-ответная система напоминает поиск: вы вводите запрос, обычно состоящий из нескольких ключевых слов, и просматриваете документы или страницы, возвращенные в ответ. Но в QA-системе обычно запросом является полное предложение, а не отдельные слова. В качестве награды за большую точность вы ожидаете получить текст значительно короче полного документа. Вообще говоря, поиск ответа на вопрос – трудная задача, но в некоторых конкретных приложениях, или жанрах она может быть решена эффективно. Ответы на многие вопросы ответы сложны и требуют глубокого понимания существа вопроса. Мы не станем задирать планку так высоко, и все-таки наша система будет работать лучше, чем стандартный поиск, в случае вопросов вида «Кто является президентом США?».

Полноценная QA-система может пытаться найти ответы на вопросы разных типов, от фактографических до более абстрактных. Не следует также забывать, что QA-система вполне может возвращать несколько абзацев или даже документов в качестве ответа, хотя в большинстве своем системы стремятся представить куда более короткие ответы. Например, особо изощренная система (которой, насколько известно авторам, пока не существует) могла бы отвечать на вопросы, требующие углубленного анализа, например: «Каковы плюсы и минусы современной кубковой системы розыгрыша чемпионата по футболу среди колледжей?» или «Каковы краткосрочные и долгосрочные последствия неумеренного потребления алкоголя?».

IBM Watson: не только Jeopardy!

Система IBM Watson была продемонстрирована в игре Jeopardy!, чтобы привлечь внимание к проблеме, но, очевидно, ее назначение заключается не в том, чтобы состязаться в игре, а в том, чтобы помочь людям быстрее и с меньшими затратами осуществлять поиск информации. Процитируем сайт IBM:

Технология DeepQA дает в руки человека мощный инструмент для сбора информации и поддержки принятия решений. Типичный сценарий: пользователь вводит запрос на естественном языке, как если бы спрашивал другого человека, а система просматривает большой объем фактической информации и возвращает ранжированный список самых точных и аргументированных ответов. К каждому ответу прилагается обоснование и свидетельства в пользу его выбора, что позволяет пользователю быстро оценить эти свидетельства и выбрать правильный ответ.

Сложность системы не позволяет подробно рассмотреть ее в этой книге, но интересующимся читателям предлагаем заглянуть на сайт проекта IBM DeepQA (http://www.research.ibm.com/deepqa/deepqa.shtml), где приведена дополнительная информация.

Если копнуть глубже, то фактографическую вопросно-ответную систему можно рассматривать как задачу неточного поиска на уровне слов и словосочетаний. А коли так, то принятый нами подход напоминает стратегию, примененную в задаче сопоставления записей с дополнительными осложнениями, касающимися определения того, каким должен быть *тип ответа* на заданный вопрос? Например, спрашивая «Кто является президентом США?», пользователь ожидает увидеть в ответ имя человека, а в ответ на вопрос «В каком году Каролина Харрикейнс выиграла Кубок Стэнли?» получить год. Но прежде чем вникать в детали реализации системы, давайте настроим программу, чтобы вы могли предметно следить за происходящим.

8.2. Установка и запуск QA-системы

Мы уже говорили, что в основу нашей системы будет положен сервер Solr, поэтому для ее установки и эксплуатации необходимо предварительно развернуть Solr примерно так, как было описано в главе о кластеризации. Но в данном случае мы сконфигурируем Solr по-другому. Если вы еще этого не сделали, выполните инструкции в файле README на сайте GitHub (https://github.com/tamingtext/book/blob/master/README). Затем выполните команду `./bin/start-`

`solr.sh solr-qa`, находясь в каталоге TT_HOME. Если все пройдет гладко, то, зайдя в браузере по адресу http://localhost:8983/solr/answe, вы увидите простой интерфейс QA-системы. Успешно запустив систему, давайте загрузим данные, чтобы она могла отвечать на вопросы.

QA-системы, которые построены поверх поисковых систем (а так устроено большинство), как легко догадаться, используют хранящееся в поисковой системе содержимое, как источник информации для выработки ответов, поскольку у самой QA-системы нет встроенных знаний обо всех вопросах и ответах. С этим требованием сопряжен и факт, от которого никуда не деться: QA-система не может быть лучше содержимого, являющегося для нее источником данных. Например, если загрузить в систему документы, написанные в Европе в доколумбову эпоху (они ведь все оцифрованы, правда?), и спросить «Какой формы Земля?», то система, скорее всего, ответит «плоская». Для своей системы мы взяли данные, выгруженные из англоязычной части википедии по состоянию на 11 октября 2010 года (первые 100 000 документов можно скачать по адресу http://maven.tamingtext.com/freebase-wex-2011-01-18-articles-first100k.tsv.gz; размер архива составляет 411 МБ). Файл, конечно, большой, но это необходимо, потому что мы намерены продемонстрировать систему на реальных данных). Скачав архив, распакуйте его с помощью программы gunzip или эквивалентной. Если этот файл кажется вам слишком большим и вы хотели бы для начала попробовать что-нибудь поменьше, можете скачать файл http://maven.tamingtext.com/freebase-wex-2011-01-18-articles-first10k.tsv, в котором собраны первые 10 000 статей из большего файла. Этот файл не сжат, так что распаковывать его нет нужды.

Полученные данные следует проиндексировать, выполнив следующие команды:

- `cd $TT_HOME/bin`;
- `indexWikipedia.sh --wikiFile <PATH TO WIKI FILE>` (В ОС *NIX) или `indexWikipedia.cmd --wikiFile <PATH TO WIKI FILE>` (в Windows). На это уйдет некоторое время. Указав флаг `--help`, вы сможете узнать обо всех режимах индексирования.

После того как индекс построится, можно приступать к исследованию QA-системы. Для начала перейдите в браузере по адресу http://localhost:8983/solr/answer – на страницу простого поль-

зовательского интерфейса на основе встроенного в Solr класса `VelocityResponseWriter`, который получает от Solr результаты и применяет к ним шаблон Apache Velocity (см. http://velocity.apache.org) (Velocity – это система шаблонов, используемая в основном для создания сайтов на Java.) Если не возникнет ошибок, то вы увидите страницу, показанную на рис. 8.2.

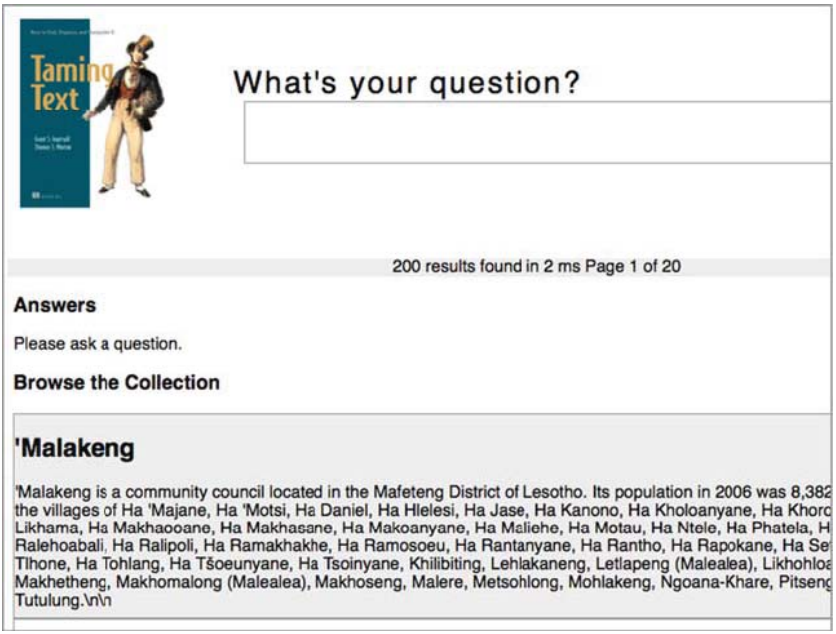

Рис. 8.2. Вид пользовательского интерфейса фактографической системы Taming Text

В предположении, что все сработало правильно, перейдем к рассмотрению архитектуры и кода системы.

8.3. Архитектура демонстрационной вопросно-ответной системы

Как и рассмотренная выше поисковая система, наша QA-система должна уметь индексировать содержимое, а также производить поиск и постобработку результатов. Что касается индексирования, то

наш дополнительный код связан главным образом с процессом анализа. Мы написали два подключаемых к Solr модуля анализа: для распознавания предложений и для выявления именованных сущностей. Тот и другой основаны на OpenNLP. В отличие от большинства имеющихся в Solr средств анализа, мы решили разбивать текст сразу на предложения, потому что тогда мы сможем отправлять эти предложения непосредственно фильтру лексем, отвечающему за выделение именованных сущностей, и заодно избежать двойного лексического анализа: на уровне Solr и на уровне OpenNLP. Поскольку распознавание предложений и именованных сущностей уже были описаны выше, то здесь мы просто сошлемся на соответствующие классы (`SentenceTokenizer.java` и `NameFilter.java`) и приведем объявление типа текстового поля в файле schema.xml (он находится в каталоге solr-qa/conf, а показанная здесь версия для краткости отредактирована). Обратите внимание, что мы нарушили обычное правило производить одинаковый анализ на этапах индексирования и поиска, потому что считаем запрос одним предложением и не хотим разбивать его на предложения. Важно то, что лексемы на выходе эквивалентны по форме (например, подвергнуты одинаковому стеммингу), а не то, как эта форма была получена. Вот объявление типа поля:

```
<fieldType name="text" class="solr.TextField" positionIncrementGap="100"
           autoGeneratePhraseQueries="true">
 <analyzer type="index">
   <tokenizer
    class="com.tamingtext.texttamer.solr.SentenceTokenizerFactory"/>
   <filter class="com.tamingtext.texttamer.solr.NameFilterFactory"/>
   <filter class="solr.LowerCaseFilterFactory"/>
   <filter class="solr.StopFilterFactory"
           ignoreCase="true"
           words="stopwords.txt"
           enablePositionIncrements="true"
   />
   <filter class="solr.PorterStemFilterFactory"/>
 </analyzer>
 <analyzer type="query">
   <tokenizer class="solr.WhitespaceTokenizerFactory"/>
   <filter class="solr.StopFilterFactory"
           ignoreCase="true"
           words="stopwords.txt"
           enablePositionIncrements="true"
   />
   <filter class="solr.WordDelimiterFilterFactory"
           generateWordParts="1" generateNumberParts="1"
```

```
            catenateWords="0" catenateNumbers="0" catenateAll="0"
            splitOnCaseChange="1"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
 </fieldType>
```

Мы опустим детали процесса анализа, поскольку они рассматривались в предыдущих главах, но все же отметим, что `NameFilterFactory` выводит как исходные лексемы, так и лексемы, соответствующие именованным сущностям. Последние расположены точно в тех же позициях, что и исходные. Например, если ввести предложение «Clint Eastwood plays a cowboy in *The Good, the Bad and the Ugly*»[3] на входящей в состав Solr странице analysis.jsp (http://localhost:8983/solr/admin/analysis.jsp), то получится четыре лексемы, занимающие первые две позиции (по две в каждой) – см. рис. 8.3.

com.tamingtext.texttamer.solr.NameFilterFactory {luceneM		
position	1	2
term text	NE_person	NE_person
	Clint	Eastwood
keyword	true	true
	true	true
startOffset	0	6
	0	6
endOffset	5	14
	5	14

Рис. 8.3. Пример, когда лексемы именованных сущностей и исходные лексемы занимают одни и те же позиции. Разбиралось предложение «Clint Eastwood plays a cowboy in The Good, the Bad and the Ugly»

На этапе поиска компонентов больше, и нам пришлось написать больше кода, который мы подробно рассмотрим ниже в этой главе. Но сама работа системы основана на двух ключевых функциях:

- разбор заданного вопроса с целью определить ожидаемый тип ответа и сгенерировать подходящий запрос;
- ранжирование документов, возвращенных в ответ на сгенерированный запрос.

На рис. 8.4 показана общая архитектура нашей системы, изображены обе стороны – индексирование и поиск.

Как уже было сказано и как видно из рис. 8.4, индексирование не представляет особых сложностей. А обработка запроса состоит из пяти шагов, которые будут рассмотрены в следующем разделе.

[3] Клинт Иствуд сыграл ковбоя в фильме *Хороший, плохой, злой*. – *Прим. перев.*

- Разбор вопроса пользователя.
- Определение типа ответа, чтобы затем искать кандидатов, наиболее точно отвечающих на вопрос.
- Генерация запроса к Solr/Lucene с учетом разобранного вопроса и вычисленного типа ответа.
- Выполнение запроса для поиска фрагментов-кандидатов, которые *могли бы* содержать ответ.
- Ранжирование ответов и возврат результатов.

Чтобы правильно понять смысл этих шагов в контексте, мы разобьем изложение на две части: установление смысла вопроса и ранжирование фрагментов-кандидатов. Эту логику мы распределили между двумя подключаемыми к Solr компонентами с четко определенным интерфейсом:

- `QParser` (и `QParserPlugin`) – обрабатывает заданный пользователем вопрос и создает запрос к Solr/Lucene query;
- `SearchComponent` – включается в цепочку других компонентов `SearchComponent` для порождения ответа. Описание принципов см. в главе 3.

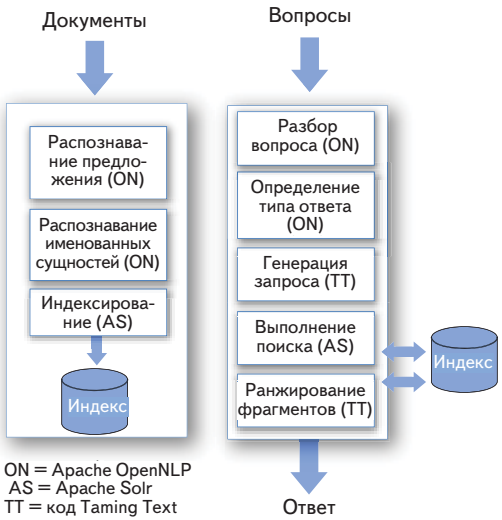

Рис. 8.4. Архитектура демонстрационной вопросно-ответной системы на базе Apache Solr, OpenNLP и нашего кода для ранжирования фрагментов текста

Ниже мы рассмотрим код более подробно, а пока советуем вернуться к главе о поиске и Solr (глава 3), а также почитать в документации по Solr, как эти компоненты взаимодействуют между собой. В предположении, что эти механизмы вам понятны, посмотрим, как устанавливается смысл вопроса пользователя.

8.4. Установление смысла вопроса и порождение ответов

Процедура установления смысла заданного вопроса и порождения результатов состоит из трех частей: определение типа ответа (TO), использование TO для генерации полезного запроса к поисковой системе и ранжирование полученных результатов. При этом важнейшую роль играет определение типа ответа, а последующая генерация запроса и ранжирование фрагментов производятся уже сравнительно просто. В нашем случае определение типа ответа включает три действия: обучение, разбиение на блоки и собственно определение TO. Все они описаны ниже, как и наш подход к генерации запроса при известном TO и к ранжированию фрагментов. Отметим также, что мы предполагаем, что пользователь вводит вопрос на естественном языке, например «Who was the Super Bowl MVP?»[4], а не в булевой форме, описанной в главе 3. Это важно, потому что, обучая систему классификации определять по заданному вопросу тип ожидаемого ответа, мы в качестве обучающих данных использовали вопросы на естественном языке.

8.4.1. Обучение классификатора типов ответов

Обучающие данные для этой системы (находятся в каталоге dist/data) состоят из 1888 вопросов, которые были вручную размечены Томом Мортоном в его кандидатской диссертации (Morton [2005]). Обучающие вопросы выглядят примерно так:

- P Какой французский монарх восстановил божественное право королей во Франции и был известен как «Король-Солнце» благодаря великолепию своего царствования?
- X В каком конкурсе победила Эймар Куинн с песней «The Voice» в 1996 году, принеся тем самым своей стране четвертую победу за пять лет?

[4] Кто был самым полезным игроком Суперкубка? – *Прим. перев.*

Первый символ обучающего вопроса – это тип ответа, остальное – текст вопроса. Наши обучающие данные поддерживают много типов, но сама система для простоты умеет распознавать только четыре (место, время, человек, организация). Поддерживаемые типы ответов и соответствующие примеры приведены в табл. 8.1.

Таблица 8.1. Типы ответов в обучающих данных

Тип ответа (код в обучающих данных)	Пример
Человек (P)	Какой баскетболист Лиги плюща заработал больше всех очков в одной игре в 1990-х годах?
Место (L)	Какой город первый в мире по выбросам двуокиси серы?
Организация (O)	Какой лыжный курорт был назван лучшим в Северной Америке читателями журнала «Conde Nast Traveler»?
Время (T)	В каком году отцы-пилигримы впервые отметили День благодарения?
Продолжительность (R)	Сколько лет шел телевизионный сериал «Дымок из ствола»?
Денежная сумма (M)	Сколько платят сальвадорским рабочим за каждую пошитую ими куртку из коллекции Лиз Клэйборн ценой 198 долларов?
Процент (C)	Какой процент выходящих в США газет заявляют, что получают прибыль от своего сайта?
Количество (A)	Какая самая низкая температура была зарегистрирована в ноябре в провинции Нью-Брансуик?
Расстояние (D)	На каком примерно максимальном расстоянии слышен удар грома?
Описание (F)	Что такое сухой лед?
Название (W)	В какой написанной в четырнадцатом веке аллитерационной поэме Вильяма Лэнгфорда рассказчику во сне является череда аллегорических видений?
Определение (B)	Что означают буквы O.H.M.S.[1] на штампе для гашения почтовой марки?
Прочее (X)	Как появился банановый сплит?

[1] On Her Majesty's Service – на службе ее величества. – *Прим. перев.*

Для обучения классификатора типов ответов мы вызываем метод `main` класса `AnswerTypeClassifier`:

```
java -cp -Dmodels.dir=<Path to OpenNLP Models Dir>
        -Dwordnet.dir=<Path to WordNet 3.0> \
        <CLASSPATH> com.tamingtext.qa.AnswerTypeClassifier \
        <Path to questions-train.txt> <Output path>
```

Мы пропустим этап тестирования, который обычно неотъемлем от построения модели классификатора, поскольку Том уже протестировал его в своей диссертации. Если вам интересно, как производится тестирование, обратитесь к главам 5 и 7.

Код этапа обучения довольно прост: с помощью средств OpenNLP производится разбиение на блоки (поверхностный разбор) и частеречная разметка обучающего набора, и результаты подаются на вход классификатора MaxEnt из того же проекта OpenNLP. Наиболее важные фрагменты кода показаны в листинге 8.1. Обучение модели типов ответов аналогично другим примерам обучения, рассмотренным в главах, посвященных OpenNLP (распознавание именованных сущностей и пометка документов).

Листинг 8.1. Обучение модели типов ответов

```
AnswerTypeEventStream es = new AnswerTypeEventStream(trainFile,
    actg, parser);
GISModel model = GIS.trainModel(100,
                                new TwoPassDataIndexer(es, 3));
new DoccatModel("en", model).serialize(
                                new FileOutputStream(outFile));
```

С помощью потока событий, по которому подаются обучающие примеры, выполнить обучение, применяя классификатор OpenNLP MaxEnt

Обучив модель, мы должны писать код, в котором эта модель будет использоваться. Для этого напишем класс, производный от встроенного в Solr класса `QParser` (и фабричный класс `QParserPlugin`), назвав его `QuestionQParser` (соответственно `QuestionQParserPlugin`). Но предварительно покажем, как выглядит конфигурация `QuestionQParser`:

```
<queryParser name="qa" class="com.tamingtext.qa.QuestionQParserPlugin"/>
```

Как видите, для Solr важно, прежде всего, чтобы в конфигурации был указан не `QParser`, а `QParserPlugin`.

Главное, что должен сделать класс `QuestionQParserPlugin`, – загрузить модель ТО и сконструировать объект `QuestionQParser`. Как и в случае класса `QParser` и ассоциированной с ним фабрики, мы хотим выполнить все длительные или одноразовые вычисления на эта-

пе инициализации `QParserPlugin`, а не в самом `QParser`, потому что экземпляры последнего конструируются для каждого запроса, тогда как `QParserPlugin` создается только один раз (до очередной фиксации). В случае `QuestionQParser` код инициализации загружает модель TO и еще один ресурс – WordNet. Код прямолинеен и показан ниже.

Листинг 8.2. Код инициализации

```
public void init(NamedList initArgs) {
 SolrParams params = SolrParams.toSolrParams(initArgs);
 String modelDirectory = params.get("modelDirectory",
        System.getProperty("model.dir"));
 String wordnetDirectory = params.get("wordnetDirectory",
        System.getProperty("wordnet.dir"));
 if (modelDirectory != null) {
  File modelsDir = new File(modelDirectory);
  try {
    InputStream chunkerStream = new FileInputStream(
       new File(modelsDir,"en-chunker.bin"));
    ChunkerModel chunkerModel = new ChunkerModel(chunkerStream);
    chunker = new ChunkerME(chunkerModel);
    InputStream posStream = new FileInputStream(
       new File(modelsDir,"en-pos-maxent.bin"));
    POSModel posModel = new POSModel(posStream);
    tagger = new POSTaggerME(posModel);
    model = new DoccatModel(new FileInputStream(
       new File(modelDirectory,"en-answer.bin")))
        .getChunkerModel();
    probs = new double[model.getNumOutcomes()];
    atcg = new AnswerTypeContextGenerator(
       new File(wordnetDirectory, "dict"));
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
 }
}
```

Каталог modelDirectory содержит все модели OpenNLP, используемые в этой книге

WordsNet – это лексический ресурс, помогающий определить тип ответа

Древовидный модуль разбиения на блоки совместно с анализатором выполняет поверхностный разбор вопросов

Tagger отвечает за частеречную разметку

Создаем саму модель и сохраняем ее для повторного использования. Сама модель потокобезопасна, но объемлющий класс – нет

Создаем объект AnswerTypeContextGenerator, отвечающий за отбор признаков

WordNet (http://wordnet.princeton.edu/) – это лексический ресурс для английского и других языков, который создан в Принстонском университете и содержит информацию о словах, в том числе о синонимах, антонимах, гипонимах и гиперонимах. Его лицензия допускает использование в коммерческих целях. Позже мы покажем, как он

помогает устанавливать смысл вопроса.

Инициализировав все ресурсы, фабрика переходит к своей главной задаче – создать объект `QuestionQParser`. Этот код показан в следующем листинге.

Листинг 8.3. Создание QuestionQParser

```
@Override
public QParser createParser(String qStr, SolrParams localParams,
                            SolrParams params,
                            SolrQueryRequest req) {
 answerTypeMap = new HashMap<String, String>();
 answerTypeMap.put("L", "NE_LOCATION");
 answerTypeMap.put("T", "NE_TIME|NE_DATE");
 answerTypeMap.put("P", "NE_PERSON");
 answerTypeMap.put("O", "NE_ORGANIZATION");
 QParser qParser;

 if (params.getBool(QAParams.COMPONENT_NAME, false) == true
      && qStr.equals("*:*") == false) {
  AnswerTypeClassifier atc =
      new AnswerTypeClassifier(model, probs, atcg);
  Parser parser = new ChunkParser(chunker, tagger);
  qParser = new QuestionQParser(qStr, localParams,
          params, req, parser, atc, answerTypeMap);
 } else {
  // если qa отключен, выполняем обычный запрос
  qParser = req.getCore().getQueryPlugin("edismax")
        .createParser(qStr, localParams, params, req);
 }
 return qParser;
}
```

Конструируем словарь типов ответов, которые готовы обрабатывать, а именно: места, люди, время и даты

По этой ветви if идем, чтобы создать обычный анализатор запросов Solr в случае, когда пользователь не ввел никакого вопроса или ввел запрос *:* (MatchAllDocsQuery)

Классификатор AnswerTypeClassifier пользуется обученной моделью типов ответов (находится в каталоге modelsDirectory) для классификации вопроса

Конструируем объект разбиения на блоки (анализатор), который будет разбирать вопрос пользователя

Создаем QuestionQParser, передавая конструктору вопрос пользователя и предварительно инициализированные в методе init ресурсы

Наибольший интерес в этом коде представляет конструирование словаря типов ответов и объекта `QuestionQParser`. Словарь сопоставляет внутренние коды, генерируемые классификатором `AnswerTypeClassifier`, меткам, которые были заданы в исходном документе (см. табл. 8.1). Например, метке *L* (географические названия). Впоследствии мы воспользуемся этим словарем для конструирования частей запроса к Solr. Объект класса `QuestionQParser`

занимается собственно разбором вопроса и созданием запроса к Solr/Lucene. А теперь снимем еще один слой и познакомимся с деталями реализации `QuestionQParser`.

У класса `QuestionQParser` три обязанности, и все они выполняются внутри метода `parse`:

- разбить вопрос на блоки и построить объект `Parse`;
- вычислить тип ответа;
- создать запрос к Solr/Lucene типа `SpanNearQuery` (подробнее об этом ниже).

8.4.2. Разбиение вопроса на блоки

Разбиение на блоки (chunking) – это облегченная форма грамматического анализа (который иногда называют *глубоким разбором*). Она бывает полезна для экономии времени процессора, когда интерес представляют только основные структуры предложения, например именная и глагольная группы, а прочие части можно проигнорировать. Это именно то, что нам сейчас нужно. Нам ни к чему глубокий разбор, достаточно выделить основные части вопроса. Из всего кода в `QParser` разбором занимается всего одна строка:

```
Parse parse = ParserTool.parseLine(qstr, parser, 1)[0];
```

Разобрать вопрос с помощью TreebankChunker. Получившийся объект Parse можно затем передать классификатору для определения типа ответа

Отметим, что в этом примере мы передаем ссылку на объект типа `Parser`. На самом деле это экземпляр написанного нами класса `ChunkParser`, который реализует интерфейс `Parser` из библиотеки OpenNLP. Объект `ChunkParser` создает результат поверхностного разбора поданного на вход вопроса, применяя класс OpenNLP `TreebankChunker`, который, как явствует из названия, пользуется ресурсами Penn Treebank (см. главу 2) с конференции по вычислительному обучению естественным языкам (Conference on Computational Natural Language Learning) 2000 (см. http://www.cnts.ua.ac.be/conll2000/chunking/) для разбиения на блоки и классом `ParserTagger` для создания объекта `Parse`. Класс `ParserTagger` отвечает за частеречную разметку (POS) слов вопроса. Этот класс является необходимым элементом модуля разбиения на блоки, потому что модель этого модуля обучалась на информации о частях речи. Иными словами, в данном случае без меток частей речи разбиение на блоки невозможно. Интуитивно это кажется разумным: выделить именные группы легче, если будут известны все имена существительные. В на-

шем примере мы подали на вход частеречному разметчику готовую модель в файле tag.bin.gz, который входит в состав OpenNLP. Аналогично объект `TreebankChunker` пользуется моделью EnglishChunk.bin.gz, которая включена в исходный код к книге, чтобы на основе результата частеречного разметчика создать объект `Parse`. Работы много (правда, вся она скрыта за вызовом одного метода), но в итоге мы получаем возможность узнать, какой ответ ожидает получить пользователь. Читайте дальше.

8.4.3. Вычисление типа ответа

Следующий шаг – определить тип ответа, а затем поискать в словаре, сопоставляющем внутренние коды меткам, проставленным на этапе индексирования. Этот код показан ниже.

Листинг 8.4. Определение типа ответа

```
String type = atc.computeAnswerType(parse);
String mt = atm.get(type);
```

Очевидно, что большой объем работы производится в классе `AnswerTypeClassifier` и вызываемых из него, поэтому рассмотрим этот класс, а уже затем перейдем к генерации запроса к Solr/Lucene.

Как следует из названия, `AnswerTypeClassifier` – классификатор, который принимает вопросы, а возвращает тип ответа. Во многих смыслах это сердце всей QA-системы, поскольку без правильного типа ответа трудно было бы найти фрагменты, в которых не просто упоминаются требуемые ключевые слова, а содержится ожидаемый ответ. Например, если пользователь спрашивает «Who won the 2006 Stanley Cup?»[5], то правильно вычисленный тип ответа подскажет, что ответ должен содержать имя человека или название организации. Затем система, встретив фрагмент, содержащий слова *won*, *2006* и *Stanley Cup*, сможет проранжировать его и определить, соответствует ли совпавшее слово или словосочетание с известным типом ответа. Например, система могла бы встретить предложение «The 2006 Stanley Cup finals went to 7 games»[6]. Здесь не упоминаются ни люди, ни организации, и, стало быть, система может отбросить этот фрагмент, поскольку он не соответствует типу ответа.

Сконструированный объект `AnswerTypeClassifier` загружает модель типов ответов, обученную ранее, и создает объект

[5] Кто выиграл Кубок Стэнли в 2006 году? – *Прим. перев.*

[6] В финале Кубка Стэнли потребовалось провести все 7 игр. – *Прим. перев.*

AnswerTypeContextGenerator, который, опираясь на WordNet и кое-какие эвристики, решает, какие признаки вернуть объекту AnswerTypeClassifier для классификации. Код класса AnswerTypeClassifier, вызывающий AnswerTypeContextGenerator, находится в методах computeAnswerType и computeAnswerTypeProbs и выглядит примерно так.

Листинг 8.5. Вычисление типа ответа

```
public String computeAnswerType(Parse question) {
 double[] probs = computeAnswerTypeProbs(question);
 return model.getBestOutcome(probs);
}
public double[] computeAnswerTypeProbs(Parse question) {
 String[] context = atcg.getContext(question);
 return model.eval(context, probs);
}
```

Получить вероятности различных типов ответа путем обращения к computeAnswerTypeProbs

Зная вероятности, запросить у модели наилучшее предсказание. Это сводится к простому вычислению максимального элемента в массиве

Запросить у AnswerTypeContextGenerator список признаков (контекст), который должен иметь достаточную предсказательную способность относительно типа ответа

Оценить полученные признаки и определить вероятности разных типов ответа

Ключ к этому коду – две строки метода computeAnswerTypeProbs. В первой у объекта AnswerTypeContextGenerator запрашивается набор признаков, полученный в результате разбора вопроса, а во второй эти признаки передаются модели для оценивания. Модель возвращает массив вероятностей каждого возможного предсказания, из них выбирается максимальная, и соответствующее предсказание возвращается в виде типа ответа.

Наверное, вы помните из материала предыдущих глав, что отбор признаков зачастую является трудной задачей, поэтому имеет смысл более пристально разобраться в работе класса AnswerTypeContextGenerator. Отбор признаков в нем возложен на метод getContext(). Этот метод реализует несколько простых правил, направленных на выбор хороших признаков, исходя из типа заданного вопроса. Большая часть правил основана на поиске основной глагольной и именной группы в вопросе. Сформулировать их можно следующим образом.

- Если присутствует вопросительное слово (who, what, when, where, why, how, whom, which)[7], то:
  - включить это вопросительное слово и снабдить его меткой *qw* (qw=who);

[7] кто, что, когда, где, почему, как, кому, какой. – *Прим. перев.*

  - включить глагол справа от вопросительного слова, снабдить его меткой *verb*, конкатенировать с вопросительным словом и снабдить результат меткой *qw_verb* (`verb=entered, qw_verb=who_entered`);
  - включить все слова справа от глагола, снабдив их меткой *rw* (`rw=monarchy`).
- Если присутствует фокальное существительное (основное существительное вопроса), то:
  - добавить начальное слово именной группы, снабдив его меткой *hw* (`hw=author`) и указав часть речи с меткой *ht* (`ht=NN`);
  - включить все слова, уточняющие существительное, снабдив их меткой *mw* (`mw=European`) и указав часть речи с меткой *mt* (`mt=JJ`);
  - включить синсет WordNet (синсетом называется группа синонимов слова) для этого существительного, снабдив его меткой *s* (`s=7347`, идентификатор синсета);
  - указать с помощью метки *fnIsLast*, является ли фокальное существительное последним в группе (`fnIsLast=true`).
- Включить признак по умолчанию `def`. Это пустая метка, добавляемая для нормирования. Она есть в любом вопросе вне зависимости от наличия других признаков и потому является базовым признаком для обучения системы.

Прежде чем переходить к обсуждению существенных особенностей этого перечня, рассмотрим признаки, отобранные для вопроса «Which European patron saint was once a ruler of Bohemia and has given his name to a Square in Prague?»[8]. Признаки таковы:

```
def, rw=once, rw=a, rw=ruler, rw=of, rw=Bohemia, rw=and,
rw=has, rw=given, rw=his, rw=name, rw=to, rw=a, rw=Square, rw=in,
rw=Prague?, qw=which, mw=Which, mt=WDT, mw=European, mt=JJ, mw=patron,
mt=NN, hw=saint, ht=NN, s=1740, s=23271, s=5809192, s=9505418,
s=5941423, s=9504135, s=23100, s=2137
```

Этот и многие другие примеры можно наблюдать воочию, выполнив метод `demonstrateATCG` из класса `AnswerTypeTest`, код которого приведен ниже:

```
AnswerTypeContextGenerator atcg =
    new AnswerTypeContextGenerator(
```

[8] В честь какого европейского святого, когда-то правившего Богемией, названа одна из площадей в Праге? – *Прим. перев*

```
        new File(getWordNetDictionary().getAbsolutePath()));
InputStream is = Thread.currentThread().getContextClassLoader()
    .getResourceAsStream("atcg-questions.txt");
assertNotNull("input stream", is);
BufferedReader reader =
    new BufferedReader(new InputStreamReader(is));
String line = null;
while ((line = reader.readLine()) != null){
 System.out.println("Question: " + line);
 Parse[] results = ParserTool.parseLine(line, parser, 1);
 String[] context = atcg.getContext(results[0]);
 List<String> features = Arrays.asList(context);
 System.out.println("Features: " + features);
}
```

Но вернемся к отбору признаков: большинство признаков отбираются с помощью простых правил или регулярных выражений, как видно из кода класса `AnswerTypeContextGenerator`. Задача отыскания фокального (или основного) существительного выбивается из общего ряда, поскольку она добавляет несколько признаков, а также потому что для ее решения нужно проделать немало работы. Фокальное существительное зависит от типа вопросительного слова (кто, что, какой и т. д.) и очень важно для определения искомого. Например, в нашем вопросе о правителе Богемии, фокальным существительным является *saint* (святой), т. е. мы ищем человека, признанного святым. Это существительное можно использовать совместно с WordNet для поиска синонимов слова saint, которые могли бы оказаться полезными для выявления других вопросов на ту же тему, выраженных иными словами. При поиске фокального существительного мы также применяем определенные правила для исключения ложных совпадений. Они опять-таки основаны на простых проверках и регулярных выражениях. Большая часть этой работы производится в методе `findFocusNounPhrase` класса `AnswerTypeContextGenerator`, который мы не приводим из-за его большого размера.

Наконец, не забывайте, что весь этот процесс отбора признаков основан на выполненном Томом анализе вопросов в части того, что именно важно при построении модели. Но это не единственный способ. Более того, может статься, что при наличии большего объема обучающих данных модель системы можно было бы обучить, и не прибегая к процессу отбора признаков. В некоторых отношениях привлечение человека к отбору признаков – компромисс между временем, потраченным на сбор и аннотирование примеров, и временем, потраченным на предварительный поиск закономерностей в сущест-

вующих вопросах. Что лучше в вашей системе, зависит от объема данных и располагаемого времени.

8.4.4. Генерация запроса

Определив тип ответа, мы должны сгенерировать запрос, который найдет фрагменты-кандидаты в нашем индексе. Чтобы фрагмент был полезен QA-системе, он должен обладать несколькими свойствами.

- В окне фрагмента должны присутствовать одно или несколько слов, соответствующих типу ответа.
- В окне фрагмента должны присутствовать один или несколько термов из исходного запроса.

Чтобы сконструировать запрос, который отберет фрагменты-кандидаты, удовлетворяющие этим требованиям, мы должны точно знать, в каком месте документа имело место совпадение. В Solr (и Lucene) для этого служат запросы типа `SpanQuery` и производных от него. Точнее, запрос `SpanQuery` ищет документы, как и другие запросы в Lucene, но ценой дополнительного времени вычислений дает еще информацию о позиции, которую мы можем проанализировать для возврата более узкого фрагмента, чем весь объемлющий документ. Наконец, для поиска фрагментов мы должны создать экземпляр производного класса `SpanNearQuery`, потому что указанные термы и тип ответа должны находиться рядом. Класс `SpanNearQuery` позволяет создавать сложные фразовые запросы, составленные из нескольких экземпляров других производных от `SpanQuery` классов. В листинге ниже показан код создания запроса.

Листинг 8.6. Генерация запроса

```
List<SpanQuery> sql = new ArrayList<SpanQuery>();
if (mt != null) {
 String[] parts = mt.split("\|");
 if (parts.length == 1) {
   sql.add(new SpanTermQuery(new Term(field, mt.toLowerCase())));
 } else {
   for (int pi = 0; pi < parts.length; pi++) {
     sql.add(new SpanTermQuery(new Term(field, parts[pi])));
   }
 }
}
try {
 Analyzer analyzer = sp.getType().getQueryAnalyzer();
 TokenStream ts = analyzer.tokenStream(field, new StringReader(qstr));
```

```
 while (ts.incrementToken()) {
   String term = ((CharTermAttribute)
      ts.getAttribute(CharTermAttribute.class)).toString();
   sql.add(new SpanTermQuery(new Term(field, term)));
 }
} catch (IOException e) {
 throw new ParseException(e.getLocalizedMessage());
}
return new SpanNearQuery(sql.toArray(new SpanQuery[sql.size()]),
          params.getInt(QAParams.SLOP, 10), true);
```

Здесь можно выделить три шага.

- Добавить тип (или типы) ответа в запрос, воспользовавшись одним или несколькими экземплярами `SpanTermQuery`. Если типов ответов несколько, они соединяются с помощью класса `SpanOrQuery`.
- Проанализировать запрос с помощью лексического анализатора для данного поля и создать объекты `SpanTermQuery` для каждого терма.
- Сконструировать объект `SpanNearQuery`, который объединяет все термы, применяя переданный пользователем коэффициент гибкости (или значение по умолчанию 10).

Это не единственный способ построить запрос. Например, можно было бы попытаться создать более избирательный запрос, для чего провести углубленный анализ вопроса с выделением словосочетаний или частей речи, так чтобы фрагмент считался кандидатом, если найденный в нем терм выступает в роли той же части речи, что и в вопросе. Но в любом случае построенный запрос передается Solr, и в ответ мы получаем список документов, который можем ранжировать с помощью написанного нами класса `PassageRankingComponent`, который рассматривается в следующем разделе.

8.4.5. Ранжирование фрагментов-кандидатов

По сравнению с разбором вопроса и отбором признаков ранжирование фрагментов в нашем случае гораздо проще – мы применяем прямолинейную процедуру, которая впервые была описана (Singhal 1999) на конференции TREC-8 по вопросно-ответным системам. И хотя другие системы прошли мимо этого подхода, он все же остается вполне разумным, прост в реализации и сравнительно эффективен в фактографической системе. Суть его заключается в том, чтобы искать

подходящие термы в последовательности окон вокруг того места, где было найдено совпадение с запросом; потому-то мы и использовали класс `SpanQuery` и производные от него. Точнее, определяется начало и конец совпадения с термами запроса, а затем строятся два окна с заданным числом термов (в нашей программе по умолчанию 25, но эту величину можно переопределить в параметрах запроса к Solr) по обе стороны окна. Это показано на рис. 8.5.

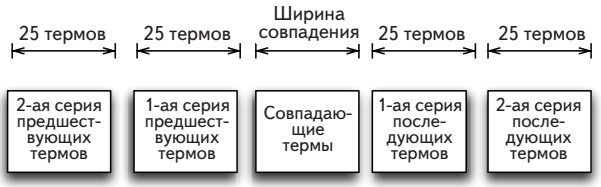

Рис. 8.5. Компонент ранжирования фрагмента строит последовательность окон вокруг совпавших термов из запроса, а затем производит ранжирование

Для эффективного построения фрагментов мы пользуемся принятым в Lucene механизмом хранения термов в векторах. Попросту говоря, *вектором термов* называется ассоциированная с каждым документом структура данных, в которой хранятся термы, их частоты и позиции в документе. В отличие от инвертированного индекса, используемого для поиска, эта структура ориентирована на документы, а не на термы. А это означает, что она вполне годится для операций, в которых требуется весь документ (например, выделение слов или анализ фрагментов), и не годится для операций, которым нужен быстрое обращение к термам (как при поиске). В предположении, что фрагмент инкапсулирован в классе `Passage`, процесс ранжирования показан в следующем листинге.

Листинг 8.7. Ранжирование фрагмента-кандидата

```
protected float scorePassage(Passage p,
                             Map<String, Float> termWeights,
                             Map<String, Float> bigramWeights,
                             float adjWeight, float secondAdjWeight,
                             float biWeight) {
  Set<String> covered = new HashSet<String>();
  float termScore = scoreTerms(p.terms, termWeights, covered);
  float adjScore = scoreTerms(p.prevTerms, termWeights, covered) +
                   scoreTerms(p.followTerms, termWeights, covered);
```

Ранжируем термы в главном окне

Ранжируем термы в окнах слева и справа от главного окна

```
float secondScore =
      scoreTerms(p.secPrevTerms, termWeights, covered)
        + scoreTerms(p.secFollowTerms, termWeights, covered);
// Премия за совпадение биграмм в главном окне
float bigramScore =
      scoreBigrams(p.bigrams, bigramWeights, covered);
float score = termScore + (adjWeight * adjScore) +
                          (secondAdjWeight * secondScore) +
                          (biWeight * bigramScore);
return (score);
}
```

Ранжируем термы в окнах слева от предыдущего и справа от последующего

Ранжируем биграммы во фрагменте

Окончательная оценка фрагмента равна взвешенной сумме отдельных оценок. За совпадение биграмм дается премия

В процессе ранжирования вычисляется взвешенная сумма оценок для каждого окна внутри фрагмента, причем максимальный вес приписывается совпадению в главном окне, а чем дальше окно от главного, тем меньше вес совпадения в нем. Если совпали два расположенных подряд слова, то к оценке прибавляется дополнительное слагаемое (премия за биграмму). Окончательная оценка служит для ранжирования фрагмента с помощью очереди с приоритетами. Имея ранжированный набор фрагментов, мы записываем результаты в структуру ответа Solr и возвращаем клиенту. Пример результата показан на рис. 8.6.

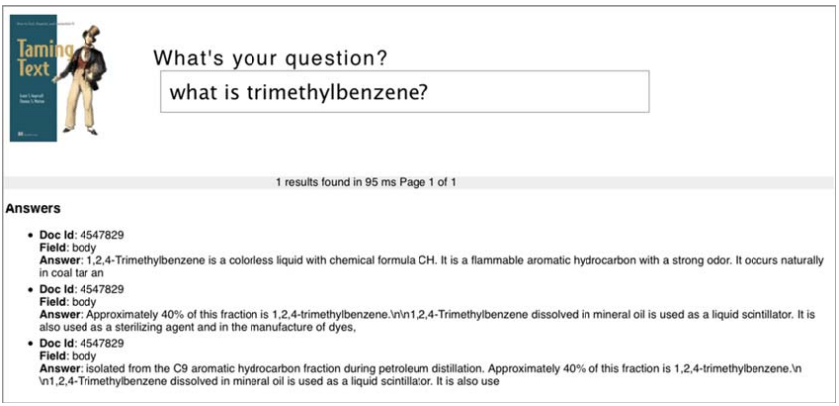

Рис. 8.6. Пример вопросно-ответной системы Taming Text в действии: ответ на вопрос «Что такое триметилбензол?»

Итак, мы имеем работоспособную систему, в которой сначала производится обработка заданного пользователем вопроса, а затем генерация поискового запроса, в ответ на который возвращаются фрагменты кандидаты. На последнем этапе фрагменты ранжируются с

помощью простого алгоритма, который ищет термы в окрестности совпавших с запросом частей. А теперь подумаем, как можно улучшить эту систему.

8.5. Усовершенствование системы

Если вы следили за кодом, то без сомнения понимаете, что есть много возможностей улучшить систему. Ниже перечислено несколько идей.

- Во многих QA-системах после анализа вопроса выбирается предопределенный шаблон вопроса, который затем используется в качестве образца для определения кандидатов на основе фрагментов, отвечающих шаблону.
- Создавать более ограничительные запросы к Lucene, требуя, чтобы тип ответа отстоял не дальше, чем на заданное число термов, или даже внутри конкретного предложения.
- Не только находить фрагмент, содержащий ответ, но и извлекать ответ из этого фрагмента.
- Если два или более фрагментов дают одинаковые или очень похожие ответы, устранять дубликаты и увеличивать ранг этого результата.
- Обрабатывать случай, когда не удается найти хороший ответ, возвращая результаты простого поиска или прибегая к другим способам анализа.
- Показывать уровни доверия или объяснения того, как был получен ответ, и предоставлять пользователю средства уточнения результатов.
- Включить специализированные способы построения запросов для вопросов определенного типа. Например, для ответа на вопросы вида «Кто такой X?» можно было бы использовать базу знаний о знаменитых людях, а не просто искать ответ в текстах.

И это лишь небольшая часть возможных улучшений. Предоставляем тебе, любезный читатель, добавить свои идеи.

8.6. Резюме

Построение работоспособной вопросно-ответной системы – отличный способ продемонстрировать в действии многие рассмотренные в

этой книге принципы. Например, на этапе анализа вопроса приходится применять методы сравнения строк, а также распознавание именованных сущностей и разметку, а для поиска фрагментов и ранжирования нужно задействовать средства глубокого поиска, позволяющие не просто найти подходящие документы, но и узнать, где именно имело место совпадение. Зная позиции, мы затем еще раз применили методы сравнения строк для ранжирования фрагментов и выработки ответа. В общем и целом, мы получили простую фактографическую вопросно-ответную систему. Сумеет ли она победить в *Jeopardy!*? Конечно, нет. Но надеемся, что ее достаточно, чтобы понять, как можно построить такую систему из свободно доступных программ с открытым исходным кодом.

8.7. Ресурсы

Morton, Thomas. 2005. Using Semantic Relations to Improve Information Retrieval. University of Pennsylvania. http://www.seas.upenn.edu/cis/grad/documents/morton-t.pdf.

Singhal, Amit; Abney, Steve; Bacchiani, Michiel; Collins, Michael; Hindle, Donald; Pereira, Fernando. 1999. "AT&T at TREC-8." AT&T Labs Research. http://trec.nist.gov/pubs/trec8/papers/att-trec8.pdf.

ГЛАВА 9. Неприрученный текст: на переднем крае

В этой главе:

- Пути развития поиска и NLP.
- Поиск в многоязычных текстах и обнаружение эмоций.
- Ресурсы, посвященные новым инструментам, приложениям и идеям.
- Обработка естественного языка высшего порядка: семантика, дискурс и прагматика.

Уф! Долгий же путь мы проделали – и речь не только о терпении, которым вам пришлось вооружиться в ожидании конца книги (поверьте, мы это высоко ценим!). Несколько лет назад все сходили с ума по поиску, а социальные сети только собирались с силами. Идеи, которые, как нам тогда казалось, только проникали в область поиска и NLP, теперь лежат в основе приложений, работающих в самых разных организациях: от крупнейших компаний из списка Fortune 100 до новообразованных стартапов – со всеми промежуточными остановками.

В этой книге мы начали с описания основ работы с текстом, поиска в нем, разметки и группировки. Мы даже написали простую вопросно-ответную систему, что позволило связать воедино многие из рассмотренных концепций. И хотя эти возможности образуют значительную часть функциональности практических приложений обработки текста, они ни в коем случае не исчерпывают всего многообразия информационного поиска и обработки естественных языков (NLP). В частности, быстро развивается такая область, как вы-

явление эмоционального отношения пользователей к определенным сущностям (торговым маркам, местам, людям и т. д.). Это связано с возросшим интересом к потоку сознания, выраженному в твитах.

В этой главе мы поговорим как об анализе тональности высказываний, так и о других продвинутых методах, и постараемся сообщить достаточно информации, чтобы вы могли приступить к делу, а заодно пробудить интерес к исследованию более трудных задач. Мы расскажем об идеях, стоящих за каждой из рассматриваемых тем, предложим материалы для дальнейшего чтения и ссылки на библиотеки и инструменты с открытым исходным кодом (если таковые имеются), которые могут помочь в реализации. Но в отличие от предыдущих глав, примеров кода здесь не будет.

Начнем с рассмотрения таких высокоуровневых аспектов языка, как семантика, дискурс-анализ и прагматика, а затем перейдем к обсуждению реферирования документов и распознавания событий и отношений. Затем мы остановимся на выявлении важности и эмоциональной окраски текста и закончим главу (и книгу) рассмотрением поиска в многоязычных текстах.

9.1. Семантика, дискурс и прагматика: высшие уровни NLP

Большая часть этой книги посвящена тому, как помочь пользователям – живым и разумным людям – извлекать смысл из текста с помощью таких действий, как грамматический разбор, пометка, поиск и другие способы выделения пригодных для восприятия информационных единиц. Но что, если мы попросим компьютер разобраться в смысле текста и сообщить нам результат? Самое простое – спросить о смысле конкретного набора слов (словосочетания, предложения), но не мог бы компьютер проделать более глубокий анализ смысла текста? Например, сказать, что имел в виду автор, или определить, похожи ли по смыслу два документа? Или вот – не может ли машина воспользоваться своими знаниями о мире, чтобы «читать между строк»?

Эти способы рассуждений о смысле и связанных с ним предметах принято относить к трем разным дисциплинам (см. Liddy [2001] и Manning [1999]).

- *Семантика* – изучение смысла слов и связей между ними, благодаря которым образуются более крупные смысловые единицы (например, предложения).

- *Дискурс* – базирующийся на семантическом уровне, дискурс-анализ ставит целью установить связи между предложениями. Некоторые авторы объединяют дискурс со следующим уровнем смысла: прагматикой.
- *Прагматика* – изучает, какой вклад в смысл текста вносят контекст, знания о мире, языковые соглашения и прочие абстрактные свойства.

Примечание. Хотя изучению смысла текста посвящены, прежде всего, эти три дисциплины, свой вклад вносят и все остальные уровни языка. Например, если выстроить цепочку случайно взятых символов, то, скорее всего, они не составят слова, а, значит, их использование в предложении бессмысленно. С другой стороны, если изменить порядок слов в предложении (синтаксис), то может измениться и его смысл. Например, в книге «Natural Language Processing» (Liddy [2001]) приводится пример двух предложений: «The dog chased the cat» и «The cat chased the dog»[1]; в них все слова одинаковы, но из-за разного порядка смысл предложения полностью меняется.

Дав определения, давайте немного углубимся в эти дисциплины и рассмотрим некоторые примеры и инструменты, которые можно использовать для извлечения смысла из текста.

9.1.1. Семантика

С практической точки зрения, приложения, занимающиеся обработкой текста на семантическом уровне, обычно интересуются двумя аспектами (семантика в целом шире).

- Смысл слов, например, определение синонимов, антонимов, гиперонимов, гипонимов и т. д. и связанная с этим задача *разрешения лексической неоднозначности* – выбор правильного смысла слова при наличии нескольких значений; например, слово «bank» может обозначать как финансовое учреждение, так и берег реки.
- Коллокации и идиомы и связанные с этим *статистически невероятные фразы* (SIP) – это группы слов, смысл которых больше, чем сумма смыслов отдельных слов. Иначе говоря, смысл целого не сводится к смыслу составных частей. Так, фраза *bit the dust*[2] относится к смерти или провалу, а не к поеданию частиц пыли.

---

[1] «Собака гналась за кошкой» и «Кошка гналась за собакой». – *Прим. перев.*

[2] Склеить ласты, потерпеть сокрушительную неудачу. Дословно «кусать пыль». Близкая по смыслу русская идиома – «пыль глотать». – *Прим. перев.*

В первом случае установление смысла слов с помощью синонимов и прочего может сильно повысить качество поиска, особенно если удается разрешить лексическую неоднозначность, отобрав только синонимы, относящиеся к контексту запроса. Но в общем случае разрешение неоднозначности – дело трудное и долгое, поэтому в реальных поисковых системах в полном объеме не применяется. Однако, если известны предметная область и наиболее вероятные интересы пользователей, то такой подход может улучшить результаты и без развертывания полноценной системы разрешения неоднозначности. Для первоначального знакомства с темой разрешения лексической неоднозначности рекомендуем главу 7 книги Manning, Schütze «Foundations of Statistical Natural Language Processing» (1999). Разрешение неоднозначности часто является обязательным требованием к программам машинного перевода с одного языка на другой. К числу программных средств разрешения лексической неоднозначности относятся:

- *SenseClusters* – http://www.d.umn.edu/~tpederse/senseclusters.html;
- *Lextor* – http://wiki.apertium.org/wiki/Lextor. Отметим, что Lextor – часть более крупного проекта; чтобы использовать его автономно, возможно, придется проделать дополнительную работу.

Многие считают, что для такого рода задач также полезен тезаурус WordNet, разработанный Принстонским университетом (см. главу 3).

У коллокаций и SIP также много применений – от поиска и генерации текстов (когда компьютер пишет за вас сочинение или составляет отчет) до построения конкорданции[3] для книги или просто для лучшего понимания языка или области исследования. Например, в поисковых приложениях коллокации можно использовать как для более точного задания запроса, так и для создания интерфейсов для извлечения информации, когда пользователю показывается список коллокаций для данного документа и ссылки на другие документы, содержащие те же словосочетания. Или можно представить приложение, которое получает на входе всю литературу по определенной теме и формирует перечень обнаруженных статистически невероятных фраз вместе с определениями, ссылками и прочей информацией, позволяющей быстрее войти в курс дела. Если вас интересует ПО, способное обрабатывать коллокации, то не нужно ходить дальше Apache

[3] Перечень в алфавитном порядке всех слов, находящихся в книге, с указанием места, где они находятся. – *Прим. перев.*

Mahout. См. https://cwiki.apache.org/confluence/display/MAHOUT/Collocations.

Полезно иметь представление и о прочих интересных вопросах семантического порядка, особенно в связи с другими уровнями обработки. Например, чтобы оценить истинность высказывания, необходимо понимать смысл слов в предложении (вдобавок к другим знаниям). Ко всему прочему, семантику бывает сложно выявить из-за наличия квантификаторов и других лексических единиц. Например, двойное отрицание, неправильно помещенные модификаторы и прочие контекстуальные проблемы могут затруднить понимание смысла предложения.

О семантике можно говорить еще много. Посвященная ей статья в википедии (http://en.wikipedia.org/wiki/Semantics) содержит неплохой список источников, с которых можно начать изучение.

9.1.2. Дискурс

Если семантика обычно ограничена одним предложением, то дискурс оперирует связями между предложениями. К дискурсу относятся и такие вещи, как манера говорить, язык тела, речевые акты и т. п., но мы акцентируем внимание на его проявлениях в письменном тексте. Отметим, что иногда дискурс рассматривают совместно со следующим предметом нашего обсуждения, прагматикой.

В контексте обработки естественного языка применение инструментов дискурса обычно сводится к разрешению анафор и определению (разметке) структур в тексте (проблема *сегментации дискурса*). Например, в тексте новости можно выделить и соответственно пометить зачин, основное содержание, ссылки на источники и т. д. *Анафорами* называются ссылки на другие части текста, обычно существительные, встречавшиеся ранее. Например, в предложении «Питера выдвинули в Президенты. Он вежливо отказался» местоимение *Он* является анафорой. В роли анафор могут выступать не только местоимения. Возьмем, к примеру, предложение «Эрик Стаал из Харрикейнс подвел итог игре, забив гол в дополнительное время. Капитан команды забросил шайбу в 2:03 первого дополнительного периода». Здесь *капитан команды* – анафора для Эрика Стаала. Разрешение анафор – частный случай более общей задачи о *разрешении кореференции*, которая нередко опирается на дискурс-анализ и другие уровни обработки. Цель разрешения кореференции заключается в том, чтобы выявить в тексте все упоминания некоторой концепции или сущности. Например, рассмотрим следующий текст.

> *Нью-Йорк Сити (NYC) – крупнейший город США. Иногда именуемый Большим яблоком, городом, который никогда не спит, и Готамом, Нью-Йорк является туристической меккой. В 2008 году в Большом яблоке побывало свыше 40 миллионов туристов. Город также является финансовым локомотивом, поскольку здесь находится Нью-Йоркская фондовая биржа и биржа NASDAQ.*

Нью-Йорк Сити, *NYC*, *Большое яблоко*, *Готам* и *город* – всё это упоминания одного и того же места. Разрешение кореференции, а значит и разрешение анафор, полезны при поиске, в вопросно-ответных системах и многих других случаях. В качестве примера использования в QA-системе предположим, что был задан вопрос «Какие президенты родом из Техаса?», и в качестве источника данных имеется следующий документ (взято из статьи википедии о Линдоне Бейнсе Джонсоне (http://en.wikipedia.org/wiki/Lyndon_B._Johnson):

> *Линдон Бейнс Джонсон (27 августа 1908 – 22 января 1973), которого часто называют просто LBJ, был 36-м президентом США с 1963 по 1969 год... Джонсон, демократ, был представителем США от штата Техас в период с 1937 по 1949 год и сенатором в период с 1949 по 1961 год...*

Применив разрешение кореференции, мы могли бы определить, что 36-ой президент Джонсон был родом из Техаса.

Желающим подробнее познакомиться с разрешением кореференции и анафор рекомендуем начать с какой-нибудь книги по дискурс-анализу, а также прочитать статью в википедии об анафорах (http://en.wikipedia .org/wiki/Anaphora_%28linguistics%29). Что касается реализации, то OpenNLP поддерживает разрешение кореференции.

Сегментация дискурса находит применения в приложениях поиска и NLP. Если говорить о поиске, то идентификация, разметка и потенциально разбиение большого документа на меньшие сегменты часто позволяет получить более точные результаты за счет того, что пользователю предъявляется именно та область документа, где обнаружено совпадение. Попутно это дает возможность назначать лексемам более точные веса. Есть и минус – дополнительная работа для обратной сборки документа в единое целое или для определения того, что результат, охватывающий несколько сегментов, лучше, чем отдельный сегмент.

Сегментация дискурса полезна и для реферирования документов, потому что позволяет получить реферат, более полно охватывающий темы и подтемы, встречающиеся в тексте (подробнее о реферировании мы будем говорить ниже). Различные подходы к этой задаче

описаны к работе «Multi-Paragraph Segmentation of Expository Text». В проекте MorphAdorner (http://morphadorner.northwestern.edu/morphadorner/textsegmenter/) имеется реализация предложенного Хирстом алгоритма TextTiling на Java (на забудьте ознакомиться с лицензией на коммерческое использование), а на сайте CPAN – реализация на Perl (http://search.cpan.org/~splice/Lingua-EN-Segmenter-0.1/lib/Lingua/EN/Segmenter/TextTiling.pm TextTiling.pm). Простую сегментацию можно произвести также с помощью Lucene и Solr на уровне логики приложения на этапах индексирования или обработки запроса (с применением объектов класса `SpanQuery`, которые обеспечивают поиск с учетом позиции). О поиске, основанном на частях документа (passage-based retrieval), написано много работ, из которых можно почерпнуть дополнительные сведения по этой теме. Мы же перейдем к прагматике.

9.1.3. Прагматика

Прагматику интересует контекст и то, как он влияет на способы нашего общения. Благодаря контексту мы можем общаться, не объясняя досконально каждую мельчайшую деталь, необходимую для понимания. Например, на стадии подачи предложения об издании этой книги и позже, на протяжении всего времени ее написания, одним из ключевых для авторов был вопрос о целевой аудитории. С точки зрения бизнеса, это полезно для определения емкости рынка и ожидаемой прибыли, а с точки зрения авторов – критически важно для установления контекста книги.

Проанализировав рынок, мы пришли к выводу, что нашей аудиторией будут разработчики, знакомые с программированием, скорее всего на Java, но при этом малознакомые с идеями и практическим применением поиска и обработки естественного языка и нуждающиеся в использовании соответствующих инструментов и методов. Мы также решили избегать сложных математических рассуждений, а вместо этого предложить работающие примеры, основанные на инструментах с открытым исходным кодом, в которых уже реализованы общеупотребительные алгоритмы. Определив для себя контекст, мы предположили, что читатели уверенно работают с консольными командами, владеют основами Java или другого языка программирования. Это позволило нам обойтись без утомительного описания настройки и простейших принципов программирования.

Проще говоря, прагматика – это сочетание наших знаний лингвистики (морфологии, грамматики, синтаксиса и т. д.) со знаниями об окружающем мире. Прагматика изучает, как «читать между строк»,

чтобы разрешить неоднозначность и понять, что человек имел в виду. Прагматические системы зачастую должны уметь рассуждать о мире, чтобы делать выводы. Например, если бы в примере с Линдоном Джонсоном был задан вопрос «Из какого штата родом Линдон Джонсон?», система должна была бы понять, что Техас – имя собственное и при том название штата.

Легко догадаться, что включение обширных знаний о мире в приложение – задача нетривиальная, поэтому и обработка прагматики обычно сталкивается с трудностями. Но есть много инструментов с открытым исходным кодом, готовых прийти на помощь, в том числе проект OpenCyc (http://www.opencyc.org), тезаурус WordNet (http://wordnet.princeton.edu) и Всемирная книга фактов ЦРУ (https://www.cia.gov/library/publications/the-world-factbook/). Как всегда, разработчики приложений обычно достигают наилучших результатов, сосредоточившись на тех ресурсах, которые действительно полезны для решения задачи, и не пытаясь объять необъятное, собрав в своей программе все, что есть на свете. Правда, это требует скрупулезного процесса предварительной оценки.

Среди других аспектов прагматики назовем использование сарказма, вежливости и других видов поведения, влияющих на общение. Во многих подобных случаях разработчики строят классификаторы, обученные помечать такие акты, как сарказм, чтобы их можно было использовать на последующих этапах обработки, например, для анализа тональности высказываний (обсуждается ниже) или в механизме логического вывода. Глава этой книги, посвященная пометке текста, может стать неплохой отправной точкой для углубленных изысканий в этой области.

В общем и целом, включить в приложение обработку на уровне прагматики нелегко. Желающим узнать о прагматике подробнее рекомендуем книгу «Pragmatics» (Peccei [1999]) или какой-нибудь учебник лингвистики начального уровня. См. также страницу http://www.gxnu.edu.cn/Personal/szliu/definition.html, где имеется хорошее введение и ссылки на литературу.

В нескольких следующих далее разделах мы покажем, как вышеупомянутые высокоуровневые аспекты языка применяются, чтобы облегчить людям работу с гигантскими объемами доступной ныне информации. Для начала посмотрим, как можно использовать NLP для существенного уменьшения количества подлежащей обработке информации путем реферирования отдельных документов и даже целых наборов.

9.2. Реферирование документов и наборов документов

Маниш Катьял

Методы реферирования документов можно использовать для того, чтобы предоставить читателям краткую сводку основного содержания длинного документа или набора документов. Например, на рис. 9.1 показан реферат статьи из газеты «Вашингтон пост» о беспорядках в Египте (из номера от 4 февраля 2011, позже была удалена).

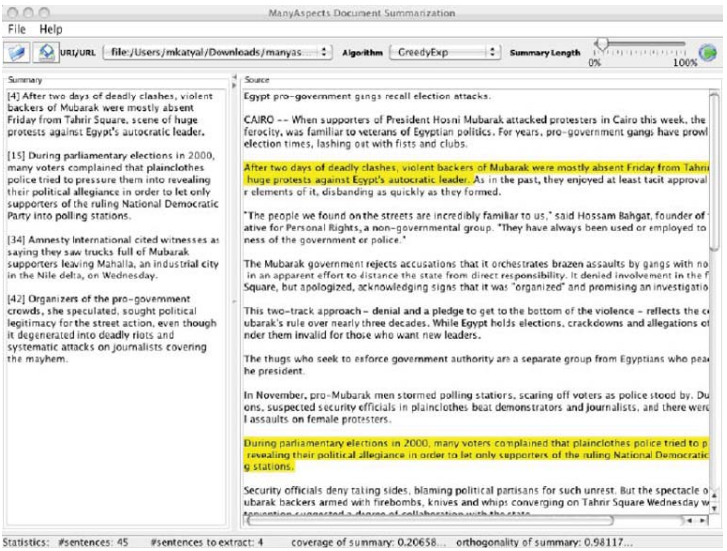

Рис. 9.1. Пример приложения, которое автоматически составляет рефераты длинных документов

Реферат был создан программой многоаспектного реферирования документов (Many Aspects Document Summarization) из проекта IBM AlphaWorks (см. http://www.alphaworks.ibm.com/tech/manyaspects). В левой части показаны предложения, несущие основную смысловую нагрузку. Они дают общее представление о статье, скопированной из газеты. Это пример реферирования одного документа. Составитель рефератов можно использовать также для реферирования связанных новостей на тему беспорядков в Египте. Сгенерированный реферат мог бы включать наиболее важную информацию, например «Волнения в Египте. Президента Мубарака вынуждают уйти в отставку.

Правительство США призывает Мубарака действовать сдержанно». Предложения могут быть взяты из разных источников, описывающих одно и то же событие. Это называется *реферированием набора* или *многодокументным реферированием*.

Другие приложения генерируют аннотации для каждой ссылки на странице результатов поиска или готовят дайджест технических новостей, собирая воедино схожие статьи с сайтов Techcrunch, Engadget и других технологических блогов. Основная цель всех этих приложений – дать читателю достаточно информации, чтобы он мог решить, следует ли читать документ целиком.

Генерация реферата разбивается на три задачи. Первая – отбор содержимого. Здесь составитель строит список предложений-кандидатов, в которых заключен главный смысл. Обычно устанавливается лимит на количество отбираемых слов или предложений. Есть разные подходы к отбору кандидатов. В одном из них составитель ранжирует предложения по важности или центральности для документа. Если некоторое предложение похоже на много других предложений, то оно содержит общую с ними информацию и потому является хорошим кандидатом на включение в реферат. Другой подход – ранжировать предложения с учетом их позиций в документе, по релевантности содержащихся в них слов или отыскивая такие ключевые фразы, как «в общем и целом», «подводя итоги» и т. д. Слово считается информативным или релевантным относительно реферирования, если оно часто встречается в документе, но реже в общем наборе документов. Для определения релевантности слова документу можно применять такие способы назначения весов, как TF-IDF или логарифмическое отношение правдоподобия. Третий подход – вычислить псевдопредложение, являющееся центроидом всех предложений, а затем найти предложения, максимально близкие к центроиду.

При реферировании наборов документов возможно появление сильно пересекающихся групп документов, поэтому составитель реферата должен следить за тем, чтобы не отбирались одинаковые или похожие предложения. Для этого можно штрафовать предложения, похожие на уже отобранные. Таким образом удается устранить избыточность и гарантировать, что каждое предложение несет новую информацию.

По нашему мнению, отбор содержимого – относительно простая задача по сравнению с двумя другими: *упорядочение предложений* и *реализация предложений*. В ходе упорядочения отобранные предложения нужно расположить так, чтобы реферат выглядел связным и информация легко доходила до читателя. В случае реферирования

одного документа можно просто сохранить исходный порядок предложений. При реферировании же набора документов задача существенно усложняется.

И последняя задача – переписать упорядоченные предложения, так чтобы они воспринимались без усилий. В частности, аббревиатуры и местоимения следует развернуть, чтобы они были понятны читателю. Например, предложение «Мубарак пообещал жестко обойтись с мятежниками» можно переписать в виде «Президент Египта, Мубарак пообещал жестко обойтись с мятежниками». На наш взгляд, эту задачу можно опустить, потому что она требует сложного лингвистического анализа текста.

Как и для большинства технологий, существует несколько программ с открытым исходным кодом для реферирования отдельных документов и наборов. Проект MEAD (http://www.summarization.com/mead/) поддерживает многодокументное реферирование. На странице http://tangra.si.umich.edu/~radev/lexrank/ имеется демонстрация алгоритма LexRank, используемого в MEAD для отбора содержимого. Ниже, в разделе о важности мы еще вернемся к рассмотрению идей, положенных в основу LexRank. Еще один проект, Texlexan, находится на сайте http://texlexan.sourceforge.net/. Он умеет выполнять реферирование, анализ текста и классификацию. Работает с текстами на английском, французском, немецком, итальянском и испанском языках.

О реферировании текстов см. книгу «Speech and Language Processing» (Jurafsky [2008]). Научные статьи на эту тему публикуются на сайте Document Understanding Conference (DUC) по адресу http://www-nlpir.nist.gov/projects/duc/pubs.html. Конференция DUC была задумана как серия конкурсов по реферированию текстов.

Как и реферирование, наша следующая дисциплина, извлечение отношений, имеет целью вычленить из текста ключевую информацию. Но в отличие от реферирования, она направлена в большей степени на добавление в текст структуры, которая может быть использована на последующих этапах обработки или представлена конечному пользователю.

9.3. Извлечение отношений

Вайджанат Н. Рао

Задача *извлечения отношений* (relation extraction, RE) – выявить отношения, упоминаемые в тексте. Обычно *отношением* называется

функция с одним или несколькими аргументами, каждый аргумент которой представляет концепцию, объект или человека в реальном мире, а отношение описывает вид ассоциации или взаимодействия между аргументами. Работы в области RE сосредоточены в основном на бинарных отношениях, т. е. на функциях с двумя аргументами, но их легко обобщить на более сложные связи, связав вместе общие сущности. В качестве примера бинарного отношения рассмотрим предложение «*Билл Гейтс* – сооснователь *Майкрософт*». RE-система могла бы извлечь из этого предложения отношение «сооснователь», представив его в виде *сооснователь*(*Билл Гейтс, Майкрософт*). Далее в этой главе мы будем иметь дело только с извлечением бинарных отношений, если явно не оговорено противное.

Другой пример RE-системы – система T-Rex. Ее общая архитектура и пример показаны на рис. 9.2. T-Rex – разработанная Шеффилдским университетом система с открытым исходным кодом для извлечения отношений. Более подробно она описывается ниже в этом разделе. Текстовые документы подаются на вход RE-системы, которая извлекает из них отношения.

RE-система в T-Rex состоит из двух основных подсистем: *Процессор* и *Классификатор*. Они будут подробно объяснены ниже. Справа на рисунке мы видим пример документа, пропущенного через систему T-Rex. Рассмотрим предложение «Альберт Эйнштейн, знаменитый физик-теоретик, родился в Ульме 14 марта 1879 года». RE-система извлечет из него отношения *Род-занятий*, *Место-рождения* и *Дата-рождения*. Следовательно, получатся такие результаты: *Род-занятий*(*Альберт Эйнштейн, физик-теоретик*), *Место-рождения*(*Альберт Эйнштейн, Ульм*), *Дата-рождения*(*Альберт Эйнштейн, 14 марта 1879*).

У извлечения отношений есть целый ряд применений, поскольку оно описывает семантические связи между сущностями, а это помогает глубже понять текст. RE часто используется в вопросно-ответных системах, а также в системах реферирования. Так, в статье «Learning Surface Text Patterns for a Question Answering System» (см. Ravichandran [2002]) описана открытая вопросно-ответная система на основе образцов текста и обучения с частичным привлечением учителя. Например, для ответа на вопрос «Где родился Эйнштейн?» предлагаются образцы вида «$<$NAME$>$ родился в $<$LOCATION$>$». Это не что иное, как отношение {\bf место-рождения}({\it Эйнштейн, Ульм}), которое могла бы найти система извлечения отношений. Памятуя об этих примерах, рассмотрим несколько подходов к выявлению связей.

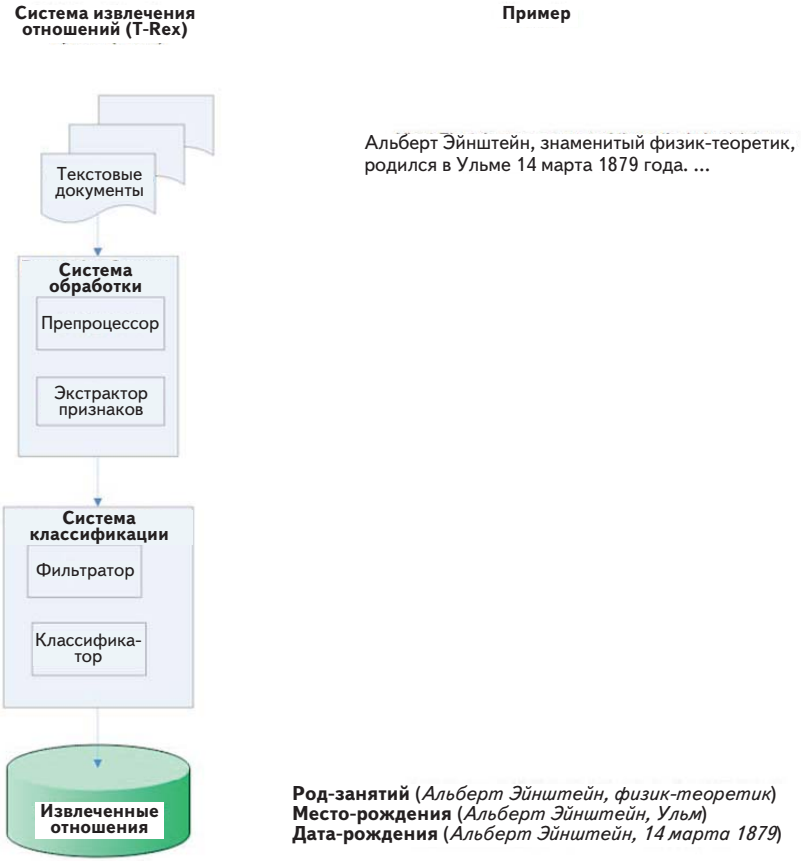

Рис. 9.2. T-Rex – пример системы извлечения отношений. На этом рисунке показаны несколько отношений, извлеченных RE-системой из предложения

9.3.1. Обзор имеющихся подходов

В области извлечения отношений проделана огромная работа. Предлагались решения на основе правил (см. Chu et al. [Chu, 2002] и Chen et al. [Chen, 2009]), когда для извлечения отношений применяются заранее определенные правила. Но для формулировки таких правил необходимо глубокое понимание предметной области. Подходы к RE можно разбить на четыре широкие категории: обучение с учителем, когда задача извлечения отношений ставится как задача двоичной

классификации после обучения на аннотированных данных; обучение с частичным привлечением учителя – главным образом, методы бутстрапинга; обучение без учителя, в том числе кластеризация, и подходы, выходящие за рамки бинарных отношений. Великолепный обзор различных методов извлечения отношений см. в «A Survey on Relation Extraction» (Nguyen [2007]).

Обучение с учителем

В этих подходах (например, Lodhi [2002]) извлечение отношений рассматривается как задача классификации. Дано множество помеченных положительных и отрицательных примеров отношений, и на нем требуется обучить двоичный классификатор.

Например, если имеется предложение, то можно воспользоваться средствами распознавания именованных сущностей типа имеющихся в проекте OpenNLP и рассмотренных ранее в этой книге. Имея эти сущности и связывающие их слова, мы можем обучить другой классификатор, который использует сущности для создания новой модели классификации, умеющей распознавать связи между сущностями. Подходы на основе обучения с учителем можно подразделить на методы на базе признаков и ядерные методы, исходя из данных, используемых для обучения классификатора.

Методы на базе признаков извлекают признаки из предложений; затем эти признаки используются для классификации. В них применяются различные способы извлечения синтаксических (Kambhatla [2004]) и семантических (GuoDong [2002]) признаков. К числу синтаксических признаков можно отнести именованные сущности, типы сущностей (например, человек, компания и т. д.), последовательность слов между сущностями, число слов между сущностями и другие. Примеры семантических сущностей: сущности в дереве разбора предложения, скажем, признак именной или глагольной группы. Затем эти признаки применяются для обучения классификатора. Естественно, не все признаки одинаково важны, и иногда трудно отыскать оптимальный набор признаков. Ядерные методы, которые мы рассмотрим ниже, – альтернативный подход, устраняющий зависимость от оптимального выбора признаков.

Ядром называется функция в многомерном пространстве, определяющая степень схожести двух объектов. Вообще говоря, для извлечения отношений применяются варианты строкового ядра (Lodhi [2002]). Наиболее распространены *ядро с мешком слов* и *сверточное ядро*. На рис. 9.3 приведен пример предложения и его представления с помощью различных ядер.

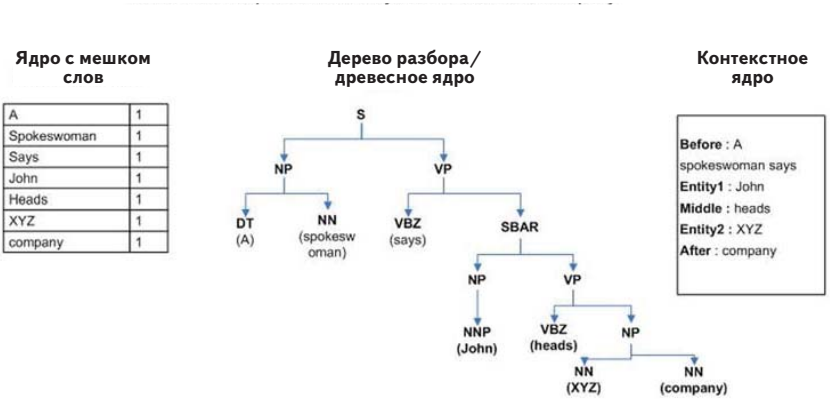

Рис. 9.3. Пример ядра с мешком слов и контекстного ядра

Ядро с мешком слов определяет, сколько раз каждое слово встречается в двух строках. Пример показан на рис. 9.3. Следуя работе «Subsequence Kernels for Relation Extraction» (Bunescu [2005]), мы можем выделить три подъядра. В примере, изображенном на рисунке, выявлены две сущности *John* и *XYZ*. *Контекстное ядро* определяет три интересующих нас участка контекста:

- *До* (Before) – слова, предшествующие сущности *John*;
- *Между* (Middle) – слова между сущностями *John* и *XYZ*;
- *После* (After) – слова, следующие после сущности *XYZ*.

Сверточные ядра (Zelenko [2003]) измеряют степень схожести между двумя структурированными экземплярами, суммируя степени схожести подструктур. Примером сверточного ядра является древесное ядро, которое определяет отношение между двумя сущностями с помощью сходства поддеревьев. В примере на рис. 9.3 отношение между сущностями *John* и *XYZ* вычисляется с помощью сходства между содержащими их деревьями.

Это очень лаконичное объяснение, дополнительные сведения можно найти в указанных источниках. Мы же перейдем к краткому описанию некоторых подходов с частичным привлечением учителя.

Обучение с частичным привлечением учителя

Методы обучения с учителем требуют большого объема обучающих данных, а также знания предметной области. Что же касается методов

с частичным привлечением учителя, то они позволяют использовать бутстрапинг для извлечения отношений из текстов в незнакомых предметных областях. Предлагавшиеся в литературе методы бутстрапинга можно подразделить на три типа. В первом (Blum [1998]) используется небольшой набор обучающих данных, называемых *затравочными* (seeds). С помощью затравочных данных и процесса бутстрапинга аннотируются новые данные. Во втором (Agichtein [2005]) предполагается предопределенный набор отношений. Для процесса обучения нужно небольшое количество обучающих данных. Для обнаружения новых примеров, содержащих определенные отношения, применяется итеративный двухшаговый процесс бутстрапинга. На первом шаге идентифицируются сущности и извлекаются образцы, а на втором они используются для выявления новых образцов. В третьем подходе (Greenwood [2007]) используется только группа документов, классифицированных как релевантные или нерелевантные конкретной задаче извлечения отношений. Из документа, входящего в релевантную или нерелевантную группу, извлекаются и соответственно ранжируются образцы. Эти ранжированные образцы затем используются в итеративном процессе бутстрапинга для выявления новых релевантных образцов, содержащих сущности.

Обучение без учителя

Подходам на основе обучения с учителем и с частичным привлечением учителя свойственны накладные расходы на адаптацию к новой предметной области и необходимость ориентироваться в ней. Эти расходы можно устранить, применяя методы обучения без учителя, в которых не требуется никаких обучающих примеров. В работе Хэчи и др. (Hachey [2009]) предложен подход, в котором схожесть и кластеризация используется для обобщенного извлечения отношений. Этот подход состоит из двух этапов: *идентификация отношений* и *характеризация отношений*. На этапе идентификации выявляются и извлекаются пары ассоциированных сущностей. Для этого используются различные признаки, в том числе окна совместной встречаемости, ограничения на сущности (например, допускаются только отношения человек-человек или человек-компания) и т. п. Сущности должны быть снабжены подходящими весами. На этапе характеризации отношения, получившие наивысшую оценку, кластеризуются, и для определения отношения используется метка кластера. Подробности см. в детальном отчете Хэчи.

9.3.2. Оценка

В 2000 году по инициативе Национального института стандартов и технологий США (NIST – http://www.nist.gov/index.html) была запущена программа Automatic Content Extraction с целью способствовать развитию технологий автоматического выведения смысла из текстовых данных. В этой программе обозначено пять задач распознавания: сущностей, значений, выражений, относящихся к времени, отношений и событий. К задачам прилагаются размеченные данные и объективные данные (аннотированные метками).

С момента появления эти данные широко использовались для оценки методов извлечения отношений, в том числе и большинства рассмотренных в этом разделе методов обучения с учителем. В числе прочих данные содержат такие типы отношений, как организация-место, организация-членство, гражданин-резидент- религия-этническая принадлежность, спорт-принадлежность и т. д. Еще одним богатым источником данных является википедия (http://www.wikipedia.org), поскольку на большинстве страниц имеются сущности, связанные гиперссылками. Эти данные использовались для извлечения отношений в работах Culotta [2006], Nguyen [2007] и других. В более ранних работах по медицине и биологии использовались наборы данных BioInfer (http://mars.cs.utu.fi/BioInfer/) и MEDLINE (см. PubMed).

Наиболее употребительными метриками для оценки качества алгоритма являются точность, полнота и F-мера. Точность и полнота были определены в главе, посвященной поиску. F-мера – это простая формула, в которой точность и полнота объединены в единый показатель (подробнее см. http://en.wikipedia.org/wiki/F1_score). Большая часть вышеупомянутых подходов на основе обучения с частичным привлечением учителя работают с большими объемами данных. Поэтому вычисляется только оценка точности – с помощью извлеченных отношений и имеющихся объективных данных. Для большого набора данных вычислить полноту трудно из-за сложности получения фактического числа отношений.

9.3.3. Инструменты для извлечения отношений

В этом разделе мы перечислим несколько наиболее известных инструментов для извлечения отношений. Выше уже отмечалось, что в системе T-Rex есть два этапа: обработка и классификация. На этапе

обработки входной текст преобразуется в признаки. На этапе классификации сначала извлекаются признаки, а затем на них обучается классификатор. Систему можно скачать со страницы проекта T-Rex по адресу http://sourceforge.net/projects/t-rex/.

В программе Java Simple Relation Extraction (JSRE) используется комбинация ядерных функций для интеграции двух источников информации: предложения, в котором встречается отношение, и контекста, окружающего сущности. Скачать дистрибутив можно со страницы проекта JSRE по адресу http://hlt.fbk.eu/en/technology/jSRE.

В библиотеке обработки естественных языков NLTK (Natural Language Toolkit), написанной на Python, используется двухпроходный алгоритм извлечения отношений. На первом проходе обрабатывается текст и извлекаются кортежи, состоящие из контекста и сущности. Контекстом могут быть слова до и после сущности (контекст может быть и пустым). На втором этапе обрабатываются тройки пар и вычисляется бинарное отношение. Разрешается также задавать ограничения на сущности, например, *организация* и *человек*, и на тип отношения, например, *юрист*, *руководитель* и т. д. Скачать NLTK можно со страницы проекта по адресу http://code.google.com/p/nltk/.

Как и во всех разделах этой главы, мы видели лишь вершину айсберга, каковым является задача извлечения отношений. Надеемся, вы уже начали задумываться о том, как применить эти методы в своем приложении. А мы оставим эту тему и перейдем к алгоритмам выявления важных идей и причастных к ним людей.

9.4. Выявление важного содержимого и людей

Принимая во внимание взрывной рост объема информации, социальных сетей и гиперсвязности современного мира, все большее значение приобретает выделение содержимого и людей в соответствии с некоторым определением приоритета или важности. В электронной почте уже сравнительно давно реализовано распознавание спама (фильтрация нежелательных сообщений), но только сейчас мы начинаем использовать компьютеры, для того чтобы понять, какие сообщения важны. Например, компания Google недавно включила в состав службы Gmail функцию Priority Inbox (см. рис. 9.4), которая призвана отделить важные сообщения от несущественных.

В социальных сетях типа Facebook и Twitter было бы желательно иметь возможность поднимать сообщения от людей, которых вы высо-

ко цените, и проставлять более низкий приоритет сообщениям, которые можно прочитать позже или вообще не читать. Понятие важности можно приписать также словам, сайтам и многому другому. В качестве последнего примера представьте, что вы ежедневно получаете сводку новостей и можете сосредоточиться только на тех, которые важны для работы, или что имеете возможность легко и быстро находить самые важные статьи по новой области научных исследований (уж мы-то сумели бы применить такой инструмент при написании этой главы!).

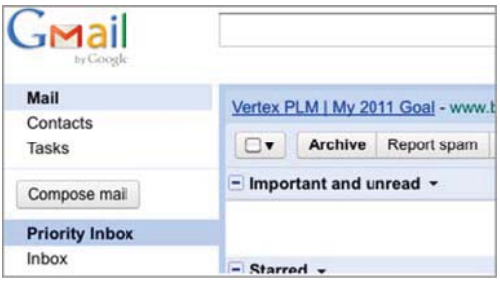

Рис. 9.4. Функция Priority Inbox в Google Mail пытается автоматически определить, что важно для пользователя. Снимок сделан 3.01.2011

Вопрос о важности труден, подходы к решению задачи зависят от приложения и зачастую от пользователей. К вопросу о важности примыкают и такие области, как ранжирование и релевантность, а также авторитетность. Основное различие заключается в том, что важное является также и релевантным, однако не все, что релевантно, можно назвать важным. Например, информация о противоядии и о том, где его достать, для человека, только что случайно проглотившего что-то ядовитое, гораздо важнее, чем сведения о способе производства противоядий. К сожалению, ответ на вопрос о том, где проходит черта, отделяющая релевантное от важного, неоднозначен и, вообще говоря, субъективен.

Когда мы писали этот раздел и изучали соответствующую литературу, стало ясно, что пока еще никакой «теории важности» нет, а есть растущий массив работ, посвященных решению конкретных задач, связанных с важностью. Существуют также связанные проблемы в теории информации, например непредвиденные ситуации (см. http://en.wikipedia.org/wiki/Self-Information), взаимная информация (см. http://en.wikipedia.org/wiki/Mutual_Information) и другие. Полезно различать два уровня важности: глобальный и персональный. Понятие глобальной важности относится к тому, что считает важным большинство членов группы, а персональной – к тому, что кажется важным лично вам. Существуют алгоритмы, работающие для каждого уровня, мы рассмотрим их ниже.

9.4.1. Глобальная важность и авторитетность

Наверное, самое масштабное приложение, в котором используется глобальная важность (или, по крайней мере, делается такая попытка), – это поисковая система Google (http://www.google.com). В ней голосование миллионов пользователей (в виде переходов по ссылкам, вводимых слов, щелчков мышью и т. п.) служит для определения самых важных (авторитетных) сайтов (и релевантных тоже) относительно конкретного запроса пользователя. Подход Google, который был назван *PageRank* и в своей первоначальной форме описан в статье «The Anatomy of a Large-Scale Hypertextual Web Search Engine» (http://infolab.stanford.edu/~backrub/google.html), представляет собой сравнительно простой итеративный алгоритм, который «соответствует главному собственному вектору нормированной матрицы ссылок в веб».

> **Примечание.** Хотя эта тема выходит за рамки книги, мы настоятельно рекомендуем любознательным читателям почитать о собственных векторах и вообще о математической теории матриц, поскольку они сплошь и рядом встречаются в различных алгоритмах NLP, машинного обучения и поиска. Для начала подойдет любой солидный учебник линейной алгебры.

Идея насчет собственных векторов нашла применение и в других областях. Так, аналогичные подходы используются при извлечении ключевых слов (TextRank – www.aclweb.org/anthology-new/acl2004/emnlp/pdf/Mihalcea.pdf), для многодокументного реферирования (LexRank – http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/jair/pub/volume22/erkan04a.pdf), а также в других стратегиях графового ранжирования (Грант иногда называет их *Rank-стратегиями). Подобные подходы часто применяются и для понимания динамики социальных сетей, поскольку они позволяют быстро находить важных людей и связи. Итеративный алгоритм просто реализуется и, по счастью, сравнительно легко масштабируется. Более сложной частью задачи является первоначальный сбор данных в большом масштабе.

9.4.2. Персональная важность

Определить, что важно для отдельного человека, во многих отношениях труднее, чем вычислить глобальную важность. Во-первых, важное для пользователя A вовсе необязательно важно и для пользователя B. Во-вторых, в зависимости от приложения ошибка в определении (как

ложноположительный, так и ложноотрицательный вывод) может дорого стоить. У приложений для определения персональной важности имеются также проблемы с бутстрапингом – ведь сложновато понять, что человек считает важным, если он никогда не взаимодействовал с системой!

На сегодняшний день в большинстве подобных приложений проблема рассматривается как задача классификации, зачастую в варианте, который обсуждался в книге ранее. Приложение обучает и тестирует n+1 модель, где n – число пользователей системы. Иными словами, по одной модели на каждого пользователя плюс дополнительная глобальная модель, обученная на глобальных признаках, имеющих отношение к важности. В пользовательских моделях обычно отражаются только отличия от глобальной модели, что позволяет сделать их более компактными и масштабируемыми. О том, как эта идея применяется на практике, см. статью Aberdeen, et al. «The Learning Behind Gmail Priority Inbox» по адресу http://research.google.com/pubs/archive/36955.pdf.

9.4.3. Ресурсы и ссылки на тему важности

К сожалению, как уже отмечалось выше, не существует единого места, где можно было бы найти теоретические основы и концепции, относящиеся к важности, – и это отличает данную тему от большинства рассматриваемых в книге. Даже в википедии, откуда принято начинать изучение многих предметов в век Интернета, нет почти ничего, кроме заготовки страницы со словом *importance* (по крайней мере, так было 21 января 2001 года[4]; см. http://en.wiktionary.org/wiki/importance). Но интересующимся читателям не стоит отчаиваться, поскольку есть много мест, содержащих кусочки, из которых можно составить целостную картину.

- Статья Google об алгоритме PageRank (http://infolab.stanford.edu/~backrub/google.html) – неплохая отправная точка для понимания того, что такое авторитетность и стратегии графового ранжирования.
- Любой достойный учебник по теории информации (начните с википедии – http://en.wikipedia.org/wiki/Information_theory). Полезно будет почитать о взаимной информации, энтропии, непредвиденных ситуациях, приросте информации и смежных темах.

[4] С тех пор ситуация не изменилась. – *Прим. перев.*

Работы по важности и расстановке приоритетов, без сомнения, находятся на переднем крае науки в силу их значимости для социальных сетей и борьбы с затоплением информацией. Активные исследования ведутся также в области анализа тональности высказываний – нашей следующей темы. Не в последнюю очередь это связано с быстрым ростом таких служб, как Facebook и Twitter.

9.5. Распознавание эмоций с помощью анализа тональности

Дж. Нил Рихтер и Роб Зинков

Анализ тональности (применяется также термин *opinion mining* – выявление мнений) – это выявление и извлечение из текста субъективной информации. Для автоматизации этого процесса применяются различные инструменты NLP и программного анализа текста. Следующий простой пример взят с сайта рецензирования фильмов RottenTomatoes.com и для большей ясности немного перефразирован.

Фильм «Поле битвы: Земля» – неудача эпического масштаба!

– Дастин Путнэм

Очевидно, что это отрицательная рецензия. Базовый вид анализа тональности – классификация эмоциональной оценки, по этому критерию предложению может быть присвоена оценка -5 из нормированного диапазона [-10,10]. В более сложных способах анализа предложение подвергается разбору, в результате чего выводятся следующие факты:

- «Поле битвы: Земля» – фильм.
- «Поле битвы: Земля» – очень плохой фильм.
- Дастин Путнэм думает, что «Поле битвы: Земля» – очень плохой фильм.

Трудность задачи также очевидна. Программа должна распознать сущности {Поле битвы: Земля, Дастин Путнэм} и иметь базу данных словосочетаний, в которых слову «неудача» сопоставлена отрицательная оценка. Она должна также понять, что дополнение «гигантского масштаба» выступает в роли прилагательного к существительному «неудача» и в первом приближении эквивалентно фразе «большая неудача», а потому усиливает негативную оценку, уже ассоциирован-

ную со словом «неудача». В связи с быстрым ростом генерируемого пользователями сети содержимого теперь стало возможно измерять мнения большого числа людей о различных темах (политика, фильмы) и предметах (конкретных изделиях). Желание выявить, проиндексировать и подытожить эти мнения очень сильно у маркетологов, корпоративных отделов обслуживания клиентов, а также у финансовых, политических и государственных организаций.

9.5.1. Исторический обзор

В статьях Pang [2008] и Liu [2004] приведен прекрасный обзор истории и современного состояния анализа тональности. Эта дисциплина имеет глубокие корни в сообществах, связанных с NLP и лингвистикой. Первые работы в этой области были выполнены Джанис Вибе с сотрудниками (1995-2001), тогда они применяли название *анализ субъективности* (subjectivity analysis). Целью исследования было классифицировать предложение как субъективное или нет в зависимости от использованных прилагательных и их эмоциональной окраски.

В 2000-м году произошло интересное событие в анализе тональности. Компания Qualcomm включила в новую версию почтового клиента Eudora функцию Mood Watch (мониторинг настроения). Эта функция выполняла простой анализ отрицательной эмоциональной окраски почтовых сообщений, выставляя оценку в диапазоне [–3, 0], которая на экране была представлена стручками жгучего перца. Систему спроектировал Дэвид Кауфер (David Kaufer) из Британского отделения Университета Карнеги-Меллон (Kaufer [2000]). На внутреннем уровне алгоритм помещал каждое сообщение в одну из восьми категорий типичных речевых образцов, встречающихся в перепалках в сети Usenet.

В начале 2000 года Нил Рихтер (Neal Richter) первым начал работать над системой анализа эмоциональной окраски для компании-производителя CRM-систем, специализирующейся на обработке электронной почты для отдела обслуживания клиентов. В то время использовался термин *измерение эмоциональности* (affective rating), а не анализ тональности (Durbin [2003]). Система была принята в эксплуатацию в 2001 году и с тех пор обрабатывает сотни миллионов запрос в месяц и переведена на 30 с лишним языков. И хотя и раньше были отдельные академические работы на эту тему, 2001 год можно считать годом становления анализа тональности как самостоятельной дисциплины в лоне обработки естественных языков.

Поначалу в центре исследований было использование базовых правил грамматики и конструкций английского языка в сочетании со словарем ключевых слов с целью дать эвристическую оценку эмоциональной окраски фрагмента текста. Словарь ключевых слов обычно создавался вручную с привлечением экспертных оценок эмоциональной окраски. Эти методы позволяют добиться приемлемой точности; например, Тэрни (Turney [2002]) сумел получить точность 74 % при оценке эмоциональной окраски отзывов о товарах. Такие методы легко реализовать, но они не улавливают присутствующие в английском языке конструкции высшего порядка. В частности, они не всегда учитывают, что слова способны изменять смысл окружающих слов, причем иногда на другом конце предложения. Поскольку такие алгоритмы извлекают лишь короткие словосочетания (два-три слова), более сложные конструкции они не видят. У них зачастую низкая полнота, они не способны к классификации.

Ограничением эвристических алгоритмов является проблема перечисления. Принимая во внимание сложную природу языка, было бы бесплодной затеей пытаться вручную сконструировать все возможные способы выражения эмоций. Поэтому на следующем этапе исследователи воспользовались постоянно растущим количеством отзывов о товарах в Интернете, чтобы вывести эмоциональные образцы или просто выдать предсказание. Один из ранних методов обучения с учителем описан в работе «Mining the Peanut Gallery: Opinion Extraction and Semantic Classification of Product Reviews» (Dave [2003]). Начав с простых приемов стемминга и предобработки, с использования TF-IDF, преобразования Лапласа и аналогичных метрик, авторы подали обработанные и оцененные отзывы на вход различных классификаторов. При использовании наивного байесовского алгоритма для бинарной классификации удалось добиться точности 87 % на конкретном наборе данных.

Более поздний высококачественный эвристический подход описан в работе Ding [2009], которую мы горячо рекомендуем. Авторы соединили большой лексикон слов, выражающих мнения, с углубленной предобработкой и учетом перехода состояний. В частности, они применили грамматику на основе правил (в смысле, употребляемом при разработке компиляторов) для логического вывода. Механизм правил многократно применяется, чтобы преобразовать размеченный текст в предложения вывода, определяющие ассоциацию между парой (слово, часть речи) и эмоциональной окраской. Им удалось добиться 80%-й полноты и точности.

Недавно простой подход к обучению без учителя был применен Хассаном и Радевом (Hassan [2010]). Они взяли за основу бутстрапинга базу данных WordNet и выполнили «случайное блуждание», начав со слова с неизвестной эмоциональной окраской. Блуждание останавливалось по достижении слова с известной окраской. Предсказание окраски усреднялось по нескольким случайным блужданиям, начинавшимся с одного и того же слова. Никаких попыток задействовать конструкции более высокого уровня не предпринималось. Метод быстрый; не требуется никакого корпуса, кроме сравнительно короткого списка термов с положительной и отрицательной эмоциональной окраской, составляющего «золотой стандарт». При очень небольшом списке начальных затравочных слов удалось добиться правильности 92–99%. Учитывая, что не привлекается корпус текстов, а WordNet содержит не все термы английского языка, эту идею лучше, пожалуй, всего использовать для бутстрапинга словаря ключевых слов, полезных при анализе тональности.

9.5.2. Инструменты и данные

В большинстве подходов к анализу тональности необходимо всего несколько простых инструментов и данные. Первый из них – *частеречный разметчик*, программное средство, которое производит разбор предложения и каждому слову сопоставляет метку, показывающую, какой частью речи оно является. Перечислим несколько общеупотребительных разметчиков.

- Разметчик OpenNLP, был рассмотрен в книге выше.
- Разметчик Эрика Брилла (Eric Brill) – http://gposttl.sourceforge.net/ (написан на C).
- Lingua-EN-Tagger – http://search.cpan.org/~acoburn/Lingua-EN-Tagger/Tagger.pm.
- Разметчик Illinois – http://cogcomptest.cs.illinois.edu/page/software_view/3.
- Demo – http://cogcomp.cs.illinois.edu/demo/pos/.

Помимо частеречного разметчика, необходима база данных с оценками эмоциональной окраски ключевых слов и словосочетаний. Их можно извлечь из аннотированных источников или путем обучения на корпусе текстов. Два известных источника данных – словарь Виссела (Whissell) «Dictionary of Affective Language» (DAL) (см. http://hdcus.com/ и http://www.hdcus.com/manuals/wdalman.

pdf) и словарь WordNet-Affect (см. http://wndomains.fbk.eu/wnaffect.html и http://www.cse.unt.edu/~rada/affectivetext/). Отметим, что для использования данных из DAL придется поработать над их извлечением. Семантический словарь может выглядеть, как показано в табл. 9.1.

Таблица 9.1. Пример семантического словаря

Словарная статья	Часть речи	Эмоциональная категория (+ положительно, – отрицательно)
happy	JJ	+
horror	NN	–
dreadful	JJ	–
fears	VBZ	–
loving	VBG	+
sad	JJ	–
satisfaction	NN	+

Еще необходимы базовые средства анализа текстов: лексического анализа, определения границ предложений, пометки стоп-слов, стемминга и т. д. Подробнее эта тема рассматривается в главе 2.

9.5.3. Базовый алгоритм определения тональности

Тональность правильнее всего представлять как поток, протекающий через весь документ. Следует ожидать, что тональность документа будет оставаться постоянной на всем его протяжении. Возьмем, к примеру, отзыв о блендере на сайте Amazon.

> *Хорошая цена и хорошо работает, только очень громко. Кроме того, при первых 10–15 запусках ощущался выраженный запах электрической изоляции. Но в общем-то сколько платишь, столько и получаешь. Устройство работает, перемешивает хорошо и мне нравится, что кувшин съемный.*

Обратите внимание, что тональность меняется с положительной на отрицательную на слове «только». А потом снова возвращается к положительной на слове «Но». Такой переход позволяет сделать вывод, что в контексте этого отзыва «громко» окрашено отрицательно. Тональность слова «громкий» обычно зависит от документа. При обсуждении рок-концерта оно, как правило, несет положительную ок-

раску. А при обсуждении блендера, скорее, отрицательную. Первые алгоритмы определения эмоциональной окраски были предложены в работах И и др. (Yi [2003]) и Тэрни (Turney [2002]). Ниже описан алгоритм И, перефразированный для простоты реализации.

- Разбить текст на лексемы.
- Найти границы предложений с помощью эвристического или иного подхода.
- Произвести частеречную разметку каждого предложения.
- Применить к каждому предложению набор образцов для выделения глагольных и именных групп.
- Сконструировать все обнаруженные бинарные и тернарные выражения.
- Найти глаголы и прилагательные и их модификаторы в семантической базе данных – при необходимости использовать варианты, подвергнутые стеммингу.
- Вывести обнаруженные ассоциативные выражения.

Примером тернарного выражения (объект, глагол, источник) может служить «the burrito», «was», «too messy», а бинарного (прилагательное, объект) – «quality», «camera». Вот полный пример применения алгоритма И к тексту «This recipe makes excellent pizza crust»[5].

- Подходящий образец тональности – <«make» ГД ГП>.
- Группа подлежащего (ГП) – This recipe.
- Группа дополнения (ГД)–pizza crust.
- Тональность ГД – положительная.
- Т-выражение – <«recipe», «make», «excellent pizza crust»>

Алгоритм Тэрни – еще один пример простого экстрактора на основе образцов, в котором метрика PMI (межточечная взаимная информация) применяется для аппроксимации эмоциональной окраски ключевого слова в зависимости от его близости к известным словам с положительной и отрицательной окраской в большом корпусе текстов. Попросту говоря, согласно PMI два терма считаются тесно связанными, если они встречаются совместно чаще, чем можно было бы предсказать на основе индивидуальных частот. В остальных отношениях этот алгоритм аналогичен предыдущему: предложения разбиваются на лексемы и подвергаются частеречной разметке, после чего извлекаются помеченные фразы из двух и трех слов

[5] По этому рецепту у пиццы получается отличная корочка. – *Прим. перев.*

(биграммы и триграммы), соответствующие образцам. Затем метод PMI применяется для классификации слов на основе их совместной встречаемости с любым известным положительно или отрицательно окрашенным словом из выделенной биграммы или триграммы.

9.5.4. Дополнительные темы

Один из наиболее мощных способов анализа тональности – применение техники *условных случайных полей* (CRF; см. Getoor [2007]). Основное преимущество CRF и других методов, управляемых данными, – их способность моделировать зависимости между словами и грамматические конструкции. Поэтому они характеризуются значительно лучшей полнотой. Во многих из ранее рассмотренных методов эмоциональная окраска каждого терма рассматривается вне зависимости от его окружения. Для многих задач обработки естественных языков это не так уж важно. Но, к сожалению, на эмоциональную окраску слова окружающие слова влияют очень сильно.

Условные случайные поля позволяют моделировать данные с помощью внутренних структур, которые налагают ограничения на метки, допустимые для термов. В задаче классификации тональности с помощью CRF можно воспользоваться латентной структурой естественного языка и таким образом достичь выдающихся результатов. Детали алгоритмов обучения и вывода выходят за рамки настоящей книги. Лучше мы сосредоточим внимание на кратком объяснении того, как определяются признаки для CRF. Для обучения моделей можно будет затем взять имеющиеся инструменты с открытым исходным кодом. Определяя CRF, мы указываем, какие признаки хотели бы принимать в расчет при вычислении эмоциональной окраски терма. Годятся некоторые из ранее рассмотренных типичных признаков.

- Какой частью речи является слово.
- Является ли слово именованной сущностью?
- Описывает ли слово именованную сущность?
- Синонимы слова.
- Корень слова.
- Начинается ли слово с заглавной буквы?
- Само слово.

Далее нужно учесть признаки, не относящиеся непосредственно к данному слову, например:

- Какими частями речи являются предыдущее и последующее слово.
- Является ли предыдущим словом *not* (не)?
- Является ли предыдущим словом *but* (но)?
- Если слово является анафорой, то признаки слова, на которое оно ссылается.

Но еще важнее то, что теперь мы можем связывать признаки; для этого нужно определить признаки, у которых есть элементы текущего терма, окружающих термов и даже далеко отстоящих термов, например, анафор. Так, можно определить в качестве признака часть речи текущего слова и слово, на которое ссылается анафора, если текущее слова является таковой. Выделенные признаки мы подаем на вход инструмента, например CRF++ (http://crfpp.sourceforge.net/). Имейте в виду, что придется еще выполнить предобработку, чтобы представить признаки в формате, который понимает CRF++. Для использования обученной модели понадобится кое-какая постобработка, но никакой черной магии в ней нет.

Еще одна типичная операция – агрегирование мнений, выраженных в нескольких термах или документах. В таких ситуациях агрегирование может производиться по документам или по сущностям, на которые есть ссылки в документе. Например, нас может интересовать усредненное мнение людей о компании Пепси. Для этого можно усреднить тональности всех сущностей, относящихся к Пепси или ссылающихся на Пепси.

Во многих случаях нам нужно нечто не столь четко определенное. Мы хотим знать тематику документа. Существуют естественные кластеры термов, о которых люди имеют связное и непротиворечивое мнение. Такие кластеры иногда называются *темами* (topic).

Под тематическим моделированием (с которым мы кратко познакомились в главе 6) понимается семейство алгоритмов, предназначенных для поиска таких кластеров в наборе документов. Они, как правило, собирают важные встречающиеся в документах термы в различные темы, репрезентативные для обрабатываемых документов. Если в вашей предметной области имеет смысл сопоставить темам тональность, то сделать это можно множеством способов. Самый простой – вычислить взвешенное среднее тональностей слов, приписав слову вес, зависящий от того, насколько сильно оно связано с темой. Можно также проводить одновременное обучение тональностям и темам. Тогда при выборе тем естественным образом будут

использоваться слова со схожими тональностями. Интуиция подсказывает, что все слова, относящиеся к теме, должны иметь примерно одинаковую тональность. Подходы, в которых находит применение эта идея, называются *моделями тем и тональностей* (sentiment-topic models).

9.5.5. Библиотеки с открытым исходным кодом для анализа тональности

Для построения моделей анализа тональности можно использовать различные библиотеки классификации (напомним, что классификация – это общий принцип, не «заточенный» специально под задачу анализа тональности), но есть несколько, которые специально настроены для этого, в том числе:

- *GATE* (http://gate.ac.uk/) – универсальная библиотека NLP, распространяемая по лицензии GPL, которая содержит в том числе и модуль анализа тональности.
- *Balie* (http://balie.sourceforge.net/) – библиотека, распространяемая по лицензии GPL, в которой реализованы распознавание именованных сущностей и анализ тональности.
- *MALLET* (http://mallet.cs.umass.edu/) – библиотека, распространяемая по лицензии Common Public, в которой наряду с другими алгоритмами реализованы условные случайные поля.

Для анализа тональности существует также ряд коммерческих инструментов (например, Lexalytics), API (например, Open Dover) и библиотек с условно открытым исходным кодом (например, LingPipe). Как обычно, прежде чем принимать решение о покупке или использовании некоторого продукта для разработки приложения, убедитесь, что у вас есть инструменты для проверки качества результатов.

И хотя анализ тональности сейчас (в 2012 году) является одной из самых активных областей исследования, все возрастающая связность мира (только в Facebook на момент написания этой книги было более 500 миллионов пользователей чуть ли не со всего земного шара) влечет за собой необходимость более эффективно преодолевать языковые барьеры. Наша следующая тема, межъязыковой поиск, – важнейшая составная часть решения, призванного упростить общение людям, говорящим на разных языках.

9.6. Межъязыковой информационный поиск

Система *межъязыкового информационного поиска* (cross-language information retrieval, CLIR) позволяет пользователям вводить запросы на одном языке, а получать результаты на разных языках. Например, китаец, не говорящий по-английски, может ввести запрос на китайском и получить релевантные документы на английском, испанском или любом другом поддерживаемом языке. И хотя, на первый взгляд, эти документы бесполезны для человека, не говорящего на соответствующем языке, в большинство реальных CLIR-систем встроен тот или иной компонент перевода (машинного или ручного) для просмотра документов на родном языке пользователя.

> **Межъязыковой и многоязыковой поиск.** В кругу специалистов можно услышать оба выражения: «многоязыковой поиск» и «межъязыковой поиск». На наш взгляд, говоря о многоязыковом поиске, имеют в виду приложение, в котором есть несколько индексов, соответствующих разным языкам, а пользователь запрашивает только документы на выбранном им языке (англоязычные пользователи получат ресурсы на английском, испаноязычные – на испанском и т. д.). В CLIR-системе пользователь явно просит преодолеть языковой барьер, например, англоговорящий запрашивает ресурсы на испанском.

В дополнение ко всем сложностям, присущим одноязычному поиску (см. главу 3), CLIR-система должна иметь дело с языковым барьером. Учитывая, как трудно большинству людей дается изучение нового языка, должно быть понятно, что хороший межъязыковой поиск – отнюдь не тривиальная задача.

CLIR-приложения традиционно строятся одним из двух способов: либо запрос переводится на целевой язык, а затем производится поиск документов на целевом языке, либо во время предобработки все документы в наборе переводятся с целевого языка на исходный.

Но при любом подходе качество системы зависит от ее умения выполнять перевод. Во всех системах, кроме совсем уж тривиальных, ручной перевод следует сразу исключить из рассмотрения, так что необходимо прибегнуть к той или иной форме автоматизированного перевода. Простейший программный подход – взять двуязычный словарь и произвести пословную подстановку, но тут сразу же возникают проблемы неоднозначности и семантики, поскольку в большинстве языков имеются идиомы, синонимы и другие конструкции, которые, мягко говоря, затрудняют дословный перевод.

Существует несколько инструментов – как коммерческих, так и с открытым исходным кодом – в которых более высокое качество машинного перевода обеспечивается применением статистических подходов на основе анализа параллельных или сравнимых корпусов, позволяющего автоматически обучаться идиомам, синонимам и другим конструкциям. (Параллельными называются корпусы, в которых каждому документу соответствует его перевод. Сравнимыми корпусами называются два набора документов на одну и ту же тему.) Самая известная система автоматизированного перевода – онлайновый переводчик Google, размещенный по адресу http://translate.google.com и показанный на рис. 9.5, но есть и другие, например, Systran (http://www.systransoft.com/) и SDL Language Weaver (http://www.languageweaver.com/). Из ПО с открытым исходным кодом назовем активно развивающийся проект Apertium (http://www.apertium.org/), который мы, правда, не оценивали. Проект Moses (http://www.statmt.org/moses/) – это статистическая система *машинного перевода* (MT), которой для построения модели нужны только параллельные корпусы текстов. Наконец, Микель Форкада (Mikel Forcada) составил изрядный перечень бесплатных и открытых систем машинного перевода и разместил его по адресу http://computing.dcu.ie/~mforcada/fosmt.html.

Рис. 9.5. Пример перевода предложения «Tokyo is located in Japan» с английского на японский с помощью Google Translate. Снимок сделан 30.12.2010

В некоторых случаях прямой перевод для данной пары языков не поддерживается, поэтому приходится переводить через общий промежуточный язык (если такой существует). Например, если существуют ресурсы для перевода с английского на французский и с французского на кантонский диалект китайского, но не напрямую с английского на кантонский, то для перевода нужно будет использо-

вать французский как промежуточный язык. Разумеется, качество перевода пострадает, но это все же лучше, чем ничего.

Даже при наличии хорошего движка перевода (а большинство из них дают возможность получить общее представление о смысле текста) часто порождается несколько результатов для одного запроса, поэтому у CLIR-системы должны быть средства, чтобы определить, когда нужно переводить, а когда нет. Кроме того, во многих языках необходима транслитерация имен собственных (представление одного алфавита символами другого). Например, в арабо-английской CLIR-системе, над которой работал Грант, транслитерация английских имен на арабский и наоборот нередко была возможна многими, иногда сотнями, способов, и надо было, исходя из статистической вероятности появления результата в корпусе, решить, какие варианты включать в качестве поисковых термов.

> **Примечание.** Если вы недоумевали, почему имя бывшего ливийского лидера Муамара Каддафи по-английски пишется то Gaddafi, то Khadafi, а то Qaddafi, так это из-за неоднозначности транслитерации, поскольку не существует единственно правильного способа сопоставления алфавитов.

Во многих случаях система перевода возвращает оценку доверия, но иногда приложение вынуждено полагаться на обратную связь с пользователями или анализ журналов. Наконец, и это очень печально, нередко бывает так, что поисковая часть CLIR-системы реализована замечательно, но пользователь судит о ней по качеству автоматизированного перевода результатов, которое почти всегда оставляет желать лучшего, даже если смысл документа удается разобрать – немного поднаторев в этом.

Для дальнейшего изучения межъязыкового информационного поиска можно начать с книги Grossman, Frieder «Information Retrieval» (Grossman [2004]) и с сайта Дуга Оурда (Doug Oard) по адресу http://terpconnect.umd.edu/~oard/research.html. Есть также несколько конференций и конкурсов (аналогичных TREC), посвященных CLIR, в том числе CLEF (Cross-Language Evaluation Forum) (http://www.clef-campaign.org) и NTCIR (http://research.nii.ac.jp/ntcir/index-en.html).

9.7. Резюме

В этой главе мы поверхностно затронули ряд тем в области поиска и обработки естественного языка. Мы начали с семантики и задачи об автоматическом понимании смысла текста, а затем перешли к таким

разноплановым вопросам, как реферирование, определение важности, межъязыковой поиск и распознавание событий и отношений. К сожалению, нет ни времени, ни места, чтобы углубиться в эти вопросы, но мы надеемся, что приведенных нами ссылок будет достаточно для желающих продолжить изыскания. Как видите, поиск и NLP изобилуют сложными задачами, которые могут стать основой для карьеры, посвященной интересной работе. Глубокое проникновение в любую из описанных дисциплин откроет путь к новым приложениям и возможностям. Мы, как авторы, искренне надеемся, что изложенный в книге материал поможет вам как в работе, так и в жизни. Успехов в приручении текста!

9.8. Ресурсы

Agichtein, Eugene. 2006. «Confidence Estimation Methods for Partially Supervised Relation Extraction». Proceedings of the 6th SIAM International Conference on Data Mining, 2006.

Blum, Avrim and Mitchell, Tom. 1998. «Combining Labeled and Unlabeled Data with Cotraining». Proceedings of the 11th Annual Conference on Computation Learning Theory.

Bunescu, Razvan and Mooney, Raymond. 2005. «Subsequence Kernels for Relation Extraction». Neural Information Processing Systems. Vancouver, Canada.

Chen, Bo; Lam, Wai; Tsang, Ivor; and Wong, Tak-Lam. 2009. «Extracting Discriminative Concepts for Domain Adaptation in Text Mining». Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Chu, Min; Li, Chun; Peng, Hu; and Chang, Eric. 2002. «Domain Adaptation for TTS Systems». Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP).

Culotta, Aron; McCallum, Andrew; and Betz, Jonathan. 2006. «Integrating Probabilistic Extraction Models and Data Mining to Discover Relations and Patterns in Text». Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics.

Dave, Kushal; Lawrence, Steve; and Pennock, David. 2003 «Mining the Peanut Gallery: Opinion Extraction and Semantic Classification of Product Reviews». Proceedings of WWW-03, 12th International Conference on the World Wide Web.

Ding, Xiaowen; Liu, Bing; and Xhang, Lei. 2009. «Entity Discovery and Assignment for Opinion Mining Applications». Proceedings of ACM SIGKDD Conference (KDD 2009). http://www.cs.uic.edu/~liub/FBS/KDD2009_entity-final.pdf.

Durbin, Stephen; Richter, J. Neal; Warner, Doug. 2003. «A System for Affective Rating of Texts». Proceedings of the 3rd Workshop on Operational Text Classification, 9th ACM SIGKDD International Conference.

Getoor, L. and Taskar, B. 2007. Introduction to Statistical Relational Learning. The MIT Press. http://www.cs.umd.edu/srl-book/.

Greenwood, Mark and Stevenson, Mark. 2007. «A Task-based Comparison of Information Extraction Pattern Models.» Proceedings of the ACL Workshop on Deep Linguistic Processing.

Grossman, David A., and Frieder, Ophir. 2004. Information Retrieval: Algorithms and Heuristics (2nd Edition). Springer.

GuoDong, Zhou; Jian, Su; Zhang, Jie; and Zhang, Min. 2002. «Exploring Various Knowledge in Relation Extraction.» Proceedings of the Association for Computational Linguistics.

Hachey, Ben. 2009. «Multi-document Summarisation Using Generic Relation Extraction». Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1.

Hassan, Ahmed and Radev, Dragomir. 2010. «Identifying Text Polarity Using Random Walks». Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL). http://www.aclweb.org/anthology-new/P/P10/P10-1041.pdf.

Hearst, Marti. 1994. «Multi-Paragraph segmentation of Expository Text». Proceedings of the Association for Computational Linguistics.

Jurafsky, Danile, and Martin, James. 2008. Speech and Language Processing, 2nd Edition. Prentice Hall.

Kambhatla, Nanda. 2004. «Combining Lexical, Syntactic, and Semantic Features with Maximum Entropy Models for Extracting Relations». Proceedings of the Association for Computational Linguistics. http://acl.ldc.upenn.edu/P/P04/P04-3022.pdf.

Kaufer, David. 2000. «Flaming: A White Paper». Carnegie Mellon. http://www.eudora.com/presskit/pdf/Flaming_White_Paper.PDF.

Liddy, Elizabeth. 2001. «Natural Language Processing». Encyclopedia of Library and Information Science, 2nd Ed. NY. Marcel Decker, Inc.

Liu, Bing, and Hu, Minqing. 2004. «Opinion Mining, Sentiment Analysis, and Opinion Spam Detection». http://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html.

Lodhi, Huma; Saunders, Craig; Shawe-Taylor, John; and Cristianini, Nello. 2002. «Text Classification Using String Kernels». Journal of Machine Learning Research.

Manning, Christopher D, and Schütze, Hinrich. 1999. Foundations of Natural Language Processing. MIT Press.

Nguyen, Bach and Sameer, Badaskar. 2007. «A Survey on Relation Extraction». Literature review for Language and Statistics II.

Pang, Bo, and Lee, Lillian. 2008. «Opinion Mining and Sentiment Analysis». Foundations and Trends in Information Retrieval Vol 2, Issue 1–2. NOW.

Peccei, Jean. 1999. Pragmatics. Routledge, NY.

PubMed, MEDLINE. http://www.ncbi.nlm.nih.gov/sites/entrez.

Ravichandran, Deepak and Hovy, Eduard. 2002. «Learning Surface Text Patterns for a Question Answering System». Proceedings of the 40th Annual Meeting on Association for Computational Linguistics.

Turney, Peter. 2002. «Thumbs Up or Thumbs Down? Semantic Orientation Applied to Unsupervised Classification of Reviews». Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL).

Yi, Jeonghee; Nasukawa, Tetsuya; Bunescu, Razvan; and Niblack, Wayne. 2003. «Sentiment Analyzer: Extracting Sentiments About a Given Topic Using Natural Language Processing Techniques». Third IEEE International Conference on Data Mining. http://ace.cs.ohiou.edu/~razvan/papers/icdm2003.pdf.

Zelenko, Dmitry; Aone, Chinatsu; and Richardella, Anthony. 2003. «Kernel Methods for Relation Extraction». Journal of Machine Learning Research.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Л

M

H

O

П

Р

C

T

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Электронный адрес: **books@alians-kniga.ru**.

Оптовые закупки: тел. +7(499) 782-38-89.

Грант С. Ингерсолл, Томас С. Мортон, Эндрю Л. Фэррис

Обработка неструктурированных текстов

Поиск, организация и манипулирование

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 $^{1}/_{16}$. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 25,36.
Тираж 200 экз.

Web-сайт издательства: www.дмк.рф