O'REILLY®

Если вы знаете, как программировать на Python и немного знаете о теории вероятности, значит, вы готовы освоить байесовскую статистику. Эта книга расскажет вам, как решать статистические задачи с помощью языка Python вместо математических формул и использовать дискретные вероятностные распределения вместо непрерывной математики. Когда вы уберете с дороги математику, байесовские основы станут яснее, и вы начнете применять эту технику для решения реальных проблем.
Байесовские статистические методы становятся все более обширными и важными. Но в помощь начинающим доступно не слишком много источников. Изложенная в этой книге методика основана на материале проводимых Алленом Дауни студенческих занятиях и точно поможет вам сделать хороший старт!

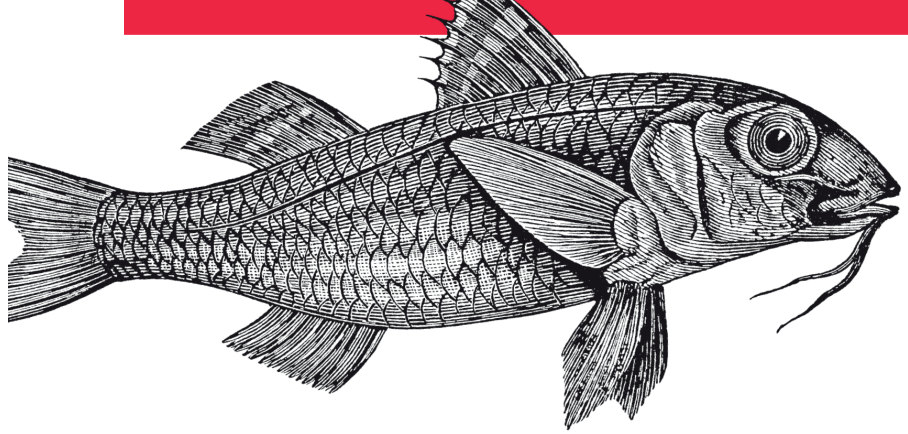
- Используйте ваше умение программировать, чтобы изучить и понять байесовскую статистику;
- работайте над задачами, требующими оценки, предсказания, анализа решений и событий и проверки гипотез;
- начните с простых примеров, использующих монеты, шоколадки M&M, компьютерную игру «Подземелья и Драконы», пейнтбол и хоккей;
- изучайте вычислительные методы для решения реальных задач, таких, как экзаменационные отметки SAT, моделирование болезни почек и наличие микроорганизмов в человеке.

Аллен Б. Дауни (Allen B. Dawney) *является профессором компьютерных наук в инженерном колледже Олин. Защитил докторскую диссертацию в области компьютерных наук в университете Беркли, преподавал информатику в колледже Уэлси, колледже Колби и университетском колледже Беркли. Он также является автором книг Think Stats и Think Python (оба вышли в издательстве O'Reilly).*

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru

ДМК ПРЕСС издательство
www.дмк.рф

ISBN 978-5-97060-664-3



O'REILLY®

Байесовская статистика на языке Python

Байесовские модели

ДМК ПРЕСС издательство

Аллен Б. Дауни

Аллен Б. Дауни

Байесовские модели

Allen B. Downey

Think Bayes

Bayesian Statistics in Python

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Аллен Б. Дауни

Байесовские модели

Байесовская статистика на языке программирования Phyton

ДМК Пресс
издательство
Москва, 2018

УДК 004.021
ББК 32.973.3
Д21

Дауни А. Б.
Д21 Байесовские модели / пер. с анг. В. А. Яроцкого. – М.: ДМК Пресс, 2018. – 182 с.: ил.

ISBN 978-5-97060-664-3

Если вы знаете, как программировать на Python, и немного знаете о теории вероятности, значит, вы готовы освоить байесовскую статистику. Эта книга расскажет вам, как решать статистические задачи с помощью языка Python вместо математических формул и использовать дискретные вероятностные распределения вместо непрерывной математики.

Байесовские статистические методы становятся все более обширными и важными. Но в помощь начинающим доступно не слишком много источников. Изложенная в этой книге методика основана на материале проводимых автором студенческих занятий и точно поможет вам сделать хороший старт!

Издание будет полезно всем специалистам по анализу данных, кто должен использовать статистические данные в условиях их неполноты или решать другие нетривиальные задачи, связанные с вероятностными распределениями.

УДК 004.021
ББК 32.973.3

ISBN 978-1-449-37078-7 (анг.)
ISBN 978-5-97060-664-3 (рус.)

Содержание

Вступительное слово 9

Предисловие 10

Глава 1. Теорема Байеса 16

Условная вероятность 16
Совместная вероятность 17
Задача о булочках 17
Теорема Байеса 18
Диахроническая интерпретация 19
Задача M&M 20
Задача Монти Холла 22
Обсуждение 24

Глава 2. Вычислительная статистика 25

Распределения 25
Задача с булочками 26
Байесовская структура 27
Задача Монти Холла 28
Формирование структуры программного пакета 29
Задача M&M 30
Обсуждение 31
Упражнение 32

Глава 3. Оценивание 33

Задача об игральных костях 33
Задача о локомотиве 34
Что насчет этого приора? 36
Альтернативный приор 37
Доверительный интервал 39
Кумулятивные функции распределения 39
Задача о немецком танке 40
Обсуждение 41
Упражнение 41

Глава 4. Больше об оценивании 43

Задача о евро 43
Итоговый постериор 44

Подавление приоров 45
Оптимизация 47
Бета-распределение 48
Обсуждение 50
Упражнения 50

Глава 5. Отношение вероятностей и добавления 52

Отношение вероятностей 52
Теорема Байеса в форме отношения вероятностей 53
Группа крови Оливера 54
Добавления 55
Максимизации 58
Перемешивание 60
Обсуждение 62

Глава 6. Анализ решений 63

Задача «Справедливой цены» 63
Приор 64
Функция плотности вероятности 65
Представление PDF 65
Моделирование участников 67
Правдоподобие 69
Обновление 70
Оптимальное предложение цены 71
Обсуждение 74

Глава 7. Предсказание 75

Задача о Бостон Брюинс 75
Процесс Пуассона 76
Постериоры 77
Распределение голов 78
Вероятность выигрыша 79
Выигрыш в дополнительное время 80
Обсуждение 82
Упражнения 83

Глава 8. Погрешность наблюдения 85

Задача о линии метрополитена 85
Модель 85
Время ожидания 87
Предсказание ожидаемого времени 89
Оценка времени прибытия 92

Включение неопределенности 94
Анализ решений 95
Обсуждение 97
Упражнение 98

Глава 9. Двумерное измерение 99

Пейнтбол 99
Пакет гипотез 99
Тригонометрия 100
Правдоподобие 102
Совместные распределения 102
Условные распределения 104
Доверительные интервалы 105
Обсуждение 107
Упражнение 108

Глава 10. Аппроксимация при байесовских вычислениях 109

Гипотеза изменчивости 109
Среднее и стандартное отклонение 110
Обновление 112
Апостериорное распределение CV 113
Потеря значимости 113
Логарифмическое правдоподобие 115
Небольшая оптимизация 116
Аппроксимация при байесовских вычислениях (ABC) 117
Робастное оценивание 118
Кто более изменчив? 120
Обсуждение 122
Упражнение 122

Глава 11. Проверка гипотез 124

Обратно к задаче о евро 124
Справедливое сравнение 125
Треугольный приор 126
Обсуждение 127
Упражнения 128

Глава 12. Свидетельства 129

Интерпретация оценки SAT 129
Шкала 129
Приор 130
Постериор 132

Улучшенная модель ... 133
Градуировка ... 135
Апостериорное распределение эффективности ... 136
Распределение предсказания ... 138
Обсуждение ... 138

Глава 13. Моделирование ... 140

Проблема опухоли почек ... 140
Простая модель ... 141
Более общая модель ... 143
Реализация ... 144
Кеширование совместного распределения ... 145
Условные распределения ... 146
Последовательная корреляция ... 147
Обсуждение ... 151

Глава 14. Иерархическая модель ... 152

Задача о счетчике Гейгера ... 152
Простое начало ... 153
Создание иерархии ... 154
Небольшая оптимизация ... 155
Извлечение постериоров ... 155
Обсуждение ... 157
Упражнение ... 157

Глава 15. Борьба с размерностью ... 158

Бактерии пупка ... 158
Львы, тигры и медведи ... 158
Иерархическая версия ... 161
Случайная выборка ... 163
Оптимизация ... 164
Сворачивание иерархии ... 165
Еще одна проблема ... 167
Мы сделали еще не все ... 168
Данные пупка ... 170
Прогнозирующее распределение ... 172
Совместный постериор ... 175
Перекрывающая зона ... 176
Обсуждение ... 178

Предметный указатель ... 180

Вступительное слово

Около десяти лет назад, когда изучение байесовских методов впервые заинтересовало меня, я обнаружил острую нехватку книг по данной теме на русском языке. Материала, в котором бы практически, с точки зрения реализации на конкретном языке программирования, описывались как базовые, так и более продвинутые методы анализа данных с помощью байесовских методов. При этом не составляло труда найти огромное количество достойных книг на английском языке, дающих глубокое практическое понимание этого отдельного важного класса методов, которые применимы в самом широком спектре областей: начиная от анализа экспериментальных данных и заканчивая современными системами принятия решений и даже блокчейном. Кстати, если говорить о последнем, то можно привести в пример NeuroChainTech – проект большой международной команды, в котором мне посчастливилось стать научным консультантом. Это реализация умного блокчейна с новым оригинальным алгоритмом консенсуса и элементами машинного обучения, включающими как раз байесовские сети. В этом году проект провел успешное ICO и в настоящее время находится в активной фазе своего развития. Кроме того, в настоящее время на байесовских методах базируются в том числе и современные системы принятия решений и анализа данных, которые активно используются для решения задач цифровизации экономики, выходящих в настоящее время на первый план в государственном и корпоративном развитии.

Несколько лет назад мои статьи на русском языке, опубликованные на популярном ресурсе в сети Интернет, в которых описывались базовые принципы имплементации байесовских методов на Python'е, нашли очень живой отклик читателей. Более того, до сегодняшнего дня, спустя пять лет с момента их публикации, мне по-прежнему поступают вопросы, связанные с практическим воплощением алгоритмов байесовского анализа, что в очередной раз подтверждает неподдельный и неснижающийся интерес широкой аудитории к пониманию и практическим аспектам реализации байесовских методов.

Появление перевода на русский язык отличной книги, подробно описывающей практическое воплощение байесовских методов на Python'е, – это безусловный повод для радости. Настоящая книга включает в себя описание базовых принципов реализации байесовских методов в самом широком спектре их применений, и я очень надеюсь, что она вызовет должный интерес у читателей и придаст новый импульс к изучению, активному применению и дальнейшему развитию байесовских методов.

Желаю читателям успешного овладения инструментарием байесовских методов и интересных его применений в будущих проектах!

Максим Иришкин, PhD,
научный консультант NeuroChainTech,
эксперт по инновационному развитию корпораций

Предисловие

Мой подход

Предпосылкой для этой книги, как и других книг серии *Think X*, является мысль о том, что если вы умеете программировать, вы можете использовать это умение, чтобы овладеть другими знаниями.

Большинство книг по байесовской статистике используют математические формулировки и представляют эти идеи как исчисление в терминах математических концепций. В этой книге вместо математики используются язык программирования Python и дискретная аппроксимация вместо непрерывной математики. В результате то, что книгах по математике является интегралом, становится суммированием, а большинство операций с вероятностными распределениями – просто циклами.

Мне кажется, что такое представление более понятно, по крайней мере для людей с навыками программиста. Оно также имеет более общий характер, потому что мы можем выбирать наиболее подходящую модель, не слишком беспокоясь, поддается ли она традиционному анализу реальных проблем. Глава 3 – хороший пример этого. Она начинается с простого примера с игральными костями – одного из основных в базовой теории вероятности. Затем небольшими шагами идет продвижение к задаче о локомотивах, которая позаимствована из книги Фредерика Мостеллера (Frederick Mosteller) «Пятьдесят интересных вероятностных задач с решениями» (Fifty Challenging Problems in Probability with Solutions. Dover, 1987) и затем к задаче о немецком танке, знаменитому успешному применению байесовского метода во время Второй мировой войны.

Моделирование и аппроксимация

Многие задачи в этой книге мотивированы реальными проблемами, что влечет за собой необходимость построения модели. Прежде чем мы применим байесовские методы (как и любой другой анализ), мы должны принять решение о том, какую часть реальной системы мы включим в модель и от каких деталей мы можем абстрагироваться.

Например, в главе 7 мотивирующей проблемой является предсказание победителя в игре в хоккей. Я применил для подсчета голов пуассоновский процесс, который подразумевает, что голы могут быть забиты равновероятно в любой момент игры. Это не совсем так, но эта модель, вероятно, подходит для многих других задач.

В главе 12 мотивацией проблемы является интерпретация экзаменационных оценок SAT (SAT является стандартизированным тестом, используемым при поступлении в колледж в США). Я начинаю с простой модели, в которой

предполагается, что все вопросы на экзамене SAT имеют одинаковую сложность. Однако фактически организаторы SAT сознательно включают часть вопросов, которые являются относительно простыми, а часть вопросов – которые являются более сложными. Я представил и вторую модель, которая учитывает это обстоятельство, и показал, что в конечном счете оно не оказывает существенного влияния на результат.

Я думаю, что важно считать моделирование неотъемлемой частью решения задачи, потому что это заставляет нас думать о модельных ошибках (то есть ошибках, появляющихся вследствие упрощений и предположений в моделях).

Многие методы в этой книге базируются на дискретных распределениях, что заставляет некоторых беспокоиться о численных ошибках. Но для реальных проблем численные ошибки почти всегда меньше модельных ошибок.

С другой стороны, непрерывные методы иногда приводят к преимуществам в производительности – например, путем замены вычислений с линейным или квадратичным временем на решение с постоянной продолжительностью.

Я рекомендую следующую общую процедуру при решении задач:

1) при исследовании проблемы начинайте с простой модели и опишите ее в ясных, читаемых и явно правильных кодах. Сфокусируйте внимание на хороших модельных решениях, не на оптимизации;
2) когда простая модель заработает, определите основные источники ошибок. Возможно, вам потребуется увеличить количество значений при аппроксимации, или увеличить число итераций в модели Монте-Карло, или добавить детали в модель;
3) если это решение приемлемо для вашего приложения, вам, возможно, нет необходимости заниматься его оптимизацией. Но если вы собираетесь это делать, существует два подхода, которые следует рассмотреть. Вы можете провести ревизию вашего кода и поискать возможности его оптимизации. Например, если вы кешировали полученные результаты, возможно, вам удастся избежать излишних вычислений. Или вы можете поискать аналитические методы, которые дают компьютерное ускорение.

Одно из достоинств этой процедуры в том, что шаги 1 и 2 склонны быть быстрыми, что дает возможность исследовать альтернативные модели до того, как вы начнете в них вкладываться.

Другим преимуществом является то, что если вы перейдете к шагу 3, вы уже начнете с некоторой справочной реализации, которая, вероятно, будет правильной, и вы сможете использовать ее для регрессионного тестирования (то есть проверяя, что оптимизированный код дает похожие результаты, по крайней мере приблизительно).

Работа с кодом

Многие примеры в этой книге используют классы и функции, определенные в модуле `thinkbayes.py`. Вы можете загрузить этот модуль из http://thinkbayes.com/thinkbayes.py.

Многие разделы содержат справки о кодах, которые вы можете загрузить из http://thinkbayes.com. Некоторые из этих файлов имеют зависимости, которые вы также должны будете загрузить. Я предлагаю вам держать все эти файлы в одной той же директории, так чтобы они могли импортироваться один в другой без изменения поискового пути Phyton.

Вы можете загрузить эти файлы один за другим, когда они потребуются, или вы можете загрузить их все сразу из http://thinkbayes.com/thinkbayes_code.zip. Этот файл также содержит файлы данных, используемые в некоторых программах. Когда вы разархивируете их, это создаст директорию, названную thinkbayes_code, которая содержит все коды, использованные в этой книге.

Или, если вы пользователь программы консольных утилит Git, вы можете сразу получить все файлы посредством ветвления и клонирования репозитория: https://github.com/AllenDowney/ThinkBayes.

Одним из модулей, которые я использовал, является `thinkplot.py`, который обеспечивает упаковку для некоторых функций в `pyplot`. Чтобы использовать, его необходимо загрузить `matplotlib`. Если у вас его еще нет, проверьте ваш менеджер пакетов на его доступность. В противном случае вы можете загрузить инструкции из http://matplotlib.org.

Наконец, некоторые программы в этой книге используют NumPy и SciPy, которые доступны на http://numpy.org и http://scipy.org.

Стиль кодов

Опытные программисты на Phyton могут заметить, что коды в этой книге не соответствуют PEP 8, являющемуся наиболее общим руководством стиля для Phyton (http://www.python.org/dev/peps/pep-0008/).

Конкретно PEP 8 требует нижнего регистра названий функций и подчеркивания между словами, `like_this`. В этой книге и сопровождающих кодах названия функции и методов начинаются с заглавной буквы и используют «верблюжий» стиль (camel style) LikeThis.

Я нарушил это правило, потому что я разработал эти коды, когда я был приглашенным ученым в Google, и следовал руководству Google по стилю кодов, который в некоторой части отличается от PEP 8. Используя стиль Google, я нашел его привлекательным, и мне было бы трудно его менять.

Также в духе этого стиля я написал «Bayes's Theorem» с одним «s» после апострофа, что предпочитают в одних руководствах и осуждают в других. У меня нет в этом вопросе каких-либо предпочтений, я должен был выбрать что-либо одно, и здесь то, что я выбрал.

И наконец, одно типографическое замечание: всюду в этой книге я использовал PMF и CDF для математической концепции функции масс вероятности и кумулятивной функции распределения и Pmf и Cdf для ссылки на объекты Phyton, которые я использую для их представления.

Предпосылки

Существует несколько замечательных модулей для работы с байесовской статистикой в Phyton, включая pym и OpenBUGS. Я решил не использовать их в этой книге, потому что вам необходимо много предварительной информации, чтобы начать работать с ними, а я хотел свести необходимость предпосылок к минимуму. Если вы знаете Phyton и немного о вероятности, вы готовы работать с этой книгой.

Глава 1 знакомит вас с вероятностью и теоремой Байеса. В ней нет кодов. Во второй главе вводится Pmf и Cdf, несколько измененный словарь Phyton, который я использовал для представления функции масс вероятности (PMF). Затем в главе 3 вводится Suite, своего рода структура для осуществления байесовских обновлений. И это, пожалуй, все.

Нет, почти все. В некоторых последних главах я использовал распределения, включающие распределение Гаусса (нормальное распределение), экспоненциальное распределение, распределение Пуассона и бета-распределение. В главе 15 я использовал не слишком употребительное распределение Дирихле, но затем я представил его в процессе дальнейшего изложения. Если вы незнакомы с этими распределениями, то можете прочитать о них в Википедии. Вы можете также прочитать о них и в одной из книг этой серии *Think Stats* и книгах по введению в статистику (хотя я боюсь, что большинство из них использует математический подход, что не особенно полезно для практики).

Список участников

Если у вас есть предложения или замечания, пожалуйста, направляйте e-mail по адресу downey@allendow.ney.com. Если я проведу изменения, основанные на ваших откликах, я включу вас в список участников (если вы не попросите этого не делать).

Если вы включите, по крайней мере, часть предложения, в котором обнаружилась ошибка, это облегчит мне поиск ее. Это касается и просто страниц и разделов, но это уже будет для меня более трудным. Спасибо!

- Прежде всего я должен упомянуть замечательную книгу Давида МакКея (David MacKay) «Теория информации, вывод и обучающие алгоритмы» (*Information Theory, Inference, and Learning Algorithms*), благодаря которой я пришел к пониманию байесовского метода. С его разрешения я использовал несколько задач из этой книги в качестве примеров.
- Эта книга улучшилась благодаря консультациям с Sanjoy Mahajan, особенно осенью 2012 года, когда я был аудитором его класса в колледже Олин.
- Часть этой книги я написал во время вечерней работы над проектом с группой Boston Python User Group, и я хотел бы поблагодарить их за сотрудничество и пиццу.
- Jonathan Edwards исправил первые опечатки.

- George Purkins нашел ошибки в разметке текста.
- Olivier Yiptong прислал несколько полезных предложений
- Yuriy Pasichnyk нашел несколько ошибок.
- Kristopher Overholt прислал длинный перечень исправлений и предложений.
- Robert Marcus нашел неправильно проставленное i.
- Max Hailperin предложил более ясное изложение главы 1.
- Markus Dobler указал на то, что вытаскивание булочек из корзины с заменой не вполне реалистичный сценарий.
- Tom Pollard и Paul A. Giannaros обнаружили проблему в версии с некоторыми величинами в задаче о локомотивах.
- Ram Limbu нашел опечатки и предложил исправления.
- Весной 2013 года студенты моего класса Вычислительная байесовская статистика (Computational Bayesian Statistics) сделали много полезных исправлений и предложений: Kai Austin, Claire Barnes, Kari Bender, RachelBoy, Kat Mendoza, Arjun Iyer, Ben Kroop, Nathan Lintz, Kyle McConnaughay, Alec Radford, Brendan Ritter и Evan Simpson.
- Greg Marra и Matt Aasted помогли мне обсуждением задачи о справедливой цене.
- Marcus Ogren указал мне на то, что первоначальное описание задачи о локомотивах было неоднозначным.

Jasmine Kwityn и Dan Fauxsmith из O'Reilly Media сделали корректуру книги и нашли много возможностей для улучшения.

Аллен Б. Дауни

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly Media очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Теорема Байеса

Условная вероятность

Основная идея всей байесовской статистики – это теорема Байеса, которую удивительно легко получить, если понять, что такое условная вероятность. Поэтому мы начнем с вероятности, затем перейдем к условной вероятности, далее к теореме Байеса и, наконец, к байесовской статистике.

Вероятность – это число между 0 и 1 (включая оба), которые представляют собой уровень уверенности в каком-либо факте или предсказании. Числом 1 представляется абсолютная уверенность, что некоторый факт справедлив. Числом 0 – абсолютная уверенность, что этот факт не справедлив. Промежуточные числа в этом интервале определяют степень уверенности. Число 0,5, часто обозначаемое как 50%, означает, что предсказанное событие в одинаковой степени может как осуществиться, так и не осуществиться. Например, при подбрасывании монеты вероятность того, что она упадет лицевой стороной вверх, близка к 50%.

Условная вероятность – это вероятность, основанная на некотором предыдущем знании. Например, я хочу узнать, какова вероятность того, что в следующем году у меня случится сердечный приступ. Данные Центра контроля заболеваний гласят: «Ежегодно сердечный приступ случается у примерно 785 тысяч американцев» (http://www.cdc.gov/heartdisease/facts.htm).

Численность населения США составляет примерно 311 миллионов человек. Значит, вероятность того, что случайным образом выбранный американец получит в следующем году сердечный приступ, составляет примерно 0,3%.

Но я не являюсь случайно выбранным американцем. Эпидемиологи определили много факторов, влияющих на риск получения сердечного приступа, и в зависимости от этих факторов мой риск получить сердечный приступ может сильно отличаться от среднего значения.

Я, 45-летний мужчина, имею пограничо высокий холестерин. Этот факт увеличивает риск, что я получу сердечный приступ. Вместе с тем у меня низкое кровяное давление и я не курю, и это уменьшает мой шанс получения сердечного приступа.

Включая всю такого рода информацию в онлайновый калькулятор, размещенный на интернет-странице http://hp2010.nhlbihin.net/atpiii/calculator.as,

я увижу, что значение риска получения мною сердечного приступа в следующем году равно 0,2%. То есть это значение гораздо меньше, чем в среднем по стране. Вот это значение и есть условная вероятность, поскольку она основана на ряде факторов, которые составляют мои «условия».

Обычное обозначение условной вероятности p(A|B) – вероятность A при условии, что справедливо B. В этом примере A является предсказанием, что я получу сердечный приступ в следующем году, а B – множество условий.

Совместная вероятность

Совместная вероятность – это способ полагать, что оба факта или предсказания окажутся осуществленными. Я напишу p(A и B), если имею в виду, что вероятность выполнения совместно и A, и B существует.

Если вы поняли вероятность в контексте подбрасывания монеты или игральной кости, вы можете понять и содержание следующей формулы:

p(A и B) = p(A) p(B). ПРЕДУПРЕЖДЕНИЕ: это не всегда так.

Например, если я подбрасываю две монеты, значение A означает, что первая монета упала лицевой стороной вверх, а значение B – что вторая тоже упала лицевой стороной вверх, то есть p(A) = p(B) = 0,25.

Но эта формула справедлива, только если A и B независимы, то есть результат первого события не изменяет вероятность второго. Или более формально p(A|B) = p(B).

Приведу другой пример, в котором события не независимы. Предположим, A означает, что сегодня идет дождь, а B – что дождь пойдет завтра. Если я знаю, что сегодня идет дождь, то весьма вероятно, что и завтра будет дождь, поэтому p(A|B) > p(B).

В общем, вероятность при логическом объединении событий

p(A и B) = p(A) p(B|A)

для любых A и B. Таким образом, шанс, что дождь пойдет в любой данный день, равна 0,5, а шанс, что дождь пойдет два дня подряд, равна не 0,25, а несколько выше.

Задача о булочках

Мы вскоре приступим к теореме Байеса. Но прежде я хочу объяснить ее с помощью примера, известного как задача о булочках[1]. Предположим, что имеется две корзины с булочками. В корзине под номером 1 лежит 30 ванильных и 10 шоколадных булочек, а в корзине под номером 2 лежит по 20 булочек обоих сортов.

[1] Основана на примере из http://en.wikipedia.org/wiki/Bayes'_theorem. Эта интернет-страница больше не существует.

Теперь предположим, что вы случайным образом выбираете одну из корзин и, не заглядывая внутрь, берете первую попавшуюся булочку. Ею оказывается ванильная булочка. Какова вероятность, что он выбрана из корзины 1?

Это условная вероятность. Мы хотим определить p(корзина1|ваниль), но не понятно, как это сделать. Проще ответить на другой вопрос – какова вероятность, что ванильная булочка лежит в корзине номер 1:

p(корзина1|ваниль) = 3/4.

Хотя p($A|B$) не то же, что p($B|A$), существует способ, как из одного получить другое. И здесь нам поможет теорема Байеса.

Теорема Байеса

Теперь у нас есть все необходимое, чтобы вывести теорему Байеса. Начнем с коммутативности объединения:

p(A и B) = p(B и A).

Это утверждение справедливо для любого из событий A и B.

Далее напишем соотношение для вероятности объединения:

p(A и B) = p(A) p($B|A$).

Так как нами ничего не было сказано о значении A и B, можно предположить, что они взаимозаменяемые. Их взаимозаменяемость приводит к утверждению:

p(A и B) = p(B) p($A|B$).

Это все, что нам необходимо. Приравнивая соотношения друг к другу, получаем:

p(B) p($A|B$) = p(A) p($B|A$).

Здесь можно сказать, что существует два способа объединения. Если у вас есть p(A), то вы умножаете его на p($B|A$). Или можете сделать по-другому: если вам известно p(B), то вы умножаете его на p($A|B$). Оба способа приводят к одному и тому же результату.

Наконец, мы можем разделить правую часть на p(B):

$$p(A \mid B) = \frac{p(A)p(B \mid A)}{p(B)}.$$

Это и есть теорема Байеса! Может быть, она простенькая на вид, но, как окажется, удивительно действенна.

Например, мы можем использовать ее для решения задачи о булочках. Я назову гипотезой B_1 событие, что булочка была вынута из корзины 1, и гипотезой V, что она оказалась ванильной. Подставляя в теорему Байеса, мы получаем:

$$p(B_1|V) = \frac{p(B_1)\, p(V|B_1)}{p(V)}.$$

Левая часть формулы – это то, что мы хотим: вероятность корзины 1 при условии, что мы выбрали ванильную булочку. В правой части формулы имеем:

- $p(B_1)$ – вероятность, что мы выбрали корзину 1 независимо от того, какую булочку мы взяли. Поскольку в задаче сказано, что выбор корзины случайный, полагаем, что $p(B_1) = 1/2$;
- $p(V|B_1)$ – это вероятность получения ванильной булочки из корзины 1, которая равна 3/4;
- $p(V)$ – вероятность вынуть ванильную булочку из любой корзины. Поскольку у нас одинаковый шанс выбора из любой корзины и в обеих корзинах находится одинаковое количество булочек, мы имеем одинаковый шанс вытащить любую булочку. В двух корзинах находится 50 ванильных и 30 шоколадных булочек, поэтому $p(V) = 5/8$.

Подставив это значение в формулу, получаем:

$$p(B_1|V) = \frac{(1/2)(3/4)}{5/8},$$

что составляет 3/5. Отсюда следует, что ванильная булочка свидетельствует в пользу гипотезы, что мы вынимали булочку из корзины 1, потому более вероятно, что ванильная булочка вынута из корзины 1.

Этот пример демонстрирует пользу теоремы Байеса: данная теорема обеспечивает стратегию выбора между $p(B|A)$ и $p(A|B)$. Эта стратегия полезна в задачах, подобных задаче о булочках, где проще вычислять значения правой части теоремы Байеса, чем левой.

Диахроническая интерпретация

Существует другой подход к теореме Байеса. Эта теорема дает возможность обновить вероятность гипотезы H при наличии некоторого объема данных D.

Такое представление теоремы Байеса называется **диахронической интерпретацией**. «Диахроническое» означает что-то, происходящее с течением времени. В таком случае при изменении в течение времени старых и появлении новых данных вероятность гипотез меняется.

Переписывая теорему Байеса с H и D, получаем

$$p(H \mid D) = \frac{p(H)\, p(D \mid H)}{p(D)}.$$

В этой интерпретации каждая составляющая формулы имеет свое наименование:

- $p(H)$ – вероятность гипотезы до получения новых данных. Данное значение называется априорной вероятностью, или просто **приор**;

- p($H|D$) – это то, что мы хотим определить: вероятность гипотезы после получения новых данных. Это значение называется апостериорной вероятностью, или **постериор**;
- p($D|H$) – вероятность данных для этой гипотезы, называемой **правдоподобием**;
- p(D) – вероятность данных для любой из гипотез, носящей название **нормализующей константы**.

Иногда мы можем вычислить приор на основе предварительной информации. Например, в задаче с булочками предопределено, что вероятность случайного выбора корзины одинакова.

В других случаях приор субъективен, то есть разумный человек может с этим не согласиться либо по причине использования другой исходной информации, либо потому, что интерпретирует ту же информацию по-иному.

Правдоподобие вычисляется наиболее просто. В задаче о булочках, если известно, из какой корзины мы достаем булочку, мы находим вероятность ванильной булочки простым подсчетом.

Нормализующая константа может быть ненадежной. Предполагается, что это вероятность полученных данных по любой гипотезе, но в самом общем случае трудно интерпретировать их значение.

Чаще всего мы упрощаем дело, определяя гипотезы как:

- *взаимно исключаемые*: не более чем одна гипотеза из данного множества может быть верной;
- *совместно исчерпывающие*: не существует других возможностей. Хотя бы одна из гипотез должна быть верной.

Я использую термин **suite** (**комплект**) относительно множества гипотез, обладающих этими свойствами.

В задаче с булочками присутствуют только две гипотезы – булочка достается из корзины 1 или из корзины 2. Эти гипотезы взаимно исключающие и совместно исчерпывающие.

В этом случае можно вычислить p(D), используя закон полной вероятности. Данный закон говорит: если имеется два исключающих варианта того, что может случиться, то вы можете добавить такую вероятность:

$$p(D) = p(B_1)\,p(D|B_1) + p(B_2)\,p(D|B_2).$$

Подставляя значения из задачи о булочках, имеем:

$$p(D) = (1/2)\,(3/4) + (1/2)\,(1/2) = 5/8.$$

То есть мы получили то же самое значение, что вычислили ранее, мысленно объединив две корзины.

Задача M&M

M&M – шоколадное драже, поверхность которого окрашена в разные цвета. Компания «Марс» (Mars, Inc.) изготавливает это драже, время от времени меняя цвета.

В 1995 году драже окрашивались в синий цвет. До этого пакетик M&M содержал драже следующих цветов: 30% коричневых, 20% желтых, 20% красных, 10% зеленых, 10% оранжевых и 10% желто-коричневых. В дальнейшем цвета драже были изменены следующим образом: 24% синих, 20% зеленых, 16% оранжевых, 14% желтых, 13% красных, 13% коричневых.

Допустим, у моего приятеля было два пакетика M&M. Один пакетик 1994 года выпуска, а другой – 1996-го. Не сообщив, какой пакетик какого года выпуска, мой приятель дал мне по одному драже из каждого. Одно драже было желтым, другое – зеленым. Какова вероятность, что желтое драже было из пакетика 1994 года выпуска?

Задача аналогична задаче о булочках. С той разницей, что я вынимал один экземпляр булочки из корзины (пакетика).

Задача о драже позволяет мне продемонстрировать табличный метод решения, который полезен для письменного решения задачи. В следующей главе мы будем решать задачу о драже на компьютере.

Сначала перечислим гипотезы. Пакетик, из которого я получил желтое драже, назовем Пакетик 1. Другой назовем Пакетик 2. Итак, мы имеем следующие гипотезы:

- A: Пакетик 1 1994 года выпуска предполагает, что Пакетик 2 выпуска 1996 года;
- B: Пакетик 1 1996 года выпуска, а Пакетик 2, наоборот, выпуска 1994 года.

Теперь составим таблицу со строками для каждой из гипотез и со столбцами для каждой составляющей теоремы Байеса.

	Приор $p(H)$	**Правдоподобие** $p(D\|H)$	$p(H)\ p(D\|H)$	**Постериор** $p(D\|H)$
A	1/2	(20)(20)	200	20/27
B	1/2	(10)(14)	70	7/27

Первый столбец содержит приоры. Исходя из условий задачи, вероятность, что Пакетик 1 1994 года выпуска, такая же, как и то, что он выпущен в 1996 году. То же самое можно сказать и о Пакетике 2. Поэтому разумно выбрать $p(A) = p(B) = 1/2$.

Второй столбец содержит правдоподобия, которые следуют из содержания задачи. Например, если справедливо A, то желтое драже с 20%-ной вероятностью поступило из пакетика 1994 года, а зеленое – с вероятностью 20% из пакетика 1996 года. Поскольку выборки независимы, мы посредством умножения получаем совместную вероятность.

Третий столбец – результат умножения данных из первых двух столбцов. Это нормализующие константы для каждой строки. Сумма нормализующих констант равна 270. Последний столбец содержит постериоры, для получения которых следует разделить нормализующую константу соответствующей строки на сумму констант.

Просто. Не правда ли?

Однако вас может беспокоить одна деталь. Я описал $p(D|H)$ в процентах, а не в значениях вероятности. Это означает увеличение относительно значения

терминов вероятности в 10 000 раз. Но эта деталь нивелируется при делении на нормализующую константу и, следовательно, не влияет на результат.

Если множество гипотез является взаимно исключаемым и совместно исчерпывающим, то, если это удобно, вы можете умножать правдоподобие на любой коэффициент, применив этот коэффициент к значениям во всех колонках.

Задача Монти Холла

Задача Монти Холла, возможно, вызвала наибольшие споры в истории вероятности. Сценарий прост. Но правильный ответ, казалось, настолько противоречит здравому смыслу, что многие никак его не могут воспринять. И даже умные люди ставят себя в неловкое положение, не только отстаивая неверный ответ, но и агрессивно защищая его на публике.

Монти Холл был первым ведущим шоу «*Давай сделаем дело*». Задача Монти Холла основана на одной из игр, регулярно проводимых на этом шоу. Если бы вы участвовали в этом шоу, то с вами происходило бы следующее:

- Монти показывал на три закрытые двери и говорил, что за каждой дверью находится приз: за одной дверью – автомобиль, а за двумя другими дверями – два различных малоценных приза. Например, арахисовое масло и накладные ногти. Призы за дверями размещались случайным образом;
- вам необходимо догадаться, за какой дверью находится автомобиль. Если вы угадывали, то получали автомобиль в качестве приза;
- есть три двери: Дверь A, Дверь B и Дверь C. Вы указывали на выбранную дверь;
- прежде чем открыть выбранную вами дверь, Монти увеличивал неопределенность, открывая либо Дверь B, либо Дверь C. Но обязательно открывал ту дверь, за которой автомобиля не было. Понятно, если автомобиль действительно находился за Дверью A, Монти мог безопасно открыть Дверь B или C, выбирая любую из них случайным образом;
- затем Монти предлагал вам выбрать: или остановиться на выбранной вами двери, или указать на дверь, оставшуюся закрытой.

Вопрос состоит в том, следует ли вам «остаться» на ранее выбранной двери или «переключиться» на оставшуюся закрытой дверь. Или это не важно.

Большинство людей чисто интуитивно полагают, что это не важно. Осталось две двери, рассуждают они. Поэтому шанс, что автомобиль находится за Дверью A, равен 50%.

Но это неверно. Фактически шанс выиграть, если вы останетесь на прежнем выборе Двери A, составит лишь 1/3. А если вы «переключитесь» на другую дверь, то ваш шанс составит 2/3.

Применив теорему Байеса, мы разобьем эту задачу на несколько частей и, возможно, убедимся, что такой ответ действительно корректен.

Для начала нам следует аккуратно выбрать исходные данные. Здесь D состоит из двух частей: Монти выбирает Дверь B, *и* за ней нет автомобиля.

Теперь мы определяем три гипотезы: A, B и C. То есть A, B и C представляют гипотезы, когда автомобиль находится за Дверью A, Дверью B и Дверью C. Вновь применим табличный метод.

	Приор p(*H*)	**Правдоподобие p(*D*|*H*)**	**p(*H*) p(*D*|*H*)**	**Постериор p(*D*|*H*)**
A	1/3	1/2	1/6	1/3
B	1/3	0	0	0
C	1/3	1	1/3	2/3

Заполнение столбца Приор не представляет трудностей, потому что нам сказано, что призы размещены случайным образом. Это предполагает, что автомобиль с одинаковой вероятностью может находиться за любой из трех дверей.

Подсчет правдоподобий потребует некоторых рассуждений. Но, соблюдая известную осторожность, мы сможем определить эти рассуждения правильно:

- если автомобиль действительно находится за дверью A, Монти мог безопасно открыть двери B и C. Поэтому вероятность, что он выберет C, равна 1/2. А поскольку автомобиль в действительности находится за дверью A, вероятность того, что автомобиль не за дверью B, равна 1;
- если автомобиль в действительности находится за дверью B, то Монти должен открыть дверь C. Поэтому вероятность, что он откроет дверь B, равна 0;
- наконец, если автомобиль находится за дверью C, то Монти открывает дверь B с вероятностью 1 и не находит там автомобиля с вероятностью 1.

Самая трудная часть решения позади. Далее простая арифметика. Сумма третьей колонки в таблице равна 1/2. Разделив полученные результаты, имеем p(*A*|*D*) = 1/3 и p(*C*|*D*) = 2/3.

Существует много вариантов задачи Монти Холла. Одна из сильных сторон байесовского подхода – в том, что он обобщает методику решения этих вариантов задач.

Например, предположим, что Монти всегда выбирает B, если есть такая возможность, и только C, если он должен это сделать (поскольку автомобиль за дверью B). В этом случае преобразованная таблица выглядит так:

	Приор p(*H*)	**Правдоподобие p(*D*|*H*)**	**p(*H*) p(*D*|*H*)**	**Постериор p(*D*|*H*)**
A	1/3	1	1/3	1/2
B	1/3	0	0	0
C	1/3	1	1/3	1/2

Единственное, что претерпело изменения, – p(*D*|*A*). Если автомобиль за дверью A, Монти может открыть или дверь B, или дверь C. Но в этом варианте он всегда выбирает B. Поэтому p(*D*|*A*) =1.

В результате правдоподобие всегда одинаково для гипотез A и C, и постериоры тоже одинаковы: p($A|D$) = p($C|D$) = 1/2. В этом случае факт, что Монти открывает B, не дает информации о размещении автомобиля. Поэтому не важно, участник «остается» на ранее выбранной двери или «переключается» на другую дверь.

С другой стороны, если он открыл дверь C, мы знаем, что p($B|D$) = 1.

Я включил задачу Монти Холла в эту главу, так как она мне кажется забавной и потому, что теорема Байеса несколько уменьшает сложность решения задачи. Но эта задача не является типичной для применения теоремы Байеса. Так что если вы нашли ее слегка сбивающей с толку, не унывайте!

Обсуждение

Для многих задач, затрагивающих условные вероятности, теорема Байеса обеспечивает стратегию «разделяй и властвуй». Если вычисление p($A|B$) или ее экспериментальное определение затруднительны, проверьте, не проще ли вычислить другие составляющие теоремы Байеса p($B|A$), p(A), p(B).

Глава 2

Вычислительная статистика

Распределения

В статистике понятие «**распределение**» обозначает некоторое множество величин и соответствующие им вероятности.

Например, если бросить игральную кость, то множество возможных выпадающих чисел (количество граней, на которых нанесены цифры) составляет 6. Поэтому вероятность, ассоциируемая с каждым из этих чисел, составляет 1/6.

Другой пример. Частота употребления слов в языке разная. Чтобы подсчитать частоту употребления каждого слова в английском языке, вы могли бы создать таблицу-распределение, в которой учитывалось бы каждое слово и частота его появления.

Чтобы представить распределение в языке программирования Phyton, можно использовать словарь, который соотносит каждой величине ее вероятность. Мною написан класс, названный `Pmf` (Probability mass function – **вероятностная функция масс**), являющийся математическим способом представления распределения.

Класс `Pmf` определен в модуле Phyton `thinkbayes.py`. Он создан мною при написании этой книги. Вы можете скачать его из http://thinkbayes.com/thinkbayes.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Для использования `Pmf` его можно импортировать следующим образом:

```
from thinkbayes import Pmf
```

Код для создания Pmf и получения распределения при бросании шестигранной игральной кости:

```
pmf = Pmf()
for x in [1,2,3,4,5,6]:
    pmf.Set(x, 1/6.0)
```

`Pmf` создает пустой класс Pmf. `Set` устанавливает соответствие вероятности 1/6 каждому числу.

Другой пример – как подсчитать, сколько раз каждое слово появляется в последовательности слов:

```
pmf = Pmf()
for word in word_list:
    pmf.Incr(word, 1)
```

`Incr` увеличивает «вероятность», соответствующую каждому слову, на 1. Если слово еще отсутствует в Pmf, то оно добавляется в словарь.

В этом примере я заключил слово вероятность в кавычки, поскольку вероятность здесь не нормализована. То есть не увеличивается *до* 1 и поэтому не является вероятностью как таковой.

Однако в этом примере подсчет слов пропорционален вероятности. Поэтому после подсчета всех слов мы можем вычислить вероятности делением на общее количество слов. `Pmf` создает метод `Normalize`, который выполняет следующее:

```
pmf.Normalize()
```

Если у вас есть Pmf-объект, вы можете запросить вероятность, ассоциированную с любой величиной:

```
print pmf.Prob('the')
```

Это выведет, например, частоту употребления слова «the» в виде дробного соотношения в списке слов.

Pmf использует словарь Phyton для запоминания величин и их вероятностей. Поэтому эти величины в Pmf могут быть любого хеш-образного типа.

Задача с булочками

В контексте теоремы Байеса естественно использовать Pmf для отображения процесса от гипотезы до ее вероятности. В задаче о булочках гипотезами являются B_1 и B_2. В Phyton я представил их в виде строк:

```
pmf = Pmf()
pmf.Set('Bowl 1', 0.5)
pmf.Set('Bowl 2', 0.5)
```

Это распределение, содержащее приоры для каждой гипотезы, называется **распределением приоры**, или **априорным распределением**. Чтобы получить распределение на основе новых данных (ванильная булочка), мы умножаем каждый приор на соответствующее правдоподобие. Правдоподобие извлечения ванильной булочки из корзины 1 равно 3/4. Вероятность для корзины 2 составляет 1/2.

```
pmf.Mult('Bowl 1', 0.75)
pmf.Mult('Bowl 2', 0.5)
```

`Mult` выполняет ожидаемое – получает вероятность для данной гипотезы и умножает на данное правдоподобие.

После этого распределение более не нормализовано. Но поскольку эти гипотезы взаимно и совместно независимые, мы можем снова провести нормализацию.

```
pmf.Normalize()
```

В результате получаем распределение, содержащее апостериорную вероятность для каждой гипотезы, которая называется **апостериорным распределением**.

Наконец, мы можем получить апостериорную вероятность для корзины 1:

```
print pmf.Prob('Bowl 1')
```

И ответ равен 0.6. Вы можете скачать этот пример с http://thinkbayes.com/cookie.py. Дополнительная информация находится в разделе «Работа с кодом» на стр. 11.

Байесовская структура

Прежде чем мы перейдем к следующей задаче, я хочу переписать код предыдущего подраздела, сделав этот код более общим. Сначала мы определим класс, чтобы сформировать код применительно к этой задаче:

```
class Cookie(Pmf):

    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()
```

Объект Cookie – это Pmf, отображающий процесс от гипотезы до ее вероятности. Метод `__init__` дает каждой гипотезе ту же априорную вероятность. Поскольку в задаче о булочках две гипотезы:

```
    hypos = ['Bowl 1', 'Bowl 2']
    pmf = Cookie(hypos)
```

`Cookie` предусматривает `Update`-метод, воспринимающий данные как параметр, и обновляет вероятности:

```
    def Update(self, data):
        for hypo in self.Values():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo, like)
        self.Normalize()
```

`Update` проходит по каждой из гипотез в наборе и умножает их вероятность на правдоподобие этих данных по гипотезе, которые вычисляются посредством `Likelihood`:

```
    mixes = {
        'Bowl 1':dict(vanilla=0.75, chocolate=0.25),
        'Bowl 2':dict(vanilla=0.5, chocolate=0.5),
         }

    def Likelihood(self, data, hypo):
        mix = self.mixes[hypo]
```

```
        like = mix[data]
        return like
```

`Likelihood` использует `mixes`, являющийся словарем, отображающим имя корзины на содержащиеся в ней булочки.

Обновление выглядит так:

```
pmf.Update('vanilla')
```

Затем мы можем вывести апостериорную вероятность каждой гипотезы:

```
for hypo, prob in pmf.Items():
    print hypo, prob
```

Результат:

```
Bowl 1 0.6
Bowl 2 0.4
```

Результат соответствует тому, который мы получили ранее. Этот код более сложный, чем тот, который был приведен в прошлом разделе. Одним из его преимуществ является то, что он обобщает случай, где мы вынимаем более одной булочки из одной и той же корзины (с замещением):

```
dataset = ['vanilla', 'chocolate', 'vanilla']
for data in dataset:
    pmf.Update(data)
```

Другое преимущество – в том, что он предоставляет структуру для решения многих подобных задач. В следующем разделе мы будем решать на компьютере задачу Монти Холла и увидим, какие части этой структуры одинаковы.

Код данного раздела доступен на http://thinkbayes.com/cookie2.py. Дополнительная информация находится в разделе «Работа с кодом» на стр. 11.

Задача Монти Холла

Для решения задачи Монти Холла я определю новый класс.

```
class Monty(Pmf):

    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()
```

Пока `Monty` и `Cookie` совершенно одинаковы. И коды, создающие Pmf, также одинаковы, за исключением имен гипотез:

```
hypos = 'ABC'
pmf = Monty(hypos)
```

Вызов Update такой же:

```
data = 'B'
pmf.Update(data)
```

И применение Update такое же:

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(data, hypo)
        self.Mult(hypo, like)
    self.Normalize()
```

Лишь часть, требующая некоторой работы Likelihood:

```
def Likelihood(self, data, hypo):
    if hypo == data:
        return 0
    elif hypo == 'A':
        return 0.5
    else:
        return 1
```

Наконец, вывод результатов тот же:

```
for hypo, prob in pmf.Items():
    print hypo, prob
```

Ответы:

```
A 0.333333333333
B 0.0
C 0.666666666667
```

Формирование структуры программного пакета

Теперь, когда мы видим одинаковые элементы структуры, мы можем внедрить их в объект: Suite – это Pmf, который предусматривает __init__, Update и Print:

```
class Suite(Pmf):
    """Represents a suite of hypotheses and their probabilities."""

    def __init__(self, hypo=tuple()):
        """Initializes the distribution."""

    def Update(self, data):
        """Updates each hypothesis based on the data."""

    def Print(self):
        """Prints the hypotheses and their probabilities."""
```

Реализация Suite находится в модуле thinkbayes.py. Чтобы использовать Suite, следует описать класс, наследуемый от него, и предусмотреть Likelihood.

Например, решение задачи Монти Холла, переписанное для использования `Suite`:

```
from thinkbayes import Suite

class Monty(Suite):

    def Likelihood(self, data, hypo):
        if hypo == data:
            return 0
        elif hypo == 'A':
            return 0.5
        else:
            return 1
```

Код, использующий этот класс:

```
suite = Monty('ABC')
suite.Update('B')
suite.Print()
```

Вы можете скачать этот пример на http://thinkbayes.com/monty2.py. Дополнительная информация находится в разделе «Работа с кодом» на стр. 11.

Задача M&M

Мы можем использовать структуру `Suite`, чтобы решить задачу M&M. Обратите внимание на хитрое написание `Likelihood` функции. Все остальное очевидно.

Сначала нам необходимо закодировать цвета драже до и после 1995 года:

```
mix94 = dict(brown=30,
             yellow=20,
             red=20,
             green=10,
             orange=10,
             tan=10)
mix96 = dict(blue=24,
             green=20,
             orange=16,
             yellow=14,
             red=13,
             brown=13)
```

Затем следует закодировать гипотезы:

```
hypoA = dict(bag1=mix94, bag2=mix96)
hypoB = dict(bag1=mix96, bag2=mix94)
```

`hypoA` представляет гипотезу, что пакетик 1 выпуска 1994 года, а пакетик 2 выпуска 1996-го. `hypoB` представляет противоположный вариант.

Далее сопоставляется имя гипотезы с представлением:

```
hypotheses = dict(A=hypoA, B=hypoB)
```

И наконец, можно написать Likelihood. В этом случае гипотеза hypo – строка как для A, так и для B. Данные являются кортежем, определяющим корзину и цвет:

```
def Likelihood(self, data, hypo):
    bag, color = data
    mix = self.hypotheses[hypo][bag]
    like = mix[color]
    return like
```

Код, который создает пакет программ и обновляет его:

```
suite = M_and_M('AB')

suite.Update(('bag1', 'yellow'))
suite.Update(('bag2', 'green'))

suite.Print()
```

Результат:

```
A 0.740740740741
B 0.259259259259
```

Апостериорная вероятность гипотезы A равна примерно 20/27. Этот результат мы получили ранее.

Код в этом разделе доступен на http://thinkbayes.com/m_and_m.py. Дополнительная информация находится в разделе «Работа с кодом» на стр. 11.

Обсуждение

В этой главе представлен класс Suite, формирующий обновленную байесовскую структуру.

Suite является **абстрактным классом**. Это означает, что он определяет интерфейс, который Suite предположительно имеет. Но его полная реализация не предусматривается. Suite-интерфейс включает Update и Likelihood, но Suite предусматривает реализацию лишь Update, но не Likelihood.

Конкретный класс – это класс, расширяющий абстрактный родительский класс и предусматривающий реализацию пропущенных методов. Например, Monty расширяет Suite и таким образом наследует Update и предусматривает Likelihood.

Те, кто знаком с шаблонным проектированием, могут рассматривать это как метод шаблонного проектирования. С этим методом вы можете познакомиться на http://en.wikipedia.org/wiki/Template_method_pattern.

Многие примеры в последующих главах следуют подобным шаблонам. Для каждой задачи мы определяем новый класс, расширяющий Suite, наследующий Update и предусматривающий Likelihood. В некоторых случаях мы подменяем Update, чтобы повысить производительность.

Упражнение

Упражнение 2.1

В разделе «Формирование структуры программного пакета» на стр. 27 было сказано, что решение задачи о булочках обобщает случай, когда вынимаются булочки с замещением.

Но более вероятен сценарий, в котором вынутая булочка съедается и правдоподобие каждого вынимания булочки зависит от прежнего.

Модифицируйте решение, полученное в данной главе, при решении задачи, когда из корзины вынимается булочка без замещения. Подсказка: добавьте переменные экземпляра в `Cookie`, чтобы представить гипотетическое состояние корзин, и модифицируйте `Likelihood` соответственно. Вы можете захотеть определить некий `Bowl`-объект.

Глава 3

Оценивание

Задача об игральных костях

Предположим у меня есть коробка с игральными костями, в которой находятся 4-гранная, 6-гранная, 8-гранная, 12-гранная и 20-гранная игральные кости. Если вы когда-либо играли в игру «Подземелья и драконы» (*Dungeons & Dragons*), то вы знаете, о чем идет речь.

Предположим, я случайным образом вынул из коробки одну кость, бросил ее и получил цифру 6. Какова вероятность того, что я бросил какую-то из находившихся в коробке костей?

Предлагается стратегия из трех шагов для разрешения такого рода задач.

1. Выберем представление для гипотез.
2. Выберем представление для данных.
3. Напишем функцию правдоподобия.

В предыдущих примерах я использовал строки для представления гипотез и данных, но в задаче об игральных костях я буду использовать номера.

Более конкретно: гипотезы будут представляться целыми числами 4, 6, 8, 12 и 20 (по количеству граней на игральных костях):

```
suite = Dice([4, 6, 8, 12, 20])
```

Данные будут представлены целыми числами от 1 до 20. Такое представление облегчает написание функции правдоподобия:

```
class Dice(Suite):
    def Likelihood(self, data, hypo):
        if hypo < data:
            return 0
        else:
            return 1.0/hypo
```

Теперь о том, как `Likelihood` работает. Если `hypo<data` (что означает, что брошенная кость больше числа граней кости, чего не может быть), то правдоподобие равно 0.

В противном случае возникает вопрос: «Пусть это будет `hypo` граней, тогда каков шанс выпавших данных?» Ответ `1/hypo`, независимо от данных.

Напишем объявление обновления (если бросили 6-гранную кость):

```
suite.Update(6)
```

Получаем апостериорное распределение:

```
4 0.0
6 0.392156862745
8 0.294117647059
12 0.196078431373
20 0.117647058824
```

После того как мы бросили кость с номером 6, вероятность для 4-гранной кости равна 0. Наиболее вероятная альтернатива – 6-гранная кость, но остается еще 12% шансов для 20-гранной кости.

А что, если мы бросим большее количество раз и получим 6, 8, 7, 7, 5 и 4?

```
for roll in [6, 8, 7, 7, 5, 4]:
    suite.Update(roll)
```

При таких данных 6-гранник исключается, и 8-граннанная кость окажется весьма вероятной. Результат таков:

```
4 0.0
6 0.0
8 0.943248453672
12 0.0552061280613
```

Теперь вероятность того, что мы бросили 8-гранную кость, стала равной 94%, а на 20-гранную отводится менее 1%.

Эта задача об игральных костях основана на примере, который я увидел в классе Санджя Махаджана[1] (Sanjoy Mahajan's class) по байсовским выводам. Код этой задачи вы можете скачать с http://thinkbayes.com/dice.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Задача о локомотиве

Задача о локомотиве была найдена в книге Фредерика Мостеллера (Frederick Mosteller) «Пятьдесят интересных вероятностных задач с решениями» (Fifty Challenging Problems in Probability with Solutions, Dover, 1987).

> «Железная дорога присваивает номера своим локомотивам в порядке 1...N. Однажды вы увидели локомотив с номером 60. Оцените, сколько локомотивов у этой железной дороги».

Основываясь на таком наблюдении, мы знаем, что железная дорога имеет как минимум 60 локомотивов. Или больше. Но насколько больше? Чтобы применить байесовский вывод, мы можем разбить эту задачу на две части:

---

[1] Индийский ученый, специализирующийся на вопросах обучения.

1. Что мы знали о числе *N* до того, как мы увидели эти данные (локомотив с номером 60)?
2. Для любого данного значения числа *N* каково правдоподобие того, что мы увидим этот локомотив (получим данные)?

Ответ на первый вопрос – это приор. Ответ на второй вопрос – правдоподобие. У нас не слишком много оснований выбрать приор, но мы можем начать с чего-либо простого, а затем рассмотреть альтернативы. Давайте предположим, что *N* одинаково вероятно равно любому числу от 1 до 1000.

```
hypos = xrange(1, 1001)
```

Теперь все, что нам надо, – это найти функцию правдоподобия. В гипотетическом составе парка локомотивов – *N* локомотивов. Какова вероятность, что мы увидим локомотив под номером 60? Если мы предположим, что речь идет только об одной железнодорожной компании (или нас интересует только одна компания) и, что одинаково вероятно, мы увидели любой из ее локомотивов, тогда наш шанс увидеть какой-то специфический локомотив равен 1/*N*.

Напишем функцию правдоподобия:

```
class Train(Suite):
    def Likelihood(self, data, hypo):
        if hypo < data:
            return 0
        else:
            return 1.0/hypo
```

Это выглядит знакомо. Функции правдоподобия для задачи о локомотивах и задачи об игральных костях идентичны.

Обновление:

```
suite = Train(hypos)
suite.Update(60)
```

Мы получаем на выходе слишком много гипотез, поэтому я отобразил результат на графике (рис. 3.1). Неудивительно, что все числа *N* меньше 60 исключены.

Наиболее вероятная величина, как можно догадаться, равна 60.

Но, возможно, это неверно сформулированная цель. Альтернативой является подсчет апостериорного распределения:

```
def Mean(suite):
    total = 0
    for hypo, prob in suite.Items():
        total += hypo * prob
    return total

print Mean(suite)
```

Или можно использовать очень похожий метод, обеспечиваемый Pmf:

```
print suite.Mean()
```

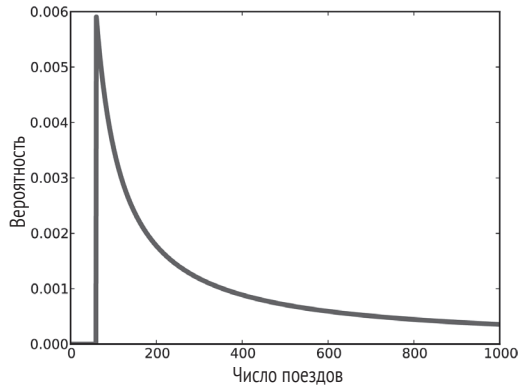


Рис. 3.1 ❖ Апостериорное распределение для задачи о локомотивах, основанное на унифицированном приоре

Среднее постериора равно 333. Поэтому будет неплохой идеей минимизировать приор. Если вы будете это делать снова и снова, используя средний постериор, то при большом количестве итераций ваша оценка будет минимизировать среднеквадратическую ошибку (см. http://en.wikipedia.org/wiki/Minimum_mean_square_error).

Вы можете загрузить этот пример из http://thinkbayes.com/train.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Что насчет этого приора?

Чтобы продвинуться в задаче о локомотивах, нам пришлось делать предположения, и некоторые из них были довольно произвольными. Особенно в том, что мы выбрали унифицированный приор от 1 до 1000 без какого-либо обоснования выбора 1000 или выбора унифицированного распределения.

Совсем не исключено, что у железнодорожной компании в эксплуатации находится 1000 локомотивов, и разумно предполагать большее или меньшее их число. Поэтому можно задаться вопросом, насколько распределение приора чувствительно к этим предположениям. С таким небольшим количеством данных одно наблюдение – это вероятно возможно.

Вспомним, что при унифицированном приоре от 1 до 1000 среднее значение постериора 333. При верхней границе 500 мы получаем средний постериор 207, а при верхней границе 2000, средний постериор равен 552.

Это плохо. Есть два пути продвинуться дальше:

- получить больше данных;
- получить больше предварительной информации.

При большем количестве данных апостериорные распределения, основанные на разных приорах, склонны сходиться. В качестве примера предположим,

что в добавление к поезду номер 60 мы видели также поезда под номерами 30 и 90. Мы тогда можем обновить распределение таким образом:

```
for data in [60, 30, 90]:
    suite.Update(data)
```

С такими данными среднее значение постериоров равно:

Верхняя граница	Средний постериор
500	152
1000	164
2000	171

Разница меньше.

Альтернативный приор

Если большее количество данных недоступно, есть другой вариант – получение большей предварительной информации. Может быть, неразумно предполагать, что компания имеет парк из 1000 локомотивов. Это так же вероятно, что компания владеет всего лишь одним локомотивом.

Приложив некоторые усилия, мы, вероятно, сможем найти перечень железнодорожных компаний, эксплуатирующих локомотивы в районе наблюдения. Или мы можем поинтересоваться у эксперта по железнодорожным перевозкам и получить информацию о типовом размере железнодорожных компаний.

Но даже без получения информации в специализированных учреждениях об экономике железнодорожных компаний мы можем сделать собственное разумное заключение. В большинстве промышленных областей много небольших компаний, больше средних компаний и только одна или две очень крупные компании. Фактически, как показал Роберт Акстелл[1] (Robert Axtell) в своей статье в журнале Science (см. http://www.sciencemag.org/content/293/5536/1818.full.pdf), распределение размера компаний подчиняется **степенному закону** (power law).

Этот закон устанавливает следующее: если имеется 1000 компаний, имеющих меньше 10 локомотивов, то там может быть 100 компаний со 100 локомотивами, 10 компаний с 1000 локомотивов и, возможно, одна компания с 10 000 локомотивов.

Математически степенной закон означает, что число компаний данного размера обратно пропорционально их размеру, или

$$\mathrm{PMF}(x) \propto \left(\frac{1}{x}\right)^{\alpha},$$

где PMF(x) – функция плотности распределения вероятности (mass function) с параметрами x и α, где α часто близка к 1.

[1] Роберт Акстелл – профессор университета Джорджа Масона, Виргиния, США.

Мы можем создать приор на основе степенного закона следующим образом:

```
class Train(Dice):

    def __init__(self, hypos, alpha=1.0):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, hypo**(-alpha))
        self.Normalize()
```

Код, который создает приор:

```
hypos = range(1, 1001)
suite = Train(hypos)
```

Верхняя граница, как и в предыдущих решениях, произвольна. Но с приором на основе степенного закона постериор менее чувствителен к этому выбору.

На рис. 3.2 показан новый постериор на основе степенного закона в сравнении с постериором на основе унифицированного приора. Используя предварительную информацию, представленную в приоре на основе степенного закона, мы можем исключить значения *N* больше 700.

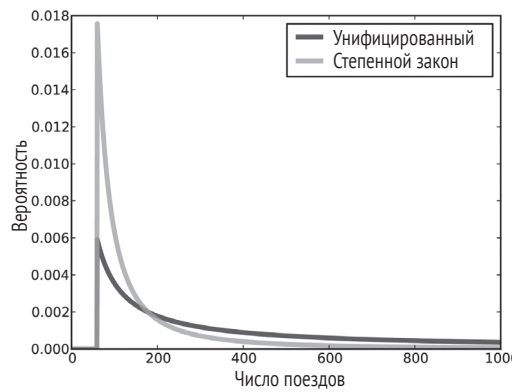


Рис. 3.2 ❖ Априорное распределение, основанное на степенном закона приора, в сравнении с унифированным приором

Если мы примем этот приор и увидим поезда 30, 60 и 90, то средние постериоры будут следующими:

Верхняя граница	Средний постериор
500	131
1000	133
2000	134

Теперь разница значительно меньше. Фактически с произвольно большей верхней границей среднее преобразуется в 134.

Доверительный интервал

После того как вы подсчитали апостериорное распределение, часто полезно подвести итог с помощью точечной или интервальной оценки. Для точечных оценок обычно это среднее, медиана или величина максимального правдоподобия.

Для интервальных оценок обычно получают два значения, которые подсчитываются так, чтобы неизвестная величина с 90%-ной вероятностью лежала в данном интервале (или с какой-либо другой вероятностью). Эти величины называют **доверительным интервалом**.

Проще всего вычислить доверительный интервал, добавив вероятностей в апостериорное распределение, и зафиксировать величины, которые соответствуют вероятностям 5% и 95%. Другими словами, 5-й и 95-й процентили.

thinkbayes предоставляет функцию, которая позволяет вычислить процентили:

```
def Percentile(pmf, percentage):
    p = percentage / 100.0
    total = 0
    for val, prob in pmf.Items():
        total += prob
        if total >= p:
            return val
```

Код, который использует ее:

```
interval = Percentile(suite, 5), Percentile(suite, 95)
print interval
```

Для предыдущего примера – задачи о локомотивах с приором по степенному закону и тремя поездами – 90%-ный доверительный интервал составляет (91, 243). Величина этого интервала, строго говоря, еще оставляет достаточно большую неопределенность в том, сколько в компании локомотивов.

Кумулятивные функции распределения

В предыдущем разделе мы вычисляли процентили путем итераций величин вероятностей в Pmf. Если нам необходимо вычислить больше, чем несколько процентилей, то более эффективно использовать кумулятивную функцию распределения, или Cdf.

Cdf и Pmf – это эквиваленты в том смысле, что они содержат одну и ту же информацию о распределении и вы всегда можете конвертировать одну в другую. Преимущество Cdf состоит в том, что вы можете вычислить процентили более эффективно.

thinkbayes предоставляет класс `Cdf`, который представляет собой кумулятивную функцию распределения. `Pmf` предоставляет метод, который создает соответствующий Cdf:

```
cdf = suite.MakeCdf()
```

А Cdf предоставляет функцию, называемую Percentile:

```
interval = cdf.Percentile(5), cdf.Percentile(95)
```

Конвертирование из Pmf в Cdf занимает время, пропорциональное количеству величин len(pmf). Cdf запоминает величины и вероятности в перечнях, отсортированных таким образом, чтобы время от наблюдения вероятности до получения соответствующей величины было «логарифмическим временем», то есть временем, пропорциональным логарифму количества величин. Время от наблюдения величины до получения соответствующей вероятности тоже логарифмическое. Поэтому Cdf при большом объеме вычислений более эффективен.

Примеры этого раздела находятся на http://thinkbayes.com/train3.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Задача о немецком танке

Во время Второй мировой войны Отдел ведения экономической войны американского посольства в Лондоне использовал статистический анализ для оценки уровня производства Германией танков и другого вооружения[1].

Западные союзники захватили технические паспорта, реестры и протоколы ремонта ходовых частей и двигателей конкретных танков.

Анализ этих документов показал следующее: серийные номера, присвоенные производителем, и тип танка в группах из 100 номеров расположены так, что номера в каждой группе использовались последовательно, и при этом в каждой группе использовались не все номера. Поэтому внутри каждой группы из 100 номеров задача оценки производства Германией танков может быть сведена к задаче о локомотивах.

Основываясь на такого рода соображениях, американские и английские исследователи получили оценку гораздо меньшего количества танков, чем оценки, полученные разведкой из других источников. По окончании войны документы подтвердили, что оценки аналитиков были значительно точнее.

Аналитиками также выполнялся подобный анализ для шин, грузовиков, ракет и другого оборудования. Тем самым была проведена точная и действенная экономическая разведка.

Задача о немецких танках представляет исторический интерес. Но эта задача является примером реального применения в жизни статистической оценки. Многие примеры в этой книге пока что касались игрушечных проблем. Но очень скоро мы примемся за реальные задачи. Я думаю, что преимущество байесовского анализа и применяемых нами методов вычислений в том, что при исследованиях данный анализ обеспечивает весьма короткий путь от базового введения до максимального результата.

---

[1] *Раггл и Броди*. Практика экономической разведки во Второй мировой войне // Журнал американской статистической ассоциации. Т. 42. № 237 (Мюнхен,1947).

Обсуждение

Среди приверженцев Байеса преобладают два подхода к выбору распределения приора. Одни рекомендуют выбор приора, наилучшим образом представляющий информационные предпосылки. В этом случае говорят об **информационном приоре**. Проблема при использовании информационного приора – в том, что могут быть использованы различные информационные предпосылки (или их различная интерпретация). Поэтому информативные приоры часто представляются субъективными.

Альтернативой является так называемый **неинформативный приор**. Неинформативный приор стремится быть настолько неограниченным, насколько это возможно, позволяя данным говорить самим за себя. В некоторых случаях вы можете идентифицировать унифицированный приор, имеющий некие желаемые свойства, как, например, представляемую минимальную априорную информацию об оцениваемой величине.

Неинформативные приоры более привлекательны, поскольку они кажутся более объективными. Но я в большинстве случаев ратую за использование информативных приоров. Почему? Во-первых, байесовский анализ всегда опирается на решения, основанные на *моделировании*. Выбор приора является одним из таких решений. Но он не единственный и, возможно, даже не самый субъективный. Поэтому даже если неинформативный приор более объективен, анализ продолжает оставаться субъективным.

Кроме того, при решении практических задач вы, вероятно, окажетесь в одной из двух ситуаций: у вас или много данных, или мало. Если вы имеете много данных, то выбор приора не очень сложен. Информативный или неинформативный приор даст примерно одинаковый результат. Пример этого мы увидим в последующих главах.

Однако если, как в задаче о танках Германии, у вас нет большого количества данных, использование релевантной предварительной информации (такой как распределение по степенному закону) приводит к существенной разнице.

Если, как в случае задачи о танках Германии, вы должны принимать решения, результат которых на грани жизни и смерти, вам, скорее всего, следует использовать всю имеющуюся в вашем распоряжении информацию, а не сохранять иллюзию объективности, делая вид, что вы знаете меньше, чем знаете на самом деле.

Упражнение

Упражнение 3.1

Чтобы написать функцию правдоподобия в задаче о локомотивах, нам нужно было ответить на вопрос: «Если железнодорожная компания имеет N локомотивов, то какова вероятность того, что мы увидим локомотив под номером 60?»

Ответ зависит от того, какую процедуру выборки, наблюдая за локомотивом, мы используем. В этой главе была допущена степень неопределенности для

одной железнодорожной компании (или мы представили, что нас интересует только одна компания).

Но можно предположить, что вместо одной имеется много компаний с различными номерами поездов. Также можно предположить, что вы равновероятно видите любой поезд, задействованный любой компанией. В этом случае функция правдоподобия будет другой, потому что вы с большей вероятностью увидите поезд, задействованный большой компанией.

Глава 4

Больше об оценивании

Задача о евро

В книге «Теория информации, вывод и обучающие алгоритмы» (*Information Theory, Inference, and Learning Algorithms*) Дэвид МакКей (David MacKay) ставит такой проблемный вопрос:

> В газете «Гардиан» (The Guardian) 4 января 2002 года появилась статистическая заметка:
>
> > «Монета в 1 евро бельгийского производства после 250 вращений на ребре 140 раз упала вверх орлом и 110 раз вверх решкой. “Это мне кажется очень подозрительным, – заявил Барри Блайт (Barry Blight), читающий лекции по статистике в Лондонской экономической школе. – Если центр массы монеты был несмещенным, то вероятность получения такого экстремального результата составляла бы менее 7%”».
>
> Но свидетельствуют ли такие данные о том, что центр массы монеты, скорее, смещен или нет?

На этот вопрос мы будем отвечать в два этапа. Первым этапом станет оценка вероятности того, что монета упадет решкой вверх. За второй этап мы попробуем определить, насколько верна гипотеза о том, что центр масс монеты смещен.

Вы можете скачать код в этом разделе из http://thinkbayes.com/euro.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Любая монета имеет некоторую вероятность x упасть решкой вверх после вращения на ребре. Разумно предположить, что эта величина x зависит от некоторых физических характеристик монеты, главным образом от распределения веса в самой монете.

Если монета хорошо сбалансирована, мы можем ожидать величину x, близкую к 50%. Но для монеты со смещенным центром массы значение x будет существенно другим. Для оценки величины x мы можем использовать теорему Байеса и данные, полученные при экспериментах.

Давайте зададимся 101 гипотезой, где H_x-гипотеза – это что вероятность решки составляет x% для значений от 0 до 100. Мы вернемся позже для рассмотрения других приоров.

Функция правдоподобия относительно проста: **если** H_x справедлива, то вероятность решки равна $x/100$ и вероятность орла $1 - x/100$.

```
class Euro(Suite):

    def Likelihood(self, data, hypo):
        x = hypo
        if data == 'H':
            return x/100.0
        else:
            return 1 - x/100.0
```

Код, создающий suite:

```
suite = Euro(xrange(0, 101))
dataset = 'H' * 140 + 'T' * 110

for data in dataset:
    suite.Update(data)
```

Результат показан на рис. 4.1.

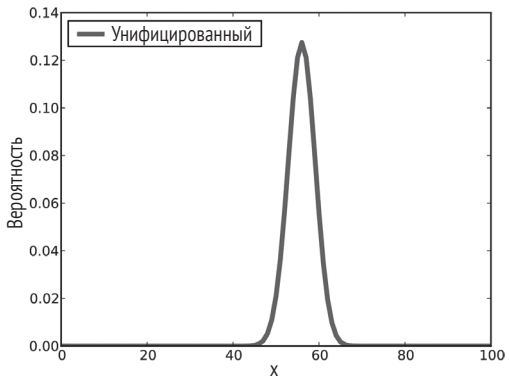


Рис. 4.1 ❖ Апостериорное распределение для задачи о евро и унифицированного приора

Итоговый постериор

Существует два варианта получения итогового апостериорного распределения. Один вариант – найти наиболее вероятную величину апостериорного распределения. В thinkbayes предусмотрена функция, которая выполняет следующее:

```
def MaximumLikelihood(pmf):
    """Returns the value with the highest probability."""
    prob, val = max((prob, val) for val, prob in pmf.Items())
    return val
```

В этом случае результат равен 56. Этот результат дает наблюдаемый процент падения монеты вверх орлом 140/250 = 0.56%. Таким образом, данный результат предполагает (и это предположение правильно), что наблюдаемый процент есть оценка максимального правдоподобия для выборки.

Мы, путем вычисления среднего и медианы, также можем получить итоговый постериор.

```
print 'Mean', suite.Mean()
print 'Median', thinkbayes.Percentile(suite, 50)
```

Среднее равно 55.95; медиана равна 56. Наконец, вычисляем доверительный интервал:

```
print 'CI', thinkbayes.CredibleInterval(suite, 90)
```

Результат – (51,61).

Теперь вернемся к поставленному вопросу. Нам хотелось узнать, смещен центр массы у монеты или нет.

Мы увидели, что доверительный интервал постериора не равен 50%. Это предполагает, что центр массы монеты является смещенным.

Но это не точный ответ на поставленный в начале главы вопрос. МакКей спрашивает: «Свидетельствуют эти данные о том, что центр массы монеты, скорее, смещен или нет?» Чтобы ответить на этот вопрос, нам необходимо быть более точным в утверждении, что данные формируют свидетельство для гипотезы. Этот вопрос мы обсудим в следующей главе.

Но, перед тем как продолжить, я хочу обратить внимание на один источник возможного заблуждения. Поскольку мы хотим знать, является ли центр массы монеты несмещенным, нам было бы интересно спросить: какова вероятность того, что *x* равен 50%.

```
print suite.Prob(50)
```

Результат 0.021. Это не существенная величина. Решение оценить 101 гипотезу было произвольным. Мы могли бы разделить этот диапазон на большее или меньшее количество отрезков. В этом случае вероятность для соответствующей гипотезы будет больше или меньше.

Подавление приоров

Мы начинали исследование с унифицированного приора. Но, возможно, это не самый лучший выбор. Если центр массы монеты смещен, можно предположить, что значение величины *x* значительно отклоняется от 50%. Однако маловероятно, что бельгийский евро не сбалансирован на 10% или целых 90%.

Возможно, более разумно выбрать приор, дающий более высокую вероятность для значений *x*, близких к 50%, и меньшую вероятность для экстремальных значений. В качестве примера был построен треугольный приор, показанный на рис. 4.2.

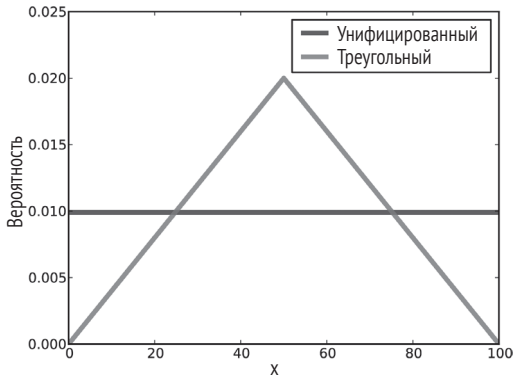


Рис. 4.2 ❖ Унифицированный и треугольный приоры для задачи о евро

Код, создающий приор:

```
def TrianglePrior():
    suite = Euro()
    for x in range(0, 51):
        suite.Set(x, x)
    for x in range(51, 101):
        suite.Set(x, 100-x)
    suite.Normalize()
```

На рис. 4.2 для сравнения показан результат и для унифицированного приора. Обновление этого приора с тем же множеством данных создает очень похожее апостериорное распределение (рис. 4.3). Медианы и доверительные интервалы идентичны, а среднее отличается менее, чем на 0.5%.

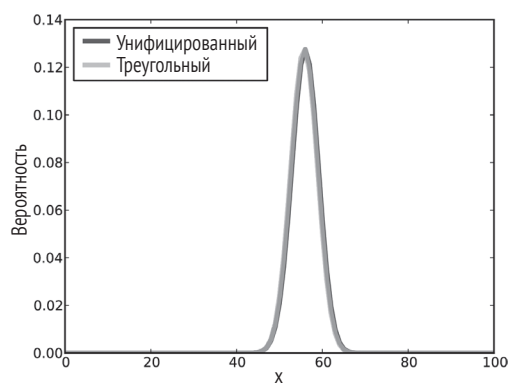


Рис. 4.3 ❖ Апостериорное распределение для задачи о евро

Это пример **подавления приоров** при достаточном количестве данных. Те, кто начинает с различных приоров, обнаружат тенденцию сходимости к тому же постериору.

Оптимизация

Показанный ранее код должен быть легко читаемым. Но эффективность его невысока. Вообще, автору нравится разрабатывать явно корректный код. А затем проверить, достаточно ли он быстр для решаемых задач. Если код отвечает всем требованиям, значит, дальше оптимизировать его незачем. Для примера о евро, если нам важно, насколько быстро выполняется программа, существует несколько способов **оптимизации программного кода**.

Первая возможность оптимизации – уменьшить число нормализаций suite. В оригинальном коде мы вызываем `Update` для каждого цикла:

```
dataset = 'H' * heads + 'T' * tails

for data in dataset:
    suite.Update(data)
```

А здесь вы видите сам `Update`:

```
def Update(self, data):
for hypo in self.Values():
    like = self.Likelihood(data, hypo)
    self.Mult(hypo, like)
return self.Normalize()
```

Каждая итерация проводит к обновлению гипотезы. Затем вызывается `Normalize`, который осуществляет новую **итерацию гипотезы**.

Чтобы сэкономить время проведения всех обновлений до нормализации, воспользуемся методом, предусмотренным `Suite`. Название этого метода – `Updateset`. Вот что делает `Updateset`:

```
def UpdateSet(self, dataset):
    for data in dataset:
        for hypo in self.Values():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo, like)
    return self.Normalize()
```

А здесь показано, как мы это можем осуществить:

```
dataset = 'H' * heads + 'T' * tails
suite.UpdateSet(dataset)
```

Данная оптимизация выполнение программного кода ускоряет. Но время выполнения все еще пропорционально количеству данных. Мы можем дополнительно ускорить выполнение программного кода. Для этого следует перепи-

сать Likelihood таким образом, чтобы обрабатывать сразу все множество данных, а не по одному данному за один цикл.

В оригинальной версии программного кода данные представлены строкой, в которой кодируется возможность падения монеты орлом или решкой вверх:

```
def Likelihood(self, data, hypo):
    x = hypo / 100.0
    if data == 'H':
        return x
    else:
        return 1-x
```

В качестве альтернативы мы можем кодировать множество данных в виде кортежа из двух целых чисел: номеров орла и решки. В этом случае правдоподобие будет выглядеть так:

```
def Likelihood(self, data, hypo):
    x = hypo / 100.0
    heads, tails = data
    like = x**heads * (1-x)**tails
    return like
```

А затем с помощью двух строк вызвать Update:

```
heads, tails = 140, 110
suite.Update((heads, tails))
```

Поскольку повторяющиеся перемножения были заменены экспонентой, в этой версии кода время исполнения программы остается одинаковым для любого количества циклов.

Бета-распределение

Есть еще один способ оптимизации, позволяющий ускорить решение задачи.

До сих пор мы использовали PMF-объект, чтобы представить дискретное множество значений для величины x. Теперь используем непрерывное распределение, так называемое бета-распределение (см. http://en.wikipedia.org/wiki/Beta_distribution).

Бета-распределение определено в интервале от 0 до 1 (включая оба значения). Так что естественно выбрать его для описания пропорций и вероятностей. Но дальше – лучше.

Оказывается, для байесовского обновления с биноминальной функцией правдоподобия бета-распределение является **сопряженным приором**. Данный прием был нами использован в предыдущем разделе. То есть если априорное распределение величины x является бета-распределением, то постериор тоже имеет бета-распределение. Но это еще не все.

Форма бета-распределения зависит от двух параметров, обозначаемых α и β, или alpha и beta. Если приор имеет бета-распределение с параметрами alpha

и beta и мы наблюдаем h орлов и t решек, постериор и есть бета-распределение с параметрами alpha+h и beta+t. Другими словами, мы можем делать обновление с двумя добавлениями.

Это, конечно, хорошо. Но все заработает, если мы сможем найти бета-распределение, которое станет хорошим выбором для приора. К счастью, для многих реальных приоров существует бета-распределение, являющееся, по крайней мере, хорошей аппроксимацией, и для унифицированного приора прекрасно подходит. Бета-распределение с параметрами alpha=1 и beta=1 является унифицированным (равномерным) от 0 до 1.

Давайте посмотрим, как из вышесказанного можно получить преимущество. В thinkbayes.py предусмотрен класс, представляющий бета-распределение:

```
class Beta(object):

    def __init__(self, alpha=1, beta=1):
        self.alpha = alpha
        self.beta = beta
```

Посредством __init__ создается унифицированное распределение. Update представляет байесовское обновление:

```
    def Update(self, data):
        heads, tails = data
        self.alpha += heads
        self.beta += tails
```

Данные – это пара целых чисел, представляющих количество падений монеты орлом и решкой вверх.

Теперь мы имеем другой путь решения задачи о евро:

```
beta = thinkbayes.Beta()
beta.Update((140, 110))
print beta.Mean()
```

Beta обеспечивает Mean, которое вычисляет простую функцию от alpha и beta:

```
    def Mean(self):
        return float(self.alpha) / (self.alpha + self.beta)
```

Для задачи о евро среднее составляет 56%. Это результат, полученный и при использовании Pmfs.

Beta также предусматривает EvalPdf, оценивающий плотность функции вероятности бета-распределения:

```
    def EvalPdf(self, x):
        return x**(self.alpha-1) * (1-x)**(self.beta-1)
```

Наконец, Beta предусматривает MakePmf, который, для генерации дискретной аппроксимации бета-распределения, использует EvalPdf.

Обсуждение

В этой главе мы решили одну и ту же задачу с двумя разными приорами и нашли, что при большом множестве данных приор подавляется. Если двое стартуют с различными представлениями о приорах, они, в общем-то, при получении большего количества данных обнаружат, что их апостериорные распределения сходятся. В некоторой точке разница между этими распределениями настолько мала, что не имеет практического значения.

В этом случае уменьшается беспокойство об объективности. Это обсуждалось в предыдущей главе. Для многих задач реального мира даже самые твердые представления о приоре в конечном итоге должны подтверждаться данными.

Но это не всегда так. Во-первых, следует помнить, что весь байесовский анализ основан на модельных решениях. Если модель не будет выбрана, можно по-разному интерпретировать данные. Тогда даже при одних и тех же данных мы получим различные правдоподобия, и наши представления о постериорах могут расходиться.

Заметим также, что при байесовском обновлении мы перемножаем каждый приор вероятности на правдоподобие. Поэтому если $p(H)$ равно 0, то и $p(H|D)$ тоже равен 0. В задаче о евро, если вы убеждены, что значение x, меньше 50%, и вы принимаете вероятность, равную 0, для всех остальных гипотез, никакое количество данных не убедит вас в обратном.

Это наблюдение является основой **правила Кромвелля** (Cromwell's rule). Это правило рекомендует избегать назначения вероятности приора равной нулю для любой гипотезы, даже которая отдаленно возможна (см. http://en.wikipedia.org/wiki/Cromwell's_rule).

Правило Кромвелля названо так по имени Оливера Кромвелля (Oliver Cromwell). Он писал: «Умоляю, ради Христа, подумайте, не ошибаетесь ли вы». Для приверженцев Байеса это хороший совет! Даже если и несколько преувеличенный.

Упражнения

Упражнение 4.1

Предположим, что вместо непосредственных наблюдений за монетой вы используете для этого некий инструмент, который не всегда правильно сообщает о наблюдениях. Более конкретно: предположим с вероятностью y, что о решках сообщается как об орлах, а об орлах как о выпавших решках.

Напишите класс, который оценивает смещение монеты в заданной серии выборок и величину y.

Как расширение апостериорного распределения зависит от y?

Упражнение 4.2

Упражнение навеяно вопросом, который был задан одним из участников по имени dominosci на форуме Реддит (http://reddit.com/r/statistics).

Реддит (Reddit) – это онлайновый форум со множеством групп по интересам, называемых сабреддиты (subreddits). Пользователи, называемые реддиты (redditors), размещают ссылки на онлайн-контент и другие веб-страницы. Другие реддиты «голосуют» по ссылке, давая голос «за» высококачественным ссылкам и «против» некачественным или нерелевантным ссылкам.

Проблема, выявленная dominosci, заключалась в том, что одни реддиты надежнее других, но на форуме Реддит не это учитывается.

Предлагалось разработать систему, задача которой – при голосовании оценивать качество ссылки в соответствии с надежностью голосующего реддитора, а надежность реддита обновлять в зависимости от качества предлагаемых ссылок.

Один из подходов – смоделировать качество ссылок как вероятность получения голосов «за» и смоделировать надежность реддитора как вероятность правильно данных голосов «за», получаемых за высококачественные ссылки.

Создайте класс определений для реддиторов и ссылок, а также функцию обновления обоих объектов, как только реддитор отдает голос.

Глава 5

Отношение вероятностей и добавления

Отношение вероятностей

Представление вероятности числом между 0 и 1 – это не единственная возможность ее представления. Если вы когда-либо делали ставку на скачках или на футбольный матч, вы, вероятно, встречались с другим представлением вероятности, называемым **отношение вероятностей**.

Вы могли слышать выражение, такое как, например, «отношение три к одному». Но что это означает, могли и не знать. **Отношение в пользу** некого события – это отношение вероятности, что данное событие случится, к тому, что оно не случится.

То есть если по моему мнению команда, за которую я болею, имеет 75% шансов выиграть, то я могу сказать: отношение вероятности в пользу этой команды три к одному, так как у нее шанс победить в три раза выше, чем проиграть.

Вы можете отношение вероятностей записать в десятичной форме. Но гораздо чаще вероятность записывают целыми числами. То есть «три к одному» записывается как 3:1.

Если вероятность невелика, принято говорить об **отношении вероятностей не в пользу**, чем отношение вероятностей в пользу. Например, если по моему мнению лошадь, на которую сделана ставка, имеет 10% шансов выиграть, можно сказать, что отношение вероятностей не в пользу выигрыша и равно 9:1.

Вероятности и отношения вероятностей – это различные представления одной и той же информации. Для данной вероятности можно вычислить отношение вероятности следующим образом:

```
def Odds(p):
    return p / (1-p)
```

Вероятность в пользу, представленную в десятичной форме, можно конвертировать в вероятность следующим образом:

```
def Probability(o):
    return o / (o+1)
```

Если отношение вероятностей представляется в виде числителя и знаменателя, то эту вероятность можно преобразовать в вероятность следующим образом:

```
def Probability2(yes, no):
    return yes / (yes + no)
```

В случае с отношением вероятностей полезно представить, что другие люди думают, например, о скачках. Если 20% из них полагают, что выбранная лошадь выиграет, то остальные 80% так не думают. Следовательно, в этом случае отношение вероятностей в пользу данной лошади 20:80, или 1:4. Если отношение вероятностей 5:1 не в пользу выбранной лошади, значит, пять из шести человек думают, что она проиграет. Поэтому вероятность выигрыша составляет 1/6.

Теорема Байеса в форме отношения вероятностей

В главе 1 была приведена теорема Байеса в **вероятностной форме**:

$$\mathrm{p}(H \mid D) = \frac{\mathrm{p}(H)\,\mathrm{p}(D \mid H)}{\mathrm{p}(D)}.$$

Если у нас есть две гипотезы A и B, то мы можем написать отношение апостериорных вероятностей следующим образом:

$$\frac{\mathrm{p}(A \mid D)}{\mathrm{p}(B \mid D)} = \frac{\mathrm{p}(A)\,\mathrm{p}(D \mid A)}{\mathrm{p}(B)\,\mathrm{p}(D \mid B)}.$$

Заметим, что константа нормализации $\mathrm{p}(D)$ в этом уравнении отсутствует.

Если A и B – взаимно исключаемые и совместно исчерпывающие события, значит, $\mathrm{p}(B) = 1 - \mathrm{p}(A)$. Поэтому мы можем написать отношение приоров и отношение постериоров как отношения вероятностей.

Записав $\mathrm{o}(A)$ для отношения вероятностей в пользу A, получим:

$$\mathrm{o}(A \mid D) = \mathrm{o}(A)\frac{\mathrm{p}(D \mid A)}{\mathrm{p}(D \mid B)}.$$

Это уравнение свидетельствует о том, что апостериорное отношение вероятностей – это априорное отношение вероятностей, умноженное на отношение правдоподобия.

Данное отношение представляет собой теорему Байеса в форме отношения вероятностей.

Эта форма теоремы Байеса максимально удобна для вычисления обновления письменно или в уме. Для примера вернемся к задаче о булочках.

> Предположим, имеется две корзины с булочками. Корзина 1 содержит 30 ванильных булочек и 10 шоколадных. Корзина 2 содержит 20 ванильных и 20 шоколадных булочек.

> Допустим, вы случайным образом выбрали одну из корзин и, не глядя в эту корзину, вытащили одну булочку. Она оказалась ванильной. Какова вероятность, что вы вынули ее из корзины 1?

Априорная вероятность составляет 50%. Поэтому априорное отношение вероятностей равно 1:1, или просто 1. Отношение правдоподобия равно $\frac{3}{4}\Big/\frac{1}{2}$, или 3/2. Отсюда апостериорное отношение вероятностей равно 3:2, а вероятность – 3/5.

Группа крови Оливера

Еще одна задача из книги МакКея «Теория информации, вывод и обучающие алгоритмы»:

> Два человека оставили следы своей крови на месте преступления. У подозреваемого Оливера взяли пробу крови и определили, что это группа «О». Группы оставленных на месте преступления следов крови оказались типа «О» (самая распространенная группа крови среди местного населения, встречающаяся с частотой 60%) и «АВ» (редкая группа с частотой 1%). Свидетельствуют ли эти данные (следы крови на месте преступления) в пользу предположения, что Оливер был одним из тех двоих, кто оставил следы крови?

Для ответа на этот вопрос нам необходимо подумать, что означает наличие данных для свидетельства в пользу (или против) некоторой гипотезы. Интуитивно можно сказать: если гипотеза с этими данными более вероятна, значит, данные в пользу гипотезы.

В задаче с булочками априорное отношение вероятностей равно 1:1, то есть вероятности 50%. Апостериорное отношение вероятностей равно 3:2, или вероятности 60%. Поэтому мы могли сказать, что ванильная булочка свидетельствует в пользу корзины 1.

Теорема Байеса в форме отношения вероятностей дает возможность сделать это интуитивное предположение более точным.

Вновь напишем:

$$\mathrm{o}(A \mid D) = \mathrm{o}(A)\frac{\mathrm{p}(D \mid A)}{\mathrm{p}(D \mid B)}.$$

И разделим обе части уравнения на o(A):

$$\frac{\mathrm{o}(A \mid D)}{\mathrm{o}(A)} = \frac{\mathrm{p}(D \mid A)}{\mathrm{p}(D \mid B)}.$$

Выражение в левой части уравнения является отношением апостериорного и априорного отношений вероятностей. Правая часть уравнения – это отношение правдоподобия, называемое также **байесовским коэффициентом**.

Если байесовский коэффициент больше 1, значит, данные в большей степени свидетельствуют в пользу A, чем в пользу B. И поскольку отношение отно-

шений вероятности больше 1, то это означает, что величина отношения вероятностей в свете полученных данных больше, чем она была до этого.

Если байесовский коэффициент меньше 1, значит, данные менее в пользу *A*, чем *B*. Поэтому шансы в пользу *A* снижаются.

Наконец, если байесовский коэффициент равен точно 1, значит, данные одинаково вероятны в пользу любой из гипотез. Поэтому шансы остаются прежними.

Теперь вернемся к задаче о группе крови Оливера. Если Оливер – один из тех, кто оставил следы крови на месте преступления, то он является причиной наличия следа крови группы «O». Поэтому вероятность полученных данных – это просто вероятность, что некий случайный человек из местного населения имеет группу крови «AB». А эта вероятность составляет 1%.

Если Оливер не оставлял следов своей крови на месте преступления, тогда мы должны принимать в расчет оба следа крови. Если случайным образом будут выбраны два человека из местного населения, то каков шанс, что группа крови одного из них будет «O», а группа крови другого – «AB»? В этом случае у нас будет два варианта. Первый – что человек, которого мы выбрали, имеет группу крови «O», а группа крови второго – «AB». Или наоборот, группа крови первого выбранного человека «AB», а второго – «O». Общая вероятность этих событий составляет 2(0.6)(0.01) = 1.2%.

Правдоподобие данных немного выше, если Оливер *не является* одним из двух, кто оставил след на месте преступления. Поскольку тогда оба следа крови, по существу, являются свидетельством невиновности Оливера.

Этот пример слегка надуманный, но он показывает, что *противоречащий* интуиции результат, указывающий на согласующиеся с гипотезой данные, *необязателен в пользу* этой гипотезы.

Если этот результат идет вразрез с вашей интуицией, вам может помочь следующий ход рассуждений. Данные состоят из обычного события – кровь группы «O» и редкого события – кровь группы «AB». Если Оливер рассматривается в качестве причины обычного события, то это оставляет редкое событие трудно объяснимым. Если Оливер не рассматривается как причина оставленного следа крови группы «O», то мы имеем два шанса найти кого-то среди населения, имеющего группу крови «AB». И этот коэффициент 2 создает различие.

Добавления

Обновление (Update) является фундаментальной операцией в байесовской статистике. Обновление на основе априорного распределения и множества данных создает апостериорное распределение. Но реальные задачи обычно вовлекают и некоторое число других операций, включая масштабирование, дополнения и другие арифметические операции, максимизацию и минимизацию, перемешивание.

В этой главе мы рассмотрим дополнения и максимизацию. Другие операции будут представлены по мере необходимости.

Первый пример основан на игре «Подземелья и драконы». Это ролевая игра, в которой результаты решений игроков определяются обычно катящимися игровыми костями. Фактически перед началом игры игроки сами создают каждую черту своего характера – сила, интеллект, мудрость, сообразительность, телосложение, харизма, – бросая трижды 6-гранную кость и суммируя результат.

Интересно узнать получаемое в результате распределение сумм. Есть два способа это сделать:

1) *моделирование*: Pmf предоставляет распределение для одного 6-гранника. Вы можете получить случайную выборку, сложить результаты и аккумулировать распределение смоделированных сумм;
2) *перечисление*: два Pmf. Вы можете пересчитать все возможные пары величин и вычислить распределение этих сумм.

thinkbayes предоставляет функции для обоих способов.

Сначала мы покажем пример вычисления для первого способа.

Вначале класс, представляющий одиночную кость, мы определим как Pmf.

```
class Die(thinkbayes.Pmf):

    def __init__(self, sides):
        thinkbayes.Pmf.__init__(self)
        for x in xrange(1, sides+1):
            self.Set(x, 1)
        self.Normalize()
```

Теперь мы можем описать 6-гранную кость:

```
d6 = Die(6)
```

И использовать thinkbayes.SampleSum, чтобы сгенерировать выборку из 1000 бросаний кости.

```
dice = [d6] * 3
three = thinkbayes.SampleSum(dice, 1000)
```

SampleSum воспринимает перечень распределений (или Pmf-, или Cdf-объекты) и размер выборки n. Он генерирует n случайных сумм и возвращает их распределение как Pmf-объект.

```
def SampleSum(dists, n):
    pmf = MakePmfFromList(RandomSum(dists) for i in xrange(n))
    return pmf
```

SampleSum использует RandomSum также в thinkbayes.py:

```
def RandomSum(dists):
    total = sum(dist.Random() for dist in dists)
    return total
```

RandomSum запускает Random для каждого из распределений и суммирует результаты.

Недостаток такого моделирования заключается в том, что результат является лишь аппроксимацией. Когда n становится больше, результат становится точнее, но также увеличивается и время вычислений.

Второй способ – перечисление всех пар величин и вычисление суммы и вероятности каждой пары. Это осуществляется в Pmf.__add__:

```
# class Pmf

    def __add__(self, other):
        pmf = Pmf()
        for v1, p1 in self.Items():
            for v2, p2 in other.Items():
                pmf.Incr(v1+v2, p1*p2)
        return pmf
```

self, конечно, является Pmf; other может быть Pmf или чем-либо еще, обеспечивающим Items. Результат является новым Pmf. Время выполнения __add__ зависит от числа Items в self и other. Время пропорционально len(self) * len(other).

Здесь показано, как это время используется:

```
three_exact = d6 + d6 + d6
```

Когда вы используете оператор + в Pmf, Python вызывает __add__. В этом примере __add__ вызывается дважды.

На рис. 5.1 показан результат аппроксимации, полученный способом моделирования, и точный результат, полученный способом перечисления.

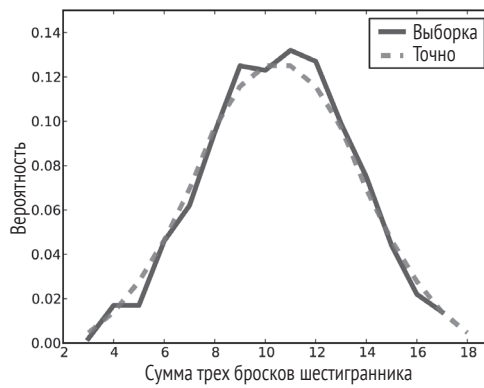


Рис. 5.1 ❖ Апроксимация и точное распределение для суммы трех бросков шестигранника

Pmf.__add__ основан на предположении, что случайные выборки из каждого Pmf независимы. В этом примере, где выполняется несколько бросков кости, это вполне приемлемо. В других случаях мы должны были бы расширить этот способ, чтобы использовать условные распределения.

Код этого раздела доступен в http://thinkbayes.com/dungeons.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Максимизации

При создании характера в игре «Подземелья и драконы» вы особенно заинтересованы в наилучших чертах, поэтому вам может быть интересно распределение максимума черт характера.

Существует три способа вычисления распределения максимума:

1) *моделирование*: Pmf представляет распределение для одиночного выбора, и вы можете генерировать случайные выборки, находить максимум и аккумулировать распределение моделируемых максимизаций;
2) *перечисление*: два Pmf. Вы можете пересчитать все возможные пары величин и вычислить распределение максимума;
3) *возведение в степень*: если мы преобразуем Pmf в Cdf, то существует простой и эффективный алгоритм нахождения Cdf-максимума.

Код, моделирующий максимизацию, почти идентичен коду моделирования сумм:

```
def RandomMax(dists):

    total = max(dist.Random() for dist in dists)
    return total

def SampleMax(dists, n):
    pmf = MakePmfFromList(RandomMax(dists) for i in xrange(n))
    return pmf
```

Все, что здесь было сделано, – это замена «sum» на «max». Код для перечисления также почти идентичен:

```
def PmfMax(pmf1, pmf2):
    res = thinkbayes.Pmf()
    for v1, p1 in pmf1.Items():
        for v2, p2 in pmf2.Items():
            res.Incr(max(v1, v2), p1*p2)
    return res
```

По существу, вы можете обобщить эту функцию, взяв подходящий оператор в качестве параметра.

Единственная проблема с этим алгоритмом в том, что если каждый Pmf имеет m величин, то время выполнения программы пропорционально m^2. Если мы хотим иметь максимум из k выборок, то это займет время, пропорциональное km^2.

При преобразовании Pmf в Cdf те же вычисления выполняются быстрее! Ключевым здесь является запоминание определения кумулятивных функций распределения:

$$CDF(x) = p(X \leqslant x),$$

где X – случайная переменная, обозначающая «величина из этого распределения, выбранная случайным образом». Так, для примера, Cdf(5) является вероятностью, что величина из этого распределения меньше или равна 5.

Если вытащить X из CDF_1, а Y из CDF_2 и вычислить максимум $Z = \max(X, Y)$, то каков шанс, что Z будет меньше или равен 5? В таком случае обе величины – и X, и Y – должны быть меньше или равны 5.

Если выборки из X и Y независимы, то

$$CDF_3(5) = CDF_1(5)CDF_2(5),$$

где CDF_3 является распределением Z. Мы выбрали величину 5, так как, по мнению автора, это делает формулу более читаемой. Но мы можем провести обобщение для любой величины z:

$$CDF_3(z) = CDF_1(z)CDF_2(z).$$

В особом случае, когда мы имеем k выборок из одного и того же распределения,

$$CDF_k(z) = CDF_1(z)^k.$$

Таким образом, чтобы найти распределение максимума из k величин, мы должны перечислить вероятности в данном Cdf и возвести их в степень k. `Cdf` обеспечит метод, который выполнит следующее:

```
# class Cdf

    def Max(self, k):
        cdf = self.Copy()
        cdf.ps = [p**k for p in cdf.ps]
        return cdf
```

`Max` берет определенное число выборок `k` и возвращает новый Cdf, представляющий распределение максимума из `k` выборок. Время выполнения для этого метода пропорционально m, числу элементов данных в Cdf.

`Pmf.Max` делает то же самое в Pmf. Необходимо выполнить несколько больший объем работы для конвертации Pmf в Cdf, и затрачиваемое время пропорционально $m \log m$. Но это все-таки лучше, чем квадратичная степень.

Наконец, приведу пример вычисления распределения наилучших черт характера:

```
best_attr_cdf = three_exact.Max(6)
best_attr_pmf = best_attr_cdf.MakePmf()
```

где `three_exact` определено в предыдущем разделе. Когда мы выведем результаты, то увидим, что шанс создания характера, по крайней мере, с одним атрибутом из 18 составляет примерно 3%. На рис. 5.2 показано распределение.

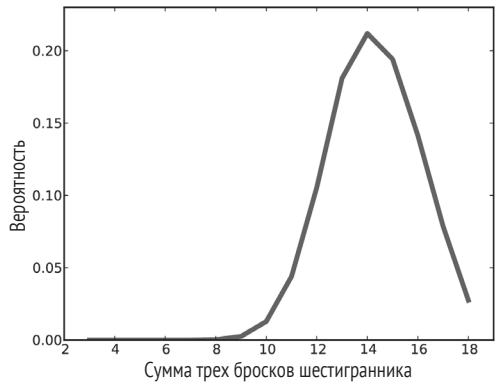


Рис. 5.2 ❖ Распределение максимума трех бросков шестигранника шесть раз

Перемешивание

Приведу еще один пример из игры «Подземелья и драконы». Предположим, у меня есть коробка со следующим содержимым:

```
 4-гранная кость – 5 шт.
 6-гранная кость – 4 шт.
 8-гранная кость – 3 шт.
12-гранная кость – 2 шт.
20-гранная кость – 1 шт.
```

Из коробки наугад выбирается кость и бросается. Какое распределение получится на выходе?

Если мы знаем, какая это кость, ответить легко. Кость с `n` гранями создает унифицированное распределение от 1 до `n`, включая границы.

Но если мы не знаем, какая это кость, то результирующее распределение является **смесью** унифицированных распределений с разными границами. В общем случае этот тип смеси не соответствует никакой простой математической модели. Тем не менее это распределение напрямую вычисляется в форме PMF.

Как всегда, одним из способов будет моделирование сценария, генерация случайной выборки и вычисление PMF этой выборки. Этот способ прост и быстро создает аппроксимацию решения. Но если мы хотим получить точное решение, необходим другой способ.

Давайте начнем с более простой версии задачи, в которой только две кости: одна с 6 гранями и одна с 8. Мы можем создать Pmf, представляющий каждую кость.

```
d6 = Die(6)
d8 = Die(8)
```

Затем создадим Pmf, представляющий смесь:

```
mix = thinkbayes.Pmf()
for die in [d6, d8]сим
    for outcome, prob in die.Items():
        mix.Incr(outcome, prob)
mix.Normalize()
```

Первый цикл перечисляет кости, второй перечисляет выходные данные и их распределения. Внутри цикла `Pmf.Incr` добавляет вклад от двух других распределений.

Этот код предполагает, что две кости одинаково вероятны. В более общем случае нам необходимо знать вероятность каждой кости, чтобы мы могли учесть вклад соответствующего выходного результата.

Сначала мы создадим Pmf, который отображает вероятность для каждой выбранной кости:

```
pmf_dice = thinkbayes.Pmf()
pmf_dice.Set(Die(4), 2)
pmf_dice.Set(Die(6), 3)
pmf_dice.Set(Die(8), 2)
pmf_dice.Set(Die(12), 1)
pmf_dice.Set(Die(20), 1)
pmf_dice.Normalize()
```

Затем нам необходима более общая версия алгоритма перемешивания:

```
mix = thinkbayes.Pmf()
for die, weight in pmf_dice.Items():
    for outcome, prob in die.Items():
        mix.Incr(outcome, weight*prob)
```

Теперь каждая кость имеет ассоциированный с ней вес (что предположительно делает ее взвешенной костью). Когда мы добавим каждый выходной результат к смеси, ее вероятность перемножается на вес.

На рис. 5.3 показан общий результат. Как ожидалось, величины от 1 до 4 наиболее вероятны, потому что они могут быть получены от любой кости. Величины более 12 маловероятны, потому что только одна кость в коробке может создать их (и она тратит на это меньше половины времени).

`Thinkbayes` предусматривает функцию, называемую `MakeMixture`, которая инкапсулирует этот алгоритм:

```
mix = thinkbayes.MakeMixture(pmf_dice)
```

Мы будем использовать `MakeMixture` в главах 7 и 8.

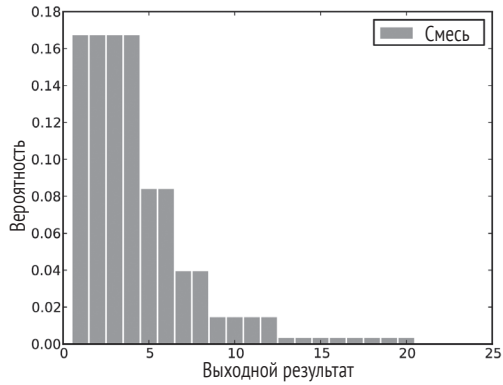


Рис. 5.3 ❖ Распределение на выходе для случайной кости из коробки

Обсуждение

В отличие от теоремы Байеса в форме отношений, эта глава не является непосредственно байесовской. Но весь байесовский анализ – это анализ распределений. Поэтому важно хорошо понимать концепцию распределений. С вычислительной точки зрения, распределением является любая структура данных, которая представляет множество величин (возможные выходы случайных процессов) и их вероятности.

Мы видели два представления распределений Pmf и Cdf. Эти представления эквивалентны в том смысле, что они содержат одинаковую информацию, и поэтому могут быть конвертированы одно в другое. Главная разница между ними заключается в исполнении: некоторые операции быстрее или проще с Pmf, другие быстрее с Cdf.

Другой целью этой главы было введение операций над распределениями, таких как `Pmf.__add__`, `Cdf.Max` и `thinkbayes.MakeMixture`. Мы будем использовать эти операции позже, но вводятся они сейчас, чтобы побудить вас думать о распределениях как о фундаментальном объекте вычислений, а не только как о контейнере для величин и вероятностей.

Глава 6

Анализ решений

Задача «Справедливой цены»

1 сентября 2007 года конкурсанты по имени Летия и Натаниель появились в телевизионном шоу «Справедливая цена» (The Price is Right)[1]. Они участвовали в игре под названием «Витрина» (Showcase), в которой надо было догадаться о цене витрины призов. Конкурсант, который называл цену, самую близкую к действительной цене вещей на витрине, не превышая ее, выигрывал призы.

Натаниель начинал первым. Его витрина состояла из посудомоечной машины, винного шкафчика, ноутбука и автомобиля. Он назвал цену 26 тысяч долларов.

Витрина Летии состояла из машины для игры в пинбол, видеоаркадной игры, бильярдного стола и круиза на Багамах. Она оценила витрину в 21 500 долларов.

Действительная цена витрины Натаниеля составляла 25 347 долларов. Его цена превышала действительную цену, и он проиграл.

Действительная цена витрины Летии составляла 21 578 долларов. Она назвала цену ниже действительной только на 78 долларов и выиграла свою витрину, а поскольку ее цена была ниже действительной цены на сумму, меньшую 250 долларов, то она выиграла также и витрину Натаниеля.

Для ценителей Байеса этот сценарий предлагает несколько вопросов:

1. Перед тем как увидеть призы, какой приор доверия относительно цены витрины следует назначить участнику игры?
2. Как участнику игры следует обновить свое доверие после того, как он увидел витрину?
3. Основываясь на апостериорном распределении, какую цену витрины должен предложить участник игры?

Третий вопрос демонстрирует обычное применение байесовского анализа: анализ решения.

---

[1] Росийское ТВ показывало аналог этого шоу под названием «Цена удачи».

Полученное апостериорное распределение мы можем выбрать в качестве ставки, которая максимизирует ожидаемый ответ участника.

Эта задача взята из книги Камерона Дэвидсон-Пилона (Cameron Davidson-Pilon) «Байесовские методы для хакеров» (*Bayesian Methods for Hackers*), в которой она была примером. Написанный для этой главы код доступен для загрузки из http://thinkbayes.com/price.py. Он содержит файлы, которые вы можете загрузить из http://thinkbayes.com/showcases.2011.csv и http://thinkbayes.com/showcases.2012.csv. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Приор

Для выбора приора распределения цен мы можем взять данные из предыдущих эпизодов шоу. К счастью, любители этого шоу сохранили детальный отчет. Когда я связался с Камероном Дэвидсон-Пилоном относительно его книги, он прислал мне данные, собранные на сайте http://tpirsummaries.8m.com Стивом Ги (Steve Gee). В эти данные входят цены каждой витрины с 2011 и 2012 сезонов игр и ставки, предложенные участниками этих игр.

На рис. 6.1 показаны распределения цен для этих витрин. Самая обычная цена для обеих витрин игры – 28 тысяч долларов. Но существовал и другой режим, при котором первая витрина имела цену около 50 тысяч долларов, а вторая – иногда более чем 70 тысяч долларов.

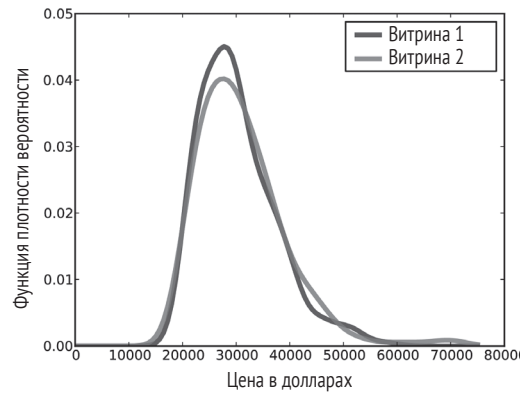


Рис. 6.1 ❖ Распределение цен на витрины шоу «Справедливая цена» в 2011–2012 годах

Распределения основаны на действительных данных, но сглажены гауссовым ядром оценки плотности (KDE – Kernel Density Estimation). Прежде чем мы продолжим, следует сделать небольшое отступление, чтобы поговорить о функции плотности вероятности и KDE.

Функция плотности вероятности

До настоящего времени мы работали с функцией вероятностной массы, или, иначе, функцией вероятностной меры (PMF). PMF отображает величину каждой случайной величины. В моей интерпретации Pmf-объект предусматривает метод, названный `Prob`, воспринимающий некоторую величину и возвращающий вероятность, известную также под названием **вероятностная масса**.

В математической интерпретации PDF обычно записывается как функция. Например, для функции распределения Гаусса (нормальное распределение) со средним 0 и стандартным отклонением 1 PDF можно записать так:

$$f(x) = \frac{1}{\sqrt{2\pi}} \exp(-x^2/2).$$

Для данной величины x эта функция определяет плотность вероятности. Плотность подобна вероятностной массе. То есть чем выше плотность вероятности некоторой величины, тем более вероятна сама эта величина.

Но плотность – это не вероятность. Плотность, в отличие от вероятности, ограниченной от 0 до 1, может быть равна нулю или любой положительной величине и не имеет границ.

Если вы интегрируете плотность в непрерывном интервале, то результатом будет вероятность. Но в этой книге это мы будем делать редко.

Вместо этого главным образом мы используем плотность как часть функции правдоподобия. Вскоре мы рассмотрим пример.

Представление PDF

Чтобы представить PDF в Phyton, в `thinkbayes.py` предусмотрен класс, названный `Pdf`. `Pdf` является **абстрактным типом класса**. То есть он определяет интерфейс, который предположительно имеет Pdf, но не обеспечивает завершенного применения. `Pdf`-интерфейс включает два метода – `Density` и `MakePmf`:

```
class Pdf(object):

    def Density(self, x):
        raise UnimplementedMethodException()

    def MakePmf(self, xs):
        pmf = Pmf()
        for x in xs:
            pmf.Set(x, self.Density(x))
        pmf.Normalize()
        return pmf
```

`Density` принимает величину `x` и возвращает соответствующую плотность. `MakePmf` создает дискретную аппроксимацию для PDF.

Pdf предусматривает применение MakePmf, но не Density, который должен быть обеспечен посредством child class.

Конкретный тип – это child class (подкласс), расширяющий абстрактный тип и обеспечивающий применение пропущенных методов. Например, GaussianPdf расширяет Pdf и обеспечивает Density:

```
class GaussianPdf(Pdf):

    def __init__(self, mu, sigma):
        self.mu = mu
        self.sigma = sigma

    def Density(self, x):
        return scipy.stats.norm.pdf(x, self.mu, self.sigma)
```

__init__ принимает mu и sigma, которые являются средним и стандартным отклонением распределения, и запоминает их как атрибуты.

Density, чтобы оценить гауссовское PDF, использует функцию из scipy.stats. Эту функцию называют norm.pdf, поскольку распределение Гаусса также имеет название «нормальное» распределение.

Гауссовское PDF описывается простой математической функцией и поэтому легко оценивается. И это полезно, потому что многое в реальном мире может быть аппроксимировано распределением Гаусса.

Но нет гарантии, что реальные данные имеют распределение Гаусса или могут быть описаны какой-либо другой простой математической функцией. В этом случае, чтобы оценить PDF всей совокупности, мы можем использовать частичную выборку.

Например, в шоу «Справедливая цена» у нас есть 313 цен для первой витрины. Мы можем использовать эти величины как выборку из совокупности всех вероятных цен витрин.

Скажем, выборка включает следующие величины (по порядку):

28 800, 28 868, 28 941, 28 957, 28 958.

В этой выборке отсутствуют величины между 28 801 и 28 867. Но из данной выборки нельзя сделать вывод о том, что величины между 28 801 и 28 867 невозможны. Основываясь на предварительной информации, мы принимаем, что все величины в этом интервале одинаково вероятны. Иными словами, мы ожидаем PDF довольно гладким.

Ядро оценки плотности (KDE) является алгоритмом, который принимает выборку и находит приемлемо гладкий PDF, который соответствует данным. С деталями вы можете познакомиться на сайте http://en.wikipedia.org/wiki/Kernel_density_estimation.

scipy обеспечивает применение KDE, а thinkbayes обеспечивает класс, называемый EstimatedPdf, который его использует:

```
class EstimatedPdf(Pdf):

    def __init__(self, sample):
```

```
        self.kde = scipy.stats.gaussian_kde(sample)

    def Density(self, x):
        return self.kde.evaluate(x)
```

__init__ принимает выборку и вычисляет ядро оценки плотности. Результатом является объект sian_kde, который обеспечивает метод evaluate.

Density принимает величину, вызывает gaussian_kde.evaluate и возвращает результирующую плотность.

Здесь произведен набросок кода, который был использован для генерации рис. 6.1.

```
prices = ReadData()
pdf = thinkbayes.EstimatedPdf(prices)

low, high = 0, 75000
n = 101
xs = numpy.linspace(low, high, n)
pmf = pdf.MakePmf(xs)
```

pdf – это Pdf-объект, который был оценен KDE. Pmf – это Pmf-объект, который аппроксимирует Pdf посредством оценки плотности последовательности одинаково распределенных в пространстве величин.

Linspace понимается как «линейное пространство». Он устанавливает диапазон low и high, а также число точек n и возвращает новое множество numpy с количеством n элементов, одинаково распределенных в пространстве между low и high, включая и границы.

Теперь вернемся к шоу «Справедливая цена».

Моделирование участников

Pdf на рис. 6.1 оценивают распределение возможных цен. Если бы вы были участником игры на шоу, то могли бы использовать это распределение, чтобы количественно определить ваш приор доверия о цене каждой витрины (до того, как вы увидели призы).

Чтобы обновить приоры, нам следует ответить на следующие вопросы:

1. Какие данные следует рассматривать и как их количественно определить?
2. Можем ли мы вычислить функцию правдоподобия? Другими словами, можем ли мы вычислить условное правдоподобие полученных данных для каждой гипотетической величины цены (price)?

Чтобы ответить на эти вопросы, смоделируем участников как инструмент, который догадывается о ценах с известной ошибкой. Другими словами, когда участник видит призы, то догадывается о цене каждого предмета. При этом каждый предмет оценивается отдельно, без учета, что приз является частью витрины. После все полученные таким образом цены суммируются. Давайте назовем это тотальной догадкой (guess).

При такой модели вопрос, на который мы должны ответить: «Если действительная цена есть price, каково правдоподобие, оцениваемое участником, которое было бы guess?»

Или, если мы определим error как

```
error = price - guess,
```

то мы могли бы спросить: «Каково правдоподобие, которое участник оценивает как безошибочное?»

Для ответа на этот вопрос мы снова можем использовать архивные данные. На рис. 6.2 показано кумулятивное распределение diff – разница между ставкой участника и действительной ценой витрины.

Определение diff таково:

```
diff = price - bid
```

Если diff имеет отрицательное значение, значит, ставка слишком высокая. Попутно мы можем использовать это распределение, чтобы вычислить вероятность того, что ставка участника больше действительной цены: первый участник превышал ставку в 25% случаев, а второй – в 29% случаев.

Мы также видим, что ставки смещены, то есть они более вероятно бывают более низкими, чем более высокими, что имеет смысл при данных правилах игры.

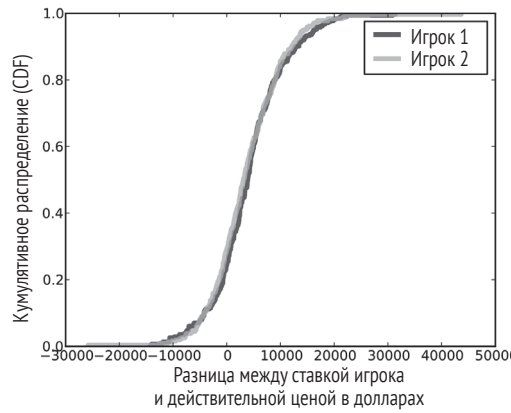


Рис. 6.2 ❖ Кумулятивное распределение разницы между ставками игроков и действительной ценой

Наконец, мы можем использовать это распределение, чтобы оценить надежность догадок участников.

Этот шаг не совсем надежен, так как мы вообще-то не знаем самих догадок участников. Нам известны только их ставки.

Поэтому нам следует сделать некоторые предположения. Предположительно распределение ошибки (error) гауссовское со средним 0 и той же дисперсией diff применяет такую модель описания:

```
class Player(object):

    def __init__(self, prices, bids, diffs):
        self.pdf_price = thinkbayes.EstimatedPdf(prices)
        self.cdf_diff = thinkbayes.MakeCdfFromList(diffs)

        mu = 0
        sigma = numpy.std(diffs)
        self.pdf_error = thinkbayes.GaussianPdf(mu, sigma)
```

`prices` – это последовательность витринных призов, `bids` – последовательность ставок и `diffs` – последовательность diffs, где по-прежнему `diff = price - bid`.

`pdf_price` является сглаженным PDF, оцениваемым KDE. `cdf_diff` является кумулятивным распределением `diff`, которое мы видим на рис. 6.2. `pdf_error` – это PDF, которое характеризует распределение ошибок, где `error = price - guess`.

Мы опять используем дисперсию `diff` для оценки дисперсии `error`. Эта оценка не идеальна, поскольку ставки участников иногда носят стратегический характер. Например, если Игрок 2 полагает, что Игрок 1 сделал очень высокую ставку, то Игрок 2 может сделать очень низкую ставку. В этом случае `diff` не отражает `error`. Если это случается часто, то наблюдаемая дисперсия в `diff` может превышать дисперсию в error. Тем не менее это, скорее всего, разумная модель решения.

В качестве альтернативы некто, готовящийся появиться на шоу, мог бы оценить свое собственное распределение ошибок, наблюдая предшествующие шоу и записывая догадки на них и действительные цены.

Правдоподобие

Теперь мы готовы написать функцию правдоподобия. Как обычно, определяется новый класс, который расширяет `thinkbayes.Suite`.

```
class Price(thinkbayes.Suite):

    def __init__(self, pmf, player):
        thinkbayes.Suite.__init__(self, pmf)
        self.player = player
```

`pmf` представляет распределение приора, а `player` – это Player-объект, описанный в предыдущем разделе. Опишем `Likelihood`:

```
    def Likelihood(self, data, hypo):
        price = hypo
        guess = data

        error = price - guess
        like = self.player.ErrorDensity(error)

        return like
```

hypo – это гипотетическая цена витрины, data – наилучшая догадка игрока относительно цены. error – разница, like – правдоподобие данных для этой гипотезы.

ErrorDensity определен в Player:

```
# class Player:

    def ErrorDensity(self, error):
        return self.pdf_error.Density(error)
```

ErrorDensity работает благодаря оценке pdf_error полученных величин ошибок. Результатом является плотность вероятности, которая реально вероятностью не является. Однако следует помнить, что в Likelihood нет необходимости в вычислении вероятности – правдоподобие должно только вычислить нечто, пропорциональное вероятности. До тех пор, пока постоянная пропорциональности одинакова для всех правдоподобий, она не используется при нормализации апостериорного распределения.

Обновление

Player обеспечивает метод, который берет догадку участника и вычисляет апостериорную вероятность:

```
# class Player

    def MakeBeliefs(self, guess):
        pmf = self.PmfPrice()
        self.prior = Price(pmf, self)
        self.posterior = self.prior.Copy()
        self.posterior.Update(guess)
```

PmfPrice генерирует дискретную аппроксимацию о ценах для PDF, используемого для создания приора.

PmfPrice использует Makepmf, который оценивает pmf_ price на последовательности величин:

```
# class Player

    n = 101
    price_xs = numpy.linspace(0, 75000, n)

    def PmfPrice(self):
        return self.pdf_price.MakePmf(self.price_xs)
```

Чтобы создать постериор, мы берем копию приора и затем активируем Update, который активирует Likelihood для каждой из гипотез. Далее умножает приор на правдоподобие и снова нормализует.

Вернемся к оригинальному сценарию. Предположим, что вы Игрок 1. Когда вы видите витрину, ваша наилучшая догадка – что общая цена призов равна 20 тысяч долларов.

На рис. 6.3 показаны приор и постериор доверий о действительной цене. Постериор сдвинут влево, поскольку ваша догадка находится на нижней границе интервала приора.

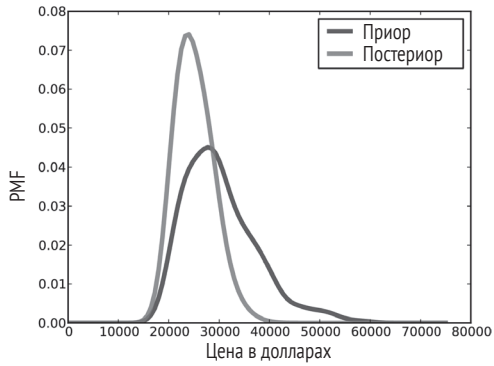


Рис. 6.3 ❖ Априорное и апостериорное распределение для Игрока 1, основанное на наилучшей догадке – 20 000 долларов

С одной стороны, результат имеет смысл. Наиболее вероятная величина в приоре 27 750 долларов, ваша лучшая догадка 20 000 долларов, а средний постериор находится где-то посередине: 25 096 долларов.

С другой стороны, это может показаться странным, потому что предполагает следующее: если вы *думаете*, что цена 20 000 долларов, то вам следует *верить,* что цена равна 24 000 долларов.

Чтобы разрешить этот очевидный парадокс, вспомните: вы объединяете два источника информации – архивные данные о прошлых витринах и догадки о призах, которые вы видите.

Мы рассматривали архивные данные в качестве приора и, основываясь на наших догадках, обновляли их. Но мы можем эквивалентно использовать ваши догадки как приор и обновлять их, основываясь на архивных данных.

Если вы будете размышлять над этим подобным образом, возможно, для вас будет менее удивительным, что наиболее вероятная величина в постериоре – вовсе не ваша оригинальная догадка.

Оптимальное предложение цены

Теперь, когда мы имеем апостериорное распределение, мы можем использовать его для вычисления оптимальной ставки, которую я определяю как максимально ожидаемый возврат (см. http://en.wikipedia.org/wiki/Expected_return).

В этом разделе я собираюсь представить методы по нисходящей. То есть вы увидите, как они используются, перед тем как они работают. Не беспокойтесь, если вы увидите незнакомый метод. Его определение появится скоро.

Чтобы вычислить оптимальную ставку, был введен класс под названием GainCalculator:

```
class GainCalculator(object):

    def __init__(self, player, opponent):
        self.player = player
        self.opponent = opponent
```

player и opponent есть Player-объекты.

GainCalculator предусматривает ExpectedGains, который вычисляет последовательность ставок и ожидаемый результат для каждой ставки:

```
    def ExpectedGains(self, low=0, high=75000, n=101):
        bids = numpy.linspace(low, high, n)

        gains = [self.ExpectedGain(bid) for bid in bids]

        return bids, gains
```

low и high являются диапазоном возможных ставок; n – число предлагаемых ставок.

ExpectedGains вызывает ExpectedGain, вычисляющий результат для данной ставки:

```
    def ExpectedGain(self, bid):
        suite = self.player.posterior
        total = 0
        for price, prob in sorted(suite.Items()):
            gain = self.Gain(bid, price)
            total += prob * gain
        return total
```

ExpectedGain перебирает значения в постериоре и с учетом действительных цен на витрину вычисляет результат для каждой ставки. Он взвешивает каждый результат с соответствующей вероятностью и возвращает итог.

ExpectedGain вызывает Gain, который берет ставку и действительную цену и возвращает ожидаемый результат:

```
    def Gain(self, bid, price):
        if bid > price:
            return 0

        diff = price - bid
        prob = self.ProbWin(diff)

        if diff <= 250:
            return 2 * price * prob
        else:
            return price * prob
```

Если ставка завышена, вы ничего не получаете. В противном случае вы вычисляете разницу между вашей ставкой и ценой. Эта разница и определяет вероятность выигрыша.

Если разница (diff) меньше 250 долларов, вы выигрываете обе витрины. Для простоты можно предположить, что обе витрины стоят одинаково. Поскольку этот вариант случается редко, большой разницы нет.

Наконец, мы должны вычислить вероятность выигрыша, основываясь на разнице diff.

```
    def ProbWin(self, diff):
        prob = (self.opponent.ProbOverbid() +
                self.opponent.ProbWorseThan(diff))
        return prob
```

Если ваш оппонент завысил ставку, вы выиграли. В противном случае вы должны надеяться, что оппонент не угадал более, чем diff. Player обеспечивает методы, вычисляющие обе вероятности:

```
# class Player:

    def ProbOverbid(self):
        return self.cdf_diff.Prob(-1)

    def ProbWorseThan(self, diff):
        return 1 - self.cdf_diff.Prob(diff)
```

Этот код может смущать, так как вычисления производятся с точки зрения оппонента, высчитывающего: «Какова вероятность, что я сделаю завышенную ставку?» и «Какова вероятность, что моя ставка будет выше, чем diff?».

Оба ответа на вопросы основываются на CDF разницы (diff). Если diff оппонента хуже, чем ваш, то вы выиграли. В противном случае вы проиграли.

Наконец, код, который вычисляет оптимальные ставки:

```
# class Player:

    def OptimalBid(self, guess, opponent):
        self.MakeBeliefs(guess)
        calc = GainCalculator(self, opponent)
        bids, gains = calc.ExpectedGains()
        gain, bid = max(zip(gains, bids))
        return bid, gain
```

Для данной догадки и оппонента OptimalBid вычисляет апостерионое распределение, реализует GainCalculator, вычисляет ожидаемый выигрыш для диапазона ставок и возвращает оптимальную ставку и ожидаемый выигрыш.

На рис. 6.4 показан результат для обоих игроков, основанный на сценарии, в котором лучшая догадка Игрока 1 составляет 20 000 долларов, Игрока 2 – 40 000 долларов. Для Игрока 1 оптимальной ставкой является 21 000 долларов, создающая ожидаемый возврат почти 16 700 долларов. Это случай (что оказывается необычным), в котором оптимальная ставка выше, чем лучшая догадка игрока.

Для Игрока 2 оптимальная ставка составляет 31 500 долларов, которая создает ожидаемый выигрыш 19 400 долларов. Это более типичный случай, когда оптимальная ставка меньше, чем лучшая догадка.

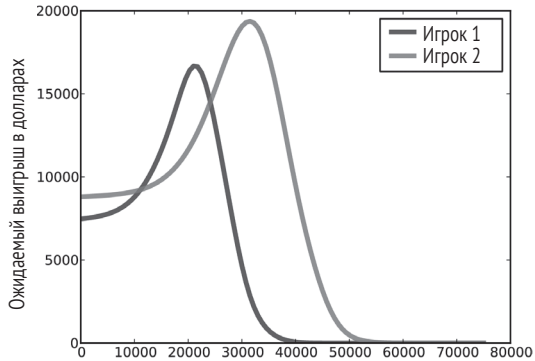


Рис. 6.4 ❖ Ожидаемый выигрыш при ставках в сценарии, когда лучшая догадка Игрока 1 составляет 20 000 долларов, а Игрока 2 – 40 000 долларов

Обсуждение

Одной из характерных черт байесовской оценки является то, что мы получаем результат в форме апостериорного распределения. Классическое оценивание обычно создает точечную оценку или интервал доверия. Это существенно, когда это последний шаг в обработке. Но если вы хотите использовать оценку как исходную для последующего анализа, точечные и интервальные оценки не слишком в этом помогают.

В данном примере мы использовали апостериорное распределение, чтобы вычислить оптимальную ставку. Возврат для данной ставки асимметричный и не непрерывный (если вы дали завышенную оценку, то вы проиграли), и поэтому было бы затруднительно решить эту задачу аналитически. Но ее относительно просто решить вычислением на компьютере.

Новички в байесовском мышлении часто имеют тенденцию суммировать апостериорное распределение, вычисляя среднее или оценку максимального правдоподобия. Эти суммирования могут быть полезными. Но если это все, что вам необходимо, то вам байесовские методы, возможно, и не нужны.

Байесовские методы наиболее полезны, когда вы можете перенести апостериорное распределение на следующую ступень анализа для выполнения анализа решений. Это мы сделали в данной главе. Можно сделать предсказание. Но этим мы займемся в следующей главе.

Глава 7

Предсказание

Задача о Бостон Брюинс

В финальной игре сезона 2010–2011 годов Национальной хоккейной лиги (НХЛ) команда Бостон Брюинс сыграла свои лучшие серии из семи чемпионатов против Ванкувер Кэнакс. Бостон проиграл первые две игры 0:1 и 2:3, затем выиграл следующие две игры 8:1 и 4:0. Какова вероятность на этом этапе серии игр, что Бостон выиграет следующую игру, и какова вероятность, что он выиграет чемпионат?

Как всегда, для ответа на подобный вопрос нам необходимо сделать некоторые предположения. Во-первых, разумно предположить, что процесс забивания голов в хоккее, по крайней мере в аппроксимации, – пуассоновский процесс. Это означает, что голы в хоккее могут забиваться равновероятно в любое время в течение игры. Во-вторых, мы можем предположить, что каждая команда имеет статистику о среднем количестве голов за игру, забитых конкретному противнику во многих играх. Обозначим это среднее как λ.

При этих предположениях стратегия для ответа на этот вопрос следующая:

1. Использовать статистику о предыдущих играх, чтобы выбрать априорное распределение для λ.
2. Использовать счет первых четырех игр финала, чтобы оценить λ для каждой из команд.
3. Использовать апостериорное распределение λ, чтобы вычислить распределение голов для каждой команды, дифференциальное распределение голов и вероятность выиграть следующую игру для каждой команды.
4. Вычислить, какова вероятность выиграть серию для каждой команды.

Чтобы выбрать априорное распределение, из http://www.nhl.com были взяты некоторые данные о среднем количестве голов каждой команды в сезоне игр 2010–2012 годов. Распределение – примерно гауссовское со средним 2.8 и стандартным отклонением 0.3.

Распределение Гаусса – непрерывное. Но мы аппроксимируем его с помощью дискретного Pmf.

`thinkbayes` обеспечивает выполнение `MakeGaussianPmf` следующего:

```
def MakeGaussianPmf(mu, sigma, num_sigmas, n=101):
    pmf = Pmf()
    low = mu - num_sigmas*sigma
    high = mu + num_sigmas*sigma

    for x in numpy.linspace(low, high, n):
        p = scipy.stats.norm.pdf(mu, sigma, x)
        pmf.Set(x, p)
    pmf.Normalize()
    return pmf
```

`mu` и `sigma` – среднее и стандартное отклонения распределения Гаусса.

`num_sigmas` – число стандартных отклонений вниз и вверх от среднего, которое Pmf будет учитывать, и `n` – число величин в Pmf.

Здесь мы снова используем `numpy.Linspace`, создающий последовательность из `n` одинаково распределенных величин между нижним и верхним значениями, включая и их.

`norm.pdf` оценивает функцию гауссовой плотности распределения вероятности (PDF).

Возвращаясь к хоккейной задаче, определяем пакет гипотез о величине λ.

```
class Hockey(thinkbayes.Suite):

    def __init__(self):
        pmf = thinkbayes.MakeGaussianPmf(2.7, 0.3, 4)
        thinkbayes.Suite.__init__(self, pmf)
```

Таким образом, априорное распределение является распределением Гаусса со средним значением 2.7, стандартной девиацией 0.3 и перекрывает 4 сигмы выше и ниже среднего.

Как обычно, нам следует решить, как представить каждую гипотезу. В данном случае гипотеза представлена как $x = \lambda$ для величины `x` с плавающей запятой.

Процесс Пуассона

В математической статистике **процесс** является стохастической моделью физической системы. Слово «стохастический» означает, что в модели присутствует некоторый элемент случайности. Например, процесс Бернулли является моделью последовательности событий, называемых испытаниями, где каждое испытание имеет два возможных исхода, таких как успех и неудача. Поэтому процесс Бернулли является естественной моделью серии подбрасываний монеты или серией выстрелов в цель.

Процесс Пуассона является непрерывной версией процесса Бернулли, когда события могут происходить в любой момент времени с одинаковой вероятностью. Процесс Пуассона может использоваться как модель покупателей, пришедших в магазин, автобусов, прибывающих на автобусную остановку, или забиваемых в хоккее голов.

Во многих реальных системах вероятность какого-либо события меняется со временем. Более вероятно, что посетители приходят в магазин в определенное время дня. Автобусы, скорее всего, приходят на остановку с заданным интервалом. А голы забиваются в более или менее различное время.

Но все процессы основываются на упрощениях. И в нашем случае моделирование хоккейной игры, основывающееся на процессе Пуассона, – это разумный выбор. Хейер (Heuer), Мюллер (Muller) и Рабнер (Rubner) (2010), анализируя результаты игр в футбол в Германии, приходят к такому же заключению (см. http://www.cimat.mx/Eventos/vpec10/img/poisson.pdf).

Преимущество использования этой модели заключается в том, что мы можем эффективно вычислить распределение количества голов за игру так же, как и распределение времени между голами. Более конкретно, если обозначить среднее число голов в игре через `lam`, распределение голов за игру можно представить следующим пуассоновским PMF:

```
def EvalPoissonPmf(lam, k):
    return (lam)**k * math.exp(-lam) / math.factorial(k)
```

И распределение времени между голами будет определяться экспоненциальным PDF:

```
def EvalExponentialPdf(lam, x):
    return lam * math.exp(-lam * x)
```

Здесь использована переменная `lam`, потому что `lambda` является зарезервированным ключевым словом в Phyton. Обе эти функции присутствуют в `thinkbayes.py`.

Постериоры

Теперь мы можем вычислить правдоподобие, которое команда с гипотетической величиной `lam` забивает `k` голов за игру:

```
# class Hockey

    def Likelihood(self, data, hypo):
        lam = hypo
        k = data
        like = thinkbayes.EvalPoissonPmf(lam, k)
        return like
```

Каждая гипотеза – это возможное значение λ; `data` – наблюдаемое число забитых голов `k`.

Вместо функции правдоподобия мы можем создать пакет для каждой команды и обновить их в соответствии с забитыми голами в первых четырех играх.

```
    suite1 = Hockey('bruins')
    suite1.UpdateSet([0, 2, 8, 4])
```

```
suite2 = Hockey('canucks')
suite2.UpdateSet([1, 3, 1, 0])
```

На рис. 7.1 показаны результирующие апостериорные распределения для `lam`. Основанные на первых четырех играх, наиболее вероятные значения для `lam` составляют для Кэнакс 2.6 и для Брюинс 2.9.

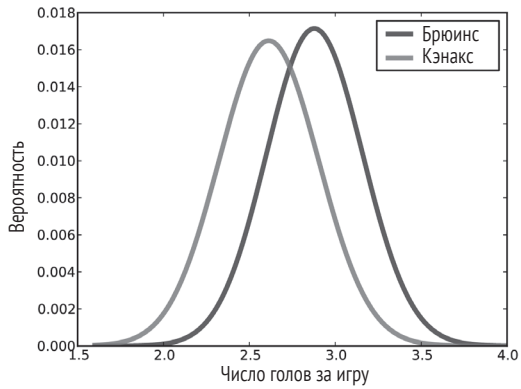


Рис. 7.1 ❖ Апостериорное распределение числа голов за игру

Распределение голов

Чтобы вычислить вероятность выигрыша каждой командой следующей игры, необходимо вычислить распределение голов для каждой команды.

Если бы мы знали величину `lam` точно, то могли бы снова использовать распределение Пуассона.

`thinkbayes.py` обеспечивает метод, вычисляющий усеченную аппроксимацию распределения Пуассона:

```
def MakePoissonPmf(lam, high):
    pmf = Pmf()
    for k in xrange(0, high+1):
        p = EvalPoissonPmf(lam, k)
        pmf.Set(k, p)
    pmf.Normalize()
    return pmf
```

Диапазон величин в Pmf составляет 0 – `high`. Поэтому если величина `lam` была точно 3.4, то мы вычислим:

```
lam = 3.4
goal_dist = thinkbayes.MakePoissonPmf(lam, 10)
```

Верхняя граница 10 была выбрана в связи с малой вероятностью, что в игре будет забито более 10 голов.

До сих пор все было просто. Но ведь точной величины `lam` мы же не знаем!

Вместо этого мы имеем только распределение вероятных значений для `lam`.

Для каждого значения `lam` распределение голов – это распределение Пуассона. Поэтому общее распределение голов является смесью распределений Пуассона, взвешенных в соответствии с вероятностями в распределении `lam`.

Учитывая апостериорное распределение `lam`, напишем код, создающий распределение голов:

```
def MakeGoalPmf(suite):
    metapmf = thinkbayes.Pmf()

    for lam, prob in suite.Items():
        pmf = thinkbayes.MakePoissonPmf(lam, 10)
        metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return mix
```

Для каждого значения `lam` мы создадим пуассоновское Pmf и добавим его к мета-Pmf. Название мета-Pmf было дано потому, что мета-Pmf содержит в качестве своих величин все Pmf.

Затем, чтобы вычислить смесь, используем `MakeMixture` (его мы видели на стр. 60).

На рис. 7.2 показано результирующее распределение для Брюинс и Кэнакс. Менее вероятно, что Брюинс забьет 3 гола или меньше в следующей игре. Более вероятно, что будет забито 4 гола или больше.

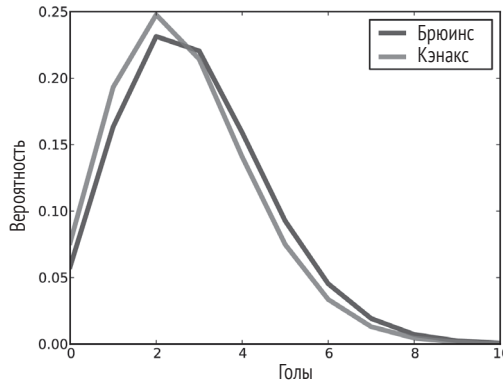


Рис. 7.2 ❖ Распределение голов в одной игре

Вероятность выигрыша

Чтобы получить вероятность выигрыша, мы вычислим сначала распределение разницы голов:

```
goal_dist1 = MakeGoalPmf(suite1)
goal_dist2 = MakeGoalPmf(suite2)
diff = goal_dist1 - goal_dist2
```

Оператор вычитания вызывает `Pmf.__sub__`, который пересчитывает пары величин и вычисляет разницу. Вычитание двух распределений почти то же, что сложение, которое мы видели в «Добавлениях» на стр. 55.

Если разница голов положительная, то Брюинс выигрывает. Если разница отрицательная, то выигрывает Кэнакс. Если разница 0, то ничья:

```
p_win = diff.ProbGreater(0)
p_loss = diff.ProbLess(0)
p_tie = diff.Prob(0)
```

С распределениями из предыдущего раздела `p_win` равен 46%, `p_loss` равен 37% и `p_tie` равен 17%.

Выигрыш в дополнительное время

Чтобы вычислить вероятность победы в дополнительное время, важна статистика об отсутствии голов в основное время и о дальнейшей игре до первого гола («внезапной смерти»). Предположение о том, что забивание голов – это процесс Пуассона, подразумевает, что время между голами распределено экспоненциально.

Учитывая `lam`, мы можем вычислить время между голами следующим образом:

```
lam = 3.4
time_dist = thinkbayes.MakeExponentialPmf(lam, high=2, n=101)
```

`high` – верхняя граница распределения. В этом случае было выбрано 2, так как вероятность того, что на протяжении более чем 2 игр голы не будут забиты, невелика.

`n` – это количество величин в Pmf.

Если бы мы точно знали `lam`, то все было бы ясно. Но этой величины мы не знаем. Вместо этого мы имеем апостериорное распределение возможных величин. Поэтому, как это делалось с распределением голов, мы создадим мета-Pmf и вычислим смесь всех Pmf.

```
def MakeGoalTimePmf(suite):
    metapmf = thinkbayes.Pmf()

    for lam, prob in suite.Items():
        pmf = thinkbayes.MakeExponentialPmf(lam, high=2, n=2001)
        metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return mix
```

На рис. 7.3 показаны результирующие распределения. Для временных значений, меньших, чем один период (одна треть игры), более вероятно, что Брюинс забьет. Более вероятно, что время, необходимое для этого Кэнаксу, больше.

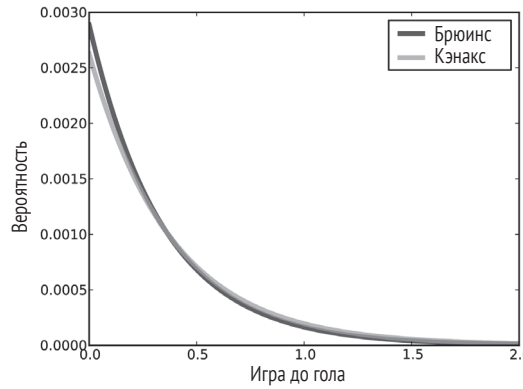


Рис. 7.3 ❖ Распределение времени между голами

Число значений `n` было принято довольно высоким, чтобы минимизировать число ничьих, так как обе команды не могут забивать голы одновременно.

Теперь мы вычислим вероятность, что Брюинс забьет первым:

```
time_dist1 = MakeGoalTimePmf(suite1)
time_dist2 = MakeGoalTimePmf(suite2)
p_overtime = thinkbayes.PmfProbLess(time_dist1, time_dist2)
```

Для Брюинс вероятность выиграть в дополнительное время равна 52%.

Наконец, общая вероятность победы есть шанс победы в конце регулярной игры плюс вероятность победы в дополнительное время:

```
p_tie = diff.Prob(0)
p_overtime = thinkbayes.PmfProbLess(time_dist1, time_dist2)

p_win = diff.ProbGreater(0) + p_tie * p_overtime
```

Для Брюинс общий шанс выиграть следующую игру равен 55%.

Чтобы выиграть серию, Брюинс может либо выиграть следующие две игры, либо проиграть следующие две игры, но выиграть третью. Снова мы можем подсчитать общую вероятность:

```
# win the next two
p_series = p_win**2

# split the next two, win the third
p_series += 2 * p_win * (1-p_win) * p_win
```

Шанс, что Брюинс выиграет серию, равен 57%. И в 2011 году они сделали это.

Обсуждение

Как обычно, анализ в этой главе основан на моделировании решений. А моделирование – почти всегда итеративный процесс. В общем, следует начать с чего-нибудь простого, что даст ответ на уровень аппроксимации. Далее идентифицировать вероятные источники ошибок и посмотреть возможность для улучшения.

В этом примере были рассмотрены эти возможности:

- был выбран приор, основанный на среднем количестве голов за игру. Но эта статистика средняя для всех команд. Относительно отдельной команды мы можем ожидать большей изменчивости. Например, если команда с лучшей игрой в нападении играет с командой, имеющей плохую защиту, то ожидаемое количество голов за игру может быть равно нескольким стандартным отклонениям относительно среднего;
- здесь использовались данные только первых четырех игр из серии. Если эти команды играли друг с другом на протяжении всего сезона, можно было бы использовать результаты и этих игр. Одной из трудностей является то, что состав команд изменяется во время всего сезона из-за нагрузки и травм. Поэтому лучше придавать больший вес недавним играм;
- чтобы использовать все преимущества информации, можно было бы использовать результаты игр всего регулярного сезона, чтобы оценить уровень забитых голов для каждой команды. Это, скорее всего, уточнило бы оценку дополнительного коэффициента парного соответствия команд. Такой метод был бы более сложным, но тем не менее осуществимым.

Чтобы дать оценку изменчивости из-за парного соответствия команд, в качестве первой возможности для улучшения можно было бы использовать результаты регулярного сезона. Благодаря Дирку Хоагу (Dirk Hoag) на странице Интернета http://forechecker.blogspot.com/ было найдено количество голов, забитых во время регулярных игр (но не в дополнительное время) в каждой игре регулярного сезона.

Команды различных конференций играли друг с другом один или два раза в регулярном сезоне. Поэтому особое внимание было уделено парам, которые играли друг с другом 4–6 раз. Для каждой пары было подсчитано среднее количество голов за игру, что является оценкой λ. Затем был построен график распределения этой оценки.

Среднее этих оценок равнялось снова 2.8. Но стандартное отклонение составило 0.85, что существенно выше, чем то, что мы получили, вычисляя одну оценку для каждой команды.

Если снова запустить анализ уже с большей дисперсией для приора, то вероятность выигрыша Брюинсом сессии составит 80%, то есть существенно выше, чем результат с приором, имеющим дисперсию 57%. Таким образом, оказыва-

ется, что результаты чувствительны к приору. Здесь можно подумать, с каким количеством данных мы должны работать. Основываясь на разнице между моделью с малой дисперсией и моделью с большой дисперсией, можно сделать вывод: целесообразно затратить усилия на поиск более правильного приора.

Код и данные этой главы доступны на странице Интернета http://thinkbayes.com/hockey.py и http://thinkbayes.com/hockey_data.csv. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Упражнения

Упражнение 7.1

Если автобус прибывает на автобусную остановку каждые 20 минут, а вы приходите на остановку в случайное время, время вашего ожидания до прибытия автобуса распределено на интервале 0–20 минут.

Но в реальности существуют колебания во времени прибытия автобусов. Предположим, вы ждете автобус и знаете архивное распределение времени между прибытиями автобусов. Вычислите распределение времени вашего ожидания.

Упражнение 7.2

Предположим, пассажиры приходят на остановку автобуса в соответствии с процессом Пуассона с параметром λ. Если вы приходите на остановку и находите там трех ожидающих автобуса пассажиров, каким будет апостериорное распределение вашего времени ожидания с момента прибытия предыдущего автобуса?

Версия этой задачи будет решена в следующей главе.

Упражнение 7.3

Предположим, вы эколог, собирающий популяцию насекомых в новом районе. Вами было расставлено 100 ловушек в проверяемой местности. На следующий день вы пришли проверить ловушки. Обнаружено, что 37 ловушек сработали и внутри них находятся пойманные насекомые. Если ловушка сработала, то она не может поймать еще одно насекомое, пока не будет переустановлена.

Если вы переустановили ловушки и вернулись через два дня, как много ловушек вы ожидаете найти сработавшими?

Упражнение 7.4

Предположим, что вы менеджер здания с апартаментами, где в местах общего пользования установлены 100 лампочек освещения. Вам необходимо менять перегоревшие лампочки.

Первого января все лампочки горели. Во время проверки лампочек первого февраля вы обнаружили, что нити накаливания трех лампочек перегорели. Если бы вы пришли первого апреля, сколько лампочек бы перегорело?

В этом упражнении можно предположить, что все события в любое время равновероятны. Правдоподобие выхода из строя лампочек освещения зависит от продолжительности их работы. Уровень отказа старых лампочек выше, так как нить накаливания при работе испаряется.

Эта проблема с более открытым конечным результатом, чем другие. Здесь вам придется принимать моделированные решения. Вы, возможно, захотите почитать о распределении Вейбулла (http://en.wikipedia.org/wiki/Weibull_distribution) или поискать информацию о надежности лампочек освещения.

Глава 8

Погрешность наблюдения

Задача о линии метрополитена

В штате Массачусетс линия метрополитена «Красная линия» соединяет Кембридж и Бостон. Когда автор работал в Кембридже, ездил по этой ветке метро от станции «Площадь Кендалла» до станции «Южная». Затем пересаживался на электричку и ехал до Нидхэма. В часы пик поезда этой линии метрополитена следуют в среднем с промежутком 7–8 минут.

Когда я приходил на станцию метро, я мог оценить время до прихода следующего поезда, основываясь на количестве пассажиров на платформе. Если их было мало, делался вывод, что поезд только что ушел, и предположение – поезд ждать придется около 7 минут. Если на платформе было больше пассажиров, то предполагалось, что поезд придет раньше. Но если на платформе было очень много пассажиров, я подозревал, что поезд не придет по расписанию. Поэтому я выходил на улицу и брал такси.

Во время ожидания поезда я размышлял о том, как байесовское оценивание может помочь предсказать время ожидания поезда. После этого предсказания мне нужно было принимать решение: ждать поезд дальше или взять такси. В этой главе представлен проведенный мною анализ.

Эта глава основана на проекте Брендана Риттера (Brendan Ritter) и Кая Остина (Kai Austin), которые вели со мной класс в колледже Олин. Код в этой главе доступен из http://thinkbayes.com/redline.py. Код, который я использовал для сбора данных, находится в http://thinkbayes.com/redline_data.py.

Модель

Прежде чем приступить к анализу, мы должны принять некоторые решения о моделировании. Во-первых, прибытие пассажиров на платформу мы рассмотрим как процесс Пуассона. Это означает следующее предположение: пассажиры с одинаковой вероятностью прибывают на платформу в любое время, и они прибывают с неизвестной периодичностью (скоростью) λ, измеряемой количеством пассажиров в минуту. Поскольку я наблюдаю пассажиров в тече-

ние короткого периода времени и каждый день в одно и то же время, я полагаю λ константой.

С другой стороны, процесс прибытия поездов – это не процесс Пуассона. Поезда до Бостона предположительно отправляются с конечной станции «Алевайв» каждые 7–8 минут во время часа пик, но ко времени, когда они прибывают на станцию «Площадь Кендалла», время между прибытием поездов лежит в интервале от 3 до 12 минут.

Собирая данные о времени прибытия поездов, я написал скрипт, который загружает данные в реальном времени из http://www.mbta.com/rider_tools/developers/, отбирает поезда, идущие в южном направлении и прибывающие на станцию «Площадь Кендалла», и записывает время их прибытия в базу данных. Я запускаю скрипт 5 дней в неделю с 4 до 5 часов вечера и записываю 15 прибытий поездов в день. Затем я подсчитываю время между последовательными прибытиями. Распределение этих промежутков времени показано на рис. 8.1 и помечено буквой z.

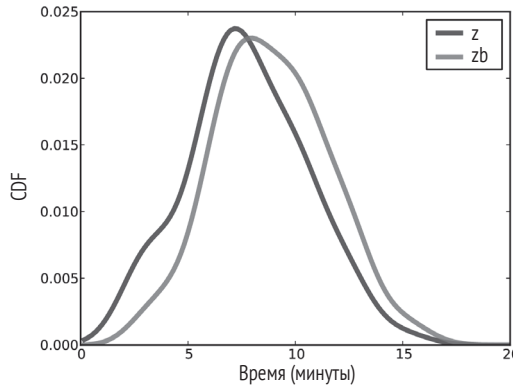


Рис. 8.1 ❖ PMF промежутков времени между поездами, основанные на накоплении данных и сглаженные KDE. z – истинное распределение, zb – смещенное распределение, увиденное случайными пассажиром

Если вы стоите на платформе с 4 до 6 часов вечера и записываете интервалы между поездами, то это и будет полученное вами распределение. Но если вы приходите в какое-то случайное время (не по расписанию поездов), то получите другое распределение. Среднее время между поездами, которое получил бы такой случайный пассажир, было бы выше, чем истинное, действительное среднее время.

Почему? Потому что пассажиры прибывают более вероятно в течение большего интервала времени, чем меньшего.

Рассмотрим простой пример: предположим, что время между поездами или 5 минут, или 10 минут с равной вероятностью. В этом случае среднее время между поездами составляет 7,5 минуты.

Но пассажиры с большей вероятностью приходят в течение 10-минутного промежутка времени, чем в течение 5-минутного, что фактически в два раза более вероятно. Если бы мы наблюдали за прибывающими пассажирами, то нашли бы следующую зависимость: 2/3 из них прибывают в течение 10-минутного промежутка времени, и только 1/3 – в течение 5-минутного промежутка времени, а среднее время между поездами, с точки зрения прибывающих пассажиров, составляет 8,33 минуты.

Такого рода **погрешность наблюдения**, или **смещение наблюдения**, появляется во многих ситуациях. Студенты думают, что классы больше, чем они есть на самом деле, потому что большинство из них бывает в больших классах. Пассажиры авиалайнеров думают, что пассажиров в самолете больше, чем это есть на самом деле. А думают так, потому что большинство из них летало на заполненных пассажирами самолетах.

В каждом случае величины из истинного распределения избыточно дискретизированы пропорционально их величине. В примере с веткой метрополитена «Красная линия» в два раза больший промежуток времени между поездами в два раза более вероятен при наблюдении.

Учитывая действительное распределение промежутков времени, мы можем вычислить распределение этих промежутков, наблюдаемое пассажирами.

`BiasPmf` выполняет эти вычисления:

```
def BiasPmf(pmf):
    new_pmf = pmf.Copy()

    for x, p in pmf.Items():
        new_pmf.Mult(x, x)

    new_pmf.Normalize()
    return new_pmf
```

`pmf` – действительное `y` распределение; `new_pmf` – смещенное распределение.

Внутри цикла мы умножаем вероятность каждой величины `x` на наблюдаемое правдоподобие, которое пропорционально `x`. Затем мы нормализуем результат.

На рис. 8.1 показаны действительное распределение промежутков, обозначенное `z`, и смещенное распределение промежутков, наблюдаемое пассажирами, обозначенное `zb`.

Время ожидания

Время ожидания, обозначенное `y`, – это время между прибытием пассажира и последующим прибытием поезда. Прошедшее время, обозначенное `x`, – это время между прибытием предыдущего поезда и прибытием пассажира. Здесь эти определения были выбраны так, что `zb = x + y`.

Имея распределение `zb`, мы можем вычислить распределение `y`. Я начну с простого случая и затем обобщу его. Предположим, как и в предыдущем при-

мере, что zb равняется либо 5 минут с вероятностью 1/3, либо 10 минут с вероятностью 2/3.

Если мы прибыли в случайное время в интервале 5 минут, то y постоянен в течение времени от 0 до 5 минут. Если мы прибыли в случайное время в интервале 10 минут, то y постоянен в интервале от 0 до 10 минут. Поэтому совокупное распределение – это смесь равномерных распределений, взвешенных в соответствии с вероятностью каждого интервала времени.

Следующая функция принимает распределение zb и вычисляет распределение y:

```
def PmfOfWaitTime(pmf_zb):
    metapmf = thinkbayes.Pmf()
    for gap, prob in pmf_zb.Items():
        uniform = MakeUniformPmf(0, gap)
        metapmf.Set(uniform, prob)

    pmf_y = thinkbayes.MakeMixture(metapmf)
    return pmf_y
```

PmfOfWaitTime создает мета-Pmf, которое отображает каждое равномерное распределение в его вероятность. Затем используется MakeMixture, которое было показано в разделе «Перемешивание» на стр. 60, чтобы вычислить смесь.

PmfOfWaitTime также использует MakeUniformPmf, определяемое как:

```
def MakeUniformPmf(low, high):
    pmf = thinkbayes.Pmf()
    for x in MakeRange(low=low, high=high):
        pmf.Set(x, 1)
    pmf.Normalize()
    return pmf
```

low и high – интервал равномерного распределения (включая обе границы). Наконец, MakeUniformPmf использует MakeRange, определяемое как:

```
def MakeRange(low, high, skip=10):
    return range(low, high+skip, skip)
```

MakeRange определяет множество возможных значений для времени ожидания (выраженного в секундах). По умолчанию общий интервал делится на 10-секундные интервалы.

Чтобы инкапсулировать процесс вычисления этих распределений, был создан класс, называемый WaitTimeCalculator:

```
class WaitTimeCalculator(object):

    def __init__(self, pmf_z):
        self.pmf_z = pmf_z
        self.pmf_zb = BiasPmf(pmf)

        self.pmf_y = self.PmfOfWaitTime(self.pmf_zb)
        self.pmf_x = self.pmf_y
```

Параметр pmf_z – несмещенное распределение z; pmf_zb – несмещенное распределение временного интервала, наблюдаемого пассажирами; pmf_y – распределение времени ожидания; pmf_x – распределение прошедшего времени, которое совпадает с распределением времени ожидания. Чтобы понять, почему, вспомним, что для специфической величины zb распределение является равномерным в интервале от 0 до zb. Также мы имеем

```
x = zb - y
```

Следовательно, распределение x тоже равномерное на интервале от 0 до zb.

На рис. 8.2 показаны распределения z, zb и y, основанные на данных, которые взяты из веб-сайта ветки метрополитена «Красная линия».

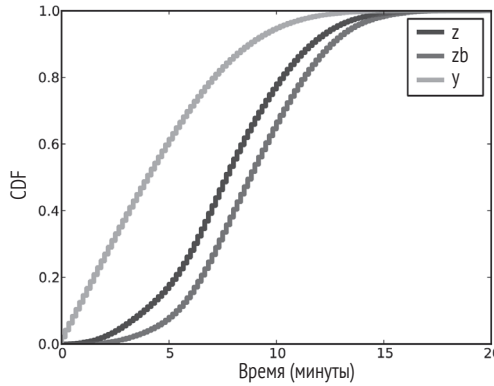


Рис. 8.2 ❖ CDF z, zb и времени ожидания y, наблюдаемого пассажирами

Чтобы показать эти распределения, Pmf был заменен на Cdf. Многие в большей мере знакомы с Pmf, но предполагается, если вы используете Cdf, то результат проще интерпретировать. И если требуется отобразить на графике несколько распределений на одной оси, то Cdf предоставляет больше возможностей.

Среднее z равно 7,8 минуты. Среднее zb равно 8,8 минуты, что на 13% больше. Среднее y равно 4,4 – наполовину меньше среднего zb.

Следует заметить: в расписании ветки метро «Красная линия» сообщается, что в часы пик интервал между поездами составляет 9 минут. Это близко к zb, но больше, чем z. MBTA (The Massachusetts Bay Transportation Authority – транспортное агентство Массачусетса) по электронной почте подтвердило, что этот интервал между поездами с учетом изменчивости сделан таким сознательно.

Предсказание ожидаемого времени

Вернемся к поставленному вопросу. Предположим, когда я прихожу на платформу, вижу 10 человек, ожидающих поезда. Как долго мне придется ждать до прихода следующего поезда?

Как обычно, давайте начнем с простейшей версии задачи. Затем будем повышать сложность. Предположим, у нас есть истинное распределение z и мы знаем, что пассажиры приходят на платформу с периодичностью 2 человека в минуту.

В этом случае мы можем:

1) для вычисления априорного распределения zb использовать распределение z, время между прибытием поездов, как это наблюдают пассажиры;
2) затем мы можем использовать число пассажиров для оценки распределения x, время, прошедшее с момента прихода последнего поезда;
3) наконец, мы используем соотношение y = zp - x, чтобы получить распределение y.

Первый шаг – создать WaitTimeCalculator, которое инкапсулирует распределения zp, x и y, приор, чтобы учесть число пассажиров.

```
wtc = WaitTimeCalculator(pmf_z)
```

pmf_z – данное распределение интервалов времени прихода поездов.

Следующий шаг – создать ElapsedTimeEstimator (определенное дальше), которое инкапсулирует апостериорное распределение x и предсказывамое распределение y.

```
ete = ElapsedTimeEstimator(wtc,
                           lam=2.0/60,
                           num_passengers=15)
```

Параметрами являются WaitTimeCalculator, периодичность прихода пассажиров, lam (выраженное в числе прибывающих пассажиров в секунду) и наблюдаемое число пассажиров, скажем 15.

Здесь определение ElapsedTimeEstimator:

```
class ElapsedTimeEstimator(object):

    def __init__(self, wtc, lam, num_passengers):
    self.prior_x = Elapsed(wtc.pmf_x)

    self.post_x = self.prior_x.Copy()
    self.post_x.Update((lam, num_passengers))

    self.pmf_y = PredictWaitTime(wtc.pmf_zb, self.post_x)
```

prior_x и posterior_x – априорное и апостериорное распределения прошедшего времени. pmf_x – предсказываемое распределение времени ожидания.

ElapsedTimeEstimator использует Elapsed и PredictWaitTime, определяемые ниже.

Elapsed – это Suite, представляющий гипотетическое распределение x. Априорное распределение x приходит прямо из WaitTimeCalculator. Затем, чтобы вычислить апостериорную вероятность, мы используем данные, состоящие из скорости (периодичности) прибытия, lam и числа пассажиров на платформе.

Здесь определение Elapsed:

```
class Elapsed(thinkbayes.Suite):

    def Likelihood(self, data, hypo):
        x = hypo
        lam, k = data
        like = thinkbayes.EvalPoissonPmf(lam * x, k)
        return like
```

Как обычно, `Likelihood` берет гипотезу и данные и вычисляет правдоподобие для данных гипотезы. В этом случае `hypo` является временем, прошедшим с момента прихода последнего поезда, а `data` – кортеж `lam` и количества пассажиров.

Правдоподобие данных есть вероятность получения `k` прибытий за время `x` для данной скорости `lam`. Мы вычисляем, используя пуассоновское распределение PMF.

Наконец, здесь определение `PredictWaitTime`:

```
def PredictWaitTime(pmf_zb, pmf_x):
    pmf_y = pmf_zb - pmf_x
    RemoveNegatives(pmf_y)
    return pmf_y
```

`pmf_zb` – распределение временных промежутков между прибытием поездов; `pmf_x` – распределение прошедшего времени, основанное на наблюдаемом числе пассажиров. Поскольку `y = zb - x`, мы можем вычислить:

```
    pmf_y = pmf_zb - pmf_x
```

Оператор вычитания вызывает `Pmf.__sub__`, перечисляет все пары `zb` и `x`, вычисляет разницу и добавляет результат в `pmf_y`.

Результирующий Pmf включает некоторые отрицательные величины, которые, как мы знаем, невозможны. Для примера, если вы пришли во время интервала между поездами 5 минут, вы не можете ждать более 5 минут.

`RemoveNegatives` удаляет неприемлемые величины из распределения и снова проводит нормализацию.

```
def RemoveNegatives(pmf):
    for val in pmf.Values():
        if val < 0:
            pmf.Remove(val)
    pmf.Normalize()
```

На рис. 8.3 показан результат. Априорное распределение `x` остается тем же самым, что и распределение `y` на рис. 8.2. Апостериорное распределение `x` показывает: после того как пришли 15 пассажиров, мы уверены, последний поезд пришел, вероятно, 5–10 минут назад. Предсказанное распределение `y` показывает с 80%-ной вероятностью, что следующий поезд ожидается менее чем через 5 минут.

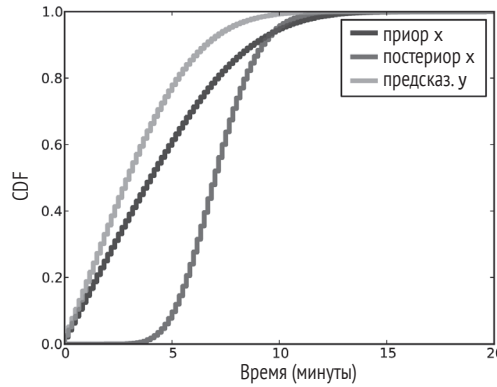


Рис. 8.3 ❖ Приор и постериор x и предсказанный y

Оценка времени прибытия

Анализ до сих пор основывался на предположении, что мы знаем (1) – распределение интервалов и (2) – периодичность прибытия пассажиров. Теперь мы готовы ослабить второе предположение.

Предположим, что вы только что приехали в Бостон и поэтому многого не знаете о скорости (периодичности) прибытия пассажиров на ветку «Красная линия». После нескольких дней поездок вы можете уверенно догадаться об этом. С некоторым усилием вы можете оценить λ и количественно.

Каждый день, приходя на платформу, вы регистрируете время и количество ожидающих пассажиров (если платформа слишком большая, вы можете просто выбрать некоторый участок). Затем вы записываете ваше время ожидания и число вновь пришедших пассажиров во время вашего ожидания.

Через пять дней вы можете иметь данные, подобные следующим:

```
k1     y     k2
--    ---    --
17    4.6     9
22    1.0     0
23    1.4     4
18    5.4    12
 4    5.8    11
```

где `k1` – число пассажиров, ожидающих, когда вы пришли, `y` – время вашего ожидания в минутах и `k2` – число пассажиров, прибывших за время вашего ожидания.

На протяжении недели вы ждали по 18 минут и видели по 36 пришедших пассажиров, так что могли оценить периодичность прихода пассажиров в 2 пассажира в минуту. Для практических целей такая оценка вполне приемлема. Но

для общности вычислим апостериорную вероятность λ и покажем, как использовать это распределение в процессе последующего анализа.

`ArrivalRateEstimator` представляет гипотезу относительно λ. Как обычно, `Likelihood` берет гипотезу и данные и вычисляет правдоподобие данных для гипотезы.

В данном случае гипотезой является λ. Данными является пара `y` и `k`, где `y` – время ожидания и `k` – число прибывших пассажиров.

```
class ArrivalRate(thinkbayes.Suite):

    def Likelihood(self, data, hypo):
        lam = hypo
        y, k = data
        like = thinkbayes.EvalPoissonPmf(lam * y, k)
        return like
```

Этот `Likelihood`, возможно, покажется вам знакомым. Он почти идентичен `Elapsed.Likelihood` в разделе «Предсказание ожидаемого времени» на стр. 89. Разница в том, что в `Elapsed.Likelihood` гипотезой является `x` – прошедшее время, а в `ArrivalRate.Likelihood` гипотезой является `lam` – периодичность прибытия. Но в обоих случаях правдоподобие `k` –вероятность наблюдать прибытие в некоторый период времени `lam`.

`ArrivalRateEstimator` инкапсулирует процесс оценки λ. Параметр `passenger_data` – это перечень `k1`, `y`, `k2` – кортежи, как в таблице выше.

```
class ArrivalRateEstimator(object):

    def __init__(self, passenger_data):
        low, high = 0, 5
        n = 51
        hypos = numpy.linspace(low, high, n) / 60

        self.prior_lam = ArrivalRate(hypos)
        self.post_lam = self.prior_lam.Copy()
        for k1, y, k2 in passenger_data:
            self.post_lam.Update((y, k2))
```

`__init__` создает гипотезу, которая является последовательностью гипотетических величин для `lam`. Затем создается априорное распределение `prior_lam`.

`for` циклически обновляет приор данными, создавая апостериорное распределение `post_lam`.

На рис. 8.4 показаны априорное и апостериорное распределения. Как ожидалось, среднее и медиана постериора близки наблюдаемой с периодичностью 2 пассажира в минуту. Но расширение апостериорного распределения накладывается на нашу неопределенность относительно λ, основанного на малой выборке.

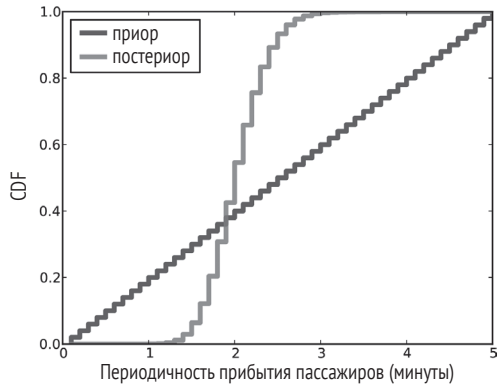


Рис. 8.4 ❖ Априорное и апостериорное распределения `lam`, основанное на данных пятидневного наблюдения за пассажирами

Включение неопределенности

Если при проведении анализа существует неопределенность относительно входных параметров, это следует учесть следующим образом.

1. Проводим анализ, основываясь на определенной величине параметра (в нашем случае λ).
2. Вычисляем распределение с неопределенным параметром.
3. Запускаем анализ для каждого параметра и генерируем множество предсказуемых распределений.
4. Вычисляем смесь предсказанных распределений, используя веса из распределений параметров.

Мы уже выполнили шаги (1) и (2). Для выполнения шагов (3) и (4) был создан класс, названный `WaitMixtureEstimator`.

```
class WaitMixtureEstimator(object):

    def __init__(self, wtc, are, num_passengers=15):
        self.metapmf = thinkbayes.Pmf()

        for lam, prob in sorted(are.post_lam.Items()):
            ete = ElapsedTimeEstimator(wtc, lam, num_passengers)
            self.metapmf.Set(ete.pmf_y, prob)

        self.mixture = thinkbayes.MakeMixture(self.metapmf)
```

`wtc` – это `WaitTimeCalculator`, содержащее распределение `zb`. `are` – это `ArrivalTimeEstimator`, содержащее распределение `lam`.

Первая строка создает мета-Pmf, которое отображает каждое возможное распределение `y` и его вероятность. Для каждого значения `lam` мы используем `ElapsedTimeEstimator`, чтобы вычислить соответствующее распределение `y` и запомнить его в мета-Pmf. Затем для вычисления смеси используется `Make.Mixture`.

На рис. 8.5 показан результат. Заштрихованные линии на заднем плане – это распределения y для каждого значения lam с толщиной линии, которая представляет вероятность. Темная линия представляет собой смесь этих распределений.

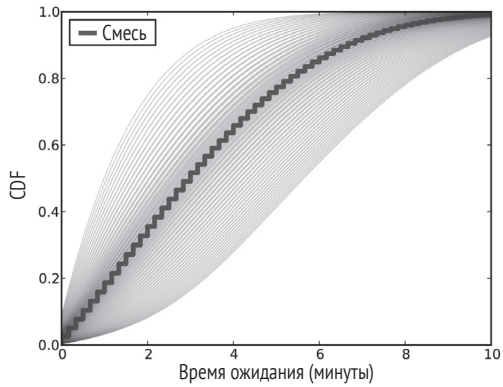


Рис. 8.5 ❖ Предсказанные распределения y для возможных значений lam

В этом случае мы можем получить очень похожий результат, используя точечную оценку lam. Поэтому нет необходимости на практике включать неопределенность в оценку.

В общем, важно включить изменчивость в отклик системы, если она является нелинейной, то есть если небольшие изменения на входе могут стать причиной больших изменений на выходе. В нашем случае изменения постериора lam невелики и отклик системы для небольших возмущений приблизительно линейный.

Анализ решений

На этом этапе мы можем использовать количество пассажиров для предсказания распределения ожидаемого времени. Настало время перейти ко второй части вопроса: в какой момент следует прекратить ждать поезд метро и идти брать такси?

Вспомним, что в оригинальном сценарии автор старался попасть на станцию метро «Южная», чтобы после пересесть на электричку. Предположим, что автор вышел из офиса, имея достаточно времени, чтобы ждать поезда в метро в течение 15 минут и все же успеть на электричку.

В этом случае автор хотел бы знать вероятность того, что y как функция num_passengers превышает 15 минут. Довольно легко использовать для этого анализ, проведенный в разделе «Предсказание времени ожидания» на стр. 89, и выполнить его для интервала num_passengers.

Но здесь есть небольшая проблема. Этот анализ чувствителен к частоте длительных задержек, и поскольку длительные задержки редки, трудно оценить их частоту.

Я имею данные только за одну неделю, и, по моим наблюдениям, самая длительная задержка была 15 минут. Поэтому нет возможности аккуратно оценить частоту более длительных задержек.

Однако можно использовать предыдущие наблюдения, чтобы сделать, по крайней мере, грубую оценку. В течение моих поездок по ветке «Красная линия» на протяжении года наблюдались 3 длительные задержки, вызванные проблемами с сигнализацией, неисправностью энергоснабжения, «действиями полиции» на другой станции. Поэтому было принято в качестве оценки 3 больших перерыва в год.

Но вспомним, что мои наблюдения были смещенными. Более вероятно, что наблюдались длительные задержки, так как они оказывали влияние на большое число пассажиров. Поэтому мы можем рассматривать эти наблюдения скорее как выборку zb, чем выборку z. Теперь как это можно сделать.

В течение года я пользовался веткой «Красная линия» около 220 раз. Поэтому следует взять наблюдаемые промежутки gap_times, сгенерировать выборку из 220 промежутков и вычислить Pmf:

```
n = 220
cdf_z = thinkbayes.MakeCdfFromList(gap_times)
sample_z = cdf_z.Sample(n)
pmf_z = thinkbayes.MakePmfFromList(sample_z)
```

Затем, чтобы получить zb, сместить pmf_z, взять выборку и добавить задержки 30, 40 и 50 минут (выраженные в секундах):

```
cdf_zp = BiasPmf(pmf_z).MakeCdf()
sample_zb = cdf_zp.Sample(n) + [1800, 2400, 3000]
```

Cdf.Sample более эффективен, чем Pmf.Sample. Поэтому, как это делалось не раз, целесообразно преобразовать Pmf в Cdf до операции с выборкой.

Затем используется выборка zb для оценки Pdf с помощью KDE, и Pdf преобразовывается в Pmf:

```
pdf_zb = thinkbayes.EstimatedPdf(sample_zb)
xs = MakeRange(low=60)
pmf_zb = pdf_zb.MakePmf(xs)
```

Наконец, чтобы получить распределение z, убирается смещение распределения zb. Распределение z используется, чтобы создать:

```
pmf_z = UnbiasPmf(pmf_zb)
wtc = WaitTimeCalculator(pmf_z)
```

Это сложный процесс. Но все эти шаги – это те же операции, которые мы выполняли ранее. Теперь мы готовы вычислить вероятность долгого ожидания:

```
def ProbLongWait(num_passengers, minutes):
    ete = ElapsedTimeEstimator(wtc, lam, num_passengers)
    cdf_y = ete.pmf_y.MakeCdf()
    prob = 1 - cdf_y.Prob(minutes * 60)
```

Данное число пассажиров на платформе `ProbLongWait` превращает в `ElapsedTimeEstimator`, выделяя распределение времени ожидания и вычисляя вероятность, что время ожидания превышает `minutes`.

На рис. 8.6 показан результат. Когда число пассажиров на платформе менее 20, мы делаем вывод, что система работает нормально. Поэтому вероятность задержки маленькая. Если на платформе 30 пассажиров, то мы приходим к выводу, что прошло 15 минут со времени прихода последнего поезда. Это дольше, чем нормальная задержка, и можно сделать вывод, что в расписании произошел какой-то сбой и следует ожидать большой задержки.

Если для нас приемлем 10%-ный шанс не успеть на электричку на станции «Южная», то следует остаться и ждать. Если на платформе меньше 30 пассажиров, стоит взять такси.

Можно пойти на шаг дальше и определить количественно цену опоздания на электричку и стоимость такси, а затем выбрать порог, минимизирующий ожидаемую стоимость.

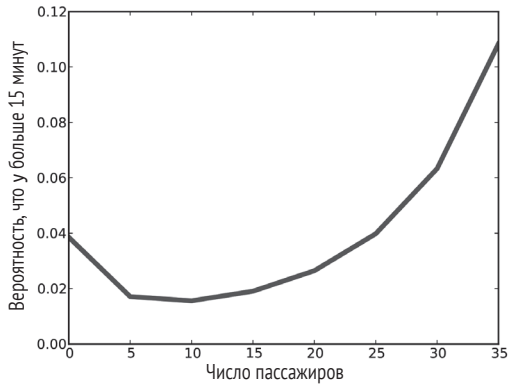


Рис. 8.6 ❖ Вероятность того, что время ожидания превысит 15 минут, как функция числа пассажиров на платформе

Обсуждение

До сих пор анализ был основан на предположении, что пассажиры приходят на платформу в один и тот же день. Для общественного транспорта в часы пик это, видимо, не такое уж плохое предположение. Но есть и очевидные исключения. Например, если в ближайшее время предстоят какие-то специальные мероприятия, то на платформу метро одновременно может прийти большое количество человек. В этом случае оценка `lam` будет слишком низкой, следо-

вательно, оценки x и y будут слишком высокими. Если какие-то необычные события происходят с частотой основных нарушений расписания, то было бы важным их включить в модель. В подобном случае, чтобы включить в расчет такие аномальные величины, стоит расширить распределение lam.

Мы начали с предположения, что знаем величину z. Поскольку пассажиры, которых вы наблюдаете, находятся на платформе только во время вашего времени ожидания y, узнать величину z не так-то и легко. Пока вы не пропустите первый поезд и не дождетесь второго, вы не узнаете промежуток между прибытием поездов z.

Однако мы можем сделать некоторые выводы о zb. Если мы после прихода зафиксируем число пассажиров на платформе, то можем оценить время x, прошедшее после прибытия последнего поезда. Затем наблюдаем y. Мы добавляем апостериорное распределение x к результату наблюдения y и получаем распределение, представляющее нашу апостериорную веру в наблюдаемую величину zb.

Мы можем использовать это распределение для обновления нашей веры относительно распределения zb. Наконец, мы можем вычислить инверсию BiasPmf, чтобы получить из распределения zb распределение z.

Я предоставляю читателю проделать это в качестве упражнения. Одна рекомендация: сначала следует прочитать главу 15. Вы можете найти контуры решения этого упражнения в http://thinkbayes.com/redline.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Упражнение

Этот пример из книги МакКея «Теория информации, вывод и обучающие алгоритмы»:

> Нестабильные частицы излучаются источником и разрушаются на расстоянии x, являющемся реальным числом и имеющем экспоненциальное распределение вероятности с (параметром) λ. Процесс распада частиц может быть наблюдаем, только если он происходит в отрезке длиной от $x = 1$ см до $x = 20$ см. N распадов были зафиксированы на расстояниях {1, 5, 2, 3, 4, 5, 12}. Каково апостериорное распределение λ?

Вы можете загрузить решение этого упражнения с http://thinkbayes.com/decay.py.

Глава 9

Двумерное измерение

Пейнтбол

Пейнтбол – это спортивная игра, в которой соревнующиеся команды пытаются стрелять друг в друга шариками, наполненными краской. Эти шарики разбиваются при ударе, оставляя цветное пятно. Обычно игра идет на площадке, оборудованной барьерами и другими препятствиями, которые могут быть использованы в качестве укрытий.

Предположим, вы играете в пейнтбол в помещении размером 30 футов в ширину и 50 футов в длину. Вы стоите около 30-футовой стены и подозреваете, что один из соперников прячется где-то поблизости. Глядя вдоль стены, вы видите на ней несколько цветных пятен одного и того же цвета. Это дает вам основание полагать, что ваш соперник недавно стрелял.

Пятна расположены на расстоянии 15, 16, 18 и 21 фут, если измерять расстояние от нижнего левого угла помещения. Как вы полагаете, основываясь на этих данных, где прячется ваш соперник?

На рис. 9.1 показан план помещения. Используя левый нижний угол как начало координат, я обозначил расположение стрелка координатами α и β (`alpha` и `beta`). Расположение пятна обозначено как `x`. Угол линии выстрела оппонента обозначен как θ (`theta`).

Задача о пейнтболе является модифицированной версией задачи о маяке – известном примере байесовского анализа. Я следовал обозначениям, использованным в книге Д. С. Сивиа «Руководство по байесовскому анализу данных, второе издание» (*Sivia D. S.* Data Analysis: a Bayesian Tutorial. 2nd ed. Oxford, 2006).

Вы можете загрузить код этой главы из веб-сайта http://thinkbayes.com/paintball.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Пакет гипотез

Для начала нам необходим пакет гипотез о расположении соперника (Suite). Каждая гипотеза состоит их двух координат (`alpha` и `beta`).

Здесь определение пакета гипотез `Suite` в задаче о пейнтболе:

```
class Paintball(thinkbayes.Suite, thinkbayes.Joint):

    def __init__(self, alphas, betas, locations):
        self.locations = locations
        pairs = [(alpha, beta)
                 for alpha in alphas
                 for beta in betas]
        thinkbayes.Suite.__init__(self, pairs)
```

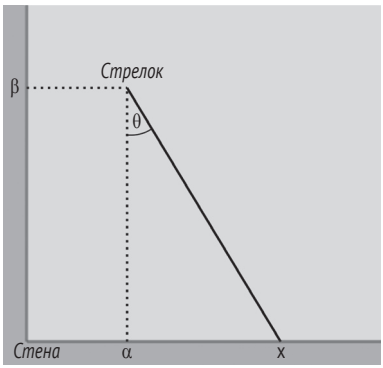


Рис. 9.1 ❖ Диаграмма расположения в задаче о пейнтболе

Класс `Paintball` наследует `Suite`, который мы видели раньше, и `Joint`, о котором будет рассказано позже.

`alphas` – перечень возможных значений для `alpha`. `betas` – перечень возможных значений для `beta`. `location` – перечень возможных расположений вдоль стены, которые запоминаются для использования в `Likelihood`.

Помещение размером 30 футов длиной и 50 футов шириной.

Здесь код, создающий `suite`:

```
alphas = range(0, 31)
betas = range(1, 51)
locations = range(0, 31)

suite = Paintball(alphas, betas, locations)
```

Априорное распределение предполагает, что все местоположения в помещении равновероятны. Представленный план помещения можно было бы детализировать, но мы начнем с простого.

Тригонометрия

Теперь нам необходима функция правдоподобия. Мы должны вычислить правдоподобие места попадания вдоль стены для данного местоположения соперника.

В качестве простой модели представим соперника в виде вращающейся турели, то есть способного равновероятно поразить цель в любом направлении. В этом случае он наиболее вероятно поразит стену в месте alpha и менее вероятно на удалении от alpha.

С помощью несложной тригонометрии мы можем вычислить вероятность места поражения стены *x*.

$$x - \alpha = \beta \tan\theta.$$

Решая это уравнение относительно θ, получаем:

$$\theta = \tan^{-1}\left(\frac{x-\alpha}{\beta}\right).$$

Берем производную по θ:

$$\frac{dx}{d\theta} = \frac{\beta}{\cos^2\theta}.$$

Эту производную мы назовем «скорость атаки». То есть скорость перемещения цели вдоль стены с увеличением θ.

Вероятность поражения цели в данной точке на стене обратно пропорциональна скорости атаки.

Если мы знаем координаты стрелка и местоположение вдоль стены, то можем вычислить скорость атаки:

```
def StrafingSpeed(alpha, beta, x):
    theta = math.atan2(x - alpha, beta)
    speed = beta / math.cos(theta)**2
    return speed
```

alpha и beta – координаты стрелка, а x – результат взятия производной x по θ.

Теперь мы можем вычислить Pmf, представляющий вероятность поражения любого места на стене. MakeLocationPmf берет alpha и beta, координаты стрелка и locations – перечень возможных значений x.

```
def MakeLocationPmf(alpha, beta, locations):
    pmf = thinkbayes.Pmf()
    for x in locations:
        prob = 1.0 / StrafingSpeed(alpha, beta, x)
        pmf.Set(x, prob)
    pmf.Normalize()
    return pmf
```

MakeLocationPmf вычисляет вероятность поражения любого местоположения, которое обратно пропорционально скорости атаки. Результат является Pmf местоположений и их вероятностей.

На рис. 9.2 показан Pmf местоположения при alpha = 10 и некотором диапазоне значений beta. Для всех значений beta наиболее вероятное положение пятна x = 10. Поскольку beta увеличивается, увеличивается и расширение Pmf.

Правдоподобие

Теперь все, что нам необходимо, – это функция правдоподобия. Мы можем использовать `MakeLocationPmf` для вычисления величины `x` при данных координатах местоположения соперника.

```
def Likelihood(self, data, hypo):
    alpha, beta = hypo
    x = data
    pmf = MakeLocationPmf(alpha, beta, self.locations)
    like = pmf.Prob(x)
    return like
```

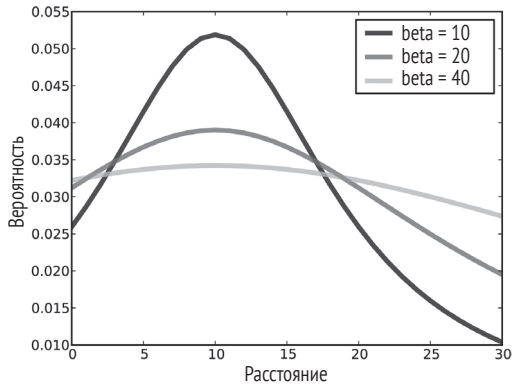


Рис. 9.2 ❖ PMF местоположения при `alpha = 10` для нескольких значений `beta`

Снова `alpha` и `beta` – это гипотетические координаты стрелка, а `x` – местоположения наблюдаемых пятен.

`Pmf` содержит вероятности каждого местоположения при данных координатах стрелка. Из этого `Pmf` мы выбираем вероятность наблюдаемого местоположения.

Все готово. Чтобы обновить suite, следует использовать `UpdateSet`, который наследует `Suite`.

```
suite.UpdateSet([15, 16, 18, 21])
```

Результат – распределение, которое отображает каждую пару координат `(alpha, beta)` в апостериорное распределение.

Совместные распределения

Когда каждая величина в распределении является кортежем переменных, это распределение называется **совместным распределением**, потому что пред-

ставляет распределения переменных «совместно». Совместное распределение содержит распределения переменных, а также информацию о соотношениях между ними.

Имея совместное распределение, мы можем вычислить распределения независимо для каждой переменной. Такие распределения называются **маргинальными распределениями** вероятностей.

`thinkbayes.Joint` обеспечивает метод, позволяющий вычислять маргинальные распределения:

```
# class Joint:

    def Marginal(self, i):
        pmf = Pmf()
        for vs, prob in self.Items():
            pmf.Incr(vs[i], prob)
        return pmf
```

`i` – индекс нужной нам переменной. В этом примере `i=0` (`i`, равное 0) указывает на распределение `alpha`, а `i=1` указывает на распределение `beta`.

Здесь код, который извлекает маргинальные распределения:

```
marginal_alpha = suite.Marginal(0)
marginal_beta = suite.Marginal(1)
```

На рис. 9,3 показан результат (преобразованный в CDF). Значение медианы близко к центру масс наблюдаемых пятен, для `alpha` равно 18. Для `beta` наиболее вероятными являются значения ближе к стене. Но за пределами 10 футов это распределение почти равномерное. Это свидетельствует о том, что данные между этими возможными местоположениями значительно не отличаются.

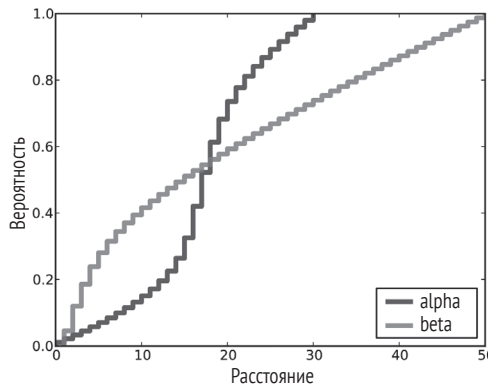


Рис. 9.3 ❖ Постериор CDF для alpha и beta для заданных данных

Имея маргиналы постериора, мы можем вычислить доверительные интервалы для каждой координаты независимо:

```
print 'alpha CI', marginal_alpha.CredibleInterval(50)
print 'beta CI', marginal_beta.CredibleInterval(50)
```

50%-ми доверительными интервалами являются (14, 21) для alpha и (5, 31) для beta. Это не слишком убедительно. 90%-ный доверительный интервал перекрывает почти все помещение.

Условные распределения

Маргинальные распределения содержат информацию о переменных независимо, и, если зависимость между переменными существует, они эту зависимость не отражают.

Одним из способов показать зависимость является вычисление **условных вероятностей**. thinkbayes.Joint обеспечивает метод, который делает это:

```
def Conditional(self, i, j, val):
    pmf = Pmf()
    for vs, prob in self.Items():
        if vs[j] != val: continue
        pmf.Incr(vs[i], prob)

    pmf.Normalize()
    return pmf
```

Здесь i снова является индексом нужной нам переменной; j является индексом условной переменной и val – условная величина.

Результатом является i-я переменная, при условии что j-я переменная является val.

Например, следующий код вычисляет условное распределение alpha для интервала величин beta.

```
    betas = [10, 20, 40]

    for beta in betas:
        cond = suite.Conditional(0, 1, beta)
```

На рис. 9.4 показан результат, который полностью описывает «апостериорное условное маргинальное распределение». Ура!

Если бы переменные были независимы, то условное распределение все равно существовало бы.

Поскольку распределения отличаются друг от друга, мы можем сказать, что переменные зависимы. Например, если мы узнаем (каким-то образом), что beta = 10, то условное распределение alpha окажется уже. Для бóльших значений beta распределение alpha будет шире.

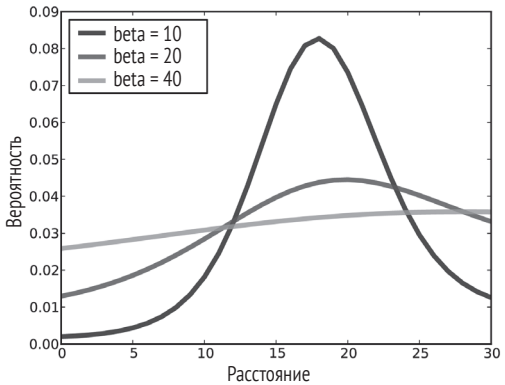


Рис. 9.4 ❖ Апостериорное распределение alpha при условии нескольких значений beta

Доверительные интервалы

Другим способом визуализации апостериорного совместного распределения является вычисление доверительных интервалов. Если вы посмотрите на доверительные интервалы раздела «Доверительные интервалы» на стр. 39, то увидите, что одна небольшая деталь не была упомянута: для данного распределения существует много интервалов с одним и тем же уровнем доверия. Если, например, вам понадобился 50%-ный интервал, вы бы могли выбрать любое множество величин, чья вероятность соответствует 50% интервала.

Если величины одномерны, то более обычным является выбор **центрального доверительного интервала**. Для примера центральный 50%-ный доверительный интервал содержит все величины между 25-м и 75-м процентилями.

При многомерных измерениях менее очевидно, каким должен быть интервал доверия. Наилучший выбор будет зависеть от контекста. Но одним из лучших выборов является максимум правдоподобия интервала доверия, который содержит наиболее вероятные величины в 50% (или каких-либо других процентах).

`thinkbayes.Joint` предоставляет метод, вычисляющий максимальное правдоподобие доверительных интервалов.

```
# class Joint:

    def MaxLikeInterval(self, percentage=90):
        interval = []
        total = 0

        t = [(prob, val) for val, prob in self.Items()]
        t.sort(reverse=True)

        for prob, val in t:
            interval.append(val)
```

```
            total += prob
            if total >= percentage/100.0:
                break

        return interval
```

Первый шаг – создать список величин в suite, расположенных в порядке уменьшающихся по величине вероятностей. Затем мы проводим инверсию списка, добавляя каждую величину в интервал, пока общая вероятность не превысит `percentage`. В результате получаем список величин из suite. Заметим, что этот список необязательно упорядоченный.

Для визуализации интервалов я написал функцию, которая делает «цветной» каждую величину в зависимости от того, сколько интервалов появляется в ней:

```
def MakeCrediblePlot(suite):
    d = dict((pair, 0) for pair in suite.Values())

    percentages = [75, 50, 25]
    for p in percentages:
        interval = suite.MaxLikeInterval(p)
        for pair in interval:
            d[pair] += 1

    return d
```

Здесь `d` – словарь, который отображает для каждого значения в suite количество интервалов, в которых оно появляется. Цикл вычисляет интервалы для нескольких величин процентов и модифицирует `d`.

На рис. 9.5 показан результат. Доверительный интервал 25% – область черного цвета около нижней части стены. Для бóльших значений процента интервал, конечно, больше и сдвинут к правой стороне помещения.

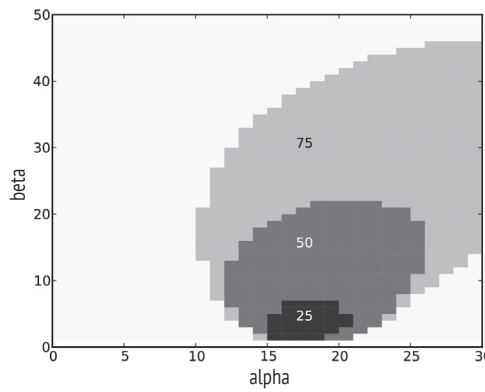


Рис. 9.5 ❖ Доверительные интервалы для координат соперника

Обсуждение

В этой главе показано, что байесовская инфраструктура предыдущих глав может быть распространена на двумерное параметрическое пространство. Единственная разница заключается в том, что каждая гипотеза представляется кортежем параметров.

Автор также представил Joint, который является родительским классом и обеспечивает применение совместных распределений `Maginal`, `Conditional` и `MakeLikeInterval`. В терминах объектно-ориентированного программирования `Joint` – это примесь (см. http://en.wikipedia.org/wiki/Mixin).

В главе много новых определений, поэтому желательно провести небольшое обозрение.

- *Совместное распределение* (*Joint distribution*): распределение, представляющее все возможные величины в многомерном пространстве и их вероятности. Пример в этой главе является двумерным пространством, составленным из величин `alpha` и `beta`. Совместное распределение представляет вероятности каждой пары (`alpha`, `beta`).
- *Маргинальное распределение* (*Marginal distribution*): распределение одного параметра в совместном распределении, где другие параметры остаются неизвестными. Как пример на рис. 9.3 показано распределение `alpha` и `beta` независимо друг от друга.
- *Условное распределение* (*Conditional distribution*): распределение одного параметра в совместном распределении, зависящее от одного или более других параметров. На рис. 9.4 несколько распределений для `alpha` при условии разных величин `beta`.

Имея совместное распределение, вы можете вычислить маргинальное и условное распределения. При достаточном количестве условных распределений вы можете воссоздать совместное распределение или, по крайней мере, его аппроксимацию. Но, имея маргинальное распределение, вы не можете получить совместное распределение, потому что потеряна информация о зависимости между параметрами.

Если существует n возможных значений для каждого из двух параметров, то большинство операций с совместным распределением занимает время, пропорциональное n^2. Если существует d параметров, то время выполнения операций составляет n^d, которое при увеличении размерности быстро становится неприемлемым.

Если вы можете обработать миллион гипотез за время, находящееся в разумных пределах, то вы можете справиться при двумерных измерениях с 1000 значений для каждого параметра, или при трехмерных измерениях со 100 значениями для каждого параметра, или при шестимерных измерениях с 10 значениями для каждого параметра.

Если вам требуется большая размерность или больше значений параметров на каждое измерение, то вы можете попробовать применить оптимизацию, которая была приведена в главе 15.

Вы можете загрузить код, использованный в этой главе, из http://thinkbayes.com/paintball.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Упражнение

В нашей простой модели соперник одинаково вероятно стреляет в любом направлении. Как упражнение давайте рассмотрим усовершенствования такой модели.

Анализ в этой главе проведен для случая, когда стрелок наиболее вероятно поражает ближайшую стену. Но в реальности, если соперник находится близко к стене, он маловероятно будет стрелять именно по стене, потому что он с небольшой вероятностью видит цель между собой и стеной.

Создайте модель, в которой учтено это обстоятельство. Постарайтесь найти более реалистичную, но не чересчур сложную модель.

Глава 10

Аппроксимация при байесовских вычислениях

Гипотеза изменчивости[1]

Мое слабое место – хитроумные теории. Недавно я побывал в замке Норумбега, вечном памятнике сомнительной теории Эбена Нортона Хорсфорда[2] (Eben Norton Horsford), изобретателя кулинарной смеси и придуманной истории. Но эта глава не об этом.

Эта глава о гипотезе изменчивости, которая

> «была выдвинута в начале девятнадцатого века Иоганном Меккелем (Johann Meckel), который утверждал, что мужчины имеют больший диапазон способностей, чем женщины, особенно интеллектуальных. Другими словами, он верил, что большинство гениев и большинство умственно отсталых людей составляют мужчины. Поскольку Меккель считал мужчин "высшими животными", он пришел к выводу, что отсутствие вариаций у женщин является признаком их неполноценности».
>
> Из http://en.wikipedia.org/wiki/Variability_hypothesis.

Мне особенно нравится заключительная часть, поскольку я подозреваю, что если бы оказалось, что женщины более изменчивы, то Меккель принял бы и это за признак неполноценности женщин. Во всяком случае, вы не будете удивлены, услышав, что доказательства гипотезы изменчивости весьма слабые.

Это утверждение укрепилось, когда недавно в моем классе мы посмотрели на данные системы наблюдения за поведенческим фактором риска Центра

---

[1] Гипотеза о том, что мужчины более склонны к изменениям, чем женщины. – *Прим. перев.*

[2] Ебен Нортон Хорсфорд – американский ученый XIX века, выдвинувший теорию о присутствии в древности нормандцев на американской земле и их поселении в Норумбеге. В Норумбеге в XIX веке был построен замок в средневековом европейском стиле. – *Прим. перев.*

контроля и предупреждения болезней (CDC's Behavioral Risk Factor Surveillance System (BRFSS)). Утверждение особенно укрепилось, когда мы увидели индивидуальные сообщения взрослых американских мужчин и женщин. Данные включают ответы 154 407 мужчин и 254 722 женщин. Вот что мы обнаружили:

- средний рост мужчин 178 см; средний рост женщин 163 см. То есть мужчины в среднем выше. Это неудивительно;
- для мужчин стандартное отклонение (роста) составляет 7,7 см; для женщин – 7,3 см, значит, в абсолютных значениях рост у мужчин может изменяться сильнее, чем у женщин;
- но сравнение изменчивости среди групп более значимо при использовании **коэффициента изменчивости** (Coefficient of Variation – CV). Этот коэффициент является стандартной девиацией, поделенной на среднее значение. Это безразмерное измерение изменчивости в относительном масштабе. Для мужчин он составляет 0,0433; для женщин – 0,0444.

Это очень близкие значения, и поэтому мы можем заключить, что подобное множество данных демонстрирует слабость доказательств в пользу гипотезы изменчивости. Применяя байесовские методы, мы можем сделать такое заключение более точным. Кроме того, ответ на этот вопрос дает нам возможность продемонстрировать некоторую технику работы со множеством данных.

Весь расчет будет сделан за несколько шагов.

1. Мы начнем с простейшего применения. Но оно работает только для набора данных до 1000 значений.
2. Вычисляя вероятности при логарифмическом преобразовании, мы можем увеличить масштаб до полного учета данных. Правда, это потребует существенного увеличения времени вычисления.
3. Наконец, мы, используя аппроксимацию при байесовских вычислениях (Approximate Bayesian Computation – ABC), существенно увеличим скорость вычислений.

Вы можете загрузить коды этой главы из http://thinkbayes.com/variability.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Среднее и стандартное отклонение

В главе 9 мы, используя совместное распределение, оценили два параметра одновременно. В этой главе, чтобы оценить параметры распределения Гаусса – среднее `mu` и стандартное отклонение `sigma`, мы используем тот же метод.

Для этой задачи я определил `Suite` под названием `Height`. Он представляет отображение каждой пары `mu` и `sigma` с ее вероятностью:

```
class Height(thinkbayes.Suite, thinkbayes.Joint):

    def __init__(self, mus, sigmas):
        thinkbayes.Suite.__init__(self)

        pairs = [(mu, sigma)
```

```
            for mu in mus
            for sigma in sigmas]

        thinkbayes.Suite.__init__(self, pairs)
```

`mus` – последовательность возможных величин для `mu`.

`sigmas` – последовательность возможных величин для `sigma`.

Априорное распределение является равномерным для всех пар `mu` и `sigma`.

Получение правдоподобия несложно. Беря гипотетические значения `mu` и `sigma`, мы вычисляем правдоподобие некоторой специальной величины `x`. Это то, что делает `EvalGaussianPdf`. Поэтому все, что нам нужно сделать, – использовать его:

```
# class Height

    def Likelihood(self, data, hypo):
        x = data
        mu, sigma = hypo
        like = thinkbayes.EvalGaussianPdf(x, mu, sigma)
        return like
```

Если вы изучали статистику, то знаете, что с математической точки зрения, когда оценивается PDF, получается плотность вероятности. Чтобы получить вероятность, необходимо проинтегрировать плотность вероятности в тех же пределах.

Но для наших целей нам не нужна вероятность. Нам только понадобится что-либо, пропорциональное вероятности. Плотность вероятности этому вполне соответствует.

Наиболее трудной частью задачи оказывается выбор надлежащих пределов для `mus` и `sigmas`. Если интервал слишком мал, то мы не учитываем значимых величин вероятности и получаем неправильный результат. Если же интервал слишком большой, то результат будет правильный, но время вычислений увеличится.

Есть возможность использовать классическую оценку для увеличения эффективности байесовской техники. Конкретно, чтобы найти вероятную локализацию для `mu` и `sigma`, мы можем использовать классические способы и, чтобы выбрать вероятное расширение, использовать стандартные ошибки этих оценок.

Если истинные параметры распределения μ и σ и мы возьмем выборку из n величин, то оценкой μ будет просто среднее `m`.

Оценкой σ будет стандартное отклонение `s`.

Стандартной ошибкой оценки μ является $s/\sqrt{n}$, а стандартной ошибкой оценки σ будет $s/\sqrt{2(n-1)}$.

Здесь код для вычисления:

```
def FindPriorRanges(xs, num_points, num_stderrs=3.0):

    # compute m and s
```

```
    n = len(xs)
    m = numpy.mean(xs)
    s = numpy.std(xs)

    # compute ranges for m and s
    stderr_m = s / math.sqrt(n)
    mus = MakeRange(m, stderr_m)

    stderr_s = s / math.sqrt(2 * (n-1))
    sigmas = MakeRange(s, stderr_s)

    return mus, sigmas
```

xs – множество данных; num_points – желаемое количество величин в интервале; num_stderrs – ширина интервала по каждую сторону оценки в числе стандартных ошибок.

Результат – пара последовательностей mus и sigmas.

Здесь MakeRange:

```
def MakeRange(estimate, stderr):
    spread = stderr * num_stderrs
    array = numpy.linspace(estimate-spread,
                           estimate+spread,
                           num_points)
    return array
```

numpy.linspace создает последовательность одинаково рассредоточенных элементов между estimate-spread и estimate+spread, включая обоих.

Обновление

И наконец, здесь код, создающий и обновляющий suite:

```
mus, sigmas = FindPriorRanges(xs, num_points)
suite = Height(mus, sigmas)
suite.UpdateSet(xs)
print suite.MaximumLikelihood()
```

Процедура может показаться ошибочной, поскольку, чтобы выбрать интервал априорного распределения, мы используем данные, а затем, чтобы провести обновление, используем данные снова. В общем-то, использование данных дважды – конечно, фикция.

Но в этом случае все хорошо. Действительно, мы используем данные для приора, но только чтобы избежать вычисления множества вероятностей, которые были бы все равно очень маленькими. С num_stderrs=4 интервал достаточно велик, чтобы перекрыть все величины с пренебрежимо малым правдоподобием. После этого их увеличение не повлияет на результаты.

В итоге приор является равномерным для всех величин mu и sigma. Но ради вычислительной эффективности мы игнорируем величины, которые не имеют значения.

Апостериорное распределение CV

Если мы имеем апостериорное совместное распределение `mu` и `sigma`, то можем вычислить распределение коэффициента изменчивости CV для мужчин и женщин, а затем вероятность того, что один превосходит другой.

Чтобы вычислить распределение CV, мы подсчитаем пары `mu` и `sigma`:

```
def CoefVariation(suite):
    pmf = thinkbayes.Pmf()
    for (mu, sigma), p in suite.Items():
        pmf.Incr(sigma/mu, p)
    return pmf
```

Затем мы используем `thinkbayes.PmfProbGreater`, чтобы вычислить вероятность, что мужчины более изменчивы.

Сам анализ прост. Но есть еще две проблемы, с которыми нам придется иметь дело:

1) когда размер множества данных растет, мы в связи с ограничениями арифметики с плавающей запятой сталкиваемся с вычислительными проблемами;
2) множество данных содержит некоторое число экстремальных значений, которые почти определенно являются ошибками. Мы должны сделать процедуру оценивания робастной в присутствии таких отклонений.

Последующие разделы поясняют эти проблемы и их решение.

Потеря значимости

Если мы возьмем первые 100 значений из множества данных BRFSS и запустим процедуру анализа, которая только что была описана, анализ пройдет без ошибок, и мы получим апостериорное распределение, которое будет выглядеть разумным.

Если мы возьмем 1000 значений и снова запустим программу, то получим ошибку в `Pmf.Normalize`:

```
ValueError: total probability is zero
```

Проблема заключается в том, что мы используем плотность вероятности, чтобы вычислить правдоподобие. А плотность непрерывного распределения стремится быть небольшой величиной. И если взять величину выборки 1000 небольших значений и перемножить их между собой, то получим очень маленькую величину. В этом случае она будет настолько маленькой, что не может быть представлена числом с плавающей запятой. Поэтому она округляется до нуля. Это явление называется **потеря значимости**. И если все вероятности в распределении равны нулю, то распределение просто отсутствует.

Возможное решение заключается в повторной нормализации Pmf после каждого обновления или после каждой выборки из 100 значений. Это будет работать, но медленно.

Лучшим решением является вычисление правдоподобия с помощью логарифмического преобразования. В этом случае вместо перемножения маленьких величин мы можем подсчитать логарифмическое правдоподобие. `Pmf`, чтобы было проще осуществить этот процесс, предусматривает методы `Log`, `LogUpdateSet` и `exp`.

`Log` вырабатывает логарифм вероятностей в Pmf:

```
# class Pmf

    def Log(self):
        m = self.MaxLike()
        for x, p in self.d.iteritems():
            if p:
                self.Set(x, math.log(p/m))
            else:
                self.Remove(x)
```

Прежде чем применить логарифмическое преобразование, `Log`, чтобы найти `m`, использует `MaxLike`, потому что наибольшая вероятность нормализуется к 1, что дает логарифм, равный 0. Другие логарифмы вероятности – все отрицательные. Если в Pmf присутствует какая-нибудь величина, равная 0, она удаляется.

Пока Pmf находится в состоянии логарифмического преобразования, мы не можем использовать `Update`, `UpdateSet` или `Normalize`. Результат был бы бессмысленным – если вы попытаетесь сделать это, то Pmf вызывает исключение. Вместо этого мы должны использовать `LogUpdate` и `LogUpdateSet`.

Здесь применение `LogUpdateSet`:

```
# class Suite

    def LogUpdateSet(self, dataset):
        for data in dataset:
            self.LogUpdate(data)
```

`LogUpdateSet` перебирает данные и вызывает `LogUpdate`:

```
# class Suite

    def LogUpdate(self, data):
        for hypo in self.Values():
            like = self.LogLikelihood(data, hypo)
            self.Incr(hypo, like)
```

`LogUpdate` подобен `Update`, за следующим исключением: он вызывает `LogLikelihood` вместо `Likelihood` и `Incr` вместо `Mult`.

Использование логарифмического правдоподобия устраняет проблему потери значимости. Но пока Pmf находится в состоянии логарифмического преобразования, мы почти ничего не сможем с ним сделать. Чтобы обратить преобразование, понадобится использовать `Exp`:

```
# class Pmf

    def Exp(self):
        m = self.MaxLike()
        for x, p in self.d.iteritems():
            self.Set(x, math.exp(p-m))
```

Если логарифмические правдоподобия являются большими отрицательными числами, результирующие правдоподобия могут потерять значимость. Поэтому `Exp` находит максимум логарифмического правдоподобия `m` и сдвигает все вероятности вверх по `m`. Результирующее распределение имеет максимум правдоподобия, равный 1. Этот процесс инвертирует логарифмическое преобразование с минимальными потерями в точности.

Логарифмическое правдоподобие

Теперь все, что нам надо, – это логарифмическое правдоподобие `LogLikelihood`:

```
# class Height

    def LogLikelihood(self, data, hypo):
        x = data
        mu, sigma = hypo
        loglike = scipy.stats.norm.logpdf(x, mu, sigma)
        return loglike
```

`norm.logpdf` вычисляет `LogLikelihood` гауссовского PDF.

Процесс общего обновления выглядит следующим образом:

```
suite.Log()
suite.LogUpdateSet(xs)
suite.Exp()
suite.Normalize()
```

Обзор: `Log` подвергает suite логарифмическому преобразованию. `LogUpdateSet` вызывает `LogUpdate`, который, в свою очередь, вызывает `LogLikelihood`. `LogUpdate` использует `Pmf.Incr`, потому что вычисление log-likelihood – это то же самое, что и умножение на правдоподобие.

После обновления логарифмические правдоподобия `LogLikelihood` становятся большими отрицательными числами. Поэтому `Exp` смещает их вверх.

Поскольку suite обращен обратным преобразованием, вероятность становится вновь «линейной», «нелогарифмической». И мы можем снова провести нормализацию.

Применяя этот алгоритм, мы можем осуществить обработку множества данных без потери значимости. Правда, значительно медленнее. На моем компьютере мне могло бы потребоваться для этого около часа. Вы можете осуществить это быстрее.

Небольшая оптимизация

Этот раздел использует математическую и компьютерную оптимизацию для ускорения с коэффициентом 100. Но в следующем разделе будет представлен еще более быстрый алгоритм. Так что если у вас есть желание перейти сразу к нему, вы можете этот раздел пропустить.

`Suite.LogUpdateSet` вызывает `LogUpdate` для каждого элемента данных. Мы можем ускорить процесс, вычисляя log-likelihood сразу для всего множества данных.

Мы начнем с гауссовского PDF:

$$\frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right].$$

И вычислим логарифм (убираем константу)

$$-\log\sigma-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2.$$

Для данной последовательности величин x_i общее логарифмическое правдоподобие:

$$\sum_i -\log\sigma-\frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2.$$

Вынося за знак суммирования величины, не зависящие от i, получаем:

$$-n\log\sigma-\frac{1}{2\sigma^2}\sum_i (x_i-\mu)^2.$$

Последнее уравнение мы можем транслировать в программу Phyton:

```
# class Height

    def LogUpdateSetFast(self, data):
        xs = tuple(data)
        n = len(xs)

        for hypo in self.Values():
            mu, sigma = hypo
            total = Summation(xs, mu)
            loglike = -n * math.log(sigma) - total / 2 / sigma**2
            self.Incr(hypo, loglike)
```

Само по себе это небольшое улучшение. Но оно открывает возможности для большего.

Заметьте, процедура суммирования зависит только от `mu` и не зависит от `sigma`. Поэтому мы можем проводить вычисления лишь один раз для каждой величины `mu`.

Чтобы избежать перекомпоновки, следует определить и **запомнить** функцию, вычисляющую суммирование, чтобы сохранить ранее вычисленные результаты в словаре (см. http://en.wikipedia.org/wiki/Memoization):

```
def Summation(xs, mu, cache={}):
    try:
        return cache[xs, mu]
    except KeyError:
        ds = [(x-mu)**2 for x in xs]
        total = sum(ds)
        cache[xs, mu] = total
        return total
```

cache запоминает ранее вычисленные суммы. Если это возможно, оператор try возвращает результат из cache. В противном случае оператор try вычисляет сумму. Затем кеширует ее и возвращает результат. Единственная загвоздка состоит в том, что ввиду наличия в кеше ключа мы не можем использовать список, поскольку это не способный хешировать тип. Вот почему LogUpdateSetFas конвертирует множество данных в кортеж.

Оптимизация, обрабатывая все множество данных 154 407 мужчин и 254 722 женщин за время менее минуты на не очень быстром компьютере, ускоряет вычисления примерно в 100 раз.

Аппроксимация при байесовских вычислениях (ABC)

Но, возможно, у вас нет столько времени. В этом случае целесообразнее использовать аппроксимацию ABC. Мотивацией для использования ABC является то, что:

1) правдоподобие любого специфического множества является очень маленьким, особенно при большом количестве данных во множестве, и мы должны использовать логарифмическое преобразование;
2) вычисления дороги, что заставляет нас применять оптимизацию;
3) на самом деле неизвестно, чего мы действительно больше хотим.

Нас не волнует получение правдоподобия точного множества данных, которое мы наблюдаем. Нас заботит получение правдоподобия какого-то множества данных, подобного тому, которое мы наблюдали. Особенно для непрерывных переменных.

Например, в задаче о евро нас не интересовал порядок, в котором мы подбрасывали монету, а интересовало только число орлов и решек. А в задаче о локомотивах нас не интересовало, какие именно поезда мы видели, а только число поездов и максимум серийных номеров.

Так же и в выборке BRFSS нас не интересует вероятность наблюдения одного какого-то множества величин (особенно поскольку в нем сотни тысяч элементов).

Более разумно было бы спросить: «Если мы опросим 100 000 человек населения с гипотетическими значениями μ и σ, каков был бы шанс выборки с наблюдаемыми средним и дисперсией?»

Для выборки из распределения Гаусса мы можем дать эффективный ответ на этот вопрос, потому что можем найти распределение статистики выборки аналитически. По существу, мы уже сделали это, когда вычисляли приор.

Если мы возьмем n величин из распределения Гаусса с параметрами μ и σ и вычислим среднее, то распределение m будет гауссовым с параметрами μ и $\sigma/\sqrt{n}$.

Подобно этому, распределение выборки стандартного отклонения s будет гауссовым с параметрами σ и $\sigma/\sqrt{2(n-1)}$.

Мы, имея гипотетические величины μ и σ, чтобы вычислить правдоподобие статистики выборки m и s, можем использовать эти выборочные распределения.

Здесь новая версия `LogUpdateSet`, которая это делает:

```
def LogUpdateSetABC(self, data):
    xs = data
    n = len(xs)

    # compute sample statistics
    m = numpy.mean(xs)
    s = numpy.std(xs)

    for hypo in sorted(self.Values()):
        mu, sigma = hypo

        # compute log likelihood of m, given hypo
        stderr_m = sigma / math.sqrt(n)
        loglike = EvalGaussianLogPdf(m, mu, stderr_m)

        #compute log likelihood of s, given hypo
        stderr_s = sigma / math.sqrt(2 * (n-1))
        loglike += EvalGaussianLogPdf(s, sigma, stderr_s)

        self.Incr(hypo, loglike)
```

На моем компьютере эта функция обрабатывает полное множество данных примерно за секунду, и результат согласуется с точным результатом до пятого знака.

Робастное оценивание

Мы почти готовы посмотреть на результат. Но нам надо преодолеть еще одну проблему. В нашем множестве данных существует некоторое число выбросов, которые почти наверняка являются ошибками. Например, там присутствуют три взрослых с объявленным ростом 61 см, что включает их перечень самых низкорослых взрослых на Земле. С другой стороны, там присутствуют четыре

женщины с объявленным ростом 229 см, что не намного ниже самой высокорослой женщины в мире.

Скорее всего, все эти данные ошибочные. Но и это допустимо. Поэтому не понятно, как с ними поступить. И нам приходится принять их как справедливые, потому что эти экстремальные значения имеют непропорциональное влияние на оцениваемую изменчивость.

Поскольку ABC в большей мере основано на статистическом суммировании, чем на полном выборочном множестве данных, мы можем сделать его более робастным в присутствии выбросов. Для этого мы выберем более устойчивое статистическое суммирование в присутствии таких выбросов.

Например, вместо использования выборочного среднего и стандартного отклонений мы могли бы использовать медиану и интер-квартиль (IQR), который является разностью между 25–75-м процентилями. В более общем смысле мы могли бы вычислять диапазон интер-процентилей (IPR), который охватывает любую заданную долю распределения, `p`:

```
def MedianIPR(xs, p):
    cdf = thinkbayes.MakeCdfFromList(xs)
    median = cdf.Percentile(50)

    alpha = (1-p) / 2
    ipr = cdf.Value(1-alpha) - cdf.Value(alpha)
    return median, ipr
```

Здесь `xs` – это последовательность величин, `p` – желаемый диапазон. Например, `p=0.5` создает интер-квартильный диапазон.

`MedianIPR` работает над вычислением CDF. Затем выводит медиану и разницу между двумя процентилями.

Мы, чтобы вычислить часть распределения, покрывающую данное число стандартных девиаций, используя гауссово CDF, можем конвертировать из `ipr` в оценку `sigma`. Например, применить широко используемое на практике правило, что 68% распределения Гаусса находится внутри стандартной девиации среднего, что оставляет по 16% с каждого края. Если мы вычисляем диапазон между 16-м и 84-м процентилями, то мы ожидаем, что результат должен быть `2*sigma`. Следовательно, `sigma` мы можем оценить, вычислив 68% IPR и разделив на 2.

В более общем виде мы могли бы использовать любое количество `sigma`.

`MedianS` представляет более общую версию этого вычисления:

```
def MedianS(xs, num_sigmas):
    half_p = thinkbayes.StandardGaussianCdf(num_sigmas) - 0.5

    median, ipr = MedianIPR(xs, half_p * 2)
    s = ipr / 2 / num_sigmas

    return median, s
```

Снова `xs` – последовательность величин; `num_sigmas` – количество стандартных девиаций, на основании которых получаем результаты. Результат – `median`, которая оценивает μ и `s`. А μ и `s`, в свою очередь, оценивают σ.

В конечном итоге мы можем переместить выборку среднего и стандартную девиацию с `median` и `s` в `LogUpdateSetABC`. И все получится.

То, что мы используем наблюдаемые процентили для оценки μ и σ, может оказаться странным. Но это является примером гибкости байесовского метода. В сущности, мы спрашиваем: «Для данных гипотетических величин μ и σ и процесса квантования, который может внести небольшую ошибку, какова вероятность создания данного множества выборочной статистики?»

Мы свободны в выборе любой желаемой статистической выборки. Но при этом понимаем, что μ и σ определяют положение и ширину распределения, и нам необходимо выбирать статистику, которая подтверждает эти характеристики.

Например, если выберем 49-й и 51-й процентили, то получим очень мало информации. Поэтому оценка σ дает относительно мало соответствующих данных. Все величины sigma будут иметь почти то же самое правдоподобие, что и создающие их наблюдаемые величины. Поэтому апостериорное распределение sigma будет во многом таким же, как и априорное.

Кто более изменчив?

Наконец, мы готовы ответить на вопрос, с которого мы начали: у кого больше коэффициент изменчивости – у мужчин или у женщин?

Используя ABC, основанный на медиане и IPR с `num_sigmas`, был вычислен постериор совместного распределения для `sigma`. На рис. 10.1 и 10.2 результат показан в виде диаграммы с `mu` по оси x, `sigma` по оси y и вероятности по оси z.

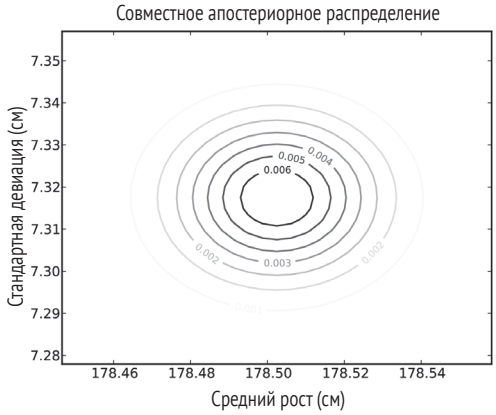


Рис. 10.1 ❖ Контурная диаграмма совместного апостериорного распределения среднего и стандартной девиации роста мужчин в США

Рис. 10.2 ❖ Контурная диаграмма совместного апостериорного распределения среднего и стандартной девиации роста женщин в США

Для каждого совместного распределения было вычислено апостериорное распределение CV. На рис. 10.3 показано распределение для мужчин и женщин, среднее для мужчин равно 0,0410; для женщин – 0,0429. Поскольку распределения не перекрываются, мы заключаем, что с большой определенностью женщины более изменчивы, чем мужчины.

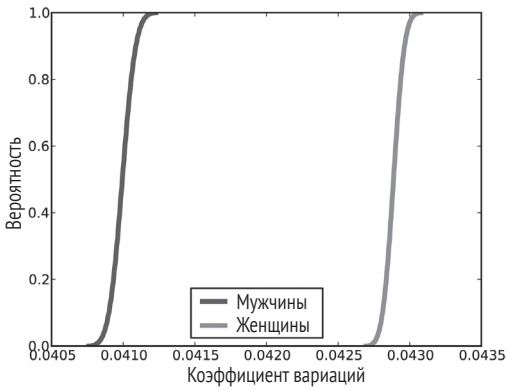


Рис. 10.3 ❖ Апостериорное распределение CV для мужчин и женщин, основанное на оценках

Итак, это конец гипотезы изменчивости? К сожалению, нет. Оказывается, что этот результат зависит от выбора диапазона интер-процентиля. С `num_sigmas=1` мы заключили, что женщины более изменчивы. Но с `num_sigmas=2` мы с таким же коэффициентом заключили, что мужчины так же более изменчивы.

Причина этой разницы – в том, что низкорослых мужчин больше и их отдаленность от среднего значения больше.

Поэтому наша оценка гипотезы изменчивости зависит от интерпретации «изменчивости». С `num_sigmas=1` мы фокусировались на людей, близких к среднему значению. Когда `num_sigmas` был увеличен, мы придали больший вес крайностям.

Чтобы решить, какие крайности приемлемы, нам необходимо более точное объявление гипотезы. Если так, то гипотеза изменчивости для ее оценки, возможно, слишком неопределенная.

Тем не менее это помогло продемонстрировать несколько новых идей, и надеюсь, что пример был интересным.

Обсуждение

Существует два возможных подхода к аппроксимации при байесовских вычислениях. Одна из интерпретаций состоит в том, что аппроксимация, как и предполагает название, означает приближение, которое вычисляется быстрее, чем точное значение.

Но следует помнить, что байесовский анализ всегда основан на моделировании решения, подразумевающего, что «точного» решения не существует. Для любой интересующей физической системы существует много возможных моделей. И каждая модель дает свой результат. Чтобы интерпретировать результаты, следует оценить модель.

Поэтому другой интерпретацией аппроксимации при байесовских вычислениях является ее представление как альтернативной модели правдоподобия. Когда мы вычисляем $p(D|H)$, то задаем вопрос: «Каково правдоподобие данных для этой гипотезы?»

Для больших множеств правдоподобие данных весьма мало. А это является предупреждением, что мы, возможно, задаем не тот вопрос. Что мы реально хотим знать? Это правдоподобие какого-либо подобного данным исхода? Причем определение «подобного» является еще одним модельным решением. Лежащая в основе ABC идея состоит в том, что два множества данных похожи, если они создают одинаковую суммарную статистику. Но в ряде случаев, как, например, в этой главе, не совсем очевидно, какую суммарную статистику следует выбрать.

Вы можете загрузить код в этой главе из http://thinkbayes.com/variability.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Упражнение

Упражнение 10.1

«Эффективный размер» – это статистика, предназначенная для измерения разницы между двумя группами (см. http://en.wikipedia.org/wiki/Effect_size).

Например, мы можем использовать данные из BRPSS, чтобы измерить разницу между мужчинами и женщинами. Для выборки из апостериорных распределений μ и σ мы можем создать апостериорное распределение этой разницы.

Но может быть лучше использовать безразмерное измерение эффективного размера, чем разницу, измеренную в сантиметрах. Одним из вариантов является использование деления на стандартное отклонение (подобно тому, как мы это делали с коэффициентом вариации).

Если параметрами для Группы 1 являются (μ_1, σ_1), а параметрами для Группы 2 являются (μ_1, σ_1), то безразмерным эффективным форматом будет:

$$\frac{\mu_1 - \mu_2}{(\sigma_1 + \sigma_2)/2}.$$

Напишите функцию, которая берет совместное распределение `mu` и `sigma` и возвращает апостериорное распределение эффективного формата.

Подсказка: если перечисление всех пар из двух распределений занимает слишком много времени, рассмотрите случайную выборку.

Глава 11

Проверка гипотез

Обратно к задаче о евро

В главе «Задача о евро» на стр. 43 была представлена задача из книги Дэвида МакКея «*Теория информации, вывод и обучающие алгоритмы*»:

> В газете «Гардиан» (The Guardian) 4 января 2002 года появилась статистическая заметка: «Монета в 1 евро бельгийского производства после 250 вращений на ребре 140 раз упала вверх орлом и 110 раз вверх решкой. “Это мне кажется очень подозрительным, – заявил Барри Блайт (Barry Blight), читающий лекции по статистике в Лондонской экономической школе. – Если центр массы монеты был несмещенным, то вероятность получения такого экстремального результата составляла бы менее 7%”.
> Но свидетельствуют ли такие данные о том, что центр массы монеты скорее смещен, чем нет?»

Мы оценили вероятность того, что монета упадет решкой вверх, но мы фактически не ответили на вопрос МакКея: «Свидетельствуют ли такие данные о том, что монета скорее смещена, чем нет?»

В главе 4 был сделано предположение, что если данные более вероятны для этой гипотезы, значит, они говорят больше в пользу гипотезы, чем в пользу альтернативы. Это эквивалентно тому, что коэффициент Байеса больше 1.

В примере с евро мы имеем для рассмотрения две гипотезы. Я буду использовать F для гипотезы, что центр массы монеты не смещен, и B для гипотезы, что центр массы смещен.

Если центр массы монеты не смещен, то правдоподобие данных $p(D|F)$ вычислить нетрудно. Мы фактически уже писали функцию, которая это делает:

```
def Likelihood(self, data, hypo):
    x = hypo / 100.0
    head, tails = data
    like = x**heads * (1-x)**tails
    return like
```

Чтобы ее использовать, мы можем создать пакет `Euro` и вызвать `Likelihood`:

```
suite = Euro()
likelihood = suite.Likelihood(data, 50)
```

p(*D*|*F*) составляет $5.5 \cdot 10^{-76}$. Это нам мало что говорит, за исключением того, что вероятность увидеть какое-то особое множество данных чрезвычайно мала. Чтобы получить отношение, необходимо два правдоподобия. Поэтому мы также должны вычислить p(*D*|*B*).

Не совсем понятно, как вычислять правдоподобие *B*, потому что не совсем ясно, что означает «смещение центра массы».

Одна из возможностей – посмотреть на данные до того, как определить гипотезу. В этом случае мы могли бы сказать, что «смещение центра массы» означает, что вероятность падения монеты орлом вверх равна 140/250.

```
actual_percent = 100.0 * 140 / 250
likelihood = suite.Likelihood(data, actual_percent)
```

Версию *B* мы назовем `B_cheat`; правдоподобие `b_cheat` равно $34 \cdot 10^{-76}$, и отношение правдоподобия равно 6.1. Поэтому мы можем сказать, что данные свидетельствуют в пользу версии *B*.

Но использование данных для формулирования гипотезы – это очевидный подлог. При таком подходе любое множество данных будет свидетельствовать в пользу *B* до тех пор, пока процент выпадения орлов не будет равен точно 50%.

Справедливое сравнение

Чтобы сделать законное сравнение, мы должны определить *B* без наличия у нас сведений о данных. Поэтому сделаем попытку дать другое определение. Если вы исследуете бельгийский евро, то можете заметить, что сторона «орла» более рельефная, чем сторона «решки». Вы можете ожидать, что это оказывает некоторое влияние на вероятность *x* упасть вверх решкой. Но вы не уверены, что от этого вероятность того, что монета упадет вверх орлом, будет большей или меньшей. Поэтому вы можете сказать: «Я думаю, что монета смещена так, что x равен или 0.6, или 0.4, но я не уверен, чему он равен».

Мы можем подумать о версии, которую автор назвал `B_two`, как о гипотезе, составленной из двух подгипотез, и можем вычислить правдоподобие для каждой из них, а затем подсчитать среднее правдоподобие:

```
like40 = suite.Likelihood(data, 40)
like60 = suite.Likelihood(data, 60)
likelihood = 0.5 * like40 + 0.5 * like60
```

Отношение правдоподобия (или байесовский коэффициент) для `b_two` равно 1.3, что показывает слабое свидетельство в пользу `b_two`.

В более общем виде, предположим, вы подозреваете, что центр массы монеты смещен. Но мы ничего не знаем о величине *x*. В этом случае вы можете создать Suite, который, чтобы представить подгипотезы от 0 до 100, был назван `b_uniform`.

```
b_uniform = Euro(xrange(0, 101))
b_uniform.Remove(50)
b_uniform.Normalize()
```

Мы инициализируем b_uniform со значением от 0 до 100. Удалим подгипотезы, при которых *x* равен 50%, так как если *x* равен 50%, то центр массы монеты не смещен. Хотя удаление или неудаление этой подгипотезы на результат влияния почти не оказывает.

Чтобы вычислить правдоподобие b_uniform, мы вычисляем каждую подгипотезу и аккумулируем взвешенное среднее.

```
def SuiteLikelihood(suite, data):
    total = 0
    for hypo, prob in suite.Items():
        like = suite.Likelihood(data, hypo)
        total += prob * like
    return total
```

Отношение правдоподобия для b_uniform равно 0.47. Это означает, что данные слабо свидетельствуют против b_uniform, по сравнению с *F*.

Если вы размышляете над вычислением, выполняемым SuiteLikelihood, то, возможно, обратили внимание на то, что это вычисление подобно обновлению. Чтобы освежить вашу память, привожу функцию Update:

```
    def Update(self, data):A здесь
        for hypo in self.Values():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo, like)
        return self.Normalize()
```

А здесь Normalize:

```
    def Normalize(self):
        total = self.Total()

        factor = 1.0 / total
        for x in self.d:
            self.d[x] *= factor

        return total
```

Значение, возвращаемое из Normalize, является суммой вероятностей в Suite, которая, в свою очередь, является средним по вероятности для подгипотез, взвешенных по предыдущим вероятностям. Update передает это значение, и, таким образом, вместо использования SuiteLikelihood мы можем вычислить вероятность b_uniform:

```
    likelihood = b_uniform.Update(data)
```

Треугольный приор

В главе 4 мы рассматривали также приор треугольной формы, который дает более высокую вероятность значения *x* вблизи 50%. Если мы выберем этот приор в качестве пакета подгипотез, то можем следующим образом вычислить его вероятность:

```
b_triangle = TrianglePrior()
likelihood = b_triangle.Update(data)
```

Отношение правдоподобия для `b_triangle` равно 0.84. Мы опять можем сказать, что данные слабо свидетельствуют против *B*, по сравнению с *F*.

Следующая таблица показывает приоры, которые мы рассмотрели, правдоподобие каждого и отношение правдоподобия (байесовского коэффициента).

Гипотеза	Правдоподобие $\cdot 10^{-76}$	Байесовский коэффициент
F	5.5	–
`B_cheat`	34	6.1
`B_two`	7.4	1.3
`B_uniform`	2.6	0.47
`B_triangle`	4.6	0.84

В зависимости от того, какое определение мы выберем, данные могут свидетельствовать за или против гипотезы, что центр массы монеты смещен. Но в каждом случае это относительно слабые свидетельства.

Таким образом, мы можем использовать байесовскую проверку гипотез для сравнения *F* и *B*, но должны проделать некоторую работу по точному определению, что понимается под гипотезой *B*. Определение этого зависит от предварительной информации о монетах и их поведении при вращении. Поэтому люди могли разумно не согласиться с правильным определением.

Мое представление этого примера следует обсуждению данной задачи Дэвидом МакКеем и приводит к такому же заключению. Вы можете загрузить код, который был использован в этой главе, из http://think.bayes.com/euro3.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Обсуждение

Байесовский коэффициент для `b_uniform` равен 0.47. Это означает, что данные свидетельствуют против этой гипотезы, по сравнению с *F*. В предыдущем разделе без объяснений было охарактеризовано это свидетельство как слабое. Частично ответ носит исторический характер. Сторонник байесовской статистики Гарольд Джеффриз (Harold Jeffreys) предложил шкалу для интерпретации байесовских коэффициентов:

Байесовский коэффициент	Сила
1–3	Не стоит упоминания
3–10	Существенно
10–30	Сильно
30–100	Очень сильно
>100	Убедительно

В примере байесовский коэффициент равен 0.47 в пользу `b_uniform`, поэтому он равен 2.1 в пользу *F*. Эта гипотеза была определена Джеффриз как «не стоящая упоминания». Другие авторы предлагают иные словесные определения. Чтобы избежать споров по поводу прилагательных, можно подумать об использовании вместо них отношения правдоподобия.

Если в вашем приоре отношение вероятностей равно 1:1 и вы видите свидетельство с байесовским коэффициентом 2, ваше апостериорное отношение вероятностей равно 2:1. В терминах вероятности данные изменяют уровень вашего доверия с 50% до 66%. Для большинства реальных задач это изменение было бы небольшим относительно ошибок моделирования и других источников неопределенности.

С другой стороны, если бы вы видели свидетельство с байесовским коэффициентом 100, ваше постериорное отношение вероятностей было бы равно 100:1, или больше чем 99%. Согласны вы или нет, что такое свидетельство «убедительно», но оно определено как сильное.

Упражнения

Упражнение 11.1

Некоторые люди склонны верить в сверхчувствительное восприятие (extrasensory perception, ESP). Например, в способность некоторых людей угадывать невидимую игральную карту с вероятностью большей, чем просто случайность.

Каково ваше априорное доверие в существование такого рода восприятия? Полагаете ли вы, что оно одинаково вероятно существует или не существует? Или вы скептически относитесь к этому? Запишите это ваше отношение вероятностей.

А теперь вычислите силу свидетельства, которая убедила бы вас, что сверхчувствительное восприятие существует, по крайней мере с 50%-ной вероятностью. Какой коэффициент Байеса был бы необходим для того, чтобы убедить вас с 90%-ной вероятностью, что сверхчувствительное восприятие существует?

Упражнение 11.2

Предположим, что ваш ответ на предыдущий вопрос 1000. То есть, чтобы изменить вашу точку зрения, свидетельство с байесовским коэффициентом 1000 в пользу ESP было бы для вас существенным. Теперь предположим, что вы прочитали статью в респектабельном медицинском научном издании, что существует свидетельство с байесовским коэффициентом 1000 в пользу ESP. Изменило бы это снова вашу точку зрения?

Если нет, то как вы разрешите очевидное противоречие? Вам может помочь в этом статья Дэвида Хьюма (David Hume) «О чудесах» в http://en.wikipedia.org/wiki/Of_Miracles.

Глава 12

Свидетельства

Интерпретация оценки SAT

Предположим, что вы декан приемной комиссии небольшого инженерного колледжа в Массачусетсе. Вы рассматриваете вопрос о двух кандидатах Алисе и Бобе, чья квалификация одинакова во многих отношениях, за исключением того, что Алиса получила лучшую оценку по математике на вступительных экзаменах SAT (Scholastic Aptitude Test). SAT – это стандартный экзамен, цель которого – определить, достаточна ли подготовка поступающего по математике для учебы в колледже.

Если Алиса получила 780 баллов, а Боб 740 (из 800 возможных), то вам хотелось бы знать, свидетельствует ли эта разница о том, что Алиса лучше подготовлена, чем Боб, а также какова сила этого факта.

По существу, обе оценки очень хорошие, и оба абитуриента, возможно, хорошо подготовлены по математике для колледжа. Поэтому настоящий декан, возможно, предложил бы для одобрения кандидатуры того или другого кандидата посмотреть, у кого из них показатели, необходимые студенту, лучше. Однако в качестве примера байесовской проверки гипотез давайте зададимся более узким вопросом: «Насколько сильным является факт того, что Алиса лучше подготовлена, чем Боб?»

Чтобы ответить на этот вопрос, нам следует принять некоторое решение о модели. Мы начнем с упрощения. Но это упрощение будет неверным. Затем мы вернемся и улучшим модель. Для начала предположим, что все экзаменационные вопросы были одинаковой сложности. На самом деле организаторы SAT выбирают вопросы в некотором диапазоне сложности, поскольку это повышает способность оценить статистические различия между абитуриентами.

Шкала

Чтобы понять порядок оценки результата в SAT, необходимо знать процедуру подсчета правильных и неправильных ответов на вопросы (исходный балл) и преобразование результатов по принятой шкале в окончательные баллы, исчисляемые в пределах от 200 до 800.

В 2009 году на экзамене по математике задавалось 54 вопроса. Исходный балл у каждого абитуриента исчислялся по принципу: один балл за количество правильных ответов и минус 1/4 балла за неправильный ответ.

Совет колледжа, организующий SAT, публикует шкалу преобразования исходных баллов в окончательные баллы. Я загрузил данные этой шкалы и построил интерполяции преобразования исходных баллов в окончательные баллы и окончательных баллов в исходные баллы. Она нам понадобится в дальнейшем.

Вы можете загрузить код для этого примера из http://thinkbayes.com/sat.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Приор

Совет колледжа публикует также окончательные баллы для всех абитуриентов. Если мы преобразуем каждый окончательный балл в исходный балл и разделим на число вопросов, то результат будет оценкой `p_correct`. Поэтому мы можем использовать распределение исходных баллов как модель априорного распределения `p_correct`.

Здесь код, который считывает и обрабатывает данные:

```
class Exam(object):

    def __init__(self):
        self.scale = ReadScale()
        scores = ReadRanks()
        score_pmf = thinkbayes.MakePmfFromDict(dict(scores))
        self.raw = self.ReverseScale(score_pmf)
        self.prior = DivideValues(raw, 54)
```

`Exam` инкапсулирует (скрывает) информацию, которую мы имеем об экзамене; `ReadScale` и `ReadRanks` считывают файл и возвращают объект, содержащий данные; `self.scale` – интерполятор, преобразующий исходные баллы в окончательные баллы и обратно; `scores` – список пары балл и частота.

`score_pmf` – Pmf окончательных баллов; `self.raw` – Pmf исходных баллов; `self.prior` – Pmf `p_correct`.

На рис. 12.1 показано априорное распределение `p_correct`. Это распределение является сжатым до максимального значения распределением Гаусса. По дизайну SAT имеет наибольшую мощность, чтобы различить абитуриентов в пределах двух стандартных отклонений среднего, и меньшую мощность вне этого диапазона.

Для каждого абитуриента был определен Suite, названный `Sat`, который представляет распределение `p_correct`.

Здесь определение:

```
class Sat(thinkbayes.Suite):

    def __init__(self, exam, score):
```

```
        thinkbayes.Suite.__init__(self)

        self.exam = exam
        self.score = score

        # start with the prior distribution
        for p_correct, prob in exam.prior.Items():
            self.Set(p_correct, prob)

        # update based on an exam score
        self.Update(score)
```

__init__ – принимает Exam-объект и окончательный балл. Он создает копию априорного распределения и затем обновляет эту копию на основании исходного балла.

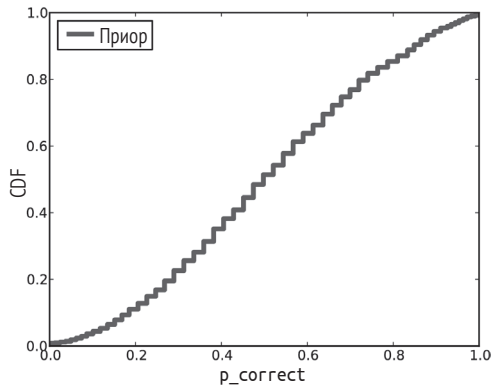


Рис. 12.1 ❖ Априорное распределение p_correct SAT абитуриентов

Как обычно, мы наследуем Update из Suite и обеспечиваем Likelihood:

```
def Likelihood(self, data, hypo):
    p_correct = hypo
    score = data

    k = self.exam.Reverse(score)
    n = self.exam.max_score
    like = thinkbayes.EvalBinomialPmf(k, n, p_correct)
    return like
```

hypo – гипотетическая величина, а data – окончательный балл.

Чтобы сохранить простоту, я интерпретирую исходный балл как число правильных ответов, игнорируя предусмотренный штраф за неправильный ответ. С этим упрощением правдоподобие принимается за биномиальное распределение, которое вычисляет вероятность k правильных ответов на n вопросов.

Постериор

На рис. 12.2 показано апостериорное распределение p_correct для Алисы и Боба по их экзаменационным оценкам. Мы можем видеть, что они перекрываются, и поэтому вполне вероятно, что p_correct фактически выше у Боба, хотя это кажется невероятным.

Это возвращает нас к первоначальному вопросу «Насколько силен факт, что Алиса лучше подготовлена, чем Боб?».

Чтобы сформулировать вопрос в терминах байесовской проверки гипотез, были определены две гипотезы:

- *A*: p_correct выше у Алисы, чем у Боба;
- *B*: p_correct выше у Боба, чем у Алисы.

Чтобы вычислить правдоподобие *A*, мы можем пересчитать все пары значений из апостериорного распределения и подсчитать полную вероятность в случаях, когда p_correct выше у Алисы, чем у Боба. И у нас уже есть функция thinkbayes.PmfProbGreater, которая выполняет такой подсчет.

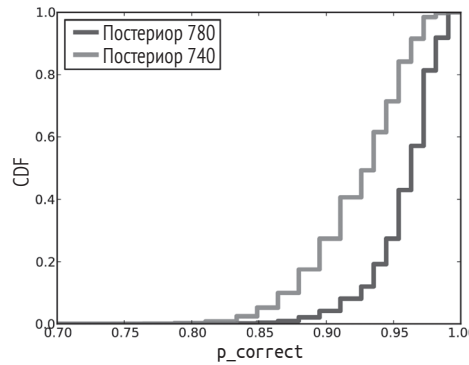


Рис. 12.2 ❖ Априорное распределение p_correct для Алисы и Боба

Из этого мы можем определить Suite, вычисляющий апостериорную вероятность *A* и *B*:

```
class TopLevel(thinkbayes.Suite):

    def Update(self, data):
        a_sat, b_sat = data

        a_like = thinkbayes.PmfProbGreater(a_sat, b_sat)
        b_like = thinkbayes.PmfProbLess(a_sat, b_sat)
        c_like = thinkbayes.PmfProbEqual(a_sat, b_sat)

        a_like += c_like / 2
        b_like += c_like / 2
```

```
        self.Mult('A', a_like)
        self.Mult('B', b_like)

        self.Normalize()
```

Обычно, когда мы вычисляем новый Suite, то наследуем `Update` и обеспечиваем `Likelihood`. В этом случае `Update` был заменен, потому что проще одновременно оценить `Likelihood` обеих гипотез.

Данные, поступающие в `Update`, – это Sat-объекты, которые представляют апостериорные распределения `p_correct`.

`a_like` – общая вероятность того, что `p_correct` выше, у Алисы; `b_like` – общая вероятность того, что `p_correct` выше, у Боба; `c_like` – общая вероятность того, что они равны. Но это равенство является искусственным решением для модели `p_correct` со множеством дискретных величин. Если мы используем больше значений, `p_correct` меньше и в пределе. Если `p_correct` непрерывен, `c_like` равен нулю. Поэтому `c_like` рассматривается как ошибка округления и делится поровну между `a_like` и `b_like`.

Здесь код, который создает и обновляет `TopLevel`:

```
exam = Exam()
a_sat = Sat(exam, 780)
b_sat = Sat(exam, 740)

top = TopLevel('AB')
top.Update((a_sat, b_sat))
top.Print()
```

Правдоподобие *A* равно 0.79, а правдоподобие B равно 0.21. Отношение правдоподобий (байесовский коэффициент) равно 3.8. Это означает, что по свидетельству тестовых баллов Алиса ответила на вопросы SAT лучше Боба. Но пока мы тестовые баллы не видели, нам казалось, что гипотезы одинаково вероятны, а когда мы узнали результат, нам следует верить, что вероятность *A* составляет 79%. То есть у Боба еще есть 21% шансов, что на самом деле он подготовлен лучше.

Улучшенная модель

Вспомним, что сделанный нами анализ был основан на упрощенной модели, в которой все экзаменационные вопросы SAT имели одинаковую сложность. В реальности некоторые вопросы сложнее, чем другие. А это означает, что разница между Алисой и Бобом может быть меньше.

Но насколько велика эта модельная ошибка? Если эта ошибка мала, то можно заключить, что первая модель, предусматривающая одинаковую сложность всех экзаменационных вопросов, вполне приемлема. Если ошибка велика, то нам необходима улучшенная модель.

В нескольких последующих разделах будет создана улучшенная модель, и выяснится (напрасная тревога!), что модельная ошибка мала. Так что если

вы удовлетворены упрощенной моделью, эту главу можете пропустить. Если же вам интересно посмотреть, как работает реальная модель, читайте дальше...

- Предположим, что каждый абитуриент имеет уровень эффективности, измеряемый его способностью ответить на вопросы SAT.
- Предположим, что каждый вопрос имеет определенный уровень сложности.
- Наконец, предположим, что шанс на то, что абитуриент ответил на вопрос правильно, есть, в соответствии с функцией, отношение эффективности абитуриента (`efficacy`) и сложности вопроса (`difficulty`):

```
def ProbCorrect(efficacy, difficulty, a=1):
    return 1 / (1 + math.exp(-a * (efficacy - difficulty)))
```

Эта функция является упрощенной версией кривой, используемой в **современной теории тестирования** (Item ResponseTheory – IRT), о которой вы можете прочитать в http://en.wikipedia.org/wiki/Item_response_theory.

`efficacy` и `difficulty` рассматриваются по одной и той же шкале, и вероятность получения вопроса зависит напрямую от разницы между ними.

Когда `efficacy` и `difficulty` равны, вероятность получения вопроса равна 50%. С увеличением `efficacy` вероятность увеличивается, достигая 100%. Если она уменьшается (или когда `difficulty` увеличивается), вероятность приближается к 0%.

Имея распределение `efficacy` среди абитуриентов и распределение `difficulty` вопросов, мы можем вычислить ожидаемое распределение исходных баллов. Сделаем это в два шага. Сначала для абитуриента с данной эффективностью мы вычислим распределение исходных баллов.

```
def PmfCorrect(efficacy, difficulties):
    pmf0 = thinkbayes.Pmf([0])

    ps = [ProbCorrect(efficacy, diff) for diff in difficulties]
    pmfs = [BinaryPmf(p) for p in ps]
    dist = sum(pmfs, pmf0)
    return dist
```

`difficulties` – список сложности каждого вопроса; `ps` – список вероятностей; `pmfs` – список двузначных Pmf-объектов.

Здесь функция, которая создает их:

```
def BinaryPmf(p):
    pmf = thinkbayes.Pmf()
    pmf.Set(1, p)
    pmf.Set(0, 1-p)
    return pmf
```

`dist` – сумма этих Pmf. Вспомним раздел «Дополнения» на стр. 55, в котором при суммировании Pmf-объектов мы в результате получали распределение

сумм. Чтобы использовать суммирование Phyton, мы должны были обеспечить идентичный для всех Pmf `Pmf0`. Поэтому `pmf + pmf0` всегда `pmf`.

Если мы знаем эффективность абитуриентов efficacy, то можем вычислить распределение их исходных баллов. Для группы абитуриентов с различной эффективностью результирующее распределение исходных баллов является смесью.

Здесь код вычисления смеси:

```
# class Exam:

    def MakeRawScoreDist(self, efficacies):
        pmfs = thinkbayes.Pmf()
        for efficacy, prob in efficacies.Items():
            scores = PmfCorrect(efficacy, self.difficulties)
            pmfs.Set(scores, prob)

        mix = thinkbayes.MakeMixture(pmfs)
        return mix
```

`MakeRawScoreDist` принимает `efficacies`, который является Pmf, представляющим распределение эффективности абитуриентов. Можно предположить, что это распределение Гаусса с нулевым средним и стандартным отклонением 1.5. Выбор в большой степени произвольный. Вероятность получения правильных ответов зависит от разницы между эффективностью и сложностью. Поэтому мы можем выбрать единицу эффективности, а затем провести соответствующую градуировку единицы сложности.

Все `pmf` – это мета-Pmf, содержащий один Pmf для каждого уровня сложности и отображающийся в коэффициент абитуриента на этом уровне. `MakeMixture` берет мета-pmf и вычисляет распределение смеси (см. раздел «Смесь» на стр. 60).

Градуировка

Если у нас есть распределение сложности задач, мы для вычисления распределения исходных баллов можем использовать `MakeRawScoreDist`. Но перед нами стоит обратная задача: у нас есть распределение исходных баллов, и мы хотим сделать вывод о распределении сложности задач.

Мы полагаем, что распределение сложности задач с параметрами `center` и `width` равномерное.

```
def MakeDifficulties(center, width, n):
    low, high = center-width, center+width
    return numpy.linspace(low, high, n)
```

После переработки нескольких комбинаций было найдено, что параметры `center=-0.05` и `width=1.8` создают распределение исходных баллов, подобных показанному на рис. 12.3.

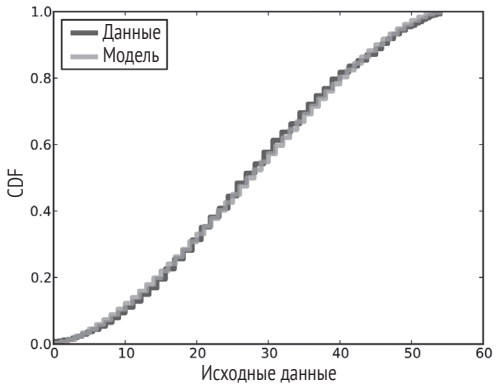


Рис. 12.3 ❖ Фактическое распределение исходных баллов и соответствующая ему модель распределения

Итак, можно предположить, что распределение сложности задач равномерное и его ширина примерно от `-1.85` до `1.75`, а эффективность является гауссовой со средним 0 и стандартным отклонением 1.5.

Следующая таблица показывает диапазон `ProbCorrect` для абитуриентов нескольких уровней их эффективности:

	Сложность		
Эффективность	**–1.85**	**–0.05**	**1.75**
3.00	0.99	0.95	0.78
1.50	0.97	0.82	0.44
0.00	0.86	0.51	0.15
–1.50	0.59	0.19	0.04
–3.00	0.24	0.05	0.01

Некто с эффективностью 3 (два стандартных отклонения от среднего) имеет 99% шансов ответить на простейшие вопросы на экзамене и 78% шансов ответить на самые сложные. На другом конце диапазона для кого-то, находящегося на два стандартных отклонения ниже среднего, шанс отвечать на простейшие вопросы равен только 24%.

Апостериорное распределение эффективности

Теперь, когда модель откалибрована, мы можем вычислить апостериорное распределение эффективности Алисы и Боба.

Здесь версия класса Sat, использующая новую модель.

```
class Sat2(thinkbayes.Suite):

    def __init__(self, exam, score):
        self.exam = exam
```

```
        self.score = score

        # start with the Gaussian prior
        efficacies = thinkbayes.MakeGaussianPmf(0, 1.5, 3)
        thinkbayes.Suite.__init__(self, efficacies)

        # update based on an exam score
        self.Update(score)
```

`Update` инициирует `Likelihood`, который вычисляет правдоподобие данного тестового балла для гипотетического уровня эффективности.

```
    def Likelihood(self, data, hypo):
        efficacy = hypo
        score = data
        raw = self.exam.Reverse(score)

        pmf = self.exam.PmfCorrect(efficacy)
        like = pmf.Prob(raw)
        return like
```

`pmf` – распределение исходных баллов абитуриента с данной эффективностью; `like` – вероятность полученных баллов.

Рисунок 12.4 показывает апостериорное распределение эффективности Алисы и Боба. Как и ожидалось, расположение распределения Алисы правее. Однако здесь опять появляется некоторое перекрытие.

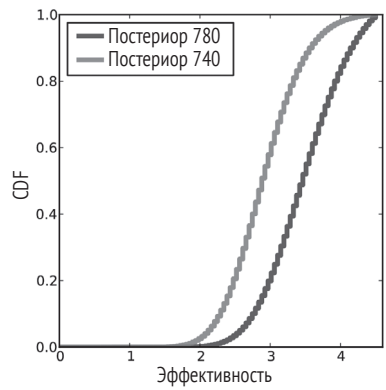


Рис. 12.4 ❖ Апостериорная вероятность эффективности Алисы и Боба

Снова, используя `TopLevel`, сравним гипотезу *A*, в которой эффективность Алисы выше, и *B*, где эффективность выше у Боба. Отношение правдоподобий равно 3.4. Это немного меньше того значения, которое было получено с простой моделью (3.8). Таким образом, модель показывает, что данные свидетельствуют в пользу Алисы. Но не так сильно, чем при прошлой оценке.

Распределение предсказания

Сделанный нами анализ пока что генерирует оценки эффективности Алисы и Боба. Но поскольку эффективность непосредственно не наблюдается, то результаты обосновать трудно.

Чтобы придать модели силу предсказания, мы можем использовать ее для ответа на вопрос: «Если Алиса и Боб снова сдадут экзамен SAT по математике, какой шанс, что результат у Алисы снова будет лучше, чем у Боба?»

Мы ответим на этот вопрос в два шага:

- сначала, чтобы создать распределение предсказания исходных баллов для каждого абитуриента, мы используем апостериорное распределение эффективности;
- далее, чтобы вычислить вероятность того, что Алиса снова получит более высокие баллы, сравним два распределения предсказания.

Мы уже имели большинство нужных нам для этого кодов. Чтобы вычислить распределения предсказания, снова используем `MakeRawScoreDist`:

```
exam = Exam()
a_sat = Sat(exam, 780)
b_sat = Sat(exam, 740)

a_pred = exam.MakeRawScoreDist(a_sat)
b_pred = exam.MakeRawScoreDist(b_sat)
```

Затем мы можем найти правдоподобие того, что Алиса лучше во втором тесте, а Боб – или лучше, или они имеют одинаковый результат:

```
a_like = thinkbayes.PmfProbGreater(a_pred, b_pred)
b_like = thinkbayes.PmfProbLess(a_pred, b_pred)
c_like = thinkbayes.PmfProbEqual(a_pred, b_pred)
```

Вероятность того, что Алиса лучше, и на втором экзамене равна 63%. Это означает, что шанс Боба получить такую же оценку или лучше равен 37%.

Заметим, у нас было больше уверенности в эффективности Алисы, чем тот результат, который мы получили во втором тесте. Постериорное отношение 3:1, и эффективность Алисы выше. Но шансы, что Алиса будет лучше на следующем экзамене, только 2:1.

Обсуждение

Мы начали эту главу с вопроса: «Насколько силен факт, что Алиса подготовлена лучше Боба?» На первый взгляд это звучит как то, что мы хотим провести тест относительно двух гипотез: кто лучше подготовлен – Алиса или Боб.

Но при вычислении правдоподобия для этих двух гипотез мы должны были решить и проблему оценки. Для каждого абитуриента требовалось найти апостериорное распределение: или `p_correct`, или `efficacy`.

Величины, подобные этим, называются **мешающими параметрами** (nuisance parameters). Поэтому нас эти параметры не интересуют. Но мы должны ответить на вопрос, который нам не интересен.

Один из способов визуализировать проведенный в этой главе анализ – это отобразить пространство параметров.

`Thinkbayes.MakeJoint` берет два Pmf, вычисляет их совместное распределение и возвращает совместный pmf каждой возможной пары величин его вероятности.

```
def MakeJoint(pmf1, pmf2):
    joint = Joint()
    for v1, p1 in pmf1.Items():
        for v2, p2 in pmf2.Items():
            joint.Set((v1, v2), p1 * p2)
    return joint
```

Эта функция предполагает, что два распределения независимы.

Рисунок 12.5 показывает совместное апостериорное распределение `p_correct` для Алисы и Боба.

Диагональная линия показывает часть пространства, где `p_correct` Алисы и Боба одинаков. Справа от этой линии Алиса более подготовлена, слева – Боб более подготовлен.

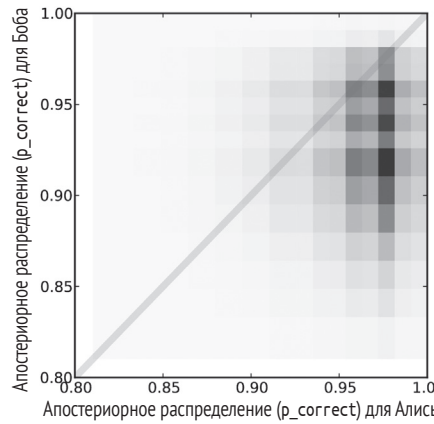


Рис. 12.5 ❖ Совместное апостериорное распределение `p_correct` для Алисы и Боба

В `TopLevel.Update`, когда мы вычисляли правдоподобия гипотез *A* и *B*, мы суммировали вероятностную массу по обе стороны этой линии. Для ячеек, которые попадают на линию, мы суммируем общую массу и делим ее между *A* и *B*.

Процедура, которую, чтобы оценить правдоподобие соревнующихся гипотез, мы применили в этой главе, – оценка мешающих параметров. Это общий байесовский метод для подобного рода задач.

Глава 13

Моделирование

В этой главе автор описывает свое решение проблемы, стоящей перед пациентами с опухолью почек. Автору кажется, что эта проблема важна и актуальна и для пациентов, страдающих этой болезнью, и для лечащих врачей.

Возможно, такое решение интересно еще и потому, что хотя это и байесовский подход к решению проблемы, но применение байесовской теоремы тут не вполне очевидно. Здесь представлено решение и код автора. А в конце данной главы мы объясним смысл байесовского подхода к решению проблемы.

Если вы хотите узнать больше технических деталей, можете прочитать статью автора об этой работе, которая расположена на http://arxiv.org/abs/1203.6890.

Проблема опухоли почек

Я частый читатель, а время от времени и участник форума статистиков на http://reddit.com/r/statistics. В ноябре 2011 года я прочитал следующее сообщение:

> «У меня IV стадия рака почки, и я хотел бы определить, появился ли у меня рак до того, как я ушел на пенсию с военной службы… Имея даты увольнения и обнаружения болезни, можно ли с вероятностью хотя бы 50/50 определить, когда я заболел? Можно ли определить вероятность заболевания на дату увольнения? При ее обнаружении опухоль была размером 15.5×15см».

Я связался с автором сообщения и получил дополнительную информацию. Я узнал, что ветераны получают различные дотации в зависимости от «более вероятно или менее вероятно», что опухоль образовалась во время их военной службы (наряду с другими обстоятельствами).

Поскольку почечная опухоль растет медленно и часто не проявляет симптомов, больные нередко остаются без лечения. Как результат врачи могут наблюдать скорость роста не подвергавшейся лечению опухоли путем сравнительного сканирования почек одного и того же пациента в различное время. Величина этой скорости упоминается в ряде статьей.

Данные были взяты из статьи Жанга и др.[1]

Я связывался с авторами статьи, чтобы узнать, могу ли я получить исходные данные. Но мне, сославшись на врачебную тайну, отказали. Я был вынужден получить необходимые мне сведения, печатая один из графиков и измеряя линейкой.

В статье скорость роста измеряется временем взаимного удвоения (RDT, Reciprocal Doubling Time). То есть в единицах удвоения в год. Опухоль с *RDT* = 1 означает увеличение объема опухоли в два раза каждый год. С *RDT* = 2 опухоль увеличивается в четыре раза за то же время, а с *RDT* = –1 увеличивается вполовину. На рис. 13.1 показано распределение *RDT* для 53 пациентов. Квадратики являются точками данных из статьи. Линия – модель распределения, которая была подогнана по данным. Положительная часть кривой хорошо аппроксимируется экспоненциальным распределением. Поэтому было использовано сочетание двух экспонент.

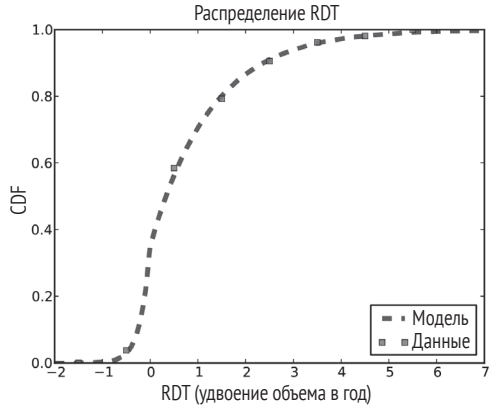


Рис. 13.1 ❖ CDF RDT в единицах удвоения в год

Простая модель

Всегда, прежде чем пытаться применить что-то более сложное, неплохо было бы начинать с простой модели. Иногда простая модель достаточна для решения проблемы. Если же нет, то простую модель можно использовать для создания более сложной модели.

Для моей простой модели я предположил, что опухоль растет с постоянным удвоением во времени, и что они трехразмерные. То есть если максимум линейного размера удваивается, то объем увеличивается в восемь раз.

---

[1] *Жанг и др.* Распределение скорости роста почечной опухоли, определенное с помощью измерителя объема // Радиология. № 250. 2009. С. 137–144. (*Zhang et al.* Distribution of Renal Tumor Growth Rates Determined by Using Serial Volumetric CT Measurements // Radiology, 250. January 2009. P. 137–144.)

Я у своего корреспондента узнал, что время между его увольнением из армии и диагнозом составляло 3291 день (около 9 лет). Поэтому моим первым вычислением стало: «Если опухоль росла с медианной скоростью, насколько большой она была бы на дату увольнения?»

Медианное время удвоения объема, согласно статье Жанга и др., составило 811 дней. Полагая, что мы имеем дело с трехразмерной геометрией, удвоение времени для линейного измерения больше в три раза.

```
# время между увольнением и диагнозом
interval = 3291.0

# удвоенное время в линейном измерении - это удвоенное время в объеме, умноженное на 3
dt = 811.0 * 3

# число удвоений со времени увольнения
doublings = interval / dt

# как велика была опухоль во время увольнения (диаметр в см)?
d1 = 15.5
d0 = d1 / 2.0 ** doublings
```

Вы можете загрузить код для этой главы из http://thinkbayes.com/kidney.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Мы получили результат `d0` примерно в 6 см. Поэтому, если бы опухоль сформировалась после даты увольнения, она должна была бы расти существенно быстрее, чем медианная скорость. Отсюда было сделано заключение, «более вероятно, чем нет», что эта опухоль сформировалась до даты увольнения.

В дополнение я вычислил скорость роста, с которой опухоль сформировалась после даты увольнения. Если принять за начальный размер 0.1 см, мы можем вычислить число удвоений до конечного размера 15.5 см:

```
# предположим, что начальный линейный размер 0.1 см
d0 = 0.1
d1 = 15.5

# сколько было бы удвоений при изменении от d0 до d1
doublings = log2(d1 / d0)

# какое линейное удвоенное время это подразумевает?
dt = interval / doublings

#вычислить время объемного удвоения и rdt
vdt = dt / 3
rdt = 365 / vdt
```

`dt` – линейное удвоенное время, поэтому `vdt` – объемное удвоенное время и `rdt` – взаимное удвоенное время.

Число удвоений в линейном измерении равно 7.3, что подразумевает RDT равным 2.4. По данным статьи Жанга и др., только 20% опухолей росло быстрее

этой скорости за период наблюдения. Поэтому я снова пришел к заключению, что «более вероятно, чем нет», что опухоль образовалась до даты увольнения.

Вычисления важны для ответа на поставленный вопрос, и по поручению моего корреспондента было написано письмо, объясняющее мое заключение в Администрацию ветеранских льгот.

Позже я рассказал о полученных результатах моему другу онкологу. Он был удивлен полученным Жангом и др. ростом скорости и тем, что они полагают относительно этих опухолей. Он предположил, что эти результаты могли бы быть интересны исследователям и врачам.

Стремясь сделать эти данные полезными, мне захотелось создать более общую модель соотношения между временем существования опухоли и ее размером.

Более общая модель

Зная размеры опухоли во время диагностики, было бы очень полезным знать вероятность, что опухоль сформировалась до данной даты. Другими словами, нас интересует распределение возрастов опухоли.

Чтобы определить вероятность распределения размера опухоли в зависимости от возраста, я смоделировал, как опухоль увеличивается. Затем, чтобы получить распределение возрастов в зависимости от размера, мы применим байесовский метод.

Моделирование начинается с маленькой опухоли и проходит следующие ступени:

1) выбор скорости роста из распределения RDT;
2) вычисление размера опухоли в конце интервала;
3) запоминание размера опухоли в каждом интервале;
4) повторение до тех пор, пока опухоль не достигнет соответствующего максимального размера.

В качестве начального размера был взят размер 0.3 см, так как зарождение раковой опухоли меньшего размера маловероятно. И еще менее вероятно снабжение опухоли такого маленького размера кровью, необходимой для быстрого роста (см. http://en.wikipedia.org/wiki/Carcinoma_in_situ).

Я выбрал интервал в 245 дней (около 8 месяцев), поскольку, по данным источника, это медианное время между измерениями. В качестве максимального размера было выбрано значение 20 см. В источнике данных диапазон наблюдаемых размеров составляет от 1.0 до 12.0 см. Поэтому проводим экстраполяцию до конца каждого интервала, но не далее. И делаем это таким образом, чтобы не оказать сильного влияния на результаты.

Моделирование основано на одном большом упрощении: скорость увеличения опухоли выбрана независимой в каждом интервале. Поэтому не зависит от возраста опухоли, ее размера или скорости во время прежних интервалов. В разделе «Последовательная корреляция» на стр. 147 я провожу анализ этих

предположений и рассматриваю более детализированные модели. Но сначала давайте рассмотрим несколько примеров. На рис. 13.2 показан размер моделируемой опухоли как функции возраста. Пунктирная линия на уровне 10 см показывает диапазон возрастов моделируемой опухоли как функции возраста: наиболее быстрый рост опухоли проявляется через восемь лет. Самый медленный – через 35 лет.

Здесь представлены результаты в терминах линейных измерений, а расчеты – в терминах объема. Чтобы преобразовать одно в другое, снова используется объем сферы данного диаметра.

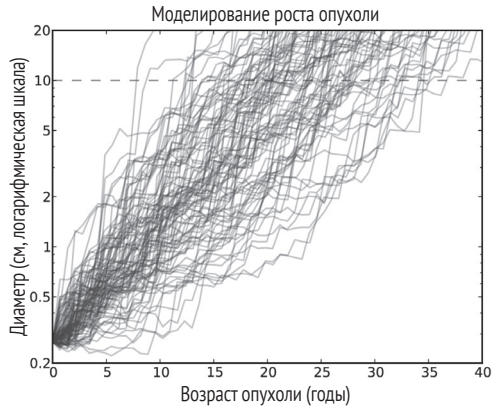


Рис. 13.2 ❖ Моделирование роста опухоли в зависимости от времени

Реализация

Здесь ядро моделирования:

```
def MakeSequence(rdt_seq, v0=0.01, interval=0.67, vmax=Volume(20.0)):
    seq = v0,
    age = 0

    for rdt in rdt_seq:
        age += interval
        final, seq = ExtendSequence(age, seq, rdt, interval)
        if final > vmax:
            break

    return seq
```

Здесь `rdt_seq` – итератор, выдвигающий случайные значения из CDF скорости роста; `v0` – начальный размер в миллилитрах (мл); `interval` – временной шаг в годах; `vmax` – окончательный объем, соответствующий линейному размеру 20 см.

Volume преобразует линейный размер, измеряемый в см. Объем измеряется в мл с упрощенным представлением опухоли в виде сферы.

```
def Volume(diameter, factor=4*math.pi/3):
    return factor * (diameter/2.0)**3
```

ExtendSequence вычисляет объем опухоли в конце интервала.

```
def ExtendSequence(age, seq, rdt, interval):
    initial = seq[-1]
    doublings = rdt * interval
    final = initial * 2**doublings
    new_seq = seq + (final,)
    cache.Add(age, new_seq, rdt)

    return final, new_seq
```

age – возраст опухоли в конце интервала; seq – кортеж, содержащий предыдущие объемы; rdt – скорость роста опухоли за время интервала; interval – размер временно́го шага в годах.

Возврат величин – final: объем опухоли в конце интервала.

new_seq – новый кортеж, содержащий величины в seq плюс новые величины final.

cache.Add записывает возраст и размер каждой опухоли в конце каждого интервала. Это будет объяснено в следующем разделе.

Кеширование совместного распределения

Здесь показано, как работает кеш:

```
class Cache(object):

    def __init__(self):
        self.joint = thinkbayes.Joint()
```

joint – совместный Pmf, который записывает частоту каждой пары возрастного размера и, таким образом, аппроксимирует совместное распределение возраста и размера.

В конце каждого моделируемого интервала ExtendSequence вызывает Add:

```
# class Cache

    def Add(self, age, seq):
        final = seq[-1]
        cm = Diameter(final)
        bucket = round(CmToBucket(cm))
        self.joint.Incr((age, bucket))
```

Опять, age – возраст опухоли, а seq – последовательность предыдущих объемов.

Прежде чем добавить в совместное распределение новые данные, мы для преобразования объема в диаметр, измеряемый в сантиметрах, используем `Diameter`:

```
def Diameter(volume, factor=3/math.pi/4, exp=1/3.0):
    return 2 * (factor * volume) ** exp
```

`CmToBucket` преобразует сантиметры в дискретный номер памяти:

```
def CmToBucket(x, factor=10):
    return factor * math.log(x)
```

Участки памяти одинаково распределены по логарифмической шкале. Использование коэффициента создает разумное число сегментов памяти. Например, 1 см отображается в сегмент 0, а 10 см – в сегмент 23.

После запуска моделирования мы можем отобразить совместное распределение как псевдоцветной график. В этом графике каждая ячейка представляет число опухолей, наблюдаемых в данной паре возраст–размер.

На рис. 13.3 показано распределение после 1000 моделирований.

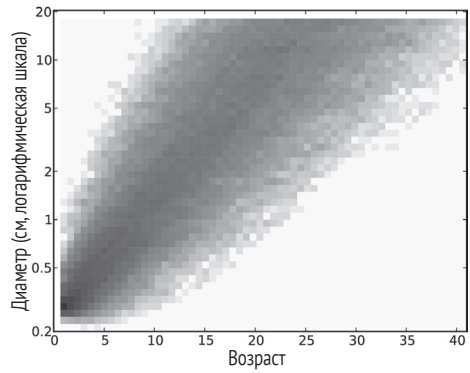


Рис. 13.3 ❖ Совместное распределение возраста и размера опухоли

Условные распределения

Сделав вертикальный срез совместного распределения, мы можем получить распределение размера опухоли для любого данного возраста. Сделав вертикальный срез, мы получим распределение возраста в зависимости от размера опухоли.

Здесь код считывает совместное распределение и строит условное распределение для данного размера опухоли:

```
# class Cache

    def ConditionalCdf(self, bucket):
        pmf = self.joint.Conditional(0, 1, bucket)
```

```
    cdf = pmf.MakeCdf()
    return cdf
```

`bucket` – целое число участков памяти, соответствующих размеру опухоли; `Joint.Conditional` – вычисляет PMF возраста, обусловленный участком памяти.

Результат – CDF возраста, обусловленного участком памяти.

На рис. 13.4 показано несколько таких CDF для диапазона размеров. Чтобы суммировать эти распределения, мы, как функции размера, можем вычислить процентили.

```
percentiles = [95, 75, 50, 25, 5]

for bucket in cache.GetBuckets():
    cdf = ConditionalCdf(bucket)
    ps = [cdf.Percentile(p) for p in percentiles]
```

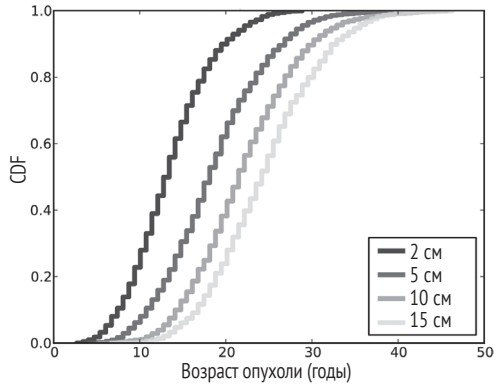


Рис. 13.4 ❖ Распределение возраста опухоли в зависимости от размера

На рис. 13.5 показаны эти процентили для участка памяти каждого размера. Точки данных вычислены из рассчитанного совместного распределения. В этой модели размер и время дискретны. Это приводит к численным ошибкам. Поэтому здесь показаны и наименьшие квадраты, соответствующие каждой последовательности процентилей.

Последовательная корреляция

Результаты до сих пор базировались на числе решений моделирования. Проведем их обзор и рассмотрим, какое из них с наибольшей вероятностью является источником ошибок.

- Чтобы преобразовать линейные размеры в объемы, мы предположили, что опухоль может быть аппроксимирована телом сферической формы. Это предположение, вероятно, подходит для опухоли в несколько сантиметров, но не для очень большой опухоли.

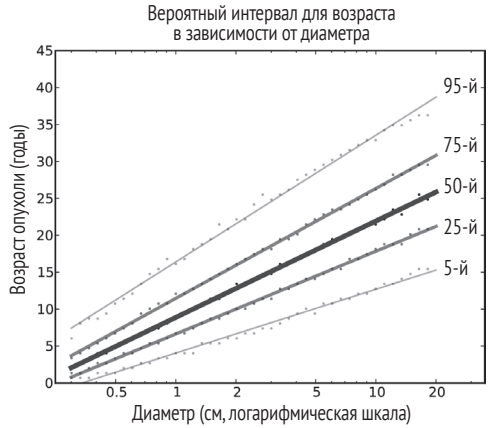


Рис. 13.5 ❖ Процентили возраста опухоли как функция размера

- Распределение скорости роста опухоли при моделировании основывалось на выбранной нами непрерывной модели. Это было сделано, чтобы удовлетворить данным, опубликованным в статье Жанга и др., и было получено на выборке из 53 пациентов. Эта выборка подходит лишь в качестве аппроксимации, и, что более важно, бóльшая выборка привела бы к другому распределению.
- Модель роста не учитывает подтипы и степень опухоли. Это соответствует заключению, опубликованному в статье Жанга и др.: «Скорость роста раковой опухоли различна в зависимости от ее величины, подтипа и степени и лежит в существенно перекрывающемся широком диапазоне». При бóльшей выборке это различие может стать очевидным.
- Распределение скорости роста не зависит от размера опухоли. Это предположение было бы нереалистичным для малых и больших опухолей, чей рост ограничен притоком крови.
- Опухоли, наблюдаемые Жангом и др., ограничены размерами от 1 до 12 см, и авторы не находят статистически значимых соотношений между размером и скоростью роста. Так что если такое соотношение существует, то оно, вероятно, слабое, по крайней мере для опухолей такого размера.
- При моделировании скорость роста опухолей во время каждого интервала не зависит от предыдущих скоростей роста. В реальности правдоподобно, что опухоль, которая росла быстро в прошлом, более вероятно будет расти и дальше быстро. Другими словами, в этом случае вероятна последовательная корреляция в скорости роста.

Из этого перечня наиболее проблематичными являются первый и последний пункты. Начнем с последовательной корреляции, а затем вернемся к сферической геометрии.

Чтобы создать модель коррелированного роста, был написан алгоритм генератора[1], создающий коррелированную последовательность из данного Cdf.

Здесь – как алгоритм работает:

1) генерируем коррелированные величины из распределения Гаусса. Это нетрудно, поскольку мы можем вычислить распределение последующей величины как зависящей от предыдущей величины;
2) используя гауссовский CDF, преобразуем каждую величину в ее кумулятивную вероятность;
3) используя данный CDF, преобразуем каждую кумулятивную вероятность в соответствующую величину.

Здесь показано, как выглядит этот код:

```
def CorrelatedGenerator(cdf, rho):
    x = random.gauss(0, 1)
    yield Transform(x)

    sigma = math.sqrt(1 - rho**2);
    while True:
        x = random.gauss(x * rho, sigma)
        yield Transform(x)
```

`cdf` – желаемый Cdf; `rho` – желаемая корреляция.

Значения `x` – из распределения Гаусса, `Transform` преобразует их в желаемое распределение.

Первые значения `x` – гауссовские со средним 0 и стандартным отклонением 1. Для последующих значений среднее и стандартное отклонение зависят от предыдущих значений. Имея предыдущее значение `x`, получаем среднее как `x*rho`, а стандартное отклонение как `1 - rho**2`.

`Transform` отображает каждое из гауссовских значений `x` в значения `y` из данного `Cdf`.

```
def Transform(x):
p = thinkbayes.GaussianCdf(x)
y = cdf.Value(p)
return y
```

`GaussianCdf` возвращает кумулятивную вероятность и вычисляет CDF стандартного распределения Гаусса `x`.

`cdf.Value` отображает кумулятивную вероятность в соответствующее значение `cdf`.

Информация, зависящая от формы `cdf`, может быть потеряна при преобразовании. Поэтому действительная корреляция может быть меньше, чем, например, при генерации 10 000 значений из распределения роста скорости с `rho=0.4`. Из этого действительная корреляция равна 0.37.

Но, поскольку мы знаем правильную корреляцию, это достаточно близко.

---

[1] Если вы незнакомы с генераторами Phyton, см. http://wiki.python.org/moin/Generators.

Вспомним, что MakeSequence принимает итератор как аргумент. Этот интерфейс позволяет ему работать с различными генераторами:

```
iterator = UncorrelatedGenerator(cdf)
seq1 = MakeSequence(iterator)

iterator = CorrelatedGenerator(cdf, rho)
seq2 = MakeSequence(iterator)
```

В данном примере seq1 и seq2 извлекаются из одного распределения. Но значения в seq1 не коррелированны, а значения в seq2 коррелированны с коэффициентом, приблизительно равным rho.

Теперь мы можем видеть, что в результатах присутствует эффект последовательной корреляции. Следующая таблица показывает процентили возраста для опухоли в 6 см при использовании некоррелированного генератора и коррелированного генератора с коэффициентом корреляции $\rho = 0.4$.

Таблица 13.1. Процентили возраста опухоли в зависимости от размера

Коррелация последовательности	Диаметр (см)	Процентили возраста опухоли				
		5-й	25-й	50-й	75-й	95-й
0.0	6.0	10.7	15.4	19.5	23.5	30.2
0.4	6.0	9.4	15.4	20.8	26.2	36.9

Корреляция делает самую большую скорость растущей опухоли еще быстрее, а самую маленькую скорость медленнее. Разница для малых процентилей скромная, но 95-й процентиль больше более, чем на 6 лет. Чтобы точнее вычислить эти процентили, нам необходимо оценить действительную последовательную корреляцию.

Но и эта модель достаточна для ответа на поставленный вначале вопрос: какова вероятность того, что опухоль с линейным диаметром 15.5 см сформировалась более 8 лет назад?

Здесь код:

```
# class Cache

    def ProbOlder(self, cm, age):
        bucket = CmToBucket(cm)
        cdf = self.ConditionalCdf(bucket)
        p = cdf.Prob(age)
        return 1-p
```

cm – размер опухоли; age – порог возраста в годах; ProbOlder преобразует размер ячейки в номер памяти, получает Cdf возраста в зависимости от ячейки памяти и вычисляет вероятность возраста превысить данную величину.

Вероятность без корреляции последовательности, что опухоль размером 15.5 см старше 8 лет, равна 0.999, или почти определено. С корреляцией 0.4 более быстрый рост опухоли более вероятен, но здесь вероятность еще только 0.995. И даже при корреляции, равной 0.8, вероятность равна 0.978.

Другим вероятным источником ошибок является предположение, что опухоль близка к сферической форме. Для опухоли с линейными размерами 15.5×15 см это предположение, скорее всего, неверное. Если опухоль такого размера сравнительно плоская, что вероятно, то ее объем равен объему сферы диаметром 6 см. С таким небольшим объемом и корреляцией 0.8 вероятность ее возраста более 8 лет остается 95%.

Итак, даже принимая во внимание ошибки моделирования, невероятно, что столь большая опухоль могла сформироваться менее, чем 8 лет назад до времени постановки диагноза.

Обсуждение

Мы прошли всю эту главу без применения теоремы Байеса и класса `Suite`, который инкапсулирует байесовское обновление. Почему?

Один подход – это думать о байесовской теореме как об алгоритме для инвертирования условных вероятностей. Имея p(B|A), мы можем вычислить p(A|B), при условии что мы знаем p(A) и p(B). Конечно, этот алгоритм полезен, только если по каким-то причинам проще вычислить p(B|A), чем p(A|B).

В таком случае применение теоремы Байеса полезно. Запуская моделирование, мы можем оценить распределение размера при условии известного возраста, или p(*size*|*age*). Но труднее получить распределение возраста при условии наличия размера или p(*age*|*size*). Поэтому неплохой возможностью кажется использование здесь теоремы Байеса.

Причиной, по которой теорема Байеса не была использована, является компьютерная эффективность. Чтобы оценить данный размер (опухоли), нам понадобилось множество моделирований. В процессе моделирования было необходимо вычисление p(*size*|*age*) для множества размеров. Фактически нам надо было вычислить все совместные распределения размера и возраста (опухоли), p(*size*, *age*).

И когда мы получили совместное распределение, нам, по сути, нет необходимости в теореме Байеса. Мы можем получить p(*age*|*size*) посредством получения разрезов совместного распределения. Как это и было продемонстрировано в `ConditionalCdf`.

Итак, мы обошлись без Байеса, но мысленно он был с нами.

Глава 14

Иерархическая модель

Задача о счетчике Гейгера

Идея для следующей задачи была позаимствована у Тома Кемпбелл-Рикеттса (Tom Campbell-Ricketts) – автора блога «Максимальная энтропия» на http://maximum-entropy-blog.blogspot.com. Он, в свою очередь, получил эту идею от Е. Т. Джейнса (E. T. Jaynes) – автора классического труда «Теория вероятности: логика науки» (*Probability Theory: The Logic of Science*).

> Предположим, что радиоактивный источник излучает частицы в сторону счетчика Гейгера со средней скоростью r частиц в секунду. Но счетчик регистрирует только часть f частиц, падающих на него. Если f равно 10%, а число регистраций 15 частиц в интервале 1 секунда, каково апостериорное распределение n действительного числа частиц, которые попали на счетчик, и каково r средней скорости излученных частиц?

Чтобы начать решать эту проблему, подумайте о цепочке причинной обусловленности, которая начинается с параметров системы и заканчивается наблюдаемыми данными:

- источник излучает частицы со средней скоростью r;
- в течение любой секунды источник излучает n частиц в направлении счетчика;
- из n этих частиц счетчик считает некоторое число частиц k.

Вероятность, что атом распадается, одинакова в любой момент времени. Поэтому радиоактивный распад хорошо моделируется процессом Пуассона. При данном r распределение Пуассона имеет параметр r.

Если мы предполагаем, что вероятность обнаружения для каждой частицы не зависит от обнаружения других частиц, то распределение k является биномиальным распределением с параметрами n и f.

Имея параметры системы, мы можем найти распределение данных. Таким образом, мы можем решить то, что называется **прямой задачей**.

Но мы пойдем другим путем: имея данные, найдем распределение параметров. Это называется **обратной задачей**. И если мы можем решить прямую задачу, то с помощью байесовских методов обратная задача также решается.

Простое начало

Давайте начнем с простой версии задачи, когда нам известна величина *r*. У нас есть величина *f*, поэтому все, что нам надо сделать, – это оценить *n*.

Сначала был определен Suite, называемый `Detector`. Suite является моделью поведения детектора. Оценим его *n*.

```
class Detector(thinkbayes.Suite):

    def __init__(self, r, f, high=500, step=1):
        pmf = thinkbayes.MakePoissonPmf(r, high, step=step)
        thinkbayes.Suite.__init__(self, pmf, name=r)
        self.r = r
        self.f = f
```

Если средняя скорость излучения равна *r* частиц в секунду, то распределение *n* является распределением Пуассона с параметром *r*. `high` и `step` определяют соответственно верхнюю и нижнюю границы для *n* и размер шага между гипотетическими величинами.

Теперь нам необходима функция правдоподобия:

```
# class Detector

    def Likelihood(self, data, hypo):
        k = data
        n = hypo
        p = self.f

        return thinkbayes.EvalBinomialPmf(k, n, p)
```

`data` – число обнаруженных частиц; `hypo` – гипотетическое число излученных частиц *n*.

Если это действительно *n* частиц и вероятность обнаружить любую одну из них равна *f*, то вероятность обнаружения *k* частиц определяется биномиальным распределением.

Это для детектора. Давайте попробуем то же сделать для диапазона величин *r*:

```
    f = 0.1
    k = 15

    for r in [100, 250, 400]:
        suite = Detector(r, f, step=1)
        suite.Update(k)
        print suite.MaximumLikelihood()
```

На рис. 14.1 показано апостериорное распределение *n* для нескольких значений *r*.

Создание иерархии

В предыдущем разделе мы предполагали, что *r* известно. Но давайте ослабим это предположение. Определим другой Suite, названный `Emitter`, который моделирует поведение эмиттера и оценивает *r*:

```
class Emitter(thinkbayes.Suite):

    def __init__(self, rs, f=0.1):
        detectors = [Detector(r, f) for r in rs]
        thinkbayes.Suite.__init__(self, detectors)
```

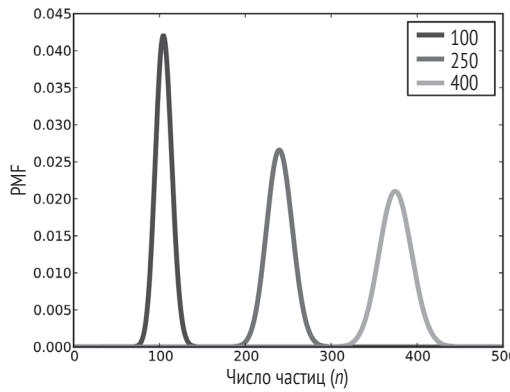


Рис. 14.1 ❖ Апостериорное распределение *n* для трех значений *r*

`rs` – это последовательность гипотетических значений *r*; `detectors` – последовательность объектов Detector, один для каждого *r*.

Эти величины находятся в Suite и Detector. Поэтому Emitter представляет собой **мета-Suite**, то есть Suite, который содержит другие Suite, как величины.

Чтобы обновить Emitter, мы должны вычислить правдоподобие данных для каждого гипотетического значения *r*. Но каждое значение представлено посредством Detector, который содержит диапазон величин *n*.

Чтобы вычислить правдоподобие данных для этого Detector, организуем цикл для величин *n* и найдем общую вероятность *k*. Это делается так:

```
# class Detector

    def SuiteLikelihood(self, data):
        total = 0
        for hypo, prob in self.Items():
            like = self.Likelihood(data, hypo)
            total += prob * like
        return total
```

Теперь напишем функцию правдоподобия для Emitter:

```
# class Detector

    def Likelihood(self, data, hypo):
        detector = hypo
        like = detector.SuiteLikelihood(data)
        return like
```

Каждый hypo – это Detector. Поэтому мы вызываем SuiteLikelihood, чтобы получить правдоподобие для гипотезы.

После обновления Emitter мы должны также обновить каждый из Detector:

```
# class Detector

    def Update(self, data):
        thinkbayes.Suite.Update(self, data)

        for detector in self.Values():
            detector.Update()
```

Такая модель со множеством уровней Suite называется **иерархической**.

Небольшая оптимизация

Вы можете вспомнить, что встречались с SuiteLikelihood на стр. 125, в разделе «Справедливое сравнение». Там было указано, что реально не нуждаемся в SuiteLikelihood, потому что общая вероятность, вычисляемая посредством SuiteLikelihood, является точной нормализованной константой, подсчитанной и возвращенной Update.

Поэтому вместо обновления Emitter и последующего обновления всех Detector мы, используя результаты из detector.Update как правдоподобие Emitter, можем сделать оба шага одновременно.

Здесь ускоренная версия Emitter.Likelihood:

```
# class Emitter

    def Likelihood(self, data, hypo):
        return hypo.Update(data)
```

И с этой версией Likelihood мы можем использовать по умолчанию версию Update. В этой версии меньше строк кода, и время выполнение этого кода быстрее, потому что не надо выполнять нормализацию дважды.

Извлечение постериоров

После обновления Emitter мы, организовав цикл для всех Detector и их вероятностей, можем получить апостериорное распределение *r*:

```
# class Emitter

    def DistOfR(self):
```

```
        items = [(detector.r, prob) for detector, prob in self.Items()]
        return thinkbayes.MakePmfFromItems(items)
```

`items` – список всех значений и их вероятностей.

Результат – Pmf *r*.

Чтобы получить апостериорное распределение *n*, следует вычислить смесь всех Detector. Здесь можно использовать `thinkbayes.MakeMixture`, который принимает мета-Pmf, отображающий каждое распределение в его вероятность. И это точно то, чем является Emitter:

```
# class Emitter

    def DistOfN(self):
        return thinkbayes.MakeMixture(self)
```

На рис. 14.2 показан результат. Неудивительно, что наиболее вероятная величина для *n* равна 150. Для данных *f* и *n* ожидаемый результат равен $k = fn$. Поэтому для данных *f* и *k* ожидаемый результат равен k/f, то есть 150.

И если 150 частиц излучается в одну секунду, то наиболее вероятной величиной *r* является 150 частиц в секунду. Следовательно, центральным значением апостериорного распределения *r* тоже является 150.

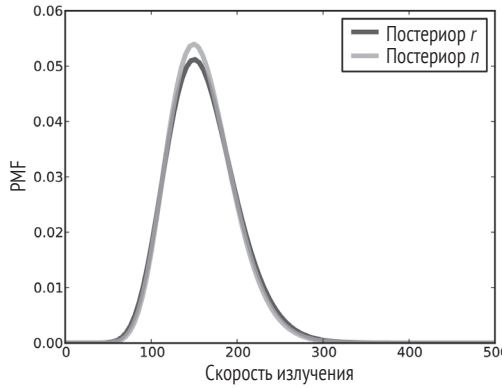


Рис. 14.2 ❖ Апостериорные распределения *n* и *r*

Апостериорные распределения *r* и *n* похожи. Единственным отличием является то, что мы меньше уверены в *n*. В общем, мы можем более определенно говорить о большом диапазоне скоростей излучения *r*, чем о числе частиц, излучаемых в любую отдельную секунду *n*.

Вы можете загрузить код этой главы из http://thinkbayes.com/jaynes.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Обсуждение

Задача о счетчике Гейгера демонстрирует связь между причинной обусловленностью и иерархическим моделированием. В этом примере скорость излучения r оказывает причинное (каузальное) влияние на число частиц n. А это оказывает причинное влияние и на подсчет частиц k.

Иерархическая модель отражает структуру системы с причинами сверху и эффектами внизу.

1. На верхнем уровне мы начинаем с диапазона гипотетических значений для r.
2. Для каждого значения r мы имеем диапазон значений для n и априорное распределение n, зависящее от r.
3. Когда мы обновляем модель, то идем снизу вверх. Мы вычисляем апостериорное распределение n для каждого значения r, а затем апостериорное распределение r.

Таким образом, каузальная информация спускается по принципу иерархии, а вывод поднимается вверх.

Упражнение

Упражнение 14.1

Это упражнение тоже навеяно примером из «Теории вероятности» Джейнса.

Предположим, что вы купили ловушку для москитов, которая предположительно уменьшает популяцию москитов около вашего дома. Каждую неделю вы опустошаете ловушку и подсчитываете число пойманных москитов. После первой недели вы насчитали 30 москитов. После второй недели вы насчитали 20 москитов. Оцените процентное изменение числа москитов в вашем дворе.

Для ответа на вопрос вы должны принять некоторое решение о модели.

Здесь некоторые предложения:

1. Предположите, что каждую неделю большое число москитов N рождается на болоте недалеко от вашего дома.
2. В течение недели некоторая часть их f_1 прилетает в ваш двор, и часть из них f_2 попадает в ловушку.
3. Ваше решение должно учитывать ваш приор доверия о том, как N вероятно будет изменяться от недели к неделе. Чтобы моделировать процентное изменение в N, добавьте уровень в иерархию.

Глава 15

Борьба с размерностью

Бактерии пупка

«Биологическое разнообразие пупка 2.0» – национальный научный проект, целью которого является выявление бактериальных видов, которые могут быть обнаружены в пупке человека (http://bbdata.yourwildlife.org). Проект может показаться несколько эксцентричным, однако он является частью увеличивающегося интереса к микрофлоре человека, ко множеству микроорганизмов, живущих в человеческой коже и других частях тела.

На начальной стадии проекта исследователи взяли мазок у 60 добровольцев, используя мультиплексное пиросеквенирование для последовательного извлечения фрагментов 16S рДНК. А затем идентифицировали разновидность или класс полученных фрагментов. Каждый идентифицированный фрагмент называется «прочитанным».

Мы можем использовать эти данные для ответа на несколько сопутствующих вопросов:

- Можем мы оценить общее число разновидностей, основываясь на числе наблюдаемых разновидностей?
- Можем мы оценить распространенность каждой разновидности, то есть части общей популяции, принадлежащей каждой разновидности?
- Если мы планируем собрать дополнительное количество образцов, можем ли мы предсказать, сколько новых разновидностей, скорее всего, обнаружим еще?
- Сколько дополнительных прочитанных фрагментов необходимо, чтобы увеличить часть наблюдаемых разновидностей до заданного порога?

Эти вопросы составляют так называемую **проблему невидимых разновидностей**.

Львы, тигры и медведи

Начнем с упрощенной версии проблемы. Мы точно знаем, что есть три разновидности. Назовем их львы, тигры и медведи. Представим, что мы посетили заповедник и увидели 3 льва, 2 тигра и одного медведя.

Если мы имеем равные шансы наблюдать любое животное в заповеднике, то число каждой разновидности, которое мы видим, подчиняется мультиномиальному распределению. Если распространенность львов, тигров и медведей в заповеднике есть `p_lion` и `p_tiger` и `p_bear` , то правдоподобность увидеть 3 львов, 2 тигров и медведя:

```
p_lion**3 * p_tiger**2 * p_bear**1
```

Метод, который напрашивается сам собой, но не является правильным, заключается в использовании бета-распределения для описания распространенности отдельно каждой разновидности. Так мы делали в разделе «Бета-распределение» на стр. 48. Например, мы заметили 3 львов. Трое остальных наблюдаемых животных ко львам не принадлежат. Если мы будем рассуждать так же, как в случае с «орлами» и «решками», то апостериорное распределение `p_lion` будет:

```
beta = thinkbayes.Beta()
beta.Update((3, 3))
print beta.MaximumLikelihood()
```

Максимум правдоподобия оценивается для `p_lion` по наблюдаемому уровню в 50%. Соответственно, среднеквадратичное отклонение для `p_tiger` равно 33% и для `p_bear` 17%.

Но здесь есть две проблемы:

1) мы неявно использовали для каждой разновидности приор, равномерный в пределах от 0 до 1. Но, поскольку мы знаем, что присутствует три разновидности, такой приор будет неверным. Правильный приор будет иметь среднее 1/3, а правдоподобие того, что распространенность каждой разновидности имеет 100%, будет равна нулю;
2) распределения для каждой разновидности не являются независимыми, поскольку распространенность разновидностей должна в сумме быть равна единице (1). Чтобы учесть эту зависимость, нам необходимо совместное распределение всех трех распространенностей.

Для решения этих двух проблем мы можем использовать распределение Дирихле (см. http://en.wikipedia.org/wiki/Dirichlet_distribution). Чтобы описать совместное распределение `p_lion`, `p_tiger` и `p_bear`, следует применить тот же способ, который мы использовали в отношении бета-распределения при решении задачи о смещении центра массы в монете.

Распределение Дирихле – это многомерное обобщенное бета-распределение. Вместо двух возможных исходов, таких как орел и решка, распределение Дирихле позволяет иметь любое число исходов. В данном случае три. Если имеется n исходов, распределение Дирихле имеет n параметров – от α_1 до α_n.

Здесь определение из класса `thinkbayes.ru`, представляющего распределение Дирихле:

```
class Dirichlet(object):

    def __init__(self, n):
```

```
        self.n = n
        self.params = numpy.ones(n, dtype=numpy.int)
```

n – число измерений; все параметры изначально равны 1.

Для запоминания параметров мы используем массив numpy. Поэтому здесь можно воспользоваться преимуществами операций с массивами.

Данное распределение Дирихле, являющееся маргинальным распределением для каждого распространения разнообразий животных, есть бета-распределение, которое может быть вычислено следующим образом:

```
def MarginalBeta(self, i):
    alpha0 = self.params.sum()
    alpha = self.params[i]
    return Beta(alpha, alpha0-alpha)
```

Здесь i – индекс желаемого маргинального распределения; alpha0 – сумма параметров; alpha – параметр для данной разновидности.

В этом примере приор маргинального распределения для каждой разновидности – Beta(1,2). Мы можем вычислить приор средних значений так:

```
dirichlet = thinkbayes.Dirichlet(3)
for i in range(3):
    beta = dirichlet.MarginalBeta(i)
    print beta.Mean()
```

Как и ожидалось, приор среднего разнообразия для каждой разновидности составляет 1/3.

Чтобы обновить распределение Дирихле, добавим наблюдения к параметрам. Сделаем это так:

```
def Update(self, data):
    m = len(data)
    self.params[:m] += data
```

Здесь data – последовательность отсчетов будет в том же порядке, как и в params. Поэтому в этом примере должно присутствовать количество львов, тигров и медведей.

data может быть короче params. В этом случае обнаружится присутствие некоторых разновидностей, которые ранее не наблюдались.

Здесь код, обновляющий dirichlet данными наблюдения и вычисляющий апостериорные маргинальные наблюдения:

```
data = [3, 2, 1]
dirichlet.Update(data)

for i in range(3):
    beta = dirichlet.MarginalBeta(i)
    pmf = beta.MakePmf()
    print i, pmf.Mean()
```

На рис. 15.1 показаны результаты. Постериор средних распространений составляет 44%, 33% и 22%.

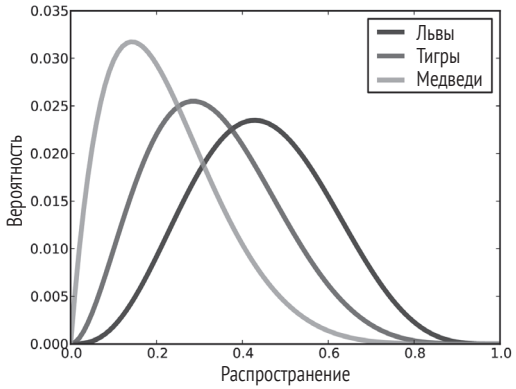


Рис. 15.1 ❖ Распределение распространений для трех разновидностей

Иерархическая версия

Мы решили упрощенную версию задачи, в которой нам известно, сколько существует разновидностей, и мы можем оценить распространенность каждой.

Теперь вернемся к начальной задаче оценки общего числа разновидностей. Чтобы решить эту задачу, был определен мета-Suite, который является Suite'том, содержащим другие Suite в качестве гипотез. В таком случае верхний уровень Suite содержит гипотезы о распространенности.

Здесь определение класса:

```
class Species(thinkbayes.Suite):

    def __init__(self, ns):
        hypos = [thinkbayes.Dirichlet(n) for n in ns]
        thinkbayes.Suite.__init__(self, hypos)
```

`__init__` принимает список возможных значений для `n` и создает список объектов Дирихле.

Здесь код, создающий верхний уровень suite:

```
ns = range(3, 30)
suite = Species(ns)
```

`ns` – это список возможных значений для `n`. У нас есть три разновидности. Следовательно, их должно быть, по крайней мере, столько же. Была выбрана верхняя граница, показавшаяся разумной. Позже следует проверить, что вероятность превышения этой границы мала. По крайней мере, вначале мы предполагаем, что любые значения в этом диапазоне равновероятны.

Для обновления иерархической модели следует обновить все уровни. Обычно первым обновляется нижний уровень. Далее происходит движение вверх. Но в данном случае мы можем обновить первым верхний уровень.

```
#class Species

    def Update(self, data):
        thinkbayes.Suite.Update(self, data)
        for hypo in self.Values():
            hypo.Update(data)
```

`Species.Update` вызывает `Update` в родительском классе, затем осуществляет цикл по всем субгипотезам и обновляет их.

Теперь нам необходима функция правдоподобия:

```
# class Species

    def Likelihood(self, data, hypo):
        dirichlet = hypo
        like = 0
        for i in range(1000):
            like += dirichlet.Likelihood(data)

        return like
```

Здесь `data` – последовательность отсчетов; `hypo` – Дирихле-объект; `Species.Likelihood` – 1000 раз вызывает `dirichlet.Likelihood` и возвращает общее количество. Почему число вызовов равно 1000? Потому что `dirichlet.Likelihood`, по существу, не вычисляет правдоподобие данных всего распределения Дирихле. Вместо этого `dirichlet.Likelihood` выбирает по одному отсчету из гипотетического распределения и вычисляет правдоподобие данных выборочного множества распространений.

Здесь показано, как это выглядит:

```
# class Dirichlet

    def Likelihood(self, data):
        m = len(data)
        if self.n < m:
            return 0

        x = data
        p = self.Random()
        q = p[:m]**x
        return q.prod()
```

Длина data – число наблюдаемых разнообразий. Если мы видим больше разнообразий, чем мы полагали, то правдоподобие равно 0.

В противном случае мы выбираем случайное множество распространений p и вычисляем мультиномиальное PMF, которое:

$$c_x p_1^{x_1} \ldots p_n^{x_n},$$

где p_i – распространение i-го разнообразия, и x_i – наблюдаемое количество. Первый элемент c_x – это мультиномиальный коэффициент. При вычислении он был отброшен, потому что этот мультиномиальный коэффициент зависит только от данных, а не от гипотез, и потому он нормализуется (см. http://en.wikipedia.org/wiki/Multinomial_distribution).

`m` – число наблюдаемых разнообразий. Нам необходимы только первые `m` элементов p. Для остальных $x_i = 0$, потому что $p_i^{x_i}$ – это 1, и мы можем исключить его из произведения.

Случайная выборка

Существует два способа генерировать случайную выборку в распределении Дирихле. Первый способ – использовать маргинальное бета-распределение. Но в этом случае нужно, получая выборку, масштабировать остальные, чтобы сумма равнялась 1 (см. en.wikipedia.org/wiki/Dirichlet_distribution#Random_number_generation).

Второй способ хоть и менее очевиден, но более быстрый. Заключается он в том, чтобы выбирать величины из n гамма-распределения, затем нормализовать их делением на общее количество.

Здесь код:

```
# class Dirichlet

    def Random(self):
        p = numpy.random.gamma(self.params)
        return p / p.sum()
```

Теперь мы можем посмотреть на некоторые результаты.

Здесь код, который извлекает апостериорное распределение `n`:

```
    def DistOfN(self):
        pmf = thinkbayes.Pmf()
        for hypo, prob in self.Items():
            pmf.Set(hypo.n, prob)
        return pmf
```

`DistOfN` осуществляет итерацию на верхнем уровне гипотез и аккумулирует вероятность каждого `n`.

На рис. 15.2 показан результат. Наиболее вероятной величиной является 4. Величины от 3 до 7 вполне вероятны, а после этого вероятности быстро убывают. Вероятность того, что присутствует 29 разновидностей, настолько мала, что близка к малосущественной. Если бы мы выбрали более высокую границу, то пришли бы к такому же результату.

Надо помнить, этот результат базируется на равномерном приоре для `n`. Если мы имеем предварительную информацию о числе разновидностей в данной среде, мы могли бы выбрать другой приор.

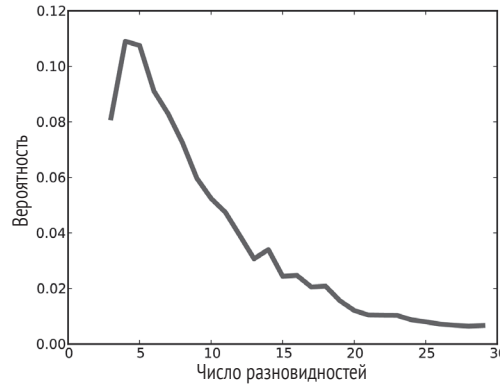


Рис. 15.2 ❖ Апостериорное распределение `n`

Оптимизация

Должен признаться, что горжусь этим примером. Проблема невидимых разновидностей непростая. И я думаю, что такое решение этой проблемы несложное, понятное и требует удивительно мало строк кода (около 50).

Единственный недостаток – в том, что это решение медленное. Оно вполне подходит, когда в нашем примере только 3 разновидности. Но когда в некоторых выборках количество данных пупка более чем 100 разнообразий, данное решение не слишком хорошее.

Следующие несколько разделов представляют собой серию оптимизаций, которые, чтобы решить проблему размерности, необходимо провести.

Прежде чем мы углубимся в детали, вот дорожная карта:

- сначала следует осознать, что распределения Дирихле мы обновляем теми же данными. Поэтому первые `m` параметров одинаковы для всех. Единственная разница – в числе гипотетических ненаблюдаемых разнообразий. Поэтому мы в действительности не нуждаемся в `n` Дирихле-объектах. Мы можем запомнить параметры в верхнем уровне иерархии. `Species2` реализует эту оптимизацию;
- `Species2` для всех гипотез использует одно и то же множество случайных величин. Это экономит время их генерации и имеет еще одно, даже более важное преимущество: предоставляя всем данным гипотезам одинаковый выбор из выборочного пространства, мы проводим сравнение между гипотезами более качественно, и, следовательно, оно требует меньше итераций для сходимости;
- даже при этих изменениях существует главная проблема реализации. С увеличением количества наблюдаемых разнообразий массив случайных распространений становится больше. Поэтому шанс выбора одного, приблизительно правильного, становится небольшим. В этом случае очень большое количество итераций создает маленькие правдоподо-

бия, вклад которых в общую сумму мал и не приводит к различию между гипотезами. Поэтому разнообразия следует обновлять по одному. По очереди, одно за другим.
Species4 – простая реализация этой стратегии использования Дирихле-объектов для представления субгипотез;

- наконец, Species5 объединяет субгипотезы на верхнем уровне и использует операции с массивом numpy, улучшая скоростные параметры.

Если вам детали не интересны, пропустите текст до раздела «Данные пупка», который находится на стр. 170, где мы увидим результаты относительно данных пупка.

Сворачивание иерархии

Все распределения Дирихле нижнего уровня обновляются одними и теми же данными. Поэтому параметры m для всех одинаковые. Мы можем исключить их и передать параметры на верхний уровень.

Species2 реализует эту операцию:

```
class (object):

    def __init__(self, ns):
        self.ns = ns
        self.probs = numpy.ones(len(ns), dtype=numpy.double)
        self.params = numpy.ones(self.high, dtype=numpy.int)
```

Здесь ns – список гипотетических величин для n; probs – список соответствующих вероятностей; params – последовательность параметров Дирихле, начальное значение всех равно 1.

Species2.Update обновляет оба уровня иерархии: сначала вероятность для каждой величины n, а затем параметры Дирихле:

```
# class Species2

    def Update(self, data):
        like = numpy.zeros(len(self.ns), dtype=numpy.double)
        for i in range(1000):
            like += self.SampleLikelihood(data)

        self.probs *= like
        self.probs /= self.probs.sum()

        m = len(data)
        self.params[:m] += data
```

SampleLikelihood возвращает массив правдоподобий, одно для каждой величины n; like аккумулирует общее для 1000 выборок правдоподобие; self.probs умножается на общее правдоподобие и затем нормализуется.

Последние две строки, которые обновляют параметры, такие же, как в Dirichlet.Update.

Теперь посмотрим на SampleLikelihood. Здесь есть две возможности оптимизации:

- когда гипотетическое число разнообразий n превышает число наблюдаемых разнообразий m, нам необходимы только первые m элементов многономинального PMF. Остальные равны 1;
- если число разнообразий велико, для плавающей запятой правдоподобие данных может быть слишком маленьким (см. раздел «Потеря значимости» на стр. 113). Поэтому надежнее вычислять логарифмические правдоподобия.

Снова многономинальное PMF есть

$$c_x p_1^{x_1} \ldots p_n^{x_n}.$$

Поэтому логарифмическое правдоподобие:

$$\log c_x + x_1 \log p_1 + \ldots + x_n \log p_n,$$

которое вычисляется быстро и просто. Снова c_x – то же для всех гипотез. Поэтому мы можем его исключить. Здесь код:

```
# class Species2

    def SampleLikelihood(self, data):
        gammas = numpy.random.gamma(self.params)
        m = len(data)
        row = gammas[:m]
        col = numpy.cumsum(gammas)

        log_likes = []
        for n in self.ns:
            ps = row / col[n-1]
            terms = data * numpy.log(ps)
            log_like = terms.sum()
            log_likes.append(log_like)

        log_likes -= numpy.max(log_likes)
        likes = numpy.exp(log_likes)

        coefs = [thinkbayes.BinomialCoef(n, m) for n in self.ns]
        likes *= coefs

        return likes
```

gammas – массив значений из гамма-распределения. Его длина равна наибольшей гипотетической величине n. row – просто первые m элементов массива gammas. Поскольку это только те элементы, которые зависят от данных, нам нужны только они.

Для каждого значения n нам надо разделить row на общую сумму первых n величин из gamma. cumsum вычисляет эти кумулятивные суммы и запоминает их в col.

Цикл осуществляет итерацию по величинам `n` и аккумулирует список логарифмических правдоподобий.

Внутри цикла `ps` сохраняет вероятности, нормализованные соответствующей кумулятивной суммой.

`terms` содержит элементы суммирования $x_i \log p_i$ и `log_likes` и их сумму.

По окончании цикла мы хотим преобразовать логарифмические правдоподобия в линейные правдоподобия. Но сместить их так, чтобы наибольшее логарифмическое правдоподобие было равно 0, не плохая идея. Это позволяет увеличить линейные правдоподобия (см. раздел «Потеря значимости» на стр. 113).

Наконец, перед тем как возвращать правдоподобие, следует применить корректирующий коэффициент, являющийся числом способов, с помощью которых мы могли бы наблюдать эти `m` разнообразий, если бы общее число разнообразий было равно `n`.

`BinomialCoefficient` вычисляет «n выбирает m», которое записывается как $\binom{n}{m}$.

Как часто бывает, оптимизированная версия менее читабельна и более склонна к ошибкам, чем оригинальная версия. Но есть одна причина, почему, по моему мнению, следует начинать с простой версии. Мы можем использовать оригинальную версию для регрессионного тестирования. Результаты были отображены для обеих версий. И то, что они приблизительно равны, подтверждается. А если число итераций увеличивается, результаты сходятся.

Еще одна проблема

Мы могли бы сделать еще больше для оптимизации этого кода. Но есть проблема, с которой нам сначала надо справиться. Когда число наблюдаемых разнообразий увеличивается, эта версия требует больше итераций для хорошей сходимости.

Проблема заключается в том, если мы выбираем разновидности из распределения Дирихле, то `ps` – неправильное, даже в приближении. Правдоподобие наблюдаемых данных близко к нулю и почти одинаково плохое для всех величин `n`. Поэтому большинство итераций не обеспечивает каких-либо полезных вкладов в общее правдоподобие. И когда число наблюдаемых разновидностей `m` становится большим, вероятность выбора `ps` с непренебрежимо малым правдоподобием становится очень маленькой.

К счастью, решение этой проблемы есть. Вспомним, при наблюдении множества данных вы можете априорное распределение всем этим множеством или обновить, или разбить его на серии обновлений с подмножеством данных. Результат все равно будет тем же.

Ключом для этого примера будет последовательное обновление одного разнообразия за другим. Таким способом мы генерируем случайное множество `ps`, только одно из которых влияет на вычисление правдоподобия. Поэтому шанс выбора одного хорошего множества значительно выше.

Здесь новая версия последовательного обновления разнообразий:

```
class Species4(Species):

    def Update(self, data):
        m = len(data)

        for i in range(m):
            one = numpy.zeros(i+1)
            one[i] = data[i]
            Species.Update(self, one)
```

Эта версия наследует __init__ из Species, поэтому представляет гипотезы как список Дирихле-объектов (в отличие от Species2).

Update осуществляет цикл наблюдаемых разновидностей и создает массив. Один массив со всеми нулями, и один – с подсчитанными разнообразиями. Затем в родительский класс вызывается Update и обновляет субгипотезы.

Таким образом, в данном примере мы имеем три обновления. Первое похоже на: «Я видел трех львов». Второе – «Я видел двух тигров и не видел новых львов». И третье – «Я видел одного медведя и не видел новых львов и тигров».

Здесь новая версия правдоподобия:

```
# class Species4

    def Likelihood(self, data, hypo):
        dirichlet = hypo
        like = 0
        for i in range(self.iterations):
            like += dirichlet.Likelihood(data)

        # correct for the number of unseen species the new one
        # could have been
        m = len(data)
        num_unseen = dirichlet.n - m + 1
        like *= num_unseen

        return like
```

Это почти то же самое, что и Species.Likelihood. Разница – это коэффициент num_unseen. Эта коррекция необходима, так как каждый раз, когда мы впервые видим некое разнообразие, мы должны учитывать, что было еще несколько невидимых разнообразий, которые мы могли видеть. Для больших значений n имеется больше разнообразий, которые мы могли видеть. А это увеличивает правдоподобие данных.

Это тонкое место. И я должен признать, что поначалу его правильно не оценил. Но после сравнения с прежней версией я в этой версии утвердился снова.

Мы сделали еще не все

Реализация последовательной процедуры обновления разнообразий решила одну проблему, но породила другую. Каждое обновление занимает время,

пропорциональное km, где k равно числу гипотез и m – числу наблюдаемых разнообразий. Если мы делаем m обновлений, то требуемое время пропорционально km^2.

Но процедуру можно ускорить. Для этого используем тот же прием, который был использован на стр. 165 в разделе «Сворачивание иерархии». Мы избавились от Дирихле-объектов, два уровня свернули в единственный объект.

Итак, вот еще одна версия `Species`:

```
class Species5(Species2):

    def Update(self, data):
        m = len(data)
        for i in range(m):
            self.UpdateOne(i+1, data[i])
            self.params[i] += data[i]
```

Эта версия наследует `__init__` из `Species2`. Поэтому она использует `ns` и `probs`, чтобы представить распределение `n`, и `params`, чтобы представить параметры распределения Дирихле.

Обновление подобно тому, какое мы видели в предыдущем разделе. Оно осуществляет цикл наблюдаемых разнообразий и вызывает `UpdateOne`:

```
# class Species5

    def UpdateOne(self, i, count):
        likes = numpy.zeros(len(self.ns), dtype=numpy.double)

        for i in range(self.iterations):
            likes += self.SampleLikelihood(i, count)

        unseen_species = [n-i+1 for n in self.ns]
        likes *= unseen_species

        self.probs *= likes
        self.probs /= self.probs.sum()
```

Эта функция похожа на `Species2.Update` с двумя изменениями:

- отличается интерфейс. Вместо всего множества данных мы получаем i, индекс наблюдаемых разнообразий и подсчет количества увиденных разнообразий;
- мы должны ввести корректирующий коэффициент к количеству ненаблюдаемых разнообразий, как делали это в `Species.Likelihood`. Разница здесь в том, что мы обновляем все правдоподобия сразу с массивом множителей.

Наконец, здесь `SampleLikelihood`:

```
# class Species5

    def SampleLikelihood(self, i, count):
        gammas = numpy.random.gamma(self.params)

        sums = numpy.cumsum(gammas)[self.ns[0]-1:]
```

```
        ps = gammas[i-1] / sums
        log_likes = numpy.log(ps) * count

        log_likes -= numpy.max(log_likes)
        likes = numpy.exp(log_likes)

        return likes
```

Это похоже на `Species2.SampleLikelihood`. Разница в том, что каждое обновление включает только отдельные разнообразия. Поэтому в цикле нет необходимости.

Время обработки этой функции пропорционально числу гипотез *k*. Она запускается `m` раз, и время обработки обновлений пропорционально *km*. А необходимое нам для точного результата число итераций обычно невелико.

Данные пупка

Достаточно о львах, тиграх и медведях. Вернемся к пупкам. Чтобы почувствовать, как выглядят данные, рассмотрим данные субъекта под номером B1242, чья выборка из 400 считываний в следующих количествах содержит 61 разнообразие:

```
92, 53, 47, 38, 15, 14, 12, 10, 8, 7, 7, 5, 5,
4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

Здесь большую часть от всего количества составляет несколько преобладающих разнообразий. И существует много разнообразий, которые считывались только один раз. Количество таких «одиночек» с большой вероятностью предполагает, что, по крайней мере, существует несколько ненаблюдаемых разнообразий.

В примере со львами и тиграми мы предполагали, что увидеть в резервации каждое из животных одинаково вероятно. Мы так же будем полагать и для данных пупка: наблюдать каждое разнообразие бактерий одинаково вероятно.

В реальности каждый шаг в процессе сбора данных мог внести смещение. Некоторые разнообразия бактерий более вероятно могут попасть на мазок. И нам следует запомнить этот возможный источник ошибки.

Разнообразие бактерий недостаточно изучено. Чтобы быть более точным, в дальнейшем будет использован термин «оперативная таксономическая единица» OTU (operational taxonomic unit).

Давайте теперь обработаем некоторые данные с пупка. Чтобы представить информацию о каждом изучаемом субъекте, был определен класс, названный `Subject`:

```
class Subject(object):

    def __init__(self, code):
        self.code = code
        self.species = []
```

Каждый субъект имеет строковый код и список. Строковый код подобен, например, коду «B1242». Список (число, имя) пар отсортирован в порядке увеличения чисел. `Subject`, чтобы сделать более простым доступ к этим числам и именам, обеспечивает несколько методов. Детали вы можете посмотреть здесь: http://thinkbayes.com/species.py. Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

`Subject` обеспечивает метод, называемый `Process`. Этот метод создает и обновляет suite `Species5`, представляющий распределения `n` и разнообразия.

`Suite2` обеспечивает `Dist0fN`, возвращающее апостериорное распределение `n`.

```
# class Suite2

    def DistN(self):
        items = zip(self.ns, self.probs)
        pmf = thinkbayes.MakePmfFromItems(items)
        return pmf
```

На рис. 15.3 показано распределение `n` для субъекта B1242. Вероятность того, что имеется ровно 61 разнообразие и отсутствуют ненаблюдаемые разнообразия, около 0. С 90%-ным интервалом доверия – от 66 до 79, – наиболее вероятная величина равна 72. Вероятность, что на верхней имеется более 87 разнообразий, почти равна нулю.

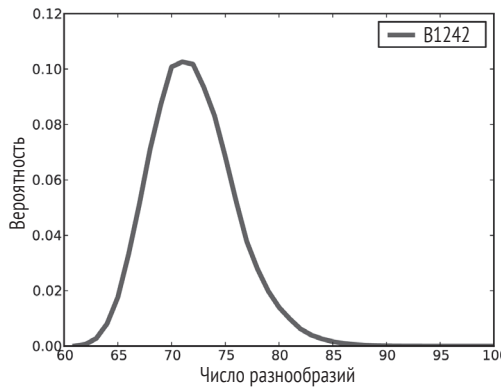


Рис. 15.3 ❖ Распределение `n` для субъекта B1242

Вычислим теперь апостериорное распределение для каждого разнообразия. `Species2` обеспечивает `DistOfPrevalence`:

```
# class Species2

    def (self, index):
        metapmf = thinkbayes.Pmf()

        for n, prob in zip(self.ns, self.probs):
            beta = self.MarginalBeta(n, index)
```

```
            pmf = beta.MakePmf()
            metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return metapmf, mix
```

`index` показывает, какие разнообразия мы хотим. Для каждого `n` имеем различные апостериорные распределения разнообразий.

Цикл обеспечивает итерацию возможных величин `n` и их вероятности. Для каждой величины `n` мы получаем бета-объект, представляющий собой маргинальное распределение указанных разнообразий. Напомню, что бета-объекты содержат параметры `alpha` и `beta`. Они, по сравнению с Pmf, не имеют величин и вероятностей, но обеспечивают `MakePmf`, генерирующий дискретные аппроксимации для непрерывного бета-распределения.

`metapmf` – это мета-Pmf, содержащий распределения разнообразий, обусловленных `n`.

`MakeMixture` организует мета-Pmf в `mix`, объединяющий обусловленные распределения в единое распределение разнообразий.

На рис. 15.4 показан результат для пяти разнообразий с наибольшим числом считываний. Большинство превалирующих разнообразий насчитывается для 23% из 400 считываний. Но, поскольку наверняка существует определенное число несчитанных разнообразий, более вероятна оценка 20% разнообразий, а с 90%-ным интервалом доверия – от 17% до 23%.

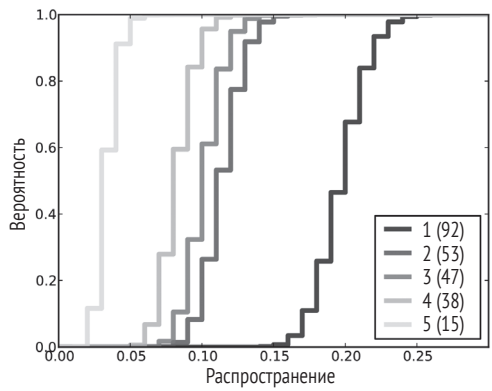


Рис. 15.4 ❖ Распределение распространения бактерий для субъекта B1442

Прогнозирующее распределение

Проблема невидимых разновидностей была представлена четырьмя вопросами. Мы ответили на первые два, вычислив апостериорные распределения для `n` и распространение каждой разновидности.

Два других вопроса были такими:

- если мы планируем собрать дополнительное количество образцов, можем ли мы предсказать, сколько новых разновидностей мы, скорее всего, обнаружим?
- сколько необходимо дополнительных прочитанных фрагментов, чтобы увеличить часть наблюдаемых разновидностей до заданного порога?

Чтобы ответить на вопросы о предсказаниях, подобных этим, смоделировать возможные в будущем события и вычислить прогнозирующие распределения числа разнообразий и часть разнообразий из общего числа, которые, вероятно, увидим, используем апостериорные распределения.

Ядро такого моделирования выглядит следующим образом:

1) выберем `n` апостериорных распределений;
2) используя распределение Дирихле, выберем распространение для каждого разнообразия, включая невидимые разновидности;
3) сгенерируем случайную последовательность будущих наблюдений;
4) вычислим как функцию числа дополнительных считываний `k` число новых разновидностей `num_new`;
5) повторим предыдущие шаги и саккумулируем совместное распределение `num_new` и `k`.

Здесь приведен код. `RunSimulation` запускает одно моделирование:

```
# class Subject

    def RunSimulation(self, num_reads):
        m, seen = self.GetSeenSpecies()
        n, observations = self.GenerateObservations(num_reads)

        curve = []
        for k, obs in enumerate(observations):
            seen.add(obs)

            num_new = len(seen) - m
            curve.append((k+1, num_new))

        return curve
```

Здесь `num_reads` – число дополнительных считываний для моделирования; `m` – число наблюдаемых разнообразий; `n` – случайные величины из апостериорного распределения; `observations` – случайная последовательность имен разнообразий.

Каждый раз в процессе цикла мы добавляем новое наблюдение к `seen`, записываем число считываний и число новых разнообразий. Результатом является **кривая разрежения**, представленная в виде списка пар с числами считываний и новых разнообразий.

Прежде чем мы увидим результат, давайте посмотрим на `GetSeenSpecies` и `GenerateObservations`:

```
#class Subject

    def GetSeenSpecies(self):
        names = self.GetNames()
        m = len(names)
        seen = set(SpeciesGenerator(names, m))
        return m, seen
```

GetNames возвращает список имен разнообразий, которые появляются в файле данных. Но многие субъекты этих имен не являются уникальными. Поэтому, чтобы расширить каждое имя серийным номером, был использован GenerateObservations:

```
    def SpeciesGenerator(names, num):
        i = 0
        for name in names:
            yield '%s-%d' % (name, i)
            i += 1

        while i < num:
            yield 'unseen-%d' % i
            i += 1
```

Давая имя типа Corybacterium, SpeciesGenerator создает Corybacterium-1. Когда список имен заканчивается, он создает имена типа unseen-62.

Здесь GenerateObservations:

```
# class Subject

    def GenerateObservations(self, num_reads):
        n, prevalences = self.suite.SamplePosterior()

        names = self.GetNames()
        name_iter = SpeciesGenerator(names, n)

        d = dict(zip(name_iter, prevalences))
        cdf = thinkbayes.MakeCdfFromDict(d)
        observations = cdf.Sample(num_reads)
        return n, observations
```

Вновь num_reads – число дополнительных считываний для моделирования; n и prevalences – это выборки из апостериорного распределения.

cdf – это объект Cdf, который отображает в кумулятивные вероятности имена с учетом ненаблюдаемых. Использование Cdf делает их эффективными для генерирования случайной последовательности имен разнообразий.

Наконец, здесь Species2.SamplePosterior:

```
    def SamplePosterior(self):
        pmf = self.DistOfN()
        n = pmf.Random()
        prevalences = self.SamplePrevalences(n)
        return n, prevalences
```

И SamplePrevalences, который генерирует выборку разнообразий, обусловленных n:

```
# class Species2

    def SamplePrevalences(self, n):
        params = self.params[:n]
        gammas = numpy.random.gamma(params)
        gammas /= gammas.sum()
        return gammas
```

В разделе «Случайная выборка» на стр. 163 мы видели этот алгоритм для генерирования случайных величин из распределения Дирихле.

На рис. 15.5 показаны 100 смоделированных кривых разрежения для субъекта B1242. Кривые «флюктуирующие». То есть, чтобы кривые не перекрывались, каждая кривая сдвинута случайным образом. Исследовав их, мы можем дать оценку: после 400 новых считываний мы, вероятно, найдем от 2 до 6 новых разнообразий.

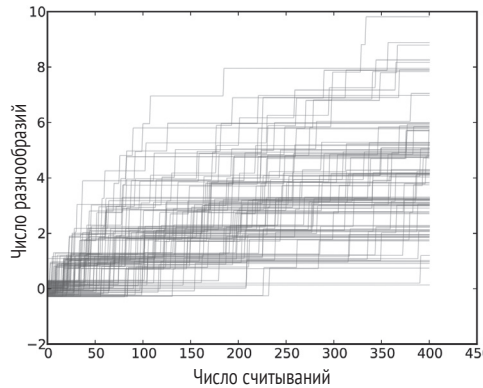


Рис. 15.5 ❖ Моделирование кривых разрежения для субъекта B1242

Совместный постериор

Мы, чтобы оценить распределение num_new и k и, таким образом, получить распределение num_new, обусловленное любой величиной k, можем использовать процедуры моделирования.

```
def MakeJointPredictive(curves):
    joint = thinkbayes.Joint()
    for curve in curves:
        for k, num_new in curve:
            joint.Incr((k, num_new))
    joint.Normalize()
    return joint
```

`MakeJointPredictive` создает объект Joint, являющийся Pmf, величины которого являются кортежами; `curves` – список разреженных кривых, созданный `RunSimulation`. Каждая кривая содержит список пар `k` и `num_new`.

Результирующее совместное распределение отображает каждую пару в вероятность их появления. Имея совместное распределение, мы можем использовать их для получения распределения `num_new`, обусловленное `k` (см. раздел «Условные распределения» на стр. 104).

`MakeConditionals` берет список `ks` и вычисляет условное распределение `num_new` для каждого `k`. Результат – список объектов Cdf.

```
def MakeConditionals(curves, ks):
    joint = MakeJointPredictive(curves)

    cdfs = []
    for k in ks:
        pmf = joint.Conditional(1, 0, k)
        pmf.name = 'k=%d' % k
        cdf = pmf.MakeCdf()
        cdfs.append(cdf)

    return cdfs
```

На рис. 15.6 показан результат. После 100 считываний среднее предсказанное число разнообразий равно 2. При 90%-ном интервале доверия – от 0 до 5. После 800 считываний мы ожидаем увидеть от 3 до 12 новых разнообразий.

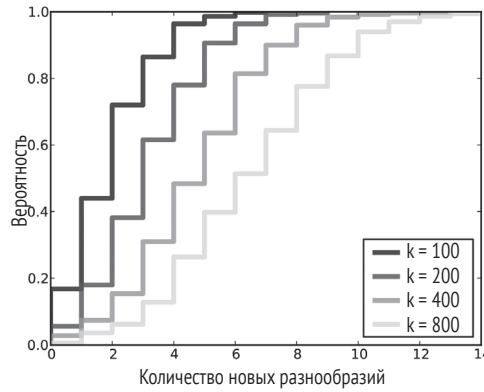


Рис. 15.6 ❖ Распределение числа новых разнообразий при условии числа дополнительных считываний

Перекрывающая зона

Последний вопрос, на который мы хотим получить ответ, звучит так: «Сколько дополнительных прочитанных фрагментов необходимо, чтобы увеличить часть наблюдаемых разновидностей до заданного порога?»

Чтобы ответить на этот вопрос, нам необходима версия RunSimulation, вычисляющая эту часть наблюдаемых разновидностей, а не количество новых разновидностей.

```
# class Subject

    def RunSimulation(self, num_reads):
        m, seen = self.GetSeenSpecies()
        n, observations = self.GenerateObservations(num_reads)

        curve = []
        for k, obs in enumerate(observations):
            seen.add(obs)

            frac_seen = len(seen) / float(n)
            curve.append((k+1, frac_seen))

        return curve
```

Далее мы запускаем цикл для каждой кривой и создаем словарь d, отображающий в список fracs количество дополнительных считываний k. То есть список величин для перекрытия, полученных после k считываний.

```
    def MakeFracCdfs(self, curves):
        d = {}
        for curve in curves:
            for k, frac in curve:
            d.setdefault(k, []).append(frac)

        cdfs = {}
        for k, fracs in d.iteritems():
            cdf = thinkbayes.MakeCdfFromList(fracs)
            cdfs[k] = cdf

        return cdfs
```

Затем для каждого k мы создаем Cdf fracs. Этот Cdf представляет распределение перекрытия после k считываний.

Вспомним, что этот Cdf говорит нам о вероятности, которая ниже заданного порога. Поэтому *дополнительный* Cdf говорит нам о вероятности, превышающей его. На рис. 15.7 показаны дополнительные Cdf для диапазона величин k.

Чтобы получить эту цифру, выберите желаемый уровень перекрытия вдоль оси *x*. Для примера выберите 90%.

Теперь вы можете найти на графике вероятность достижения 90%-го перекрытия после k считываний. Для примера при 200 считываниях вы имеете около 40% шансов получить 90%-ное перекрытие. При 1000 считываниях вы имеете 90% шансов получить 90%-ное перекрытие.

Таким образом, мы ответили на все четыре вопроса, которые составляют проблему ненаблюдаемых разновидностей. Чтобы подкрепить алгоритмы этой главы реальными данными, следовало бы добавить деталей. Но эта глава и так слишком длинная. Поэтому дополнительные детали мы обсуждать не будем.

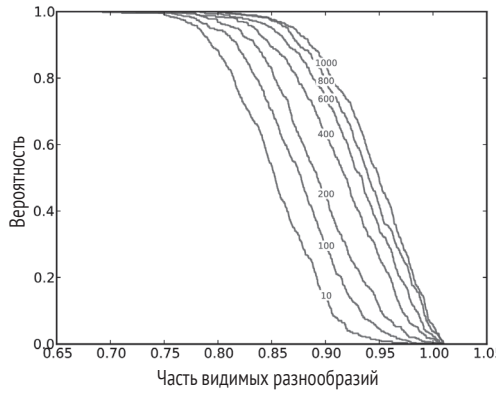


Рис. 15.7 ❖ Дополнительные Cdf-перекрытия для диапазона дополнительных считываний

Об этих проблемах и о моих к ним обращениях вы можете прочитать на http://allendowney.blogspot.com/2013/05/belly-button-biodiversity-end-game.html.

Коды для этой главы вы можете загрузить из http://thinkbayes.com/species.py.

Для получения большей информации посмотрите раздел «Работа с кодом» на стр. 11.

Обсуждение

Проблема невидимых разновидностей – область активных исследований. Поэтому я полагаю, что алгоритмы в этой главе – новый вклад в решение данной проблемы. В тексте объемом менее 200 страниц вопрос был изложен от вероятностных основ до передового рубежа исследований. Меня это очень радует.

Цель этой книги – представить три взаимосвязанные идеи:

- **Байесовское мышление.** Основой байесовского анализа является идея использовать для представления неопределенности доверий вероятностные распределения. Для этого потребуются данные для обновления этих распределений и результат для предсказаний и унифицированных решений.
- **Вычислительные методы.** Предпосылкой к этой книге служит то, что при использовании вычислений, а не математических абстракций, понимание байесовского анализа становится проще. Применять байесовские методы проще, многократно используя строительные блоки, которые могут быть быстро реконфигурированы для решения реальных проблем.
- **Итеративное моделирование.** Многие реальные проблемы требуют модельных решений и компромисса между реальностью и сложностью. Часто невозможно заранее знать, какой фактор следует включить в модель, а от какого фактора можно отказаться. Лучшее решение –

применить итерацию. Начинать с простых моделей и, постепенно эту модель усложняя, использовать ее для проверки достоверности других моделей.

Эти идеи универсальны и действенны. Они применимы для решения проблем как во всех областях науки, так и инженерии. От простых задач до новейших современных исследований.

Если вы проводите такие исследования, то вам следует использовать эти идеи применительно к вашим проблемам. Я надеюсь, вы найдете эти идеи полезными. Дайте мне знать, если это так!

Предметный указатель

A
ABC, 116

C
CV, 110

K
KDE, 64

P
PMF, 65

S
SAT, 129

А
Абстрактный класс, 31, 65
Апостериорное распределение, 27
Аппроксимация, 117
Априорное распределение, 26, 113

Б
Байесовская структура, 27
Байесовский коэффициент, 54
Бета-распределение, 48, 159
Биноминальная функция распределения, 48
Биологическое разнообразие, 167

В
Вероятностная функция, 25
Вероятностная функция масс, 25

Г
Ги Стив, 64
Гипотеза, 20, 33, 109

Д
Джейнс Е. Т., 152
Джеффриз Гарольд, 127
Диахроническая интерпретация, 19
ДНК, 158
Дэвидсон-Пилон Камерон, 64

З
Значимость, 113

И
Изменчивость, 110
Интер-квартиль, 119
Интер-процентиль, 119
Итеративное моделирование, 178
Итерация гипотезы, 47

К
Кемпбелл-Рикеттс Том, 152
Кеш, кеширование, 117
Комплект, 20
Конкретный класс (тип), 31, 66
Коэффициент изменчивости, 110, 113
Кумулятивная функция распределения, 39, 58

Л
Логарифмическое преобразование, 114

М
Маккей Дэвид, 13, 43, 54, 124
Маргинальное распределение, 103, 107, 160
Мета-PMF, 79, 135, 157
Мета-Suite, 154
Мешающие параметры, 139
Многономинальное PDF, 166
Моделирование, 41, 82, 173
Модельная ошибка, 133
Монти Холл, 22
Мостеллер Фредерик, 34

Н
Невидимые разновидности, 158
Нормализующая константа, 20

Нормальное распределение, 65

О

Обратная задача, 152
Оптимизация программного кода, 47
Остин Кай, 85
Отношение вероятностей, 52
Оценка изменчивости, 82
Оценка максимального правдоподобия, 45
Ошибки моделирования, 147

П

Погрешность наблюдения, 87
Подземелье и драконы, игра, 33, 56
Последовательная корреляция, 147
Постериор, 20, 77
Правдоподобие, 20
Правило Кромвелля, 50
Предсказание, 138
Процесс Бернулли, 76
Процесс Пуассона, 76
Прямая задача, 152

Р

Распределение Дирихле, 159
Риттер Брендан, 85

С

Сивиа Д. С., 100
Скорость атаки, 101
Смесь, 60
Совместное распределение, 107
Сопряженный приор, 48
Средний постериор, 38
Степенной закон, 37
Структура программного пакета, 29

Т

Треугольный приор, 46, 127

У

Унифицированное (равномерное) распределение, 49
Условная вероятность, 16, 104
Условное распределение, 104, 107

Ф

Форум Реддит, 51
Функция вероятностной меры, 65
Функция плотности вероятности, 65
Функция правдоподобия, 102

Х

Хейер Андреас, 77
Хитроумные теории, 105
Хоаг Дирк, 82
Хорсфорд Эбен Нортон, 109

Ч

Частотность слов, 26

Я

Ядро оценки плотности, 67

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**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Аллен Б. Дауни

Байесовские модели

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Яроцкий В. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 16,6875. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**