Российская академия наук
Санкт-Петербургский институт информатики и автоматизации РАН

С. И. Николенко, А. Л. Тулупьев

САМООБУЧАЮЩИЕСЯ СИСТЕМЫ

Москва
Издательство МЦНМО
2009

УДК 004.85
ББК 22.18
Н63

Рецензенты:

В. Б. Невзоров, д-р физ.-мат. наук, проф. кафедры теории вероятностей и математической статистики математико-механического факультета СПбГУ,

Е. О. Степанов, д-р физ.-мат. наук, зав. кафедрой математического моделирования естественнонаучного факультета СПбГУ ИТМО

Николенко С. И., Тулупьев А. Л.
Н63 Самообучающиеся системы. — М.: МЦНМО, 2009. — 288 с.: 24 илл.

ISBN 978-5-94057-506-1

Книга посвящена одной из самых практически применимых, активных и быстроразвивающихся областей современной информатики, объединяющей множество методов из различных областей математики и не только математики — машинному обучению. В книге обсуждаются основы многих базовых аппаратов машинного обучения: деревья принятия решений, искусственные нейронные сети, генетические алгоритмы, байесовские классификаторы, алгоритмы кластеризации и обучение с подкреплением. Изложение ведётся увлекательным языком, книгу интересно читать, и она доступна даже не очень подготовленному читателю. Однако при этом сохраняется математическая строгость, а наиболее сложные части изложения заинтересуют и профессионалов. Книга снабжена обширной аннотированной библиографией.

Читать книгу смогут даже старшеклассники, хотя она будет представлять несомненный профессиональный интерес и для студентов всех курсов, изучающих математику и информатику, а также для специалистов и аспирантов, ведущих исследования в соответствующих областях. В этом отношении значительная часть материала монографии сможет сыграть роль углублённого учебного пособия.

ББК 22.18

ISBN 978-5-94057-506-1

Оглавление

Благодарности 7

Введение 11
Об искусственном интеллекте 12
Машинное обучение и интеллектуальные агенты 17
На чём стоит искусственный интеллект 19

Глава 1. Деревья принятия решений 25
§ 1.1. Введение 26
§ 1.2. Структура дерева принятия решений 27
§ 1.3. Энтропия и прирост информации 30
§ 1.4. Деревья принятия решений и булевские функции 37
§ 1.5. Алгоритм ID3 39
§ 1.6. Реализация ID3 на языках Python и Ruby 39
§ 1.7. Проблема критерия прироста информации 46
§ 1.8. Оверфиттинг 48
§ 1.9. Заключение 50

Глава 2. Обучение концептам 53
§ 2.1. Введение 54
§ 2.2. Частичные порядки 55
§ 2.3. Лирическое отступление: функция Мёбиуса 60
§ 2.4. Алгоритм Find-S 68
§ 2.5. Реализация алгоритма Find-S 71
§ 2.6. Алгоритм исключения кандидатов 75
§ 2.7. Заключение 79

Глава 3. Нейронные сети 81
§ 3.1. Введение 83
§ 3.2. Перцептрон 85
§ 3.3. Обучение перцептрона 89

§ 3.4. Обучение перцептрона на практике 94
§ 3.5. Метод градиентного спуска 99
§ 3.6. Нелинейные перцептроны. Сигмоид. 104
§ 3.7. Алгоритм обратного распространения ошибки 106
§ 3.8. Реализация нейронной сети на Java 109
§ 3.9. Заключение 117

Глава 4. Генетические алгоритмы 119
§ 4.1. Введение 121
§ 4.2. Схема генетического алгоритма в деталях 122
§ 4.3. Генетические операции 124
§ 4.4. Представление данных 127
§ 4.5. Отбор 132
§ 4.6. Дарвин, Ламарк и Болдуин 134
§ 4.7. Генетические алгоритмы на деревьях 138
§ 4.8. Генетическое программирование 140
§ 4.9. Генетическое программирование на практике 145
§ 4.10. Язык программирования LISP 155
§ 4.11. Заключение 158

Глава 5. Байесовское обучение и классификаторы 161
§ 5.1. Введение 162
§ 5.2. Теорема Байеса 163
§ 5.3. Априорные и апостериорные вероятности 168
§ 5.4. Теорема Байеса, данные и гипотезы 170
§ 5.5. MAP и задачи классификации 173
§ 5.6. Оптимальный и гиббсовский классификаторы 175
§ 5.7. Наивный байесовский классификатор 178
§ 5.8. Атрибуция текстов 180
§ 5.9. Байесовское обучение и нейронные сети 189
§ 5.10. Принцип наименьшей длины описания 191
§ 5.11. Заключение 193

Глава 6. Алгоритмы кластеризации 195
§ 6.1. Введение 196
§ 6.2. Постановка задачи и виды кластеризации 199
§ 6.3. Иерархическая кластеризация 202
§ 6.4. Кластеризация методами теории графов 203
§ 6.5. Алгоритм EM 208

§ 6.6. Кластеризация при помощи EM 215
§ 6.7. Алгоритм k-средних 223
§ 6.8. Нечёткие алгоритмы кластеризации 230
§ 6.9. Заключение 232

Глава 7. Обучение с подкреплением 233
§ 7.1. Введение 234
§ 7.2. Как оценивать поведение агента? 236
§ 7.3. Многорукие бандиты 240
§ 7.4. Доказуемо оптимальные алгоритмы 241
§ 7.5. Другие стратегии 246
§ 7.6. Поиск стратегий в известной модели 248
§ 7.7. Поиск оптимальных стратегий без модели 250
§ 7.8. Поиск моделей и оптимальных стратегий по ним 256
§ 7.9. Игрушечный пример Q-обучения и Dyna 260
§ 7.10. Заключение 268

Литература 271

Благодарности

Благодарности от С. И. Николенко

Идея этой книги выросла из конспектов лекций вводного курса по машинному обучению. Я читал этот курс сначала в Академии Современного Программирования[1], а затем в Санкт-Петербургском государственном университете информационных технологий, механики и оптики (ИТМО), в котором и сейчас являюсь ассистентом кафедры компьютерных технологий факультета информационных технологий и программирования. Затем этот курс (уже в несколько расширенном виде) читался и в рамках «Computer Science Club»[2], и в Академическом физико-технологическом университете РАН (АФТУ РАН). Разработка и преподавание курса происходили при непосредственном участии и финансовой поддержке Антона Лиходедова, которому я благодарен в первую очередь.

Разумеется, всё это никогда бы не состоялось, если бы не мой научный руководитель в аспирантуре ПОМИ РАН Эдуард Алексеевич Гирш. Я благодарен ему и руководителю моего дипломного проекта Николаю Александровичу Вавилову — надеюсь, не будет нескромным сказать, что я считаю их своими учителями. Спасибо Александру Шеню, который прочёл рукопись и высказал много ценных замечаний; надеюсь, когда-нибудь у меня получится написать книгу, которую он назовёт хорошей.

В работе над книгой принимали деятельное участие студенты ИТМО; им принадлежит, например, подавляющее большинство примеров программного кода, рассыпанных по этой

[1] Бывшая «Академия Борланд», http://www.borland-academy.ru/.

[2] При Санкт-Петербургском отделении Математического Института им. В. А. Стеклова (ПОМИ РАН), http://logic.pdmi.ras.ru/~infclub/.

книге. Спасибо Ольге Большаковой, Сергею Вишнякову, Андрею Вокину, Евгению Кирпичёву, Илье Колыхматову, Дмитрию Кочелаеву, Ольге Комалёвой, Тимуру Магомедову, Илье Пименову, Роману Сатюкову, Фёдору Царёву, Антону Яковлеву... уверен, что многих забыл, и искренне прошу прощения. А за возможность работать со всеми этими замечательными людьми — спасибо декану факультета информационных технологий и программирования университета ИТМО Владимиру Глебовичу Парфёнову.

И, конечно, спасибо Вам, читатель, за то, что не поленились заглянуть даже в раздел «Благодарности».

Благодарности от А. Л. Тулупьева

Мой вклад в книгу был бы невозможен, если бы в свои студенческие годы мне не выпала удача слушать лекции и вообще учиться у замечательных преподавателей математико-механического факультета Санкт-Петербургского государственного университета: Николая Кирилловича Косовского, Виктора Григорьевича Кузьменко, Юрия Владимировича Матиясевича, Валерия Борисовича Невзорова, Анатолия Олесьевича Слисенко, Владимира Олеговича Сафонова. Всем им я бесконечно признателен.

Понимание того, что знания передаются с помощью паттернов — а значит, математические модели знаний с неопределённостью должны быть соответствующим образом структурированы, — пришло, когда мне удалось продолжить обучение на факультете социологии университета: с благодарностью вспоминаю, какие интереснейшие курсы по культуральной антропологии и этносоциологии вели Юрий Владимирович Емельянов и Николай Генрихович Скворцов.

Владимир Иванович Городецкий позволил мне присоединиться к его исследованиям в области искусственного интеллекта, познакомил с парадигмой алгебраических байесовских сетей и сформировал на много лет вперёд рациональную программу теоретических исследований в этой области. С исключительной теплотой вспоминаю те годы, когда Владимир Иванович руководил моей диссертационной работой.

Заметная часть материала этой книги сформировалась в результате обзорно-аналитической работы и научных исследований, которые проводились совместно с Анной Камоевной Абрамян, Андреем Владимировичем Лобацевичем, Леонидом Леонидовичем Налчаджи, Дмитрием Александровичем Никитиным, Александром Владимировичем Сироткиным. С радостью пользуюсь возможностью поблагодарить их и за сотрудничество, и за поддержку, и за терпение.

Антон Алексеевич Афанасьев, Максим Владимирович Ментюков, Сергей Сергеевич Синчук, Алексей Михайлович Левин, Дмитрий Михайлович Столяров, будучи студентами математико-механического факультета, принимали активнейшее участие в лекционных и практических занятиях, на которых обсуждались логико-вероятностные модели знаний с неопределённостью; их вклад существенно повлиял и на содержание соответствующих разделов книги и на форму его изложения.

Введение

Читатель, добравшийся по крайней мере до предисловия и недовольный им, заплатил за книгу деньги и хотел бы знать, какое ему будет возмещение. В этом случае моё последнее прибежище — напомнить ему, что он знает различные способы извлечь пользу из книги без того, чтобы и впрямь её прочитать. Она может, подобно многим другим, прикрыть зазор в его библиотеке, где, аккуратно переплетённая, она наверняка будет хорошо смотреться. Или же он может положить её на туалетный или чайный столик своей учёной подруги. Или же, наконец, он может её отрецензировать — это, без сомнения, лучший способ её употребить, и я особо его рекомендую.

Мир как воля и представление
Артур Шопенгауэр

В начале каждой главы её основные события будет предвосхищать Винни. Да-да, тот самый Винни-Пух, который машет вам на рисунке. Истории про Пуха искушённый читатель, конечно, может презрительно пропустить, но мы надеемся, что они будут интересными и достаточно тесно связанными с сутью происходящего. А сейчас Пух просто хочет поздороваться, ведь он действительно очень добрый медвежонок и искренне рад вас видеть.

Об искусственном интеллекте

Идея искусственного интеллекта занимала людей довольно давно. Гефест создавал из металла помощников для своей кузницы, а одного гигантского человекоподобного робота — Талоса — даже подарил Миносу для охраны Крита (по другой версии, Зевс подарил его Европе). Пигмалион создал искусственный интеллект из слоновой кости — правда, без Афродиты у него ничего бы не получилось.

В средние века подобные достижения в основном приписывались магии — например, Вильгельм Парижский писал, что Альберт Великий[1] изготовил голову, говорящую человеческим голосом. Впрочем, были и реальные попытки создать видимость искусственного интеллекта — знаменитый шахматный автомат «Турок» управлялся сидящим внутри карликом–шахматистом, но свою роль в популяризации идеи искусственного интеллекта сыграл. В иудейской традиции искусственный интеллект отражён в виде *големов* — сделанных из глины искусственных слуг, обладающих интеллектом; сделать себе голема могли только самые святые и мудрые. Впрочем, големы умели только понимать речь, но не издавать её: по мнению иудеев, если бы они умели говорить, это бы означало, что у них есть душа. Литературную часть исторического экскурса стоит, пожалуй, закончить доктором Виктором Франкенштейном — дальше идея искусственного интеллекта появляется слишком часто, чтобы подробно её здесь анализировать.

[1] Альберт Великий (Albertus Magnus, ок. 1200–1280) — монах–доминиканец, один из лучших учёных своего времени. Он первым попытался увязать Аристотеля и христианские каноны, развил доктрину свободной воли (на которую много и с удовольствием ссылался, например, Данте), оставил след в логике, ботанике, географии, астрономии (и астрологии, конечно же), минералогии, химии (в частности, ему приписывают открытие мышьяка), зоологии, физиологии, френологии, теории музыки... Разумеется, занятия наукой привели к тому, что Альберта Великого и при жизни, и в особенности после неё считали магом и волшебником. Несмотря на занятия алхимией — Альберт Великий не подтверждает, что нашёл философский камень, но пишет, что считали наблюдал изготовление золота трансмутацией — он был беатифицирован в 1622 году, а в 1931 канонизирован. Фома Аквинский был учеником Альберта Великого.

Первое полноценное философское обоснование мечты об искусственном интеллекте получили в рационалистических работах Рене Декарта[1]. Декарт считал, что в сознании мыслящих людей от рождения заложены теоретические идеи, из которых всевозможные знания получаются сугубо дедуктивным путём, от общего к частному; в этом и есть суть рационализма, в отличие от эмпиризма, который предпочитает идти индуктивно, от частного к общему. Отсюда уже полшага до искусственного интеллекта: дедукцию гораздо легче автоматизировать, чем индукцию (хотя и на этом пути есть свои проблемы), а дальше нужно будет только выделить и записать эти «основные теоретические идеи». Декарту вообще было свойственно пытаться механизировать человека и человеческие рассуждения: в известной цитате он сравнивает больного человека со сломанными часами, а одной из главных задач декартовой аналитической геометрии была попытка автоматизировать, упростить красивые индуктивные геометрические построения, которыми до той поры приходилось доказывать теоремы.

Поворотным моментом в истории искусственного интеллекта стало, разумеется, изобретение первых компьютеров. Собственно, история искусственного интеллекта как раздела информатики начинается с начала 1950-х гг. — в 1950 году Алан Тьюринг[2] предложил свой знаменитый тест, который считается

[1] Рене Декарт (René Descartes, 1596–1650) — французский учёный, философ, математик, писатель и многое-многое другое. Декарт родился во Франции, выучился на юриста, а затем поступил на военную службу в революционной Голландии, где впервые заинтересовался математикой. Именно в Голландии прошли наиболее плодотворные годы Декарта: с 1628 по 1649 он написал подавляющее большинство своих работ. А работы эти фактически заложили фундамент всей современной науки: в «Рассуждении о методе» Декарт, основываясь на скептическом методе, создаёт рационалистическую систему познания, основанную на дедуктивных рассуждениях, а также разрабатывает дуалистическую теорию, в которой «душа», мышление отделяются от тела и рассматриваются как внефизические процессы. В математике Декарт известен как автор аналитической геометрии; кроме того, именно он впервые применил анализ бесконечно малых к задаче о касательной, что позволило Ньютону и Лейбницу разработать основы математического анализа.

[2] Алан Тьюринг (Alan Turing, 1912–1954) — английский математик, логик и криптограф. Он во многом создал современную теоретическую информатику, формализовав понятие алгоритма, введя в рассмотрение машины Тьюринга и сформулировав общепринятую версию тезиса Чёрча (в

отправной точкой AI[1] как науки [138, 159]. Суть теста Тьюринга в том, что компьютер должен суметь успешно выдать себя за человека в (письменном) диалоге между судьёй-человеком и компьютером: судья одновременно ведёт два диалога, один с настоящим человеком, а другой с компьютером, и после этого должен суметь ответить, кто из них кто. Сам Тьюринг, впрочем, тут же указывал, что наоборот у нас, людей, уже никак не получится: человека от компьютера легко отличит любая достаточно сложная арифметическая задачка. Отметим здесь же, что в статье самого Тьюринга есть некоторые детали, которые обычно ускользают из современных формулировок. Например, Тьюринг предлагал сравнивать не компьютер, имитирующий человека, и настоящего человека, а компьютер, выдающий себя за женщину, и мужчину, выдающего себя за женщину; иначе говоря, сравнивать *имитирующий кого-то другого* компьютер с *имитирующим кого-то другого* человеком (подробнее о тесте Тьюринга см. [138]).

Пятидесятые–шестидесятые годы XX века были временем безудержного оптимизма. Казалось, что ещё чуть-чуть, ещё немного — и гигантские холлы, заставленные громоздкими ламповыми устройствами, напечатают на перфокартах: «Кто я?». Научная фантастика того времени (которое заслуженно считается «золотым веком» научной фантастики вообще) уже не просто мечтает об искусственном интеллекте, а считает его свершившимся фактом, делом недалёкого будущего. Мысль движется дальше: фантасты задаются вопросами о будущем сосуществовании человека и разумной машины. Всем известны три закона

англоязычной литературе называемого обычно Church–Turing thesis). Именно Тьюринг переформулировал теорему Гёделя о неполноте, применив её к гильбертовской «Entscheidungsproblem», то есть к задаче о том, можно ли алгоритмически искать доказательства математических утверждений. Кроме того, он был одним из отцов-основателей современного искусственного интеллекта; ему принадлежит формулировка «теста Тьюринга», который должна пройти машина, претендующая на право считаться «искусственным интеллектом»: суметь поддержать разговор (хотя бы в текстовом формате) с человеком на естественном языке так, чтобы человек не понял, машина или другой человек с ним разговаривает.

[1] AI — общепринятое сокращение от Artificial Intelligence, искусственный интеллект.

робототехники, сформулированные Айзеком Азимовым[1]; они, если вдуматься, предельно неформальны, понять их и принять к исполнению — тест гораздо сильнее тьюринговского.

Литературный и общественный оптимизм отразился и в науке. Пятидесятые годы — время расцвета математической логики; несмотря на уже доказанную и осознанную теорему Гёделя о неполноте, для многих ограниченных логик уже были известны разрешающие процедуры. Да, они бывали экспоненциальными, дважды экспоненциальными и даже более того, но общий вектор развития виделся ясным — автоматический логический вывод казался делом ближайшего будущего, а на этой основе можно было строить и ещё более амбициозные планы.

В 1956 году Джон Маккарти[2], Марвин Мински[3], Натаниэль Рочестер[4] и Клод Шеннон организовали знаменитую летнюю школу в Дартмуте, где мозговым штурмом попытались

[1] Айзек Азимов (Isaac Asimov, 1920–1992) — американский фантаст и популяризатор науки. Родился Азимов в деревне Петровичи Могилёвской губернии, ныне Смоленской области (звали его тогда Исаак Озимов, от слова «озимые», которыми торговал его прадед). Хотя Азимов главным образом известен как фантаст — его достижения в этой сфере мы здесь не будем и пытаться перечислять — он также написал множество научно-популярных книг, от «Краткой истории химии» до «Путеводителя Азимова по Шекспиру». Одной из главных тем творчества Азимова стало взаимодействие человека с мыслящими машинами, для которого он сформулировал свои знаменитые три закона.

[2] Джон Маккарти (John McCarthy, p. 1927) — американский информатик и когнитивист. Маккарти придумал термин «artificial intelligence», а также создал язык программирования Lisp, ставший для раннего AI основным [104]. Кроме того, Маккарти много работал в математической логике, в частности, над немонотонными логиками [105].

[3] Марвин Мински (Marvin Minsky, p. 1927) — американский исследователь в области искусственного интеллекта, один из основателей лаборатории AI в Массачусетском технологическом институте, автор классической книги о нейронных сетях «Перцептроны» [111]. Примечательно, что Мински консультировал Артура Кларка при написании и постановке его «Космической Одиссеи 2001» и даже весьма лестно там упоминается — см. эпиграф к главе 3; цитата, кстати, весьма характерная для тех оптимистических времён.

[4] Натаниэль Рочестер (Nathaniel Rochester, 1919–2001) — американский инженер и математик. Рочестер стоял у истоков не только искусственного интеллекта, но и других областей: в компании IBM в начале пятидесятых он создал первый компьютер компании IBM 701, а в шестидесятые годы занимался криогеникой.

заложить основы будущей теории искусственного интеллекта. В заявке на спонсирование этой летней школы Маккарти писал [106]:

> Мы предлагаем исследование искусственного интеллекта сроком на два месяца с участием десяти человек летом 1956 года в Дартмутском колледже, Гановер, Нью–Гемпшир. Исследование основано на предположении, что всякий аспект обучения или любое другое свойство интеллекта может в принципе быть столь точно описано, что машина сможет его имитировать. Мы попытаемся понять, как обучить машины использовать естественные языки, формировать абстракции и концепции, решать задачи, сейчас подвластные только людям, и улучшать самих себя. Мы считаем, что существенное продвижение в одной или более из этих проблем вполне возможно, если специально подобранная группа учёных будет работать над этим в течение лета.

Вот такой незамысловатый проект — по сути ерунда, работы немного, десятку человек на одно лето... В реальности же оказалось, что искусственный интеллект — задача почти безнадёжная. Несмотря на многолетние усилия лучших умов мира, создать пусть даже не лучший из умов пока не получается. Какой там тест Тьюринга — даже простое чтение безо всякого понимания оказывается для компьютера непосильной задачей, о чём свидетельствуют многочисленные CAPTCHA–тесты на различных сайтах[1].

Проблемы с «человекоподобными» задачами появляются даже в настольных играх, где, на первый взгляд, задача достаточно хорошо формализована. Компьютер может хорошо играть в шахматы, даже обыгрывать за доской ведущих гроссмейстеров

[1] CAPTCHA (Completely Automated Public Turing Test to tell Computers and Humans Apart) — это тест, в котором от человека требуется распознать написанный текст или какое-нибудь другое свойство предъявленного ему изображения. Такие тесты часто используются для «защиты от ботов», — доказательства того, что тест проходит живой человек, а не программа (бот) [1, 2].

(на сегодня рейтинг Эло лучших компьютеров составляет около 3100–3200, а лучших живых шахматистов — 2700–2800), потому что в шахматах не слишком большая разветвлённость дерева поиска даёт большой простор для применения вычислительных мощностей. Но, например, в игре го, где нужно значительно тоньше понимать позицию на всей доске и где умение быстро считать варианты не может радикально усилить игру, компьютеры пока не выдерживают никакой критики. Лучшие компьютерные программы играют в го на уровне не слишком увлечённого любителя, а профессионалам проигрывают с гигантской форой, доходящей до 20–25 камней (кому может чемпион мира по шахматам дать ферзя и двух ладей вперёд?) [23, 115]. И это — строго формализованная игра на конечной, весьма ограниченной доске[1]. О какой-либо деятельности, где нужно понимать естественный язык, конечно, и говорить не приходится.

Однако поле искусственного интеллекта весьма широко, и было бы в корне неправильно оценивать успехи отрасли сугубо бинарно, по факту достижения основной цели. На пути к ней были разработаны многие подходы, которые получили широкое распространение и оказались очень полезны на практике. Большая часть этих подходов объединена под названием *машинного обучения* (*machine learning*). Именно ему посвящена эта книга.

Машинное обучение и интеллектуальные агенты

Что такое машинное обучение? Общее (хотя и не слишком формальное) определение таково: алгоритм является алгоритмом машинного обучения, если он улучшает своё поведение по мере накопления опыта. Что это значит? Это значит, что алгоритм *обучает* параметры модели либо на заранее подготовленных тестовых примерах, либо на собственных ошибках, и со временем решает поставленную задачу всё лучше и лучше. Некоторые алгоритмы машинного обучения способны подмечать ранее неизвестные закономерности в данных, выделять *знания* (тоже

[1] Позднейший комментарий: к 2008 году наметился прогресс и в го; программа MoGo, основанная на специальном алгоритме построения дерева поиска, смогла вывести компьютерное го на новый уровень; правда, о мировом господстве пока мечтать рано, но уровень где-то первого дана достигнут [46, 47].

термин и предмет искусственного интеллекта), которых раньше не было — словом, делать то, что долгое время считалось прерогативой исключительно *Homo Sapiens*.

Обучающиеся агенты — частный случай интеллектуальных агентов, которые изменяют своё поведение в зависимости от реакции окружающей среды. Главным отличием обучающихся агентов от всех остальных является то, что они не просто действуют по какой-либо программе, в которую заранее заложена реакция на разные действия среды, а *улучшают* своё поведение *с опытом*. Иными словами, в определении алгоритма машинного обучения всегда присутствует требование повышать эффективность по ходу работы или при увеличении начальной обучающей выборки.

Обучающиеся агенты всегда действуют в условиях либо неполной информации, либо информации, полностью обработать которую вычислительно нереально — иначе можно было бы просто выбирать оптимальное поведение, и обучаться было бы не нужно. А нетривиальные алгоритмы для обучающихся агентов появляются только тогда, когда оптимальное со статистической точки зрения поведение слишком трудно вычислить полным перебором. Приведём здесь же ссылки на главные источники по машинному обучению — книги [5, 19, 63, 92, 101, 107–109, 113].

На рисунке 1.1 изображена схема (очень общая и приблизительная) самообучающегося агента. Давайте рассмотрим основные её элементы:

— *решатель* — мозг самообучающегося агента, именно он получает информацию из внешнего мира и выбирает, какие действия производить;
— *критик* связывается с неким стандартом (оценкой, идеалом) эффективности и определяет, насколько хорошо отработал агент;
— *обучающийся элемент* улучшает поведение агента, изменяя его параметры в соответствии с тем, хорошо ли получилось у решателя (о том, что такое хорошо и что такое плохо, этой крохе сообщит критик); он же сообщает генератору задач, какой ещё нужен опыт для дальнейшего обучения;

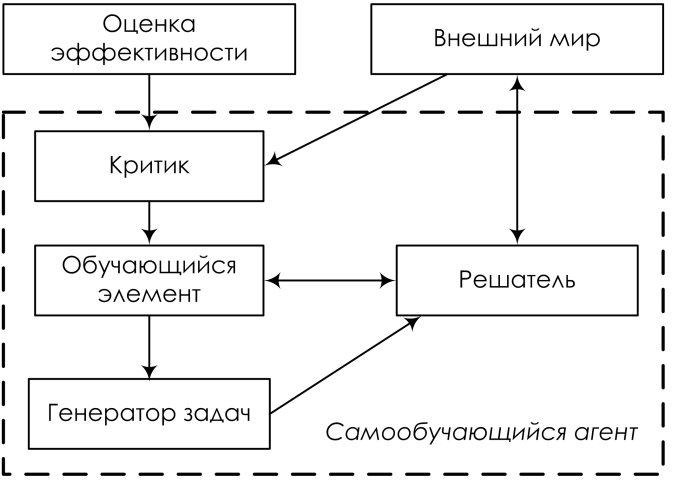


Рис. 1.1. Схема самообучающегося агента

— *генератор задач* предлагает действия, которые привели бы к новому, полезному опыту: во многих областях применения машинного обучения агенту приходится выбирать, какие ещё стратегии исследовать (а может, удовольствоваться тем, что он уже знает).

Не во всех аппаратах машинного обучения присутствуют все эти элементы, и во многих из них несколько элементов схемы объединяются в один. Однако схема помогает понять, как действует большинство алгоритмов машинного обучения.

На чём стоит искусственный интеллект

Искусственный интеллект вообще и машинное обучение в частности — поле в высшей степени междисциплинарное. В нём много раз использовались идеи из разных областей знания, на первый взгляд совершенно не похожих ни друг на друга, ни на информатику. В книге мы неоднократно будем указывать на подобные обстоятельства, а сейчас просто вкратце опишем главное: на чём стоит современный искусственный интеллект, фундамент, без которого его трудно было бы себе представить.

Математика, конечно, в любом случае остаётся царицей и служанкой любой науки. А поскольку искусственный интеллект — раздел информатики, в котором речь идёт об идеализированных объектах (функциях, алгоритмах, оценках сложности, структурах данных), без математики здесь никуда. Однако некоторые разделы математики встречаются в искусственном интеллекте значительно чаще, чем другие.

Теорию вероятностей и *статистику* можно без преувеличения назвать основой всего машинного обучения вообще и значительной доли других исследований в рамках искусственного интеллекта. Даже если алгоритм на первый взгляд не использует вероятности или случайные процессы, при ближайшем рассмотрении наверняка окажется, что для его анализа придётся привлекать вероятность. Об этом мы будем подробно говорить в главе 5, да и во всех остальных.

Математическая логика пронизывает всю информатику. Бинарная логика, созданная Джорджем Булем[1], стала основным инструментом информатики. Да и вообще информатика была во многом мотивирована логикой — оптимизм логицистов начала XX века, веривших, что всю математику можно автоматически вывести из умело подобранных аксиом, был одной из отправных точек развития автоматического вывода. Апофеозом логицизма стал многотомный труд Альберта Уайтхеда и Бертрана Рассела «Principia Mathematica» («Принципы математики»), в котором из аксиом выводились арифметические утверждения. Книга эта очень похожа на автоматически порождённый логический вывод, и к странице 379 первого издания авторы наконец-то заключают: «Когда мы определим арифметическое сложение, из этого предложения будет вытекать, что $1+1=2$».

[1] Джордж Буль (George Boole, 1815–1864) — английский математик и философ, один из основателей информатики. Трудно рассказать о Буле какой-нибудь забавный случай, потому что жизнь его была на редкость стабильной и безынтересной — он всю жизнь проработал преподавателем математики, сначала в школе, потом в колледже. Трудом его жизни стал двухтомник «Исследование законов мышления, на которых основаны математические теории логики и вероятностей» [20], где Буль исследовал связь между алгеброй, логикой и вероятностью и развил законы алгебраической записи логических законов и бинарной логики. Идеи Буля в XIX веке никого не заинтересовали — потребовался Клод Шеннон, чтобы понять, что булева логика может стать основой информатики.

Правда, оптимизма логицистам изрядно поубавили теоремы Гёделя[1] о неполноте, но задача автоматизации логического вывода с повестки дня не сошла. Она и по сей день остаётся одним из краеугольных камней не только теории доказательств, но и искусственного интеллекта: многие аппараты должны проводить логический вывод (или его аналоги). Кроме того, искусственный интеллект часто работает с различными мерами неопределённости [57, 176, 177]. В этом ему помогают вероятностные логики [176], основы которых заложил ещё Буль [20], и нечёткие логики [40, 117].

Теория игр изучает взаимодействие агентов, каждый из которых стремится выбрать стратегию поведения, оптимальную с точки зрения той или иной целевой функции. Разумеется, такого рода задачи тесно связаны с задачами машинного обучения. Теория игр изучает те случаи, в которых машинное обучение уже не обязательно: она предоставляет готовые оптимальные рецепты в той или иной ситуации. Однако зачастую, прежде чем использовать эти рецепты, нужно установить некоторые параметры ситуации, в которой находится самообучающийся агент, или задать принципы взаимодействия нескольких агентов. Поэтому теория игр активно используется в алгоритмах машинного обучения. Примерно ту же роль играет и *теория управления*, которая тоже решает задачи максимизации заданной функции.

[1] Курт Гёдель (Kurt Gödel, 1906-–1978) — австрийский математик. В 1924 г. поступил в Венский университет, в 1930 г. защитил диссертацию по математике; в своей диссертации Гёдель доказал полноту исчисления предикатов первой ступени, что дало надежду на доказательство непротиворечивости и полноты всей математики. Однако уже в 1931 г. Гёдель надежду отнял, доказав свою знаменитую теорему о неполноте. Согласно этой теореме, любая процедура доказательства истинных утверждений арифметики обречена на неполноту. Следовательно, внутренняя непротиворечивость любой математической теории не может быть доказана иначе, как с помощью обращения к другой теории, использующей более сильные аксиомы, а значит, менее надёжной. Методы, использованные Гёделем при доказательстве теоремы о неполноте, сыграли в дальнейшем важную роль в информатике. Правда, позитивных утверждений у Гёделя было всё-таки больше: например, в 1938 г. он доказал, что присоединение аксиомы выбора и континуум–гипотезы к обычным аксиомам теории множеств не приводит к противоречию.

Как ни странно, в ряду полезных для машинного обучения разделов математики не обойтись без *теории дифференциальных уравнений*. Встречаются и динамические системы, и уравнения в частных производных (то есть по сути математическая физика). Такая «тяжёлая артиллерия» обычно используется для доказательства сходимости различных алгоритмов машинного обучения. Доходит до забавного: например, чтобы доказать сходимость сетей Хопфилда [67, 134] (в этой книге мы их, к сожалению, рассматривать не будем — следите за новыми публикациями), нужно начать со свободной энергии системы из нескольких элементарных частиц с различными спинами!

Из нематематических дисциплин в первую очередь следует отметить *биологию*. *Нейробиология* — источник идей для искусственных нейронных сетей, обучения по Хеббу и других подобных алгоритмов, а *генетика* породила обширную область генетических алгоритмов и генетического программирования. Здесь ещё стоит отметить, что всё та же генетика сейчас предоставляет большое количество интересных задач для информатики вообще и машинного обучения в частности: работа по расшифровке геномов различных биологических видов и выделению из них полезной информации немыслима без компьютеров, ведь геном — это последовательность, состоящая порой из миллиардов нуклеотидов, которую ещё нужно построить из обрывков, а затем искать в ней какие-либо закономерности. Подробнее о биоинформатике вообще можно прочесть в [114, 161], а конкретно о связи биоинформатики и машинного обучения — в [9, 81]. Мы же только заметим, что подобную роль — постановку хорошо определённых, полезных и сложных задач — в развитии того или иного раздела науки трудно переоценить. Если нет чётко поставленных целей, у ведущих учёных обычно нет и большого интереса этой наукой заниматься, не говоря уже о том, что задачи, полезные человечеству вообще и фармацевтическим компаниям в частности, могут привлечь к исследованиям значительные деньги.

Наконец, кроме биологии, искусственному интеллекту помогает также и *психология*. Точнее, в основном *когнитивистика* (cognitive science) — раздел психологии, изучающий законы разума, интеллекта, поведения мыслящих субъектов. Не

стоит забывать, что искусственный-то, конечно, искусственный, но всё-таки интеллект: чем лучше мы понимаем, как работает наш собственный мозг и наше собственное мышление, тем более вероятно, что мы сможем и сами сделать нечто подобное. Похожую роль играет и *лингвистика*, хотя она, в некоторых аспектах тесно смыкаясь с математической логикой, в основном направлена конкретно на то, чтобы обучить компьютер понимать естественные языки. Однако задача эта столь сложна и многогранна, что её решение или даже путь к нему оказывают влияние на весь искусственный интеллект.

Конечно, математика и информатика остаются для искусственного интеллекта базовыми дисциплинами. Пока мы не построим математические модели, не изучим их свойства, не снабдим их соответствующими алгоритмами и структурами данных, никакой автоматизации (то есть никакого искусственного интеллекта) у нас не получится.

Однако, как мы увидели на вышеприведённых примерах, искусственный интеллект — точка пересечения очень многих областей знания. Список, который мы привели, наверняка можно продолжить. Кто знает, может быть, вы, читатель, сможете привнести в машинное обучение новые методы, позаимствовав их основные идеи из совершенно других, не перечисленных здесь областей. Если так — это станет ещё одним доказательством того, насколько многообразен мир искусственного интеллекта.

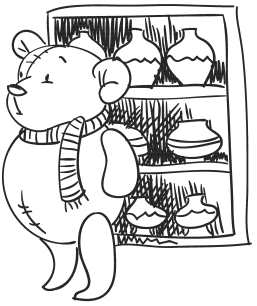
Глава 1

Деревья принятия решений

> — А всё из-за того, — признался он наконец, когда перекувырнулся ещё три раза, пожелал всего хорошего самым нижним веткам и плавно приземлился в колючий-преколючий терновый куст, — всё из-за того, что я слишком люблю мёд!
>
> *Винни-Пух и Все-Все-Все*
> *Алан Александр Милн* в переводе *Бориса Заходера*

Дома у Винни-Пуха стоит целый шкаф пустых горшочков. Кристофер Робин только что пригласил медвежонка на обед, и Винни наверняка знает, что у него получится на обратном пути захватить с собой немножко мёда. Нужно только понять, какой пустой горшочек (воистину полезная вещь!) для этого выбрать.

Но вот незадача: некоторые из горшочков треснули, а в некоторых Винни долго хранил крысиный яд и не очень тщательно мыл. Винни (как следует поразмыслив, конечно) понимает, что ни треснутые, ни ядовитые горшочки под мёд лучше не использовать, иначе либо испортится мёд, либо испортится сам Винни. Винни выбирает горшочек, который и не треснул, и без крысиного яда, и радостно идёт в гости к Кристоферу Робину.



Говоря математически, медвежонок Пух выявил два атрибута горшочка и построил два логических правила: «Если горшочек треснул — мёд не наливать» и «Если в горшочке был крысиный яд — мёд не наливать». Таким образом, горшочки оказались расклассифицированы относительно атрибутов «треснул» и «из-под яда» и целевой функции «можно наливать мёд».

Но и это ещё не вся история.

На самом деле Винни, конечно, давно забыл, в каких именно горшочках у него был крысиный яд. Он всего лишь предполагает (бедный, наивный медвежонок...), что его горшочки в этом смысле похожи на горшочки из шкафа Совы, на которых она уже написала — она умеет! — был в них яд или нет. Винни должен внимательно изучить шкаф Совы и понять принцип — Пух уверен, что принцип есть — которым Сова руководствовалась при этом. Например, «если горшочки синенькие и с белой полоской, то в них был крысиный яд, а если стеклянные и объёмом в поллитра, то не было». А затем медвежонку предстоит экстраполировать эти закономерности на свой собственный шкаф... на этом, пожалуй, милостиво опустим занавес.

То, что Винни только что сделал, мы должны будем научиться делать автоматически, на основе достаточно большого массива данных с разными атрибутами.

§ 1.1. Введение

Рассмотрим некоторую предметную область. Предположим, что мы хотим в этой предметной области что-нибудь расклассифицировать, и у нас есть некоторые атрибуты, по которым это можно было бы сделать. Для решения этой задачи можно применять много разных идей; в этой главе мы рассмотрим один из не слишком сложных, но зачастую полезных аппаратов.

Деревья принятия решений (decision trees) используются для решения задач классификации данных или, иначе говоря, для задачи *аппроксимации заданной булевской функции* (булевская функция — это функция, у которой всего два значения: true и false; алгебраисты скажут, что она действует в $\mathbb{F}_2$). Ситуация, в которой применяются деревья принятия решений, обычно выглядит так: есть много случаев, каждый из которых описывается некоторым *конечным* набором *дискретных* атрибутов, и в

каждом из случаев дано значение некоторой неизвестной булевской функции, зависящей от этих атрибутов. Задача — создать достаточно экономичную конструкцию, которая бы описывала эту функцию и позволяла классифицировать новые, поступающие извне данные.

Пример 1.1. Постановка задачи классификации

Предположим, что нас интересует, выиграет ли футбольный клуб «Зенит» (Санкт-Петербург) свой следующий матч. Мы знаем, что это зависит от ряда параметров; перечислять их все — задача безнадёжная, поэтому ограничимся основными:

— выше или ниже находится соперник по турнирной таблице;
— дома ли играется матч;
— пропускает ли матч кто-либо из лидеров команды;[1]
— идёт ли дождь.

У нас есть некоторая статистика на этот счёт — см. табл. 1.1 (совпадения случайны). Мы хотим предсказать, выиграет ли «Зенит» в следующей игре. Предположим, что сегодняшний соперник стоит ниже «Зенита», игра происходит на выезде, лидеры на месте, а дождя нет. Эта ситуация описана в последней строке таблицы, но исход, правильный ответ нам пока неизвестен. Как предсказать исход? На этот вопрос могут ответить деревья принятия решений.

Разумеется, если мы сможем обучить компьютер обрабатывать данные и предсказывать результат в таких ситуациях, это будет типичной постановкой задачи машинного обучения. При этом качество наших прогнозов должно быть тем лучше, чем больше информации предоставлено. Посмотрим, как деревья принятия решений справляются с такими задачами.

§ 1.2. Структура дерева принятия решений

Продолжим рассматривать пример 1.1 и попытаемся построить настоящее дерево принятия решений. Напомним на всякий

[1] Этот пример мы придумали в конце 2006 года, когда игра Александра Кержакова и Андрея Аршавина определяла лицо «Зенита» в подавляющем большинстве матчей. С тех пор уже (начало 2009) многое изменилось, и сегодняшний «Зенит» играет по-другому. Однако, поскольку автоматически следить за футбольной конъюнктурой эта книга всё равно не сможет, мы решили оставить этот пример как есть.

Таблица 1.1. Как играет «Зенит»

Соперник	Играет	Лидеры	Дождь	Победа
Выше	Дома	На месте	Да	Нет
Выше	Дома	На месте	Нет	Да
Выше	Дома	Пропускают	Нет	Да
Ниже	Дома	Пропускают	Нет	Да
Ниже	В гостях	Пропускают	Нет	Нет
Ниже	Дома	Пропускают	Да	Да
Выше	В гостях	На месте	Да	Нет
Ниже	В гостях	На месте	Нет	*???*

случай, что в теории графов *деревьями* называются графы, в которых нет циклов даже без учёта направления рёбер, а *листьями* называются вершины дерева, из которых не исходит ни одного ребра (на рис. 1.2 листьями являются все вершины с 0 или 1).

Так вот, в узлах дерева принятия решений, не являющихся листьями, находятся атрибуты, по которым различаются случаи. В листьях находятся значения целевой функции. А по рёбрам мы будем спускаться, чтобы классифицировать имеющиеся случаи.

Для начала просто используем атрибуты в порядке, в котором они указаны в таблице. Тогда у нас получится структура, изображённая на рис. 1.2 (1 означает победу, 0 — поражение или ничью).

Для классификации новых примеров нужно пройти по дереву сверху вниз и определить, в какой из листьев попадает интересующая нас ситуация (чем-то это нам напоминает тот долгий путь, который проделал Винни-Пух по милости неправильных пчёл).

Пример 1.2. Использование дерева принятия решений

Вспомним условия примера 1.1. Соперник стоит ниже «Зенита», следовательно, спускаемся налево, матч проходит в гостях — спускаемся направо и получаем, что «Зенит», судя по нашему дереву, этот матч выиграть не должен.

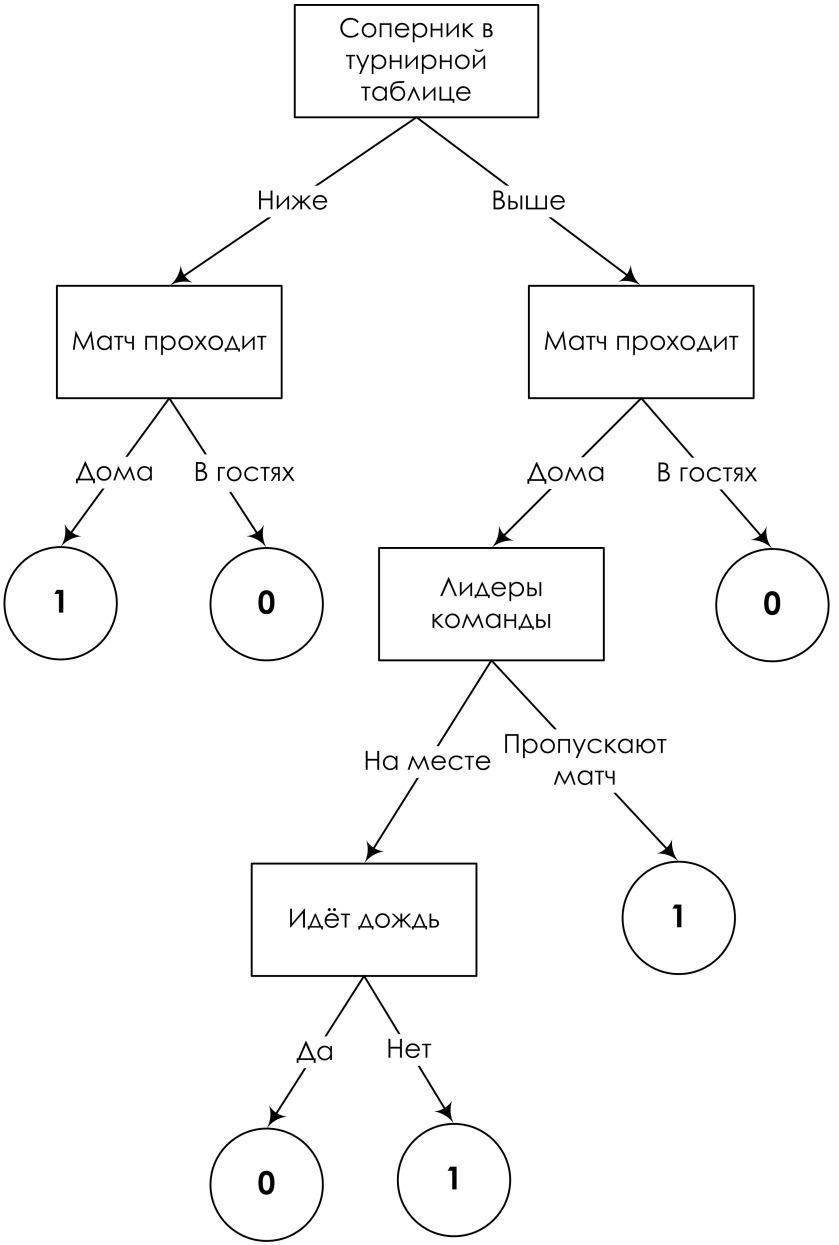


Рис. 1.2. Первый вариант дерева принятия решений

Для того чтобы реализовать машинное обучение, нужно просто видоизменять дерево принятия решений в соответствии со вновь поступающей информацией. Например, если бы в вышеописанном случае «Зенит» проиграл, ничего менять было бы не нужно. Но если бы выиграл, то дерево принятия решений пришлось бы дополнить ещё одним узлом, и оно приняло бы вид, изображённый на рис. 1.3.

«Качество» дерева принятия решений напрямую связано с его глубиной: чем меньше глубина, тем быстрее можно по дереву спуститься и получить ответ. Построенное нами дерево далеко не идеально — например, глубина его равна четырём. Чтобы сделать его компактнее, мы могли бы взять за основу другой атрибут. Давайте попробуем поместить в корень вопрос о том, идёт ли дождь во время матча. В случае, если дождя нет, следующим атрибутом будет положение соперника в турнирной таблице, а в случае, если дождь идёт, мы будем смотреть на то, проходит ли матч дома или в гостях. В таком случае глубина нашего дерева будет равна всего лишь двум (см. рис. 1.4), но оно по-прежнему удовлетворяет всем тестовым примерам. Легко убедиться, что ни один атрибут сам по себе не может однозначно разделить значения функции, поэтому дерева глубины 1 построить не получится. Следовательно, изображённое на рис. 1.4 дерево оптимально относительно глубины (оптимальное дерево, конечно, не обязано быть единственным).

§ 1.3. Энтропия и прирост информации

Введённое в теорию информации Клодом Шенноном[1] понятие энтропии играет важнейшую роль в современной информатике. Пригодится оно и для деревьев принятия решений; но

[1] Клод Шеннон (Claude Shannon, 1916–2001) — американский инженер и математик, отец теории информации и теории кодирования. Он учился на инженера-электрика и быстро понял, что к электрическим схемам можно с успехом приложить идеи Джорджа Буля; его институтский диплом [143] называют самым важным и самым знаменитым магистерским дипломом двадцатого века. В нём он развил теорию электрических схем, способных решать логические задачи, заложив основу архитектуры будущих компьютеров. Затем он фактически с нуля создал теорию информации, основанную на понятии энтропии [144]. У Шеннона также немало достижений в искусственном интеллекте и криптографии.

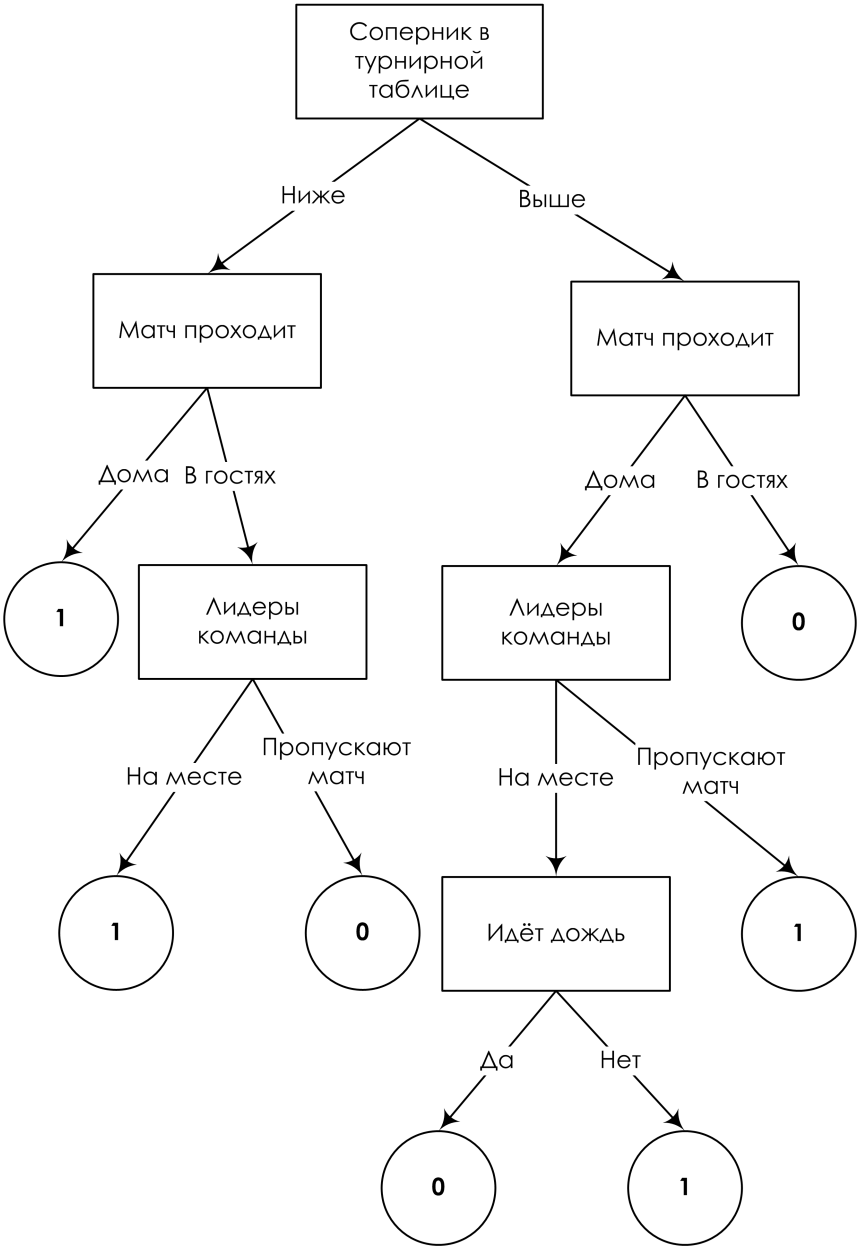


Рис. 1.3. Дерево после добавления ещё одного случая

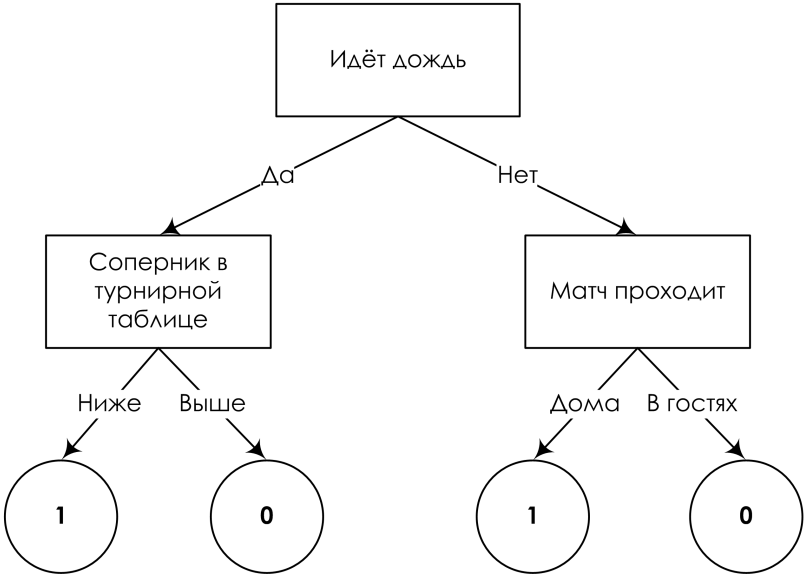


Рис. 1.4. Оптимальное дерево принятия решений

сначала мы напомним читателю о том, что такое энтропия в интересующем нас дискретном случае, — для более фундаментального и математически строгого изложения как дискретной, так и непрерывной теории энтропии можно порекомендовать, например, книгу [171].

Философски говоря, энтропия — это мера неопределённости события. На самом деле, даже такое абстрактное и, на первый взгляд, ничего строго математического не дающее определение позволяет выяснить, как должна выглядеть функция энтропии, практически не оставив свободы выбора. Для этого нужно просто подумать, какие интуитивные требования мы предъявляем к мере неопределённости. Давайте попробуем эти интуитивные требования изложить, а затем формализовать (формализации ниже будут пронумерованы от **UM1** до **UM3**, от слов «uncertainty measure»).

Во-первых, примем естественное предположение, что неопределённость зависит не от самих событий, а только от их вероятностей. Во-вторых, понятно, что мера неопределённости Λ должна быть равна нулю, если событие точно произойдёт:

ЦМ1. $\Lambda(1) = 0$.

В-третьих, если одно событие имеет меньшую вероятность, чем другое, то наступление первого из них несёт больше информации, сильнее уменьшает неопределённость; значит, и энтропия у него должна быть больше:

ЦМ2. Если $x > y$, то $\Lambda(x) < \Lambda(y)$, то есть Λ как функция монотонно убывает.

И, наконец, в-четвёртых, логично предположить, что если два события независимы, то наступление каждого из них уменьшает общую неопределённость независимо от другого. Иными словами, если два независимых события наступают одновременно (вероятность этого равна произведению их вероятностей), то это уменьшает неопределённость на величину, равную сумме неопределённостей этих событий:

ЦМ3. $\Lambda(xy) = \Lambda(x) + \Lambda(y)$.

Если ещё добавить желательное условие непрерывности, то эти условия, особенно **ЦМ3**, будут фактически предопределять вид меры неопределённости — она обязана быть логарифмической функцией. А *энтропия случайной величины* — это математическое ожидание её меры неопределённости. В дискретном случае это приводит нас к следующему определению.

Определение 1.1. *Энтропия случайной величины* $\hat{\xi}$ со множеством возможных исходов $\{\xi_1, \ldots, \xi_n\}$, определяется как

$$H(\hat{\xi}) = \sum_{i=1}^{n} p(\xi_i) \log_2 \frac{1}{p(\xi_i)} = -\sum_{i=1}^{n} p(\xi_i) \log_2 p(\xi_i).$$

Стоит отметить, что энтропия не зависит от собственно случайной величины, а только от ее распределения $p_{\hat{\xi}}$. В конечном случае разумно будет задать энтропию на векторе $\mathbf{P}_{\hat{\xi}}$ значений вероятностей дискретных исходов:

$$H(\mathbf{P}_{\hat{\xi}}) = -\sum_{i=1}^{n} \mathbf{P}^i_{\hat{\xi}} \log_2 \mathbf{P}^i_{\hat{\xi}}.$$

Ещё одно важное свойство, напрямую вытекающее из **ЦМ3**, состоит в том, что энтропия, задаваемая совместным распределением двух случайных величин, максимальна тогда, когда эти две случайные величины независимы.

Пример 1.3. Энтропия двух случайных величин

Предположим, что у нас есть две случайные величины $\hat{\xi}$ и $\hat{\psi}$, у каждой из которых два исхода (обозначим их через 0 и 1). Пусть

$$\begin{aligned} p_{\hat{\xi}}(0) &= p(\hat{\xi}=0) = 0{,}4, \\ p_{\hat{\psi}}(0) &= p(\hat{\psi}=0) = 0{,}8. \end{aligned}$$

Тогда их энтропия вычисляется следующим образом:

$$\begin{aligned} H(\hat{\xi}) &= -p_{\hat{\xi}}(0)\log_2 p_{\hat{\xi}}(0) - p_{\hat{\xi}}(1)\log_2 p_{\hat{\xi}}(0) \approx 0{,}2923, \\ H(\hat{\psi}) &= -p_{\hat{\psi}}(0)\log_2 p_{\hat{\psi}}(0) - p_{\hat{\psi}}(1)\log_2 p_{\hat{\psi}}(0) \approx 0{,}2173. \end{aligned}$$

При этом энтропия тем больше, чем ближе вероятности исходов друг к другу; для случайной величины с двумя равновероятными исходами энтропия была бы равна $-0{,}5\log_2 0{,}5 - 0{,}5\log_2 0{,}5 \approx 0{,}3010$.

Вычислим теперь совместную энтропию этих двух случайных событий в двух разных случаях. В первом случае события будут независимы, то есть

$$p(\hat{\xi}=\xi, \hat{\psi}=\psi) = p_{\hat{\xi}}(\xi)p_{\hat{\psi}}(\psi).$$

В этом предположении энтропия будет равна

$$\begin{aligned} H(\widehat{\xi\psi}) = {} & p_{\hat{\xi}}(0)p_{\hat{\psi}}(0)\log_2(p_{\hat{\xi}}(0)p_{\hat{\psi}}(0)) + \\ & + p_{\hat{\xi}}(0)p_{\hat{\psi}}(1)\log_2(p_{\hat{\xi}}(0)p_{\hat{\psi}}(1)) + \\ & + p_{\hat{\xi}}(1)p_{\hat{\psi}}(0)\log_2(p_{\hat{\xi}}(1)p_{\hat{\psi}}(0)) + \\ & + p_{\hat{\xi}}(1)p_{\hat{\psi}}(1)\log_2(p_{\hat{\xi}}(1)p_{\hat{\psi}}(1)) \approx 0{,}5096. \end{aligned}$$

С другой стороны, давайте подсчитаем энтропию в обратном предположении: пусть теперь событие $\{\hat{\xi}=0\}$ является подмножеством события $\{\hat{\psi}=0\}$, то есть

$$\begin{aligned} &p(\hat{\xi}=0,\hat{\psi}=0) = 0{,}4, \quad p(\hat{\xi}=1,\hat{\psi}=0) = 0{,}4, \\ &p(\hat{\xi}=0,\hat{\psi}=1) = 0, \qquad p(\hat{\xi}=1,\hat{\psi}=1) = 0{,}2. \end{aligned}$$

Тогда энтропия будет вычисляться так (функцию $x\log x$ в таких случаях доопределяют в нуле по непрерывности, то есть $0\log 0 = 0$):

$$\begin{aligned} H(\widehat{\xi\psi}) = {} & p(\hat{\xi}=0,\hat{\psi}=0)\log_2 p(\hat{\xi}=0,\hat{\psi}=0) + \\ & + p(\hat{\xi}=1,\hat{\psi}=0)\log_2 p(\hat{\xi}=1,\hat{\psi}=0) + \\ & + p(\hat{\xi}=0,\hat{\psi}=1)\log_2 p(\hat{\xi}=0,\hat{\psi}=1) + \\ & + p(\hat{\xi}=1,\hat{\psi}=1)\log_2 p(\hat{\xi}=1,\hat{\psi}=1) \approx 0{,}4580. \end{aligned}$$

Как видно, она получилась меньше, что подтверждает тезис о том, что энтропия максимизируется на независимых событиях.

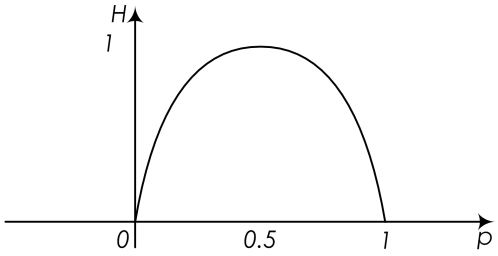


Рис. 1.5. Зависимость энтропии от пропорции

Понятие энтропии тесно связано с теорией информации. С этой точки зрения энтропия представляет собой среднее количество битов, которые требуются, чтобы закодировать атрибут S у элемента множества A. Если вероятность появления S равна $1/2$, то энтропия равна 1, и нужен полноценный бит; а если S появляется не равновероятно, то можно закодировать последовательность элементов A более эффективно. Мы не будем углубляться здесь в теорию кодирования; заинтересованный читатель сможет найти гораздо больше информации и об энтропии, и об оптимальных кодах в обширной существующей литературе, например в [14, 71, 171, 174].

Вернёмся теперь к деревьям принятия решений. Интуитивно понятно, что для того, чтобы получить оптимальные деревья принятия решений, нужно на каждом шаге выбирать атрибуты, которые «лучше всего» характеризуют целевую функцию. Это требование формализуется как раз через понятие энтропии.

Энтропия зависит от пропорции, в которой разделяется множество. По мере возрастания этой пропорции от 0 до $1/2$ энтропия тоже возрастает, а после $1/2$ — симметрично убывает (в случае деления на две части; см. рис. 1.5). А в случае деления множества на несколько частей энтропия максимальна тогда, когда части равновеликие, и равна нулю, когда одна из частей занимает всё множество.

Всё это приводит нас к мысли о том, что при выборе атрибута для классификации нужно выбирать его так, чтобы после классификации энтропия в каждом потомке (потомком узла x называется узел, в который из x ведёт ребро) стала как можно

меньше (свойство S в данном случае — значение целевой булевской функции). Идеальный случай — когда в каждом потомке энтропия равна нулю, то есть функция полностью классифицирована текущим атрибутом. Энтропия при этом будет разной в разных потомках, и общую сумму нужно считать с учётом того, сколько исходов осталось в рассмотрении в каждом из потомков. Общепринятое в теории деревьев принятия решений определение выглядит так.

Определение 1.2. Рассмотрим множество A, элементы которого характеризуются свойством S. Пусть при этом элементы A классифицированы посредством некоторого атрибута Q, имеющего q возможных значений. Тогда *прирост информации (*information gain*)* определяется как

$$\mathrm{Gain}(A,Q) = H(A,S) - \sum_{i=1}^{q} \frac{|A_i|}{|A|} H(A_i,S),$$

где A_i — множество элементов A, на которых атрибут Q имеет значение i, а $|A|$ — мощность множества A.

Теперь можно просто применить так называемый *жадный алгоритм*. Жадные алгоритмы на каждом шаге стараются выбирать максимальный прирост целевой функции; для некоторых задач это может привести к глобальной неоптимальности (иногда нужно сначала час потерять, чтобы потом за пять минут долететь), но в нашем случае всё будет хорошо. Жадный алгоритм на каждом шаге выбирает тот атрибут, для которого прирост информации максимален.

Пример 1.4. Вычисление энтропии и прироста информации

Попытаемся определить оптимальный атрибут для примера 1.1. Вычислим исходную энтропию (для максимизации прироста это, конечно, не нужно, но всё же):

$$H(A,\text{Победа}) = -\frac{4}{7}\log_2\frac{4}{7} - \frac{3}{7}\log_2\frac{3}{7} \approx 0{,}9852.$$

Теперь вычислим приросты информации для различных атрибутов:

$$\mathrm{Gain}(A,\text{Соперник}) = H(A,\text{Победа}) -$$
$$- \frac{4}{7}H(A_{\text{выше}},\text{Победа}) - \frac{3}{7}H(A_{\text{ниже}},\text{Победа}) \approx 0{,}9852 -$$

$$-\frac{4}{7}\left(-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}\right)-\frac{3}{7}\left(-\frac{2}{3}\log_2\frac{2}{3}-\frac{1}{3}\log_2\frac{1}{3}\right)\approx 0{,}0202.$$

Мы явно выбрали на рис. 1.2 не слишком удачный атрибут для корня дерева...

$$\mathrm{Gain}(A,\text{Играет}) = \mathrm{H}(A,\text{Победа})- \\ -\frac{5}{7}\mathrm{H}(A_{\text{дома}},\text{Победа})-\frac{2}{7}\mathrm{H}(A_{\text{в гостях}},\text{Победа})\approx 0{,}4696.$$

$$\mathrm{Gain}(A,\text{Лидеры}) = \mathrm{H}(A,\text{Победа})- \\ -\frac{3}{7}\mathrm{H}(A_{\text{на месте}},\text{Победа})--\frac{4}{7}\mathrm{H}(A_{\text{пропускают}},\text{Победа})\approx 0{,}1281.$$

$$\mathrm{Gain}(A,\text{Дождь}) = \mathrm{H}(A,\text{Победа})- \\ -\frac{3}{7}\mathrm{H}(A_{\text{да}},\text{Победа})-\frac{4}{7}\mathrm{H}(A_{\text{нет}},\text{Победа})\approx 0{,}1281.$$

Итак, вычисление прироста информации подсказывает нам, что нужно сначала классифицировать по тому, домашний был матч или гостевой. Это выглядит логично: одна из веток оборвётся сразу («Зенит» в нашем примере ещё не выиграл ни одного гостевого матча), а во второй только в одном случае из пяти результат будет отличаться от других. Однако если мы воспользуемся нашим алгоритмом дальше, то глубина этого дерева будет равна трём: одного атрибута окажется недостаточно, чтобы выделить этот один случай из пяти. Но листьев у дерева останется, как и на рис. 1.4, четыре.

§ 1.4. Деревья принятия решений и булевские функции

Мы уже говорили о том, что основную задачу, которую решают деревья принятия решений, можно сформулировать как задачу аппроксимации некоторой не полностью заданной булевской функции. Но и наоборот, дерево принятия решений можно прочесть как булевскую функцию. Она будет естественным образом выражаться в конъюнктивной нормальной форме (в форме дизъюнкции конъюнкций). Для этого нужно выписать конъюнкции атрибутов, стоящих на рёбрах каждого из путей, приводящих в значение 1, а затем объединить их все дизъюнкцией. При таком подходе КНФ равна единице тогда и только тогда, когда реализуется путь, приводящий в узел с меткой 1. Например, дерево на рис. 1.2 соответствует функции

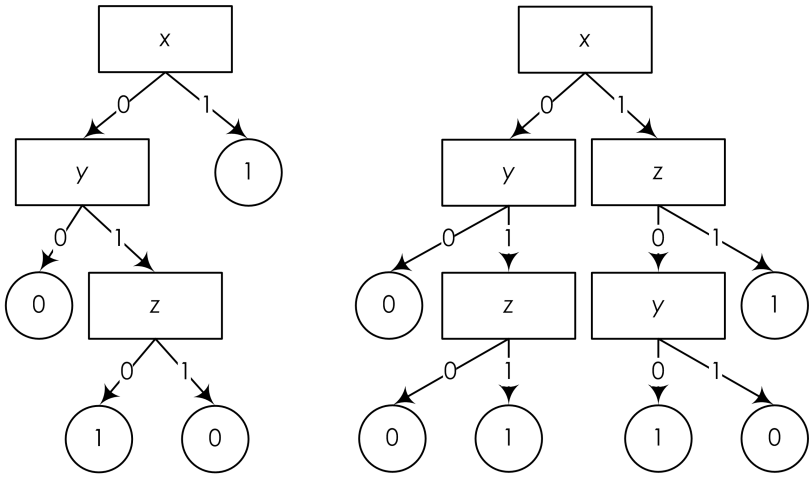


Рис. 1.6. Деревья для булевских функций:
а — $x \vee (y \wedge \bar{z})$; *б* — $(x \vee y) \wedge (\bar{y} \vee z)$

$$\big((\text{Соперник} = \text{Ниже}) \wedge (\text{Играет} = \text{Дома})\big) \vee \big((\text{Соперник} = \text{Выше}) \wedge (\text{Играет} = \text{Дома}) \wedge (\text{Лидеры} = \text{Пропускают})\big) \vee \vee \big((\text{Соперник} = \text{Выше}) \wedge (\text{Играет} = \text{Дома}) \wedge (\text{Лидеры} = \text{На месте}) \wedge (\text{Дождь} = \text{Нет})\big).$$

А дерево на рис. 1.4 — функции

$$\big((\text{Дождь} = \text{Да}) \wedge (\text{Соперник} = \text{Ниже})\big) \vee \vee \big((\text{Дождь} = \text{Нет}) \wedge (\text{Играет} = \text{Дома})\big).$$

Обе функции корректно описывают один и тот же набор данных. Какую из них предпочесть? В § 5.5 мы увидим, что без априорных предположений математических оснований для выбора нет. Но более короткая гипотеза кажется более правдоподобной — бритву Оккама все мы усвоили прочно.[1]

[1] Кстати, о бритве Оккама мы тоже поговорим — в § 5.10; окажется, что этот принцип вполне допускает математическую трактовку.

Пример 1.5. Деревья и булевские функции

На рис. 1.6а изображено дерево принятия решений, соответствующее функции

$$x \vee (y \wedge \bar{z}).$$

А на рис. 1.6б — дерево, соответствующее функции

$$(x \vee y) \wedge (\bar{y} \vee z).$$

Обратите внимание, что функцию нужно сначала привести к дизъюнктивной нормальной форме:

$$(x \vee y) \wedge (\bar{y} \vee z) = (x \wedge z) \vee (x \wedge \bar{y}) \vee (y \wedge z).$$

§ 1.5. Алгоритм ID3

Сейчас мы сведём всё то, о чём до сих пор говорили, в единый рекурсивный алгоритм для построения дерева принятия решений. Этот алгоритм носит название ID3 и был придуман Джоном Квинланом (John R. Quinlan) [130].

Алгоритм выписан на рис. 1.7. В нём, по сравнению с предыдущими примерами, встречаются два тривиальных, но важных обобщения. Во-первых, алгоритм обрабатывает ситуацию, когда одному и тому же набору атрибутов соответствует несколько случаев с разными исходами (например, «Зенит» сыграл в одних и тех же условиях два матча, но один проиграл, а другой выиграл). В такой ситуации мы рано или поздно исчерпаем все атрибуты, которые могли бы разделить исходы, а исходы всё ещё будут разными. Для этого служит пункт 4 алгоритма — решение будем принимать простым большинством.

Другое обобщение — атрибут теперь может иметь несколько значений, более двух, как было у нас раньше. Например, у атрибута «Дождь» могут быть варианты «Нет», «Слабый» и «Ливень». К тому же может так случиться, что какой-то из этих вариантов не реализуется; в таком случае мы заполняем соответствующий лист в зависимости от того, каких исходов было больше в его предке (за это отвечает пункт 7б).

§ 1.6. Реализация ID3 на языках Python и Ruby

В этом параграфе мы опишем программную реализацию алгоритма ID3 на языках Python и Ruby. Программа достаточно

ID3(A,S,$\mathcal{Q}$):

1. Создать корень дерева.
2. Если S выполняется на всех элементах A, поставить в корень метку 1 и выйти.
3. Если S не выполняется ни на одном элементе A, поставить в корень метку 0 и выйти.
4. Если $\mathcal{Q} = \emptyset$, то:
   а) если S выполняется на половине или большей части A, поставить в корень метку 1 и выйти;
   б) если S не выполняется на большей части A, поставить в корень метку 0 и выйти.
5. Выбрать $Q \in \mathcal{Q}$, для которого Gain(A,Q) максимален.
6. Поставить в корень метку Q.
7. Для каждого значения q атрибута Q:
   а) добавить нового потомка корня и пометить соответствующее исходящее ребро меткой q;
   б) если в A нет случаев, для которых Q принимает значение q (т.е. $|A_q| = 0$), то
      (i) пометить этого потомка в зависимости от того, на какой части A выполняется S (аналогично пункту 4),
      (ii) иначе запустить ID3(A_q,S,$\mathcal{Q} \setminus \{Q\}$) и добавить его результат как поддерево с корнем в этом потомке.

Рис. 1.7. Алгоритм ID3

большая, причём значительную её часть занимает чтение и обработка исходного текстового файла и вывод в файл результатов.

Тем не менее, мы приводим её целиком по двум причинам. Во-первых, для того, чтобы читатель освоился с работой со строками и текстовыми файлами в Python и Ruby. Во-вторых, для того, чтобы привести пример полной, реально работающей программы. Сейчас это, конечно, уже не актуально, но в 1980-х

годах Дональда Кнута[1] специально просили издать исходный код его системы TEX (которая, кстати, использовалась и для подготовки этой книги к публикации), потому что это был бы первый пример полностью изданного и полностью прокомментированного кода достаточно сложной программы. Кнут так и сделал [83].

Программа обрабатывает только бинарные атрибуты. Формат входного файла таков (построчно):

— число атрибутов n;
— n строк формата

«[Название] [Значение 1] [Значение 0]»;

— название целевого атрибута;
— число тестовых примеров m;
— m строк формата

«[атрибут$_1$=значение$_1$] ... [атрибут$_n$=значение$_n$]».

Имя файла подаётся в качестве первого параметра функции `applyID3(...)`; второй её параметр — имя файла, куда программа запишет результат своей работы. Результатом является дерево, уровни которого отделяются табуляциями. Например, входной файл, соответствующий примеру 1.1, будет выглядеть так:

```
5
Opponent Higher Lower
Match Home Away
Leaders Play Pass
Rain Yes No
Victory Yes No
Victory
7
```

[1] Дональд Кнут (Donald Knuth, p. 1938) — один из самых знаменитых представителей современной информатики. Он был одним из отцов-основателей *теории сложности*, математического анализа сложности алгоритмов. Первые три тома его известнейшего труда — «Искусство программирования» — вышли в конце 1960-х, на заре computer science как науки, и до сих пор читаются и переиздаются [84–86] — долговечность, просто неслыханная для информатики. Кнут сейчас (2009) по кусочкам выпускает четвёртый том (Enumeration and Backtracking) и планирует взяться за пятый (Syntactical Algorithms). Кроме того, Дональд Кнут — автор популярнейшей, особенно в точных науках, издательской системы TEX [82, 83].

```
Opponent=Higher Match=Home Leaders=Play Rain=Yes Victory=No
Opponent=Higher Match=Home Leaders=Play Rain=No Victory=Yes
Opponent=Higher Match=Home Leaders=Pass Rain=No Victory=Yes
Opponent=Lower Match=Home Leaders=Pass Rain=No Victory=Yes
Opponent=Lower Match=Away Leaders=Pass Rain=No Victory=No
Opponent=Lower Match=Home Leaders=Pass Rain=Yes Victory=Yes
Opponent=Higher Match=Away Leaders=Play Rain=Yes Victory=No
```

А результат применения `applyID3(...)` к этому файлу выглядит так (как мы и предупреждали выше, дерево получится глубины 3):

```
Match=Home
    Leaders=Play
        Rain=Yes
            0
        Rain=No
            1
    Leaders=Pass
        1
Match=Away
    0
```

Теперь — собственно текст программы.

Листинг 1.1. ID3 с бинарными атрибутами на языке Python

```
def ParseAttributes(infname):
    f = open(infname,'r')
    attr,attrnames,tests,attrvals = {},[],[],{}
    attrnum = int(f.readline())
    for i in xrange(attrnum):
        fWords = f.readline().split()
        attrnames.append(fWords[0])
        attr[fWords[0]] = [i,fWords[1],fWords[2]]
        attrvals[fWords[1]],attrvals[fWords[2]] = 1,0
    num = attr[f.readline().strip()][0]
    testnum = int(f.readline())
    for i in xrange(testnum):
        fWords = f.readline().split()
        test = [0 for i in xrange(attrnum)]
        for j in xrange(attrnum):
            attrib = fWords[j][:fWords[j].find('=')]
            value = fWords[j][fWords[j].find('=') +
              1:len(fWords[j])]
            test[ attr[attrib][0] ] = attrvals[value]
        tests.append(test)
    f.close()
```

```
    return [attrnames,attr,tests,num]
def gain(tests,attrnum,num):
    import math
    def entropy(array):
        def log2(x): return math.log(x)/math.log(2)
        neg = float(len(filter(lambda x:(x[num]==0),array)))
        tot = float(len(array))
        if ((neg==tot) or (neg==0)): return 0
        return -(neg/tot)*log2(neg/tot)
               -((tot-neg)/tot)*log2((tot-neg)/tot)
    res = 0
    for i in xrange(2):
        arr = filter(lambda x:(x[attrnum]==i),tests)
        res += entropy(arr)*len(arr)/float(len(tests))
    return entropy(tests)-res

def ID3(tests,num,f,tabnum,usedattr,attrnames,attr):
    def findgains(x):
        if usedattr[x]: return 0
        return gain(tests,x,num)
    def fwriteline(x): f.write('\t'*tabnum+x+'\n')
    def majority():
        neg = len(filter(lambda x:(x[num]==0),tests))
        pos = len(filter(lambda x:(x[num]==1),tests))
        if (neg>pos): return '0'
        else: return '1'
    gains = map(findgains,xrange(len(tests[0])))
    maxgain = gains.index(max(gains))
    if (gains[maxgain]==0):
        fwriteline(majority())
        return
    arrpos=filter(lambda x:(x[maxgain]==1),tests)
    arrneg=filter(lambda x:(x[maxgain]==0),tests)
    newusedattr=[(usedattr[i] or (i==maxgain))
                 for i in xrange(len(usedattr))]
    fwriteline(attrnames[maxgain] + '=' +
               attr[attrnames[maxgain]][1])
    if (len(arrpos)==0):
        fwriteline('\t'+majority())
    else:
        ID3(arrpos,num,f,tabnum+1,newusedattr,attrnames,attr)
    fwriteline(attrnames[maxgain] + '=' +
               attr[attrnames[maxgain]][2])
    if (len(arrneg)==0):
        fwriteline('\t'+majority())
    else:
        ID3(arrneg,num,f,tabnum+1,newusedattr,attrnames,attr)
```

```
def applyID3(infname,outfname):
    parsed = ParseAttributes(infname)
    attrnames, attr = parsed[0], parsed[1]
    tests, num = parsed[2], parsed[3]
    f = open(outfname,'w')
    usedattr = [(i==num) for i in xrange(len(attr))]
    ID3(tests,num,f,0,usedattr,attrnames,attr)
```

И реализация той же программы с теми же форматами входных и выходных файлов на Ruby:

Листинг 1.2. ID3 с бинарными атрибутами на языке Ruby

```
def ParseAttributes(infname)
    f = open(infname, 'r')
    attr, attrnames, tests = {}, [], []
    attrvals, attrnum = {}, f.readline.to_i
    (0...attrnum).each{ |i|
        fWords = f.readline.split
        attrnames[i] = fWords[0]
        attr[fWords[0]] = [i, fWords[1], fWords[2]]
        attrvals[fWords[1]], attrvals[fWords[2]] = 1, 0
    }
    num = attr[f.readline.strip][0]
    testnum = f.readline.to_i
    (0...testnum).each{ |i|
        test = (0...attrnum).map{ 0 }
        f.readline.split.each{ |w|
            attrib, value = w.split('=')
            test[ attr[attrib][0] ] = attrvals[value]
        }
        tests[i] = test
    }
    f.close
    [attrnames, attr, tests, num]
end

def gain2(tests, attrnum, num)
    def elog(a, b)
        if (a == b) or (a == 0) then 0
        else (a / b) * Math.log(a / b) / Math.log(2) end
    end
    def entropy(array,num)
        neg = array.find_all{ |x| x[num] == 0 }.size
        tot = array.size.to_f
        - elog(neg, tot) - elog(tot - neg, tot)
```

```
    end
    res, splitinfo = 0, 0
    for i in (0..1)
        arr = tests.find_all{ |t| t[attrnum] == i }
        res += entropy(arr, num) * arr.size / tests.size.to_f
        splitinfo -= elog(arr.size, tests.size.to_f)
    end
    splitinfo = -1 if splitinfo == 0
    (entropy(tests, num) - res) / splitinfo
end

def ID3(tests, num, f, tabnum, usedattr, attrnames, attr)
    fwriteline = proc{ |x|
        f.write("\t" * tabnum + "#{x}\n")
        }
    def majority(tests, num)
        neg, pos = 0, 0
        tests.each{ |t|
            if t[num] == 0 then neg += 1 else pos += 1 end
        }
        if neg > pos then '0' else '1' end
    end
    gains = (0...tests[0].size).map{ |x|
        if usedattr[x] then 0 else gain2(tests, x, num) end
        }
    maxgain = gains.index(gains.max)
    if gains.max == 0
        fwriteline.call(majority(tests, num))
        return
    end
    arrpos, arrneg = tests.partition{ |t| t[maxgain] == 1 }
    newusedattr = Array.new(usedattr.size){ |i|
        (usedattr[i] or (i == maxgain))
        }
    fwriteline.call(attrnames[maxgain] + '=' +
                    attr[attrnames[maxgain]][1])
    p = proc{ |a|
        if a.empty?
            fwriteline.call("\t" + majority(tests, num))
        else
            ID3(a, num, f, tabnum + 1,
                newusedattr, attrnames, attr)
        end
    }
    p.call(arrpos)
    fwriteline.call(attrnames[maxgain] + '=' +
                    attr[attrnames[maxgain]][2])
    p.call(arrneg)
```

```
end

def applyID3(infname,outfname)
    attrnames, attr, tests, num = ParseAttributes(infname)
    usedattr = (0...attr.size).map{ |i| i == num }
    f = open(outfname, 'w')
    ID3(tests, num, f, 0, usedattr, attrnames, attr)
    f.close
end
```

Как видно, языки эти на требуемом нам уровне не очень сильно отличаются. И вообще, комментаторы соглашаются, что Python и Ruby практически эквивалентны; при решении, какой из этих двух языков использовать, лучше всего смотреть на то, насколько удобный и подходящий для решения вашей задачи существует для них инструментарий.

§ 1.7. Проблема критерия прироста информации

У критерия прироста информации есть одна существенная проблема. Дело в том, что прирост информации часто будет выбирать атрибуты, у которых больше всего значений.

Пример 1.6. Проблема критерия прироста информации

Предположим, что в нашей таблице игр «Зенита» были записаны ещё и даты игр, причём у каждого матча, что вполне логично, своя дата. Вычислим в таком случае прирост информации:

$$\mathrm{Gain}(A,\text{Дата}) = \mathrm{H}(A,\text{Победа}) - \sum_{i=1}^{n} \frac{1}{n}\mathrm{H}(A_{\text{Дата}=i},\text{Победа}) = \\ = \mathrm{H}(A,\text{Победа}),$$

потому что в каждой из веток только один случай, и энтропия в каждой ветке равна нулю. Таким образом, прирост информации в этом случае будет максимальным из возможных. Но полученное дерево принятия решений будет, конечно, абсолютно бесполезным — даже если бы дата игры имела какое-нибудь астрологическое отношение к результату, всё равно уже прошедшие даты никогда не повторятся.

Что же делать? Очевидно, нужно модифицировать критерий прироста информации так, чтобы он смог справиться с этой

проблемой. Надо как-то уменьшать желательность атрибута с ростом числа его значений.

Здесь полезно посмотреть на обучение деревьев принятия решений с точки зрения теории информации. Минимизируя энтропию, мы на самом деле минимизируем *длину описания* тестовых примеров при условии, что они действительно описываются нашим деревом принятия решений. Действительно, энтропия $H(A,\text{Победа})$ — это в точности среднее количество битов, которое нужно, чтобы описать один исход; на каждом шаге мы минимизируем количество битов, которое нам *ещё осталось* потратить. Но где же биты, которые мы потратили на разделение *на текущем шаге*?

В работе [130] был предложен критерий *Gain Ratio*. Идеологически он учитывает не только количество информации, требуемое для записи результата, но и количество информации, требуемое для разделения по текущему атрибуту. Эта поправка выглядит как

$$\text{SplitInfo}(A,Q) = -\sum_{i=1}^{q} \frac{|A_i|}{|A|} \log_2 \frac{|A_i|}{|A|},$$

а сам критерий — как максимизация величины

$$\text{GainRatio}(A,Q) = \frac{\text{Gain}(A,Q)}{\text{SplitInfo}(A,Q)}.$$

Пример 1.7. Gain Ratio

У атрибута «Дата», описанного в примере 1.6 (мы предполагаем, что все даты разные), показатель SplitInfo равен

$$\text{SplitInfo}(A,\text{Дата}) = -\sum_{i=1}^{7} \frac{1}{7} \log_2 \frac{1}{7} \approx 2{,}80735\ldots,$$

и Gain Ratio получается равным

$$\text{GainRatio}(A,\text{Дата}) = \frac{\text{Gain}(A,\text{Дата})}{\text{SplitInfo}(A,\text{Дата})} = \frac{H(A,\text{Победа})}{\text{SplitInfo}(A,\text{Дата})} \approx \frac{0{,}9852\ldots}{2{,}80735\ldots} \approx 0{,}350935\ldots$$

А для атрибута, показывающего, где проходит матч,

$$\text{SplitInfo}(A,\text{Играет}) = -\frac{5}{7} \log_2 \frac{5}{7} - \frac{2}{7} \log_2 \frac{2}{7} \approx 0{,}86312\ldots,$$

и итоговый Gain Ratio получается

$$\mathrm{GainRatio}(A,\text{Играет}) = \frac{\mathrm{Gain}(A,\text{Играет})}{\mathrm{SplitInfo}(A,\text{Играет})} \approx \frac{0{,}4696\ldots}{0{,}8613\ldots} \approx 0{,}5452\ldots$$

Ещё один аналогичный часто используемый критерий [58, 162] — так называемый *индекс Джини*[1]. Для набора тестов A и свойства S, имеющего s значений, этот индекс вычисляется как

$$\mathrm{Gini}(A,S) = 1 - \left(\sum_{i=1}^{s} \frac{|A_i|}{|A|}\right)^2.$$

Соответственно, для набора тестов A, атрибута Q, имеющего q значений, и целевого свойства S, имеющего s значений, индекс Джини вычисляется следующим образом:

$$\mathrm{Gini}(A,Q,S) = \mathrm{Gini}(A,S) - \sum_{j=1}^{q} \frac{|A_j|}{|A|}\mathrm{Gini}(A_j,S).$$

Индекс Джини отчасти компенсирует уклон критерия прироста информации в сторону выбора самых «развесистых» атрибутов. Однако в остальном они очень похожи; проведённое теоретическое исследование показало, что результаты использования индекса Джини и критерия прироста информации различаются очень редко [131].

§ 1.8. Оверфиттинг

Когда дерево принятия решений строится посредством алгоритма ID3 или аналогичного ему, полученное дерево всегда удовлетворяет *всем* имеющимся данным. Однако на практике

[1] Коррадо Джини (Corrado Gini, 1884–1965) — итальянский статистик и экономист. В своей статье 1912 года «Variabilitá e mutabilitá» [50] Джини ввёл так называемый *коэффициент Джини* — экономическую меру неравенства в распределении доходов. Коэффициент Джини показывает, насколько большая доля общего дохода или благосостояния сосредоточена в руках немногочисленного большинства. Это — один из весьма характерных примеров того, как идеи машинного обучения приходят из самых неожиданных областей; позднее мы ещё столкнёмся с применениями идей из термодинамики и биологии.

в результате часто получаются слишком детализированные деревья, и число ошибок при дальнейшем использовании построенного дерева растёт. Это явление называется *оверфиттингом* (overfitting).

Пример 1.8. Когда возникает оверфиттинг

Эффект оверфиттинга легко проиллюстрировать на следующем примере. Предположим, что игра нашего многострадального «Зенита» дома в ясную погоду вообще ни от чего больше не зависит: «Зенит» просто выигрывает в 90% случаев. Но среди исходных данных имеется одно домашнее поражение «Зенита» в ясную погоду. Тогда ID3 старательно учтёт все (не имеющие никакого отношения к сути дела) «причины» этого поражения и будет в дальнейшем предсказывать, что «Зенит» проиграет в аналогичных ситуациях.

На самом же деле «Зенит», конечно, будет выигрывать всё с той же вероятностью 90%. Таким образом, получившееся в результате работы совершенно корректного алгоритма обучения дерево будет на деле плохо справляться с задачей классификации.

Можно, конечно, просто искусственно останавливать рост дерева в глубину. Но когда, на каком этапе? Обычно, чтобы избежать оверфиттинга, используют так называемое *обрезание* (pruning), то есть сначала позволяют дереву беспрепятственно разрастись, а затем обрезают лишние ветки. Обрезание дерева в некотором узле заключается в том, что поддерево с корнем в этом узле удаляется из дерева, а в узел помещается метка с тем исходом, которых в этом узле большинство. Однако, чтобы применить такой метод, нужно сначала понять, какие именно ветки следует обрезать.

В машинном обучении часто используется стандартная процедура для самотестирования. Предположим, что у нас есть достаточно большой объём тестовых примеров. Разобьём эти исходные данные на множество, на котором дерево будет строиться (teaching set), и множество, на котором мы будем дерево тестировать (validation set). По первой части построим дерево принятия решений. Теперь у нас есть дерево и набор эталонных тестов, правильные ответы на которые нам известны. Если же «выбрасывать» часть драгоценных данных на тестирование жалко, можно разделить данные на k частей и за k проходов

использовать каждую часть по одному разу для тестирования и $k-1$ раз для обучения (так сказать, «скользящее окно»).

Теперь изложим один из возможных методов обрезания вершин. Главной мерой успеха для нас будет то, насколько хорошо дерево справляется с тестами. Соответственно, можно просто рассмотреть каждую вершину как кандидата на обрезание и проверить, станет ли сокращённое дерево лучше справляться с тестами. На каждом шаге мы будем выбирать такую вершину, обрезание которой максимально улучшает поведение дерева принятия решений. Эти вершины будем оставлять обрезанными и продолжать до тех пор, пока обрезание любой вершины не будет приводить к более слабому результату на эталонных тестах.

§ 1.9. Заключение

Мы начали эту книгу с несложного, но достаточно показательного аппарата машинного обучения — обучения деревьев принятия решений. В этой главе мы научились автоматически строить из данных модель, позволяющую аппроксимировать заданную булевскую функцию. Этой моделью стали деревья принятия решений: деревья, в которых на каждом уровне происходит разветвление по возможным значениям одного из атрибутов.

Но, конечно, многое осталось за кадром. Нам по ходу главы уже стало ясно, что разные функции имеют разную глубину оптимального дерева принятия решений. Людям, знакомым со структурной теорией сложности, а в особенности со схемной сложностью, уже, наверное, тоже ясно, что это неплохая мера сложности булевской функции: простые функции можно выразить деревом небольшой глубины, а самые сложные несколькими выборами не решишь. Такой подход действительно был развит; фактически, появились сложностные классы, связанные с деревьями принятия решений, аналогичные классическим P, NP и coNP (кстати, в контексте этой теории $P = NP \cap coNP$) [6].

Но и с точки зрения машинного обучения деревьями принятия решений дело не ограничилось. Деревья быстро превратились в *графы принятия решений* (decision graphs); в них

несколько путей от корня могут снова сойтись при помощи узладизъюнкции (в этой главе они могли только разделяться узламиконъюнкциями) [119–121]. Затем последовали и ещё более сложные конструкции; если читатель хочет выяснить, какие основанные на деревьях принятия решений конструкции сейчас рассматриваются в машинном обучении, рекомендуем заглянуть в [152, 153].

А мы будем двигаться дальше; в следующей главе нас ждёт та же задача и другой способ её решения. И другой результат.

Глава 2

Обучение концептам

Однажды все жители Леса собрались на День Рождения к Кристоферу Робину. После торжественного чаепития, большого медового торта и задувания свечек Винни-Пух, Пятачок, Кролик, Иа-Иа и все-все-все остальные остались у Кристофера Робина для дружеской беседы. Вдруг умиротворённый обедом Винни проурчал:

— Кристофер Робин, а правда, мы с тобой большие-большие друзья?

— Конечно, правда, Пух! — ответил Кристофер Робин.

— А правда, у тебя нету друга лучше меня? — спросил Пух.

— Конечно, нету, Пух! — радостно подтвердил Кристофер Робин.

В гостиной повисла зловещая тишина. Кристофер Робин растерянно оглянулся по сторонам: остальные звери казалось, изо всех сил старались не смотреть ни друг на друга, ни на Пуха, ни на мальчика. Вдруг в тишине раздалось приглушённое всхлипывание. Это не выдержал Пятачок:

— Кристофер Робин, то есть мы... мы хуже Пуха, да?

— Ну что ты, Пятачок! Что ты, Кролик, что ты, Иа-Иа! Друзей лучше вас у меня тоже нет! — сказал Кристофер Робин самым весёлым голосом, на какой только был способен.

— Но как же так? — спросил Кролик. — Разве может быть, что и Пух самый лучший, и мы самые лучшие?

— Конечно, может, глупышка! — продолжал Кристофер Робин. — Пух спросил, есть у меня друзья лучше, чем он. Ну конечно, у меня нет друзей лучше Пуха. Но и друга лучше, чем ты, Кролик, у меня тоже нету! Вы просто разные, вас нельзя сравнить друг с другом и сказать, что кто-то лучше. Вы все — мои лучшие друзья!

Вскоре все успокоились, ведь никому не хотелось верить, что Кристофер Робин их не любит. Жители Леса ещё долго вели интересную и познавательную беседу, а потом, когда за окном стемнело и стихло всё, кроме уютного трещания камина, гости Кристофера Робина по одному заснули прямо в креслах. Мальчик обвёл их взглядом, вздохнул и с грустной улыбкой пробормотал:

— Все вы у меня... максимальные.

§ 2.1. Введение

В этой короткой главе мы рассмотрим ещё один способ обучить компьютер классифицировать данные по заданному конечному числу атрибутов — так называемое *обучение концептам*. Мы решаем ту же задачу, что и в предыдущей главе: найти наиболее разумное — что часто эквивалентно наиболее короткому — описание имеющихся данных. Главное же отличие от алгоритмов классификации методом деревьев принятия решений заключается в представлении гипотез и, как следствие, в представлении результатов классификации.

Гипотеза в обучении концептам — это набор причин, из которых следует, что целевая функция равна единице (истине). Да, алгоритмы обучения концептам несимметричны — если вместо поиска гипотез, из которых следует истинность целевой функции, искать гипотезы, из которых следует её ложность, получатся совсем другие результаты. Это мы ниже используем в другом алгоритме обучения концептам. Но прежде чем заняться алгоритмами вплотную, давайте сделаем небольшой шаг в сторону и немного поговорим о бинарных отношениях и частичных порядках.

§ 2.2. Частичные порядки

В этом параграфе мы рассмотрим несколько очень простых вещей, относящихся к дискретной математике. К сожалению (и, честно говоря, к вящему изумлению), их очень нечасто включают в базовые математические курсы. Но в жизни математика и даже программиста, тем не менее, весьма полезно понимать разницу между «наибольшим» и «максимальным». Поэтому попробуем вкратце изложить основные определения этого раздела дискретной математики (тем более что они нам в этой главе пригодятся), а заинтересовавшегося читателя отсылаем к замечательной книге [175], а также к [25, 140].

Начнём издалека.

Определение 2.1. *Бинарное отношение* на множествах X и Y — это подмножество декартова произведения $X \times Y$.

Обычно вместо $(x,y) \in R$, где R — бинарное отношение, пишут $x\,R\,y$ (сравните это с $x < y$, и вы поймёте, откуда взялось такое странное обозначение).

Пример 2.1. Шекспир на большом экране

В качестве X рассмотрим множество кинорежиссёров

$$X = \{\text{Франко Дзефирелли, Лоуренс Оливье, Квентин Тарантино}\},$$

а в качестве Y — множество шекспировских трагедий

$$Y = \{\text{«Гамлет», «Ричард III», «Ромео и Джульетта»}\}.$$

Тогда бинарное отношение «режиссёр ставил фильм по пьесе» на этих множествах будет выглядеть следующим образом:

$$R = \big\{ (\text{Франко Дзефирелли, «Ромео и Джульетта»}),\ (\text{Франко Дзефирелли, «Гамлет»}),\ (\text{Лоуренс Оливье, «Гамлет»}),\ (\text{Лоуренс Оливье, «Ричард III»}) \big\} \subseteq X \times Y.$$[1]

Бинарное отношение — очень общее понятие. В качестве бинарного отношения можно представить многие математические

[1] По данным 2009 года; кто знает, может быть, Гамлету ещё предстоит вести с Горацио фирменный тарантиновский диалог...

объекты. Например, *функция* — это бинарное отношение специального вида: отношение называют *функциональным*, если для всяких $x \in X$ и $y,z \in Y$ из $x\,R\,y$ и $x\,R\,z$ следует, что $y = z$. Кстати говоря, равенство, точнее говоря, *тождество* — это тоже частный случай бинарного отношения, так называемое *диагональное отношение* Δ на $X \times X$: $(x,y) \in \Delta$ тогда и только тогда, когда $x = y$. Введём несколько (только самых-самых основных) свойств бинарных отношений.

Определение 2.2. Бинарное отношение R на множествах X и Y называется:

- *тотальным слева*, если $\forall x \in X\ \exists y \in Y$: $x\,R\,y$;
- *сюръективным* (тотальным справа), если $\forall y \in Y\ \exists x \in X$: $x\,R\,y$;
- *инъективным*, если $\forall x,z \in X, \forall y \in Y$, если $x\,R\,y$ и $z\,R\,y$, то $x = z$;
- *функциональным*, если $\forall x \in X, \forall y,z \in Y$, если $x\,R\,y$ и $x\,R\,z$, то $y = z$;
- *биективным*, если оно тотально слева, сюръективно, инъективно и функционально.

Бинарное отношение R на множестве X ($R \subseteq X \times X$) называется:

- *рефлексивным*, если $\forall x \in X\ x\,R\,x$;
- *иррефлексивным*, если $\forall x \in X$ неверно, что $x\,R\,x$;
- *симметричным*, если $\forall x,y \in X$ из $x\,R\,y$ следует $y\,R\,x$;
- *антисимметричным*, если $\forall x,y \in X$ из $x\,R\,y$ и $y\,R\,x$ следует $x = y$;
- *транзитивным*, если $\forall x,y,z \in X$ из $x\,R\,y$ и $y\,R\,z$ следует $R\,z$;
- *линейным*, если $\forall x \in X$ верно либо $x\,R\,y$, либо $y\,R\,x$ (либо они одновременно выполняются).

Конечно, столь многочисленные определения требуют конкретных примеров.

Пример 2.2. Примеры бинарных отношений

Введённое нами в примере 2.1 отношение «режиссёр ставил фильм по пьесе» не было

- ни тотальным слева (Тарантино не ставил Шекспира),

— ни функциональным (Дзефирелли и Оливье экранизировали по две из трёх пьес из множества Y),
— ни инъективным («Гамлета» экранизировали два из трёх режиссёров множества X),
— ни, разумеется, биективным.

Из введённых нами свойств оно обладало только свойством сюръективности: любую из пьес множества Y экранизировал хотя бы один режиссёр множества X.

Отношение строгого порядка $<$ на множестве вещественных чисел $\mathbb{R}$ является:

— иррефлексивным (а вот отношение нестрогого порядка $\leqslant$ было бы рефлексивным);
— антисимметричным ($\leqslant$ тоже было бы антисимметричным);
— транзитивным (как и $\leqslant$);
— линейным (как и $\leqslant$).

Есть два особенно часто употребимых класса бинарных отношений на $X \times X$ (понятно, что функции употребимы не реже, но их редко рассматривают как бинарные отношения). Первый из них — рефлексивные симметричные транзитивные отношения; они называются *отношениями эквивалентности* (например, равенство чисел — типичное отношение эквивалентности). О втором же классе поговорим подробнее.

Определение 2.3. *Частичный порядок* — это транзитивное антисимметричное бинарное отношение. Рефлексивный частичный порядок называется *нестрогим*; напротив, иррефлексивный частичный порядок называется *строгим*.

Приведём классический пример конечного частично упорядоченного множества: рассмотрим множество подмножеств конечного множества X, на котором введён частичный порядок $\subseteq$ (включение). Проще говоря, из двух множеств A и B бóльшим считается то, которое *полностью* включает другое. Если же и разность $A \setminus B$, и разность $B \setminus A$ непусты, то такие подмножества считаются несравнимыми.

На рис. 2.1 изображено множество подмножеств множества из трёх элементов $X = \{x,y,z\}$. При этом подмножества упорядочены снизу вверх: на одном уровне находятся несравнимые

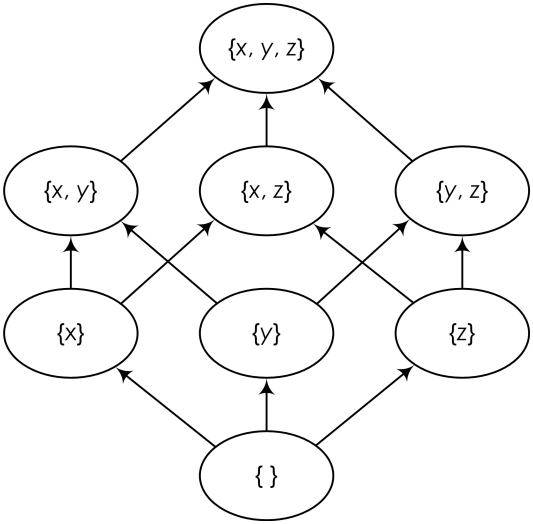


Рис. 2.1. Множество подмножеств $X = \{x,y,z\}$

подмножества (т.е. такие элементы a и b, что в нашем частичном порядке $\subseteq$ ни $a \subseteq b$, ни $b \subseteq a$), а из меньших подмножеств в бóльшие ведут стрелки. Такую диаграмму можно нарисовать для любого частичного порядка; она называется *диаграммой Хассе*.

Множество, снабжённое частичным порядком, называется *частично упорядоченным множеством*. Слова эти иногда сокращают до *ч.у.м.* или просто *чум*, а в англоязычных текстах общепринят термин *poset* (от слов «partially ordered set»).

На этом месте можно было бы долго развивать теорию частично упорядоченных множеств. Стоит упомянуть слово *решётка* (частный случай частично упорядоченного множества), как оно сразу же потянет за собой понятие *алгебры*, а там уже недалеко до универсальной алгебры, теории булевых алгебр, и всё это непосредственно связано с логикой вообще и теорией моделей в частности... но здесь любопытствующий читатель сможет найти много интересного в [25], а нам потребуется буквально пара определений.

Определение 2.4. *Наибольшим* элементом множества X относительно частичного порядка $\succ$ называется такой $x \in X$,

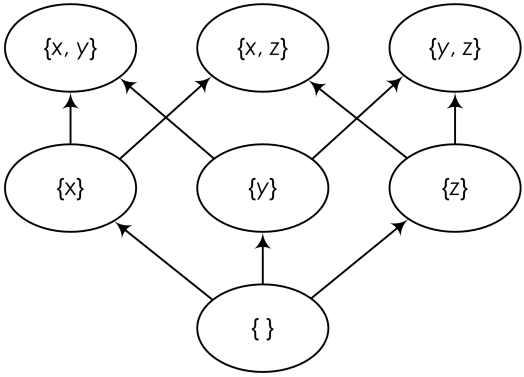


Рис. 2.2. Множество подмножеств $X = \{x,y,z\}$ без наибольшего элемента

что $\forall y \in X\ y \succ x$ или $y = x$.[1] Аналогично, $x \in X$ называется *наименьшим*, если $\forall y \in X\ x \succ y$ или $x = y$.

Максимальным элементом множества X относительно частичного порядка $\succ$ называется такой $x \in X$, что если $x \succ y$ для какого-либо $y \in X$, то $y = x$:

$$(y \in X \wedge x \succ y) \Rightarrow y = x.$$

Аналогично, $x \in X$ называется *минимальным*, если для всякого $y \in X$ из $y \succ x$ следует $y = x$.

Следует чётко осознавать разницу между максимальными и наибольшими элементами: наибольший — это тот, который *больше всех*, а максимальный — тот, *больше которого нету*. Наибольший элемент может быть только один (если x и y — наибольшие, то по свойству y из $x \preceq y$, что верно по свойству x, следует $x = y$). А максимальных может быть много. Чтобы далеко не ходить, рассмотрим тот же пример — множество подмножеств $X = \{x,y,z\}$, упорядоченное по включению, — но выкинем из него наибольший элемент $\{x,y,z\}$. В получившемся частично упорядоченном множестве окажется сразу три максимальных элемента: $\{x,y\}$, $\{x,z\}$ и $\{y,z\}$ (см. рис. 2.2).

[1] Случай равенства приходится выделять, потому что порядок может оказаться строгим.

На этом уже можно было бы и завершить наш краткий экскурс в дискретную математику и вернуться к алгоритмам машинного обучения... но есть одно лирическое отступление, которое очень хочется сделать на этом месте. Введённые нами определения частичных порядков позволят определить одну очень красивую конструкцию дискретной математики: *функцию Мёбиуса*. Для дальнейшего понимания следующий параграф совершенно не важен, поэтому читатель, желающий поскорее перейти к сути обучения концептов, может переходить к § 2.4.

§ 2.3. Лирическое отступление: функция Мёбиуса

Возможно, вы встречали функцию Мёбиуса[1] на натуральных числах:

$$\mu(n) = \begin{cases} 1, & n = p_1 \dots p_{2l}, \\ -1, & n = p_1 \dots p_{2l+1}, \\ 0, & p^2 \mid n, \end{cases}$$

где $n \in \mathbb{N}$, а p_i — простые делители n. Другими словами, $\mu(n)$ равна нулю, если в разложении n на простые множители встречается квадрат какого-нибудь простого числа, равна единице, если квадратов нет и разных простых чисел в разложении n чётное число, и равна -1, если число простых чисел в разложении нечётно.

У этой функции много интересных свойств и применений. Главное свойство — формула обращения Мёбиуса (см., например, [175]).

[1] Август Фердинанд Мёбиус (August Ferdinand Möbius, 1790–1868) — немецкий математик и астроном. В молодости он учился астрономии и математике у Гаусса, который, мягко говоря, без большого желания брал учеников, а затем перешёл под руководство Пфаффа (того самого, в чью честь названы пфаффианы). У Мёбиуса были важные работы по астрономии (его диссертация называлась «О покрытии неподвижных звёзд»), а в математике он занимался теорией чисел (отсюда и функции Мёбиуса) и проективной геометрией (отсюда преобразования Мёбиуса; именно Мёбиус, кстати, ввёл однородные координаты). Мёбиус известен каждому как автор «ленты Мёбиуса» — неориентируемой двумерной поверхности с одним краем и одной стороной; впрочем, как это часто бывает, приоритет этого открытия принадлежит не Мёбиусу, в честь которого оно названо, а другому немецкому математику, Иоганну Листингу.

Предложение 2.1. *Для арифметических (то есть заданных на множестве натуральных чисел) функций* f *и* g

$$g(n) = \sum_{d|n} f(d) \quad \Leftrightarrow \quad f(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) g(d).$$

Аналогичный факт верен и не обязательно в целых числах:

$$g(x) = \sum_{n \leqslant x} f\left(\frac{x}{n}\right) \quad \Leftrightarrow \quad f(x) = \sum_{n \leqslant x} \mu(n) g\left(\frac{x}{n}\right).$$

Иначе говоря, функция Мёбиуса позволяет обратить сумму по делителям. В частности, если подставить вместо f саму функцию μ, получится другое важное свойство функции Мёбиуса:

$$\sum_{d|n} \mu(d) = \begin{cases} 1, & n = 1, \\ 0, & n > 1. \end{cases}$$

За следующим примером неподготовленному читателю может оказаться трудно проследить во всех деталях. Однако если принять на веру некоторые базовые вещи, понять суть всё же можно.

Пример 2.3. Газ из примонов и гипотеза Римана

Функция Мёбиуса находит совершенно неожиданное применение... здесь хочется написать «в физике», но это было бы некорректно: всё-таки модель сугубо математическая, и физические аналогии появляются уже после математической постановки. Но аналогии эти чрезвычайно интересные.

Давайте рассмотрим модель так называемого *свободного риманова газа*, то есть газа, состоящего из частиц, так называемых *примонов* (primons, от слова, разумеется, «prime» — «простой»), которые не взаимодействуют между собой и имеют уровни энергии $\log p$ для каждого простого числа p.

Эту модель можно мотивировать, рассмотрев одно из самых важных и базовых понятий современной физики — понятие *спектра*. Мы не будем углубляться в основы физики, хотя, наверное, стоило бы написать отдельную книгу «Физика для информатика–теоретика», так много физических концепций и трюков применяется в теоретической информатике и, в частности, в машинном обучении. Вспомним только сугубо математический аспект: спектр кольца $\mathrm{Spec}(R)$ — это множество его простых идеалов. Связь с сугубо физическим понятием спектра (множеством частот, с которыми может колебаться система)

здесь хоть и неочевидна, но всё же есть: частный случай простых идеалов, *максимальные идеалы* — это возможные ядра гомоморфизмов из кольца в какое-либо поле (для $\mathbb{Z}$ все простые идеалы являются максимальными); коммутативное кольцо — это множество функций на том или ином пространстве, и эти самые пространства получаются как множества гомоморфизмов кольца в то или иное поле; а спектр некоторой алгебры (C^*-алгебры, если быть точным) функций A в математической физике — это как раз множество гомоморфизмов $\lambda : A \to \mathbb{C}$, то есть, грубо говоря, множество возможных наборов значений этих самых функций. Множество наборов значений пространства функций — это уже спектр в его самом физическом смысле: множество частот системы — это множество значений функционала энергии этой системы (частота и энергия в квантовой механике отличаются только на постоянную Планка), так называемого *гамильтониана*.

У свободного риманова газа каждая частица (примон) имеет уровни энергии $\log p$, соответствующие простым числам:

$$(\log 2, \log 3, \log 5, \log 7, \ldots).$$

Частицы не взаимодействуют, то есть общая энергия системы (гамильтониан) является суммой энергий частиц. Значит, общий гамильтониан системы определяется целым числом:

$$|n\rangle = |p_1, p_2, p_3, \ldots\rangle = |p_1\rangle|p_2\rangle|p_3\rangle \ldots,$$

ведь суммарная энергия равна сумме логарифмов, а каждое натуральное число однозначно соответствует некоторому произведению простых чисел.

Так вот, это всё разумно, если частицы являются бозонами (частицами с целым спином: фотон, мезоны). А если мы хотим рассмотреть газ из фермионов (частиц с половинным спином: протоны, электроны), на нашем пути возникает *принцип Паули*, он же принцип запрета: два тождественных фермиона не могут находиться в одном квантовом состоянии. О чём это говорит с математической точки зрения? Правильно, о том, что в разложении целого числа запрещаются квадраты простых. Оператор, который разделяет бозоны и фермионы в свободном римановом газе — это как раз функция Мёбиуса (подробнее об этом см. [146]).

Кстати, эта модель риманова газа вообще очень интересна математически. Из статистической физики известно, что вероятность того, что энергия системы будет равна E, пропорциональна $e^{-\beta E}$, где β — некоторая константа, зависящая от *температуры* (вот что такое температура на самом деле!). Чтобы получить собственно вероятности,

эти числа нужно нормализовать — разделить каждое из них на сумму всех чисел, чтобы сумма полученных отношений равнялась единице. Нормировочная константа, то есть сумма всех чисел вида $e^{-\beta E}$, в физике называется *статистической суммой* (partition function) и играет важную роль: через неё выражаются разные термодинамические параметры.

В модели риманова газа статистическая сумма будет равна

$$\sum_{n\in\mathbb{N}} e^{-\beta E(n)} = \sum_{n\in\mathbb{N}} e^{-\beta \ln n} = \sum_{n\in\mathbb{N}} n^{-\beta},$$

то есть в точности дзета-функции Римана! Отсюда возникают идеи сугубо физического доказательства гипотезы Римана... но в это мы углубляться не будем, а если читатель заинтересовался — обращайтесь в [22].

Однако главная суть функции Мёбиуса — не в её арифметической постановке, а в более общей ситуации, когда функцию Мёбиуса вводят на произвольном частично упорядоченном множестве. Начнём с определения объекта, на котором все эти функции будут действовать.

Определение 2.5. *Интервал* $[a,b]$ в частичном упорядоченном множестве X для некоторых $a \preceq b \in X$ — это множество вида

$$[a,b] = \{x : a \preceq x \preceq b\} \subseteq X.$$

Частично упорядоченное множество называется *локально конечным*, если в нём конечен всякий интервал.

Локально конечное множество совершенно не обязано быть конечным.

Пример 2.4. Локально конечное бесконечное множество

Натуральные числа $\mathbb{N}$ с естественным порядком локально конечны, потому что каждый интервал $[n, m]$ содержит конечное число натуральных чисел. Но при этом само множество $\mathbb{N}$ счётно.

Легко привести и пример локально конечного несчётного множества. Рассмотрим, например, $\mathbb{R}$ с отношением делимости нацело: $a \preceq b$ тогда и только тогда, когда $b = an$ для некоторого $n \in \mathbb{N}$. Тогда в интервале от a до b может поместиться не более n чисел, но само множество при этом несчётно.

Определение 2.6. Для любого локально конечного частично упорядоченного множества X и любого коммутативного кольца с единицей R[1] определим *алгебру инцидентности*. Её элементы — функции f, которые действуют на множестве интервалов: $f([a,b]) \in R$ (в дальнейшем будем использовать сокращённое обозначение $f(a,b) = f([a,b])$). Сложение и умножение на скаляр в этой алгебре вводятся покомпонентно, а умножение функций — как свёртка

$$(f * g)(a,b) = \sum_{a \preceq x \preceq b} f(a,x)g(x,b).$$

Алгебра инцидентности — один из самых базовых, но вместе с тем и самых интересных объектов дискретной математики. Например, попробуйте догадаться — какая будет единица в этой алгебре?

Предложение 2.2. Дельта-функция, *определённая равенством*

$$\delta(a,b) = \begin{cases} 1, & a = b, \\ 0, & a \prec b, \end{cases}$$

является единицей алгебры инцидентности.

Доказательство. Нужно проверить, что при умножении на дельта-функцию хоть слева, хоть справа элементы алгебры инцидентности не меняются. Давайте умножим δ на некоторую функцию f:

$$(f * \delta)(a,b) = \sum_{a \preceq x \preceq b} f(a,x)\delta(x,b) = f(a,b),$$

$$(\delta * f)(a,b) = \sum_{a \preceq x \preceq b} \delta(a,x)f(x,b) = f(a,b),$$

так как δ равна нулю на всех интервалах, по которым ведётся суммирование, кроме одного. □

Ещё одна интересная функция, присутствующая в любой алгебре инцидентности — это дзета-функция $\zeta(a,b) = 1$ (т.е. константа). Умножение на ζ — это дискретный аналог интегрирования; в результате получается сумма по всем промежуточным

[1] Если вы не знаете, что такое кольцо, можете считать, что R — это просто вещественные числа. Или целые. Или комплексные.

интервалам:

$$(f * \zeta)(a,b) = \sum_{a \preceq x \preceq b} f(a,x)\zeta(x,b) = \sum_{a \preceq x \preceq b} f(a,x),$$

$$(\zeta * f)(a,b) = \sum_{a \preceq x \preceq b} \zeta(a,x)f(x,b) = \sum_{a \preceq x \preceq b} f(x,b),$$

Так вот, оказывается, что ζ можно обратить. Обратная к функции ζ — это и есть функция Мёбиуса μ.

Пример 2.5. Функция Мёбиуса на натуральных числах

Классическая функция Мёбиуса на числах получается, если рассмотреть частично упорядоченное множество натуральных чисел, упорядоченных по делимости ($a \preceq b$, если a делит b). Вспомним определение функции Мёбиуса, которое мы ввели в начале параграфа:

$$\mu(n) = \begin{cases} 1, & n = p_1 \dots p_{2l}, \\ -1, & n = p_1 \dots p_{2l+1}, \\ 0, & p^2 \mid n, \end{cases}$$

где n — натуральное число, а p_i — его простые делители. Нам нужно ввести определение функции Мёбиуса на интервалах; не мудрствуя лукаво, положим $\mu(a,b) = \mu(b/a)$. Заметим, что в этом случае *интервалом* $[a,b]$ для чисел a и b, где b делится на a, будет множество чисел, которые делятся на a и делят b.

А теперь попробуем умножить μ на ζ в алгебре инцидентности:

$$(\mu * \zeta)(a,b) = \sum_{a \preceq x \preceq b} \mu(a,x)\zeta(x,b) = \\ = \sum_{a \preceq x \preceq b} \mu(a,x) = \sum_{a|x|b} \mu\left(\frac{x}{a}\right) = \sum_{y|\frac{b}{a}} \mu(y)$$

(в последнем равенстве мы просто сделали замену переменных). Теперь по следствию из предложения 2.1 очевидно, что

$$(\mu * \zeta)(a,b) = 1$$

тогда и только тогда, когда $a = b$, а в противном случае $(\mu*\zeta)(a,b) = 0$. Иначе говоря,

$$\mu * \zeta = \delta,$$

а δ — единица нашей алгебры инцидентности. Совершенно точно так же проверяется, что $\zeta * \mu = \delta$, а эти два факта вместе как раз и означают, что $\mu = \zeta^{-1}$.

Очень интересный пример получится, если взять натуральные числа с естественным порядком. Тогда функция Мёбиуса превратится в разностный оператор Δ:

$$\mu(x,y) = \begin{cases} 1, & y - x = 0, \\ -1, & y - x = 1, \\ 0, & y - x > 1. \end{cases}$$

Мы оставляем читателю в качестве несложного упражнения проверить, что так определённая функция μ действительно обратна функции ζ. Интуитивно понятно: оператор конечной разности — это оператор дискретного дифференцирования, и поэтому он будет обратным к дискретному интегрированию ζ.

А если в том же частично упорядоченном множестве $\mathbb{N}$ приравнять значения функций $f(a,b) = f(a+c,b+c)$, то функции будут соответствовать последовательностям элементов из R. При таком подходе из алгебры инцидентности получится кольцо формальных степенных рядов с коэффициентами из R (нужно только ограничиться рассмотрением начинающихся в нуле интервалов; в дальнейшем $f(a) = f(0,a)$). Ведь свёртка двух функций в точности превратится в произведение рядов:

$$(f * g)(a) = (f * g)(0,a) = \sum_{i=0}^{a} f(0,i)g(i,a) = \sum_{i=0}^{a} f(i)g(a-i).$$

При таком подходе δ — это формальный ряд $(1,0,0,\ldots)$, т.е. единица (по-прежнему $\delta(a,b) = 1$ тогда и только тогда, когда $a = b$). А ζ — это ряд $(1,1,1,\ldots)$, и если расписать формальный степенной ряд, получится

$$\zeta = 1 + x + x^2 + x^3 + \ldots.$$

Но мы же умеем обращать такой ряд: $1 + x + x^2 + \ldots$ — это геометрическая прогрессия со знаменателем x, то есть (в лучших традициях Эйлера закроем глаза на вопросы сходимости)

$$1 + x + x^2 + \ldots = \frac{1}{1-x}.$$

Тогда обратный к ζ ряд μ — это ряд вида

$$\mu = (1,-1,0,0,\ldots),$$

в точности так, как получалось два абзаца назад. Вот такая занимательная функция Мёбиуса.

Можно рассмотреть и другой, не менее важный пример — булеву алгебру подмножеств некоторого множества X. В этом случае (доказывать этот факт мы не будем; см., например, [175]) функция Мёбиуса будет равна

$$\mu(T, S) = (-1)^{|S \setminus T|}.$$

Это называется *принципом включения–исключения*: для всякой функции f, определённой на подмножествах S,

$$g(S) = \sum_{|T \subseteq S|} f(T) \quad \Leftrightarrow \quad f(S) = \sum_{T \subseteq S} (-1)^{|S \setminus T|} g(T).$$

Пример применения этого принципа можно найти, например, в элементарной теории вероятностей. Формула вероятности пересечения двух событий:

$$p(X \cup Y) = p(X) + p(Y) - p(X \cap Y),$$

является простейшим частным случаем формулы включения–исключения. Её легко обобщить на произвольное конечное число событий (попробуйте доказать следующее предложение по индукции), и получится в точности принцип включения–исключения для частично упорядоченного по включению множества событий.

Предложение 2.3. *Пусть* $X_1, X_2, \ldots, X_i$ — *события. Тогда*

$$p\left(\bigcup_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} p(X_i) - \sum_{1 \leqslant i < j \leqslant n} p(X_i \cap X_j) +$$

$$+ \sum_{1 \leqslant i < j < k \leqslant n} p(X_i \cap X_j \cap X_k) - \ldots + (-1)^{n-1} p(X_1 \cap X_2 \cap \ldots \cap X_i).$$

На этом, пожалуй, завершим лирические отступления; желающие могут найти массу не менее интересных теорем и приложений функции Мёбиуса в [175] и других источниках. А мы перейдём собственно к обучению концептам.

Таблица 2.1. Как играет «Зенит»

Соперник	Играет	Лидеры	Дождь	Победа
Выше	Дома	На месте	Да	Нет
Выше	Дома	На месте	Нет	Да
Выше	Дома	Пропускают	Нет	Да
Ниже	Дома	Пропускают	Нет	Да
Ниже	В гостях	Пропускают	Нет	Нет
Ниже	Дома	Пропускают	Да	Да
Выше	В гостях	На месте	Да	Нет

§ 2.4. Алгоритм Find-S

Вернёмся к примеру 1.1. Предположим, что нам нужно классифицировать удачные и неудачные игры «Зенита» по заданной таблице результатов (см. табл. 2.1).

Для игр «Зенита» множество атрибутов выглядит так:

$$\langle \text{Соперник, Играет, Лидеры, Дождь} \rangle .$$

Приведём теперь несколько примеров гипотез, описывающих ситуации, в которых «Зенит» побеждает. В записи гипотез вопросительные знаки означают, что данный атрибут не специфицирован, то есть на месте вопросительного знака может стоять любое значение атрибута; а пустое множество, напротив, означает, что здесь не может быть никакого значения, гипотеза всё равно будет отвергнута:

$$\langle \text{Выше, Дома, ?, ?} \rangle ,$$

$$\langle \text{?, В гостях, На месте, ?} \rangle ,$$

$$\langle \text{?, ?, ?, ?} \rangle ,$$

$$\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle .$$

Отметим, что если хоть один элемент гипотезы равен $\emptyset$, вся гипотеза целиком эквивалентна пустой гипотезе $\langle \emptyset, \ldots, \emptyset \rangle$ (то есть «Зениту» уже ничто не поможет).

Следующий этап — каким-либо образом ранжировать гипотезы. Оказывается, на гипотезах есть естественный порядок, от общего к частному.

Таблица 2.2. Как выигрывает «Зенит»

Соперник	Играет	Лидеры	Дождь	Победа
Выше	Дома	На месте	Нет	Да
Выше	Дома	Пропускают	Нет	Да
Ниже	Дома	Пропускают	Нет	Да
Ниже	Дома	Пропускают	Да	Да

Определение 2.7. Гипотеза h_1 называется *более общей*, чем гипотеза h_2 (обозначаем $h_1 \succ h_2$), если для всякого тестового примера d из $h_2(d) = 1$ следует $h_1(d) = 1$.

Очевидно, что гипотеза $\langle\emptyset, \ldots, \emptyset\rangle$ является самой частной из всех гипотез, а гипотеза $\langle ?, \ldots, ?\rangle$ — самой общей. Но порядок, несмотря на то, что в нём есть максимальный и минимальный элементы, всё равно частичный, а не общий. Например, гипотезы $h_1 = \langle$Выше, Дома, ?, ?$\rangle$ и $h_2 = \langle$?, В гостях, На месте, ?$\rangle$ несравнимы: например, для тестового примера

$$\langle\text{Выше, Дома, Пропускают, Да}\rangle$$

$h_1 = 1$, а $h_2 = 0$, в то время как для тестового примера

$$\langle\text{Ниже, В гостях, На месте, Да}\rangle,$$

напротив, $h_1 = 0$ и $h_2 = 1$.

Идея алгоритма Find-S очень проста: нужно начать с самой частной гипотезы, а затем обобщать её так, чтобы она включала в себя все тестовые примеры. На каждом шаге, если текущая гипотеза h неправильно классифицирует пример ($h(d) = 0$), нужно искать минимальную h' из тех, для которых $h' \succ h$ и $h'(d) = 0$. Иными словами, нужно просто заменять неподходящие значения на вопросительные знаки. В результате получится максимально частная гипотеза, отвечающая всем тестовым данным.

Отметим, что негативные тестовые примеры вообще игнорируются, т.е. реально для алгоритма Find-S интерес представляет только часть табл. 2.1, представленная в табл. 2.2.

Схема алгоритма представлена на рис. 2.3.

Давайте подробно разберём работу алгоритма в нашем случае; для краткости рассмотрим в следующем примере только три первых тестовых примера.

FindS(D):

1. $h := \langle \emptyset, \ldots, \emptyset \rangle$.
2. Для всех $d \in D^+$:
   a) Если $h(d) = 0$:
      (i) $h' := h$.
      (ii) Для каждого атрибута a, если $h[a] \neq d[a]$, то $h'[a] := ?$.
      (iii) $h := h'$.
3. Выдать h.

Рис. 2.3. Алгоритм Find-S

Пример 2.6. Работа алгоритма Find-S

Возьмём игры «Зенита» и первые три тестовых примера, представленных в табл. 2.2 (они там только положительные).

Мы начинаем с гипотезы

$$h = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle.$$

Поступает первый положительный тестовый пример из таблицы 2.2:

$$\langle \text{Выше, Дома, На месте, Нет} \rangle.$$

h выдаёт ноль, значит, гипотезу надо обобщать. Максимально частная гипотеза просто совпадёт с этим тестовым примером:

$$h = \langle \text{Выше, Дома, На месте, Нет} \rangle.$$

Поступает второй пример:

$$\langle \text{Выше, Дома, Пропускают, Нет} \rangle.$$

И снова h выдаёт 0. Максимально частная гипотеза, объединяющая h и этот пример, обобщит значение атрибута Лидеры:

$$h = \langle \text{Выше, Дома, ?, Нет} \rangle.$$

И, наконец, третий пример

$$\langle \text{Ниже, Дома, Пропускают, Нет} \rangle$$

снова удостоверит нас, что текущая гипотеза несостоятельна. Максимально частная гипотеза, объединяющая h и этот пример, обобщит значение атрибута Соперник. В итоге мы получаем гипотезу, объясняющую все тестовые примеры:

$$h = \langle ?, \text{Дома}, ?, \text{Нет} \rangle.$$

Find-S — очень простой и быстрый алгоритм. Его легко реализовать (сейчас мы к этому перейдём), и время его работы линейно зависит от числа тестовых примеров: к каждому из них мы обращаемся один раз. Главный его недостаток в том, что Find-S совсем не выразительный — у него очень бедное множество гипотез. Чтобы классифицировать можно было при помощи Find-S, нужно, чтобы целевая функция выражалась фактически одной веткой дерева принятия решений, дизъюнкции не разрешаются. Но зато, если всё же выражается, то полученный результат будет правильно классифицировать и негативные примеры тоже (т.к. результат максимально частный).

§ 2.5. Реализация алгоритма Find-S

Мы рассмотрим пример реализации алгоритма Find-S на языке C#. Начнём с двух достаточно тривиальных вспомогательных классов. Первый из них описывает тестовый пример, предоставляя интерфейс для создания и дальнейшего использования примеров.

Листинг 2.1. Find-S на C#: класс TestSample

```
public class TestSample {
    private bool[] _attributes;
    private bool _value;

    public TestSample(bool[] attributes, bool val) {
        _attributes = attributes;
        _value = val;
    }
    public bool GetValue() {
        return _value;
    }
    public bool GetAttribute(uint num) {
        if ( num < _attributes.Length ) return _attributes[num];
        throw new IndexOutOfRangeException();
    }
    public uint GetNumberOfAttributes() {
        return (uint)_attributes.Length;
    }
}
```

Второй класс реализует гипотезу; он позволяет задавать значения атрибутов, выбирая их из перечисляемого типа, в котором заданы четыре значения: ∅, 0, 1 и ?.

Листинг 2.2. Find-S на C#: класс Hypothesis

```
public class Hypothesis {
    public enum AttrValue {
        NONE,
        ONE,
        ZERO,
        ANY
    };
    private AttrValue[] _attributes;

    public Hypothesis(uint num) {
        _attributes = new AttrValue[num];
        for (uint i = 0; i < num; i++) {
            _attributes[i] = AttrValue.NONE;
        }
    }

    public void SetAttrValue(uint num, AttrValue val)   {
        if ( num < _attributes.Length ) {
            _attributes[num] = val;
        }
    }

    public AttrValue GetAttrValue(uint num) {
        if ( num < _attributes.Length ) {
            return _attributes[num];
        }
        throw new IndexOutOfRangeException();
    }

    public uint GetNumberOfAttributes() {
        return (uint)_attributes.Length;
    }
}
```

Третий листинг содержит код основного класса. Именно здесь происходит собственно работа алгоритма Find-S: описанный в листинге 2.3 класс FindSWrapper по заданному набору тестовых примеров вычисляет гипотезу.

Листинг 2.3. Find-S на C#: класс FindSWrapper

```
public class FindSWrapper {
    static public Hypothesis FindS(ArrayList testSamples) {
        // Отфильтровываем только позитивные примеры
        ArrayList positiveTests = new ArrayList();
        foreach ( TestSample test in testSamples ) {
            if ( test.GetValue() ) {
                positiveTests.Add( test );
            }
        }
        if ( positiveTests.Count > 0 ) {
            IEnumerator it = positiveTests.GetEnumerator();
            it.MoveNext();
            TestSample firstSample = (TestSample)it.Current;
            uint numOfAttributes =
              firstSample.GetNumberOfAttributes();
            Hypothesis hyp = new Hypothesis( numOfAttributes );

            // Инициализируем гипотезу первым тестовым примером
            for ( uint i = 0; i < numOfAttributes; i++) {
                if ( firstSample.GetAttribute(i) ) {
                    hyp.SetAttrValue(i, Hypothesis.AttrValue.ONE);
                } else {
                    hyp.SetAttrValue(i, Hypothesis.AttrValue.ZERO);
                }
            }

            // Подправляем гипотезу дальнейшими примерами
            foreach ( TestSample test in positiveTests ) {
                for ( uint i = 0; i < numOfAttributes; i++ ) {
                    Hypothesis.AttrValue hypVal =
                      hyp.GetAttrValue(i);
                    if ( hypVal != Hypothesis.AttrValue.ANY ) {
                        bool bHypVal =
                          ( hypVal == Hypothesis.AttrValue.ONE );
                        if ( bHypVal != test.GetAttribute(i) ) {
                            hyp.SetAttrValue(
                              i, Hypothesis.AttrValue.ANY );
                        }
                    }
                }
            }
            return hyp;
        }
        return new Hypothesis(0);
    }
}
```

И, наконец, по традиции приведём реализацию основного класса программы; на этот раз мы реализовали консольный интерактивный ввод тестовых примеров с клавиатуры (примеры чтения данных из файла уже приводились в большом количестве, хоть и для других языков программирования).

Листинг 2.4. Find-S на C#: класс ConsoleTester

```
class ConsoleTester
{
    [STAThread]
    static void Main(string[] args) {
        ArrayList tests = new ArrayList();
        uint numOfParameters;
        uint numOfTests;
        string tmp;
        Console.Write("Enter the number of
                       parameters in goal function: ");
        tmp = Console.ReadLine();
        numOfParameters = uint.Parse(tmp);
        Console.Write("Enter the number of tests: ");
        tmp = Console.ReadLine();
        numOfTests = uint.Parse(tmp);
        Console.Write("Test input format: ");
        for ( uint i = 0; i < numOfParameters; i++) {
            Console.Write("p");
        }
        Console.WriteLine(" v");
        Console.WriteLine("p - parameter value (0 or 1).");
        Console.WriteLine("v - target function
                           value (0 or 1).");
        for ( uint i = 0; i < numOfTests; i++) {
            bool val;
            bool[] attributes = new bool[numOfParameters];
            Console.Write("Enter test parameters: ");
            tmp = Console.ReadLine();
            for (int j = 0;
                 j < numOfParameters && j < tmp.Length;
                 j++) {
                attributes[j] = tmp[j] == '1';
            }
            Console.Write("Enter test goal function value: ");
            tmp = Console.ReadLine();
            val = tmp[0] == '1';
                tests.Add( new TestSample(attributes, val) );
        }

        // вызываем реализацию FindS
```

```
        Hypothesis globalHypothesis =
          FindSWrapper.FindS(tests);

        Console.WriteLine("Global hypothesis: ");
        for (uint i = 0;
             i < globalHypothesis.GetNumberOfAttributes();
             i++) {
            switch(globalHypothesis.GetAttrValue(i)) {
                case Hypothesis.AttrValue.ANY:
                    Console.Write("?");
                    break;
                case Hypothesis.AttrValue.ONE:
                    Console.Write("1");
                    break;
                case Hypothesis.AttrValue.ZERO:
                    Console.Write("0");
                    break;
                case Hypothesis.AttrValue.NONE:
                    Console.Write("#");
                    break;
                default:
                    break;
            }
        }
        Console.ReadLine();
    }
}
```

Конечно, мы привели далеко не самый экономичный с точки зрения количества строк кода пример реализации алгоритма Find-S. Читатель может попробовать переписать эту программу на Perl или Ruby и убедиться, что всё то же самое можно сделать и компактнее (правда, в программировании, особенно промышленном, краткость — очень дальняя родственница таланта).

§ 2.6. Алгоритм исключения кандидатов

В предыдущем параграфе мы узнали, что алгоритм Find-S позволяет найти наиболее частную гипотезу, совместную с положительными примерами. Очевидным образом напрашивается двойственная задача: найти наиболее *общую* гипотезу, совместную с *негативными* примерами.

Таблица 2.3. «Зенит» для алгоритма исключения кандидатов

Соперник	Играет	Лидеры	Дождь	Победа
Выше	Дома	На месте	Нет	Да
Ниже	В гостях	Пропускают	Нет	Нет
Ниже	Дома	Пропускают	Да	Нет
Выше	Дома	Пропускают	Нет	Да

Кроме того, Find-S позволяет найти только одну из гипотез, совместных с имеющимися данными, а их на самом деле может быть очень много. Соответственно, нужно найти разумный способ описания множества *всех* гипотез, совместных с тестовыми примерами.

Оказывается, что решение первой из этих задач является важным шагом на пути к решению второй. Если мы для одних и тех же данных найдём одновременно и наиболее общую гипотезу, совместную с негативными примерами, и наиболее частную гипотезу, совместную с позитивными примерами, то, казалось бы, они должны полностью описывать множество всех совместных с данными гипотез вообще.

Сложность заключается в том, что порядок, который мы ввели в определении 2.7, частичный. А это значит, что у множества может быть несколько максимумов и несколько минимумов; но в остальном всё верно. Итак, нам хочется найти два множества: множество минимальных относительно $\succ$ гипотез (наиболее частных), совместных с данными, и множество максимальных (наиболее общих). Таким образом мы найдём «границы» всего множества допустимых гипотез: любая другая гипотеза, совместная с тестовыми данными, будет меньше либо равна одной из гипотез верхней границы и больше либо равна одной из гипотез нижней границы.

Алгоритм исключения кандидатов (candidate elimination) устроен именно так. Он поддерживает два множества: G — множество максимальных (общих) гипотез, и S — множество минимальных (частных) гипотез. Когда на вход поступает новый пример d, есть две возможности в зависимости от значения целевой функции t_d.

$t_d = 1$. В этом случае, если оказывается, что какая-либо гипотеза из G неверно классифицирует d, её нужно удалить. А если неверно классифицируют гипотезы из S, их нужно соответственно обобщить. Но обобщить так, чтобы для каждой из них оставалась минимальной хоть какая-то из гипотез из G. Для этого иногда приходится «размножать» гипотезу; собственно, так и возникают множества с несколькими максимумами из начальной позиции $\langle ?, \ldots, ? \rangle$.

$t_d = 0$. В этом случае, если оказывается, что гипотеза из S неверно классифицирует d, её нужно удалить. А если неверно классифицируют гипотезы из G, их нужно соответственно специализировать (добавить все минимальные специализации, совместные с d и такие, чтобы была соответствующая гипотеза в h). При этом, опять же, одна гипотеза может превратиться в несколько.

Алгоритм формально представлен на рис. 2.4. Стоит ещё раз подчеркнуть, что результатом операций max и min может оказаться целое множество гипотез, а не одна.

Пример 2.7. Работа алгоритма исключения кандидатов

Возьмём игры «Зенита» и несколько тестовых примеров, как положительных, так и отрицательных (см. табл. 2.3). Мы начинаем с тривиальных гипотез

$$H = \{\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\}, \quad G := \{\langle ?, ?, ?, ? \rangle\}.$$

Первый пример (здесь вышеупомянутые t_d — это значения целевой функции, то есть значения, записанные в последнем столбце):

$$\langle \text{Выше, Дома, На месте, Нет, Да} \rangle.$$

H[0] выдаёт 0, значит, надо обобщать. Максимально частная гипотеза просто совпадёт с этим тестовым примером, и гипотезы примут вид

$$H = \{\langle \text{Выше, Дома, На месте, Нет} \rangle\}, \quad G := \{\langle ?, ?, ?, ? \rangle\}.$$

Второй пример — отрицательный:

$$\langle \text{Ниже, В гостях, Пропускают, Нет, Нет} \rangle.$$

G[0] выдаёт 1 на этом примере — значит, надо специализировать. И сразу же мы сталкиваемся с эффектом «размножения»: максимально общие гипотезы, которые обобщают H[0] и выдают 0, появляются сразу в количестве трёх штук:

$$\langle \text{Выше}, ?, ?, ? \rangle, \langle ?, \text{Дома}, ?, ? \rangle, \langle ?, ?, \text{На месте}, ? \rangle,$$

CandidateElimination(D)

1. $H := \{\langle \emptyset, \ldots, \emptyset \rangle\}$.
2. $G := \{\langle ?, \ldots, ? \rangle\}$.
3. Для всех $d \in D$:
   а) Если $t_d = 1$:
      (i) Для каждой $h \in G$, если $h(d) = 0$,
      $$G := G \setminus \{h\}.$$
      (ii) Для каждой $h \in S$, если $h(d) = 0$,
      $$S := S \setminus \{h\} \cup \mathrm{Gen}(h), \text{ где } \mathrm{Gen}(h) =$$
      $$= \min \left\{ h' \mid h' \succ h, h'(d) = 1, \exists h_g \in G : \ h_g \succ h' \right\}.$$
   б) Если $t_d = 0$:
      (i) Для каждой $h \in S$, если $h(d) = 1$,
      $$S := S \setminus \{h\}.$$
      (ii) Для каждой $h \in G$, если $h(d) = 1$,
      $$G := G \setminus \{h\} \cup \mathrm{Spe}(h), \text{ где } \mathrm{Spe}(h) =$$
      $$= \max\{h' \mid h \succ h', h'(d) = 0, \exists h_s \in S : \ h' \succ h_s\}.$$
4. Выдать G, H.

Рис. 2.4. Алгоритм исключения кандидатов

и текущее положение дел принимает вид

$$H = \{\langle \text{Выше, Дома, На месте, Нет} \rangle\},$$

$$G = \{\langle \text{Выше, ?, ?, ?} \rangle, \langle ?, \text{Дома, ?, ?} \rangle, \langle ?, ?, \text{На месте, ?} \rangle\}.$$

Третий пример

$$\langle \text{Ниже, Дома, Пропускают, Да, Нет} \rangle.$$

продолжает дело специализации общих гипотез; хотя G[0] и G[2] с ним справляются, G[1] выдаёт 1. Специализация получается следующая:

$$\mathrm{Spe}(G[1]) = \{\langle \text{Выше, Дома, ?, ?} \rangle, \langle ?, \text{Дома, На месте, ?} \rangle, \langle ?, \text{Дома, ?, Нет} \rangle\}.$$

После поглощения первой из этих гипотез уже существующей в G более общей множества принимают вид

$$H = \{\langle \text{Выше, Дома, На месте, Нет} \rangle\},$$

$$G = \{\langle \text{Выше, ?, ?, ?} \rangle, \langle ?, ?, \text{На месте, ?} \rangle, \langle ?, \text{Дома, ?, Нет} \rangle\}.$$

Четвёртый и последний пример

$$\langle \text{Выше, Дома, Пропускают, Нет, Да} \rangle.$$

даст нам возможность удалить одну из гипотез. G[1] выдаёт 0, поэтому её удаляем. А H[0] надо обобщить, в результате чего получается ⟨Выше, Дома, ?, Нет⟩.

В итоге мы получили максимально частную гипотезу, объясняющую все позитивные тестовые примеры:

$$h = \langle ?, \text{Дома}, ?, \text{Нет} \rangle,$$

а также набор максимально общих гипотез, совместных со всеми негативными свидетельствами:

$$G = \{\langle \text{Выше}, ?, ?, ? \rangle, \langle ?, \text{Дома}, ?, \text{Нет} \rangle\}.$$

§ 2.7. Заключение

В этой главе мы рассказали о не слишком часто применяющемся, но полезном и показательном аппарате обучения концептам. Алгоритмы, разработанные в этой главе, ничуть не сложнее, чем алгоритм ID3, которым мы в § 1.5 обучали деревья принятия решений, и решают очень схожую задачу. Мы обсудили два алгоритма; первый из них — совсем простой `Find-S`, который быстро обрабатывает множество тестовых примеров в том случае, когда каждый тестовый пример имеет один и тот же ответ.

Второй алгоритм, исключения кандидатов, как и `Find-S`, работает хорошо и быстро, а результат его является в соответствующих предположениях исчерпывающе полным — описывает все без исключения совместные с данными гипотезы. Есть только одно «но»: предположения требуются очень уж жёсткие. Алгоритм исключения кандидатов работает только в том случае, если целевая функция содержится во множестве возможных гипотез. Конечно, если истина описывается нехитрой конъюнкцией нескольких значений атрибутов, она непременно будет где-то между полученными в результате работы алгоритма пределами. Но если она вообще не может описываться подобного рода гипотезами, то алгоритм может сойтись к пустому множеству — а может и не сойтись, если данных недостаточно.

Это фактически единственный недостаток алгоритма исключения кандидатов. Но он очень серьёзный, потому что множество гипотез в обучении концептам действительно весьма бедное. Например, нам пришлось каждый раз довольствоваться некоторым подмножеством данных табл. 2.1, потому что вся таблица просто не может описываться одной конъюнкцией нескольких атрибутов. Ей нужна бóльшая выразительность, которую обеспечивают, например, деревья принятия решений. Но если вы уверены, что для вашей задачи такого множества гипотез достаточно, смело применяйте алгоритм исключения кандидатов.

В следующей главе мы перейдём от задачи аппроксимации булевской функции к задаче аппроксимации функции вещественнозначной. И помогут нам в этом искусственные нейронные сети.

Глава 3

Нейронные сети

К 80-м годам двадцатого столетия... Минский и Гуд разработали методику автоматического зарождения и самовоспроизведения нервных цепей в соответствии с любой произвольно выбранной программой. Оказалось, что искусственный мозг можно «выращивать» посредством процесса, поразительно сходного с развитием человеческого мозга. Точные детали этого процесса в каждом отдельном случае так и оставались неизвестными; впрочем, будь они даже известны, человеческий разум не смог бы постичь всю их сложность.

Космическая Одиссея 2001
Артур Кларк. Пер. *Норы Галь.*

Однажды, солнечным зимним днём, Винни-Пух и Пятачок пили чай с мёдом в то самое время, когда завтрак давно закончился, а обед ещё и не думал начинаться. Вдруг медвежонок вспомнил:

— Пятачок, а помнишь, как в прошлом году мы охотились на Буку?

— Помню, Винни, — предчувствуя неладное, осторожно сказал Пятачок, — а, собственно, что?

— Я тут подумал, что тогда была жуткая метель. А сейчас солнышко светит. Если бы мы прямо сейчас пошли искать Буку, мы бы наверняка его нашли! Тем более что мёд у нас уже кончился.

Против последнего аргумента возразить было нечего, и Пятачок, горестно вздохнув, поплёлся вслед за Винни-Пухом. Впрочем, ещё оставалась надежда, что дело кончится ничем:

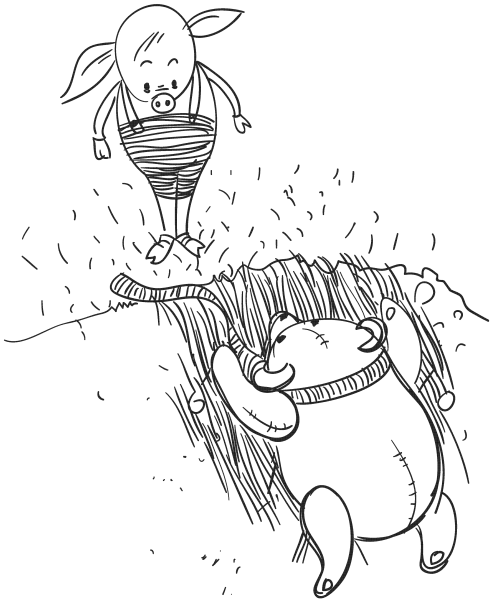
— Винни, а как мы будем его искать? Где живут Буки?

— Это я твёрдо знаю, Пятачок! — уверенно ответил Пух. — Бука живёт в Самой Глубокой Яме во всём Лесу.

— Ага. Понятно, — упавшим голосом пробормотал Пятачок, но надежда вновь затеплилась. — Но Винни, как же мы найдём Самую Глубокую Яму?

— Пятачок, это же совсем просто! Смотри, вот тут начинается склон вниз. Я пойду туда и буду спускаться и спускаться, пока совсем не спущусь, — Пух продемонстрировал, как он будет спускаться, но на третьем же шаге поскользнулся и дальнейший путь проделал сидя. — Ох, как быстро получилось! Вот я и на дне.

— И как, там есть Бука? — спросил Пятачок, опасливо заглядывая в овраг, на дне которого оказался Винни.



— Похоже, что нет, — задумчиво ответил Пух. — Но как же так получилось... А! Понял! Эта яма — просто не Самая Глубокая. Я сейчас из неё вылезу, потом найду другой склон и спущусь в другую яму, и если та, другая, будет Самой Глубокой, то Бука точно объявится, — с этими словами медвежонок начал упорно выбираться из оврага.

Пух и Пятачок ещё долго искали Самую Глубокую Яму в лесу. Они испробовали много разных склонов, и сидя, и стоя, и лёжа на животе. Было очень весело, но, видимо, глобального минимума найти не получилось — Бука так и не показался...

§ 3.1. Введение

Пока что ни один, даже самый мощный и современный, компьютер не может по своим способностям сравниться с человеческим мозгом. Да, компьютеры гораздо быстрее выполняют арифметические операции, но если мы хотим научить компьютер чему-либо более сложному, более «человеческому» — понимать естественный язык, узнавать людей, принимать решения в сложных ситуациях, обучаться в широком смысле этого слова — компьютер пасует перед человеком.

Почему так происходит? Однозначного ответа на этот вопрос, конечно, нет. Например, Роджер Пенроуз в книге «Новый ум короля» [173] рассматривает и отнюдь не отвергает возможность того, что человеческий мозг в своей «вычислительной» деятельности использует квантовые феномены; вполне возможно, что каждый из нас носит в голове мощнейший из известных квантовых компьютеров.

Но даже если оставить квантовые вычисления за бортом — всё-таки это вовсе не общепринятая теория — устройство нашего мозга предоставляет пищу для любопытных размышлений. В нейросетевом подходе предполагается, что мозг состоит из *нейронов*, которые соединяются друг с другом в единую большую сеть. Когда нейрон возбуждается, он передаёт своим соседям электрические сигналы определённого вида. Соседи их обрабатывают и передают дальше, и так далее.

Примечательно, что соотношение скоростей передачи сигнала и человеческой реакции таково, что на самом деле цепочка

нейронов, которые успели бы возбудиться перед принятием решения, не может быть длиннее нескольких сот штук! Мозг добивается своей, простите за каламбур, головоломной сложности и эффективности не огромным количеством *последовательных* вычислений, как современные компьютеры, а удачными связями между своими нейронами, которые позволяют для решения каждой конкретной задачи задействовать цепочки небольшой глубины (то есть, можно сказать, не только быстрыми алгоритмами, но и удачными структурами данных).

В истории науки нередки случаи, когда люди заимствовали инженерные решения у матушки-природы. Многие животные, сами того не подозревая, становились прообразами конструктивных элементов зданий, машин и прочей аппаратуры. Поэтому нет ничего удивительного, что люди решили использовать свои знания о структуре мозга для того, чтобы имитировать его работу — а там, чем чёрт не шутит, может, компьютер и мыслить начнёт... Так и появились *искусственные нейронные сети.*[1]

Стоит отметить, что искусственные нейронные сети — не единственный аппарат машинного обучения, основанный на идеях нейропсихологии. Современные исследователи полагают, что *обучение по Хеббу*[2] (Hebbian learning), в котором сети из связанных друг с другом нейронов обучаются сходиться к заданным «впечатлениям», ещё более походит на работу мозга, чем нейронные сети. Однако в этой главе мы сконцентрируемся на последних.

---

[1] Существует направление исследований, посвящённых более точной имитации работы мозга, чем та, которую мы будем рассматривать в этой главе. Однако их цель — не создание эффективных алгоритмов машинного обучения, а исследование мыслительных процессов человека. Поэтому мы не будем касаться этой проблематики, а заинтересованных читателей — там есть чем заинтересоваться, наука очень красивая и в высшей степени математическая! — отсылаем к [74, 76, 87].

[2] Дональд Олдинг Хебб (Donald Olding Hebb, 1904–1985) — канадский психолог, один из основателей нейропсихологии. Его главный труд — «Организация поведения» (1949) [65]. В нём Хебб фактически впервые предложил удовлетворительную теорию, связывающую физические и биологические процессы, протекающие в мозге, и его мыслительную деятельность. Главная идея теории заключается в следующем: если один нейрон часто активирует другой, ему это становится делать всё проще (второй нейрон «запоминает», что он часто получал импульс от первого) [65].

§ 3.2. Перцептрон

Нейрон человеческого мозга возбуждается, если получает достаточно мощный электрический импульс. При этом, если нейрон соединён с несколькими другими нейронами, то импульс может получаться как некоторая функция импульсов от входящих нейронов. Простейшая математическая модель одного нейрона — это *перцептрон* (perceptron).

Перцептрон реализует вышеописанную модель линейными функциями. Он подсчитывает некоторую линейную форму от своих входов и сравнивает её с заданным значением — *порогом активации* (threshold). Если у перцептрона n входов $x_1, \ldots, x_n$, то в нём должны быть заданы n весов $w_1, w_2, \ldots, w_n$ и порог активации w_0. Перцептрон выдаёт 1, если линейная форма от входов с коэффициентами w_i превышает $-w_0$ (минус нужен, чтобы перенести w_0 в левую часть уже со знаком «плюс»). Иначе говоря, выход перцептрона $o(x_1, \ldots, x_n)$ вычисляется так:

$$o(x_1, \ldots, x_n) = \begin{cases} 1, \text{ если } w_0 + w_1 x_1 + \ldots + w_n x_n \geqslant 0, \\ -1 \text{ в противном случае.} \end{cases}$$

В дальнейшем мы не будем различать w_i, $i = 1..n$, и w_0, а просто предположим, что у перцептрона есть ещё один вход x_0, на который всегда подаётся единица. Тогда условие срабатывания перцептрона можно записать как $\sum_0^n w_i x_i > 0$.

Пример 3.1. Примеры перцептронов

Мы будем изображать перцептроны в виде узлов графа, помечая их значением $-w_0$. Значения w_i мы будем указывать на входящих рёбрах.

На рис. 3.1 изображён общий вид перцептрона, а также даны примеры перцептронов, которые реализуют дизъюнкцию и конъюнкцию. Для дизъюнкции достаточно сделать порог активации ниже, чем любой из весов на входе; тогда единица, полученная на любом входе, приведёт к срабатыванию. А чтобы получилась конъюнкция n входов, нужно установить такой порог активации, чтобы сумма $n-1$ входа была меньше него, а сумма всех n входов — уже больше. И дизъюнкция, и конъюнкция — частные случаи функций вида «m из n», которые легко реализуются перцептроном.

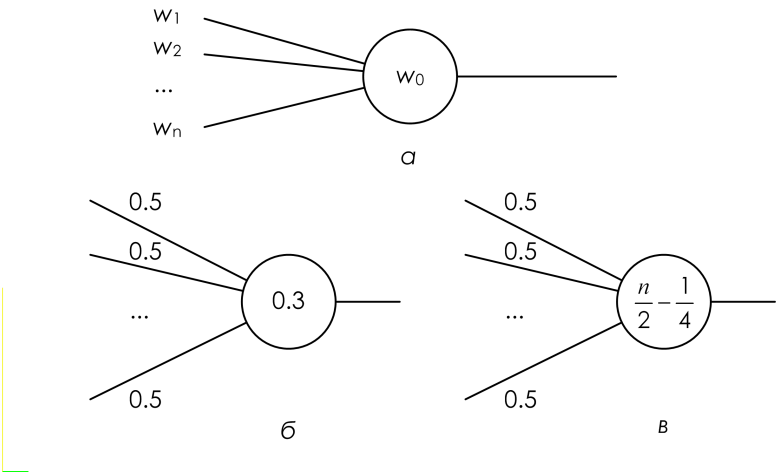


Рис. 3.1. Примеры перцептронов: *а* — общий вид перцептрона; *б* — перцептрон, реализующий дизъюнкцию; *в* — перцептрон, реализующий конъюнкцию

Один перцептрон — это уже достаточно мощный инструмент: он может реализовать любую гиперплоскость, рассекающую пространство возможных решений. Например, если целевая функция (функция, которую мы пытаемся приблизить при помощи нейронной сети) позволяет *линейно отделить* свои положительные значения от отрицательных (то есть если существует такая гиперплоскость, что все положительные значения лежат по одну её сторону, а отрицательные — по другую), то вполне достаточно будет создать сеть из *одного* перцептрона. Кстати, задачи с линейно отделимыми множествами решений возникают на практике очень часто, хотя обычно для их решения используют не перцептроны, а *метод опорных векторов* (support vector machines), который мы в этой книге не рассматриваем [34].

Мы объединяем перцептроны в сети, подавая выходы одних перцептронов на входы других. Эта конструкция оказывается очень выразительной. Фактически, уже сеть глубины 2 — это всё, что может нам понадобиться для того, чтобы решить любую булевскую задачу.

Теорема 3.1. *Любая булевская функция представима в виде построенной из перцептронов искусственной нейронной сети глубины 2.*

Доказательство. Известно, что всякую булевскую функцию f можно представить в виде дизъюнктивной нормальной формы (ДНФ) — конъюнкции нескольких дизъюнкций — и в виде конъюнктивной нормальной формы (КНФ) — дизъюнкции нескольких конъюнкций. Как мы уже видели выше (см. рис. 3.1), существуют перцептроны, реализующие дизъюнкцию и конъюнкцию. Таким образом, чтобы представить f, достаточно сети глубины 2, выражающей ДНФ функции f. □

Стоит отметить, что представление функции в виде ДНФ вполне может оказаться неэффективным, в том числе даже экспоненциально более неэффективным, чем какое-нибудь другое, более хитрое представление. Поэтому не стоит торопиться выражать все функции сетями глубины 2 — не исключено, что для этого потребуется слишком много нейронов, а сеть большей глубины могла бы решить ту же самую задачу более разумно.

Пример 3.2. Сеть, реализующая XOR

Поучительный пример — сеть, реализующая исключающее ИЛИ (XOR). На рис. 3.2а изображена сеть глубины 2, реализующая XOR двух переменных $x \oplus y$ (проверьте это!). На рис. 3.2в аналогичная сеть реализует XOR трёх переменных $x \oplus y \oplus z$.

Дальнейшее понятно: каждая новая переменная приносит лишь линейное увеличение в числе требуемых перцептронов, и общее их количество либо квадратично (если считать так, как показано на рис. 3.2), либо линейно (для этого нужно либо предположить, что вход z можно подать сразу на вход перцептронам третьего уровня, без промежуточных тривиальных перцептронов, либо просто построить бинарное дерево из XOR'ов, в котором на каждом уровне, то есть на каждых двух уровнях нейронной сети, число переменных уменьшается вдвое).

Если же ограничиваться сетями глубины 2, то дело значительно хуже. И ДНФ, и КНФ функции PARITY (чётность; $x_1 \oplus \ldots \oplus x_n$) имеют экспоненциальный размер; были доказаны и нижние оценки для собственно перцептронов [10,55,56]. На рис. 3.2б приведён пример сети глубины 2, реализующей функцию чётности трёх переменных (чтобы не загромождать рисунок, мы размножили входы; на самом деле, конечно, на x, y и z приходится по одной вершине).

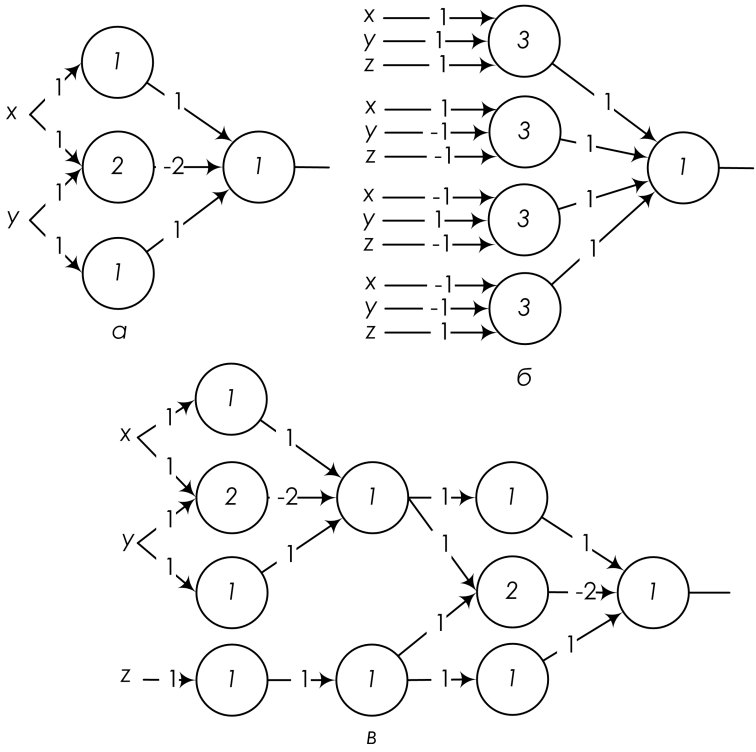


Рис. 3.2. Сети, реализующие XOR: *а* — двух переменных; *б* — сеть глубины 2, реализующая XOR трёх переменных; *в* — сеть глубины 4, реализующая XOR трёх переменных

Главное ограничение линейных перцептронов без лимита активации состоит в том, что они реализуют исключительно линейные функции. А композиции линейных функций снова оказываются линейными. В результате объединять такие перцептроны в сеть бессмысленно: вся сеть будет эквивалентна *одному-единственному* перцептрону.

На самом же деле искусственные нейронные сети могут использоваться и для куда более амбициозных задач. Но для этого им потребуются иные, более мощные элементы, о которых пойдёт речь ниже. А сейчас мы вспомним об основной стоящей перед нами задаче и рассмотрим *обучение* одного перцептрона.

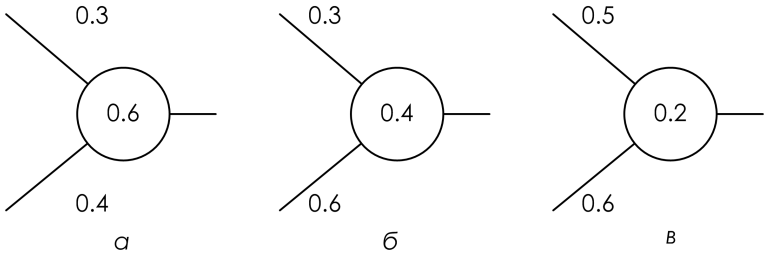


Рис. 3.3. Обучение перцептрона: *а* — перед началом обучения; *б* — после первого шага обучения; *в* — перцептрон, реализующий дизъюнкцию

§ 3.3. Обучение перцептрона

Уже понятно, что перцептроны отличаются друг от друга исключительно весами w_i. Значит, обучение должно заключаться в том, чтобы подправлять эти веса в соответствии с поведением перцептрона на тестовых примерах. При этом, если перцептрон отработал правильно, веса, скорее всего, измениться не должны, а если неправильно, то должны сдвинуться в сторону желаемого.

Будем пока рассматривать сеть из одного перцептрона. Постановка задачи: научить такую сеть распознавать частично заданную булевскую функцию на основании данных о том, как справился перцептрон с тестовыми данными.

Простейший алгоритм так и называется в литературе — *правило обучения перцептрона* (perceptron training rule). На каждом шаге обучения вес w_i изменяется в соответствии с правилом

$$w_i := w_i + \eta(t - o)x_i,$$

где t — значение целевой функции, o — выход перцептрона, $\eta > 0$ — небольшая константа (обычно 0,05–0,2), которая задаёт скорость обучения.

Пример 3.3. Обучение перцептрона

Предположим, что мы хотим обучить перцептрон, изображённый на рис. 3.3а, реализовывать дизъюнкцию своих входов. Для этого нужно, чтобы значение и w_1, и w_2 стало больше значения порога срабатывания w_0. Пусть $\eta = 0{,}1$, и у нас есть следующий набор тестовых

данных:

$$\begin{aligned} x_1 = 0, \quad x_2 = 1 \quad &\Rightarrow \quad t = 1, \\ x_1 = 1, \quad x_2 = 0 \quad &\Rightarrow \quad t = 1. \end{aligned}$$

Первый же тест приведёт к ошибке: перцептрон выдаст $o = -1$, а искомый выход — $t = 1$. Таким образом, веса перцептрона будут подправлены следующим образом (см. рис. 3.3б):

$$\begin{aligned} w_0 &:= w_0 + \eta(t-0)x_0 = \quad -0{,}6 + 0{,}1 \cdot (1-(-1)) \cdot 1 = \quad -0{,}4, \\ w_1 &:= w_1 + \eta(t-0)x_1 = \quad 0{,}3 + 0{,}1 \cdot (1-(-1)) \cdot 0 = \quad 0{,}3, \\ w_2 &:= w_2 + \eta(t-0)x_2 = \quad 0{,}4 + 0{,}1 \cdot (1-(-1)) \cdot 1 = \quad 0{,}6. \end{aligned}$$

В полученном перцептроне один из весов на входе уже больше, чем порог активации. Однако второй тест подаёт на этот вход ноль, и снова выясняется, что перцептрон не сработает. На этот раз веса придётся изменить так:

$$\begin{aligned} w_0 &:= w_0 + \eta(t-0)x_0 = \quad -0{,}4 + 0{,}1 \cdot (1-(-1)) \cdot 1 = \quad -0{,}2, \\ w_1 &:= w_1 + \eta(t-0)x_1 = \quad 0{,}3 + 0{,}1 \cdot (1-(-1)) \cdot 1 = \quad 0{,}5, \\ w_2 &:= w_2 + \eta(t-0)x_2 = \quad 0{,}6 + 0{,}1 \cdot (1-(-1)) \cdot 0 = \quad 0{,}6. \end{aligned}$$

Полученный перцептрон (рис. 3.3в) реализует дизъюнкцию своих входов, то есть обучен и готов к труду и обороне.

Может случиться так, что одной итерации по всем тестовым примерам будет недостаточно для того, чтобы исправить каждую ошибку (если бы в примере 3.3 η была бы равна не 0,1, а 0,01, веса были бы исправлены в нужную сторону, но недостаточно). Поэтому нужно запускать алгоритм по имеющимся тестовым примерам до тех пор, пока очередной прогон алгоритма по всем тестам не оставит все веса на месте. Одно применение алгоритма ко всем тестовым примерам называется *эпохой* (epoch). Получившийся алгоритм представлен на рис. 3.4. На вход ему подаётся набор из m тестовых примеров $\{x_i^j, t_j\}$ и скорость обучения η, а на выходе получаются значения w_i перцептрона, который линейно разделяет эти тестовые примеры.

Алгоритм на рис. 3.4 сходится к правильному перцептрону всегда, когда это возможно.

Теорема 3.2. *Если некоторое подмножество точек n-мерного булевского куба $C_1 \subset \{0,1\}^n$ можно в $\{0,1\}^n$ отделить гиперплоскостью от другого подмножества точек*

PerceptronTraining($\eta, \{x_i^j, t^j\}_{i=1,j=1}^{n,m}$)

1. Инициализировать $\{w_i\}_{i=0}^{n}$ маленькими случайными значениями.
2. WeightChanged := true.
3. Пока WeightChanged = true:
   а) WeightChanged := false.
   б) Для всех j от 1 до m:
      (i) Вычислить
      $$o^j := \begin{cases} 1, \text{ если } w_0 + w_1 x_1^j + \ldots + w_n x_n^j > 0, \\ -1 \text{ в противном случае.} \end{cases}$$
      (ii) Если $o^j \neq t^j$:
         (A) WeightChanged = true.
         (B) Для каждого i от 0 до n изменить значение w_i по правилу
         $$w_i := w_i + \eta(t^j - o^j)x_i^j.$$
4. Выдать значения $w_0, w_1, \ldots, w_n$.

Рис. 3.4. Алгоритм обучения перцептрона

$C_2 \subset \{0,1\}^n$, *то алгоритм обучения перцептрона за конечное число шагов выдаёт параметры перцептрона, который успешно разделяет множества* C_1 *и* C_2.

Доказательство. Итак, мы предполагаем, что входы принадлежат к отделимым гиперплоскостью множествам; это означает, что существует такой вектор $\mathbf{u}$ (нормаль к той самой гиперплоскости), что

$$\forall \mathbf{x} \in C_1 \quad \mathbf{u}^\top \mathbf{x} > 0,$$
$$\forall \mathbf{x} \in C_2 \quad \mathbf{u}^\top \mathbf{x} < 0.$$

Цель обучения перцептрона — сделать так, чтобы веса перцептрона w образовывали такой вектор u.

Прежде всего заменим C_2 на $-C_2 = \{-\mathbf{x} \mid \mathbf{x} \in C_2\}$. Это позволит нам объединить два неравенства в одно:

$$\forall \mathbf{x} \in C \; \boldsymbol{w}^\top \mathbf{x} > 0,$$

где $C = C_1 \cup (-C_2)$.

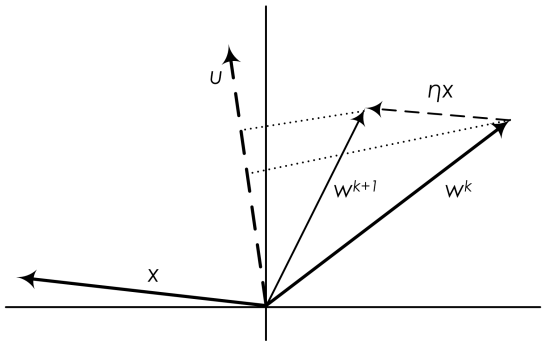


Рис. 3.5. К доказательству теоремы 3.2

Теперь давайте посмотрим на стоящую перед нами задачу с геометрической стороны. Мы хотим построить вектор w, который образует острые углы со всеми тестовыми примерами x. Для этого мы его обучаем: если вдруг обнаружится вектор, с которым w образует тупой угол, мы просто прибавим его к w (предварительно, конечно, домножив на константу η).

Осталось только понять, почему же этот процесс закончится. Для этого мы рассмотрим вектор, который *действительно* образует с ними тупые углы (по предположению теоремы, такой вектор существует), и будем доказывать, что последовательность длин проекций w на этот вектор не может быть бесконечной.

На рис. 3.5 изображено происходящее: мы подправляем w и пытаемся при этом проконтролировать проекцию w на u. Давайте теперь выразим всё это формально.

Обозначим через $\boldsymbol{w}^0, \boldsymbol{w}^1, \ldots$ векторы весов перцептрона на соответствующих этапах обучения, через $\boldsymbol{x}^0, \boldsymbol{x}^1, \ldots$ — векторы тестовых примеров. Предположим (без потери общности), что $\boldsymbol{w}^0 = 0$, все тестовые примеры принадлежат C, и на всех тестовых примерах $(\boldsymbol{w}^k)^\top \boldsymbol{x}^k < 0$, то есть перцептрон не справляется с тестом (если он справляется с тестом, то веса перцептрона никак не меняются, поэтому такие тесты можно просто исключить из последовательности). В таком случае наше правило обучения выглядит как

$$\boldsymbol{w}_i^{k+1} = \boldsymbol{w}_i^k + \eta \boldsymbol{x}_i^k,$$

и легко видеть, что в наших предположениях

$$\boldsymbol{w}^k = \eta \sum_{j=0}^{k-1} \boldsymbol{x}^j.$$

Идея доказательства состоит в том, чтобы получить две серии противоположных оценок на длину векторов $\|\boldsymbol{w}^i\|$ и показать, что они не могут обе иметь место до бесконечности. Это и будет означать, что любая конечная последовательность тестов из двух фиксированных линейно отделимых множеств рано или поздно закончится, и все последующие тесты перцептрон будет проходить успешно.[1]

Рассмотрим вектор $\mathbf{u}$, являющийся решением, и обозначим через α минимальную проекцию любого из $\boldsymbol{x}^j$ на $\mathbf{u}$:

$$\alpha = \min_j \mathbf{u}^\top \boldsymbol{x}^j.$$

Поскольку *разных* тестов конечное число, $\alpha > 0$.

Тогда

$$\mathbf{u}^\top \boldsymbol{w}^{k+1} = \eta \mathbf{u}^\top \sum_{j=0}^{k} \boldsymbol{x}^j \geqslant \eta\alpha k$$

(на самом деле даже $\eta\alpha(k+1)$, но для простоты мы сделаем оценку ещё сильнее). Вспомним неравенство Коши–Буняковского: в пространстве со скалярным произведением

$$\|\boldsymbol{x}\|^2 \|\boldsymbol{y}\|^2 \geqslant (\boldsymbol{x}^\top \boldsymbol{y})^2.$$

Применительно к нашей задаче неравенство Коши-Буняковского утверждает, что

$$\|\mathbf{u}\|^2 \|\boldsymbol{w}^{k+1}\|^2 \geqslant (\eta\alpha k)^2, \quad \|\boldsymbol{w}^{k+1}\|^2 \geqslant \frac{(\eta\alpha k)^2}{\|\mathbf{u}\|^2}.$$

Иными словами, на каждой итерации длина вектора w^k возрастает линейно (а её квадрат $\|\boldsymbol{w}^k\|^2$ — пропорционально k^2). Остальные члены правой части неравенства — константы.

[1] Можно подумать, что «конечная последовательность тестов» закончится в любом случае; но напомним, что наш алгоритм повторяет тесты до тех пор, пока перцептрон не научится проходить их все. Речь идёт лишь о том, что в этой (возможно, бесконечной) последовательности тестов участвует конечное число различных векторов $\boldsymbol{x}^i$.

С другой стороны, $\boldsymbol{w}^{k+1} = \boldsymbol{w}^k + \eta \boldsymbol{x}^k$. Поскольку $(\boldsymbol{w}^k)^\top \boldsymbol{x}^k < 0$,

$$\|\boldsymbol{w}^{k+1}\|^2 = \|\boldsymbol{w}^k\|^2 + 2\eta(\boldsymbol{w}^k)^\top \boldsymbol{x}^k + \eta^2\|\boldsymbol{x}^k\|^2 \leqslant \|\boldsymbol{w}^k\|^2 + \eta^2\|\boldsymbol{x}^k\|^2.$$

Следовательно, $\|\boldsymbol{w}^{k+1}\|^2 - \|\boldsymbol{w}^k\|^2 \leqslant \eta^2\|\boldsymbol{x}^k\|^2$. Суммируя по k, имеем:

$$\|\boldsymbol{w}^{k+1}\|^2 \leqslant \eta^2 \sum_{j=0}^{k} \|\boldsymbol{x}^k\|^2 \leqslant \eta^2 \beta k,$$

где $\beta = \max_j \|\boldsymbol{x}^j\|^2$ (мы опять воспользовались тем, что тестовых примеров конечное число).

Итак, у нас получились две оценки:

$$\frac{(\eta\alpha)^2}{\|\boldsymbol{u}\|^2} k^2 \leqslant \|\boldsymbol{w}^{k+1}\|^2 \leqslant \eta^2 \beta k.$$

Очевидно, что рано или поздно с ростом k (k — это натуральное число) эти оценки войдут в противоречие друг с другом. Это значит, что последовательность применения одних и тех же тестовых примеров не может быть бесконечной: рано или поздно все примеры, если это было вообще возможно, будут перцептроном проходиться правильно.

□

§ 3.4. Обучение перцептрона на практике

Обучение перцептрона — очень простая и доступная задача, которую на подавляющем большинстве языков программирования можно реализовать в два-три десятка строчек. Поэтому оно хорошо подходит для того, чтобы продемонстрировать некоторые языки программирования, которые будут в дальнейшем использоваться в этой книге (напоминаем, что владеть всеми представленными здесь языками программирования вовсе не обязательно, задача, как правило, в том, чтобы сначала найти язык, наилучшим образом подходящий для решения стоящей перед вами задачи, а потом уже можно изучить его тонкости).

В листинге 3.1 приведён код программы на языке `Python`, реализующей обучение одного перцептрона. На вход подаются значение η и массив тестовых примеров в формате

$$[t^j, x_1^j, \ldots, x_n^j].$$

В строке 4 значения весов инициализируются небольшими случайными значениями. В строке 9 мы читаем формат входных данных и заменяем для удобства t^j на единицу, то есть на x_0^j. Последняя строка — пример использования реализованной подпрограммы; на выходе должно получаться нечто вроде перцептрона, реализующего конъюнкцию трёх входов («нечто вроде», потому что функция задана частично, и вполне возможно, что при некоторых комбинациях начальных данных результат будет не совсем идеальным). В листинге 3.2 приведён тот же алгоритм с тем же форматом входа и выхода, но на языке Ruby.

Листинг 3.1. Обучение перцептрона на языке Python

```
def PerceptronTraining(eta,x):
    import random
    w=[]
    for i in range(len(x[0])):
        w.append((random.randrange(-5,5))/50.0)

    WeightsChanged=True
    while (WeightsChanged==True):
        WeightsChanged=False
        for xj in x:
            t,o,curx=xj[0],0,[1]+xj[1:len(xj)]
            for i in xrange(len(w)): o+=w[i]*curx[i]
            if o>0: o=1
            else: o=-1
            if (o==t): continue
            WeightsChanged=True
            for i in xrange(len(w)): w[i]+=eta*(t-o)*curx[i]
    return w
```

Листинг 3.2. Обучение перцептрона на языке Ruby

```
def PerceptronTraining(eta, x)
    w = Array.new(x[0].size){(rand(11) - 5) / 50.0}

    weightsChanged = true
    while weightsChanged
        weightsChanged = false
        x.each { |xj|
            t, o, curx = xj[0], 0, [1] + xj[1..-1]
            (0...w.size).each { |i| o += w[i] * curx[i] }
            if o > 0 then o = 1 else o = -1 end
```

```
            if o != t
                weightsChanged = true
                (0...w.size).each { |i|
                    w[i] += eta * (t - o) * curx[i]
                }
            end
        }
    end
    w
end
```

Третий пример в этом параграфе будет на языке логического программирования — Prolog[1]. Основой языка является *автоматический доказатель* (theorem prover), который может производить логический вывод на хорновских формулах. Процесс выполнения программы — это доказательство или опровержение заданного факта путём логического вывода из кода программы. Функциональные языки очень хорошо реализуют рекурсию; классический пример — вычисление факториала — на языке Prolog выглядит так:

```
factorial(0,F,F).
factorial(N,A,F) :-
    N > 0,
    A1 is N*A,
    N1 is N-1,
    factorial(N1,A1,F).
```

Обратите внимание, что в Prolog приходится вводить третий параметр, в котором, собственно, и будет вычисляться факториал: предикат `factorial` будет истинным только для одного значения `F`. Дело в том, что, вообще говоря, Prolog работает с предикатами; так, например, предикат, проверяющий, является ли одно число факториалом другого, выглядел бы проще. Для более подробного ознакомления с языком Prolog можно порекомендовать [33, 43], а мы сейчас просто приведём конкретный пример.

[1] Авторы, Ален Колмероэ (Alain Colmerauer) и Филип Руссель (Philip Roussel), были французами, поэтому Prolog — это «*Pro*grammation en *Log*ique».

Листинг 3.3. Обучение перцептрона на языке Prolog

```
write_list([A|B]):-
    write(A),
    nl,
    write_list(B).

write_list([]).

% случай W0 + W1*X1 + ... + Wn*Xn > 0, t=1
testsum_is_bigger_then_w0([T|T_TAIL],[W|W_TAIL],SUM):-
    T_TAIL=[],
    W_TAIL=[],
    SUM> -(W),
    T=:=1,!.

% случай W0 + W1*X1 + ... + Wn*Xn <= 0, t=0
testsum_is_bigger_then_w0([T|T_TAIL],[W|W_TAIL],SUM):-
    T_TAIL=[],
    W_TAIL=[],
    SUM =< -(W),
    T=:=0,!.

testsum_is_bigger_then_w0([T|T_TAIL],[W|W_TAIL],SUM):-
    NEXT_SUM is SUM+(T*W),
    testsum_is_bigger_then_w0(T_TAIL,W_TAIL,NEXT_SUM).

one_test_passed([T|T_TAIL],[W|W_TAIL]):-
    SUM is T*W, % "инициализация" SUM
    testsum_is_bigger_then_w0(T_TAIL,W_TAIL,SUM).

% проверка тестов T при весах W
check_tests([T|TAIL],W):-
    one_test_passed(T,W),
    check_tests(TAIL,W).

check_tests([],_).

% Здесь 0.1 - константа, задающая скорость обучения.
correct([T|[]],[W|[]],[CRTD_W|[]],O):-
    CRTD_W is W+0.1*(T-O).

% Применение формулы.
correct([T|T_TAIL],[W|W_TAIL],[CRTD_W|CW_TAIL],O):-
    CRTD_W is W+0.1*(T-O)*T,
    correct(T_TAIL,W_TAIL,CW_TAIL,O).

process_count_iter([_|[]],[W|[]],CUR_OO,OO):-
    OO is CUR_OO + W.
```

```
process_count_iter([T|T_TAIL],[W|W_TAIL],CUR_OO,OO):-
    NEW_OO is CUR_OO + T*W,
    process_count_iter(T_TAIL,W_TAIL,NEW_OO,OO).

% получение результата функции при текущих весах
get_cur_res([T|T_TAIL],[W|W_TAIL],O):-
    START_OO is T*W,
    process_count_iter(T_TAIL,W_TAIL,START_OO,OO),
    OO > 0,
    O = 1.

get_cur_res([T|T_TAIL],[W|W_TAIL],O):-
    START_OO is T*W,
    process_count_iter(T_TAIL,W_TAIL,START_OO,OO),
    OO =< 0,
    O = -1.

check_and_correct([T|T_TAIL],W,NEW_W):-
    one_test_passed(T,W),
    check_and_correct(T_TAIL, W, NEW_W).

check_and_correct([T|T_TAIL],W,NEW_W):-
    get_cur_res(T,W,O),
    correct(T,W,CRTD_W,O),
    write('correction:'),nl,
    write_list(CRTD_W),nl,
    check_and_correct(T_TAIL,CRTD_W, NEW_W).

check_and_correct([],W,W).

train_perceptron(T,W,RETURN_W):-
    check_tests(T,W),
    RETURN_W=W.

train_perceptron(T,W,RETURN_W):-
    check_and_correct(T,W,NEW_W),
    train_perceptron(T,NEW_W,RETURN_W).

start:-

% Теперь - ввод и вывод. T - тесты. Например,
%         T=[[0,1,1],[1,0,1]]
% означает
%     [x1=0, x2=1, t=1], [x1=1, x2=0, t=1]
    write('Enter test cases
           [[x1,x2,...t], [x1,x2,...t], ...].'),nl,
    read(T),
```

```
% W - веса. Например,
%         W=[0.3,0.4,-0.6]
% означает
%       [w1 = 0.3, w2 = 0.4, w0 = -0.6]
    write('Enter weights  [w1, w2, ..., w0].'),nl,
    read(W),
    train_perceptron(T,W,CORRECT_W),
    write('result:'),nl,
    write_list(CORRECT_W),nl.
```

§ 3.5. Метод градиентного спуска

В примере 3.3 нам повезло: дизъюнкция оказалась функцией, положительные значения которой можно линейно отделить от отрицательных. Но что делать, если нужно реализовать функцию, для которой это невозможно? Если нельзя увеличивать сложность нейронной сети (а мы пока останемся в рамках одного перцептрона), то точного попадания в функцию не достичь.

Нужно найти перцептрон, который бы в каком-то смысле *минимизировал* ошибку. Сначала немного обобщим стоящую перед нами задачу: откажемся от идеи порога срабатывания и предположим, что мы просто вычисляем перцептроном функцию

$$o(x_0,\ldots,x_n) = \sum_{i=0}^{n} w_i x_i,$$

а нужно нам как можно лучше приблизить функцию

$$f(x_0,\ldots,x_n),$$

которая на тестовых примерах x^j, $j = 1..m$ задана значениями $f(x_0^j,\ldots,x_n^j) = t_j$.

В качестве меры ошибки рассмотрим среднеквадратичное отклонение от целевых значений:

$$E(w_0,\ldots,w_n) = \frac{1}{2}\sum_{j=1}^{m}(t_j - o(x_0^j,\ldots,x_n^j))^2.$$

Эта мера широко используется в статистике; она как нельзя лучше подходит и для нашей задачи (разговор о том, *почему* подходит, предстоит нам в § 5.9).

GradientDescent($\eta, \{x_i^j, t^j\}_{i=1,j=1}^{n,m}$)

1. Инициализировать $\{w_i\}_{i=0}^n$ маленькими случайными значениями.
2. Повторить NUMBER_OF_STEPS раз:
   а) Для всех i от 1 до n $\Delta w_i := 0$.
   б) Для всех j от 1 до m:
      (i) Для всех i от 1 до n
      $$\Delta w_i := \Delta w_i + \eta \left(t^j - \sum_0^n w_i x_i^j \right) x_i^j.$$
   в) $w_i := w_i + \Delta w_i$.
3. Выдать значения $w_0, w_1, \ldots, w_n$.

Рис. 3.6. Алгоритм градиентного спуска для одного перцептрона

Цель — минимизировать функцию E на пространстве возможных весов $\{w_i\}$. График функции E представляет собой параболическую поверхность, и у неё должен быть один-единственный минимум. А вопрос о том, как исправлять веса так, чтобы двигаться в сторону этого минимума, давным-давно решён в математическом анализе. Для этого нужно двигаться в направлении, противоположном *градиенту* — вектору, вдоль которого производная максимальна.[1] Градиент вычисляется следующим образом:

$$\nabla E(w_0, \ldots, w_n) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right].$$

Таким образом, чтобы определить, как подправлять веса, мы должны вычислить градиент и отнять вектор градиента, умноженный на какую-нибудь наперёд заданную константу (это та самая константа скорости обучения η), от имеющегося вектора

[1] Стоит отметить, конечно, что ещё раньше был решён и вопрос о том, где у квадратичной функции минимум. Иначе говоря, в данном случае мы могли бы не подправлять веса, а напрямую, решив систему линейных уравнений, получить оптимальное значение весов перцептрона. Однако такой подход сработал бы только для линейных перцептронов, а нам приходится держать в голове и более сложные случаи; они начнутся уже в § 3.6.

весов:

$$w_i := w_i - \eta \frac{\partial E}{\partial w_i}.$$

Чтобы реализовать это программно, нужно научиться дифференцировать функцию E; к счастью, это совсем несложно:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1}^{m} \frac{\partial}{\partial w_i} \left(t^j - \sum_{0}^{n} w_i x_i^j \right)^2 = \sum_{j=1}^{m} \left(t^j - \sum_{0}^{n} w_i x_i^j \right) (-x_i^j).$$

Это значит, что нам нужно подправлять веса после каждого тестового примера по следующему правилу:

$$w_i := w_i + \eta \sum_{j} \left(t^j - \sum_{0}^{n} w_i x_i^j \right) x_i^j.$$

Новый, адаптированный алгоритм называют *алгоритмом градиентного спуска*; он показан на рис. 3.6. Правда, нужно внести и другие изменения. Во-первых, мы больше не можем рассчитывать на то, что за конечное время достигнем идеальной гармонии с исходными данными; поэтому нам нужно научиться останавливаться в какой-то момент. В качестве условия для остановки здесь можно принять простое повторение алгоритма достаточное число раз. Другое изменение — в том, что если оставлять значение η постоянным, то на каком-то этапе вектор весов перестанет приближаться к искомому минимуму, а начнёт его «перепрыгивать» на каждой итерации, то в одну сторону, то в другую. Поэтому η нужно уменьшать со временем, например иногда делить пополам. Записывать это в алгоритм излишне, и мы реализуем эту особенность в тексте собственно программы (строки 8–10), а в алгоритм вносить не будем.

Листинг 3.4. Градиентный спуск на языке Python

```
def PerceptronGradientDescent(eta0,x,NUMBER_OF_STEPS):
    import random
    eta,w,deltaw=eta0,[],[]
    for i in xrange(len(x[0])):
        w.append((random.randrange(-5,5))/50.0)
        deltaw.append(0)
    for i in xrange(NUMBER_OF_STEPS):
        if ((NUMBER_OF_STEPS > 100) and
            (i % (NUMBER_OF_STEPS/5))==0):
```

```
            eta = eta/2.0
        for i in xrange(len(w)): deltaw[i]=0
        for xj in x:
            t,o,curx=xj[0],0,[1]+xj[1:len(xj)]
            for i in xrange(len(w)): o+=w[i]*curx[i]
            for i in xrange(len(w)):
                deltaw[i]+=eta*(t-o)*curx[i]
        for i in xrange(len(w)): w[i]+=deltaw[i]
    return w
```

Как видно, здесь нет ничего нового для нас в смысле программирования. Но вот следующий листинг уже представляет значительный интерес; в нём мы впервые столкнёмся с языком ML. Язык ML, как и Prolog, — функциональный язык; ML означает «metalanguage», а язык создавался специально как внутренний язык для одного из автоматических доказателей. Главная причина, по которой ML стал чем-то большим, чем ещё один малоизвестный функциональный язык, — это появление его потомка Caml (ML тут уже расшифровывается подругому: Categorical Abstract Machine Language), который позднее был расширен до Ocaml (Objective Caml), стал объектноориентированным и сейчас достаточно активно используется. Вот как вычисляется факториал на ML (обратите внимание, что, в отличие от Prolog, ML «мыслит» функциями, а не предикатами):

```
fun fac 0 = 1
  | fac n = n * fac(n-1)
```

Подробнее о языках семейства ML (в основном об Ocaml) можно прочесть в [26, 29], а мы сейчас перейдём к обучению перцептрона.

Листинг 3.5. Градиентный спуск на языке ML

```
structure grad_descent : sig
    val main : unit -> unit;
end =
struct
    fun scalar_product((x : real) :: xt, (y : real) :: yt) =
        case xt of
            [] => x * y
          | xx :: xtt => x * y + scalar_product(xt, yt)
```

```
fun sigma(x : real) = 1.0 / (1.0 + Math.exp(0.0 - x))
fun perc_value(w:real list, x:real list) =
      sigma(scalar_product(w, x))
fun adjust(eta : real, (wi : real) :: wt,
          (xi : real) :: test, answer : real,
          perc_ans : real) =
    case wt of
        [] => [(wi + eta * perc_ans *
                (1.0-perc_ans) * (answer-perc_ans) * xi)]
      | wii :: wtt => (wi + eta*perc_ans*
        (1.0-perc_ans)*(answer-perc_ans)*xi)
        ::
        adjust(eta, wt, test, answer, perc_ans)
fun one_step(eta : real, w : real list,
             (cur_test : real list) :: xt,
             (cur_ans : real) :: t) =
    case xt of
        [] => adjust(eta, w, cur_test,
                     cur_ans, perc_value(w, cur_test))
      | ct :: xtt => let
                val new_w = adjust(eta, w, cur_test,
                  cur_ans, perc_value(w, cur_test))
            in
                one_step(eta, new_w, xt, t)
            end
fun gradient_descent(eta : real, number_of_steps : int,
  w : real list, x : real list list, t : real list) =
    case number_of_steps of
        0 => w |
        n => let
                val new_w = gradient_descent(eta,
                  n - 1, w, x, t)
            in
                one_step(eta, new_w, x, t)
            end
fun random_list 0 = []
| random_list n = let
    val gen = System.Random()
in
    gen.#NextDouble() :: random_list(n - 1)
end
fun print_list [] = ()
    | print_list ((x:real)::xs) =
    (print (Real.toString x); print " "; print_list xs)
fun main () =
let
    val test1 = [1.0, 1.0, 1.0, 0.0]
    val ans1 = 1.0
```

```
        val test2 = [1.0, 0.0, 0.0 - 1.0, 1.0]
        val ans2 = 0.0
        val init = random_list(4)
    in
        print_list (
            gradient_descent(0.001, 10000, init,
                             [test1, test2], [ans1, ans2])
        )
    end
end
```

Пример 3.4. Обучение перцептрона градиентным спуском

Попытаемся выразить перцептроном линейную форму $x_1 - x_2 + x_3$. Для этого вызовем процедуру из листинга 3.4 так:

```
print PerceptronGradientDescent(0.1, [[1,1,1,1], [1,0,-1,0],
    [1,1,0,0], [1,0,0,1], [0,0,1,1], [2,1,0,1]], 1000)
```

(подаём на вход константу обучения 0,1, вектор из шести тестовых примеров и количество итераций 1000). Тогда на выходе мы получим вектор

$$[w_0, w_1, w_2, w_3] = [0{,}00078384114974986135,$$
$$0{,}99962820707427769,\ -0{,}99942100811538592,\ 0{,}99902715041783252],$$

что уже является очень хорошим приближением к искомой линейной форме.

На рис. 3.7 изображена стохастическая модификация алгоритма градиентного спуска. Этот алгоритм может преодолевать локальные минимумы (у рассматривавшегося нами до сих пор параболоида локальных минимумов быть не может, но в более сложных случаях они встречаются). Его отличие в том, что он не собирает всю информацию со всех тестовых примеров, а модифицирует веса сразу же, после каждого примера.

§ 3.6. Нелинейные перцептроны. Сигмоид.

Теперь, когда мы разобрались с обучением одного перцептрона, нужно научиться обучать (простите за тавтологию) целые сети перцептронов. Однако для этого нам придётся снова изменить функцию, вычисляемую перцептронами. Дело в том,

GradientDescent($\eta, \{x_i^j, t^j\}_{i=1,j=1}^{n,m}$)

1. Инициализировать $\{w_i\}_{i=0}^{n}$ маленькими случайными значениями.
2. Повторить NUMBER_OF_STEPS раз:
   a) Для всех j от 1 до m:
      (i) Для всех i от 1 до n

$$w_i = w_i + \eta \left(t^j - \sum_0^n w_i x_i^j \right) x_i^j.$$

3. Выдать значения $w_0, w_1, \ldots, w_n$.

Рис. 3.7. Алгоритм стохастического градиентного спуска

что те линейные перцептроны без порога активации, для которых у нас уже есть отличный метод градиентного спуска, для сетей не годятся. Суперпозиции линейных функций снова линейны, и сеть любой глубины можно было бы заменить одним-единственным перцептроном. Лучше обстоят дела у перцептронов с порогом активации: там уже увеличение глубины с 1 до 2 приносит реализацию всех булевских функций. Но с ними другая проблема: функция, которую они реализуют, лишь кусочно-гладкая, и требующий дифференцируемости метод градиентного спуска не сработает.

Решение, разумеется, простое: нужно «сгладить» линейный результат суммирования с весами w_i, т.е. в качестве выхода перцептрона использовать гладкую монотонную, но не линейную функцию. Для этого обычно используют *сигмоид* — функцию вида

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Применять мы её будем после того, как подсчитаем скалярное произведение $\sum_i w_i x_i$. То есть общая формула работы перцептрона такова:

$$o(x_1, \ldots, x_n) = \frac{1}{1 + e^{-\sum_{i=1}^n w_i x_i}}.$$

Стоит отметить, что приведённая выше $\sigma(x)$ — это не единственная функция, носящая гордое имя сигмоида. Сигмоид-кривые так называются просто потому, что имеют форму буквы «S»; избранная нами функция происходит из так называемой *логистической функции*

$$Z(\theta) = \frac{e^{\theta}}{1 + e^{\theta}},$$

которая используется, кроме нейронных сетей и статистики, для моделирования, например, роста популяции в условиях ограниченных ресурсов или роста опухоли в организме.[1] Но для тех же целей можно применять и *функцию Гомперца*

$$y(t) = ae^{be^{ct}}$$

с некоторыми отрицательными параметрами b и c. А можно пользоваться и обычным арктангенсом, гиперболическим тангенсом или функцией $f(x) = \frac{x}{\sqrt{1+x^2}}$. Мы в дальнейшем будем рассматривать сигмоид $\sigma(x) = \frac{1}{1+e^{-x}}$; приведённые ниже формулы легко модифицировать и для других аналогичных функций (хотя они могут потерять изрядную долю своего изящества).

Наш сигмоид обладает важным преимуществом: от него легко считать производную. Верна формула:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

Поэтому правило изменения веса в случае одного перцептрона (на одном тестовом примере) будет выглядеть так:

$$w_i := w_i + \eta o(x)(1 - o(x))(t(x) - o(x))x_i,$$

где $t(x)$ — целевое значение интересующей нас функции. Впрочем, все эти сложности мы вводили не для того, чтобы ограничиваться одним перцептроном.

§ 3.7. Алгоритм обратного распространения ошибки

Итак, отныне и впредь у нас не один перцептрон, а целая их сеть, примерно такая, как показано на рис. 3.8. С точки зрения структуры нейронная сеть обычно представляет собой граф, вершины которого разделены на несколько уровней, и между

[1] Впрочем, это одно и то же: опухоль — это и есть популяция клеток, растущая в условиях ограниченных ресурсов окружающего организма.

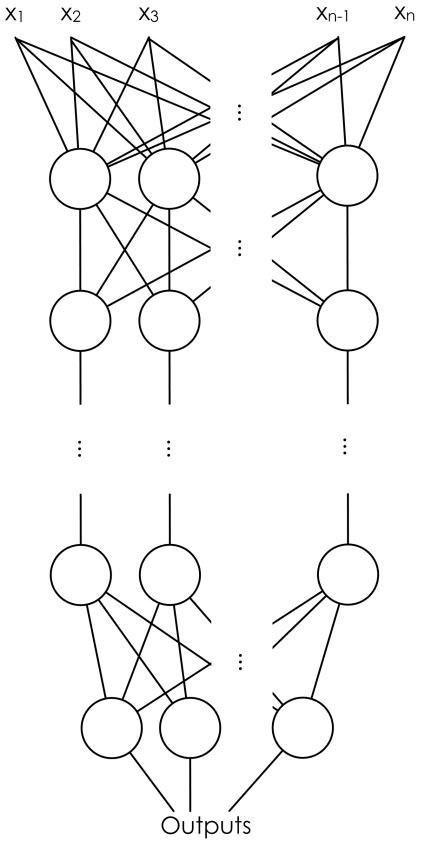


Рис. 3.8. Нейронная сеть

всеми вершинами каждой пары соседних уровней проведены все рёбра (рис. 3.8 это и иллюстрирует). Если на самом деле окажется, что какие-то входы не должны оказывать влияние на некоторые перцептроны, эти перцептроны просто обучат себе нулевые веса в нужных местах.

У нейронной сети есть входы $x_1, \ldots, x_n$, выходы Outputs и внутренние узлы. Перенумеруем все узлы (включая входы и выходы) числами от 1 до N. Обозначим через w_{ij} вес, стоящий на ребре, соединяющем i-й и j-й узлы, а через o_i — выход i-го узла. Поскольку выходов теперь несколько, функция ошибки станет сложнее. Если у нас, как и прежде, m тестовых примеров с

целевыми значениями выходов $\{t_k^d\}_{,d=1..m,k\in Outputs}$, то функция ошибки выглядит так:

$$E(\{w_{ij}\}) = \frac{1}{2}\sum_{d=1}^{m}\sum_{k\in Outputs}\left(t_k^d - o_k(x_1^d,\ldots,x_n^d)\right)^2.$$

Как модифицировать веса? Мы будем реализовывать стохастический градиентный спуск, то есть будем подправлять веса после каждого тестового примера. Как и раньше, нам нужно двигаться в сторону, противоположную градиенту, то есть добавлять к каждому весу w_{ij}

$$\Delta w_{ij} = -\eta\frac{\partial E^d}{\partial w_{ij}}, \text{ где } E^d\left(\{w_{ij}\}\right) = \frac{1}{2}\sum_{k\in Outputs}\left(t_k^d - o_k^d\right)^2.$$

Как подсчитать эту производную? Пусть сначала интересующий нас вес входит в перцептрон последнего уровня, то есть $j \in Outputs$. Сначала отметим, что w_{ij} влияет на выход перцептрона только как часть суммы $S_j = \sum_i w_{ij}x_{ij}$, где сумма берётся по входам j-го узла. Поэтому

$$\frac{\partial E^d}{\partial w_{ij}} = \frac{\partial E^d}{\partial S_j}\frac{\partial S_j}{\partial w_{ij}} = x_{ij}\frac{\partial E^d}{\partial S_j}.$$

Аналогично, S_j влияет на общую ошибку только в рамках выхода j-го узла o_j (напоминаем, что это выход всей сети). Поэтому (по формуле дифференцирования сложной функции)

$$\frac{\partial E^d}{\partial S_j} = \frac{\partial E^d}{\partial o_j}\frac{\partial o_j}{\partial S_j} = \left(\frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in Outputs}(t_k - o_k)^2\right)\left(\frac{\partial\sigma(S_j)}{\partial S_j}\right) =$$

$$= \left(\frac{1}{2}\frac{\partial}{\partial o_j}(t_j - o_j)^2\right)(o_j(1-o_j)) = -o_j(1-o_j)(t_j - o_j).$$

Если же j-й узел — не на последнем уровне, то у него есть выходы; обозначим их через $ch(j)$. В этом случае

$$\frac{\partial E^d}{\partial S_j} = \sum_{k\in ch(j)}\frac{\partial E^d}{\partial S_k}\frac{\partial S_k}{\partial S_j},$$

и

$$\frac{\partial S_k}{\partial S_j} = \frac{\partial S_k}{\partial o_j}\frac{\partial o_j}{\partial S_j} = w_{jk}\frac{\partial o_j}{\partial S_j} = w_{jk}o_j(1-o_j).$$

Здесь $\frac{\partial E^d}{\partial S_k}$ — это в точности аналогичная поправка, но вычисленная для узла следующего уровня (будем обозначать её через δ_k — от Δ_k она отличается отсутствием множителя $-\eta x_{ij}$). Поскольку мы научились вычислять поправку для узлов последнего уровня и выражать поправку для узла более низкого уровня через поправки более высокого, можно уже писать алгоритм. Именно из-за этой особенности вычисления поправок он называется *алгоритмом обратного распространения ошибки* (backpropagation). Краткое резюме проделанной работы:

— для узла последнего уровня

$$\delta_j = -o_j(1 - o_j)(t_j - o_j);$$

— для внутреннего узла сети

$$\delta_j = -o_j(1 - o_j) \sum_{\text{Outputs}(j)} \delta_k w_{jk};$$

— после предварительного вычисления вспомогательных переменных δ_j для всех узлов

$$\Delta w_{ij} = -\eta \delta_j x_{ij}.$$

Получающийся алгоритм представлен на рис. 3.9. На вход алгоритму, кроме указанных на рис. 3.9 параметров, нужно также подавать в каком-нибудь формате структуру сети. На практике очень хорошие результаты показывают сети достаточно простой структуры, состоящие всего лишь из двух уровней нейронов — скрытого уровня (hidden units) и нейронов-выходов (output units); каждый вход сети соединён со всеми скрытыми нейронами, а результат работы каждого скрытого нейрона подаётся на вход каждому из нейронов-выходов. В таком случае достаточно подавать на вход число нейронов скрытого уровня.

§ 3.8. Реализация нейронной сети на Java

В этом параграфе мы рассмотрим пример реализации алгоритма обратного распространения ошибки на языке Java. Начнём со вспомогательного класса `Perceptron`; этот класс создаёт и поддерживает перцептрон с произвольным числом весов, а также умеет его случайно инициализировать. Сигмоид-функция здесь вычисляется в функции `getOutput()`.

BackPropagation($\eta, \{x_i^d, t^d\}_{i=1,d=1}^{n,m}$, NUMBER_OF_STEPS)

1. Инициализировать $\{w_{ij}\}_{i,j}$ маленькими случайными значениями.
2. Повторить NUMBER_OF_STEPS раз:
   a) Для всех d от 1 до m:
      (i) Подать $\{x_i^d\}$ на вход сети и подсчитать выходы o_i каждого узла.
      (ii) Для всех $k \in$ Outputs
      $$\delta_k = o_k(1 - o_k)(t_k - o_k).$$
      (iii) Для каждого уровня l, начиная с предпоследнего:
         (A) Для каждого узла j уровня l вычислить
         $$\delta_j = o_j(1 - o_j) \sum_{k \in \mathrm{Children}(j)} w_{jk}\delta_k.$$
      (iv) Для каждого ребра сети $\{ij\}$
      $$w_{ij} = w_{ij} + \eta\delta_j x_{ij}.$$
3. Выдать значения w_{ij}.

Рис. 3.9. Алгоритм обратного распространения ошибки

Листинг 3.6. Нейронная сеть на Java: класс Perceptron

```
class Perceptron {
    private double[] weights;
    public Perceptron(int inputCount) {
        weights = new double[inputCount];
        for (int i = 0; i < weights.length; i++) {
            weights[i] = Math.random();
        }
    }

    public int getInputCount() {
        return weights.length;
    }

    public double getWeight(int i) {
        return weights[i];
    }
```

```
    void setWeight(int i, double value) {
        weights[i] = value;
    }

    public double getOutput(double[] input) {
        double sum = 0;
        for (int i = 0; i < input.length; i++) {
            sum += weights[i] * input[i];
        }
        return 1.0 / (1.0 + Math.exp(-sum));
    }
}
```

Основной класс программы — класс NeuralNetwork, в котором происходят все основные операции с нейронной сетью; в частности, этот класс реализует обучение сети на заданных тестовых примерах и подсчёт результатов работы сети.

Листинг 3.7. Нейронная сеть на Java: класс NeuralNetwork

```
public class NeuralNetwork {
    // перцептроны
    private final Perceptron[][] percs;
    // константа обучения
    private final double eta = 0.15;

    /**
     * Создаёт нейронную сеть с заданным количеством
     * входов и выходов; сеть может быть произвольной
     * глубины.
     * @param perceptronNumber число перцептронов на уровне
     * @param inputCount число входов
     */
    public NeuralNetwork(int[] perceptronNumber,
                         int inputCount) {
        if (perceptronNumber.length == 0) {
            throw new IllegalArgumentException(
              "wrong number of layers");
        }
        if (inputCount <= 0) {
            throw new IllegalArgumentException(
              "wrong number of inputs");
        }
        for (int i = 0; i < perceptronNumber.length; i++) {
            if (perceptronNumber[i] <= 0) {
                throw new IllegalArgumentException(
```

```
                "wrong number of perceptrons in a layer");
        }
    }
    percs = new Perceptron[perceptronNumber.length][];
    for (int i = 0; i < percs.length; i++) {
        percs[i] = new Perceptron[perceptronNumber[i]];
        int num = (i == 0) ?
          inputCount : percs[i - 1].length;
        for (int j = 0; j < percs[i].length; j++) {
            percs[i][j] = new Perceptron(num);
        }
    }
}

/**
 * Обучает сеть по заданному набору тестовых примеров.
 * @param numberOfSteps количество шагов
 * @param inputs входы тестовых примеров
 * @param answers верные значения выходов
 */
public void train(int numberOfSteps, double[][] inputs,
                  double[][] answers) {
    if (inputs.length != answers.length) {
        throw new IllegalArgumentException(
          "no. of inputs does not match no. of answers");
    }
    for (int i = 0; i < inputs.length; i++) {
        if (inputs[i].length !=
            percs[0][0].getInputCount()) {
            throw new IllegalArgumentException(
              "wrong number of inputs");
        }
    }
    for (int i = 0; i < answers.length; i++) {
        if (answers[i].length !=
            percs[percs.length - 1].length) {
            throw new IllegalArgumentException(
              "wrong number of answers");
        }
    }
    for (int i = 0; i < numberOfSteps; i++) {
        for (int j = 0; j < inputs.length; j++) {
            train(inputs[j], answers[j]);
        }
    }
}

private void train(double[] inputs, double[] answers) {
```

```
    double[][] outputs = calculateOutputs(inputs);
    final List<Double> deltas = new ArrayList<Double>();
    final List<Double> previousDeltas =
      new ArrayList<Double>();
    for (int i = percs.length - 1; i >= 0; i--) {
        for (int j = 0; j < percs[i].length; j++) {
            // подсчёт delta
            double delta;
            final double output = outputs[i][j];
            if (i == percs.length - 1) {
                delta = answers[j] - output;
            } else {
                delta = 0;
                for (int k = 0;
                     k < percs[i + 1].length; k++) {
                    delta += previousDeltas.get(k) *
                             percs[i + 1][k].getWeight(j);
                }
            }
            delta *= output * (1 - output);
            deltas.add(delta);
            // изменение весов
            for (int k = 0;
                 k < percs[i][j].getInputCount(); k++) {
                final double in = (i == 0) ? inputs[k]
                                           : outputs[i - 1][k];

                percs[i][j].setWeight(k,
                  percs[i][j].getWeight(k) +
                  eta * delta * in);
            }
        }
        previousDeltas.clear();
        previousDeltas.addAll(deltas);
        deltas.clear();
    }
}

private double[][] calculateOutputs(double[] inputs) {
    final double[][] result = new double[percs.length][];
    for (int i = 0; i < percs.length; i++) {
        result[i] = new double[percs[i].length];
        for (int j = 0; j < percs[i].length; j++) {
            final double[] in = (i == 0) ?
              inputs : result[i - 1];
            result[i][j] = percs[i][j].getOutput(in);
        }
    }
```

```
        return result;
    }

    /**
     * Вычисление выхода сети по заданному входу
     * @param inputs входные значения
     * @return network выходные значения
     */
    public double[] getOutput(double[] inputs) {
        if (inputs.length != percs[0][0].getInputCount()) {
            throw new IllegalArgumentException("
              wrong number of inputs");
        }
        return calculateOutputs(inputs)[percs.length - 1];
    }
}
```

Последний, собственно запускаемый класс Main демонстрирует, как использовать класс NeuralNetwork. Мы специально приводим в коде много реальных примеров того, как можно протестировать нейронную сеть; если вы когда-нибудь будете реализовывать нейронную сеть самостоятельно, то мы рекомендуем вам присмотреться к этим примерам и убедиться, что ваша сеть тоже адекватно обрабатывает простейшие тесты.

Листинг 3.8. Нейронная сеть на Java: класс Main

```
public class Main {

    private final static int NUMBER_OF_STEPS = 10000;
    public static void main(String[] args) {
        testConstant();
        testAnd();
        testBigFunction();
        testRatio();
        testSum();
        testProduct();
    }

    // обучение константы
    private static void testConstant() {
        double[][] inputs = {{1}};
        double[][] answers = {{1}};
        NeuralNetwork network =
          new NeuralNetwork(new int[]{1}, 1);
```

```
        network.train(NUMBER_OF_STEPS, inputs, answers);
        System.out.println("Function: 1");
        for (int i = 0; i < inputs.length; i++) {
            System.out.println("answer: " + answers[i][0] +
            ";\t result: "+network.getOutput(inputs[i])[0]);
        }
        System.out.println();
    }

    // обучение конъюнкции
    private static void testAnd() {
        double[][] inputs = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
        double[][] answers = {{0}, {0}, {0}, {1}};
        NeuralNetwork network =
          new NeuralNetwork(new int[]{2, 3, 1}, 2);
        network.train(NUMBER_OF_STEPS, inputs, answers);
        System.out.println("Function: x1 and x2");
        for (int i = 0; i < inputs.length; i++) {
            System.out.println("answer: " + answers[i][0] +
            ";\t result: "+network.getOutput(inputs[i])[0]);
        }
        System.out.println();
    }

    // обучение булевской функции (x1 and x2) or x3
    private static void testBigFunction() {
        double[][] inputs = {{0, 0, 0}, {0, 0, 1},
                             {0, 1, 0}, {0, 1, 1},
                             {1, 0, 0}, {1, 0, 1},
                             {1, 1, 0}, {1, 1, 1}};
        double[][] answers = {{0}, {1},
                              {0}, {1},
                              {0}, {1},
                              {1}, {1}};
        NeuralNetwork network =
          new NeuralNetwork(new int[]{2, 3, 1}, 3);
        network.train(NUMBER_OF_STEPS, inputs, answers);
        System.out.println("Function: (x1 and x2) or x3");
        for (int i = 0; i < inputs.length; i++) {
            System.out.println("answer: " + answers[i][0] +
            ";\t result: "+network.getOutput(inputs[i])[0]);
        }
        System.out.println();
    }

    // обучение функции обращения числа
    private static void testRatio() {
        double[][] inputs = {{1}, {2}, {3}, {4}, {5}};
```

```
        double[][] answers =
          {{1.0}, {0.5}, {0.33}, {0.25}, {0.2}};
        NeuralNetwork network =
          new NeuralNetwork(new int[]{1, 5, 1}, 1);
        network.train(10 * NUMBER_OF_STEPS, inputs, answers);
        System.out.println("Function: 1/x");
        for (int i = 0; i < inputs.length; i++) {
            System.out.println("answer: " + answers[i][0] +
            ";\t result: "+network.getOutput(inputs[i])[0]);
        }
        System.out.println();
    }

    // обучение функции суммирования двух аргументов
    private static void testSum() {
        double[][] inputs = {{0.1, 0.0}, {0.2, 0.1},
          {0.3, 0.4}, {0.5, 0.5}, {0.7, 0.1}};
        double[][] answers =
          {{0.1}, {0.3}, {0.7}, {1.0}, {0.8}};
        NeuralNetwork network =
          new NeuralNetwork(new int[]{2, 3, 1}, 2);
        network.train(10 * NUMBER_OF_STEPS, inputs, answers);
        System.out.println("Function: x1 + x2");
        for (int i = 0; i < inputs.length; i++) {
            System.out.println("answer: " + answers[i][0] +
            ";\t result: "+network.getOutput(inputs[i])[0]);
        }
        System.out.println();
    }

    // обучение функции произведения двух аргументов
    private static void testProduct() {
        double[][] inputs = {{0.1, 0.0}, {0.2, 0.1},
          {0.3, 0.4}, {0.5, 0.5}, {0.7, 0.1}};
        double[][] answers =
          {{0.0}, {0.02}, {0.12}, {0.25}, {0.07}};
        NeuralNetwork network =
          new NeuralNetwork(new int[]{2, 5, 1}, 2);
        network.train(10*NUMBER_OF_STEPS, inputs, answers);
        System.out.println("Function: x1 * x2");
        for (int i = 0; i < inputs.length; i++) {
            System.out.println("answer: " + answers[i][0] +
              ";\t result: "+network.getOutput(inputs[i])[0]);
        }
        System.out.println();
    }
}
```

§ 3.9. Заключение

В этой главе мы обсудили отдельные перцептроны, нейронные сети и обучение их параметров. Конечно, мы лишь слегка коснулись всего разнообразия методов, подходов и применений, которые нейронные сети накопили за свою долгую историю.

Одно очень интересное развитие идеи нейронных сетей — самоорганизующиеся обучающиеся системы, которые получаются, если объединить нейроны в двумерную решётку, а не просто расположить их в линейном порядке послойно. К таким системам относятся сети Хопфилда и карты Кохонена.

Самоорганизующиеся карты Кохонена решают любопытную и важную задачу: они проецируют многомерные данные в пространство низшей размерности (обычно размерности два, чтобы человек мог в полной мере насладиться результатом). Карта Кохонена является решёткой, состоящей из нейронов (чем больше, тем лучше); она сначала строится по размерности входных данных, а затем каждый тестовый пример начинает влиять на ближайшие к нему нейроны, модифицируя их «под себя». А сами нейроны начинают модифицировать друг друга. В результате получается, что многомерные кластеры схлопываются в двумерные области, но между этими двумерными областями всё ещё сохраняются весьма плавные переходы; процесс успешно сходится [64, 89, 90, 134].

А сети Хопфилда и того интереснее: они реализуют ассоциативную память, в которой сеть (по форме она опять представляет собой двумерную решётку) «запоминает» несколько желаемых состояний-ассоциаций (они становятся локальными минимумами энергии сети) и затем сходится к одному из них (или к другому минимуму) из любого начального состояния. Таким образом становится возможным моделировать процессы ассоциативного вспоминания, когда ассоциаций несколько, и всплывает самая близкая из них [67, 134].

Теория нейронных сетей, конечно, гораздо шире; здесь мы лишь поверхностно коснулись самых её азов. Читателю, желающему ознакомиться с ней подробнее, мы рекомендуем [64, 134], а пока приглашаем в следующую главу, в которой речь пойдёт о генетических алгоритмах.

Глава 4

Генетические алгоритмы

> Если дарвинизм в том виде, в каком он вышел из-под пера Дарвина, находился в противоречии с идеалистическим мировоззрением, то развитие материалистического учения ещё более углубляло это противоречие. Поэтому реакционные биологи сделали всё от них зависящее, чтобы выбросить из дарвинизма его материалистические элементы. Отдельные голоса прогрессивных биологов, вроде К. А. Тимирязева, тонули в дружном хоре антидарвинистов из лагеря реакционных биологов всего мира. В последарвиновский период подавляющая часть биологов мира, вместо дальнейшего развития учения Дарвина, делали всё, чтобы опошлить дарвинизм, удушить его научную основу. Наиболее ярким олицетворением такого опошления дарвинизма являются учения Вейсмана, Менделя, Моргана, основоположников современной генетики.
>
> *Из речи на сессии ВАСХНИЛ 1948 г.* [172]
> *Т. Д. Лысенко*

Винни-Пух и Кристофер Робин очень любили играть в «Пустяки»[1]. Суть игры проста: мальчик и медвежонок выбирали себе по палочке и бросали их с моста в реку, а потом переходили на другую

[1] Гениальный перевод Заходера — от слова «Poohsticks».

сторону моста и смотрели, чья палочка быстрее выплывет. Мальчик обычно выигрывал, но однажды Винни-Пух, прочитав книгу «Почему вы проигрываете в пустяки», решил как следует потренироваться и научиться-таки обыгрывать Кристофера Робина.

И работа закипела. Единственный элемент игры, который игроки контролируют, — форма палочек. Мудрый Пух решил, что провести гидродинамические расчёты течения и математически вычислить оптимальную форму — это не для медвежонка, у которого в голове опилки. Оставалось действовать методом проб и ошибок.

Винни вырезал себе кучу палочек и стал проводить эксперименты. Для этого он брал по две палочки, бросал их в реку и смотрел, которая из них выплывет раньше. Так он отсеял половину палочек. Затем Пух подумал, что надо, наверное, попробовать и другие формы палочек — но ведь эти уже такие замечательные, они победили своих оппонентов! Пух посмотрел на две палочки из победительниц и увидел на одной из них оч-чень любопытный сучок, а на другой — весьма подозрительную выемку. Винни решил сделать просто: вырезал себе новую палочку, которую сам снабдил и сучком, и выемкой. Так же он поступил и с другими интересными особенностями, поначалу попадавшимися случайно. Кроме того, в процессе вылавливания палочек некоторые из них слегка повредились — где-то отлетел кусочек коры, а где-то палочка и просто сломалась пополам. Но Пух решил быть честным и давать шанс даже в таких случаях.

Неделя прошла в напряжённой работе. Пух вырезал палочки, бросал их в реку, сравнивал, подправлял, менял и снова вырезал...

И когда в очередное воскресенье Кристофер Робин и Винни стали играть в пустяки, мальчик просто опешил: Винни поднял с земли ветку и быстрыми, уверенными движениями вырезал из неё нечто совершенно несусветное, напоминавшее то ли запутавшегося в собственных щупальцах маленького осьминога, то ли китайские игрушки слоновой кости — резной шар в резном шаре в резном шаре, и никакого клея, из единого цельного куска. Творение медвежонка побило рекорд, державшийся с предыдущей весны, когда две недели подряд лил дождь и река была куда быстрее, чем сейчас.

Чтобы обыграть Кристофера Робина, медвежонок пользовался самым настоящим генетическим алгоритмом, с кроссовером и мутациями. А отбор Пух проводил турнирным методом. О том, как это применять в более серьёзном контексте, мы сейчас и поговорим.

§ 4.1. Введение

Предыдущая глава была посвящена искусственным нейронным сетям. В основе этих сетей, как мы уже обсуждали, лежит структура человеческого мозга: исследователи решили её скопировать (хотя бы приблизительно) и натолкнулись на настоящую золотую жилу. Результаты до сих пор применяются в самых разнообразных областях искусственного интеллекта, от компьютерных игр до распознавания текстов.

Но это не единственный случай, когда идеи алгоритмов машинного обучения были скопированы с природы. В этой главе речь пойдёт о *генетических алгоритмах* [44, 112, 168] — аппарате, списанном с идей эволюции, высказанных ещё Чарльзом Дарвином[1].

[1] Чарльз Дарвин (Charles Darwin, 1809–1882) — английский натуралист, заложивший основы теории эволюции. Во время своего знаменитого кругосветного путешествия на «Бигле» (заметим в скобках, что Дарвин жестоко страдал от морской болезни) он собрал огромное количество биологических и палеонтологических данных и, что главное, сумел обобщить их в единую теорию эволюции, которую и опубликовал в книге «Происхождение видов путём естественного отбора, или сохранение благоприятствуемых пород в борьбе за жизнь» (1859). Говоря в терминах Томаса Куна, Дарвин вместе со своим тёзкой Лайелем несёт ответственность за научную революцию XIX века — переход от картины стабильного, созданного раз и навсегда мира к миру эволюционирующему, постоянно изменяющемуся, находящемуся в динамическом равновесии.

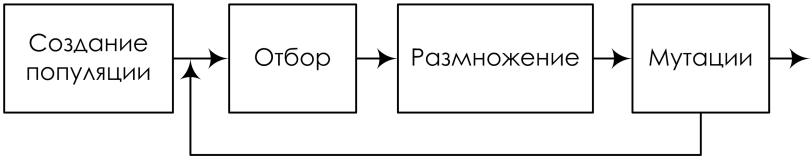


Рис. 4.1. Общая схема генетического алгоритма

Как известно, основная идея дарвиновской эволюции заключается в *естественном отборе*: наименее приспособленные организмы умирают раньше и в больших количествах, наиболее приспособленные выживают и дают потомство. Их потомство уже оказывается в среднем более приспособленным к окружающей среде, но среди них опять выделяются наиболее приспособленные особи, и так далее.

В генетическом алгоритме происходит абсолютно то же самое. Есть некоторое пространство гипотез, из которых мы должны выбрать лучшую. Есть функция приспособленности Fitness (её ещё называют goodness of fit), которая определяет, насколько хорошо тот или иной организм приспособлен к «окружающей среде». Есть набор генетических операций, при помощи которых на свет появляются новые особи. И, наконец, есть некоторое целевое значение $\text{Fitness}_{\max}$, к которому мы стремимся — как только мы его достигнем, работу алгоритма можно будет прекратить[1]. На рис. 4.1 изображена общая схема генетического алгоритма. Рассмотрим её подробнее.

§ 4.2. Схема генетического алгоритма в деталях

1. *Инициализация.* Как говорил кот Матроскин, прежде чем продать что-нибудь ненужное, нужно купить что-нибудь ненужное.[2] Прежде чем отбросить часть популяции, не выдержавшую естественного отбора, нужно её

[1] Разумеется, можно предложить и другие условия остановки: запускать алгоритм на определённое число поколений, например.

[2] Не стоит забывать о том, что Матроскин после этого добавлял: «а у нас денег нет»; генетические операции, в том числе операцию инициализации, нужно реализовывать эффективно.

откуда-то породить. А перед первым шагом нужно создать некую начальную популяцию; даже если она окажется совершенно неконкурентоспособной, генетический алгоритм всё равно достаточно быстро переведёт её в жизнеспособную популяцию. Таким образом, на первом шаге обычно просто рассматривают случайно порождённых особей; достаточно, чтобы они соответствовали формату особей популяции, и на них можно было подсчитать функцию Fitness. Итогом первого шага является популяция $\mathcal{H}$, состоящая из N особей.

2. *Отбор.* На этапе отбора нужно из всей популяции выбрать некоторую её долю, которая останется «в живых» на этом этапе эволюции. Есть разные способы проводить отбор — о них мы подробно поговорим в § 4.5, а пока упомянем очевидное: вероятность выживания особи $h \in \mathcal{H}$ должна зависеть от значения функции приспособленности $\mathrm{Fitness}(h)$. Сама доля выживших s обычно является параметром генетического алгоритма, и её задают заранее. По итогам отбора из N особей популяции $\mathcal{H}$ должны остаться sN особей, которые войдут в итоговую популяцию $\mathcal{H}'$. Прочие, увы, не выживут.

3. *Размножение.* Размножение в генетических алгоритмах обычно половое — чтобы произвести потомка, нужно обычно двое родителей, хотя иногда бывает и больше. Размножение в разных алгоритмах определяется поразному — оно может зависеть от представления данных. Главное требование к размножению — чтобы потомок или потомки имели возможность унаследовать черты обоих родителей, «смешав» их каким-либо достаточно разумным способом.

   В § 4.3 мы подробно рассмотрим разные варианты операции размножения на битовых строках. А вообще говоря, для этапа размножения нужно выбрать случайным образом $\frac{(1-s)N}{2}$ пар особей из H и провести с ними размножение, получив по два потомка от каждой пары (если размножение определено так, чтобы давать одного потомка, нужно выбрать $(1-s)N$ пар), а затем добавить

этих потомков в $\mathcal{H}'$[1]. В результате $\mathcal{H}'$ будет состоять из N особей. Этот процесс проиллюстрирован на рис. 4.2.

Почему особи для размножения обычно выбираются из всей популяции $\mathcal{H}$, а не только из выживших на первом шаге элементов $\mathcal{H}'$? Дело в том, что главный бич многих генетических алгоритмов — недостаток разнообразия (diversity) в особях. Достаточно быстро выделяется один-единственный генотип, который представляет собой локальный максимум, а затем все элементы популяции проигрывают ему отбор, и вся популяция «забивается» копиями этой особи. Есть разные способы борьбы с таким нежелательным эффектом; один из них — выбор для размножения не самых приспособленных, но вообще всех особей.

4. *Мутации.* К мутациям относится всё то же самое, что и к размножению: есть некоторая доля мутантов m, являющаяся параметром генетического алгоритма. На шаге применения мутаций нужно выбрать из популяции mN особей, а затем каждую из них изменить в соответствии с некоторыми заранее определёнными операциями мутации.

О том, какими могут быть генетические операции на практике, мы поговорим в следующем параграфе.

§ 4.3. Генетические операции

Для начала давайте предположим, что элементы популяции уже зашифрованы в виде битовых строк (о том, как это сделать, мы поговорим в § 4.4). Как известно, с битовыми строками можно делать практически всё, что угодно. Строки легко разрезать на части, комбинируя затем эти части, легко менять полностью или частично, легко заменять в них отдельные биты. Давайте рассмотрим, как на строках битов работают стандартные генетические операции.

[1] Может показаться, что столько пар в популяции и вовсе нету, но обычно в генетических алгоритмах пары выбирают из всей популяции, не отказывая уже размножившимся особям в ещё одном шансе.

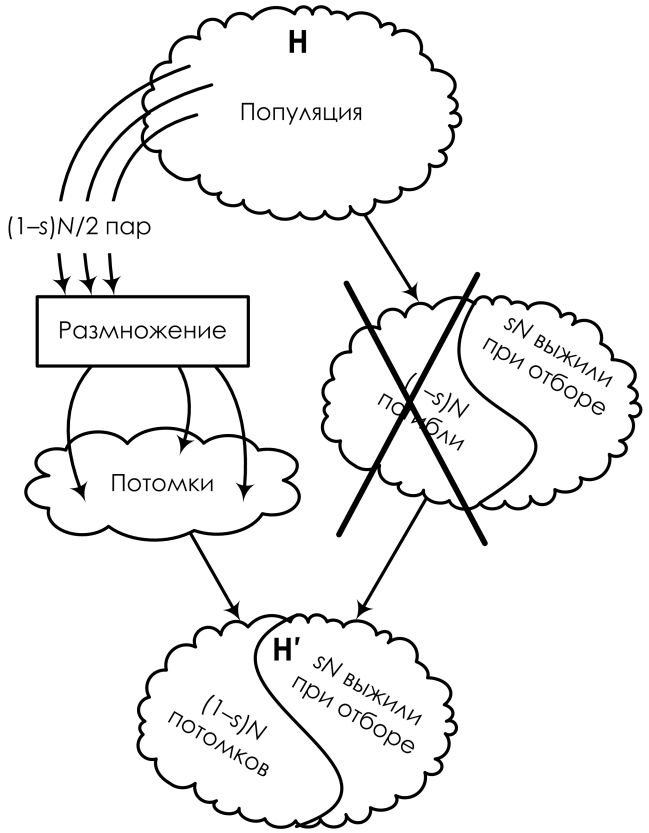


Рис. 4.2. Схема отбора на одном шаге

С инициализацией всё понятно: порождать случайные битовые строки несложно.[1] Для отбора конкретная форма представления данных не очень важна: достаточно уметь вычислять функцию приспособленности Fitness. Таким образом, остаются размножение и мутации.

[1] Ох, обманываем мы вас... сложно это, действительно сложно. Псевдослучайные генераторы — область уже довольно хорошо изученная, но алгоритмы там совсем не простые. Да и в ней, как и вообще во всей криптографии, нет ни одного безусловно доказанного утверждения [6, 51–53, 80]. Но программист может не задумываться о теории, скрытой за используемым интерфейсом, и для программиста это действительно легко: достаточно инициализировать генератор да вызвать функцию `random` (или как там она в вашем языке программирования называется).

При размножении особь должна унаследовать черты обоих предков. На битовых строках это можно реализовать достаточно естественным путём: собрать итоговую строку из частей строк-родителей. Такая операция называется *кроссовером* (crossover). Чтобы сделать кроссовер двух строк, нужно выбрать определённую *маску*, а затем в соответствии с этой маской выбрать те или иные биты родителей.

Определение 4.1. Результатом *кроссовера* с битовой *маской* $m_1 \dots m_n$ из двух битовых строк одинаковой длины $x_1 \dots x_n$ и $y_1 \dots y_n$ является строка $z_1 \dots z_n$, где

$$z_i = \begin{cases} x_i, & \text{если } m_i = 1, \\ y_i, & \text{если } m_i = 0. \end{cases}$$

В зависимости от того, как выбирается маска, различают несколько видов кроссовера.

1. *Одноточечный кроссовер* (single-point crossover). Простейший вид кроссовера; в нём случайно выбирается одна позиция в строке, и маска состоит из единиц левее этой позиции и нулей — правее (т.е. левее выбранной позиции потомок совпадает с первым родителем, а правее — со вторым). Например:

Исходные строки	Маска	Результат
1001101011, 0010**101100**	1111100000	1001101100

2. *Двухточечный кроссовер* (double-point crossover). То же самое, что и в предыдущем случае, но случайно выбираются две позиции, и между этими двумя позициями берутся биты одного из родителей, а вне выбранных позиций — другого. Если по каким-то причинам желательно, чтобы в потомке битов родителей было поровну, можно выбирать только одну позицию, а затем отсчитывать от неё ровно половину общей длины строки (возвращаясь в начало, если достигнут конец строки). Например:

Исходные строки	Маска	Результат
100**1101011**, **00**10101**00**	0001111100	0011101000

3. *Однородный кроссовер* (uniform crossover). Здесь маска выбирается случайным образом, равномерно. Например:

Исходные строки	Маска	Результат
1001101011, **0010101100**	0110100110	0000101010

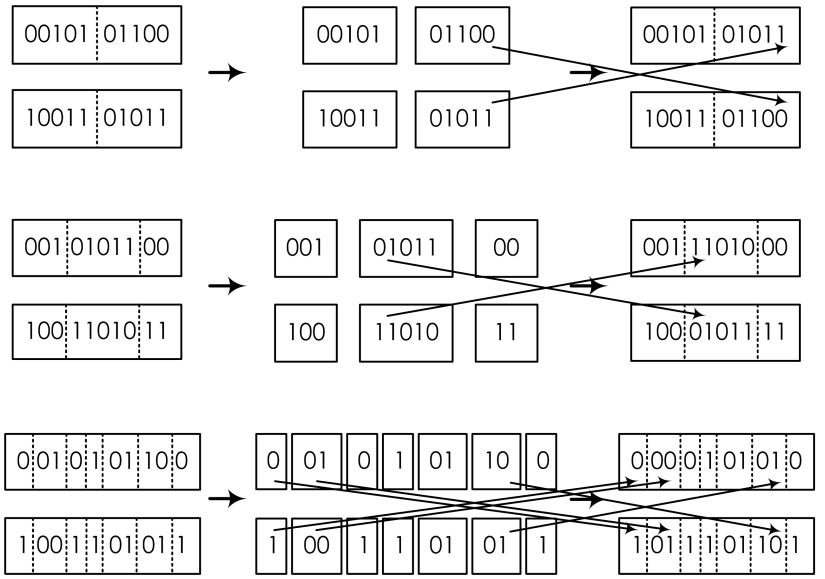


Рис. 4.3. Виды кроссовера. Сверху вниз — одноточечный кроссовер, двухточечный кроссовер, однородный кроссовер

На рис. 4.3 проиллюстрированы все эти виды кроссовера.

Итак, мы рассмотрели различные виды кроссовера на битовых строках. Осталось понять, какими могут быть мутации. Наиболее распространённая мутация — это, конечно, инвертирование случайного бита строки. Её уже достаточно для многих приложений. Для того чтобы определить другие типы мутаций, нужно располагать более подробными сведениями о структуре предметной области. Мы приведём пример более специфичной мутации в следующем параграфе, а пока отметим, что даже обычное инвертирование бита уже является очень мощным инструментом.

§ 4.4. Представление данных

В предыдущем параграфе мы предполагали, что элементы популяции генетического алгоритма — это строки битов. Действительно, в таком случае все генетические операции легко

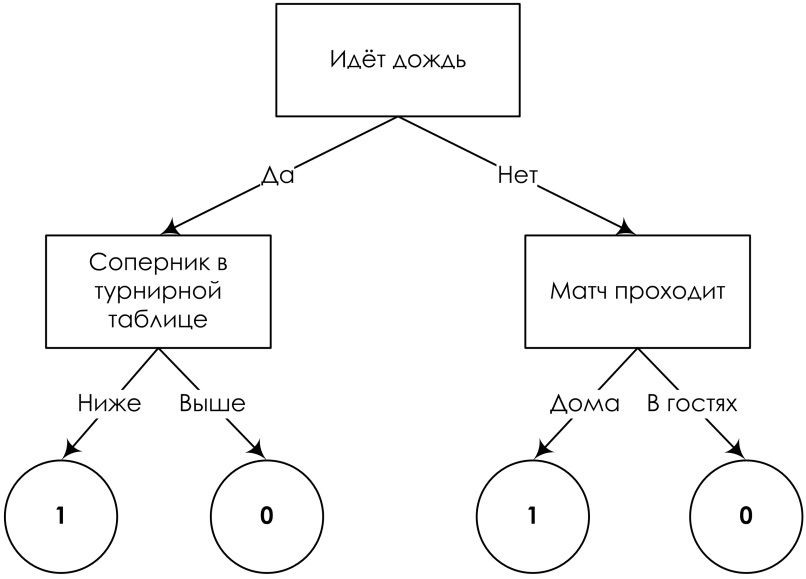


Рис. 4.4. Дерево принятия решений для игр «Зенита»

описываются и применяются, давая разумные результаты. Теоретически этого вполне достаточно: понятно, что любую, сколь угодно сложную структуру можно отобразить в слова алфавита из двух символов. Но как это сделать на практике?

Давайте вернёмся к задаче классификации из примера 1.1. Пусть результаты игр «Зенита» находятся в зависимости от четырёх параметров (см. Главу 1, где это описано более подробно). Тогда классификация, представленная в виде дерева, может быть записана в виде набора гипотез, соответствующих листьям дерева (точнее говоря, путям от корня к каждому из листьев). Например, крайняя правая ветка дерева с рис. 4.4 представляет собой гипотезу

$$(\text{Дождь} = \text{Нет}) \wedge (\text{Играет} = \text{В гостях}) \Rightarrow (\text{Победа} = 0).$$

Как закодировать гипотезу такого рода в виде строки битов? Первая мысль — просто выделить по одному биту на каждый (бинарный) атрибут и на целевую функцию. Например, в примере с играми «Зенита» четыре атрибута — турнирное положение соперника, место проведения матча, игра лидеров и погода, и

строка 00101 соответствовала бы гипотезе

$$(\text{Соперник} = \text{Ниже}) \wedge (\text{Играет} = \text{В гостях}) \wedge$$
$$\wedge (\text{Лидеры} = \text{На месте}) \wedge (\text{Дождь} = \text{Нет}) \Rightarrow (\text{Победа} = \text{Да}).$$

Это кодирование имеет ряд достоинств. Во-первых, оно весьма эффективно. Во-вторых, что также способствует успеху генетического алгоритма, все строки имеют одинаковую длину. Наконец, каждая строка заданной длины имеет смысл (т.е. описывает реальную гипотезу) — это важно для успешного размножения, когда строки начнут случайным образом перемешиваться. Однако его недостатки также видны невооружённым взглядом. Как представить гипотезу, в которой не все значения атрибутов специфицированы, а, наоборот, от некоторых из них ничего не зависит? Если это невозможно (а в нашем текущем представлении это невозможно), то любое дерево придётся «разворачивать» до полного набора гипотез (говоря языком логики высказываний, любую дизъюнктивную нормальную форму придётся дополнять до совершенной), что явно неэффективно. Можно было бы представлять гипотезы с атрибутами без фиксированных значений строками меньшей длины, но тогда было бы непонятно, какие атрибуты заданы, а какие — нет.

В данном случае общепринятым решением возникшей проблемы будет следующая конструкция. Для каждого атрибута нужно выделить столько битов, сколько у него возможных значений (мы до сих пор рассматривали только бинарные атрибуты, но ничто не помешает рассмотреть и атрибуты с бóльшим числом значений). Ноль в той или иной позиции означает, что данное значение *не* встречается в посылке правила, а единица — что встречается. В такой нотации, если все биты, соответствующие значениям данного атрибута, равны 1, это означает, что значение его (атрибута) не играет никакой роли для применения этого правила. Для значения функции мы оставим один бит — правила, где целевая функция не определена или разрешено любое её значение, лишены смысла. Таким образом, в случае примера с играми «Зенита» четыре бинарных атрибута дадут восемь бит посылки плюс один бит значения целевой

функции. Например, строка

$$01\ 10\ 11\ 11\ 1$$

эквивалентна правилу

$$(\text{Соперник} = \text{Выше}) \wedge (\text{Играет} = \text{В гостях}) \Rightarrow (\text{Победа} = \text{Да}).$$

У этой системы кодирования осталось только одно тонкое место: неясно, что делать, если в результате кроссовера или мутации в гипотезе получится набор из символов 00. Такая гипотеза не может быть применена ни разу: её условия соответствуют пустому множеству примеров. Здесь есть два пути. Первый — просто запретить появление таких гипотез: повторять кроссовер или мутацию до тех пор, пока результат не будет разумным. Второй — оставлять такие гипотезы в популяции, но присваивать им нулевое значение функции Fitness; тогда эти гипотезы наверняка будут «выполоты» на следующем шаге отбора. Оба подхода имеют право на существование; позволим себе рекомендовать второй подход как более идейный: отбраковка гипотез должна производиться посредством естественного отбора, а не искусственных ограничений.

В § 4.3 мы обещали привести пример более специфичной мутации, зависящей от представления данных. В случае задачи классификации весьма успешной мутацией оказывается удаление из посылки правила случайно выбранного атрибута (замена строки, соответствующей этому атрибуту, на строку из единиц). Такая мутация соответствует обобщению полученных гипотез и зачастую приводит к хорошим результатам.

Впрочем, пока что это всё ещё не совсем то, что нужно для решения задачи. Ведь мы до сих пор описывали отдельные правила, т.е. отдельные ветки дерева принятия решений. Но одна гипотеза — это не одна ветка, а несколько (гипотеза соответствует целому дереву).

Как закодировать несколько правил? Здесь, к счастью, ответ прост: при фиксированном наборе атрибутов все правила будут иметь постоянную длину, равную суммарному количеству возможных значений атрибутов плюс один бит на значение целевой функции (или больше, если целевая функция может принимать больше двух значений). Поэтому можно просто записывать правила подряд — никакой многозначности это не внесёт.

Гораздо более серьёзные трудности возникают от того, что в разных деревьях разное число веток и, следовательно, в разных гипотезах будет разное число правил.

Как же делать кроссовер со строками переменной длины? Нам нужно не только суметь скрестить две строки длиной $l_1 n$ и $l_2 n$ каждая, где n — длина правила, а l_i — число правил в каждой гипотезе, но и получить в результате строку длиной ln — будет неприятно получить гипотезу, в которой дробное число правил. Причём желательно, чтобы это l выбиралось более или менее случайно из промежутка от 1 до $l_1 + l_2$, а не всегда равнялось, скажем, $\max\{l_1, l_2\}$.

Одно из возможных решений таково: будем использовать двухточечный кроссовер так, чтобы сохранять постоянное расстояние до краёв правил. Тогда при замене участка первой гипотезы на участок второй, хотя эти участки могут быть разной длины, их длина по модулю n будет постоянной. Этот принцип проще всего объяснить на примере.

Пример 4.1. Кроссовер на строках переменной длины

Предположим, что в генетическом алгоритме, в котором нужно выполнить кроссовер на строках переменной длины, участвуют правила длины 5, и на очередном этапе алгоритм случайно выбрал две точки из первой гипотезы:

0[0101 110]10.

Тогда во второй гипотезе нужно выбирать такие точки, чтобы расстояние от левой точки до левого края правила и от правой точки до правого края правила были теми же. Например, в правиле длины 15 могут быть варианты:

1[10]11 01010 01110	1[1011 010]10 01110
1[1011 01010 011]10	11011 0[10]10 01110
11011 0[1010 011]10	11011 01010 0[11]10

Теперь кроссовер будет порождать корректные гипотезы:

Исходные строки	Результат
0[0101 110]10	01010
1[10]11 01010 01110	10101 11011 01010 01110

Этот вид кроссовера проиллюстрирован на рис. 4.5.

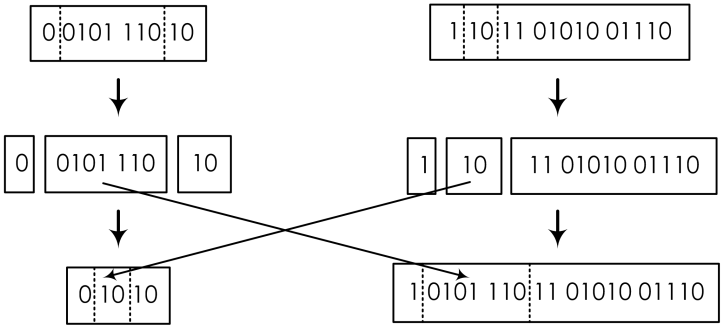


Рис. 4.5. Кроссовер на строках переменной длины

§ 4.5. Отбор

Мы уже рассмотрели все элементы типичного генетического алгоритма на примере задачи классификации, кроме одного — собственно естественного отбора. Более того, мы даже не определили функцию приспособленности Fitness. В условиях отбора гипотез при имеющемся фиксированном наборе тестовых примеров мы будем определять эту функцию следующим образом:

$$\mathrm{Fitness}(h) = \left(\frac{\mathrm{Correct}(h)}{\mathrm{TotalExamples}}\right)^2,$$

где $\mathrm{Correct}(h)$ — число примеров, верно расклассифицированных гипотезой h, TotalExamples — общее число примеров. Отметим, что такая функция приспособленности прекрасно справляется с проблемой строк, не соответствующих никаким корректным гипотезам — они не смогут классифицировать ни одного примера, и их приспособленность будет равна нулю.

Как же теперь выбрать наиболее приспособленные особи? Нам нужно отобрать sN особей из популяции численности N. Не забудем то, о чём мы уже упоминали в § 4.2: главная беда генетических алгоритмов — недостаточная гибкость, которая приводит к тому, что популяция становится слишком однородной и не может выбраться из локального максимума. Поэтому просто отбирать верхние sN особей по функции Fitness — не самая хорошая идея.

В реальных приложениях для отбора обычно используют один из двух наиболее популярных методов. Первый из них —

Genetic(N,p,s,m,Fitness,$\text{Fitness}_{\max}$)

1. Создать N случайных гипотез $\mathcal{H} = \{h_1, \ldots, h_n\}$.
2. Для каждой $h \in \mathcal{H}$ вычислить Fitness(h).
3. Пока $\max_{h \in \mathcal{H}} \text{Fitness}(h) < \text{Fitness}_{\max}$:
   а) $\mathcal{H}' = \emptyset$.
   б) Случайно выбрать sN гипотез из $\mathcal{H}$ и добавить их в $\mathcal{H}'$. При этом вероятность выбрать гипотезу $h_i \in \mathcal{H}$ равна
   $$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^{N} \text{Fitness}(h_j)}.$$
   в) Случайно выбрать $\frac{(1-s)p}{2}$ пар гипотез из $\mathcal{H}$ с теми же вероятностями. Для каждой пары (h_i, h_j) запустить операцию кроссовера и добавить её результат в $\mathcal{H}'$.
   г) Равномерно выбрать mN случайных гипотез из $\mathcal{H}'$ и в каждой из них инвертировать случайный бит.
   д) $\mathcal{H} := \mathcal{H}'$.
   е) Для каждой $h \in \mathcal{H}$ вычислить Fitness(h).
4. Выдать $\text{argmax}_{h \in \mathcal{H}} \text{Fitness}(h)$.

Рис. 4.6. Пример генетического алгоритма

метод рулетки (roulette wheel selection). В этом методе у каждой гипотезы с ненулевой функцией приспособленности есть шанс быть выбранной, и вероятность её выживания пропорциональна её функции приспособленности: у каждой гипотезы h_i вероятность быть выбранной

$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^{N} \text{Fitness}(h_j)}.$$

При этом одна и та же особь может быть выбрана несколько раз; алгоритм просто проводит выбор с заданной вероятностью в течение нужного числа итераций (sN). Этот метод — один из классических, но на практике используется не так уж часто. Во-первых, такая выборка достаточно дорога вычислительно,

а во-вторых, опять же, этому методу зачастую не хватает разнообразия в получающихся гипотезах. Вторую проблему можно отчасти решить, перейдя от метода рулетки к *ранговому методу*. В ранговом методе гипотезы (особи) сначала сортируются по приспособленности, а затем используется такая же выборка, что и в обычном методе рулетки, но вероятность быть выбранной у гипотезы прямо пропорциональна не абсолютному значению её приспособленности, а её рангу. Даже если гипотеза далеко отрывается от своих конкурентов по значению Fitness, ранговый метод всего лишь поставит её на первое место, девальвируя разницу в абсолютных значениях.

Другой часто используемый подход к отбору — *турнирный метод* (tournament selection). Сначала равномерно выберем две гипотезы. Затем с некоторой фиксированной вероятностью p (обычно, конечно, $p > \frac{1}{2}$) выживает более приспособленная, с вероятностью $(1 - p)$ — менее приспособленная гипотеза.

Давайте попробуем объединить всё то, что мы до сих пор изучили, в единую схему алгоритма. На рис. 4.6 изображён пример генетического алгоритма. В нём мы избрали метод рулетки в качестве метода естественного отбора, а инвертирование случайного бита — в качестве единственного типа мутаций.

§ 4.6. Дарвин, Ламарк и Болдуин

Как известно, Дарвин был не единственным учёным, предложившим схему эволюции. Жан Батист Ламарк[1] в своей книге «Философия зоологии», опубликованной за полстолетия до Дарвина, уже высказывал основные идеи эволюционного развития видов. При жизни Ламарка его мысли не получили поддержки

[1] Жан Батист Пьер Антуан де Моне, шевалье де Ламарк (Jean-Baptiste Pierre Antoine de Monet, Chevalier de Lamarck, 1744–1829) — французский биолог, создатель первой теории биологической эволюции. Ламарки издавна служили в армии, и Жан Батист не стал исключением: он проявил недюжинную храбрость во время Семилетней войны. Однако когда один из сослуживцев, играя, приподнял его за голову (похоже, армия везде одинакова...), у него случилось воспаление в лимфатических узлах шеи, и на этом военная карьера Ламарка закончилась. Главный его труд — «Философия зоологии» (1809); там он и изложил свои эволюционные идеи. Кстати, в другой работе, «Гидрогеологии» (1802), Ламарк придумал впоследствии прижившийся термин «биология».

среди научного сообщества. Вспомнили о нём только после выхода в свет книги Дарвина, когда реального научного значения его идеи уже не имели — все известные факты указывали на то, что прав был Дарвин, а не Ламарк.

Основной идеей Ламарка было то, что организмы изменяются под воздействием окружающей среды и условий их жизнедеятельности, причём, в отличие от дарвиновской теории, особи могут изменяться в течение своей жизни, а не только на генетическом уровне. Грубо говоря, киплинговский слонёнок даже без помощи крокодила действительно мог бы за свою жизнь чуть-чуть вытянуть хобот, это бы передалось его детям, те вытянули бы хоботы ещё чуть-чуть, и рано или поздно они бы эволюционировали в современных слонов. У Дарвина же длина хобота слонёнка задана генетически и в течение жизни не меняется, зато слоны с длинными хоботами чаще выживают, привлекают, разумеется, повышенное внимание слоних и в результате начинают доминировать среди потомства.

Хотя теория Ламарка не выдерживает биологической критики[1], кто сказал, что она совсем бесполезна для искусственного интеллекта? Фактически, все программы машинного обучения, которые мы рассматривали в предыдущих главах — это ламаркианские виды, которые обучаются по ходу жизни, а не только посредством естественного отбора. И в рамках общей парадигмы генетических алгоритмов мы тоже можем совместить естественный отбор (его идею Ламарк, кстати, тоже выдвинул — выживают те слонята, которые сумели вытянуть хобот максимально далеко) с обучением «on-the-fly».

Для этого нужно будет в качестве гипотез (особей популяции) использовать какой-нибудь из аппаратов машинного обучения, и обучать каждое их поколение на тестовых примерах. Обычно в качестве такого аппарата применяют нейронные сети: помимо прочего, их ещё и легко обучить «чуть-чуть», так,

[1] Стоит, правда, отметить, что эволюция по Ламарку бывала и общепризнанной, а критике подвергались совершенно другие люди. Трофим Денисович Лысенко прямо так и говорил в речи, которая уже цитировалась в эпиграфе: «Мы, представители советского мичуринского направления, утверждаем, что наследование свойств, приобретаемых растениями и животными в процессе их развития, возможно и необходимо».

чтобы «генетический код» не потерялся от обучения; а, например, деревья принятия решений после работы алгоритма ID3 все были бы на одно лицо.

Другая мысль, также пришедшая в искусственный интеллект из генетики — так называемый *эффект Болдуина*.[1] Он связан со способностью организма обучаться в течение жизни. Как повлияет такая способность на эволюцию? Она сможет «сгладить» зависимость приспособленности от генотипа: чем лучше обучается особь, тем меньше она зависит от генотипа. Например, человек в процессе своей жизни обучается стольким разным вещам, что в итоге от генотипа конкретного человека зависит в его жизни очень мало, и совершенно не понятно, какие мутации могут оказаться для человека благоприятными и быть закреплёнными естественным отбором — по всей видимости, естественный отбор в его дарвиновском смысле для человека сильно замедлился.

Болдуиновский эффект не является, как может показаться, развитием ламаркианских идей. Приобретённые в результате обучения навыки не передаются по наследству. Слонята учатся вытягивать хобот, чтобы достать еду, но от этого их хоботы не становятся длиннее. Однако обучение делает их более приспособленными к окружающей среде и даёт популяции слонов время для того, чтобы развить у себя длинный хобот сугубо дарвиновскими методами [123].

Применительно к искусственному интеллекту эффект Болдуина выражается в том, что в естественном отборе участвуют обучающиеся особи, причём их способность к обучению также является предметом естественного отбора.

Пример 4.2. Генетический алгоритм с эффектом Болдуина

Опишем конкретную (впрочем, опишем мы её в достаточно общем виде, без элементов программной реализации) схему построения генетического алгоритма (основных его операций) на нейронных сетях, в основном следуя [151].

Каждый элемент популяции — нейронная сеть глубины 2 с N входами, M нейронами на скрытом уровне и N нейронами на выходе.

[1] На самом деле его независимо и практически одновременно — в 1896 году — открыли Болдуин (Baldwin), Морган (Morgan) и Осборн (Osborn).

Таким образом, у такой особи образуются $(N+1)M+(M+1)N$ генов, соответствующих весам нейронной сети («плюс единицы» образуются потому, что у каждого нейрона есть ещё вес w_0, добавляющий константу к взвешенной сумме входов — см. Главу 3). Более того, у каждой особи есть ещё столько же бинарных генов (*генов пластичности*), которые определяют, может ли соответствующий вес изменяться в процессе обучения или нет.

Каждая итерация алгоритма состоит из двух частей. В течение первой части имеющиеся особи обучаются на тестовых примерах; при этом обучение учитывает также гены пластичности: для веса w обучение происходит по формуле

$$\Delta w = -\eta p \sum_{v \in V} \frac{\partial E_v(w)}{\partial w},$$

где η — скорость обучения, V — тестовые примеры, E_v — функция ошибки, а p — это как раз соответствующий весу w ген пластичности. Таким образом, если этот ген равен 1, то вес может изменяться в процессе обучения, а если он равен 0, то вес на протяжении обучения останется постоянным.

На втором этапе обученные нейронные сети участвуют в генетических операциях. Эти операции производятся с функцией приспособленности

$$\mathrm{Fitness} = 1.0 - \frac{1}{N2^N} \sum_{v \in V} \sum_{i=0}^{N-1} (\mathrm{Out}_{v,i} - \mathrm{Target}_{v,i})^2,$$

где $\mathrm{Target}_{v,i}$ — значение целевой функции на i-м выходе сети в v-м тестовом примере, а $\mathrm{Out}_{v,i}$ — реальный выход i-го выходного нейрона сети.

Отбор и размножение в [151] велись очень простым способом: наихудшая особь популяции заменялась копией наилучшей особи. Можно было бы заменять несколько худших на несколько лучших или всё-таки использовать более традиционные методы; с другой стороны, большая вычислительная стоимость каждой итерации (нужно обучить каждую особь) приводит к тому, что в популяции должно быть не так уж много особей. Возможно, с учётом этого обстоятельства выбор авторов [151] действительно оптимален.

Главная изменчивость в этом примере достигается за счёт мутаций. Мутации бывают двух видов. Мутации весов нейронной сети изменяют случайно выбранный вес на случайное значение, равномерно выбранное из заданного интервала $[-d,d]$. Мутации генов пластичности — обычное инвертирование случайного бита.

На практике оказывается, что ламаркианский подход даёт неплохие сети достаточно быстро, в то время как болдуиновский подход позволяет получить сети более высокого качества, но медленнее. Если у задачи нет большого количества локальных минимумов, и результат можно получить относительно быстро, то лучше использовать ламаркианскую стратегию; в общем же случае с задачей лучше справляется подход, основанный на эффекте Болдуина [28].

§ 4.7. Генетические алгоритмы на деревьях

До сих пор мы рассматривали генетические алгоритмы, в которых элементами популяции являлись битовые строки. Однако это отнюдь не единственное возможное кодирование, для которого можно ввести разумные операции кроссовера и мутаций. Так, например, сейчас мы рассмотрим, как определять генетические операции на деревьях, и убедимся, что это вполне возможно и только немногим сложнее, чем генетические операции на строках битов. В следующих параграфах мы разовьём идею генетического программирования — основного примера генетических алгоритмах на деревьях, а в этом параграфе ограничимся более общей ситуацией.

Как мы уже знаем, для того чтобы определить генетический алгоритм, нужно уметь выполнять инициализацию, кроссовер и мутации, а также вычислять функцию приспособленности особей. С функцией приспособленности мы сейчас иметь дело не будем — она всё равно зависит в первую очередь от конкретной задачи и предметной области. Зато подробно рассмотрим все остальные операции, считая, что особями в популяции выступают деревья.

Инициализация обычно сводится к тому, чтобы случайным образом породить дерево. Есть разные способы порождать случайные деревья. Обычно главная сложность заключается не в том, чтобы породить дерево хоть как-нибудь (с этим проблем нет), а в том, чтобы вероятностное распределение на порождённых деревьях было равномерным, т.е., в частности, чтобы у любого дерева был шанс стать результатом алгоритма порождения. Эти задачи подробно рассмотрены в книге [4], а работа [3]

содержит описание (достаточно несложного) линейного алгоритма порождения деревьев, который позволяет приблизиться к равномерному распределению.

Как правило, эффективные алгоритмы порождения деревьев (не забудем, что на практике могут потребоваться разные деревья: бинарные, тернарные, произвольные, леса из деревьев того или иного типа и т.д.) работают следующим образом: создаётся некий язык и устанавливается взаимно однозначное соответствие между словами этого языка и деревьями требуемого типа. Таким образом, проблема порождения дерева сводится к проблеме выбора случайного слова из этого языка — а для этого достаточно набрасывать случайным образом буквы соответствующего алфавита. Работа [3] именно по такому принципу и построена. Однако для наших целей можно не углубляться в эти тонкости; достаточно даже самого тривиального алгоритма. Единственная мелочь, которую стоит учитывать — тот факт, что деревья у нас будут с метками, и метка вершины будет определять число потомков у этой вершины.

Поэтому генерировать деревья будем следующим образом: предположим, что имеется некоторый набор меток

$$L = L_0 \cup L_1 \cup \ldots \cup L_s,$$

где в каждом из множеств L_i находятся метки, соответствующие числу потомков у вершин, в которых они должны стоять. Затем, начав с одной вершины, выберем ей случайным образом метку $l \in L$ и добавим соответствующее метке количество ещё не помеченных потомков. А затем будем итерировать по непомеченным потомкам до тех пор, пока таковые не закончатся.

У этого алгоритма только один недостаток: «таковые» не закончатся никогда. Если только нульарных (*терминальных*) вершин не значительно больше, чем остальных, алгоритм будет постоянно производить новые и новые уровни дерева.

Справиться с этим просто: достаточно ограничить глубину дерева. Иными словами, на некоторой максимальной глубине d_{max} мы будем выбирать метки вершин не из всего L, а из L_0. Тогда дерево будет гарантированно ограниченным. В приложении к генетическому программированию ограничение глубины, как правило, является вполне разумным: может существовать

априорное ограничение глубины, или решение с той или иной глубиной может быть уже известно, а хочется найти решение с меньшей глубиной (то есть де-факто более быстрый алгоритм).

Кроссовер на деревьях принципиально очень похож на кроссовер на строках. Как и в том случае, нужно выбрать случайную позицию (в данном случае — случайный узел). Затем нужно разбить каждое из двух участвующих деревьев на две части: часть с корнем в выбранном узле и дерево, полученное из исходного удалением этого узла, а затем обменяться поддеревьями.

Мутации, как обычно, — самая сложная, но вместе с тем и самая плодотворная для дальнейших поисков часть конструкции. Успех генетических алгоритмов, как это часто бывает, зависит от мутаций. В случае, когда речь идёт об алгоритмах на деревьях, простейшей мутацией, соответствующей изменению одного бита в битовой строке, было бы изменение одной метки в одном узле дерева. Беда в том, что с изменением метки может измениться число потомков, которое должно быть у этого узла, или их характер.

Основных вариантов два. Первый — изменять ту или иную метку только на метку с тем же числом потомков. Но если, например, в языке есть только одна метка с тремя потомками, то она уже никогда не мутирует. Другой, чаще употребляемый, подход состоит в том, чтобы при мутации наращивать новое случайное дерево из мутировавшего узла. При этом мутации становятся достаточно радикальными — если, например, мутирует корень дерева (вот уж в самом буквальном смысле *радикальная* мутация!), то дерево просто становится совершенно другим.

§ 4.8. Генетическое программирование

Генетическое программирование — реализация давней мечты всех программистов: заставить компьютер самого себя программировать. О том, как решать задачи поиска оптимальных программ и алгоритмов методами, основанными на дарвиновском естественном отборе, учёные задумались достаточно давно; во всяком случае, уже Алан Тьюринг в своём программном эссе «Умные машины» («Intelligent Machines») писал (мы цитируем Тьюринга по [98]):

> Есть и генетический, или эволюционный, поиск, при котором ищется комбинация генов, а критерием является уровень выживаемости. Удивительный успех такого поиска до некоторой степени подтверждает, что интеллектуальная деятельность состоит в основном из различных видов поиска».

А в статье, вышедшей в 1950 году, Тьюринг уже напрямую закладывает основы генетического программирования:

> Мы не можем ожидать, что найдём хорошую машину с первой попытки. Нужно поэкспериментировать с обучением одной такой машины и посмотреть, как хорошо она обучается. Затем можно попробовать другую и выяснить, лучше она или хуже. Есть очевидная связь между этим процессом и эволюцией, ведь можно отождествить структуру машины с генетическим материалом, изменения в ней с мутациями, а естественный отбор — с суждением экспериментатора.

После Тьюринга эту мысль надолго оставили. В 1975 году Джон Холланд[1] в своей знаменитой книге «Адаптация в естественных и искусственных системах» [66] вновь вернулся к этой мысли, но она требовала слишком больших вычислительных ресурсов, в то время недоступных. Уже в девяностые годы прошлого века одним из основных исследователей, работающих в этой области, стал Джон Коза (John Koza), для которого генетическое программирование стало делом всей научной жизни [94–98].

Ажиотаж, возникший вокруг этого направления, основан отнюдь не на пустом месте. У генетического программирования уже есть успехи, которые прекрасно укладываются в «статут» искусственного интеллекта: при помощи генетического программирования компьютер уже смог придумать несколько новых, более эффективных алгоритмов решения известных задач. По итогам работ Джона Козы и его соавторов были даже поданы

[1] Джон Холланд (John Holland, p. 1929) — американский математик. Он считается «отцом» генетических алгоритмов.

две патентные заявки на изобретения, совершённые компьютером![1]

Итак, генетическое программирование. Его суть в том, что элементами популяции становятся не битовые строки, а программы, обычно представленные в виде деревьев. В узлах деревьев находятся функции, а в их потомках — аргументы этих функций. Здесь нам очень пригодится предыдущий параграф, ведь мы находимся точно в той ситуации, которая была там описана: арность функций строго задаёт число потомков у вершин дерева. Отметим, что представление данных в генетическом программировании очень похоже на один из функциональных языков программирования. Недаром Джон Коза избрал для иллюстраций в своей первой книге язык LISP даже для собственно реализации генетического программирования, не только для представления программ. Мы подробнее поговорим об этом в § 4.10, а пока ограничимся более простым примером, на котором продемонстрируем основные идеи генетического программирования.

Предположим, что мы хотим, чтобы полученная программа вычисляла некоторую функцию. Пусть для простоты это будет многочлен от одной переменной $x^2 - 1$. Опишем конструкцию генетического алгоритма, который должен будет найти эту программу.

Прежде всего нужно описать множество функций, из которых будет состоять программа. Пусть мы заранее знаем, что наша цель — вычислить некую арифметическую функцию; тогда мы можем изначально ограничиться арифметическими функциями $+$, $-$, $*$, $/$, %. В качестве *терминальных символов*, они же константы и переменные, они же нульарные функции, будут выступать переменная x и численные константы от -5 до 5. На рис. 4.7 изображены возможные элементы начальной популяции программ с такими функциями (ограничимся деревьями глубиной не более трёх), а рис. 4.8 показывает, как происходит последовательное «выращивание» случайного элемента популяции. Как только достигнута предельная глубина дерева, случайный выбор на последнем уровне происходит только из терминальных символов.

[1] Подробности — на `http://www.genetic-programming.org`.

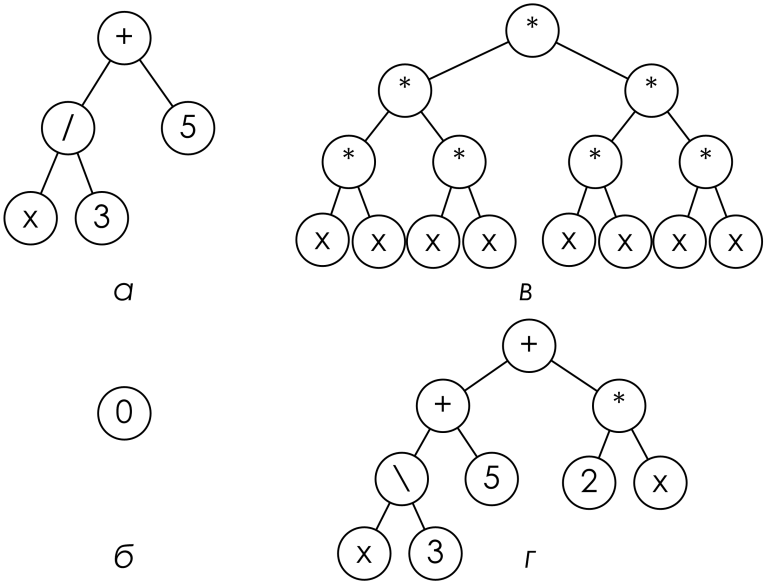


Рис. 4.7. Возможные элементы популяции:
а) $x/3+5$; *б*) 0; *в*) x^8; *г*) $x/3+5+2x$

Кроссовер на деревьях мы будем проводить так, как указывали в предыдущем параграфе: в участвующих в кроссовере деревьях мы будем выбирать два узла, а затем менять местами поддеревья, «растущие» из этих узлов. В качестве мутаций примем схему, которая выбирает случайный узел, а затем «выращивает» из него новое случайное поддерево.

Остаётся только один вопрос — как же оценивать особей популяции? Что принять за функцию приспособленности?

Поскольку мы хотим научиться вычислять функцию, достаточно логичным выглядит принять за меру качества нашего приближения этой функции разницу между ней и целевой функцией. Роль такой разницы может сыграть, например, интеграл модуля разности между нашей функцией и целевой; а чтобы интеграл этот не был всегда бесконечным, нам придётся либо ограничиться каким-нибудь отрезком, либо ввести достаточно быстро убывающие веса (ядра интегралов), чтобы все интегралы сходились. Мы для простоты возьмём отрезок $[-5;5]$ и будем минимизировать интеграл по нему.

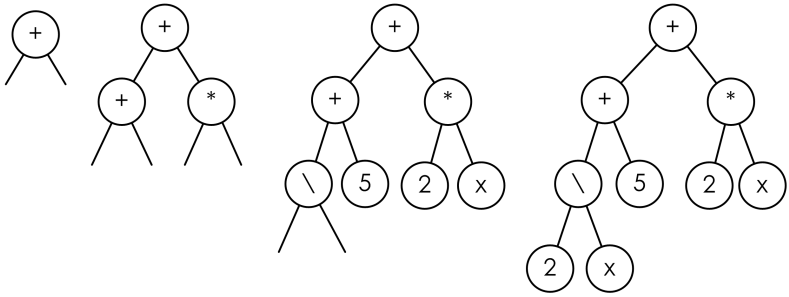


Рис. 4.8. Случайное порождение дерева программы

Для кроссовера на каждом шаге мы будем случайным образом выбирать из четырёх особей в популяции две пары. В качестве отбора будем брать лучшие четыре «программы» из получающихся; мутации будем применять случайным образом к одному из четырёх потомков.

Теперь можно применить схему генетического алгоритма с учётом всего вышесказанного и всего описанного в § 4.7. Наша цель — программа, вычисляющая функцию $x^2 - 1$. Начнём с функций, уже представленных на рис. 4.7:

$$x/3 + 5,\ 0,\ x^8,\ 7x/3 + 5.$$

Первый цикл алгоритма показан на рис. 4.9. На первом шаге будем скрещивать первую особь с четвёртой, а вторую — с третьей. Выберем случайные ветви, которые кроссовер будет заменять друг на друга (они показаны сверху чёрным, а остаток дерева — серым).

В результате скрещивания получатся особи, изображённые на рис. 4.9 во второй строке. Обратите внимание, что третья особь в результате скрещивания с нулём случайно оказалась тождественно равной нулю, хотя она всё ещё представляет собой достаточно большое дерево. Здесь можно было бы «пропагировать», передать ноль снизу вверх (мы же знаем, что $0 * x = 0$), но можно и оставить дерево как есть — вдруг оно не умрёт на этапе отбора и потом ещё пригодится. Мы для простоты (чтобы не определять формально правила «пропагации нулей» и прочих упрощений) будем оставлять деревья в покое.

Затем применим мутацию к одному из потомков; мутация состоит в выращивании случайного дерева и заменой на него случайной ветви дерева. В результате у нас (с учётом начальных) получились восемь особей, реализующих следующие восемь функций:

$$\begin{array}{llll} x/3+5, & 0, & x^8, & 5+7x/3, \\ 2x+5, & x^4, & 0, & 5-2x/3. \end{array}$$

На этапе отбора мы для каждой из этих функций $f(x)$ подсчитаем интеграл модуля разности между $f(x)$ и целевой функцией:

$$\mathrm{Fitness}(f) = \int_{-5}^{5} \left| f - x^2 + 1 \right| dx.$$

Мы получим следующие (приближённые численные) значения функции Fitness:

$$\begin{aligned} \mathrm{Fitness}(x/3+5) &= 62.7976, \\ \mathrm{Fitness}(0) &= 76, \\ \mathrm{Fitness}(x^8) &= 43395.44, \\ \mathrm{Fitness}(5+7x/3) &= 76.5912, \\ \mathrm{Fitness}(2x+5) &= 72.7207, \\ \mathrm{Fitness}(x^4) &= 1176.67, \\ \mathrm{Fitness}(5-2x/3) &= 63.6189. \end{aligned}$$

В результате отбора выживут особи, изображённые в нижней строке рис. 4.9 (мы случайным образом выбрали один из двух тождественных нулей). Пока их функции приспособленности далеки от идеала, и на протяжении следующих поколений генетический алгоритм должен будет уменьшить её до нуля. А мы на этом оставим теоретические примеры и перейдём к практическим.

§ 4.9. Генетическое программирование на практике

В этом параграфе мы рассмотрим программу, которая реализует генетическое программирование для решения конкретной задачи. Задача будет во многом игрушечная, немногим более сложная, чем задача из предыдущего примера: мы будем вычислять норму матрицы 2×2. Точнее говоря, квадрат нормы матрицы — сумму квадратов её элементов.

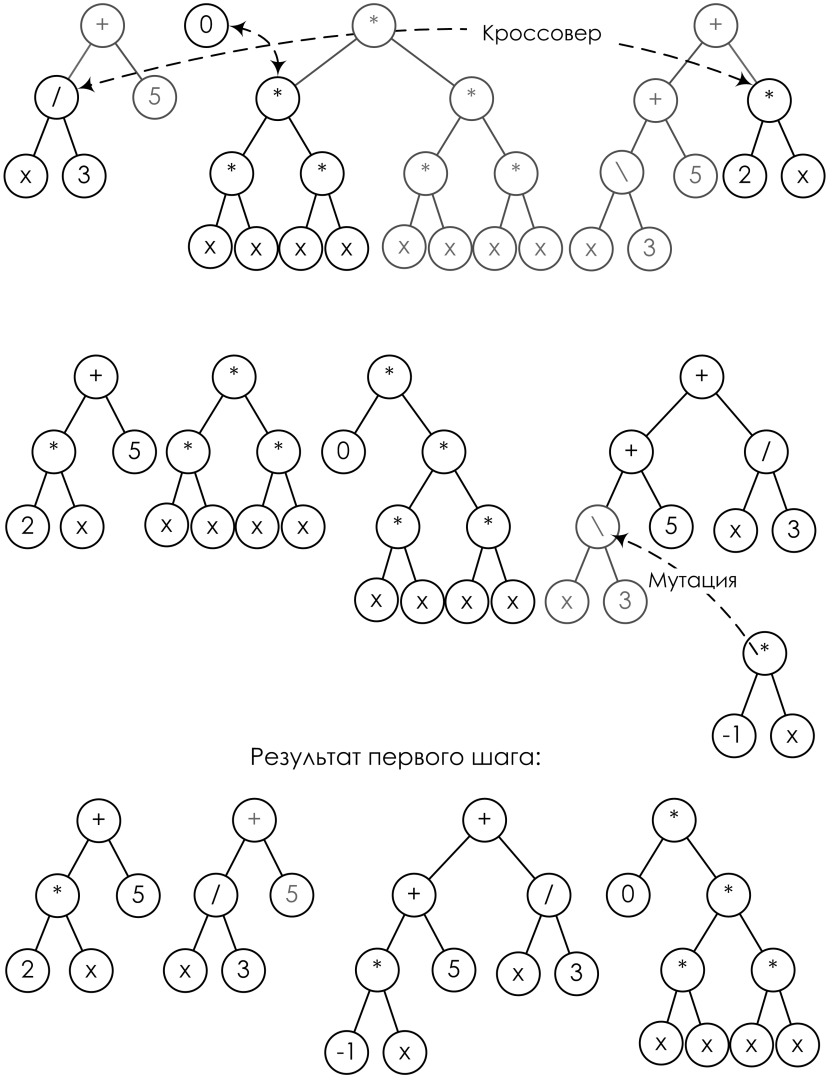


Рис. 4.9. Первый шаг генетического программирования

Мы приведём полную, работающую программу и разъясним, как работают её отдельные части. Конечно, программы для решения возникающих на практике проблем могут быть (и скорее всего будут) сложнее, но общая схема достаточно чётко видна

даже на таком небольшом примере, как вычисление нормы матриц 2×2. Программу будем реализовывать на языке Java; директивы import в листингах мы будем для краткости опускать.

Во-первых, нам понадобится генератор тестовых примеров: разумеется, мы не горим желанием задавать вручную многочисленные матрицы и их нормы. В этом классе мы просто генерируем случайную матрицу, подсчитываем её норму, а затем добавляем вычисленную норму как целевую функцию для этого тестового примера.

Листинг 4.1. Генетическое программирование: SampleMaker

```
public class SampleMaker {
    public static void MakeSamples(double[][] matrices,
                                   double[] results) {
        assert (matrices != null);
        assert (results != null);
        assert (matrices.length == results.length);

        for (int i = 0; i < matrices.length; i++) {
            if (matrices[i] == null) {
                matrices[i] = new double[4];
            }
            for (int j = 0; j < 4; j++) {
                matrices[i][j] = Math.random();
            }
            matrices[i][3] += 1.4 * Math.random();
            if (matrices[i][3] > 1) {
                matrices[i][3] = 1;
            }
            results[i] = 0;
            results[i] = matrices[i][0] * matrices[i][0] +
                         matrices[i][1] * matrices[i][1] +
                         matrices[i][2] * matrices[i][2] +
                         matrices[i][3] * matrices[i][3];
        }
    }
}
```

Во-вторых, мы должны реализовать класс, подсчитывающий функцию приспособленности. При инициализации такому классу передаётся количество случайных тестовых примеров,

которые он должен порождать; примеры он генерирует при вызове процедуры `updateSamples()`, а затем подсчитывает приспособленность по формуле

$$\text{Fitness} = \frac{1}{1 + \sum_i \delta_i},$$

где δ_i — модуль разности настоящей нормы тестовой матрицы и значения функции, которую вычисляет данная особь (напомним, что особь — это дерево из арифметических операций). Если же функция настолько плоха, что вызывает деление на ноль или какую-нибудь другую ошибку, Fitness принимается равным нулю.

Стоит отметить ещё две особенности. Во-первых, если все элементы популяции окажутся с нулевой приспособленностью, алгоритм захочет разделить на ноль (их суммарную вероятность). Чтобы избежать этого, в таком случае мы просто присваиваем неприспособленным бедолагам хоть какие-нибудь вероятности.

Во-вторых, мы хотели бы ограничить общую глубину получающегося дерева; поэтому, если дерево в результате размножения или мутации оказывается слишком глубоким, мы просто заменяем его на свежевыращенное случайное дерево (возможно, это не оптимальный способ — можно было бы модифицировать кроссовер и мутации так, чтобы максимальная глубина не нарушалась, — но мы сейчас не будем этого делать).

Листинг 4.2. Генетическое программирование: FitnessCalc

```
public class FitnessCalc {
    private static final int SIZE = 4;
    private final double[][] smpMatrices;
    private final double[] smpResults;
    FitnessCalculator(int n) {
        smpMatrices = new double[n][SIZE];
        smpResults = new double[n];
    }
    void updateSamples() {
        SampleMaker.MakeSamples(smpMatrices, smpResults);
    }
    double fitness(TreeNode tree) {
        assert (smpMatrices.length == smpResults.length);
        double fitness = 0;
        for (int i = 0; i < smpMatrices.length; i++) {
```

```
            double delta = Math.abs(
                tree.eval(smpMatrices[i])- smpResults[i]);
            fitness += delta;
        }
        if (Double.isNaN(fitness)) fitness = 0;
        else fitness = 1 / (1 + fitness);
        tree.setFitness(fitness);
        return fitness;
    }
}
```

Основным классом программы является класс, при помощи которого осуществляется собственно естественный отбор. Трудно пересказать его своими словами так, чтобы всё было понятно в деталях, поэтому мы подробно прокомментировали сам код.

Листинг 4.3. Генетическое программирование: NaturalSelection

```
public class NaturalSelection {
    private TreeNode[] popul;
    private final int maxDepth;
    private final FitnessCalc fitnessCalc;
    private final double survRate;
    private final double mutRate;
    private Writer w;

   /**
     * Конструктор класса, реализующего естественный отбор
     * populSize -- размер поколения
     * mDepth -- ограничение на глубину дерева особи
     * nSurvivors -- среднее число выживающих особей
     *  (в одном раунде естественного отбора)
     * nMutants -- среднее число мутирующих особей
     */
    public NaturalSelection(int populSize, int mDepth,
      int nSurvivors, int nMutants)
      throws FileNotFoundException {
        // Экземпляр класса, подсчитывающего fitness
        // Количество тестов, конечно, тоже можно было бы
        // передавать как параметр
        fitnessCalc = new FitnessCalc(50);
        // Доли выживающих и мутирущих особей
        survRate = nSurvivors / (double) populSize;
        mutRate = nSurvivors / (double) populSize;
        // В этом массиве хранится текущее поколение
        popul = new TreeNode[populSize];
```

```
        maxDepth = mDepth;
        // Инициализируем популяцию случайными особями
        for (int i = 0; i < populSize; i++)
            popul[i] =
              RandomTreeFactory.createTree(maxDepth);
    }

    // Выполняем один раунд естественного отбора
    private boolean doRound() throws IOException {
        // Обновить тестовые примеры
        fitnessCalc.updateSamples();

        // Подсчитать Fitness для каждой особи
        for (int i = 0; i < popul.length; i++) {
            fitnessCalc.fitness(popul[i]);
        }

        // Отсортировать популяцию по приспособленности
        Arrays.sort(popul);
        // Вспомогательный debug-вывод
        printpopul();

        // У идеальной особи Fitness == 1.0
        // Если нашли такую особь -- выдаём ответ
        if (popul[popul.length - 1].getFitness() > 0.99) {
            System.out.print("We reached the optimum: ");
            System.out.print(
              popul[popul.length - 1].toString());
            return true;
        }

        // Создаём массив интегральных вероятностей и
        // заполняем его так, чтобы разность
        // (probs[i + 1] - probs[i])
        // была пропорциональна Fitness(i) .
        // С этими вероятностями особи потом
        // будут выбираться для генетических операций.
        double[] probs = new double[popul.length + 1];
        probs[0] = 0;
        for (int i = 1; i < popul.length + 1; i++) {
            probs[i] = probs[i-1] +
                              popul[i-1].getFitness();
        }

        // Если все особи никуда не годятся,, нужно тем не
        // менее присвоить им какие-нибудь вероятности.
        if (probs[probs.length - 1] == 0.0) {
            for (int i = 0; i < probs.length; i++) {
```

```
        probs[i] = i;
    }
}

// Нормируем
for (int i = 0; i < probs.length; i++) {
    probs[i] /=
      probs[probs.length - 1];
}

// Создаём массив для записи следующего поколения
TreeNode[] newpopul = new TreeNode[popul.length];

// Принцип элитизма -- две лучшие особи выживают
newpopul[popul.length-1] = popul[popul.length-1];
newpopul[popul.length-2] = popul[popul.length-2];
int k = 0;
// Но эти две особи должны быть разными
for (k = popul.length - 1; k >= 0; k--) {
    if (popul[k] != popul[popul.length-1] &&
        popul[k].getFitness() !=
          popul[popul.length-1].getFitness()) {
        newpopul[popul.length - 2] = popul[k];
        break;
    }
}

// Заполняем "оставшиеся места" в новом поколении
// результатами генетических операций над особями
// старого поколения
for (int i = 0; i < popul.length - 2; i++) {
    // Далее будем случайным образом выбирать
    // генетическую операцию, сравнивая случайное
    // число с вероятностями выживания
    // и мутации особи.
    double rndVal = Math.random();
    if (rndVal < survRate) {
        // Операция воспроизведения.
        double randomValue = Math.random();
        int index = Arrays.binarySearch(
          probs, randomValue);
        if (index < 0) {
            index = -index - 2;
        } else if (index == popul.length) {
            index--;
        }
        // Копируем особь в новое поколение
        newpopul[i] = popul[index];
```

```
            newpopul[i].setOrigin("R");
        } else if (rndVal < 1 - mutRate) {
            // Кроссовер.
            int i1 = Arrays.binarySearch(
              probs, Math.random());
            int i2 = i1;
            while (i2 == i1) {
                i2 = Arrays.binarySearch(
                  probs, Math.random());
            }
            if (i1 < 0) {
                i1 = -i1 - 2;
            } else if (i1 == popul.length) {
                i1--;
            }
            if (i2 < 0) {
                i2 = -i2 - 2;
            } else if (i2 == popul.length) {
                i2--;
            }
            // Производим кроссовер.
            newpopul[i] =
              popul[i2].crossover(popul[i1]);
            newpopul[i].setOrigin("C");
        } else {
            // Мутация.
            int index = Arrays.binarySearch(
              probs, Math.random());
            if (index < 0) {
                index = -index - 2;
            } else if (index == popul.length) {
                index--;
            }

            // Мутация и запись в новое поколение
            newpopul[i] =
              popul[index].randomMutation(maxDepth);
            newpopul[i].setOrigin("M");
        }
    }

    // Новое поколение сменяет старое
    popul = newpopul;

    // Вместо слишком глубоких деревьев помещаем в
    // новое поколение случайные (мы предполагаем,
    // что искомая особь представляет собой не слишком
    // глубокое дерево).
```

```
        for (int i = 0; i < popul.length; i++) {
            if (popul[i].depth() > maxDepth * 2) {
                popul[i] = RandomTreeFactory.createTree(
                  maxDepth);
                newpopul[i].setOrigin("N");
            }
        }

        return false;
    }

    // Процедура тестового вывода
    private void printpopul() throws IOException {
        for (int i = 0; i < popul.length; i++) {
            w.write(popul[i].toString()+" "+
              popul[i].getOrigin()+" "+popul[i].depth()+" "+
              popul[i].getFitness()+"\n");
        }
        w.write("--\n");
    }

    // Процедура, обобщающая doRound() до отбора
    public boolean doSelection(int iterLimit)
      throws IOException {
        int i = 0;
        w = new FileWriter("output.txt", true);
        while (!doRound()) {
            i++;
            if (i%1000 == 0) {
                System.out.println(i+" generations.\n");
            }
            if (i > iterLimit) {
                w.close();
                return false;
            }
        }
        w.close();
        return true;
    }
}
```

Теперь, когда все процедуры готовы, осталось реализовать лишь не слишком содержательный запуск всех этих процедур. Константы из нижеприведённого кода (количество перезапусков при неудаче, максимальное количество поколений в поиске)

взяты с потолка; на практике их пришлось бы подбирать экспериментально.

Листинг 4.4. Генетическое программирование: Start

```
public class Start {
    public static void main(String[] args) {
        Writer w = null;
        try {
            w = new FileWriter("output.txt", false);
            w.write("*\n");
            // Провести естественный отбор
            performSelection();
        } catch (IOException e) {
            e.printStackTrace();
        }  finally {
            try {
                if (w != null) {
                    w.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static int performSelection()
      throws IOException {
        NaturalSelection selection;
        // Повторяем несколько раз естественный отбор
        // "с нуля", пока не найдем подходящую особь (если
        // совсем не везёт, делаем 40 повторов).
        for (int i = 0; i < 40; i++) {
            // Делаем естественный отбор "с нуля"
            // Лимит поколений - 3000.
            selection = new NaturalSelection(30, 3, 2, 10);
            if (selection.doSelection(3000)) {
                // Нашли подходящую особь.
                break;
            }
        }
        return 0;
    }

    public static void test() {
        TreeNode add = new DivisionTreeNode();
        NumberTreeNode n1 = new NumberTreeNode(15);
        NumberTreeNode n2 = new NumberTreeNode(5);
```

```
        add.addChild(0, n1);
        add.addChild(1, n2);

        add.toString();
        System.out.println();
        System.out.println(add.eval(null));

        TreeNode rnd = RandomTreeFactory.createTree(6);
        rnd.toString();
        System.out.println();

        int a[] = { 5, 2, 7 };
        Arrays.sort(a);
        for (int i = 0; i < 3; i++)
            System.out.print(a[i] + " ");

        System.out.println();

        TreeNode tree1 = RandomTreeFactory.createTree(2);
        tree1.toString();
        System.out.println();

        TreeNode tree2 = tree1.randomMutation(3);
        tree2.randomMutation(3);
        tree2.toString();
        System.out.println();
        tree2.getRandomBranch().toString();

        System.out.println();
        tree2.crossover(tree1).toString();
    }
}
```

В результате своей работы эта программа выдаёт функцию наподобие (результат может меняться, ведь многие вычисления были случайными)

$$((m[1] * m[1]) + ((sqr(m[3]) + sqr(m[0])) + sqr(m[2]))),$$

где m[i] обозначают элементы матрицы, а sqr — возведение в квадрат.

§ 4.10. Язык программирования LISP

Как мы уже видели, для того чтобы естественным и классическим образом применить генетическое программирование,

нужно, чтобы программа записывалась в виде дерева. Разумеется, при должной изобретательности можно любую программу на любом языке программирования представить в виде дерева; но для большинства императивных языков это будет не слишком естественным представлением — программы там представляют собой *последовательности команд*.

Зато это вполне естественно для функциональных языков, где программы представляют собой деревья вызовов функций (как мы и представляли особей популяции в наших предыдущих примерах). Поэтому неудивительно, что знаковым для генетического программирования (и вообще для искусственного интеллекта) стал язык программирования LISP. В этом параграфе мы бегло коснёмся основных положений языка LISP. Это изложение ни в коей мере не претендует на полноту, и мы всячески рекомендуем использовать такие более содержательные источники, как [100, 118, 147, 158].

LISP — один из старейших среди языков программирования, которые используются до сих пор (старше него разве что Fortran). Однако основные его идеи до сих пор отнюдь не общеизвестны среди программистов, даже несмотря на то, что знакомство с ними очень помогает и в программировании на «обычных» языках. Если вы уже программировали на LISP, то можете смело пропустить этот параграф.

LISP легко узнать по его синтаксису; основу LISP составляют списки и, как их разновидность, так называемые s–выражения (s–expressions): список из названия функции и её аргументов. То, что в большинстве других языков программирования записывается как `f(x,y,z)`, в LISP записывается как `(f x y z)`. Вызовы функции, разумеется, можно вкладывать друг в друга; собственно, любая функция на LISP и представляет собой одно сложно структурированное s–выражение. В LISP также присутствуют обычные арифметические операторы и удобные функции работы со списками. Вложенность и структура языка естественным образом намекает на рекурсию. Вот, например, функция вычисления факториала на LISP (обратите внимание на синтаксис: условный переход и арифметика выполняются теми же s–выражениями):

```
(defun factorial (n)
```

```
(if (<= n 1)
  1
  (* n (factorial (- n 1)))))
```

LISP включает в себя оператор lambda, который может определять функцию «на лету»; например, этот код:

```
((lambda (x)
     (* x x) )
  4)
```

выдаст в ответ 16, возведя 4 в квадрат. LISP также может передавать одним функциям другие функции в качестве параметра; это и обеспечивает его гибкость и универсальность. В качестве простого примера приведём функцию find, которая позволяет в качестве параметра :test: задавать функцию сравнения, а затем ищет в заданном списке, пользуясь этой функцией. Например, код

```
(defun NumbersAreClose (x y)
  (<= (abs (- x y)) 0.1))
(find pi '(3.0 3.1 3.2 3.3 3.4 3.5)
    :test #'NumbersAreClose)
```

выдаст в качестве результата 3,1 — первое вхождение в список числа, менее чем на 0,1 отклоняющегося от π. Здесь defun — определение функции; его первый аргумент — имя функции, второй — список её аргументов, третий — тело функции.

Всё это делает LISP удобным и очень мощным языком, а его очевидная древовидная структура позволяет удобно применять этот язык в качестве основы для генетического программирования. Программы на LISP могут получаться весьма компактными, что также делает генетическое программирование более эффективным: чем меньше размер объекта, тем легче его автоматически найти или породить.

Хотя это уже в меньшей степени относится к основанному на LISP генетическому программированию, сила этого языка проявляется в том, что код программы для него является такими же данными, как и всё остальное; LISP позволяет, в основном через определение макросов, быстро и эффективно создавать достаточно сложные программы. К сожалению, доказать это

утверждение вряд ли получится: нужно достаточно долго разбираться в LISP и написать несколько программ на нём, чтобы перейти к макросам, а мы сейчас всё-таки говорим не столько о программировании, сколько об искусственном интеллекте и алгоритмической стороне вопроса. Позже, в § 6.6, мы ещё приведём пример полноценной программы на LISP, которая будет реализовывать один из алгоритмов кластеризации, но и её будет недостаточно. Мы можем только порекомендовать читающим эту книгу программистам изучить LISP — ведущие специалисты соглашаются, что этот язык может вывести программиста на новый уровень понимания; см., например, статью Пола Грэхема «Beating the Averages» [54].

§ 4.11. Заключение

Подводя итог главе о генетических алгоритмах, напомним основную идею. Генетические алгоритмы — это не набор конкретных рецептов для конкретных задач, а философия, метод, позволяющий достаточно быстро «выводить» хороших «особей», если можно разумно определить операцию «скрещивания». Поэтому генетические алгоритмы применяются не просто широко, а очень широко; они встречаются в сочетании чуть ли не с каждым другим аппаратом машинного обучения (здесь по-хорошему нужно было бы дать ссылки на такие применения, но собрать *настолько* широкую библиографию — это достойно отдельного исследования, ведь, разумеется, ссылками на базовые работы по генетическому программированию или по идеям генетических алгоритмов тут не отделаться).

В конце главы позволим себе ещё одно небольшое лирическое отступление. Читатели уже наверняка заметили, что мы часто употребляем слова, которые странно видеть в математическом тексте: «возможно», «вероятно», «зачастую», «неплохо», «относительно быстро» и т.д. Дело в том, что в искусственном интеллекте обычно можно чем угодно сделать что угодно. Теоретически никто не мешает генетическим алгоритмом «вывести», например, программу, которая обыграет в шахматы чемпиона мира. Однако на практике этот метод оказывается неэффективным. С другой стороны, для своих задач генетические алгоритмы оказываются эффективнее других аппаратов

машинного обучения (как правило, это касается задач порождения программ или схем, как в § 4.8). Математически доказать утверждения вроде «лучше справляются с практическими задачами» обычно трудно. Приходится высказываться осторожно и обтекаемо и при первом же удобном случае проверять такого рода утверждения практикой. Но практика вполне может зависеть от деталей конкретной инженерной реализации, да и просто от конкретных констант и параметров. Задача этой книги — не в том, чтобы строго доказать, что тот или иной аппарат машинного обучения работает лучше всех остальных, а в том, чтобы дать читателю некий набор инструментов, box of tools, которые читатель сможет применить к своей конкретной задаче. Однако заранее, абстрактно, ответить на вопрос о том, что именно лучше применять, невозможно: у большинства описываемых аппаратов есть свои достоинства и недостатки.

В следующей главе мы расскажем, как численно сравнивать гипотезы друг с другом, оценивая их правдоподобие; теория вероятностей, лежащая в основе всего машинного обучения, наконец-то станет предметом нашего разговора.

Глава 5

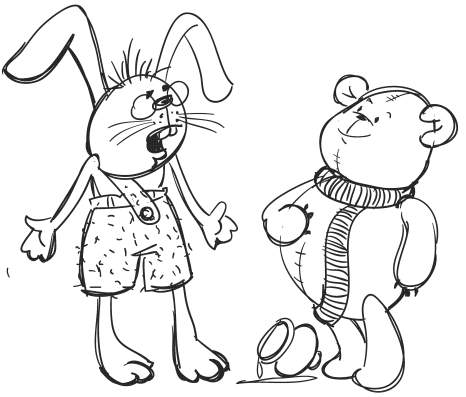
Байесовское обучение и классификаторы

> Случай порождает Отклонения от закономерностей, однако бесконечно велики Шансы, что с течением Времени эти Отклонения окажутся пренебрежимо ничтожными относительно повторяемости того Порядка, который естественным образом является результатом Божественного Предначертания.
>
> *Теория случайностей*
> *Абрахам де Муавр*

Когда Иа-Иа потерял хвост, все жители Леса очень хотели помочь ослику отыскать его.

«Может, хвост Иа-Иа захотел мёда», — вполне логично подумал Винни-Пух и решил посмотреть у себя в шкафу. В шкафу стояли пустые горшочки, которые Винни уже классифицировал, но хвоста не оказалось. Зато среди пустых горшочков оказался один полный...

Когда к Пуху пришёл Кролик, сытый и довольный медвежонок рассказал ему, как он догадался поискать хвост в шкафу.



— Хвост? Мёда?! Это совершенно *не-ве-ро-ят-но*! — отчеканил Кролик.

— Не-ве-ро-ят-но? — повторил Пух. — Что это значит? А если бы хвост Иа-Иа оказался у меня в шкафчике с мёдом, это вдруг стало бы ве-ро-ят-но, да?

— Пух, но ведь шансы обнаружить там хвост были ничтожно малы! — раздражённо объяснил Кролик. — Конечно, если бы он там *вдруг* оказался... но это же абсолютно невозможно!

— Зато я нашёл там горшочек с мёдом, — не согласился Пух. — А поиски хвоста — это такое сложное дело, что обязательно нужно было как следует подкрепиться.

Так получилось, что медвежонок выдвинул гипотезу: «хвост Иа-Иа у меня в шкафу». И решил там его поискать. Однако, с точки зрения Кролика, эта гипотеза настолько маловероятна, что действовать в соответствии с ней было глупо; Кролик скорее стал бы искать хвост на любимых осликом кустах чертополоха. А Пух, видимо, попутно оценивал вероятность гипотезы «у меня в шкафу найдётся чем подкрепиться», и результат этой оценки оказался весьма приятным...

§ 5.1. Введение

Методы, которые мы до сих пор рассматривали, могли иметь успех и без обращения к теории вероятностей. В этой главе мы переходим к рассмотрению так называемых *байесовских методов* обучения — методов, для которых теория вероятностей играет не вспомогательную, а основополагающую роль.

Сразу отметим, что вероятностные рассуждения в машинном обучении могут употребляться в двух разных контекстах. Первый — когда вероятности или их аналоги реально применяются алгоритмами машинного обучения для вывода. Например, алгоритм EM для кластеризации (см. § 6.6) поддерживает и пересчитывает вероятности принадлежности каждой точки каждому из кластеров, и результат его работы — это фактически и есть эти самые вероятности.

Второй контекст — когда сам алгоритм или вообще не имеет вероятностной природы, или она надёжно скрыта, но при этом вероятностные рассуждения используются для *анализа* работы алгоритма. Например, вероятностные методы часто позволяют

доказать, что алгоритм находит оптимальную гипотезу или сходится к ней, то есть работает наилучшим возможным образом (см., например, § 5.5, где мы проанализируем с байесовской точки зрения уже знакомые нам по Главе 1 алгоритмы классификации).

Трудно переоценить важность доказательств такого рода. Алгоритмы машинного обучения так часто используют всяческие эвристики и так редко гарантируют результат, что доказательство того, что алгоритм действительно выдаёт оптимальный результат или хотя бы сходится к нему — очень важный шаг. Дальше неплохо было бы понять, *как быстро* он к нему сходится. Здесь, конечно, вероятностные методы тоже играют важнейшую роль, но в этой книге мы не будем приводить примеров подобных доказательств.

§ 5.2. Теорема Байеса

Начнём с того, что вкратце напомним собственно теорему Байеса. Мы предполагаем, что читатель знает о том, что такое вероятность: функция p, заданная на некотором пространстве событий $\langle \mathcal{S}, \mathcal{X} \rangle$ (где $\mathcal{S}$ — пространство элементарных событий, а $\mathcal{X}$ — сигма-алгебра на нём) так, чтобы выполнялись следующие условия.

P1. $p(X) \geqslant 0$ для любого $X \in \mathcal{X}$.

P2. $p(\mathcal{S}) = 1$.

P3. Если не более чем счётное множество событий $\{X_i\}_{i \in I}$ таково, что $X_i \cap X_j = \emptyset$ для всех $i \neq j$, и $X_i \in \mathcal{X}$, $\bigcup_{i \in I} X_i \in \mathcal{X}$, то

$$p\left(\bigcup_{i \in I} X_i\right) = \sum_{i \in I} p(X_i).$$

При выполнении этих условий тройку $\langle \mathcal{S}, \mathcal{X}, p \rangle$ называют *вероятностным пространством*. Подробнее о современном понятии вероятности можно прочесть в [178, 179]; в дальнейшем мы не будем пользоваться теорией меры, которая неизбежно сопутствует серьёзным вероятностным рассуждениям, а ограничимся элементарной теорией вероятностей, как правило, на конечном множестве событий.

Для нас особенный интерес представляют *условные вероятности*, поскольку именно на условных вероятностях, как мы вскоре увидим, построен анализ буквально всех аппаратов машинного обучения.

Определение 5.1. *Условной вероятностью* события X при условии события Y с $p(Y) > 0$ называется величина

$$p(X|Y) = \frac{p(XY)}{p(Y)}.$$

Легко видеть, что выполняются следующие свойства условных вероятностей:

$$p(X|X) = 1,$$
$$p(\emptyset|X) = 0,$$
$$p(X|Y) = 1, \quad X \supseteq Y,$$
$$p(X_1 \cup X_2|Y) = p(X_1|Y) + p(X_2|Y), \quad X_1 \cap X_2 = \emptyset.$$

Ограничение $p(Y) > 0$ можно снять, если переформулировать определение так: условная вероятность X при условии Y — это такая величина $p(X \mid Y)$, что $p(X \mid Y)p(Y) = p(XY)$. В таком случае $p(X \mid Y)$ при $p(Y) = 0$ не остаётся неопределённой, как в определении 5.1, а просто может принимать любое значение (наверное, это ещё одно применение принципа Дунса Скота[1]: «из лжи следует что угодно»).

Важно заметить, что если зафиксировать множестве A, то условная вероятность $p(\cdot|A)$ обладает теми же свойствами на

[1] Иоанн Дунс Скот (John Duns Scotus, ок. 1266–1308) — философ, теолог и логик, родившийся, по одной из версий, в Ирландии, а по другой — в Шотландии, в городе Дунс. Как и Оккам, он был британцем и францисканцем, как и Оккаму, ему пришлось уехать из Франции (правда, по прямо противоположной причине: поддержал папу в борьбе против Филиппа Красивого, и Филипп уволил Скота из Парижского Университета) и обосноваться в Германии (в Кёльне, где он в конце концов умер и был похоронен). Скот, в отличие от Оккама, был реалистом (в смысле противоположности номинализму), а как теолог проповедовал волюнтаризм (свободу человеческой воли). Его главные научные работы относятся к логике, где Скот впервые ввёл многие современные логические концепции, в том числе и принцип «из лжи следует всё что угодно». Стоит отметить, что последователи Скота были не столь умны, как сам философ — английское слово «dunce» («дурак, болван») происходит от имени Duns и изначально относилось к чересчур упорным в своих заблуждениях дунсистам.

пространстве $(\mathcal{S} \cap A, \mathcal{X} \cap A)$, что и исходная вероятность p на пространстве $(\mathcal{S}, \mathcal{X})$.

Теперь приведём несколько фактов об условных вероятностях, которые будут постоянно использоваться в дальнейшем. Будем говорить, что события $X_1, \ldots, X_i$ образуют *разбиение* $\mathcal{S}$, если $\bigcup_{i=1}^{n} X_i = \mathcal{S}$, $X_i \cap X_j = \emptyset$ и $p(X_i) > 0$, $i,j \in 1..n$, $i \neq j$.

Предложение 5.1.

1. Формула полной вероятности. *Если события* $X_1, \ldots, X_i$ *образуют разбиение* $\mathcal{S}$, *то*

$$p(Y) = \sum_{i=1}^{n} p(Y|X_i)p(X_i).$$

В частности, если $p(X) > 0$, *то*

$$p(Y) = p(Y \mid X)p(X) + p(Y \mid \bar{X})p(\bar{X}).$$

2. Формула умножения вероятностей. *Если события* X_1, $X_2, \ldots, X_i$ *таковы, что* $p(X_1 \cap X_2 \cap \ldots \cap X_i) > 0$, *то*

$$p(X_1 \cap \ldots \cap X_i \cap Y) =$$
$$= p(X_1)p(X_2 \mid X_1)p(X_3 \mid X_1 \cap X_2) \ldots p(Y \mid X_1 \cap \ldots \cap X_i)).$$

Доказательство. Доказательство всех трёх пунктов представляет собой несложную индукцию. Мы оставим его читателю в качестве упражнения (см., например, [178]). □

Центральным для всех наших дальнейших вероятностных рассуждений станет одно следствие предложения 5.1.

Теорема 5.1 (Теорема Байеса). *Если события* $X_1, \ldots, X_i$ *образуют разбиение* $\mathcal{S}$, *и* $p(Y) > 0$, *то*

$$p(X_i|Y) = \frac{p(X_i)p(Y|X_i)}{\sum_{j=1}^{n} p(X_j)p(Y|X_j)}.$$

В частности,

$$p(X \mid Y) = \frac{p(X)p(Y \mid X)}{p(Y)}.$$

Доказательство. Немедленно следует из формулы полной вероятности: $p(XY) = p(X|Y)p(Y) = p(Y|X)p(X)$. □

Формулировка теоремы Байеса[1], как видите, очень проста; её математический смысл в том, что теорема Байеса устанавливает несложное соответствие между $p(x|y)$ и $p(y|x)$. Однако это несложное соответствие и методы, на нём основанные, оказываются крайне важными для всей математической статистики и машинного обучения.

Мы завершим этот параграф забавным и весьма показательным примером, который продемонстрирует, что не всё в формуле полной вероятности и теореме Байеса так уж интуитивно очевидно.

Пример 5.1. Задача о трёх ящиках

Представьте, что вы играете в азартную телеигру и вдруг получили шанс на случайный выигрыш. Перед вами три чёрных ящика, из которых приз лежит только в одном (да, жизнь — тяжёлая штука).

Вы выбираете один из трёх ящиков, после чего ведущий открывает один из двух оставшихся (очевидно, он всегда может так сделать) и показывает вам, что в нём ничего нет. Вопрос: выгодно ли вам изменить свой выбор, то есть взять второй, оставшийся закрытым ящик?

Интуитивно может показаться, что выбор менять бессмысленно, ведь ведущий не дал вам никакой информации: вам и так было известно, что один из оставшихся ящиков пустой. Задайте своим знакомым этот вопрос, и вы увидите: многие решат, что менять решение смысла нет. Однако давайте попробуем подсчитать это формально.

[1] Томас Байес (Thomas Bayes, 1702–1761) — удивительный пример человека, который почти не публиковался и был долгое время не очень известен, но чьё имя осталось в веках в бесчисленных определениях с прилагательным «байесовский», появившихся в последние полвека. Преуспевающий пресвитерианский священник за всю жизнь опубликовал только два труда: «Благость господня, или попытка доказать, что конечной целью божественного провидения и направления является счастье его созданий» и анонимно опубликованное «Введение в теорию флюксий, или в защиту математиков от нападок автора “Комментатора”», где Байес защищал ньютоновский анализ от критики Беркли. Его работа по теории вероятностей вышла уже после смерти, в 1763 году, и в ней Байес ответил на один из вопросов, оставленных открытым основателем теории вероятностей де Муавром (правда, и здесь с именем и приоритетом не всё так очевидно — говорят, Николь Саундерсон его опередил [148]). Впрочем, стоит подчеркнуть, что теорема Байеса — это для нас несложное следствие очевидных свойств вероятности; во времена Байеса и де Муавра эти свойства ещё не были столь чётко сформулированы, и само понятие вероятности ещё не было толком разработано.

Рассмотрим случайную величину m (от слова money) с тремя значениями, соответствующую шкатулке, в которой лежат деньги. Изначально у вас не было возможности предпочесть один из ящиков, то есть вероятности всех исходов изначально равны:

$$p(m = 1) = p(m = 2) = p(m = 3) = \frac{1}{3}.$$

Для определённости предположим, что вы выбрали шкатулку номер один. Рассмотрим новую случайную величину x, соответствующую тому, какую из двух шкатулок открыл ведущий (у этой величины два значения) и предположим для простоты, что если вы угадали правильно, он открывает один из двух других ящиков случайным образом. Поведение ведущего тогда можно описать такой таблицей:

$$p(x = 2|m = 1) = 1/2, \quad p(x = 2|m = 2) = 0, \quad p(x = 2|m = 3) = 1,$$
$$p(x = 3|m = 1) = 1/2, \quad p(x = 3|m = 2) = 1, \quad p(x = 3|m = 3) = 0.$$

Предположим, что ведущий открыл ящик номер 2 (случай третьего ящика ничем не отличается). Вычислим условные вероятности попадания приза в ящики 1 или 3, используя теорему Байеса:

$$p(m = 1|x = 2) = \frac{p(m = 1)p(x = 2|m = 1)}{\sum_m p(m)p(x = 2|m)} = \frac{\frac{1}{3} \cdot \frac{1}{2}}{\frac{1}{3} \cdot \frac{1}{2} + \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1} = \frac{1}{3},$$

$$p(m = 3|x = 2) = \frac{p(m = 3)p(x = 2|m = 3)}{\sum_m p(m)p(x = 2|m)} = \frac{\frac{1}{3} \cdot 1}{\frac{1}{3} \cdot \frac{1}{2} + \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1} = \frac{2}{3}.$$

Иначе говоря, изменить решение в среднем вдвое выгоднее, чем не менять его!

Оставляем читателю в качестве упражнения случай, когда ведущий выбирает открываемый ящик не равновероятно ($p(x = 2|m = 1)$ не обязательно равно $1/2$).

«Парадоксальный» ответ примера 5.1 можно, конечно, понять и интуитивно. Всё дело в том, что два ящика выбирать действительно выгоднее, чем один. Если бы вам безо всяких открываний предложили изменить решение и взять два оставшихся ящика, вы бы не колеблясь так и поступили. Однако на самом деле ведущий предлагает именно это: он делает выбор между двумя оставшимися ящиками за вас, и теперь, если приз был в одном из них, вы его непременно получите.

§ 5.3. Априорные и апостериорные вероятности

Итак, нашим основным инструментом станет теорема Байеса:

$$p(x \wedge y) = p(x|y)p(y) = p(y|x)p(x).$$

Отсюда следует, что если $p(y) \neq 0$, то

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}.$$

Давайте рассмотрим это выражение подробнее; для успешного применения формулы Байеса нам придётся дать имена отдельным её частям и понять, что они означают в философском смысле.

Формула для $p(x|y)$ выражает нашу цель: мы хотим найти условную вероятность события x (или, если x и y — случайные величины, того или иного значения случайной величины x) при условии, что событие y произошло (соответственно, при данном фиксированном значении случайной величины y). Это *апостериорная* вероятность x — от слов *a posteriori*, то есть вероятность «после опыта», после того, как мы узнали значение y.

Соответственно, *априорная* (a priori) вероятность — это значение вероятности x до того, как мы зафиксировали y. В формулировке теоремы Байеса присутствует априорная вероятность $p(x)$ события x; апостериорная вероятность ей прямо пропорциональна.

Ввыражение $p(y|x)$, стоящее в числителе правой части, называется *правдоподобием* x при заданном y. Именно поэтому гипотезы, которые мы в следующем параграфе назовём максимальными апостериорными гипотезами, часто называют *гипотезами максимального правдоподобия*[1]. Если быть точным, *функция правдоподобия* — это функция вида $f_a(y) = p(x = a|y)$. Теорема Байеса утверждает, что апостериорная вероятность события прямо пропорциональна его правдоподобию.

[1] Мы бы и сами использовали тот же термин, но боимся запутать читателя; дело в том, что основная масса литературы по машинному обучению написана на английском языке, и в ней уже устоялся даже не просто термин maximal a posteriori, а его аббревиатура MAP.

А вот $p(y)$ не имеет своего специального названия, её можно назвать *маргинальной вероятностью*, но это просто означает, что она не является условной. Эта вероятность и вовсе играет здесь вспомогательную роль. Подсчитать её обычно или совсем просто, или совсем сложно, и в любом случае не нужно: достаточно просто подсчитать произведения $p(y|x)p(x)$ для разных значений x, а затем вспомнить, что $p(x|y)$ — распределение вероятностей, а, значит,

$$\sum_a p(x = a|y) = \sum_a p(y|x = a)p(x = a) = 1.$$

Иначе говоря, в теореме Байеса $p(y)$ просто играет роль нормировочной константы. Теорему часто прямо в таком виде и записывают:

$$p(x = a|y = b) = \frac{p(y = b|x = a)p(x = a)}{\sum_{a'} p(y = b|x = a')p(x = a')}.$$

Завершим мы этот параграф любопытным применением теоремы Байеса (по неожиданности, на мой взгляд, даже превосходящим пример 5.1), в котором мы будем решать в точности обратную задачу теории вероятностей.

Пример 5.2. Тест с ошибкой

Обратимся к самой классической области применения статистики и байесовского вывода — к медицинской диагностике. Пусть некий тест на какую-нибудь болезнь имеет вероятность успеха 95%, причём ошибки в нём могут быть в любую сторону, 5% — вероятность как позитивной, так и негативной ошибки. Пусть всего болезнь имеется у 1% респондентов (отложим на время то, что они разного возраста и профессий). И, наконец, пусть некий человек получил позитивный результат теста (тест говорит, что он болен). С какой вероятностью он действительно болен?

Если задать этот вопрос достаточно большой группе людей, то, по нашим наблюдениям, бóльшая часть присутствующих проголосует за то, что человек болен с вероятностью более 80%, и уж точно почти все будут уверены, что человек скорее болен, чем нет. Согласны ли вы с этой оценкой?

Обозначим через t результат теста, через d — наличие болезни. Тогда

$$p(d = 1) = p(d = 1|t = 1)p(t = 1) + p(d = 1|t = 0)p(t = 0).$$

Используем теорему Байеса:

$$p(d = 1|t = 1) = \frac{p(t = 1|d = 1)p(t = 1)}{p(d = 1|t = 1)p(t = 1) + p(d = 1|t = 0)p(t = 0)} = \\ = \frac{0{,}95 \times 0{,}01}{0{,}95 \times 0{,}01 + 0{,}05 \times 0{,}99} = 0{,}16.$$

Вот такой ответ, на первый взгляд парадоксальный: получается, что среди имеющих позитивный результат теста со столь высокой степенью надёжности реально больны лишь около 16 процентов!

На самом деле этот пример, как и пример 5.1, тоже несложно интуитивно объяснить. Вероятность ошибки теста впятеро больше априорной вероятности наткнуться на больного пациента. Поэтому ситуация «больной пациент, правильный тест» будет встречаться гораздо реже ситуации «здоровый пациент, ошибка в тесте». Но практика показывает, что интуиция эта далеко не всегда встречается даже среди людей,прошедших курсы теории вероятностей. Надеемся, что наш читатель в такие ловушки уже не попадётся.

§ 5.4. Теорема Байеса, данные и гипотезы

В предыдущих главах мы рассмотрели несколько весьма различных алгоритмов машинного обучения. Однако в их структуре было очень много общего. Это общее мы сейчас выделим и поймём, что на самом деле все описанные нами алгоритмы пытались сделать одно и то же.

Во-первых, у каждого алгоритма есть некоторое *множество гипотез*, из которых он пытается выбрать наилучшую. Например, для алгоритма обучения концептам Find-S (см. § 2.4) множеством гипотез было множество правил вида «из такого-то набора атрибутов следует, что целевая функция равна единице». У алгоритма ID3 (см. § 1.5) множеством гипотез было множество возможных деревьев принятия решений или, что для нас сейчас эквивалентно, множество всевозможных дизъюнкций гипотез алгоритма Find-S.[1] У нейронных сетей множество возможных гипотез было более богатым — для сетей с линейными

[1] Деревья, конечно, — более экономичный способ записи, чем набор дизъюнкций, но нам это сейчас не важно.

нейронами это было множество линейных форм в пространстве, размерность которого зависела от числа входов, а для сетей с нелинейными нейронами — множество функций определённого вида. Общий вид гипотез для генетических алгоритмов мы подробно обсуждали, когда старались закодировать их наиболее эффективным путём в § 4.4. В общем, у каждого алгоритма есть множество гипотез, на котором он ведёт свой поиск.

Во-вторых, процесс обучение немыслим без *данных*, на основании которых алгоритмы принимают свои решения и выбирают ту самую наилучшую гипотезу. Здесь уже такого разнообразия не наблюдается: фактически, любой алгоритм готов принять любые данные, нужно только отформатировать их подходящим образом. Важно, что гипотезу выбирают так, чтобы она максимально хорошо подходила под данные. Алгоритмы классификации вроде Find-S и ID3, которые рассчитаны на работу без «шума» в данных, выбирают гипотезу так, чтобы она идеально подходила под данные. Нейронные сети стараются минимизировать среднеквадратичную ошибку — в случае вещественнозначных функций идеального соответствия данным ожидать сложно.[1]. Генетические алгоритмы не могут гарантировать идеального соответствия, но стараются максимизировать функцию Fitness, которая обычно напрямую зависит от того, насколько удачно описываются имеющиеся данные.

Эти два компонента в самом общем их виде — почти всё, что нам сейчас потребуется. Стоит сделать только одно замечание. И гипотезы, и данные нам могут помочь только в каких-либо *предположениях*. Вероятностные предположения всегда нужно вербализовывать, эксплицировать... в частности, знать их заранее. Иногда получается, что предположения, которые неявно требуются для того или иного вывода, на самом деле могут дать гораздо больше. А иногда получается, что даже довольно естественные предположения оказываются достаточно сильными, чтобы однозначно решить требуемую задачу (нечто подобное, хоть тогда мы этого и не доказывали, у нас происходило

[1] Более того, в случае вещественнозначных функций довольно проблематично даже определить, что такое «идеальное соответствие данным», особенно если речь идёт о компьютерных, то есть дискретных и конечных вычислениях.

в § 1.3, когда естественных предположений о мере неопределённости оказалось достаточно для того, чтобы определить её практически однозначно).

Итак, давайте предположим, что у нас есть некоторое множество гипотез H и множество имеющихся данных D. Тогда цель любого алгоритма машинного обучения — найти гипотезу, которая была бы *наиболее вероятной* при имеющихся данных. Действительно, такое поведение было бы оптимальным — ничего более эффективного ожидать невозможно. Математически это можно записать следующим образом.

Определение 5.2. Пусть H — множество гипотез, D — множество имеющихся данных. Гипотеза $h \in \mathcal{H}$ называется *максимальной апостериорной гипотезой* (MAP, *maximum* a posteriori *hypothesis*), если

$$h = \operatorname{argmax}_{h \in \mathcal{H}} p(h|D).$$

Давайте перепишем это по теореме Байеса:

$$h = \operatorname{argmax}_{h \in \mathcal{H}} p(h|D) =$$

$$= \operatorname{argmax}_{h \in \mathcal{H}} \frac{p(D|h)p(h)}{p(D)} = \operatorname{argmax}_{h \in \mathcal{H}} p(D|h)p(h),$$

потому что $p(D)$ от h не зависит.

Часто предполагают, что гипотезы изначально равновероятны: $p(h_i) = p(h_j)$ для всех гипотез $h_i, h_j \in \mathcal{H}$. Тогда предыдущее выражение можно переписать ещё проще:

$$h = \operatorname{argmax}_{h \in \mathcal{H}} p(D|h).$$

Эти выражения — суть поиска оптимальной гипотезы. По ним можно сразу построить алгоритм поиска MAP-гипотезы: нужно для каждой гипотезы $h \in \mathcal{H}$ вычислить её апостериорную вероятность $p(h|D)$, а затем выбрать ту гипотезу, для которой эта вероятность максимальна. Разумеется, мы не рекомендуем применять такой алгоритм на практике: в любой хоть немного реалистичной задаче размер множества гипотез сравним с числом элементарных частиц во Вселенной.[1]

[1] Физики оценивают это количество как число порядка 2^{100}; иными словами, если гипотезами могут быть строки из 100 бит (не так уж и много,

Но, хотя по ним и можно построить алгоритм, основная функция выражений для поиска максимальной апостериорной гипотезы — не прямо практическая. MAP-гипотеза нужна для того, чтобы сравнивать с ней другие алгоритмы и выяснять, когда они работают оптимально. Пример такого подхода мы приведём в следующем параграфе.

§ 5.5. MAP и задачи классификации

Для того чтобы найти максимальную апостериорную гипотезу, нужно научиться вычислять $p(h)$ и $p(D|h)$. Пусть выполняются следующие условия:

- в D нет «шума» (т.е. все тестовые примеры содержат правильные ответы);
- целевая функция c содержится среди гипотез $\mathcal{H}$;
- нет априорных причин верить, что одна из гипотез более вероятна, чем другая.

Эти предположения весьма обычны для задач классификации. Они выполнялись и для алгоритма ID3 из Главы 1, и для алгоритмов FindS и элиминации кандидатов из Главы 2. Единственное, что может осложнить поиск максимальной апостериорной гипотезы — это возможный «шум» в данных, которого мы уже касались, обсуждая проблему оверфиттинга в § 1.8. Но мы пока закроем глаза на эту проблему — всё равно большинство алгоритмов классификации, нами рассмотренных, справиться с ней сами по себе, без внешней помощи, не могут.

Из третьего условия следует:

$$p(h) = \frac{1}{|\mathcal{H}|} \text{ для всех } h \in \mathcal{H}.$$

Условная вероятность $p(D|h)$ — это вероятность наблюдать значения целевых функций $\langle t_1, \ldots, t_m \rangle$ для фиксированного набора входных данных $\langle d_1, \ldots, d_m \rangle$ при условии, что выполняется гипотеза h. Поскольку «шума» нет, $p(t_i|h) = 1$, если $t_i = h(d_i)$, и $p(t_i|h) = 0$ в противном случае. Итого:

$$p(D|h) = \begin{cases} 1, & \text{если } d_i = h(x_i) \text{ для всех } d_i \in D, \\ 0, & \text{в противном случае.} \end{cases}$$

правда? эта сноска раз в десять длиннее), размер множества гипотез как раз будет порядка количества частиц во Вселенной.

(здесь вход гипотезы x_i является частью тестового примера d_i).

Для полноты картины, хотя это и не нужно для определения MAP-гипотезы, давайте подсчитаем вероятность $p(D)$. Обозначим через $\mathrm{Cons}(D)$ множество гипотез $h \in \mathcal{H}$, совместных с D. Тогда

$$p(D) = \sum_{h \in \mathcal{H}} p(D|h)p(h) = \sum_{h \in \mathrm{Cons}(D)} \frac{1}{|\mathcal{H}|} = \frac{|\mathrm{Cons}(D)|}{|\mathcal{H}|}.$$

Итак, получаем:

$$p(h|D) = \frac{p(D|h)p(h)}{p(D)} = \begin{cases} \frac{1}{|\mathrm{Cons}(D)|}, & \text{если } \forall i\ d_i = h(x_i), \\ 0, & \text{в противном случае.} \end{cases}$$

Иными словами, каждая гипотеза, совместная со всеми данными — это максимальная апостериорная гипотеза. О чём это нам говорит? О том, что все алгоритмы, которые выдают гипотезы, совместные со всеми входящими данными, *в вышеописанных предположениях* выдают максимальные апостериорные гипотезы. Это значит, что они работают идеально — ничего более хорошего сделать принципиально невозможно.

Но позвольте — скажет внимательный читатель, — разве мы не посвятили добрую половину Главы 1 тому, чтобы выбирать из разных деревьев, одинаково идеально подходящих тестовым данным, *оптимальное* (под оптимальным мы обычно понимали дерево, имеющее минимальную глубину или минимальное число узлов)? А теперь вдруг оказывается, что всё это было не важно, и на самом деле *любое* подходящее дерево реализует максимальную апостериорную гипотезу... нехорошо как-то получается. Внимательный читатель будет абсолютно прав. Но причин для беспокойства всё равно нет — дело в том, что в этом параграфе мы изначально предположили, что все гипотезы a priori равновероятны. А в Главе 1 мы, напротив, предполагали, что деревья меньшей глубины и размера будут *более вероятны*, чем более развесистые деревья. В контексте байесовского вывода такое предположение выражается присваиванием *разных априорных вероятностей* разным гипотезам. Поиски подходящих поправок на размер дерева в Главе 1 (критерий

прироста информации, индекс Джини) были на этом языке поисками адекватных априорных вероятностей для деревьев разного размера. Предоставляем читателю самому вывести формулы априорных вероятностей различных деревьев, при которых алгоритм ID3, использующий критерий прироста информации, будет осуществлять поиск максимальной апостериорной гипотезы.

В данном случае мы байесовскими методами пришли к доказательству того, что какие-то другие (небайесовские по сути) алгоритмы работают оптимально. Это одна из важнейших функций байесовских методов — тот редкий в искусственном интеллекте случай, когда что-то действительно можно (и совсем несложно, как мы уже убедились) математически доказать.

Впрочем, результат, который мы получили, даёт больше информации, чем простое подтверждение того, что тот или иной алгоритм решает поставленную задачу. Мы получили конкретный, чётко математически определённый набор предположений, при которых алгоритм классификации работает оптимальным образом. Это уже напрямую даёт программу действий: если найденные предположения выполняются, то можно пользоваться имеющимся алгоритмом, не желая ничего лучшего. А вот если какие-то условия не выполняются, то можно искать (это, правда, ещё не значит, что удастся найти) алгоритмы, которые работали бы лучше. Более того, в таком случае мы будем знать, какие именно предположения не выполняются, и это, скорее всего, поможет разработать алгоритм, который будет ближе к оптимальному.

Позднее мы приведём другие примеры аналогичного применения байесовских методов, а пока обратимся к более практическим вопросам — построению байесовских классификаторов.

§ 5.6. Оптимальный и гиббсовский классификаторы

До сих пор мы отвечали на вопрос: «Какова наиболее вероятная гипотеза при имеющихся данных?». Но на самом деле нас не интересуют гипотезы — что проку в том, чтобы знать, что данные игр «Зенита» с наибольшей вероятностью описываются той или иной гипотезой? Практически важный вопрос стоит по-другому: нужно реально продолжать функцию классификации,

а не просто выдвигать о ней правдоподобные гипотезы. Таким образом, теперь пора ответить на вопрос: «В какой класс новый пример при имеющихся данных попадает с наибольшей вероятностью?».

Казалось бы, можно просто применить максимальную апостериорную гипотезу. Однако этот метод не всегда приводит к оптимальным результатам.

Пример 5.3. Когда не нужно применять MAP

Пусть множество гипотез состоит из четырёх элементов, и их апостериорные вероятности равны 0,2, 0,2, 0,2 и 0,4 соответственно. Четвёртая гипотеза — максимальная апостериорная. Но если новый пример классифицируется первыми тремя гипотезами положительно, а четвёртой — отрицательно, то общая вероятность его положительной классификации составит 0,6, и применять MAP было бы неправильно.

Пусть имеются данные D и множество гипотез H. Для вновь поступившего примера x нужно выбрать такое значение $v \in V$ (через V мы здесь обозначаем множество возможных классов, куда может попасть $h(x)$), чтобы максимизировать $p(v|D)$. Иными словами, наша задача — найти

$$\operatorname{argmax}_{v \in V} \sum_{h \in \mathcal{H}} p(v|h)p(h|D).$$

Определение 5.3. Любой алгоритм, который решает задачу

$$\operatorname{argmax}_{v \in V} \sum_{h \in H} p(v|h)p(h|D),$$

называется *оптимальным байесовским классификатором (*optimal Bayes classifier*)*.

Пример 5.4. Продолжение примера 5.3

В примере 5.3 мы рассматривали четыре гипотезы h_i, $i = 1..4$ с множеством значений $V = \{0,1\}$. Вероятности распределялись следующим образом:

$$\begin{array}{ll} p(h_1|D) = p(h_2|D) = p(h_3|D) = 0{,}2, & p(h_4|D) = 0{,}4, \\ p(x = 1|h_1) = p(x = 1|h_2) = p(x = 1|h_3) = 1, & p(x = 1|h_4) = 0, \\ p(x = 0|h_1) = p(x = 0|h_2) = p(x = 0|h_3) = 0, & p(x = 0|h_4) = 1. \end{array}$$

Тогда

$$\sum_{i=1}^{4} p(x=1|h_i)p(h_i|D) = 0{,}6, \quad \sum_{i=1}^{4} p(x=0|h_i)p(h_i|D) = 0{,}4,$$

и оптимальный классификатор будет классифицировать этот пример как положительный, а не как отрицательный, хотя так рекомендует поступить MAP-гипотеза.

Отметим, что оптимальный классификатор действительно оптимален: никакой другой метод не может в среднем превзойти его. Он может даже классифицировать данные по гипотезам, не содержащимся в H; в частности, он может классифицировать по любому элементу линейной оболочки H.

Но, с другой стороны, оптимальный байесовский классификатор обычно не получается эффективно реализовать — нужно перебирать все гипотезы, а всех гипотез очень много. Поэтому приходится искать возможности ускорить этот процесс.

В оптимальном классификаторе мы должны для каждого возможного ответа вычислить взвешенную сумму правдоподобий всех гипотез при условии такого ответа. Это выражение до известной степени напоминает выражение для определения математического ожидания. Поэтому напрашивается вполне логичное ускорение алгоритма: вместо суммы по всем гипотезам $\sum_{h\in\mathcal{H}} p(v|h)p(h|D)$ можно рассмотреть случайную гипотезу, взятую с распределением $p(h|D)$. Такой метод называется *классификатором Гиббса*.[1]

В итоге алгоритм получается довольно простой.

1. Выбрать случайную гипотезу $h \in \mathcal{H}$ согласно распределению их апостериорных вероятностей.

[1] Джосайя Уиллард Гиббс (Josiah Willard Gibbs, 1839–1903) — знаменитый американский физик, который фактически создал теоретические основы химической термодинамики; достаточно неожиданно увидеть его имя в связи с этими алгоритмами, к которым он не имел ни малейшего отношения. На самом деле это имя попало сюда транзитом через *сэмплирование по Гиббсу* (Gibbs sampling), которое так называется потому, что это сэмплирование напоминает статистическую физику. Кстати, сэмплирование по Гиббсу тоже не Гиббс придумал — алгоритм был разработан через восемьдесят лет после его смерти [49]; более подробно о сэмплировании по Гиббсу см. [27, 48, 133].

2. Классифицировать новый случай x согласно h.

Мы здесь оставляем за кадром подробности того, как сэмплировать с заданными вероятностями — это может зависеть от конкретного множества гипотез. Но обычно это получается реализовать гораздо более эффективно, чем вычислять сумму по всем гипотезам. Главное преимущество этого метода в том, что гиббсовский классификатор сходится к оптимальному решению с ошибкой, лишь вдвое больше ошибки оптимального классификатора (при определённых не слишком жёстких условиях) [35, 99, 103, 142]. Поэтому алгоритм Гиббса обычно оказывается предпочтительнее.

§ 5.7. Наивный байесовский классификатор

Оптимальный классификатор и гиббсовский классификатор оставляли за кадром задачу поиска апостериорных вероятностей гипотез. Свой ответ на этот вопрос предлагает *наивный байесовский классификатор* (naïve Bayes)[1]. Впервые его использовали ещё в начале шестидесятых [102, 110], и с тех пор наивный байесовский классификатор используется очень широко. Он делает настолько сильные предположения, что даже не верится, что он может хорошо работать. Однако практика показывает, что наивный байесовский классификатор оказывается весьма эффективен для задач классификации с большим количеством параметров, таких, например, как классификация текстов. В последние лет десять, кстати, появились теоретические доказательства эффективности наивного байесовского классификатора, и хотя они не входят сейчас в нашу задачу, мы порекомендуем читателю эти действительно весьма интересные исследования [39, 61, 166].

Постановка задачи ничем не отличается от любой другой задачи классификации: каждый пример x принимает значения из некоторого множества V и описывается атрибутами

$$\langle a_1, a_2, \ldots, a_n \rangle.$$

Требуется найти наиболее вероятное значение данного атрибута:

$$v_0 = \operatorname{argmax}_{v \in V} p(x = v | a_1, a_2, \ldots, a_n).$$

[1] Иногда его даже называют Idiot's Bayes, но мы не рискнём употреблять этот термин и переводить его на русский язык.

По теореме Байеса,

$$v_{\text{MAP}} = \operatorname{argmax}_{v \in V} \frac{p(a_1, a_2, \ldots, a_n | x = v) p(x = v)}{p(a_1, a_2, \ldots, a_n)} =$$
$$= \operatorname{argmax}_{v \in V} p(a_1, a_2, \ldots, a_n | x = v) p(x = v).$$

Оценить $p(x = v)$ легко: при наличии даже не слишком большого числа тестовых примеров можно оценить частоту встречаемости каждого из значений x. Но оценить разные вероятности

$$p(m = 1, m = 2, \ldots, m = n | x = v)$$

не представляется возможным — их слишком много. Для того чтобы получить оценки этих вероятностей, нам фактически нужно каждую из возможных комбинаций атрибутов пронаблюдать по несколько раз. Это, разумеется, на практике невозможно.

Поэтому наивный байесовский классификатор предполагает условную независимость атрибутов при условии данного значения целевой функции. Иначе говоря,

$$p(a_1, a_2, \ldots, a_n | x = v) = p(a_1 | x = v) p(a_2 | x = v) \ldots p(a_n | x = v).$$

Теперь уже даже относительно небольшого количества тестовых примеров достаточно для того, чтобы весьма надёжно оценить каждую из этих вероятностей.[1]

Итак, наивный байесовский классификатор выбирает v как

$$v_{\text{NB}}(a_1, a_2, \ldots, a_n) = \operatorname{argmax}_{v \in V} p(x = v) \prod_{i=1}^{n} p(a_i | x = v).$$

В практической задаче распознавания текстов (например, при создании спам-фильтра) атрибутом будет появление того или иного слова в тексте. Соответственно, число атрибутов получается совершенно запредельным для прямого перебора: их по меньшей мере несколько десятков тысяч. Более того, если строить полную модель текста, то нужно учитывать местоположение слов друг относительно друга, что умножает число атрибутов на длину документа. Поэтому метод, устанавливающий

[1] «Надёжно» оценить вероятность — это значит построить оценку с достаточно маленьким *доверительным интервалом*.

условные независимости атрибутов друг относительно друга, совершенно необходим. Но наивный байесовский классификатор предполагает, на первый взгляд, какие-то совершенно фантастические вещи: получается, что появление слова не зависит от других слов, его окружающих, и даже, более того, не зависит от длины документа. И, тем не менее, наивный байесовский классификатор оказывается на редкость эффективен.

§ 5.8. Атрибуция текстов

В этом параграфе мы рассмотрим пример конкретного применения наивного байесовского классификатора. Задача достаточно традиционна для подобного рода инструментов и относится к широкому классу задач «интеллектуального анализа текстов». Мы будем заниматься *атрибуцией текстов*: определением по данному тексту его авторства.

Наша программа будет реализована на языке C#. Поразительно, но для достижения результата потребуется не так уж много кода.

Итак, первый файл содержит класс `MyDictionary`, в котором реализован словарь, пополняющийся для каждого из тестовых текстов. Отметим, что русский язык достаточно сложен для классификации: нужно уметь распознавать одинаковые слова, хотя они могут стоять в разных грамматических формах. В идеале для этого нужно реализовывать достаточно полный словарь всего языка или интеллектуальную систему распознавания словоформ; однако для нашего примера мы ограничимся удалением окончаний; массив `ending` содержит наиболее распространённые окончания, которые мы будем просто «отрезать» от слов, если они на них заканчиваются. А чтобы в результате не получались многочисленные одно-двухбуквенные слова, мы введём ограничение на длину слова в словаре (не менее четырёх символов, не считая окончаний).

Листинг 5.1. Naive Bayes на C#: `MyDictionary`

```
using System;
using System.Collections.Generic;
using System.IO;

namespace TextClassificator {
```

```
public class MyDictionary {
    private static readonly string[] ending =
      new string[51] {
        "ому", "ему", "ами", "ыми", "ями", "ими", "ашь",
        "ешь", "ишь", "ого", "его", "ый", "ой", "ий",
        "ые", "ие", "ое", "ее", "ия", "ая", "ию", "ии",
        "ие", "ую", "ах", "ых", "их", "ть", "ов", "ев",
        "им", "ом", "ым", "ам", "ям", "ем", "ут", "ют",
        "ит", "ет", "ат", "ят", "а", "я", "о", "е", "ы",
        "и", "ь", "ю", "у"};

    private Dictionary<string, double[]> dict =
        new Dictionary<string, double[]>();
    public double[] this[string key] {
        get {
            string word = key;
            removeEnding(ref word);
            if(word.Length < 4) { return null; }
            if (dict.ContainsKey(word)) {
                return dict[word];
            } else {
                return null;
            }
        }
    }

    private int numOfAuthors;
    public int NumOfAuthors {
        get { return numOfAuthors; }
    }

    private int currentAuthor = 0;
    private string text;
    private int[] wordsPerAuthor;

    public MyDictionary(string path) {
        string[] dirs = Directory.GetDirectories(path);
        numOfAuthors = dirs.Length;
        wordsPerAuthor = new int[numOfAuthors];
        for (int i = 0; i < numOfAuthors; i++) {
            wordsPerAuthor[i] = 0;
            currentAuthor = i;
            string[] files = Directory.GetFiles(dirs[i]);
            for (int j = 0; j < files.Length; j++) {
                readFromFile(files[j]);
            }
        }
        writeToFile("befornorm");
```

```
        decreaseDimension();
        writeToFile("afternorm");
    }

    public void readFromFile(string filename) {
        StreamReader sr = File.OpenText(filename);
        text = sr.ReadToEnd();
        int j = 0;
        while (j < text.Length - 5) {
            readNextWord(ref j);
        }
    }

    private void readNextWord(ref int j) {
        string word = "";
        while(j < text.Length) {
            if (((text[j]>='a') && (text[j]<='я')) ||
                ((text[j]>='A') && (text[j]<='Я'))) {
                char ch = Char.ToLower(text[j]);
                word = word + ch;
                j++;
            } else {
                addWordToDictionary(word);
                if (j != text.Length) {
                    j++;
                    return;
                }
            }
        }
        Console.WriteLine(word);
        addWordToDictionary(word);
    }

    private void removeEnding(ref string word) {
        if (word.EndsWith("сь") || word.EndsWith("ся")) {
            word = word.Substring(0, word.Length - 2);
        }
        for (int i = 0; i < ending.Length; i++)
        {
            if (word.EndsWith(ending[i])) {
                word = word.Substring(
                  0, word.Length - ending[i].Length);
                return;
            }
        }
    }

    private void addWordToDictionary(string word) {
```

```
        removeEnding(ref word);
        if(word.Length < 4) {
            return;
        }

        if (dict.ContainsKey(word)) {
            dict[word][currentAuthor] += 1;
        } else {
            dict.Add(word, new double[numOfAuthors]);
            for (int j = 0; j < numOfAuthors; j++) {
                dict[word][j] = 0.1;
            }
            dict[word][currentAuthor] += 1;
        }
        return;
    }

    public void decreaseDimension() {
        List<string> delString =
          new List<string>(dict.Count);
        foreach (
          KeyValuePair<string, double[]> entry in dict) {
            for (int i = 0;
              i < entry.Value.Length; i++) {
                wordsPerAuthor[i] += (int)entry.Value[i];
            }
            bool allEquals = true;
            for (int i = 0; i < entry.Value.Length; i++) {
                if((i != 0) &&
                  (entry.Value[i - 1]!=entry.Value[i])) {
                    allEquals = false;
                    goto LABEL;
                }
            }
        LABEL:
            if(allEquals) { delString.Add(entry.Key); }
        }
        foreach (string s in delString) {
            dict.Remove(s);
        }
        foreach(
          KeyValuePair<string, double[]> entry in dict) {
            for(int i = 0; i < entry.Value.Length; i++) {
                entry.Value[i] =
                  entry.Value[i]/wordsPerAuthor[i];
            }
        }
    }
```

```
        public void writeToFile(string filename) {
            StreamWriter file = File.CreateText(filename);
            foreach (
              KeyValuePair<string, double[]> entry in dict)
            {
                file.Write(entry.Key);
                for (int i = 0;
                     i < entry.Value.Length; i++) {
                    file.Write(" " + entry.Value[i]);
                }
                file.WriteLine();
            }
            file.Close();
        }
    }
}
```

Следующий файл реализует собственно классификатор: он читает файл, поданный для классификации, и подсчитывает вероятность того, что это произведение того или иного из занесённых в класс `MyDictionary` авторов.

Листинг 5.2. Naive Bayes на C#: `TextClassificator`

```
using System;

namespace TextClassificator {
        public class Classificator {
        private MyDictionary mydict;
        private int numOfAuthors;

        public Classificator(MyDictionary mydict) {
            this.mydict = mydict;
            numOfAuthors = mydict.NumOfAuthors;
        }

        public double[] whoIsWriter(string s) {
            int j = 0;
            double[] results = new double[numOfAuthors];
            for (int l = 0; l < results.Length; l++) {
                results[l] = 1;
            }
            while (j < s.Length) {
                string word = readNextWord(s, ref j);
                if (mydict[word] != null) {
                    for (int k = 0; k < numOfAuthors; k++) {
```

```
                    results[k] *= mydict[word][k];
                }
                if(results[0] < 0.00001) {
                    for(int g = 0;
                      g < numOfAuthors; g++) {
                        results[g] *= 10000;
                    }
                }
            }
        }
        double sum = 0;
        for (int y = 0; y < results.Length; y++) {
            sum += results[y];
        }
        for (int c = 0; c < results.Length; c++) {
            results[c] = results[c]/sum;
        }
        return results;
    }

    private string readNextWord(string s, ref int j) {
        string word = "";
        while (j < s.Length) {
            if (((s[j] >= 'a') && (s[j] <= 'я')) ||
                ((s[j] >= 'A') && (s[j] <= 'Я')))  {
                char ch = Char.ToLower(s[j]);
                word = word + ch;
                j++;
            } else {
                j++;
                return word;
            }
        }
        return word;
    }
  }
}
```

И последний файл — файл с классом Main, собственно запускаемый файл. Как видно из него, нужно положить тексты для обучения в каталог test/learning (внутри которого каталоги будут соответствовать авторам), а тексты для последующей классификации — в каталог test/checking. Конечно, в реальном приложении возможность выбора рабочих каталогов стоит предоставить пользователю.

Листинг 5.3. Naive Bayes на C#: Main

```
using System;
using System.IO;

namespace TextClassificator
{
    class main
    {
        public static void Main(string[] args)
        {
            MyDictionary dict =
              new MyDictionary("test\\learning");
            Classificator cl = new Classificator(dict);
            string[] dirs =
              Directory.GetDirectories("test\\rchecking");
            for (int p = 0; p < dirs.Length; p++)
            {
                string[] files =
                  Directory.GetFiles(dirs[p]);
                for(int r = 0; r < files.Length; r++)
                {
                    StreamReader sr =
                      File.OpenText(files[r]);
                    string s = sr.ReadToEnd();
                    double[] res = cl.whoIsWriter(s);
                    for (int t = 0; t < res.Length; t++)
                    {
                        Console.Write(res[t] + " ");
                    }
                    Console.WriteLine();
                }
                Console.WriteLine("----------------");
            }
        }
    }
}
```

Этот классификатор показывает весьма приличные результаты.

Как известно, язык Perl создавался во многом специально для работы с текстом и текстовых преобразований; кроме того, он создавался как ёмкий и компактный язык. Поэтому можно предположить, что и задачу атрибуции текстов на Perl решить

можно с меньшим количеством кода и элегантнее. Действительно, так и происходит. Правда, компактность достигается во многом за счёт читабельности кода. Поэтому мы не будем подробно комментировать нижеприведённую программу: для заинтересовавшегося читателя задача самостоятельно в ней разобраться станет хорошим упражнением в языке Perl. Добавим лишь, что не все регулярные выражения одинаково полезны, и некоторые из используемых в этой программе нужны только для удаления HTML–разметки, т.е. их можно не использовать, если на вход подаётся обычный текст.

Листинг 5.4. Наивный байесовский классификатор на Perl

```
sub getText {
    $dn = shift;
    $_ = '';
    opendir adir, $dn;
    foreach $fn(readdir adir) {
        if (!-f $dn.$fn) { next; }
        print "$dn$fn\n";
        open tm, $dn.$fn;
        $_ = $_.join ' ', <tm>;
        close tm;
    }
    closedir adir;

    s/\r|\n/ /g;
    s/<head.*?<\/head>//gi;
    s/<script.*?<\/script>//gi;
    s/<style.*?<\/style>//gi;
    s/<.*?>//gi;
    s/ / /g;
    s/&\w{1,10};/ /g;
    s/&#\d*;?/ /g;
    s/[^A-Za-zA-ЯЁa-яё]/ /g;
    s/ +/ /g;
    tr/a-za-яё/A-ZA-ЯЁ/;
    return $_;
}

sub getWords {
    my $d;
    @w = sort split ' ', getText shift;
    push @w, '$';

    $n = 0;
```

```
    $w = $w[0];
    foreach (@w) {
        if ($w ne $_) {
            $d->{$w} = $n;
            $n = 1;
            $w = $_;
        } else {
            $n++;
        }
    }
    return $d;
}

print "Reading data...\n";
opendir dir, 'Data';
foreach $a(readdir dir ) {
    if ($a =~ m/\.$/) { next; }
    $dict->{$a} = getWords "Data\\$a\\";
}
closedir dir;
print "\n";
$ndict = getWords 'new\\';

print "\nClassification...\n";
@a = sort keys %$dict;
foreach (@a) {
    $dict->{$_}->{'$'} = 1;
}

foreach $w (keys %$ndict) {
    $mp = -1;
    foreach (@a) {
        $p = $dict->{$_}->{$w} - $ndict->{$w};
        $p = ($p == 0 ? 0.1 : abs $p);
        $dict->{$_}->{'$'} *= $p;
        $x = $dict->{$_}->{'$'};
        if (($mp > $x) || ($mp == -1)) {
            $mp = $x;
        }
    }
    foreach (@a) {
        $dict->{$_}->{'$'} /= $mp;
    }
}

$m = -1;
foreach (@a) {
    $x = $dict->{$_}->{'$'};
```

```
    if (($m > $x) || ($m == -1)) {
        $m = $x;
        $aut = $_;
    }
}
print "\nresult: $aut\n";
```

§ 5.9. Байесовское обучение и нейронные сети

В § 5.5 мы уже упоминали, что байесовское обучение часто оказывается стандартом, идеалом, с которым следует сравнивать другие методы обучения и выяснять, насколько тот или иной алгоритм приближается к поиску максимальной апостериорной гипотезы. В § 5.5 было доказано, что алгоритмы деревьев принятия решений, рассмотренные нами в Главе 1, и алгоритмы обучения концептам, рассмотренные в предыдущих параграфах, работают оптимальным образом. Теперь настала пора отдать старый долг: в Главе 3 мы строили алгоритмы с целью минимизировать среднеквадратичное отклонение от целевой функции

$$E(w_0,\ldots,w_n) = \frac{1}{2}\sum_{j=1}^{m}(t_j - o(x_0^j,\ldots,x_n^j))^2,$$

а в § 3.5 обещали, что расскажем здесь, почему нужно минимизировать именно её.

Итак, мы пытаемся приблизить нейронной сетью целевую функцию

$$f : \mathcal{X} \longrightarrow \mathbb{R},$$

имея набор тестовых примеров

$$D = \{\langle x_1, t_1\rangle, \ldots, \langle x_m, t_m\rangle\}.$$

В условиях приближения вещественнозначных функций не приходится ожидать, что данные точно лягут на график целевой функции. Наше главное предположение касается распределения шума, т.е. отклонения тестовых примеров от идеального графика f.

Мы предполагаем, что $t_i = f(x_i) + e_i$, где e_i — *нормально распределённый шум с нулевым средним*. Иначе говоря,

мы предполагаем, что полученный результат тестового примера укладывается на график искомой функции с точностью до нормально распределённого отклонения. Кроме того, мы предполагаем независимость тестовых примеров.

Вследствие независимости тестовых примеров можно искать максимальную апостериорную гипотезу в виде

$$h_{MAP} = \operatorname{argmax}_{h\in\mathcal{H}} p(D|h) = \operatorname{argmax}_{h\in\mathcal{H}} \prod_{i=1}^{m} p(t_i|h),$$

где $p(t_i|h)$ — нормальное распределение с вариацией σ и с центром в $\mu = h(x_i)$. Такой центр распределения получается потому, что вероятность ищется при условии гипотезы h, т.е. при условии, что гипотеза h верно описывает целевую функцию f, а это и означает, что $f(x_i) = h(x_i)$.

Тогда

$$h_{MAP} = \operatorname{argmax}_{h\in\mathcal{H}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - h(x_i))^2}.$$

Для удобства вычислений воспользуемся несложным приёмом, давно ставшим стандартным в этой науке: возьмём логарифм от функции, стоящей под argmax; на значение argmax это не повлияет:

$$h_{MAP} = \operatorname{argmax}_{h\in\mathcal{H}} \sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2 \right).$$

Удалив из-под argmax константы, как аддитивные, так и мультипликативные, которые на него также не влияют (только знак «минус» превратит argmax в argmin), получим:

$$h_{MAP} = \operatorname{argmin}_{h\in\mathcal{H}} \sum_{i=1}^{m} (d_i - h(x_i))^2.$$

Тем самым мы доказали, что минимизация среднеквадратичной ошибки ведёт к максимальной апостериорной гипотезе. Это математически оправдывает алгоритмы обучения нейронных сетей в том виде, в котором они были представлены в Главе 3.

§ 5.10. Принцип наименьшей длины описания

Всем известен философский принцип «бритвы Оккама»; его обычно записывают как «Entia non sunt multiplicanda praeter necessitatem» («Не следует умножать сущности без необходимости»). Отметим, что сам Оккам[1] так не писал, самое близкое, что удаётся найти в его трудах — «Numquam ponenda est pluralitas sine necessitate» («Не следует утверждать многое без необходимости»). Да и приоритет не очевиден: аналогичные идеи высказывали Фома Аквинский[2] и Иоанн Дунс Скот. На самом

[1] Уильям Оккам (William of Ockham или Occam, ок. 1288 – ок. 1348) — английский францисканский монах и схоласт, обучался в Оксфорде, но так и не получил даже степени M. Sc. В 1324 году он поехал в Авиньон, по одной версии — в качестве профессора философии, а по другой — в качестве ответчика, обвиняемого в ереси. Как бы то ни было, к 1327 году Оккам уже успел не сойтись с папой по одному богословскому вопросу (не о св. Августине, а о бедности Христа и апостолов — францисканцы считали, что ни у самого Иисуса, ни у его ближайших учеников вовсе не было собственности, а папе эта идея нравилась значительно меньше), и ему пришлось бежать в Баварию. С философской точки зрения, Оккам — один из первых номиналистов, некоторые называют его основателем современной эпистемологии (теории познания вообще и научного познания в частности). Главный научный труд Оккама — «Summa logicae», где он последовательно и полно изложил основы силлогистической логики.

[2] Фома Аквинский (Thomas Aquinas, ок. 1225–1274) — теолог и философ, доминиканец. Он родился в Неаполитании и по материнской линии был родственником Гогенштауфенов. С некоторыми трудностями получив M. Sc., Фома Аквинский читал лекции в Париже, Риме и других крупных городах, много путешествовал и был одним из виднейших церковных чинов. Фома Аквинский фактически открыл западному миру Аристотеля, примирив его идеи с христианскими догмами и выделив из Аристотеля его научный метод. Как теолог Фома искал свидетельства существования Бога-создателя, свидетельства наличия замысла в окружающем мире, природе, естественных вещах и процессах и не пытался прибегать для этого к откровению или трансцендентальной, априорно-логической теологии. В «Summa Theologiae» он приводит и подробно обсуждает свои знаменитые пять доказательств существования Бога, *quinquae viae*, те самые доказательства, которые пытался разрушить беспокойный старик Иммануил Кант. По иронии судьбы, 6 декабря 1273 года Фома, по его собственным словам, во время мессы испытал некий мистический опыт, получил откровение, которое чуть было не заставило его бросить недописанную «Summa Theologiae». Фома Аквинский был одним из самых влиятельных христианских философов; его труды определяли официальную позицию католической церкви в течение многих столетий.

деле нечто подобное можно найти даже у самого Аристотеля[1]; «бритва» получила имя Оккама потому, что последний очень активно применял её в своих теологических рассуждениях. Казалось бы, это базовый философский принцип — неужели его можно доказать математически и при чём вообще тут математика?

Оказывается, можно сформулировать (и доказать!) вполне формальное математическое утверждение, очень близкое к самой настоящей «бритве Оккама». Это утверждение называют «принципом наименьшей длины описания» (minimum description length principle, или просто MDL).

Чтобы получить математическую формулировку MDL, вернёмся к формуле максимальной апостериорной гипотезы и возьмём в ней логарифм:

$$h_{MAP} = \operatorname{argmax}_{h\in\mathcal{H}} p(D|h)p(h) = \\ = \operatorname{argmax}_{h\in\mathcal{H}} \left(\log_2 p(D|h) + \log_2 p(h)\right).$$

Умножая на -1, получим:

$$h_{MAP} = \operatorname{argmin}_{h\in\mathcal{H}} \left(-\log_2 p(D|h) - \log_2 p(h)\right).$$

Но $-\log_2 p(D|h)$ — это в точности шенноновская энтропия D при условии h, то есть длина описания D в оптимальном кодировании при условии того, что гипотеза h соответствует действительности. А $-\log_2 p(h)$, соответственно, — длина описания самой гипотезы h.

Иначе говоря, формула поиска максимальной апостериорной гипотезы уже сама по себе рекомендует не умножать сущности,

[1] Аристотель (Ἀριστοτελης, 384–322 до н.э.) — греческий философ, ученик Платона и учитель Александра Великого. В течение 20 лет он слушал лекции Платона в Афинах, а после смерти учителя поступил на службу сначала к Гермию, а затем к Филиппу Македонскому, который доверил Аристотелю воспитание сына; когда Александр перестал нуждаться в учителе философии, Аристотель вернулся в Афины и основал там свою знаменитую перипатетическую школу. Как учёный Аристотель фактически сформировал современный научный метод, создал формальную логику, основанную на силлогизмах. В рамках этой сноски его научные идеи невозможно даже кратко перечислить — они охватывают фактически весь корпус знаний, существовавший в то время: физику (науку о движении), метафизику (первую философию, учение об основных принципах бытия), этику и политику (учение о государстве).

а использовать кратчайшую из возможных записей описываемой ситуации! Это и называется принципом минимальной длины описания. Им часто пользуются в статистическом машинном обучении.

§ 5.11. Заключение

В этой главе мы смогли поставить основную задачу всего машинного обучения — *поиск максимальной апостериорной гипотезы.* Эта задача появляется в разных видах, под разными масками, порой она выглядит как задача аппроксимации, порой — как задача классификации, порой ещё как-нибудь. Но основная задача в машинном обучении всегда одна: есть множество гипотез, есть некоторые (зачастую скрытые) распределения вероятностей, есть неизвестные нам параметры этих распределений, и требуется найти гипотезу, которая является в имеющихся предположениях наиболее вероятной при условии имеющихся данных.

Глава 6

Алгоритмы кластеризации

Всё истинное знание, которым мы обладаем, зависит от методов, которыми мы отличаем похожее от непохожего. Чем больше естественных различий содержат эти методы, тем яснее становится наша идея вещей. Чем более многочисленны объекты, которые занимают наше внимание, тем сложнее и тем более необходимым становится создание такого метода. Мы не должны ни объединять одним родом лошадь и свинью, хотя оба вида непарнокопытны, ни различать в разных родах козла, лося и северного оленя, хотя форма их рогов различается. Мы должны, таким образом, внимательным и усердным наблюдением определить границы родов, поскольку их нельзя определить *a priori*. Это — главная задача, наиважнейшая работа, ибо если перепутаны роды, запутается и всё остальное.

Genera Plantarum
Карл Линней

После долгого и всестороннего анализа проблемы, консультаций с Кристофером Робином и чтения подшивок журнала «Пчеловодство» за последние тридцать лет Винни-Пух всё-таки завёл себе правильных пчёл. Они делают правильный мёд, живут в правильных ульях неподалёку от домика медвежонка и почти не кусаются. В конце концов даже скептически настроенный Кролик был вынужден признать, что пчеловод из Винни-Пуха получился весьма приличный.

Однажды, тёплым летним днём, Пух проснулся от угрожающего жужжания. Выбежав из дома, Винни увидел картину, которая запомнилась ему на всю жизнь: из его ульев вылетели сразу три новых роя! Пчёлы бешено кружились, и медвежонок подумал, что, пожалуй, его пчёлы сегодня куда больше похожи на тучки, чем он сам.

Прибежавший на шум Кролик тут же перевёл разговор в практическую плоскость:

— Пух! Скорее! Рои надо ловить!

— Но Кролик, как же я их поймаю? Я даже не могу отличить, какие пчёлы в одном рое, а какие — в другом... их тут слишком много.

И Пух с Кроликом занялись тем, что стали показывать лапами на пчёл и вместе определять, из какого роя вылетела эта пчёлка. Правда, вскоре выяснилось, что они очень редко показывали на одну и ту же пчелу. А потом выяснилось, что пчёлы летают так быстро, что опилки в голове у медвежонка не могут за ними угнаться, и Пух не может запомнить, каких пчёлок они уже обсудили. Пух и Кролик всё говорили и говорили, так долго, что незаметно наступил вечер, все три роя один за другим улетели, и на пасеке воцарилась полная тишина...

§ 6.1. Введение

Наверняка читатель уже слышал слово «кластеризация» и имеет визуальное представление о задаче кластеризации: частично сгруппированные точки на плоскости (или в пространстве более высокой размерности, в зависимости от воображения читателя) постепенно разбиваются на близкорасположенные группы, как на рис. 6.1. Картинка красивая и очень близкая к истине; кластеризация именно в том м заключается, чтобы по

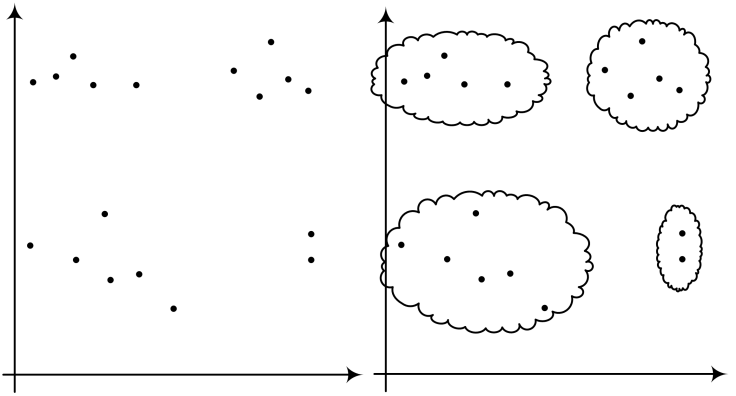


Рис. 6.1. Кластеризация

набору точек в каком-то пространстве (не обязательно евклидовом) разбить их на «близкие» группы.

Исторически одной из первых возникших в естественных науках задач кластеризации стала задача, описанная в эпиграфе к этой главе: как разделить различные виды животных на роды (мельчайшие единицы — виды — выделить проще, и Линней считает, что виды уже определены)? У животных масса параметров, и основной задачей классификаторов было выделить признаки (как пишет Линней, «естественные различия»), по которым их различать, то есть отделить *существенные* различия от *несущественных*. В математических задачах кластеризации функция различия обычно одна, но суть та же: найти некую внутреннюю структуру в данных, некую форму классификации, по которой можно различать разные «роды» данных.

Для чего нужно решать задачи кластеризации в «народном хозяйстве»? Во-первых, они лежат в основе анализа данных (data mining) в самом широком смысле [36,59,60,62,164,167]. Человечество всегда очень любило с истинно плюшкинским прилежанием накапливать огромные объёмы самых разных данных, которые некому было анализировать. Но особенно актуальным это стало в последнее десятилетие, со всеобщей компьютеризацией и развитием интернета. Каждая уважающая себя компания сейчас хранит огромные объёмы истории своей деятельности

(например, у каждого супермаркета хранится история всех покупок, которые в нём когда-либо делали), а интернет предоставляет гигантские массивы данных для самых разных, но неизменно полезных задач. Например, задача разделения документов на жанры и тематики, из чего могут получаться каталоги, автоматически обновляющиеся ленты новостей и многое другое, — это в точности задача кластеризации [141].

Во-вторых, кластеризация — важная часть задачи распознавания образов. Например, если заданное изображение можно разделить на несколько областей-кластеров, это значит, что тут же появляются определённые границы и формы этих областей, а также их отличительные особенности. Вся эта информация может использоваться для распознавания [17, 18, 41, 45, 157]. Сами изображения тоже можно кластеризовать и искать среди них похожие или выделять общие признаки.

Третья весьма популярная область, где без кластеризации не обойтись — биоинформатика. Биоинформатика решает задачи, возникающие при анализе генетической и молекулярной информации, например, при анализе длинных последовательностей атомов в белках или не менее длинных последовательностей генов. В частности, в биоинформатике часто возникает задача разбиения огромных множеств генов на кластеры. В транскриптомике (разделе биологии, изучающем транскриптомы — наборы информационных РНК в клетке) результаты такой кластеризации зачастую содержат связанные друг с другом гены, кодирующие функционально схожие белки; их выделение и отделение от других позволяет решать задачу аннотации генома. А в анализе последовательностей генов (sequence analysis) обнаруженные кластеры генов помогают выделить семейства (комплексы) генов, отвечающие за то или иное биологическое свойство и встречающиеся у нескольких видов организмов. Такие семейства — один из важнейших результатов биоинформатики [9, 81, 88, 114, 129, 161].

В качестве ещё одного примера упомянем маркетологические исследования, где кластеризация помогает разделить данные на группы и таким образом сегментировать рынок, а также анализ социальных сетей, где кластеризация может распознавать сообщества людей, схожие в стереотипах поведения или в

способах принятия решений (что, впрочем, применяться будет наверняка во всё тех же маркетологических целях) [13, 24, 116, 160]. Используется кластеризация и в медицине, и в химии, и в геологии... думаем, читатель легко сможет продолжить этот список примеров.

Есть, конечно, и подвох, который отличает обычно возникающие на практике задачи кластеризации от общепринятой картинки, нарисованной в первых строках этого параграфа. Красивые картинки получаются в размерности два или три, (да-да, «камерный оркестр», тривиальный случай, $k = 3$), а размерность реальных задач гораздо выше, она измеряется сотнями и тысячами. Поэтому сложность алгоритмов кластеризации следует оценивать относительно сразу двух параметров: и числа точек, и размерности пространства; и искать разумный компромисс в зависимости от стоящей перед вами задачи (размерность обычно несложно оценить заранее).

В этой главе мы лишь немного коснёмся основных методов кластеризации. Кроме книг по анализу данных и машинному обучению, которые мы здесь уже неоднократно цитировали, порекомендуем читателю и обзорные источники, целиком посвящённые кластеризации [12, 75, 93, 165].

§ 6.2. Постановка задачи и виды кластеризации

Кластеризация — типичная задача статистического анализа: задача классификации объектов одной природы в несколько групп так, чтобы объекты в одной группе обладали одним и тем же свойством. Под «свойством» здесь обычно понимается близость друг к другу относительно выбранной метрики.

Давайте попробуем сказать то же самое чуть более формально. Предположим, что дан набор тестовых примеров

$$X = \{x_1, \ldots, x_n\}$$

и функция расстояния между примерами $\rho : X \times X \to \mathbb{R}$.

Требуется разбить X на непересекающиеся подмножества, которые, собственно, и называются *кластерами*, так, чтобы каждое подмножество состояло из *близких* объектов, а объекты разных подмножеств *существенно* различались. Определить, что

такое «близкие объекты» и что такое «существенно различаться» — это в каком-то смысле и есть задача кластеризации; то, насколько близки должны быть элементы X, чтобы считаться «близкими», определяется тем, попадают ли они в один кластер.

Пример 6.1. Расстояние tf-idf

Рассмотрим в качестве примера задачу о том, как определить расстояние между двумя текстовыми документами. Это задача первостепенной важности для интеллектуального анализа текстов (text mining), и расстояние tf-idf, которое мы сейчас опишем, активно применяется на практике [15, 42, 73, 79, 91, 141].

Текст, конечно, сам по себе плохая модель, в нём много лишнего. Его нужно сперва подготовить к анализу, выделив из него главное. Обычно документ представляется в виде вектора из n термов с некоторыми весами; разные подходы к анализу текстов различаются в том, что такое терм и как определять веса.

Обычно термы — это слова, встречающиеся в документе; документ превращается при этом в неупорядоченный набор слов (bag of words). Практика показывает, что использовать более сложные структуры в качестве термов (например, синтаксически или статистически выделенные фразы) нецелесообразно — либо нужно будет слишком много текстов, и толком обучиться не получится, либо просто результат не будет существенно лучше, чем при bag-of-words подходе.

Для определения весов обычно используют два основных подхода: либо бинарный атрибут со значениями 0–1 (есть слово или нет слова), либо весовую функцию, меру tfidf (term frequency — inverse document frequency). Её-то мы сейчас и рассмотрим.

Эта мера была предложена в начале 1970-х годов [145] и с тех пор активно используется в анализе текстовой информации и information retrieval [136, 137]. Название подразумевает, что мера tfidf состоит из двух других: tf (частота терма, term frequency) и idf (обратная частота терма в документах, inverse document frequency).

Частота терма — это доля числа появлений этого терма по отношению к размеру всего документа. Математически говоря,

$$\mathrm{tf}(t_k, d_j) = \frac{\#(t_k, d_j)}{\sum_k \#(t_k, d_j)},$$

где $\#(t_k, d_j)$ — число, показывающее, сколько раз терм t_k встречается в документе d_j.

Обратная частота терма idf показывает, насколько терм вообще важен, насколько он характерен для данного массива текстов. Ясно,

что чем реже встречается терм в имеющемся массиве, тем он характернее.[1] Соответственно,

$$\mathrm{idf}(t_k, d_j) = \log \frac{|D|}{\#_D t_k},$$

где D — имеющийся набор данных, а $\#_D t_k$ — количество документов из D, в которых хотя бы однажды встречается t_k.

Теперь легко объединить всё это в единую меру. Мера tfidf для терма t_k и документа d_j в массиве D равна

$$\mathrm{tfidf}(t_k,d_j) = \mathrm{tf}(t_k, d_j)\mathrm{idf}(t_k, d_j) = \frac{\#(t_k,d_j)}{\sum_k \#(t_k,d_j)} \log \frac{|D|}{\#_D t_k}.$$

Вектор весов ещё можно потом нормализовать:

$$w_{kj} = \frac{\mathrm{tfidf}(t_k,d_j)}{\sqrt{\sum_{s=1}^{r}(\mathrm{tfidf}(t_k,d_j))^2}}.$$

Однако это ещё не всё. Мы сумели выразить вес каждого терма в каждом документе. Теперь документ можно представить в виде вектора, размерность которого равна количеству термов (здесь очень важно сохранять словарь не слишком большим за счёт удачного выбора термов — но это предмет совсем другой науки, feature selection, и в неё мы сейчас не будем углубляться). Но как же выразить расстояние между документами? Для этого обычно употребляется не простое декартово расстояние, а *угол* между векторами (предлагаем читателю представить себе получившееся пространство и подумать, почему угол — это более правильная мера в данном случае, чем простое расстояние). Получается так называемая *косинусоидальная мера похожести* (cosine similarity measure):

$$\theta(d_1, d_2) = \arccos \frac{d_1 \cdot d_2}{\|d_1\|\|d_2\|}.$$

Различают несколько видов кластеризации. Во-первых, алгоритмы кластеризации делятся на *иерархические* и *неиерархические*. При иерархической кластеризации алгоритм последовательно строит кластеры из уже найденных кластеров, а при

[1] Тут не нужно доходить до абсурда: слова, которые встречаются ровно один раз на весь массив документов, скорее всего, тоже не показательны. Но в процессе предобработки обычно слишком редкие (как и слишком частые — служебные слова, местоимения) слова из документов удаляются, поскольку классификации они могут только помешать.

неиерархической пытается распознать всю структуру сразу или строит кластеры один за другим, не разделяя и не соединяя уже выделенные кластеры.

Иерархическая кластеризация делится (не слишком содержательно) на два типа. Она бывает *агломеративной* (объединительной), когда алгоритм начинает с индивидуальных элементов (кластеров из одного элемента), а затем последовательно объединяет их, получая требуемую структуру, и *разделительной*, когда алгоритм начинает с одного кластера, содержащего все точки, а потом последовательно делит его на части.

Неиерархические методы объединены тем, что все они, как правило, стремятся оптимизировать некую целевую функцию, которая описывает качество кластеризации. Мы рассмотрим алгоритмы, основанные на методах теории графов, алгоритм EM, имеющий вероятностную природу, его частный (но часто применяющийся) случай — алгоритм k-средних, а также нечёткие алгоритмы.

§ 6.3. Иерархическая кластеризация

Начнём с одного из простейших методов — агломеративной кластеризации. Предположим, как это обычно бывает, что нам нужно кластеризовать точки $x_1, x_2, \ldots, x_n$ в некотором метрическом пространстве с метрикой ρ.

Идея алгоритма в том, что на первом шаге мы считаем каждую точку отдельным кластером. Затем ближайшие точки объединяем и далее относимся к ним как к единому кластеру. При итерации этого процесса получается дерево, в листьях которого — отдельные точки, а в корне — кластер, содержащий все точки вообще. Из такого дерева кластеров можно выбрать кластеризацию с требуемой степенью детализации; обычно просто выбирают то или иное максимальное расстояние и считают, что если кластеры находятся на этом расстоянии, то объединять их уже не нужно. Схема алгоритма показана на рис. 6.2.

Вроде бы всё просто и понятно. Но остаётся вопрос: как подсчитывать расстояние между кластерами, если задана функция ρ, которая подсчитывает расстояние между отдельными точками?

`HierarchyCluster`($X = \{x_1, \ldots, x_n\}$):

1. Инициализируем $C = X$, $G = X$.
2. Пока в C больше одного элемента:
   а) Выбираем два элемента C c_1 и c_2, расстояние между которыми минимально.
   б) Добавляем в G вершину c_1c_2, соединяем её с вершинами c_1 и c_2.
   в) $C := C \cup \{c_1c_2\} \setminus \{c_1, c_2\}$.
3. Выдаём G.

Рис. 6.2. Агломеративная кластеризация

Здесь проявляется разница между двумя основными подходами к иерархической кластеризации: single-link и complete-link алгоритмами (насколько нам известно, переводы этих терминов в русскоязычной литературе ещё не устоялись, и их обычно оставляют в английском написании).

Single-link алгоритмы в качестве расстояния между кластерами берут *минимум* из возможных расстояний между парами объектов, находящихся в кластерах. А complete-link алгоритмы подсчитывают *максимум* из этих расстояний.

Поэтому при использовании single-link алгоритмов кластеры могут разрастаться, «вытягивая щупальца», постепенно захватывая точки, которые находятся, может быть, и далеко от основной массы точек кластера, но связаны с ним «путём» из близкорасположенных точек. А кластеры у complete-link алгоритма будут оставаться компактными (пока им не придётся объединяться с далёкими точками, разумеется), стремиться принимать сферическую форму и разрастаться во все стороны одновременно. На рис. 6.3 показано, что в одной и той же ситуации и с одним и тем же порогом максимального размера кластера single-link алгоритм, скорее всего, объединит в один кластер больше точек, образующих этот своеобразный «Млечный Путь».

§ 6.4. Кластеризация методами теории графов

Кластеризация в метрическом пространстве естественным образом подталкивает к использованию теории графов. Точки

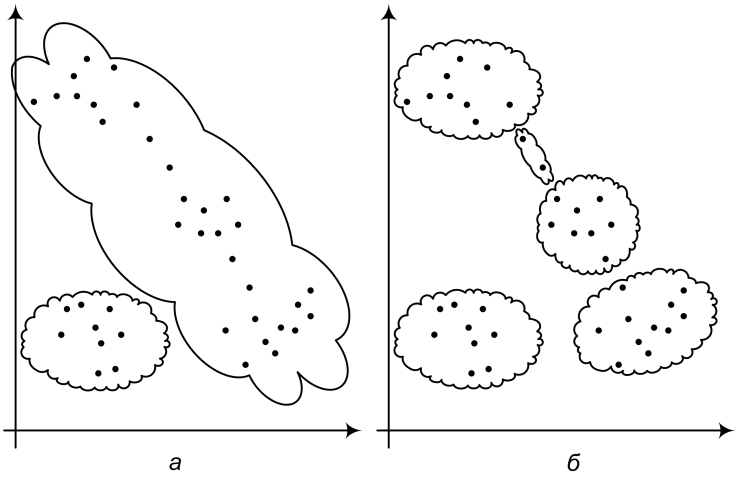


Рис. 6.3. Виды иерархической кластеризации: *а* — single-link; *б* — complete-link

в пространстве с расстояниями между ними можно рассматривать как полный граф с весами на рёбрах, равными расстоянию между точками, соединёнными данным ребром.

Легко построить тривиальный алгоритм кластеризации: для этого нужно выбрать порог расстояния r и выбросить все рёбра длиннее r. Компоненты связности полученного графа и станут кластерами.

Более изощрённый алгоритм кластеризации, основанный на идеях теории графов, использует понятие минимального остовного дерева.

Определение 6.1. *Остовное дерево* (spanning tree) графа $G = \langle V,E\rangle$ — это связный подграф G, содержащий все вершины G и являющийся деревом. *Минимальное остовное дерево* (minimal spanning tree) графа $G = \langle V,E\rangle$ с заданными на рёбрах весами $w : E \to \mathbb{R}$ — это такое остовное дерево T, что его суммарный вес не превосходит суммарного веса любого другого остовного дерева T':

$$\forall T' \quad \sum_{e\in T'} w(e) \geqslant \sum_{e\in T} w(e).$$

Чтобы использовать это понятие для кластеризации, нужно сначала построить минимальное остовное дерево, а потом выкидывать из него рёбра максимального веса до тех пор, пока не получится нужное число кластеров. Сколько рёбер выбросим, столько кластеров получим (т.к. рёбра мы выбрасываем из дерева, и каждое удалённое ребро будет добавлять новую компоненту связности).

Таким образом, единственная разница между разными алгоритмами здесь будет заключаться в методе поиска минимального остовного дерева. Для решения этой задачи обычно применяются два широко известных алгоритма. Простейший из них — алгоритм Краскала (Kruskal): на каждом шаге выбираем ребро с минимальным весом, если оно соединяет два дерева, добавляем его в остовное дерево, если нет, пропускаем. Схема этого алгоритма приведена на рис. 6.4.

Более сложный алгоритм — алгоритм Борувки (Bůruvka) — изображён на рис. 6.5. Сам Отакар Борувка опубликовал его в 1926 году для решения задачи связности электрических сетей [21], и про него все, конечно, благополучно забыли. В 1938 году его переоткрыл Густав Шоке [32], в 1951 — четыре польских автора, одним из которых был Ян Лукасевич, но и это не помогло. Окончательно его ввёл в употребление (вновь независимо придумав) уже в шестидесятые годы Соллин, поэтому в литературе его иногда называют Sollin's algorithm. Алгоритм Борувки иногда кажется сложным для понимания, но идея его на самом деле проста: можно строить минимальное остовное дерево, начав с одной вершины и добавляя рёбра минимального веса, пока не покроем весь граф (такой алгоритм тоже существует и называется *алгоритмом Прима* — Prim's algorithm). А теперь будем делать то же самое, но во всех вершинах одновременно, то есть «распараллелим» этот процесс. Получится алгоритм Борувки.

И алгоритм Краскала, и алгоритм Борувки работают за время $O(|E|\log|V|)$, где E — множество рёбер графа, а V — множество его вершин. Однако для того чтобы алгоритм Краскала работал так эффективно, нужно использовать особые, достаточно сложные структуры данных, ведь если использовать тривиальное тестирование на связность, оно будет на каждом шаге

`Kruskal`$(G = \langle V, E\rangle, w : E \to \mathbb{R})$:

1. Отсортировать рёбра G по возрастанию веса, инициализировать подграф $S \subseteq G$, $S := \emptyset$.
2. Для каждого ребра e в порядке возрастания веса:
   а) если конечные точки e ещё не связаны в S, добавить e в S.

Рис. 6.4. Алгоритм Краскала

`Boruvka`$(G = \langle V, E\rangle, w : E \to \mathbb{R})$:

1. Инициализируем список из n деревьев L, в каждом дереве по одной вершине.
2. Пока в L больше одного дерева:
   а) для каждого $T \in L$ найти ребро минимального веса, соединяющее T с $G \setminus T$;
   б) добавить все эти рёбра к минимальному остовному дереву;
   в) объединить пары пересекающихся деревьев в L (размер L при этом уменьшается вдвое).

Рис. 6.5. Алгоритм Борувки

занимать $O(|V|)$, и в сумме получится $O(|E||V|)$. А алгоритм Борувки сложных структур данных не требует, поэтому на самом деле именно он является самым простым алгоритмом для эффективной реализации. Правда, эти алгоритмы — не рекордсмены по теоретической сложности: самый эффективный из ныне известных алгоритмов поиска минимального остовного дерева работает за время $O(|E|\alpha(|V|))$, где α — обратная к функции Аккермана [30]! Но для практической реализации обычно разница между $\alpha(|V|)$ и $\log V$ совершенно не принципиальна, тем более что для алгоритма из [30] нужно сначала отсортировать рёбра.

Мы воспользуемся тем, что алгоритм Краскала в его тривиальном виде достаточно прост, и приведём здесь реализацию кластеризации минимальным остовным деревом на весьма популярном и очень изящном функциональном языке программирования Haskell. Язык был назван в честь известного логика

Хаскелла Карри[1]. Сначала его так и хотели назвать Curry, но потом решили всё-таки не делать язык предметом постоянных насмешек; впрочем, всё равно не удержались и выпустили первый релиз языка 1 апреля 1990 года [70]. В качестве учебников языка мы можем порекомендовать [37, 68, 72].

Листинг 6.1. Кластеризация остовным деревом на Haskell

```
import System.Environment
import Control.Monad
import Data.Array
import Data.List
import Data.Tuple
import Data.Graph.Inductive.Graph
import Data.Graph.Inductive.Tree
import Data.Graph.Inductive.Query.DFS

type Point = (Double, Double)
type PtGraph = Gr Point Double

kruskal :: PtGraph -> [LEdge Double]
kruskal g = kruskal' g sorte c
    where
        sorte = sortBy(\c@(la,lb,lw) d@(ra,rb,rw) ->
          compare lw rw) $ labEdges g
        c = array (1, n) $ zip [1..n] [1..n]
        n = length $ nodes g

kruskal' :: PtGraph -> [LEdge Double] ->
  Array Node Node -> [LEdge Double]
kruskal' _ [] _ = []
kruskal' g (x@(f, s, w):tail) c
    | (c ! f) == (c ! s) = kruskal' g tail c
    | otherwise = x:kruskal' g tail nc
    where
```

[1] Хаскелл Брукс Карри (Haskell Brooks Curry, 1900–1982) — американский логик и математик. Он наиболее известен как автор *парадокса Карри*: рассмотрим высказывание «Если это высказывание верно, то я папа римский». Предположим, что оно верно. Тогда, поскольку оно само так утверждает, я — папа римский. Значит, из исходного высказывания следует, что я папа римский; обозначив высказывание за $X \to Y$, получим $(X \to Y) \to Y$. Теперь применим правило сокращения и получим $X \to Y$; иначе говоря, в результате получили, что исходное высказывание верно безо всяких уже условий. А из него следует Y. Иначе говоря, мы только что совершенно честно логически доказали, что я — папа римский. Этот парадокс относится к классу «self-referential paradoxes», как и, например, парадокс Эпименида: критянин Эпименид говаривал, что все критяне лжецы.

```
        n = length $ nodes g
        nc = array(1,n) $ zip [1..n] $
          map(\a -> if a==fc then sc else a) $ elems c
        fc = c ! f
        sc = c ! s

clusters' :: PtGraph -> Int -> [[Node]]
clusters' g count = components $ gr
    where
    gr :: PtGraph
    gr = mkGraph (labNodes $ g) selEdges
        selEdges = drop (count - 1) $ reverse $ kruskal g

dist :: Point -> Point -> Double
dist a b = sqrt((fst a - fst b)*(fst a - fst b)+
                (snd a - snd b)*(snd a - snd b))

clusters :: [Point] -> Int -> [[Point]]
clusters pts count = map (\a -> map (\b -> point ! b) a) $
  clusters' gr count
    where
        gr :: PtGraph
        gr = mkGraph (zip [1..n] pts) edg
        point = array (1, n) $ zip [1..n] pts
        edg = [(i, j, dist (point ! i) (point ! j)) |
               i <- [1..n], j <- [1..n] ]
        n = length pts

main = do
    filename:[] <- getArgs
    w <- liftM words $ readFile filename

    let n = read $ head w
        points = getPoints $ map read $ tail w
        getPoints (a:b:r) = (a, b) : getPoints r
        getPoints [] = []

    print $ clusters points n
```

§ 6.5. Алгоритм EM

Алгоритм EM — это скорее подход, метод, целый набор алгоритмов для решения самых разнообразных задач, связанных со скрытыми переменными. Мы начнём знакомство с ним с самого

прямолинейного случая: попробуем получить (хотя бы асимптотически) оценку θ на основе неполного свидетельства (свидетельства с пропусками). Этот подход также предполагает, что пропуски в данных не зависят от наблюдавшихся значений.

Аббревиатура EM означает «expectation–maximization». Это значит, что одна итерация алгоритма EM во всех его модификациях состоит из двух шагов: E-шаг (expectation step), на котором вычисляется математическое ожидание скрытых переменных в предположении текущей гипотезы, и M-шаг (maximization step), на котором вычисляется новая гипотеза. Иначе говоря, основная суть алгоритма в том, чтобы модифицировать гипотезу, основываясь на ней самой! Несмотря на кажущуюся нелогичность такого поведения, легко увидеть, что на самом деле на M-шаге по делу используются тестовые данные, и новая гипотеза уже лучше им отвечает. Эта схема довольно давно неявно применялась для различных задач статистики, но впервые была явно выписана и исследована лишь в 1977 году [38].

Алгоритм EM в данном случае возвращает точечную оценку $\hat{\theta}$ для параметра θ. Эта оценка может быть получена либо методом максимального правдоподобия (ML) — максимизируя $p(e|\hat{\theta})$, либо методом максимальной апостериорной вероятности (MAP) — максимизируя $p(\hat{\theta}|e)$.

Перед началом работы алгоритма нужно выбрать точность ϵ, которую требуется достигнуть в вычислении $\hat{\theta}$.

Первый содержательный шаг цикла называется шагом математического ожидания, поскольку он обычно реализуется с помощью вычисления математического ожидания достаточной статистики для пропущенного e^*, нежели чем самого распределения вероятностей. Достаточная статистика — это статистика, которая «суммирует» набор данных и которая сама по себе содержит всю информацию, необходимую для обсуждаемого вывода. То есть достаточной статистики хватает для описания выборки, из неё можно получить всю релевантную информацию о распределении вероятностей, которую из выборки вообще можно получить. Например, μ и σ — достаточные статистики для нормального распределения.

Таким образом, s — достаточная статистика для θ относительно e^*, если и только если θ не зависит от e^* при известной

1. Присвоить $\hat{\theta}$ максимальное допустимое значение; $\hat{\theta}' := \mathrm{MAXINT}$.
2. Пока $|\hat{\theta}' - \hat{\theta}| > \epsilon$:
   а) $\hat{\theta} := \hat{\theta}'$ (за исключением первой итерации).
   б) Шаг матожидания. Вычислить распределение над пропущенными значениями:
   $$p(e^*|e,\hat{\theta}) = \frac{p(e|e^*,\hat{\theta})p(e^*|\hat{\theta})}{\sum_{e^*} p(e|e^*,\hat{\theta})p(e^*|\hat{\theta})}$$
   в) Шаг максимизации. Вычислить новую оценку $\hat{\theta}'$ согласно либо ML, либо MAP при данном $p(e^*|e,\hat{\theta})$.

Рис. 6.6. Максимизация математического ожидания — EM

s, то есть $(\theta \perp\!\!\!\perp e^*|s)$. При такой заданной статистике s второй шаг её использует, чтобы максимизировать значение обучаемого параметра (то есть параметра, оценка которого вычисляется).

Алгоритм EM обычно быстро сходится к наилучшей точечной оценке параметра; однако это — наилучшая оценка лишь *локальльно*. Она может не оказаться наилучшей *глобальльно*. Другой недостаток: мы всё же получаем лишь точечную оценку θ, а не все вероятностное распределение.

Теперь мы представим алгоритм EM в его двух формах: поиск максимального правдоподобия (ML) и поиск максимальной апостериорной вероятности (MAP).

Целью первой формы является ML-оценка для θ при данном неполном e. Как водится, нужно выбрать точность ϵ, которую требуется достигнуть в вычислении $\hat{\theta}$. А затем — как на рис. 6.7. Счётчик N_{ijk} хранит число реализаций возможных совместных означиваний X_i и $\mathrm{pa}(X_i)$, занумерованные индексами k для X_i и j для $\mathrm{pa}(X_i)$. Матожидания этих счётчиков совместно обеспечивают достаточную статистику для e^*. Для каждого отдельного счётчика его матожидание подсчитывается суммированием вероятностей из правой части равенства. Поскольку на этом шаге мы имеем заранее оценённый параметр $\hat{\theta}$, мы можем вычислить

1. Присвоить $\hat{\theta}$ максимальное допустимое значение: $\hat{\theta}' := \mathrm{MAXINT}$.
2. Пока $|\hat{\theta}' - \hat{\theta}| > \epsilon$:
   а) $\hat{\theta} := \hat{\theta}'$ (за исключением первой итерации);
   б) вычислить матожидание достаточной статистики для e^*:
$$E_{p(X|e,\hat{\theta})} N_{ijk} = \sum_{l=1}^{l=N} p(X_{ik}, pa(X_{ij}) | Y_l, \hat{\theta});$$
   в) максимизировать $p(e^*|\hat{\theta}')$ с использованием
$$\hat{\theta}'_{ijk} = \frac{E_{p(X|e,\hat{\theta})} N_{ijk}}{\sum_{k'} E_{p(X|e,\hat{\theta})} N_{ijk'}}.$$

Рис. 6.7. Алгоритм ML/EM

правую часть. А на шаге максимизации нужно использовать математические ожидания статистик как их самих. Аналогичная процедура на рис. 6.8 приводит к максимизации $p(\hat{\theta}|e)$.

Но EM — это гораздо больше, чем просто ценная максимизация. Давайте рассмотрим один из простейших примеров чуть менее тривиального применения алгоритма EM; этот пример приведёт в Главе 6 к массе эффективных алгоритмов кластеризации. Пусть случайная переменная x сэмплируется из суммы двух нормальных распределений. Дисперсии даны (и они у обоих распределений одинаковые), нужно найти только средние μ_1, μ_2.

По начальным данным нельзя понять, какие x_i были порождены каким распределением; таким образом возникает классический пример скрытых переменных. Один тестовый пример можно полностью описать как тройку

$$\langle x_i, z_{i1}, z_{i2} \rangle,$$

где $z_{ij} = 1$ тогда и только тогда, когда x_i был сгенерирован j-м распределением.[1]

[1] Здесь можно было бы обойтись одной переменной, ведь $z_{i2} = 1 - z_{i1}$. Мы вводим две лишь для удобства и единообразия.

1. Присвоить $\hat{\theta}$ максимальное допустимое значение: $\hat{\theta}' := \mathrm{MAXINT}$.
2. Пока $|\hat{\theta}' - \hat{\theta}| > \epsilon$:
   а) $\hat{\theta} := \hat{\theta}'$ (за исключением первой итерации);
   б) вычилить матожидание достаточной статистики для e^*:
   $$\mathrm{E}_{p(X|e,\hat{\theta})} N_{ijk} = \sum_{l=1}^{l=N} p(X_{ik}, \mathrm{pa}(X_{ij}) | Y_l, \hat{\theta});$$
   в) максимизировать $p(\hat{\theta}'|e^*)$ с использованием
   $$\hat{\theta}'_{ijk} = \frac{\alpha_{ijk} + \mathrm{E}_{p(X|e,\hat{\theta})} N_{ijk}}{\sum_{k'} \alpha_{ijk'} + \mathrm{E}_{p(X|e,\hat{\theta})} N_{ijk'}},$$
   где α_{ijk} — параметр распределения Дирихле.

Рис. 6.8. Алгоритм MAP/EM

Переменные z_{ij} формализуют неопределённость в условии; если раньше она выражалась не слишком математическим «мы не знаем, какое распределение породило x_i», то теперь мы переформулировали это как «мы не знаем значения переменных z_{ij}». Разница для удобства дальнейших рассуждений принципиальнейшая.

Разберём суть алгоритма EM на примере с двумя нормальными распределениями. Общая схема EM в этом случае отражена на рис. 6.9. Суть E-шага в том, чтобы пересчитать значения (математических ожиданий) скрытых переменных на основе текущей гипотезы. В случае нормальных распределений и смеси двух гауссианов получается так:

$$\mathrm{E}(z_{ij}) = \frac{p(x = x_i | \mu = \mu_j)}{p(x = x_i | \mu = \mu_1) + p(x = x_i | \mu = \mu_2)} = \\ = \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{e^{-\frac{1}{2\sigma^2}(x_i - \mu_1)^2} + e^{-\frac{1}{2\sigma^2}(x_i - \mu_2)^2}}.$$

EMGauss($\{x_i\}_{i=1}^m$):

1. Сгенерировать какую-нибудь гипотезу
$$h = (\mu_1, \mu_2).$$
2. Пока не достигнут локальный максимум (пока μ_1 и μ_2 достаточно сильно изменяются):
   а) вычислить ожидание $E(z_{ij})$ в предположении текущей гипотезы (E-шаг):
   $$E(z_{ij}) := \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{e^{-\frac{1}{2\sigma^2}(x_i-\mu_1)^2} + e^{-\frac{1}{2\sigma^2}(x_i-\mu_2)^2}};$$
   б) вычислить новые значения μ_1 и μ_2, предполагая, что z_{ij} принимают значения $E(z_{ij})$ (M-шаг):
   $$\mu_j := \frac{1}{m}\sum_{i=1}^{m} E(z_{ij})x_i.$$

Рис. 6.9. Алгоритм EM для смеси двух нормальных распределений

Мы подсчитываем эти ожидания, а потом, на M-шаге, подправляем гипотезу $h = (\mu_1, \mu_2)$ в соответствии с новыми значениями $E(z_{ij})$:

$$\mu_j := \frac{1}{m}\sum_{i=1}^{m} E(z_{ij})x_i.$$

Для полноты картины приведём написанный на языке Java класс `MixSolver`, реализующий этот алгоритм. Он реализует чуть более общий вариант этого алгоритма, в котором разделяется смесь из k гауссианов, и их дисперсии могут различаться. Класс можно использовать в виде

```
double[] e = solver.solver(sigma, x);
```

где `sigma` — массив дисперсий (из длины этого массива класс понимает, сколько именно нормальных распределений смешано в тестовых данных), x — массив тестовых примеров, а получаемый в результате массив e содержит полученные алгоритмом оценки средних этих распределений. Константу `SEED_CONSTANT`

можно заменить на любое число или на случайную функцию вроде того, сколько в текущем часе прошло миллисекунд.

Листинг 6.2. EM для разделения гауссианов на Java

```
public class MixSolver {
    private static Random rnd = new Random(SEED_CONSTANT);

    private double[] init(double[] sigma, double[] x) {
        int k = sigma.length;
        int m = x.length;
        double[] r = new double[k];
        int[] t = new int[k];
        for (int i = 0; i < m; i++) {
            int j = rnd.nextInt(k);
            r[j] += x[i];
            t[j]++;
        }
        for (int i = 0; i < k; i++) {
            if (t[i] > 0) {
                r[i] /= t[i];
            } else {
                r[i] = 0;
            }
        }
        return r;
    }

    public double[] solver(double[] sigma, double[] x) {
        int k = sigma.length;
        int m = x.length;
        if (k == 0)
            return new double[] { };
        if (k == 1) {
            double e = 0;
            for (int i = 0; i < m; i++) {
                e += x[i];
            }
            e /= m;
            return new double[] { e };
        }
        double[] e = init(sigma, x);
        double[][] z = new double[m][k];
        for (int step = 0; step < 100; step++) {
            // подсчитываем z[i][j]
            for (int i = 0; i < m; i++) {
                double d = 0;
                for (int j = 0; j < k; j++) {
```

```
                d += prob(x[i], e[j], sigma[j]);
            }
            for (int j = 0; j < k; j++) {
                z[i][j] = prob(x[i],e[j],sigma[j]) / d;
            }
        }
        for (int j = 0; j < k; j++) {
            e[j] = 0;
            double cur = 0.0;
            for (int i = 0; i < m; i++) {
                e[j] = e[j] + z[i][j] * x[i];
                cur += z[i][j];
            }
            e[j] /= cur;
        }
    }
    return e;
}

/**
* Подсчитывает плотность вероятности y
* нормального распределения (e, sigma).
*/
private double prob(double v, double e, double sigma) {
    return Math.exp(-(v - e) * (v - e) /
                    2.0 / sigma / sigma);
}
}
```

§ 6.6. Кластеризация при помощи EM

В § 6.5 мы рассматривали алгоритм EM (expectation-maximization), который умеет разделять смеси несколько вероятностных распределений. В частности, мы рассмотрели алгоритм (см. рис. 6.9), который разделял смесь двух гауссианов. Не правда ли, это сильно напоминает кластеризацию: взять два распределения, набросать точек по этим распределениям, а потом разделять, какая точка какому распределению раньше принадлежала... Но как же воспользоваться алгоритмом EM для «настоящей» кластеризации?

Ещё раз напомним, что для того чтобы воспользоваться статистическим алгоритмом, нужно в первую очередь сформулировать гипотезы о распределении данных.

Первой такой гипотезой в данном случае будет *гипотеза о природе данных*: тестовые примеры появляются случайно и независимо, согласно вероятностному распределению, равному смеси распределений кластеров

$$p(x) = \sum_{c \in C} w_c p_c(x), \quad \sum_{c \in C} w_c = 1,$$

где w_c — вероятность появления объектов из кластера c, а p_c — плотность распределения кластера c. Эта гипотеза говорит о том, что за кластерами действительно стоит нечто реальное, то есть разные кластеры порождаются реально различающимися вероятностными распределениями. Формально же она позволяет сформулировать задачу кластеризации в виде задачи разложения смеси распределений, к которой мы уже умеем применять алгоритм EM.

Осталось только выяснить, какими должны быть распределения p_c. Точнее говоря, не какими они должны быть в реальности — это от нас не зависит, — а какими мы должны их предполагать для успешного решения задачи. Это сильно зависит от конкретной постановки задачи, поэтому в дальнейшем мы будем предполагать, что работаем в $\mathbb{R}^n$ — обычном n-мерном евклидовом пространстве.

В качестве неизвестных распределений часто берут сферические гауссианы, но это на самом деле не слишком гибкий вариант: кластер ведь может быть вытянут в ту или иную сторону. Мы будем рассматривать эллиптические гауссианы, с осями разной длины вдоль разных осей координат. Это добавит выразительной силы, но вместе с ней добавятся и новые параметры для обучения. Итак, *гипотеза о виде скрытых распределений* звучит следующим образом: каждый кластер c описывается d-мерной гауссовской плотностью с центром $\mu_c = \{\mu_{c1}, \ldots, \mu_{cd}\}$ и диагональной матрицей ковариаций $\Sigma_c = \mathrm{diag}(\sigma_{c1}^2, \ldots, \sigma_{c2}^2)$ (иначе говоря, у этого распределения по каждой координате своя дисперсия, и распределения по различным координатам независимы).

В этих предположениях кластеризация получается в точности эквивалентной задаче разделения смеси вероятностных распределений. Для этого мы и будем применять EM-алгоритм.

Каждый тестовый пример описывается своими координатами $(f_1(x), \ldots, f_n(x))$. Скрытые переменные в данном случае — вероятности g_{ic} того, что объект x_i принадлежит тому или иному кластеру $c \in C$.

Схема алгоритма по-прежнему состоит из E-шага и M-шага. На E-шаге по формуле Байеса вычисляются скрытые переменные g_{ic}:

$$g_{ic} = \frac{w_c p_c(x_i)}{\sum_{c' \in C} w_{c'} p_{c'}(x_i)}.$$

А на M-шаге с использованием g_{ic} уточняются параметры кластеров w, μ, σ:

$$\begin{aligned} w_c &= \frac{1}{n} \sum_{i=1}^{n} g_{ic}, \\ \mu_{cj} &= \frac{1}{n w_c} \sum_{i=1}^{n} g_{ic} f_j(x_i), \\ \sigma_{cj}^2 &= \frac{1}{n w_c} \sum_{i=1}^{n} g_{ic} \Big(f_j(x_i) - \mu_{cj} \Big)^2. \end{aligned}$$

Рис. 6.10 содержит формальное описание этого алгоритма.

Остаётся одна проблема: нужно задавать количество кластеров. А что, если оно неизвестно? Решение этого вопроса мы отложим на потом, а сейчас приведём реализацию алгоритма EM на языке программирования LISP (в § 4.10 мы обещали привести пример полноценного алгоритма кластеризации на LISP). Мы снабдили код подробными комментариями; надеемся, что заинтересованному читателю их будет достаточно (возможно, вкупе со справочником по языку), чтобы разобраться в том, как работает программа. Для понимания того, как программировать на LISP, рекомендуем обратить особое внимание на функции `fold` и `product` — это полноценные макросы, которыми LISP и славен.

Листинг 6.3. Алгоритм кластеризации EM на LISP

```
; Математические функции

; Возведение в квадрат
(defun sqr (x) (* x x))
; Сложение векторов
```

EMCluster(X,|C|):

1. Инициализировать $|C|$ кластеров; начальное приближение:
$$w_c := 1/|C|, \quad \mu_c := \text{случайный } x_i,$$
$$\sigma_{cj}^2 := \frac{1}{n|C|}\sum_{i=1}^{n}(f_j(x_i) - \mu_{cj})^2.$$
2. Пока принадлежность кластерам не перестанет изменяться:
   а) E-шаг:
   $$g_{ic} := \frac{w_c p_c(x_i)}{\sum_{c' \in C} w_{c'} p_{c'}(x_i)}.$$
   б) M-шаг:
   $$w_c = \frac{1}{n}\sum_{i=1}^{n} g_{ic}, \quad \mu_{cj} = \frac{1}{nw_c}\sum_{i=1}^{n} g_{ic} f_j(x_i),$$
   $$\sigma_{cj}^2 = \frac{1}{nw_c}\sum_{i=1}^{n} g_{ic}\,(f_j(x_i) - \mu_{cj})^2.$$
   в) Определить принадлежность x_i к кластерам:
   $$\text{clust}_i := \text{argmax}_{c \in C} g_{ic}.$$

Рис. 6.10. Алгоритм кластеризации EM

```
(defun v+ (&rest vectors) (apply 'map 'vector '+ vectors))
; Вычитание векторов
(defun v- (&rest vectors) (apply 'map 'vector '- vectors))
; Покомпонентное умножение векторов
(defun v* (&rest vectors) (apply 'map 'vector '* vectors))
; Умножение на скаляр
(defun vs* (num vec)
    (map 'vector (lambda (x) (* num x)) vec))
; Покомпонентное возведение в квадрат
(defun vsqr (a) (v* a a))
; Выбор числа с заданной вероятностью
(defun with-prob (prob item)
  (if (< (random 1.0) prob) item nil))
```

```
; Функция свёртки, позволяющая комбинировать функции.
; Например, инструкция (fold + 0 i (1 5) (* 2 i))
; подсчитает
; 0 + 2*1 + 2*2 + 2*3 + 2*4 + 2*5.
(defmacro fold (func init index (low up) &body body)
    (let* ((res (gensym)))
      `(let ((,res ,init))
         (loop for ,index from ,low to ,up do
               (setq ,res (,func ,res (progn ,@body))))
         ,res)))

; Частный случай свёртки для произведений
(defmacro product (index (low up) &body body)
  `(fold * 1 ,index (,low ,up) ,@body))

; Многомерный гауссиан с диагональной матрицей ковариаций
(defun gauss (x mean sigma2)
  (product j (0 (1- (length x)))
     (let ((xj (aref x j))
           (mj (aref mean j))
           (sj (aref sigma2 j)))
       (if (zerop sj) 1
          (/ (exp (- (/ (sqr (- xj mj))
                        (* 2 sj))))
          (sqrt (* 2 pi)) sj)))))


;;; ---------- Операции над кластерами ----------
(defun make-cluster (w mean sigma2)
  (cons w (cons mean sigma2)))
(defun w (cluster) (car cluster))
(defun mean (cluster) (cadr cluster))
(defun sigma2 (cluster) (cddr cluster))


;;; ---------- Операции над списками точек ------------
; Создаёт список точек для многомерных векторов, например:
; (make-points 3 (1 2 3) (4 5 6)) -->
;   (3 #(1 2 3) #(4 5 6))
(defun make-points (dim points)
    (cons dim
        (loop for point in points
            collect (make-array dim :initial-contents point
                    :element-type 'fixnum))))
; Выдаёт размерность списка точек
(defun dim (points) (car points))

; Выдаёт сам список
```

```
(defun content (points) (cdr points))

; Случайно выбирает count элементов из списка
; items без повторений
(defun sample (items count)
    (let ((len (length items)) (selected 0))
    (loop for item in items
        for n downfrom (length items)
        for p = (/ (- count selected) n)
        for x = (with-prob p item)
        until (= selected count)
        counting x into selected
        unless (null x)
        collect x)
    )
)

; Вычисляет вектор дисперсий списка
; точек вокруг данного центра
(defun dispersions (points center)
    (let ((len (length (content points)))
        (res (make-array (dim points) :initial-element 0
                         :element-type 'fixnum)))
    (loop for point in (content points) do
        (loop for i from 0 below (dim points) do
            (setf (aref res i) (+ (aref res i)
                (sqr (- (aref point i) (aref center i)))))
        )
    )
    (vs* (/ 1.0 (dim points) len) res)
    )
)

; Создает count кластеров из items
(defun init-clusters (items count)
    (let ((centers (sample (content items) count)))
    (loop for center in centers collecting
        (make-cluster (/ 1.0 count) center
          (dispersions items center))
    )
    )
)

; Оценивает для каждого элемента items вероятность
; попасть в тот или иной кластер; результат вида
; ((p(x1 in c1) p(x1 in c2) p(x1 in c3) ...)
;  (p(x2 in c1) p(c2 in c2) ...)
;  ...)
```

```
(defun estimate-probs (items clusters pfunc)
    (let
        ((estimate-item-probs (item)
        (let* ((probs-unnormalized
            (map 'list (lambda (cluster) (* (w cluster)
                (funcall pfunc item cluster)))
                clusters))
            (sum (apply '+ probs-unnormalized)))
            (map 'list (lambda (x)
                (if (zerop sum) 0 (/ x sum)))
                probs-unnormalized)
        )))
    (map 'list #'estimate-item-probs (content items)))
)

; Транспонирует матрицу.
; Например: ((1 a) (2 b) (3 c)) --> ((1 2 3) (a b c))
(defun transpose-list-matrix (m)
    (if (null (car m)) nil
    (let ((cars (map 'list 'car m))
        (cdrs (map 'list 'cdr m)))
    (cons cars (transpose-list-matrix cdrs))))
)

; Вычисляет новое среднее кластеры
(defun update-mean (cluster items probs)
    (let* ((len (length (content items))))
        (vs* (/ 1 len (w cluster))
        (apply 'v+ (map 'list 'vs* probs (content items))))
    )
)

; Вычисляет новое значение дисперсии кластера
(defun update-sigma2 (cluster items probs)
    (let* ((len (length (content items)))
        (center (lambda (item) (v- item (mean cluster))))
        (centered-items (map 'list center (content items))))
    (vs* (/ 1 len (w cluster))
        (apply 'v+ (map 'list 'vs* probs
            (map 'list 'vsqr centered-items)))
    ))
)



; Пересчитывает вес, среднее и квадрат дисперсии кластера
(defun update-cluster (cluster items probs)
    (let* ((len (length (content items)))
        (weight (/ (apply '+ probs) len))
```

```
        (mean (update-mean cluster items probs))
        (sigma2 (update-sigma2 cluster items probs)))
    (make-cluster weight mean sigma2))
)

; M-шаг алгоритма EM
(defun update-clusters (clusters items probs)
    (let* ((len (length items))
        (clusters-probs (transpose-list-matrix probs)))
    (map 'list (lambda (cluster cluster-probs)
        (update-cluster cluster items cluster-probs))
            clusters clusters-probs)))

; Находит индекс максимального элемента в массиве
(defun argmax (list)
    (loop with res = 0
        for x in list and i upfrom 1
        maximizing x into max
        do (if (= x max) (setq res i))
        finally (return res))
)

; По матрице вероятностей вычисляет для каждой точки,
; к какому кластеру она наиболее вероятно относится
(defun distribute-into-clusters (item-probs)
    (map 'list 'argmax item-probs))

; Читает точку из строки формата "1 2"
(defun read-point (dim line)
    (with-input-from-string (stream line)
        (loop repeat dim collect (read stream))))

; Читает файл в формате
; <размерность>
; <точка1> (в формате <коорд1> ... <коорд_dim>)
; <точка2>
; ...
; <точкаN>
(defun read-points (filename)
    (with-open-file (file filename)
    (let* ((dim (read file))
        (cluster-count (read file))
        (points (catch 'points
            (let ((points nil))
            (loop (multiple-value-bind (line is-end)
                (read-line file)
                (push (read-point dim line) points)
                (if is-end (throw 'points (nreverse points)))
```

```
            )))
        )))
    (values
        (apply #'make-points dim (list points)) cluster-count)
    ))
)

; вместо "input.txt" можно подставить другой файл
(multiple-value-bind (points cluster-count)
    (read-points "input.txt")
    (format t "~%Distribution: ~a~%"
    (let ((clusters (init-clusters points cluster-count))
        (pfunc (lambda (item cluster)
            (gauss item (mean cluster) (sigma2 cluster))))
        (prev-distribution nil))
    (catch 'result
        (loop (let* ((probs
            (estimate-probs points clusters pfunc))
            (distribution (distribute-into-clusters probs)))
        (if (equal distribution prev-distribution)
            (throw 'result distribution))
;(format t "~%Current distribution: ~A~%" distribution)
        (setq prev-distribution distribution)
        (setq clusters
            (update-clusters clusters points probs)
        )))
    )))
)
```

§ 6.7. Алгоритм k-средних

Алгоритм k-средних (k-means clustering) фактически является упрощением алгоритма EM; изобретён он был, как это часто бывает с упрощениями, раньше самого EM [69]. Разница в том, что теперь мы не подсчитываем вероятности принадлежности объекта разным кластерам, а жёстко приписываем каждый объект одному кластеру. Кроме того, в алгоритме k-средних форма кластеров не настраивается (но это с учётом первого замечания уже не так важно).

Цель алгоритма k-средних — минимизировать меру ошибки

$$E(X, C) = \sum_{i=1}^{n} \|x_i - \mu_i\|^2,$$

kMeans(X,|C|):

1. Инициализировать центры $|C|$ кластеров:
$$\mu_1, \ldots, \mu_{|C|}.$$
2. Пока принадлежность кластерам не перестанет изменяться:
   а) определить принадлежность x_i к кластерам:
   $$\text{clust}_i := \text{argmin}_{c \in C} \rho(x_i, \mu_c);$$
   б) определить новое положение центров:
   $$\mu_c := \frac{\sum_{\text{clust}_i = c} f_j(x_i)}{\sum_{\text{clust}_i = c} 1}.$$

Рис. 6.11. Алгоритм кластеризации k-средних

где μ_i — ближайший к x_i центр кластера. В этом алгоритме мы не относим точки к кластерам, а двигаем центры. Принадлежность точек при этом определяется автоматически: точка принадлежит кластеру с ближайшим к ней центром.

Общая идея алгоритма та же, что в EM: сначала инициализировать центры кластеров каким-нибудь начальным разбиением, затем классифицировать точки по ближайшему к ним центру кластера (аналог E-шага), а затем перевычислить каждый из центров (аналог M-шага). Если ничего не изменилось, остановиться, если изменилось — повторить.

Алгоритм k-средних очень часто применяется на практике; он предоставляет разумный компромисс между вычислительной сложностью и качеством работы. Однако он имеет ряд недостатков. Во-первых, как и в более общем алгоритме EM, необходимо точно знать число кластеров заранее. Во многих задачах либо уже дано это число, либо его нетрудно как-то оценить; однако встречаются ситуации, в которых само это число уже представляло бы интересный результат кластеризации. Для таких ситуаций алгоритм k-средних сам по себе, без помощи других алгоритмов кластеризации, неприменим.

Второй недостаток заключается в том, что качество результата сильно зависит от начального разбиения. Причём зависит

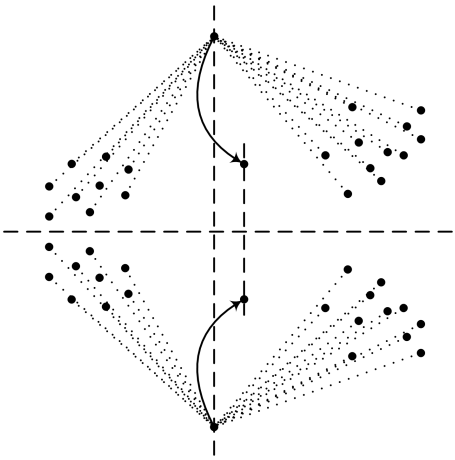


Рис. 6.12. Пример плохой работы алгоритма k-средних

даже не напрямую от его качества; не просто чем лучше начальное разбиение, тем лучше кластеризация, а просто — одно разбиение приведёт к хорошей кластеризации, другое — к плохой, в которой алгоритм застрянет в достаточно «мелком» локальном максимуме. На рис. 6.12 приведён типичный пример: точки чётко разделились на два кластера, но при этом центры кластеров на следующем шаге сдвинутся ближе к центральной линии, но останутся симметричными, а затем вовсе перестанут двигаться. «Правильные» кластеры окажутся разделёнными на две «неправильные» части — часть выше горизонтальной пунктирной линии и часть ниже неё.

Поэтому на практике часто применяют сначала другой метод кластеризации (более дешёвый вычислительно, иначе смысла в этом никакого не будет), получая число кластеров и начальное разбиение, а затем уже с этими начальными данными запускают алгоритм k-средних. Есть, конечно, и масса других способов, интересных методов эффективной реализации алгоритма k-средних, а также теоретических и практических оценок эффективности [7, 78].

Ниже мы приводим реальный пример реализации алгоритма k-средних на языке Java. Это полноценная программа, а не отдельный класс, у неё есть метод `main`, и её можно запускать.

Листинг 6.4. Алгоритм k-средних на Java

```
import java.io.*;
import java.util.StringTokenizer;

public class Solution {
    public static void main(String[] arg) {
        int n, m, k;
        BufferedReader br = new BufferedReader(
          new InputStreamReader(System.in) );
        try {
            n = Integer.parseInt(br.readLine());
            m = Integer.parseInt(br.readLine());
            k = Integer.parseInt(br.readLine());

            // читаем вход
            Point[] x = new Point[m];
            for (int i = 0; i < m; i++) {
                x[i] = new Point(n);
                StringTokenizer tok =
                  new StringTokenizer(br.readLine());
                for (int j = 0; j < n; j++) {
                    x[i].set(j,
                      Float.parseFloat(tok.nextToken()));
                }
            }

            // выполняем кластеризацию
            Point[] pp = clusterize(x, k);

            // выводим результат
            for (Point point : x) {
                int index = getClusterIndex(pp, point);
                System.out.println("(" + point.toString()+
                  " ) --> " + index);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static int getClusterIndex(
      Point[] centers, Point p) {
        float min   = p.dist(centers[0]);
        int index = 0;
        for (int i = 1; i < centers.length; i++) {
            float curDist = p.dist(centers[i]);
            if (curDist < min) {
```

```
                min = curDist;
                index = i;
            }
        }
        return index;
    }

    public static Point[] clusterize(
      Point[] points, int k) {
        Point[] centers = new Point[k];
        Point[] bbox = Point.calcBoundingBox(points);
        for (int i = 0; i < k; i++) {
            centers[i] = new Point(bbox);
        }
        boolean changed = false;
        int iterations = 0;
        while (!changed && iterations < 10000) {
            iterations++;
            changed = false;
            for (int i = 0; i < centers.length; i++) {
                Point temp =
                  new Point(points[0].x.length).clear();
                int n = 0;
                for (Point p : points) {
                    if (getClusterIndex(centers, p) == i) {
                        temp.add(p);
                        n++;
                    }
                }
                temp.scale(n > 0 ? 1.0f / n : 1.0f);
                changed = !centers[i].equals(temp);
                centers[i] = temp;
            }
        }
        for (Point p : centers) {
            System.out.println("cc " + p );
        }
        return centers;
    }
}

class Point {
    public Point(int dimension) {
        x = new float[dimension];
        for (int i = 0; i < dimension; i++) {
            x[i] = (float)Math.random() * 10;
        }
    }
```

```
public Point(Point p) {
    x = new float[p.x.length];
    for (int i = 0; i < p.x.length; i++) {
        x[i] = p.x[i];
    }

}
public Point(Point[] bbox) {
    int dimension = bbox[0].x.length;
    x = new float[dimension];
    for (int i = 0; i < dimension; i++) {
        float width = (bbox[1].x[i]-bbox[0].x[i]);
        x[i] = (float)(Math.random()*width-width/2);
    }
}
public void set(int i, float val) {
    x[i] = val;
}
public float dist(Point p) {
    float r = 0;
    for (int i = 0; i < x.length; i++) {
        r += Math.pow(x[i] - p.x[i], 2);
    }
    return (float)Math.sqrt(r);
}
public void add(Point p) {
    for (int i = 0; i < x.length; i++) {
        x[i] += p.x[i];
    }
}
public Point clear() {
    for (int i = 0; i < x.length; i++) {
        x[i] = 0;
    }
    return this;
}
public void scale(float times) {
    for (int i = 0; i < x.length; i++) {
        x[i] *= times;
    }
}
public boolean equals(Point p) {
    for (int i = 0; i < x.length; i++) {
        if (Math.abs(x[i] - p.x[i]) > 0.01) {
            return false;
        }
    }
    return true;
```

```
    }
    public boolean less(Point p) {
        for (int i = 0; i < x.length; i++) {
            if (x[i] > p.x[i]) {
                return false;
            }
        }
        return true;
    }
    public String toString() {
        String s = "";
        for (float f : x) {
            s += " " + f;
        }
        return s;
    }
    public static Point[] calcBoundingBox(Point[] points) {
        int dimension = points[0].x.length;
        Point left = new Point(points[0]);
        Point right = new Point(points[0]);
        for (int i = 1; i < dimension; i++) {
            for (int j = 0; j < points.length; j++) {
                if (left.x[i] > points[j].x[i]) {
                    left.x[i] = points[j].x[i];
                }
                if (right.x[i] > points[j].x[i]) {
                    right.x[i] = points[j].x[i];
                }
            }
        }
        Point result[] = new Point[2];
        result[0] = left;
        result[1] = right;
        return result;
    }
    float[] x;
}
```

Осталось обсудить ещё одну особенность алгоритмов EM и k-средних. И EM, и k-средние хорошо обобщаются на случай *частично обученных кластеров* (по-английски это называется semi-supervised clustering). Частично обученные кластеры возникают в ситуации, когда про часть точек уже известно, какому кластеру они принадлежат. Как это учесть в алгоритмах?

Чтобы учесть информацию о точке x_i, достаточно для EM положить скрытую переменную g_{ic} равной тому кластеру, которому нужно, с вероятностью 1, а остальным — с вероятностью 0, и не пересчитывать эту вероятность при итерациях. Для k-means нужно то же самое сделать с переменной clust_i. Оба алгоритма в таком случае правильно обрабатывают дополнительную информацию и дают разумные результаты с учётом этой заранее заданной структуры.

§ 6.8. Нечёткие алгоритмы кластеризации

Во всех вышеприведённых алгоритмах один объект принадлежал строго одному кластеру (правда, иногда мы говорили, что он принадлежит кластеру с какой-то вероятностью). Теперь же мы откажемся от этого ограничения и введём *меру принадлежности* кластеру. Тем самым мы вводим нечёткость в её самой классической форме, по Заде[1] [8, 40, 117]. Мера принадлежности — вещественное число из $[0,1]$, и точки на краю кластера «меньше принадлежат» кластеру, чем в центре.

Будем обозначать принадлежность кластеру $c \in C$ через $u_c(x)$. Меры принадлежности обычно выбирают так, чтобы

$$\forall x\ u_c(x) \geqslant 0 \quad \sum_{c \in C} u_c(x) = 1.$$

Нечёткие алгоритмы кластеризации (одним из которых является алгоритм c-средних) минимизируют ту или иную меру ошибки. Часто применяется мера

$$E(C) = \sum_{c \in C} \sum_{x \in X} u_c^m(x) \rho^2 \left(x, \text{Center}_c\right),$$

где m — некоторый вещественный параметр. Доказывать, что минимизируется именно она, мы не будем, но эта функция ошибки выглядит вполне естественной.

Алгоритм c-средних очень похож на алгоритм k-средних и, как следствие, на алгоритм EM. На E-шаге нужно определить

[1] Лотфи Заде (Lotfi Zadeh, род. 1921) — математик, профессор университета Беркли, основатель теории нечётких множеств. Родился в Азейбарджане (звали его тогда Лотфи Алескерзаде), потом жил в Иране, в 1944 году переехал в США.

cMeans(X,|C|):

1. Случайно выбрать коэффициенты $u_c(x)$ для всех $x \in X$ и $c \in C$.
2. Пока алгоритм не сойдётся:
   а) Для всех $c \in C$
   $$\mathrm{Center}_c := \frac{\sum_{x \in X} u_c(x)^m x}{\sum_{x \in X} u_c(x)^m}.$$
   б) Для всех $c \in C$ и всех $x \in X$
   $$u_c(x) := \frac{1}{\sum_{c' \in C} \left(\frac{\rho(\mathrm{Center}_c, x)}{\rho(\mathrm{Center}_{c'}, x)} \right)^{2/(m-1)}}.$$

Рис. 6.13. Алгоритм кластеризации c-средних

центр кластера. Для этого обычно рассматривают взвешенную посредством u_c сумму по всем точкам:

$$\mathrm{Center}_c = \frac{\sum_x u_c(x)^m x}{\sum_x u_c(x)^m},$$

где m — некоторый вещественный параметр. Затем, на M-шаге, меры принадлежности нужно перевзвесить относительно новых центров; результат по сути происходит из

$$u_c(x) := \frac{1}{\rho(\mathrm{Center}_c, x)},$$

но его нужно ещё немного «фаззифицировать», ввести нечёткость. Точная формула и формальное описание алгоритма отражены на рис. 6.13.

Отметим, что если $m = 2$, то перевзвешивание эквивалентно линейной нормализации коэффициентов так, чтобы их сумма была равна единице. А при $m \to 1$ всё больший и больший вес придаётся самому близкому кластеру, и алгоритм становится всё более похож на алгоритм k-средних. Получается, что алгоритм c-средних обобщает алгоритм k-средних, позволяя варьировать его поведение.

§ 6.9. Заключение

Задачи кластеризации столь часто появляются и столь многогранны, что глупо было бы и пытаться охватить все методы их решения в этой короткой главе. Мы наметили несколько подходов, в том числе самый часто встречающийся метод k-средних, а также его обобщение, алгоритм кластеризации EM, который на самом деле является одним из применений гораздо более общей схемы.

Задача кластеризации — наверное, самая важная из всех задач машинного обучения. Большинство методов машинного обучения могут решать (и решают) задачи кластеризации, а на практике к кластеризации сводятся многие задачи анализа данных (data mining). Однако в следующей главе мы рассмотрим задачу, которая к кластеризации не сводится. Мы рассмотрим обучение в ситуации, когда агент должен действовать в незнакомой ему среде, обучаясь по дороге и параметрам окружающей среды, и оптимальному для себя поведению.

Глава 7

Обучение с подкреплением

— Как хорошо, что я взял его с собой,— сказал он.— Немало медведей в такой жаркий день и не подумали бы захватить с собой то, чем можно немножко подкрепиться!..

Винни-Пух и все-все-все
Алан Александр Милн в переводе *Бориса Заходера*

Когда Винни-Пух ещё не научился как следует играть в пустяки, Кристофер Робин неосторожно пообещал: если Винни обыграет мальчика двадцать раз подряд, Кристофер Робин позволит медвежонку съесть в доме всё, что пожелают его душа и животик. Расчёт Кристофера Робина был верный: пока мальчик и медвежонок выбирали почти одинаковые случайные палочки, двадцать побед подряд были крайне маловероятны. Однако после того, как Винни принялся за дело всерьёз (см. Главу 4), Кристофер Робин быстро проиграл десять раз подряд, затем пятнадцать, а затем и двадцать. Мальчику не хотелось сердить родителей, но слово было дороже.

Так Винни-Пух осуществил свою давнюю мечту: залез в погреб. Там родители Кристофера Робина хранят множество разнообразных ёмкостей: горшочков, кастрюлек, мешочков и так далее, и тому подобное. В этих ёмкостях вполне может оказаться что-нибудь вкусненькое, чем можно как следует подкрепиться.

Света в погребе, конечно, нету. Винни может на ощупь передвигаться по погребу, подбирать и ощупывать разные типы ёмкостей, а также пробовать то, что в них находится. При этом иногда Винни везёт, и в горшочке оказывается мёд или что-нибудь не менее вкусное. Тогда, собственно, и происходит процесс *подкрепления*. А иногда не везёт, и приходится пробовать всякую гадость: горчицу, чёрный перец или — упаси Боже! — сыр.

Винни должен побыстрее научиться распознавать горшочки с вкусными вещами и пропускать горшочки с невкусными, даже не пробуя. Но при этом нельзя просто выбрать один конкретный вид горшочка, не попробовав всё: вдруг именно в таких вот кастрюльках семья Кристофера Робина хранит особо вкусный мёд урожая 1804 года (того самого, когда Наполеон I Бонапарт стал императором Французской республики).

Правда, у родителей Кристофера Робина отнюдь не всегда под рукой оказывался именно нужный горшочек: зачастую приходилось складывать во что есть. Поэтому содержимое в результате разместилось по ёмкостям с изрядной долей случайности. Так что даже если в мешочке оказался сыр, это вовсе не обязательно означает, что больше такие мешочки трогать нельзя: вдруг это просто весьма досадная случайность...

§ 7.1. Введение

До сих пор мы рассматривали задачи обучения на некотором наборе тестовых примеров. Иначе говоря, до сих пор задача ставилась так: есть набор «правильных ответов», который нужно продолжить на всё пространство.

Но разве так работает обучение в реальной жизни? Мы далеко не всегда знаем набор правильных ответов заранее, мы просто выполняем то или иное действие и получаем результат. Если программа обучается игре в шашки, человек не сможет сказать ей, какой ход единственно правильный в сложившейся ситуации. Всё, что у программы получится выяснить, — это то, выиграла она в итоге или проиграла.[1] Если робот учится ходить,

[1] Эта фраза была написана в начале 2007 года, а уже к его концу стало ясно, что человек на самом деле уже мог бы сказать программе правильный ответ — недавно построили (разумеется, не без помощи гигантского компьютерного перебора) оптимальную стратегию для игры в шашки на доске 8×8 [31]. Можно было бы просто заменить шашки на шахматы или

человек не сможет сказать ему, какой именно мускул и насколько сильно нужно напрячь в данную секунду — а просто скажет, получилось сделать шаг или нет.

Такие ситуации, близкие к реальному обучению, можно охарактеризовать единой общей моделью. В этой модели обучающийся агент взаимодействует с окружающей средой, предпринимая какие-то действия (выбирая их, конечно, из фиксированного набора); окружающая среда его каким-то образом поощряет или наказывает за эти действия, а агент продолжает их предпринимать. Единственный способ для агента понять, что он делает правильно, — следить за поощрениями от окружающей среды. Такой вид машинного обучения называется *обучением с подкреплением* (reinforcement learning).

Попробуем поставить задачу формально. Пусть на каждом шаге агент может находиться в состоянии s из некоторого множества состояний S. На каждом шаге он выбирает из имеющегося набора действий A некоторое действие a. В ответ на это окружающая среда сообщает агенту, какую награду r он за это получил и в каком состоянии s' после этого оказался.

Пример 7.1. Диалог обучающегося агента и окружающей среды

Среда: Агент, ты в состоянии 1; есть 5 возможных действий.
Агент: Делаю действие 2.
Среда: Даю тебе 2 единицы за это. Попал в состояние 5, есть 2 возможных действия.
Агент: Делаю действие 1.
Среда: Отнимаю у тебя 5 единиц. Попал в состояние 1, есть 5 возможных действий.
Агент: Делаю действие 4.
Среда: Даю тебе 14 единиц за это. Попал в состояние 3, есть 3 возможных действия...

В этом примере агент успел вернуться в состояние 1 и исследовать ранее не пробовавшуюся опцию 4 (получив за это существенную награду). То, что агент успел узнать об окружающей среде, представлено на рис. 7.1. Так и в общем случае — агент должен исследовать окружающую среду и выбирать оптимальное поведение.

го, но мы решили не упускать повод упомянуть это весьма интересное достижение.

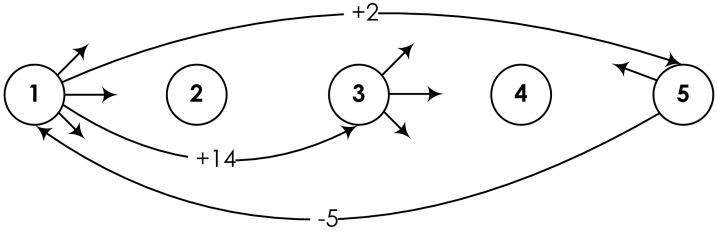


Рис. 7.1. Пример взаимодействия агента со средой

Знакомые с теорией вероятностей читатели уже, наверное, догадались, что здесь не обойдётся без марковских процессов. Действительно, мы к ним ещё вернёмся, и они будут играть важную роль в построении новых алгоритмов, но пока обратимся к тому, как сравнивать и оценивать алгоритмы обучения с подкреплением. Для более подробного анализа описанных в этой главе алгоритмов рекомендуем [77, 122, 132, 135, 163].

§ 7.2. Как оценивать поведение агента?

Мы исследуем алгоритмы, которые обучаются получать награду от окружающей среды. Разумеется, мы хотим получить алгоритмы, которые «хорошо себя ведут», то есть получают большую награду. Но то такое «хорошо»? Как оценивать поведение алгоритма в приведённых выше обстоятельствах?

На этот вопрос есть несколько ответов, выбор между которыми зависит от того, на какой срок жизни агента мы рассчитываем и на что хотим сделать упор в своих оценках.

Первая, простейшая модель — это так называемая *модель конечного горизонта* (finite horizon model). В ней качество поведения агента измеряется только по отношению к следующим h шагам, и максимизировать нужно величину

$$E\left[\sum_{t=0}^{h} r_t\right].$$

Эта модель соответствует ситуации, когда у агента ограничен срок жизни: мы знаем, что у нас всего десять попыток, и поэтому нас не интересует, успешно ли мы обучимся производить одиннадцатую.

В более естественной ситуации, однако, хотелось бы учесть все возможные шаги в будущем, потому что время жизни агента может быть не определено. Но при этом в большинстве реальных задач чем раньше мы получим награду от окружающей среды, тем лучше. Например, рубль сегодня — это гораздо лучше, чем рубль через год, ведь его за это время можно разумно вложить и приумножить[1].

Как это учесть в математической модели? Для этого достаточно придать более быстрой прибыли больший вес, а более отдалённой во времени — меньший. Так мы одновременно убьём двух зайцев, ведь за счёт введённого коэффициента γ ещё и возникающий при суммировании по бесконечной длине жизни агента ряд начнёт сходиться (мы считаем, что выплаты r_t ограничены сверху — в конце концов, у нас конечное число разных состояний и разных выплат). Итак, мы подсчитываем математическое ожидание суммы ряда

$$E\left[\sum_{t=0}^{\infty}\gamma^t r_t\right],$$

где γ — некоторая константа (в англоязычных текстах её обычно называют discount factor). Разумеется, имеет смысл только случай, когда $0 < \gamma < 1$.

Такая модель называется *моделью бесконечного горизонта* (infinite horizon model). Далее мы будем в основном пользоваться именно ею.

Третья, реже встречающаяся, модель — *модель среднего вознаграждения* (average-reward model). В ней максимизации подлежит предел ожидания среднего вознаграждения

$$\lim_{h\to\infty} E\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right].$$

Эта модель не даёт быстрой прибыли преимущества перед отложенной, поэтому на практике нечасто оказывается лучше модели бесконечного горизонта.

[1] «Нет! расчёт, умеренность и трудолюбие: вот мои три верные карты, вот что утроит, усемерит мой капитал...» — говорил пушкинский Герман. Впрок ему это не пошло, но только потому, что вмешался совсем уж человеческий фактор...

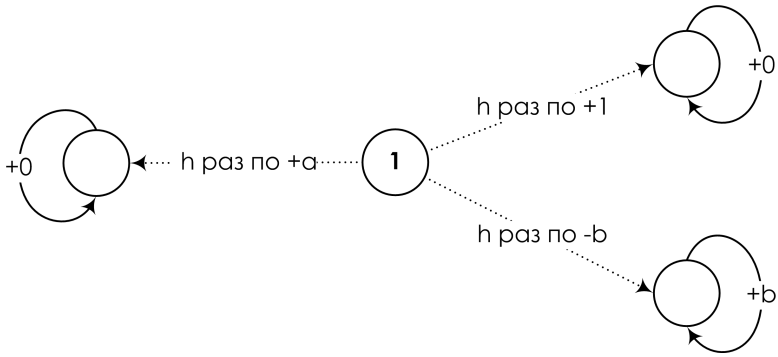


Рис. 7.2. Различия между оценками качества

Пример 7.2. Различия между моделями оценки качества

Все вышеописанные модели разные, и несложно построить граф состояний, на котором все три модели будут приводить к разным оптимальным стратегиям.

Рассмотрим граф на рис. 7.2. Пометки на рёбрах соответствуют получаемой при данном действии прибыли. Можно просчитать работу каждой из стратегий на каждой из ветвей графа переходов и вычислить вознаграждения в разных моделях. Вот что получается.

Ветвь	Конечный горизонт	Бесконечный горизонт	Среднее вознаграждение
Левая	ah	$a\frac{1-\gamma^h}{1-\gamma}$	0
Верхняя	h	$\frac{1}{1-\gamma}$	1
Нижняя	$-bh$	$\frac{b}{1-\gamma}\left(2\gamma^h-1\right)$	b

Выберем теперь подходящие значения a и b:

$$a=\frac{2-\gamma^h}{2(1-\gamma^h)},\quad b=\frac{\gamma^h}{2\gamma^h-1}.$$

При таких a и b оптимальные значения в таблице располагаются по диагонали: модель конечного горизонта выберет левую ветвь, бесконечного горизонта — верхнюю, а модель среднего вознаграждения — нижнюю.

Кроме поведения уже готового алгоритма, нужно также научиться оценивать и качество его обучения — то, насколько быстро он стремится к своему пределу.

Главное (и обычно не так уж сложно доказуемое) свойство алгоритмов обучения с подкреплением — сходимость к оптимальному (для данной модели, конечно). Это полезное свойство, но оно бинарно — или сходится, или не сходится; этот бит ничего не говорит ни о скорости сходимости, ни о том, с какими потерями мы подойдём к оптимальной стратегии.

Более разумны оценки скорости сходимости. Здесь возможны два альтернативных подхода: оценки скорости сходимости к какой-то фиксированной доле оптимальности (собственно оптимального алгоритма обучающийся агент обычно «достигает» лишь на бесконечности) и оценки качества работы алгоритма по истечении некоторого фиксированного времени. К сожалению, в обоих случаях возникают конкретные вопросы: какая нужна доля оптимальности? После какого времени нужно сравнивать качество алгоритмов? Однозначный ответ на эти вопросы найти трудно.

Третий возможный подход — минимизировать ожидаемую расплату (regret) за выбор той или иной стратегии обучения. Любая обучающаяся стратегия не может принести больший доход, чем если бы агент следовал оптимальной стратегии с самого начала. Иными словами, задачу оптимального обучения можно рассматривать как задачу минимизации разницы между общей суммой выигрыша при обучении по сравнению с применением оптимальной стратегии с самого начала. Если нам удастся подсчитать эту разницу, то можно очень просто сравнивать алгоритмы обучения — у кого разница меньше, тот и победил. Это очень хорошая мера, ведь, собственно, она и является настоящей конечной целью происходящего. Но, к сожалению, она плохо поддаётся анализу: разумные результаты об этой мере получить очень сложно.

И последнее замечание, прежде чем мы перейдём к непосредственному изучению отдельных алгоритмов. Каждый алгоритм обучения с подкреплением должен и изучать окружающую среду, и пользоваться своими знаниями, чтобы максимизировать прибыль. Возникает вопрос — как достичь оптимального соотношения? Та или иная стратегия может быть хороша, но вдруг она не оптимальная, и нужно искать другую? А, может быть, она уже идеальна, и нужно просто следовать ей, не тратя

времени на бессмысленные поиски? Эта дилемма (по-английски она звучит особенно красиво: «exploitation vs. exploration») всегда присутствует в обучении с подкреплением. Однозначного ответа на этот вопрос, конечно, нет и быть не может, но для многих частных случаев разумный компромисс можно определить и вычислить. Этим мы и будем заниматься в оставшейся части главы.

§ 7.3. Многорукие бандиты

Будем двигаться от простого к сложному. Первой остановкой на этом пути станут агенты, у которых есть только одно состояние. Формально они ничем не отличаются от введённых в § 7.1, просто $|S| = 1$. У такого агента есть фиксированный набор действий A и возможность выбора из этого набора действий.

В искусственном интеллекте общепринята изящная и доступная для понимания модель поведения такого агента. Предположим, что агент находится в комнате с несколькими игровыми автоматами («однорукими бандитами», далее — просто «бандитами»); в литературе устоялось вынесенное в заголовок параграфа название: если объединить нескольких «одноруких бандитов», получится один «многорукий бандит» (multi-armed bandit). У каждого «бандита» своё, неизвестное агенту ожидание выигрыша. Агент должен за ограниченное число попыток получить максимальную прибыль.

Отметим, что такая, казалось бы, сильно упрощённая модель действительно реализует все возможные ситуации с одним состоянием. За каждое предпринятое действие среда даёт вознаграждение (возможно, случайное), а затем возвращает агента в исходное состояние: результат следующего подхода к «бандиту» не зависит от предыдущих.

Прежде чем перейти к конкретным алгоритмам, рассмотрим общие принципы, которые должны руководить агентом в такой ситуации. Разумеется, хочется применить в каком-то смысле «жадную» стратегию, т.е. всегда выбирать стратегию, максимизирующую прибыль. Однако совершенно очевидно, что агент в комнате не знает с самого начала, что ему делать, и не может выбрать самую лучшую стратегию. Более того, даже если он уже дёрнул по одному разу за ручку каждого «бандита» и

только один раз победил, это вовсе не означает, что нужно теперь всё время выбирать только принесший прибыль автомат. Как же соблюсти баланс между изучением окружающей среды и использованием уже разработанных стратегий?

Как ни странно, единый ответ существует. Он вполне логичен: полезная эвристика в такой ситуации — *оптимизм при неопределённости*. Иначе говоря, выбирать стратегию поведения нужно жадно, но при этом следует быть весьма оптимистичным относительно ожидаемой прибыли. Должны быть получены серьёзные отрицательные свидетельства, прежде чем та или иная стратегия может быть отклонена.

Чуть позже мы увидим, как эта рекомендация реализуется на практике, а пока рассмотрим методы, которые *гарантированно* находят оптимальное решение или сходятся к нему.

§ 7.4. Доказуемо оптимальные алгоритмы

В этом параграфе мы рассматриваем алгоритмы, анализ работы которых проводится достаточно несложно и приводит к хорошим результатам. Это, правда, вовсе не означает, что мы докажем все утверждения о рассмотренных здесь алгоритмах; но значительная их часть будет непосредственно очевидной.

Первая, простейшая постановка будет решена при помощи динамического программирования. Принцип динамического программирования широко используется для самых различных задач [16, 170]; его история восходит к самой заре информатики, сороковым годам XX века [11].

Не стало исключением и обучение с подкреплением. Для решения этой задачи динамическое программирование можно применить, если заранее известен срок жизни агента: предположим, что агент действует на протяжении h шагов.

Тогда для определения оптимальной стратегии можно использовать несложный байесовский подход. Задача состоит в том, чтобы вычислить отображение из всех возможных *состояний опыта* (belief states) агента во множество действий, таким образом задав ему стратегию поведения.

Состояние опыта агента в каждый момент времени мы будем записывать как

$$\mathcal{S} = \{(n_1, w_1), \ldots, (n_k, w_k)\},$$

что означает, что каждого «бандита» i запустили n_i раз, получив при этом выигрыш w_i раз (здесь и далее мы считаем, что результат бинарный — либо выиграл, либо проиграл).

Обозначим через $V^*(\mathcal{S})$ ожидаемый оставшийся выигрыш для состояния с данной историей. Базой для рекуррентных соотношений будет служить случай, когда $\sum_{i=1}^{k} n_i = h$. В этом случае агенту больше нечего делать, и $V^*(\mathcal{S}) = 0$. Если же мы знаем V^* для всех состояний, когда у агента ещё осталось t попыток, мы сможем пересчитать V^* и для состояний с $t+1$ оставшейся попыткой:

$$V^*(n_1,w_1,\ldots,n_k,w_k) = \\ = \max_i \Big(\rho_i(1 + V^*(\ldots,n_i+1,w_i+1,\ldots)) + \\ + (1-\rho_i)V^*(\ldots,n_i+1,w_i,\ldots)\Big),$$

где ρ_i — апостериорные вероятности того, что действие i оправдается, при данных опыта (n_i, w_i). Давайте на минутку отвлечёмся и приведём простой, но характерный пример вычисления такого рода апостериорных вероятностей.

Пример 7.3. Испытания Бернулли

Постановка задачи проста: у нас имеется последовательность подбрасываний монетки, о «честности» которой нам ничего не известно. Требуется определить, какова вероятность получить орёл или решку в следующем подбрасывании. Задача эта была решена ещё Лапласом; правда, в тогдашней формулировке она была куда поэтичнее: «Какова вероятность того, что завтра снова взойдёт Солнце?» Имеется в виду ситуация, в которой мы ничего не знаем о Солнце априори, но уже много (по мнению современников Лапласа, около 6000) лет его наблюдали.

Отметим, что это задача с сильным байесовским «душком»: перед нами распределение вероятностей, параметров (в данном случае — одного параметра) которого мы не знаем, и данные. От нас требуется дать прогноз, каким-то образом эти параметры обучив по данным.

Обозначим вероятность выпадения орла через q: мы сейчас пытаемся вывести формулу для плотности вероятности $f(q)$. Обозначим случайные величины, соответствующие данным — через X_i, а сами результаты — через x_i (будем считать, что орёл соответствует единице,

а решка — нулю). Тогда, по теореме Байеса,

$$p(q \mid X_1 = x_1, \ldots, X_n = x_n) = \frac{p(X_1 = x_1, \ldots, X_n = x_n \mid q)p(q)}{p(X_1 = x_1, \ldots, X_n = x_n)}.$$

Мы будем считать, что априорное распределение «честности» монетки *равномерное*, то есть пока мы не видели ни одного результата, мы вообще ничего не можем сказать о вероятности выпадения орла и считаем все такие вероятности одинаково возможными. Это значит, что вероятность $p(q)$ тождественно равна 1 для всех $0 \leq q \leq 1$ и равна нулю вне отрезка $[0,1]$.

Давайте вычислим правдоподобие q:

$$p(X_1 = x_1, \ldots, X_n = x_n \mid q) = \prod_{i=1}^{n} q^{x_i}(1-q)^{1-x_i} = q^s(1-q)^{n-s},$$

где s — число орлов в имеющихся данных, $s = x_1 + \ldots + x_n$.

Осталось только подсчитать $p(X_1 = x_1, \ldots, X_n = x_n)$, то есть фактически нормировочную константу. Чтобы вычислить эту вероятность, нам придётся просуммировать условную вероятность $p(X_1 = x_1, \ldots, X_n = x_n \mid q)$ по всем значениям q; в непрерывном случае, естественно, из суммы получится интеграл:

$$p(X_1 = x_1, \ldots, X_n = x_n) = \int_0^1 q^s(1-q)^{n-s}\,dq = \frac{s!(n-s)!}{(n+1)!}.$$

(мы предполагаем, что читателю известно — например, из курса математического анализа, — как брать интегралы вида $\int x^a(1-x)^b dx$). Таким образом, мы получаем формулу для плотности вероятности q:

$$f(q) = \frac{(n+1)!}{s!(n-s)!}q^s(1-q)^{n-s}.$$

Это так называемое *бета-распределение*, которое часто появляется в статистике, обычно именно как сопряжённое априорное распределение для испытаний Бернулли. Нас сейчас интересует только математическое ожидание вероятности следующего орла.

$$E[q] = \int_0^1 q\frac{(n+1)!}{s!(n-s)!}q^s(1-q)^{n-s}\,dq =$$

$$= \frac{(n+1)!}{s!(n-s)!}\int_0^1 q^{s+1}(1-q)^{n-s}\,dq = \frac{(n+1)!}{s!(n-s)!}\,\frac{(s+1)!(n-s)!}{(n+2)!} = \frac{s+1}{n+2}.$$

Получилось, что в случае, когда мы вообще ничего заранее не знаем заранее, следующего орла после получения последовательности из s орлов и $n-s$ решек можно ожидать с вероятностью $\frac{s+1}{n+2}$. В частности, если данных вообще нет, эта вероятность равна $\frac{1}{2}$. Что и логично.

`DynamicReinforcement(h,k)`:

1. Инициализировать таблицу
$$\{V^*(n_1,w_1,\ldots,n_k,w_k)\}_{n_i,w_i=1}^{h}.$$
2. Для всех элементов, для которых $\sum_{i=1}^{k} n_i \geqslant h$, присвоить $V^*(n_1,w_1,\ldots,n_k,w_k) := 0$.
3. Для всех t от h до 0:
   a) Для всех элементов, для которых $\sum_{i=1}^{k} n_i = t$, присвоить
$$V^*(n_1,w_1,\ldots,n_k,w_k) := \\ := \max_i \left\{ \frac{w_i+1}{n_i+2}\,(1+V^*(\ldots,n_i+1,w_i+1,\ldots)) + \right. \\ \left. + \left(1-\frac{w_i+1}{n_i+2}\right) V^*(\ldots,n_i+1,w_i,\ldots) \right\}.$$
4. На каждом шаге с данным фиксированным состоянием опыта $(n_1,w_1,\ldots,n_k,w_k)$ выбирать действие i, для которого достигается
$$\max_i \left\{ \frac{w_i+1}{n_i+2}\,(1+V^*(\ldots,n_i+1,w_i+1,\ldots)) + \right. \\ \left. + \left(1-\frac{w_i+1}{n_i+2}\right) V^*(\ldots,n_i+1,w_i,\ldots) \right\}.$$

Рис. 7.3. Динамическое программирование для обучения с подкреплением

Таким образом, в случае с испытаниями Бернулли в виде «одноруких бандитов»
$$\rho_i = \frac{w_i+1}{n_i+2}.$$

Получившийся алгоритм представлен на рис. 7.3. Как видно, он получился не слишком-то эффективным: цена построения такой таблицы линейна от произведения числа состояний опыта на число действий $|A|$, что экспоненциально зависит от h. Иначе говоря, динамическое программирование можно использовать только в случаях, когда нужно планировать не слишком далеко. Но зато такой подход гарантированно приводит к оптимальной стратегии.

Читатель, наверное, уже обратил внимание, что предвычисленная таблица V^* совершенно не зависит от конкретной ситуации — её можно подсчитать хоть раз и навсегда, а от конкретных ожиданий выигрыша конкретных «одноруких бандитов» зависеть будет только путь по этой таблице, который изберёт оптимальная стратегия. Может быть, можно сделать всё заранее, один раз, а потом брать значения из таблицы? Да, можно.

Пусть мы уже n раз дёрнули за ручку какого-то «бандита» и получили w единичек. Существуют предвычисленные таблицы *индексов Гиттинса* (Gittins allocation indices) $I(n,w)$, которые учитывают как ожидаемую прибыль, так и количество новой информации, которую мы получим, если предпримем это действие ещё раз. Таким образом, оптимальная стратегия крайне проста: достаточно выбирать «бандита», для которого $I(n,w)$ максимален.

Индексы Гиттинса работают отлично, но, к сожалению, пока не обобщаются — их аналоги для нескольких состояний агента неизвестны.

Третий приём из тех, анализ которых провести достаточно просто, связан с «тренировкой» оптимальной стратегии. Действительно, в модели с комнатой с «однорукими бандитами» любой алгоритм можно представить как набор вероятностей, с которыми он выбирает то или иное действие (всё остальное не важно для конечного результата). И эти вероятности можно тренировать примерно так же, как мы в Главе 3 тренировали перцептроны. *Алгоритм линейного вознаграждения–бездействия* (linear reward–inaction algorithm) линейно увеличивает вероятность действия a_i, если оно привело к успеху:

$$\begin{aligned} p_i &:= p_i + \alpha(1 - p_i), \\ p_j &:= p_j - \alpha p_j, \quad j \neq i. \end{aligned}$$

Если же действие безуспешно, вероятности сохраняются. В этих формулах α — константа, задающая скорость обучения (вспомните аналогичную константу в алгоритмах обучения нейронных сетей). Алгоритм представлен на рис. 7.4.

Алгоритм линейного вознаграждения–бездействия с вероятностью 1 сходится к вектору, в котором только одна единичка,

`LinearRewardInaction`(α):

1. Случайно инициализировать вероятности p_i.
2. Пока агент продолжает работу:
   а) Выбрать действие i с вероятностью p_i.
   б) Если действие привело к успеху:

$$\begin{aligned} p_i &:= p_i + \alpha(1 - p_i), \\ p_j &:= p_j - \alpha p_j, \quad j \neq i. \end{aligned}$$

Рис. 7.4. Алгоритм линейного вознаграждения–бездействия

а остальные компоненты — нули. Он не всегда сходится к оптимальной стратегии; вполне вероятно, что не очень удачные первые попытки предопределят сходимость к неоптимальной стратегии. Но вероятность ошибки у этого алгоритма можно сделать сколь угодно малой, уменьшая α.

§ 7.5. Другие стратегии

В этом параграфе мы рассмотрим две стратегии. Первая из них — типичный представитель случайных стратегий, а вторая — представитель «оптимистично-жадных» методов, которых мы уже коснулись в § 7.3.

Простейшая случайная стратегия выбирает действие с наилучшей ожидаемой прибылью с вероятностью p, а с вероятностью $1 - p$ выбирает какое-нибудь другое случайное действие. Обычно, для того чтобы сначала исследовать побольше разных вариантов, а впоследствии — почаще использовать найденную оптимальную стратегию, начинают с малого p и постепенно его увеличивают. Очевидный минус этого подхода в том, что алгоритм не отличает хорошую неоптимальную альтернативу от плохой, выделяя только оптимальную стратегию.

В принципе понятно, как исправить этот недостаток — нужно снабжать разные стратегии вероятностями, которые тем больше, чем больше ожидаемая выгода от данной стратегии. Но при этом нужно обеспечить тот же эффект, что и раньше: сначала давать больший вес исследованиям окружающей среды, а со временем переходить на следование оптимальному курсу. Элегантное решение приходит из... статистической физики.

Этот метод так и называется — *исследование по Больцману*[1] (Boltzmann exploration):

$$p(a) = \frac{e^{ER(a)/T}}{\sum_{a'} e^{ER(a')/T}},$$

где ER — ожидаемая прибыль, а T — параметр, отвечающий за соотношение исследования и оптимального поведения (этот параметр называется *температурой* — в аналогичном уравнении статистической физики именно температура отвечает за то,). Как и в предыдущей стратегии, во время исследования по Больцману температура обычно понижается со временем.

Теперь давайте рассмотрим один из самых естественных способов применить оптимистично-жадный метод. Предположим, что для каждого действия мы храним статистику n (общее количество испытаний) и w (количество успехов), а потом хотим найти *оптимистичную* границу доли успехов в испытаниях. Как это сделать?

Разумеется, первый приходящий на ум статистический аппарат — *доверительные интервалы*. Они и лягут в основу нашего оптимистично-жадного метода. Возьмём доверительный интервал для вероятности успеха испытаний с границей $1 - \alpha$, а для выбора стратегии будем использовать верхнюю границу этого интервала.

Пример 7.4. Пример вычисления доверительных интервалов

Рассмотрим испытания Бернулли (подбрасывание монетки). Тогда с вероятностью 0,95 среднее лежит в интервале

$$\left(\bar{x} - 1{,}96\frac{s}{\sqrt{n}}, \bar{x} + 1{,}96\frac{s}{\sqrt{n}}\right),$$

[1] Людвиг Эдуард Больцман (Ludwig Eduard Boltzmann, 1844–1906) — австрийский физик, основатель статистической механики и молекулярно-кинетической теории. Его жизненный путь был сложным: он начинал в университете Граца, позже читал публичные лекции по философии, которыми интересовался сам император, но к концу жизни оказался подвержен депрессиям — увы, человеку науки трудно их избежать. Один из приступов депрессии закончился трагично: Больцман покончил жизнь самоубийством. Разумеется, он не имел никакого отношения к обучению с подкреплением — но путь идей причудлив, и уравнения, которые мы здесь приводим, весьма схожи с уравнениями статистической физики.

где n — число испытаний, $s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$, а константа 1,96 является функцией от 0,95 и берётся из таблиц классического *распределения Стьюдента*[1] [178, 179].

§ 7.6. Поиск стратегий в известной модели

В этом параграфе мы наконец-то перейдём от «многоруких бандитов», которые являются полезной, но всё же в большинстве случаев недостаточной абстракцией, к реальной ситуации: несколько состояний, функция вознаграждения, зависящая от состояния, и вероятностные переходы между состояниями. Начнём с основного определения.

Определение 7.1. *Марковский процесс принятия решений* (Markov decision process) состоит из:

- множества состояний S;
- множества действий A;
- функции поощрения $R : S \times A \to \mathbb{R}$;
- функции перехода между состояниями

$$T : S \times A \to \Pi(S),$$

где $\Pi(S)$ — множество распределений вероятностей над S. Вероятность попасть из s в s' после a равна $T(s,a,s')$.

Слово «марковский» здесь, как и во всей теории вероятностей, означает, что вероятности переходов между состояниями не зависят от истории предыдущих переходов. Действительно, на вход функции T предыдущих состояний агента не подаётся. Агент должен за время своей работы выбрать оптимальное поведение, не зная заранее модели.

[1] Уильям Госсет (William Sealy Gosset, 1876–1937) — английский статистик. Госсет был учеником Карла Пирсона (того самого, чей тест), но работал не в биометрии, как Пирсон, а в знаменитой пивоварне «Guinness & Son». Боясь разгласить важные ноу-хау, Гиннесс запретил своим работникам что бы то ни было публиковать. Госсет, конечно, всё равно тайком публиковал свои результаты, но для этого ему пришлось использовать псевдоним Student. Поэтому его самый важный результат мы знаем как «распределение Стьюдента», а не «распределение Госсета».

Но давайте на минутку предположим, что мы уже точно знаем нашу модель. Задача от этого вовсе не становится тривиальной: всё ещё нужно отыскать оптимальную стратегию поведения для агента в этой модели. В реальной ситуации, конечно, мы модель не знаем, но давайте сначала решим первую задачу, ведь без неё все равно вряд ли получится вторая.

Введём ещё одно важное определение; здесь и далее мы будем оперировать в рамках модели бесконечного горизонта.

Определение 7.2. *Оптимальное значение состояния* — это ожидаемая суммарная прибыль, которую получит агент, если начнёт с этого состояния и будет следовать оптимальной стратегии:

$$V^*(s) = \max_\pi \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Оптимальное значение состояния можно определить как решение уравнений

$$V^*(s) = \max_{a \in A}\left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')\right).$$

Если знать оптимальные значения состояний, то задача становится практически тривиальной: чтобы получить оптимальную стратегию, достаточно вычислить

$$\pi^*(s) = \operatorname{argmax}_{a \in A}\left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')\right).$$

Таким образом, вопрос об оптимальном поведении с известной моделью фактически свёлся к вопросу о том, как решать вышеприведённые уравнения. Рассмотрим несколько вариантов.

Как известно из теории вычислительных методов, решения сложных уравнений обычно ищут итеративными методами; это идеально сходится с общей концепцией машинного обучения, и именно так мы и поступим. Первый вариант — итеративное решение *по значениям*: мы будем искать оптимальные значения состояний простым итеративным алгоритмом (рис. 7.5).

`ValueIteration`(α):

1. Инициализировать $V(s)$.
2. Пока стратегия недостаточно хороша:
   а) для всех $s \in S$ и $a \in A$

$$Q(s,a) := R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s');$$

   б) $V(s) := \max_a Q(s,a)$.

Рис. 7.5. Алгоритм итерации по значениям

Недостаток алгоритма на рис. 7.5 состоит в том, что пересчёт в нём использует информацию от всех состояний-предшественников. Поэтому можно рассмотреть другой, так сказать, «стохастический» вариант:

$$Q(s,a) := Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right).$$

Доказано, что модифицированный таким образом алгоритм сходится к правильным значениям, если каждая пара (s,a) встречается бесконечное число раз, s' выбирают из распределения $T(s,a,s')$, а r сэмплируют со средним $R(s,a)$ и ограниченной вариацией.

Несмотря на то, что вышеприведённые алгоритмы сходятся, они могут сходиться достаточно медленно; кроме того, нет чёткого критерия остановки алгоритма. Возможно, было бы лучше, если бы мы на каждом шаге улучшали не функции значений, а саму стратегию, ведь стратегий всего конечное число (не больше, чем $|A|^{|S|}$). Критерий остановки тогда тоже был бы ясен: как только стратегия перестаёт изменяться, можно останавливаться. Такой алгоритм тоже можно построить — это несложная модификация предыдущего. Он приведён на рис. 7.6.

§ 7.7. Поиск оптимальных стратегий без модели

Научившись полностью решать простой случай, когда модель известна заранее и полностью, теперь можно перейти к более реалистичным случаям — когда модель нужно строить с нуля. Можно выделить два основных подхода к решению этой

PolicyIteration(α):

1. Инициализировать π.
2. Повторять следующие инструкции.
   а) Вычислить значения состояний для стратегии π, решив систему линейных уравнений
   $$V_\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s') V_\pi(s').$$
   б) Улучшить стратегию на каждом состоянии:
   $$\pi'(s) := \operatorname{argmax}_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_\pi(s') \right).$$
   в) Если $\pi \neq \pi'$, повторить, иначе закончить.

Рис. 7.6. Алгоритм итерации по стратегиям

задачи. В первом из них мы ищем только стратегию, не пытаясь в явном виде выяснять, каков мир вокруг нас, а во втором также параллельно формируем и обучаем модель ситуации, в которой оказался агент.

Характерный представитель первого подхода — так называемый *адаптивный эвристический критик* (adaptive heuristic critic). Основная идея проста: пусть один элемент алгоритма ищет стратегию, а другой — функцию значений, и пусть они друг с другом в этом соперничают. Такой метод читателю уже знаком: в § 6.5 мы рассматривали алгоритм EM, состоящий из двух частей, из которых одна ищет скрытые параметры, а другая переобучает гипотезу на основе этих параметров.

В контексте обучения с подкреплением роль E- и M-шагов выполняют два компонента: *критик* (AHC, от слов «adaptive heuristic critic») и *компонент обучения с подкреплением* (RL, от слов «reinforcement learning»). На месте RL может быть любой алгоритм обучения с подкреплением для решения задачи о k «бандитах». Задача RL — максимизировать значение эвристики v, вычисляемой критиком. А критик AHC в это время использует полученное от RL значение для пересчёта ожидаемых значений прибыли от состояний.

Легче всего представить себе работу адаптивного эвристического критика, вообразив, будто это происходит по очереди: сначала мы фиксируем стратегию π и подсчитываем функцию значений V_π. Затем фиксируем V_π и учим новую стратегию π', которая максимизирует V_π, и так далее. Точь-в-точь EM-принцип, только вероятности здесь скрыты чуть поглубже.

Давайте сначала займёмся вопросом о том, как формировать функцию значений по стратегии.

Начнём с того, что опыт, который получает агент в течение своей жизни, нужно как-то структурировать. Мы будем представлять его в виде набора так называемых *кортежей опыта* (experience tuples)

$$\langle s, a, r, s' \rangle,$$

где s — состояние перед переходом, a — действие, r — полученное поощрение, s' — следующее состояние. Данные r и s', вообще говоря, являются функциями от a и s, но функции эти случайные, и агенту для успешной работы как раз и неплохо было бы найти параметры этих распределений.

Такой кортеж полностью содержит всю новую информацию, которую агент получает после очередного действия (вспомним, что у нас марковские процессы, и история предыдущих действий и поощрений не важна).

Правила, которыми пользуется адаптивный эвристический критик для пересчёта функции значений при получении нового опыта, обычно имеют вид $\mathrm{TD}(\lambda)$, где λ, определяет, насколько «глубоко» будет воздействовать каждое изменение на ранее посещённые состояния. Определим сначала простейшую версию такого правила — $\mathrm{TD}(0)$:

$$V(s) := V(s) + \alpha \left(r + \gamma V(s') - V(s) \right).$$

Здесь, как и раньше, γ — константа, отличающая прибыль сейчас от прибыли на следующем шаге, а α — константа обучения. Смысл правила $\mathrm{TD}(0)$ заключается в том, чтобы обновить функцию значения (фактически — ценность) состояния, из которого был совершён переход, на основании того, куда агент попал в результате. Если агент смог попасть в хорошее место, функция значения увеличивается, если в плохое — функция значения уменьшается.

Очевидно, что при помощи той же самой логики можно сделать и следующий шаг: заглянуть на несколько ходов вперёд. Иначе говоря, если мы сейчас смогли попасть в очень хорошее состояние, которое принесло нам много «денег», то это должно увеличить не только ценность непосредственно предшествующего ему состояния, но и состояний, которые были раньше — ведь из них тоже можно попасть в это замечательное место, просто не за один ход, а за несколько. Но при этом, конечно, нужно вводить какие-то понижающие коэффициенты: чем дальше путь, тем больше вероятность, что пройти его не удастся (напомним, что переходы между состояниями в общем случае тоже вероятностные), да и приз при временно́м удалении тоже уменьшается.

В этом и заключается суть семейства стратегий $\mathrm{TD}(\lambda)$. В общем виде правило $\mathrm{TD}(\lambda)$ выглядит как

$$V(u) := V(u) + \alpha\left(r + \gamma V(s') - V(s)\right)\epsilon(u).$$

При этом правило $\mathrm{TD}(0)$ после очередного шага применяется к каждому состоянию u, но модифицируется за счёт коэффициента применимости (eligibility) $\epsilon(u)$:

$$\epsilon(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k}[s = s_k],$$

где $[s = s_k] = 1$, если $s = s_k$, и 0 в противном случае. Здесь $[s = s_k]$ — пример так называемой *нотации Айверсона* — весьма изящного инструмента, позволяющего более компактно записывать очень многие факты в дискретной математике. Попытку её популяризировать недавно предприняли авторы известной книги «Конкретная математика» [169]. Для обозначения равенства двух величин подошла бы, конечно, и *дельта Кронекера*[1]

[1] Леопольд Кронекер (Leopold Kronecker, 1823–1891) — немецкий логик и математик. Как логик Кронекер известен своими финитарными взглядами на математику, которые сделали его предтечей интуиционизма. По мнению Кронекера, «Господь Бог создал целые числа; всё остальное — дело рук человеческих»; известно, что когда Линдеманн представил своё доказательство трансцендентности π, Кронекер поздравил его с блестящим доказательством, но тут же добавил, что оно ничего не доказывает, ведь трансцендентных чисел не существует. Такие взгляды привели Кронекера к серьёзным спорам со своим учеником Георгом Кантором, создателем теории множеств, и Вейерштрассом, который исповедовал функциональный взгляд

$\delta_{s,s_k} = [s = s_k]$, но нотация Айверсона позволяет, во-первых, не вводить новых букв (δ — не такая уж редкая буква в математике, чтобы закреплять её за дельтой Кронекера), а во-вторых, может содержать любое логическое условие, не обязательно равенство.

Применимость зависит от того, насколько часто это состояние посещалось в недавнем прошлом. Если $\lambda = 0$, $\mathrm{TD}(\lambda)$ — это $\mathrm{TD}(0)$ (при условии, конечно, что $0^0 = 1$).

Отметим, что применимость можно также пересчитывать в реальном времени, после каждого шага, не вычисляя всю сумму заново. Пересчёт можно вести по формуле

$$\epsilon(s) := \gamma\lambda\epsilon(s) + [s = \text{текущему состоянию}].$$

Теперь осталось только понять, как мы будем пересчитывать стратегии. Для этого введём набор весов $w(s,a)$, соответствующих действиям a, которые можно предпринять в состоянии s. Веса будут определять вероятность, с которой мы выбираем следующее действие; вероятности будут зависеть от весов экспоненциально (или, с другой стороны, веса, как это очень часто бывает, будут представлять собой логарифмы вероятностей):

$$p(a|s) = \frac{e^{w(s,a)}}{\sum_{a'} e^{w(s,a')}}.$$

Эти веса мы будем обучать при помощи той же техники, что и раньше — при поступлении кортежа опыта $\langle s,a,r,s'\rangle$

$$w(s,a) := w(s,a) + \beta(r + \gamma V(s') - V(s)),$$

где β — константа обучения, возможно, отличная от α. Суть адаптивного критика в том, что мы в этой формуле используем значения V, полученные тут же при помощи того же обучающегося алгоритма. На рис. 7.7 показана схема алгоритма адаптивного эвристического критика.

Вариантом адаптивного эвристического критика является так называемое Q-обучение (Q-learning). Оно работает примерно на тех же принципах, что и адаптивный эвристический критик, но вместо функции значения $V(s)$ мы рассматриваем функцию с

на вещи. В математике Кронекеру принадлежат значительные результаты в теории чисел, теории эллиптических функций и высшей алгебре.

AdaptiveHeuristicCritic(α,β):

1. Инициализировать $e(s)$, $V(s)$, $w(s,a)$ для каждого состояния s и действия a.
2. Для каждого кортежа опыта $\langle s,a,r,s'\rangle$:
   а) Для каждого состояния u
   $$e(u) := \gamma\lambda e(u) + [u = s].$$
   б) Для каждого состояния u
   $$V(u) := V(u) + \alpha(r + \gamma V(s') - V(s))e(u).$$
   в) $w(s,a) := w(s,a) + \beta(r + \gamma V(s') - V(s))$.
   г) Выбрать в качестве следующего действия случайное действие a' по распределению
   $$p(a') = \frac{e^{w(s',a')}}{\sum_a e^{w(s',a)}}.$$

Рис. 7.7. Адаптивный эвристический критик

двумя параметрами $Q^*(s,a)$ — ожидаемую оптимальную награду за действие a в состоянии s (здесь и далее Q^* и V^* обозначают оптимальные значения Q и V). Функция Q отличается от V тем, что зависит от двух параметров; зная Q, нетрудно получить V:

$$V^*(a) = \max_a Q^*(s,a).$$

Если $Q^*(s,a)$ известны, то можно легко найти оптимальную стратегию:

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a).$$

А если подставить в описанные в предыдущем параграфе формулы Q^* вместо V^*, то можно получить и уравнения на $Q^*(s,a)$:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'\in S} T(s,a,s') \max_{a'} Q^*(s',a').$$

Идея самого алгоритма точно такая же, но мы храним и обучаем не V, а Q. Правило обучения не отличается от адаптивного эвристического критика:

$$Q(s,a) := Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)),$$

QLearning(α):

1. Инициализировать $Q(s,a)$ для каждого состояния s и действия a.
2. Для каждого кортежа опыта $\langle s,a,r,s'\rangle$:
   а) обновить значения $Q(s,a)$:
   $$Q(s,a) := Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a));$$
   б) выбрать в качестве следующего действия
   $$\pi(s') = \mathrm{argmax}_a Q(s',a).$$

Рис. 7.8. Алгоритм Q-обучения

где $\langle s,a,r,s'\rangle$ — кортеж опыта. Схема представлена на рис. 7.8.

И для адаптивного эвристического критика, и для Q-обучения доказано, что соответствующие алгоритмы со временем сходятся к правильным значениям [77, 150] (ещё быстрее сходится вариант Q-обучения — *отложенное Q-обучение*, delayed Q-learning [149]); однако эти доказательства мы здесь рассматривать не будем.

§ 7.8. Поиск моделей и оптимальных стратегий по ним

Последним подходом к обучению с подкреплением, который мы рассмотрим, станет подход, при котором алгоритм одновременно и поддерживает модель окружающего мира, и ищет оптимальную стратегию.

Первым на ум приходит алгоритм, который сначала какое-то время занимается исследованиями, обновляя модель окружающего мира, а затем пользуется полученной моделью для выбора оптимальной стратегии. Недостатки такого подхода очевидны: во-первых, когда можно с уверенностью сказать, что пора остановиться? Нужно проводить какую-то границу между обучением и действием, и где именно её проводить — априори совершенно не понятно. Во-вторых, сбор данных может быть весьма неэффективен: ведь чтобы проверить, как то или иное состояние реагирует на все возможные действия, нужно их все предпринять, причём в вероятностной ситуации — не по одному разу. И, в-третьих, в реальных задачах часто возникают ситуации, когда

обучающемуся алгоритму нужно уметь подстраиваться под меняющуюся окружающую среду. Если окружающая агента среда вдруг изменится, когда он уже завершил обучение и находится на стадии действия, он этого не заметит и будет продолжать работать по старой неэффективной программе.

Будем действовать примерно так же, как и в предыдущем параграфе — будет одновременно строить модель по получаемому опыту, а по текущему состоянию модели — оптимизировать стратегию.

Один из самых успешных алгоритмов, которые действуют таким образом, называется Dyna. При подаче на вход кортежа опыта $\langle s,a,r,s'\rangle$ Dyna сначала обновляет модель, добавив статистику для перехода из s в s' под действием a и для получения награды за a в состоянии s. Тем самым в алгоритме поддерживаются в явном виде функции $\hat{T}$ и $\hat{R}$ — выведенные из опыта модели реальных функций перехода и вознаграждения T и R.

Затем, обновив модель, Dyna обновляет стратегию в состоянии s:

$$Q(s,a) := \hat{R}(s,a) + \gamma \sum_{s'} \hat{T}(s,a,s') \max_{a'} Q(s',a').$$

Но этого оказывается недостаточно: изменившаяся модель влияет не только на это состояние, и если мы будем менять только его, то нам придётся обучаться слишком долго, прежде чем агент сможет приблизиться к целевым функциям. Поэтому Dyna заранее выбирает k (в зависимости от имеющихся вычислительных ресурсов — чем их больше, тем большим можно брать k) и делает ещё k обновлений для k случайно выбранных пар (s_k,a_k):

$$Q(s_k,a_k) := \hat{R}(s_k,a_k) + \gamma \sum_{s'} \hat{T}(s_k,a_k,s') \max_{a'} Q(s',a').$$

Затем выбирается оптимальное действие a' в новом состоянии s' на основе значений Q (возможно, исправленных стратегией изучения среды — на начальном этапе, как и в предыдущих алгоритмах, имеет смысл немножко «подкрутить» алгоритм так, чтобы он предпочитал исследование новых возможностей), и это действие даёт новый кортеж опыта, на котором цикл повторяется.

Dyna(k):

1. Инициализировать
$$T(s,a,s'),\ R(s,a),\ Q(s,a),\ n(s,a),\ n(s,a,s')$$
для всех состояний s, s' и действия a.
2. Для каждого кортежа опыта $\langle s,a,r,s'\rangle$:
   а) $n(s,a) := n(s,a) + 1$, $n(s,a,s') := n(s,a,s') + 1$.
   б) $R(s,a) := R(s,a) + \frac{r - R(s,a)}{n(s,a)}$.
   в) Для каждого состояния u
$$T(s,a,u) := T(s,a,u) + \frac{[u = s'] - T(s,a,u)}{n(s,a,u)}.$$
   г) Обновить стратегию:
$$Q(s,a) := R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a').$$
   д) Для i от 1 до k:
      (i) Выбрать случайную пару (s_i, a_i).
      (ii) Обновить стратегию:
$$Q(s_i,a_i) := R(s_i,a_i) + \gamma \sum_{s'} T(s_i,a_i,s') \max_{a'} Q(s',a').$$
   е) Выбрать в качестве следующего действия
$$\pi(s') = \operatorname{argmax}_a Q(s',a).$$

Рис. 7.9. Алгоритм Dyna

Схема алгоритма приведена на рис. 7.9. Для хранения данных об уже полученном опыте используются вспомогательные счётчики $n(s,a)$ и $n(s,a,s')$ — переменные, в которых записано, сколько раз алгоритм выбирал действие a в состоянии s и сколько раз переходил после этого в состояние s' соответственно; эти счётчики в начале работы программы инициализируются нулями.

Dyna работает хорошо, но она бесцельна: обновляет случайные пары. Улучшенный алгоритм, так называемый алгоритм «уборки по приоритетам» (prioritized sweeping) делает то же, но у каждого состояния есть *приоритет*, и для каждого состояния

PrioritizedSweeping(k):

1. Инициализировать

$$T(s,a,s'),\ R(s,a),\ V(s),\ n(s,a),\ n(s,a,s'),\ pr(s,a)$$

для всех состояний s, s' и действия a.

2. Для каждого кортежа опыта $\langle s,a,r,s'\rangle$:
   а) $n(s,a) := n(s,a) + 1$, $n(s,a,s') := n(s,a,s') + 1$.
   б) $R(s,a) := R(s,a) + \frac{r - R(s,a)}{n(s,a)}$.
   в) Для каждого состояния u

$$T(s,a,u) := T(s,a,u) + \frac{[u = s'] - T(s,a,u)}{n(s,a,u)}.$$

   г) Выбрать k пар с максимальными $pr(s,a)$:

$$\{(s_i,a_i)\}_{i=1}^{k}.$$

   д) Для i от 1 до k:
      (i) $V_{old} := V(s_i)$.
      (ii) Обновить значения состояний:

$$V(s_i) := \max_a \left(R(s_i,a) + \gamma \sum_{s'} T(s_i,a,s')V(s') \right).$$

      (iii) $pr(s_i,a_i) := 0$.
      (iv) Для всех состояний s и действий a

$$pr(s,a) := pr(s,a) + |V_{old} - V(s_i)|T(s,a,s_i).$$

   е) Выбрать в качестве следующего действия

$$\pi(s') = \operatorname{argmax}_a \left(R(s',a) + \sum_u T(s',a,u)V(u) \right).$$

Рис. 7.10. Алгоритм уборки по приоритетам

запоминаются его предшественники — состояния, из которых можно попасть в это состояние с ненулевой вероятностью.

Алгоритм поддерживает точно такую же статистику, как и Dyna, но значения теперь присваиваются состояниям, а не парам (состояние, действие), и вместо k случайных пар обновляются

значения k наиболее приоритетных. Потом приоритеты обновлённых пар обнуляются (чтобы поддерживать ротацию), а увеличиваются приоритеты тех состояний, из которых можно попасть в состояния, которые только что обновились. Но не просто увеличиваются, а увеличиваются пропорционально изменению функции V.

Это приводит к интересному и разумному эффекту. Суть в том, что когда V изменяется значительно, это значит, что текущее действие принесло много информации, результат его был весьма удивительным для нашего агента. И в этом случае алгоритм старается как можно быстрее распространить эту новую информацию предкам «интересного» состояния. А если действие принесло тот результат, который и ожидался, то новой информации мало, и нет большого смысла торопиться её распространять. Формальная схема алгоритма изображена на рис. 7.10.

Практические тесты показывают, что алгоритм уборки по приоритетам требует для сходимости наименьшего числа итераций, но при этом каждая итерация требует значительно более серьёзных вычислительных ресурсов, чем Dyna [77]. Что предпочесть при реализации, может определить только конкретная постановка задачи. Если агент может безболезненно пробовать, ошибаться и учиться, но при этом обладает не слишком мощным вычислительным аппаратом, Dyna, возможно, лучше подойдёт для решения задачи; а если время обучения ограничено жёстко, то стоит предпочесть алгоритм уборки по приоритетам.

§ 7.9. Игрушечный пример Q-обучения и Dyna

В этом параграфе, завершающем главу, мы приведём пример реального программного кода, реализующего Q-обучение и алгоритм Dyna, о которых мы говорили в предыдущих параграфах. Для этого давайте сначала опишем ту постановку задачи (весьма игрушечную, конечно), в которой будут действовать эти алгоритмы.

Предположим, что на небольшом поле (у нас поле будет размера 10×10, но это не принципиально) находятся несколько существ (их у нас будет 20, но это тем более не принципиально). Мы хотим обучить существа атаковать друг друга (или,

Рис. 7.11. Зона видимости агента

наоборот, говорить добрые слова — это, опять же...). Иначе говоря, у каждого существа есть четыре возможных действия:

ACT_MOVELEFT	—	сдвинуться налево;
ACT_MOVERIGHT	—	сдвинуться направо;
ACT_MOVEFORWARD	—	сдвинуться вперёд;
ACT_ATTACK	—	атаковать.

При сдвиге налево, направо или вниз агент соответствующим образом поворачивается; положение «вперёд» отсчитывается от текущей ориентации агента.

Мы признаём атаку удачной, если агент выбрал действие ACT_ATTACK в момент, когда непосредственно перед ним стоит другой агент; в этом случае мы будем прибавлять ему награду REWARD_ATTACK (у нас она будет равна 10), а тому, кого он атаковал, прибавим награду REWARD_ATTACKED (она, разумеется, будет отрицательной: -5).

Приведём для начала вспомогательный файл constants.py, в который вынесены константы и вспомогательные процедуры. Чтобы не загромождать книгу, мы опустили тексты нескольких совсем уж технических процедур. Суть функций incx, decx, incy и decy в том, чтобы вывести увеличенные или уменьшенные на заданную величину координаты по осям x и y нашего

поля (это не вполне тривиально, потому что для простоты поле представляет собой поверхность тора, на котором нет никаких границ). Функция withinsight определяет «зону видимости» агента: она выдаёт, как агент в точке (x,y) видит агента, находящегося в точке (x_1,y_1); возможны варианты C_SEE_AHEAD, C_SEE_LEFT, C_SEE_RIGHT, C_SEE_BEHIND и C_SEE_HERE (см. зоны видимости агента на рис. 7.11).

Листинг 7.1. Файл constants.py стимулируемого обучения

```
MAPWIDTH, MAPHEIGHT = 10, 10
N = 20

MOVESPERTIMEUNIT = 100

ALPHA, GAMMA, DYNA_K = 0.05, 0.8, 100
REWARD_ATTACK, REWARD_ATTACKED = 10, -5

ACTIONNUMBER = 4
ACT_MOVELEFT, ACT_MOVERIGHT = 0, 1
ACT_MOVEFORWARD, ACT_ATTACK = 2, 3

C_SEE_NONE = -1
C_SEE_AHEAD = 0
C_SEE_LEFT = 1
C_SEE_RIGHT = 2
C_SEE_BEHIND = 3
C_SEE_HERE = 4
CONFIGNUMBER = 2 << (C_SEE_HERE + 1)

DIR_TOP, DIR_LEFT, DIR_DOWN, DIR_RIGHT = 0, 1, 2, 3

def calcConfig(seeing):
    config = 0
    pow2 = 1
    for i in xrange(C_SEE_HERE + 1):
        if (seeing[i] > 0):
            config += pow2 * seeing[i]
        pow2 = pow2 << 1
    return config

def seeingByConfig(config):
    c = config
    seeing = [0] * (C_SEE_HERE+1)
    for i in xrange(C_SEE_HERE+1):
        if (c & 1): seeing[i] = 1
        c = c >> 1
```

```
    return seeing


def incx(x, step=1):
    ...
def incy(x, step=1):
    ...
def decx(x, step=1):
    ...
def decy(x, step=1):
    # Эти функции увеличивают и уменьшают координаты
    # с учётом топологии торической доски.
    ...

def withinsight(x, y, x1, y1):
    # Результат этой функции показывает, видна ли для
    # существа, находящегося в точке (x, y), точка
    # (x1, y1), и если видна, то где именно.
    ...

def performMoveAction(cr, action):
    # Эта процедура перемещает существо (учитывая
    # в том числе и направление, куда оно смотрит).
    ...
```

Состояний у наших существ-агентов довольно много, целых 32, ведь нужно учесть все возможные комбинации того, видны ли другие существа спереди, сзади, слева, справа и прямо на том же месте. Однако обучаются агенты с тридцатью двумя состояниями на удивление быстро; дело ещё и в том, что некоторые состояния выпадают очень редко, и хорошо их обучать и не обязательно, и не получится.

Теперь — основной файл, в нём реализована основная логика эксперимента. Отметим техническую деталь: мы использовали систему модулей pygame[1], предназначенных для разработки компьютерных игр на Python; правда, использовали только для обработки клавиатуры. Структура программы достаточно прозрачна; половина агентов обучаются по методике Q-обучения, а половина — по алгоритму Dyna (можно ещё сравнивать их с агентами, которые случайно выбирают своё следующее действие).

[1] http://www.pygame.org/.

Листинг 7.2. Файл main.py стимулируемого обучения

```
import math, random, constants, pygame
from constants import *
from random import random, randrange
from pygame.locals import *

# Класс, описывающий одно существо
class Creature():

    def __init__(self, id, x, y):
        self.id, self.x, self.y, self.dir = id, x, y, DIR_TOP
        self.totalreward = 0
        self.Q, self.R, self.n = [], [], []
        for i in xrange(CONFIGNUMBER):
            self.Q.append( [1] * ACTIONNUMBER )
            self.R.append( [1] * ACTIONNUMBER )
            self.n.append( [1] * ACTIONNUMBER )
        self.T, self.nn = [], []
        for i in xrange(CONFIGNUMBER):
            self.T.append([])
            self.nn.append([])
            for j in xrange(ACTIONNUMBER):
                self.T[i].append(
                  [1 / float(CONFIGNUMBER)] * CONFIGNUMBER )
                self.nn[i].append( [1] * CONFIGNUMBER )

    def chooseAction(self, config):
        if (self.id < N / 2):
            return self.chooseActionQ(config)
        elif (self.id >= N / 2):
            return self.chooseActionDyna(config)
        else:
            return randrange(ACTIONNUMBER)

    def learn(self, action, config, reward, newconfig):
        self.totalreward = self.totalreward + reward
        if (self.id == 0):
            self.learnQ(action, config, reward, newconfig)
        elif (self.id == 1):
            self.learnDyna(action, config, reward, newconfig)

    def chooseActionQ(self, config):
        sum = 0
        for i in xrange(ACTIONNUMBER):
            sum = sum + math.exp(self.Q[config][i])
        r = random()
        curweight = 0
        for i in xrange(ACTIONNUMBER):
```

```
            curweight = curweight +
              math.exp(self.Q[config][i]) / sum
            if (r < curweight):
                return i

    def learnQ(self, action, config, reward, newconfig):
        maxq = 0
        for i in xrange(ACTIONNUMBER):
            if (self.Q[newconfig][i] > maxq):
                maxq = self.Q[newconfig][i]
        self.Q[config][action] = self.Q[config][action] +
          ALPHA*(reward+GAMMA*maxq-self.Q[config][action])

    def chooseActionDyna(self, config):
        max, maxq = 0, 0
        for i in xrange(ACTIONNUMBER):
            if (self.Q[config][i] > maxq):
                max, maxq = i, self.Q[config][i]
        return max

    def learnDyna(self, a, s, reward, news):
        self.n[s][a] = self.n[s][a] + 1
        self.nn[s][a][news] = self.nn[s][a][news] + 1
        self.R[s][a] = self.R[s][a] +
          (reward - self.R[s][a]) / float(self.n[s][a])
        for u in xrange(CONFIGNUMBER):
            if (u == news):
                self.T[s][a][u] = self.T[s][a][u] +
                  (1-self.T[s][a][u])/float(self.nn[s][a][u])
            else:
                self.T[s][a][u] = self.T[s][a][u] -
                  self.T[s][a][u] / float(self.nn[s][a][u])
        maxq = []
        for i in xrange(CONFIGNUMBER):
            maxq.append(0)
            for j in xrange(ACTIONNUMBER):
                if (self.Q[i][j] > maxq[i]):
                    maxq[i] = self.Q[i][j]
        temp_sum = 0
        for i in xrange(CONFIGNUMBER):
            temp_sum = temp_sum + self.T[s][a][i] * maxq[i]
        self.Q[s][a] = self.R[s][a] + GAMMA * temp_sum
        for k in xrange(DYNA_K):
            rand_s = randrange(CONFIGNUMBER)
            rand_a = randrange(ACTIONNUMBER)
            temp_sum = 0
            for i in xrange(CONFIGNUMBER):
                temp_sum = temp_sum +
```

```
                self.T[rand_s][rand_a][i] * maxq[i]
            self.Q[rand_s][rand_a] =
              self.R[rand_s][rand_a] + GAMMA * temp_sum

# Класс, описывающий поле
class World():

    def __init__(self):
        self.n = N
        self.wid = MAPWIDTH
        self.hei = MAPHEIGHT
        self.map = []
        self.timeunits = 0
        for i in xrange(self.wid):
            self.map.append([0 for j in xrange(self.hei)])
        self.creatures = []
        for i in xrange(N):
            creat = Creature(
              i, randrange(self.wid), randrange(self.hei))
            self.creatures.append(creat)

    def doCreatureAction(self, cr, action):
        if (action < ACT_ATTACK):
            performMoveAction(cr, action)

    def calculateReward(self, i, action, rewards):
        if (action == ACT_ATTACK):
            for j in xrange(N):
                if ( (i != j) and (self.creatures[j].x ==
                    self.creatures[i].x) and
                  (self.creatures[j].y ==
                    incy(self.creatures[i].y)) ):
                    rewards[i] = rewards[i] + REWARD_ATTACK
                    rewards[j] = rewards[j] + REWARD_ATTACKED

    def getConfig(self, crnum):
        x = self.creatures[crnum].x
        y = self.creatures[crnum].y
        seeing = [0] * (C_SEE_HERE + 1)
        for i in xrange(N):
            if (i != crnum):
                see = withinsight(x, y,
                  self.creatures[i].x, self.creatures[i].y)
                if (see != C_SEE_NONE):
                    seeing[see] = C_SEE_CREATURE
        return calcConfig(seeing, C_HEAR_NONE)

    def update(self):
```

```
        self.timeunits = self.timeunits + 1
        for t in xrange(MOVESPERTIMEUNIT):
            config, reward, action = [], [0] * N, []
            for i in xrange(N):
                config.append(self.getConfig(i))
                action.append(
                  self.creatures[i].chooseAction(config[i]))
                self.calculateReward(i, action[i], reward)
            for i in xrange(N):
                self.doCreatureAction(
                  self.creatures[i], action[i])
                self.creatures[i].learn(action[i],
                  config[i], reward[i], self.getConfig(i))

# Здесь начинается собственно игровой процесс
pygame.init()
size   = [100, 100]
screen = pygame.display.set_mode(size)
background = pygame.Surface(screen.get_size())
pygame.event.set_grab(1)
world = World()
done = False

while not done:
    events = pygame.event.get( )
    for e in events:
        if (e.type == KEYDOWN):
            if( e.key == K_ESCAPE ):
                done = True
                break
            else:
                world.update()
```

Не будем отнимать у читателя удовольствия попробовать самому реализовать эти (или другие) алгоритмы для этого игрушечного примера и увидеть, что получится. В среднем у нас получалось, что Dyna работает хорошо, а Q-обучение — ещё лучше. Думается, связано это с тем, что модель совсем уж простая. Здесь ни к чему сложное обучение алгоритма Dyna, здесь достаточно простого Q-обучения: увидел врага — беги и убивай. В более сложных ситуациях Dyna покажет своё превосходство, но поля этой книги не так велики, чтобы приводить соответствующие программы.

§ 7.10. Заключение

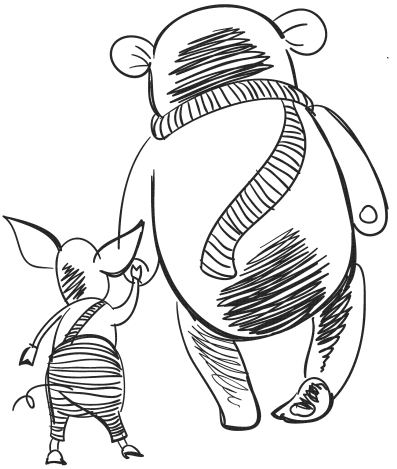
Обучение с подкреплением позволяет обучаться в ситуации, когда агенту заранее неизвестны параметры среды, и их приходится познавать по ходу «жизни». Более того, оно позволяет рассмотреть ситуацию, когда результат поведения задан не однозначно, а только как часть более длинной цепочки действий. Всё это делает обучение с подкреплением достаточно привлекательным для реальных применений — если, конечно, предметная область действительно настолько сложна.

Один пример предметной области, где всё именно так плохо — это попытки обучить компьютер играть в игры. Мы уже упоминали программу TD-Gammon, которая достигла уровня людей-профессионалов в игре в нарды; она была сделана на основе обучения с подкреплением [154–156]. В нардах каждый ход зависит от бросков кубиков, что приводит к достаточно равномерному покрытию дерева возможных вариантов при обучении. Поэтому вероятностное обучение, предполагаемое семейством стратегий $\mathrm{TD}(\lambda)$, хорошо подходит для обучение игре в нарды. Отметим, что в программе TD-Gammon использовался также и другой аппарат машинного обучения: искусственная нейронная сеть проводила оценку позиции, а $\mathrm{TD}(\lambda)$-алгоритм определял, как изменять веса при обучении (ведь тестовых примеров в классическом понимании нет, есть только результат большого числа последовательных выборов хода).

Но ещё более естественными оказались применения в роботике. Когда робот двигает «рукой», он совершает действие, состоящее из многих микродействий, причём для каждого конкретного решения, принимаемого роботом, нет чёткого ответа на вопрос, насколько оно успешно, оценить можно только результат действия в целом. Казалось бы, это создаёт идеальные условия для того, чтобы применить обучение с подкреплением [127, 139, 150]. В роботике, правда, приходится рассматривать более или менее произвольные движения в трёхмерном пространстве (даже более того, потому что шарниров у манипулятора робота будет, скорее всего, несколько). И для «обычных» алгоритмов, которые мы рассматривали в этой главе, данных оказывается маловато. Но ведь человек как-то учится делать руками то, что надо (хоть и с переменным успехом), а мышц у

него ещё побольше будет, чем шарниров у робота? Сейчас исследователи начинают использовать в обучении более «человеческие» механизмы: не только оценивать результат, но ещё и сначала посмотреть, как *это* делает кто-то другой. Так появилась одна из недавних разработок на эту тему — алгоритмы *актёра–критика* (Natural Actor–Critic и его вариация Episodic Natural Actor–Critic) [124–126, 128].

В общем, обучение с подкреплением ещё далеко не сказало своего последнего слова. И действительно, разве мы не именно так учимся всю жизнь?



Литература

[1] *Ahn L., Blum M., Hopper N., Langford J.* CAPTCHA: Using hard AI problems for security // Proceedings of Eurocrypt '03. 2003. P. 294–311.
Статья о CAPTCHA–тестах.

[2] *Ahn L., Blum M., Langford J.* Telling humans and computers apart automatically // Communications of the ACM. 2004. Vol. 47, N. 2. P. 56–60.
Журнальная статья, посвящённая CAPTCHA–подобным тестам.

[3] *Alonso L., Remy J.-L., Schott R.* A Linear-Time Algorithm for the Generation of Trees // Algorithmica. 1997. Vol. 17, N. 2. P. 162–183.
Линейный алгоритм равномерного порождения случайных деревьев.

[4] *Alonso L., Schott R.* Random Generation of Trees. Springer, 1994. 220 p.
В книге изложено большое количество алгоритмов случайного равномерного порождения деревьев.

[5] *Alpaydin E.* Introduction to Machine Learning (Adaptive Computation and Machine Learning). MIT Press, 2004.
Книга со введением в машинное обучение.

[6] *Arora S., Barak B.* Complexity Theory: A Modern Approach. Cambridge University Press, 2009.
Самый современный (на момент написания книги ещё даже не вышедший) учебник по теории сложности.

[7] *Arthur D., Vassilvitskii S.* How Slow is the k–means Method? // Proceedings of the 2006 Symposium on Computational Geometry (SoCG). 2006.
Одна из наиболее свежих работ об эффективности алгоритма k–*средних.*

[8] *Babuška R.* Fuzzy Modeling for Control. Kluwer Academic Publishers, Boston, 1998.
Книга о применении нечёткости для решения практических задач, в том числе задачи кластеризации.

[9] *Baldi P., Brunak S.* Bioinformatics: The Machine Learning Approach. MIT Press, 2001.
Книга о том, как идеи и аппараты машинного обучения применяются в биоинформатике.

[10] *Beigel R.* Perceptrons, PP, and the Polynomial Hierarchy // Structure in Complexity Theory Conference. 1992. P. 14–19.
Замечена и описана связь между перцептронами и схемами из класса $\mathrm{PP}^{\mathrm{PH}}$.

[11] *Bellman R.* Dynamic Programming. Princeton University Press, 1957.
Первая книга о динамическом программировании от его автора.

[12] *Berkhin P.* Survey Of Clustering Data Mining Techniques: Tech. rep. San Jose, CA: Accrue Software, 2002.
Обзор методов анализа данных, основанных на кластеризации.

[13] *Berkowitz S. D.* An Introduction to Structural Analysis: The Network Approach to Social Research. Toronto: Butterworth, 1982.
Классический источник об анализе социальных сетей.

[14] *Berlekamp E.* Algebraic Coding Theory. Aegean Park Press, 1984.
Введение в теорию кодирования, в том числе в её алгебраические аспекты. Указано второе издание, первое — New York, McGraw-Hill, 1968. *Существует и русский перевод: Берлекэмп Э.,* Алгебраическая теория кодирования, *М., Мир,* 1971, *но эту книгу уже трудно где-либо найти.*

[15] *Berry M.* Survey of Text Mining: Clustering, Classification, and Retrieval. Springer, 2003.
Книга о методах анализа текстовой информации.

[16] *Bertsekas D. P.* Dynamic Programming and Optimal Control, vols. 1 & 2. Athena Scientific, 2000.
Двухтомник о динамическом программировании.

[17] *Bhagat P.* Pattern Recognition in Industry. Elsevier, 2005.
Книга о применениях и промышленных реализациях методов распознавания образов.

[18] *Bishop C. M.* Pattern Recognition and Machine Learning. Springer, 2006.
Книга о методах машинного обучения в применении к распознаванию образов.

[19] *Bishop C. M.* Pattern Recognition and Machine Learning. Springer, 2007.
Книга о машинном обучении, в основном на примерах, связанных с распознаванием чего-либо.

[20] *Boole G.* An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities. Cambridge: Macmillan/London: Walton & Maberly, 1854.
Работа Джорджа Буля, предвосхитившая многие идеи, на которых основана современная вероятностная логика. Репринтное издание вышло в 1951 году, Dover Publications, N.Y..

[21] *Boruvka O.* O jistém problému minimalním // Prace mor. prírodověd. spol. v Brně. 1926. Vol. 3. P. 37–58.

Первоисточник алгоритма Борувки. Название переводится как «Об одной задаче минимизации».

[22] *Bost J.-B., Connes A.* Hecke Algebras, Type III factors and phase transitions with spontaneous symmetry breaking in number theory // Selecta Math. (New Series). 1995. Vol. 1. P. 411–457.
О том, как свободный риманов газ может помочь справиться с гипотезой Римана.

[23] *Bouzy B., Cazenave T.* Computer Go: An AI oriented survey // Artificial Intelligence. 2001. Vol. 132, N. 1. P. 39–103.
Обзор компьютерных программ, играющих в го, особенностей их дизайна, применяемых при этом аппаратов искусственного интеллекта.

[24] *Breiger R. L.* The Analysis of Social Networks // Handbook of Data Analysis / Ed. by M. Hardy, A. Bryman. London: Sage Publications, 2004.
Обзор методов анализа социальных сетей.

[25] *Burris S. N., Sankappanavar H. P.* A Course in Universal Algebra. Springer Verlag, 1981.
Книга об универсальной алгебре: теория решёток, теория алгебр, теория булевых алгебр и связь с теорией моделей. Книга бесплатно доступна онлайн: `http://www.math.uwaterloo.ca/ snburris/htdocs/ualg.html`.

[26] The Caml Language Documentation.
`http://caml.inria.fr/resources/doc/index.en.html`.
Документация языка Caml (и Ocaml). Содержит reference manual, tutorials и ссылки на различную литературу.

[27] *Casella G., George E. I.* Explaining the Gibbs sampler // The American Statistician. 1992. Vol. 46. P. 167–174.
Обзор о сэмплировании по Гиббсу.

[28] *Castillo P. A., Arenas M. G., Castellano J. G., Merelo J. J., Prieto A., Rivas V., Romero G.* Lamarckian Evolution and Baldwin Effect in Evolutionary Neural Networks // Primer congreso espanol de Algoritmos Evolutivos y Bioinspirados (AEB02), Merida (Spain). 2002. P. 494–498.
В статье сравниваются два подхода к эволюционным нейронным сетям: ламаркианский подход и подход, использующий эффект Болдуина. Обширная библиография по обоим направлениям. Статья также доступна как препринт arXiv.

[29] *Chailloux E., Manoury P., Pagano B.* Developing Applications with Objective Caml. O'Reilly, 2000.
Подробный учебник и справочник функционального языка программирования Objective Caml. Исходной версией книги является французская, на сайте `http://caml.inria.fr/pub/docs/oreilly-book/` *можно найти её (бесплатный) перевод на английский язык.*

[30] *Chazelle B.* A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity // Journal of the ACM. 2000. Vol. 47. P. 1028–1047.

Алгоритм для поиска минимального остовного дерева, работающий за время, обратное функции Аккермана.

[31] Checkers Is Solved / J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, S. Sutphen // Science. 2007. Vol. 317. P. 1518–1522.

Доказательство того, что игра в шашки на доске 8×8 при правильной игре заканчивается вничью.

[32] *Choquet G.* Étude de certains réseaux de routes // Comptes-rendus de l'Académie des Sciences. 1938. Vol. 206. P. 310–313.

Густав Шоке переоткрывает алгоритм Борувки.

[33] *Covington M. A.* Prolog Programming in Depth. Prentice Hall, 1997. 516 p.

Подробный учебник языка логического программирования Prolog, полезный как для начинающих, так и для опытных программистов.

[34] *Cristianini N., Shawe-Taylor J.* An Introduction to Support Vector Machines and other kernel-based learning methods. Cambridge University Press, 2000.

Классический источник о методе опорных векторов.

[35] *D. M., I. Y.* Bayesian graphical models for discrete data // International Statistical Review. 1995. Vol. 63. P. 215–232.

О сходимости сэмплирования по Гиббсу и некоторых других вопросах.

[36] The Data Mining and Knowledge Discovery Handbook / Ed. by O. Maimon, L. Rokach. Springer, 2005.

Сборник обзорных статей об анализе данных.

[37] *Daume H.* Yet Another Haskell Tutorial.
`http://darcs.haskell.org/yaht/yaht.pdf`.

Классическое введение в язык Haskell. Свободно распространяется, написано подробно и отлично подходит для новичков, желающих изучить Haskell.

[38] *Dempster A. P., Laird N. M., Rubin D. B.* Maximum Likelihood from Incomplete Data via the EM Algorithm // Journal of the Royal Statistical Society. Series B (Methodological). 1977. Vol. 39, N. 1. P. 1–38.

Классическая статья, в которой впервые явно вводится общая конструкция алгоритма EM и доказываются общие свойства его сходимости.

[39] *Domingos P., Pazzani M.* On the optimality of the simple Bayesian classifier under zero-one loss // Machine Learning. 1997. Vol. 29. P. 103–137.

О том, что наивный байесовский классификатор оказывается оптимальным в одной из моделей ошибок.

[40] *Dubois D., Prade H.* Fuzzy Sets and Systems: Theory and Applications. N.Y., London: Academic Press, 1980.
Книга о нечётких множествах и системах, написанная классиками теории доверия.

[41] *Duda R. O., Hart P. E., Stork D. G.* Pattern Classification. Wiley, New York, 2001.
Книга о методах распознавания и классификации образов.

[42] *Feldman R., Sanger J.* The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge University Press, 2006.
Сборник статей об интеллектуальном анализе текстов.

[43] *Fischer J. R.* Prolog :– Tutorial.
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html.
Простое и увлекательное введение в язык Prolog. Автор один за другим разбирает примеры реальных программ, затем переходит к последовательному изложению основ языка.

[44] *Fogel D. B.* Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. IEEE Press, Piscataway, NJ, 2006.
Книга о генетических алгоритмах.

[45] *Fukunaga K.* Statistical Pattern Recognition. Morgan Kaufmann, 1990.
Классическая книга об анализе паттернов.

[46] *Gelly S., Silver D.* Combining Online and Offline Knowledge in UCT // Proceedings of the International Conference of Machine Learning. 2007. P. 273–280.
Статья о последних достижениях MoGo (см. также [47]).

[47] *Gelly S., Wang Y.* Exploration exploitation in Go: UCT for Monte-Carlo Go // Proceedings of NIPS-2006, Online trading between exploration and exploitation. 2006.
Статья о программе MoGo, выведшей компьютерное го на новый уровень.

[48] *Gelman A., Carlin J. B., Stern H. S., Rubin D. B.* Bayesian Data Analysis. London: Chapman and Hall, 1995.
Подробный рассказ об анализе данных с байесовской точки зрения.

[49] *Geman S., Geman D.* Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images // IEEE Transactions on Pattern Analysis and Machine Intelligence. 1984. Vol. 4. P. 721–742.
Первое описание алгоритма сэмплирования по Гиббсу.

[50] *Gini C.* Variabilitá e mutabilitá // Memorie di metodologica statistica / Ed. by E. Pizetti, T. Salvemini. New York, NY, USA: Rome: Libreria Eredi Virgilio Veschi, 1955.
Приводится ссылка на перепечатку 1955 года; оригинальная статья вышла в 1912.

[51] *Goldreich O.* Foundations of Cryptography. Basic Tools. Cambridge University Press, 2001.

Первая часть фундаментальной монографии классика современной криптографии. На данный момент — самый полный источник по криптографии; но изложение достаточно тяжёлое.

[52] *Goldreich O.* Foundations of Cryptography II. Basic Applications. Cambridge University Press, 2004.

Вторая часть монографии [51].

[53] *Goldwasser S., Bellare M.* Lecture notes on cryptography. Summer course on cryptography at MIT, 2001.

Широко известные конспекты лекций Голдвассер–Белларе по криптографии.

[54] *Graham P.* Beating the Averages. http://www.paulgraham.com/avg.html.

Пол Грэм предлагает свою версию того, как стать хорошим программистом — нужно понимать концепции функционального программирования.

[55] *Green F.* An oracle separating $\oplus$P from $\mathrm{PP}^{\mathrm{PH}}$ // Information Processing Letters. 1991. Vol. 37. P. 149–153.

Нижняя оценка для сложности реализации перцептронами функции PARITY ($x_1 \oplus \ldots \oplus x_n$). Впрочем, перцептроны здесь перцептронами не называются.

[56] *Green F.* A Lower Bound for Monotone Perceptrons // Mathematical Systems Theory. 1995. Vol. 28, N. 4. P. 283–298.

Нижняя оценка для сложности монотонных перцептронов глубины 3 (реализуется функцией PARITY, т.е. XOR'ом большого количества переменных).

[57] *Halpern J. Y.* Reasoning about uncertainty. Cambridge, MA: MIT Press, 2003.

Книга о логиках, работающих с неопределённостью, аппаратах её представления и тому подобных вопросах.

[58] *Han E., Kumar V., Shekhar S., Ganesh M., Srivastava J.* Search framework for mining classification decision trees: Tech. Rep. TR-96-023: Department of Computer Science, University of Minnesota, Minneapolis, 1996.

Статья о применении метода деревьев принятия решений.

[59] *Han J., Kamber M.* Data Mining: Concepts and Techniques. Morgan Kaufmann, 2006.

Книга о методах анализа данных.

[60] *Hand D., Mannila H., Smyth P.* Principles of Data Mining. MIT Press, 2001.

Учебник, посвящённый интеллектуальному анализу данных.

[61] *Hand D. J., Yu K.* Idiot's Bayes - not so stupid after all? // International Statistical Review. 2001. Vol. 69, N. 3. P. 385–399.
О статистических свойствах наивного байесовского классификатора.

[62] The Handbook of Data Mining / Ed. by N. Ye. Lawrence Erlbaum Associates, Inc., 2004.
Сборник обзоров об анализе данных.

[63] *Hastie T., Tibshirani R., Friedman J.* The Elements of Statistical Learning. Springer, 2001.
Подробная и интересная книга о статистическом подходе к машинному обучению.

[64] *Haykin S.* Neural networks: A Comprehensive Foundation. Prentice-Hall, 1999.
Подробный учебник о нейронных сетях.

[65] *Hebb D. O.* The Organization of Behavior: A Neuropsychological Theory. Lawrence Erlbaum, 2002. 336 p.
Книга, заложившая основы нейропсихологии и предложившая адекватную теорию обучения нейронов. Приводятся данные переиздания 2002 г., оригинальное издание выпущено в 1949 г.

[66] *Holland J.* Adaptation in Natural and Artificial Systems. Ann Arbor, Michigan, 1975.
В книге основоположника генетических алгоритмов излагаются основные идеи этого раздела искусственного интеллекта. Второе издание этой книги вышло в 1992 году в издательстве MIT Press.

[67] *Hopfield J. J.* Neural networks and physical systems with emergent collective computational abilities // Proceedings of the National Academy of Sciences of the USA. 1984. Vol. 79, N. 8. P. 2554–2558.
Работа, в которой Хопфилд ввёл свои сети.

[68] *Hudak P.* The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge University Press, 2000.
Классический учебник по языку Haskell, в котором описываются многие практические аспекты его применения.

[69] *Hudak P., Hughes J., Jones S. P., Wadler P.* Some Methods for classification and Analysis of Multivariate Observations // Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability. Berkeley, University of California Press, 1967. P. 281–297.
Появление алгоритма k*–средних.*

[70] *Hudak P., Hughes J., Jones S. P., Wadler P.* A History of Haskell: Being Lazy with Class // The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III). San Diego, California, 2007.
История языка Haskell, его основные свойства и новизна.

[71] *Huffman W. C., Pless V.* Fundamentals of Error-Correcting Codes. Cambridge University Press, 2003.
Введение в теорию кодирования.

[72] *Hutton G.* Programming in Haskell. Cambridge University Press, 2007.
Один из самых свежих и лучших учебников по языку Haskell.

[73] *Ikonomakis M., Kotsiantis S., Tampakas V.* Text Classification Using Machine Learning Techniques // WSEAS Transactions on Computers. 2005. Vol. 4, N. 8. P. 966–974.
Подробный обзор алгоритмов классификации текстов.

[74] *Izhikevich D. M.* Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting. Cambridge, MA: MIT Press, 2007.
Монография о моделировании нейронов и нейронных сетей динамическими системами; в биологии этот метод сейчас доминирует.

[75] *Jain A. K., Murty M. N., Flynn P. J.* Data clustering: a review // ACM Computing Surveys. 1999. Vol. 31, N. 3. P. 264–323.
Отличный (хотя сейчас уже немного устаревший) обзор методов кластеризации.

[76] *Johnston D., Wu S.* Foundations of Cellular Neurophysiology. Cambridge, MA: MIT Press, 1997.
Учебник по молекулярной нейрофизиологии; глава 6 рассказывает о биологических моделях нейронов.

[77] *Kaelbling L. P., Littman M. L., Moore A. W.* Reinforcement Learning: A Survey // Journal of Artificial Intelligence Research. 1996. Vol. 4. P. 237–285.
Подробный и хорошо написанный обзор обучения с подкреплением.

[78] *Kanungo T., Mount D. M., Netanyahu N., Piatko C., Silverman R., Wu A. Y.* An Efficient k–means Clustering Algorithm: Analysis and Implementation // IEEE Trans. Pattern Analysis and Machine Intelligence. 2002. Vol. 24. P. 881–892.
Современная статья об алгоритме k–*средних, оценках его эффективности и методах по его улучшению.*

[79] *Kao A., Poteet S.* Natural Language Processing and Text Mining. Springer, 2006.
Книга об анализе текстовой информации на естественных языках.

[80] *Katz J., Lindell Y.* Introduction to Modern Cryptography. Chapman & Hall/Crc, 2007.
Легко читаемый, но вместе с тем математически строгий и достаточно полный учебник по криптографии.

[81] *Keedwell E.* Intelligent Bioinformatics: The Application of Artificial Intelligence Techniques to Bioinformatics Problems. Wiley Interscience, 2005.
Применение искусственного интеллекта к биоинформатике.

[82] *Knuth D.* The TeXbook (Computers and Typesetting, Volume A). Reading, Massachusetts: Addison-Wesley, 1984.
Первоисточник всякой информации о системе TEX.

[83] *Knuth D.* TeX: The Program (Computers and Typesetting, Volume B). Reading, Massachusetts: Addison-Wesley, 1986.
Второй том издания Дональда Кнута о языке TEX(первый том — знаменитая TeXbook *[82]). Содержит полностью документированный исходный код TEX.*

[84] *Knuth D.* The Art of Computer Programming. Volume 1: Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley, 1997.
Третье издание первого тома.

[85] *Knuth D.* The Art of Computer Programming. Volume 2: Seminumerical Algorithms. Reading, Massachusetts: Addison-Wesley, 1997.
Третье издание второго тома.

[86] *Knuth D.* The Art of Computer Programming. Volume 3: Sorting and Searching. Reading, Massachusetts: Addison-Wesley, 1998.
Второе издание третьего тома.

[87] *Koch C., Segev I.* Methods in Neuronal Modeling, 2. Cambridge, MA: MIT Press, 1998.
Монография о биологических методах моделирования нейронов.

[88] *Kohane I. S., Kho A., Butte A. J.* Microarrays for an Integrative Genomics. MIT Press, 2002.
Книга об основных комбинаторных объектах биоинформатики — микромассивах, содержащих генетическую информацию.

[89] *Kohonen T.* Self-organized formation of topologically correct feature maps // Biological Cybernetics. 1982. Vol. 43. P. 59–69.
Основополагающая работа Кохонена о самоорганизующихся картах.

[90] *Kohonen T.* Self-Organizing Maps. Springer, Berlin, Heidelberg, New York, 2001.
Третье, расширенное издание монографии Тейво Кохонена о своих самоорганизующихся картах.

[91] *Konchady M.* Text Mining Application Programming (Programming Series). Charles River Media, 2006.
Книга для программистов о том, как, собственно, реализовать на практике системы анализа неструктурированного текста.

[92] *Kondratoff Y., Michalski R. S.* Machine Learning: An Artificial Intelligence Approach, Volume III. Morgan Kaufmann, 1990.
Третья книга четырёхтомника [107].

[93] *Kotsiantis S., Pintelas P.* Recent Advances in Clustering: A Brief Survey // WSEAS Transactions on Information Science and Applications. 2004. Vol. 1, N. 1. P. 73–81.

Небольшой обзор современных алгоритмов кластеризации.

[94] *Koza J.* Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.
Первая книга Джона Козы о генетическом программировании.

[95] *Koza J.* Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, 1994.
Вторая книга Джона Козы о генетическом программировании.

[96] *Koza J., Bennett III F. H., Andre D., Keane M. A.* Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, 1999.
Третья книга Джона Козы о генетическом программировании.

[97] *Koza J., Keane M. A., Streeter M. J., Mydlowec W., Yu J., Lanza G.* Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, 2003.
Четвёртая книга Джона Козы о генетическом программировании.

[98] *Koza J. R., Bennett III F. H., Andre D., Keane M. A.* Genetic Programming: Biologically Inspired Computation that Creatively Solves Non-Trivial Problems // Proceedings of DIMACS Workshop on Evolution as Computation / Ed. by L. Landweber, E. Winfree, R. Lipton, S. Freeland. Princeton University: Springer-Verlag, 11–12 1999.
Статья, в которой описаны успехи генетического программирования.

[99] *Langford J.* Tutorial on practical prediction theory for classification // Journal of Machine Learning Research. 2005. Vol. 6. P. 273–306.
Подробный обзор байесовской теории классификаторов.

[100] Association of LISP users. http://lisp.org/alu/home.
Ассоциация пользователей языка LISP.

[101] *MacKay D. J.* Information Theory, Inference and Learning Algorithms. Cambridge University Press, 2003.
Отличная книга о машинном обучении. Излагается в основном байесовский подход, всё в высшей степени математически строго и законченно.

[102] *Maron M. E.* Automatic Indexing: An Experimental Inquiry // Journal of the ACM. 1961. Vol. 8, N. 3. P. 404–417.
Одно из первых появлений наивного байесовского классификатора.

[103] *McAllester D.* PAC-Bayesian stochastic model selection // Machine Learning. 2003. Vol. 51. P. 5–21.
Решение вопроса о вероятностной аппроксимационной полноте для гиббсовского классификатора и схожих моделей.

[104] *McCarthy J.* Recursive functions of symbolic expressions and their computation by machine // Communications of the ACM. 1960. Vol. 3, N. 40. P. 184–195.

Основополагающая работа Джона Маккарти о том, как автоматизировать лямбда-исчисление (фактически, о LISP).

[105] *McCarthy J.* Circumscription: A form of non-monotonic reasoning // Artificial Intelligence. 1980. Vol. 13, N. 1–2. P. 23–79.
Работа Маккарти о новом виде немонотонного рассуждения.

[106] *McCarthy J., Minsky M. L., Rochester N., Shannon C. E.* A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence. http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html. 1955.
Заявка на первый workshop по искусственному интеллекту.

[107] *Michalski R. S., Carbonell J. G., Mitchell T. M.* Machine Learning: An Artificial Intelligence Approach. Tioga Publishing Company, 1983.
Вторая книга подробнейшего четырёхтомника о машинном обучении.

[108] *Michalski R. S., Carbonell J. G., Mitchell T. M.* Machine Learning: An Artificial Intelligence Approach, Volume II. Morgan Kaufmann, 1986.
Вторая книга четырёхтомника [107].

[109] *Michalski R. S., Tecuci G.* Machine Learning: A Multistrategy Approach, Volume IV. Morgan Kaufmann, 1994.
Четвёртая книга четырёхтомника [107].

[110] *Minsky M.* Steps toward Artificial Intelligence // Proceedings of the IRE. 1961. Vol. 49, N. 1. P. 8–30.
Крайне интересная «постановочная» статья от одного из создателей искусственного интеллекта как области знания, где он обрисовывает основные задачи, стоящие перед AI.

[111] *Minsky M., Papert S.* Perceptrons. MIT Press, 1969.
Одна из первых классических книг по искусственному интеллекту, посвящённая искусственным нейронным сетям. Существует также расширенное переиздание 1988 года.

[112] *Mitchell M.* An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, 1996.
Учебное пособие о генетических алгоритмах.

[113] *Mitchell T. M.* Machine Learning. McGraw Hill, 1997. 414 p.
Один из основных источников о машинном обучении.

[114] *Mount D. W.* Bioinformatics: Sequence and Genome Analysis. Spring Harbor Press, 2002.
Подробное изложение основ биоинформатики, постановка её основных задач.

[115] *Myers B.* Computer Go.
http://www.intelligentgo.org/en/computer-go/overview.html.
Небольшая популярная статья, описывающая основные трудности создания играющих в го программ.

[116] Network Analysis: Methodological Foundations / Ed. by U. Brandes, T. Erlebach. Berlin, Heidelberg: Springer-Verlag, 2005.
Сборник статей об анализе социальных сетей.

[117] *Nguen H. T., Walker E. W.* A First Course in Fuzzy Logic. N.Y., London, Washington: Chapman & Hall/CRC, 2000. 373 p.
Один из лучших современных учебников по нечёткой логике; изложение выстроено математически очень строго и логично.

[118] *Norvig P.* Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.
Учебник по искусственному интеллекту с многочисленными практическими примерами. Более того, все примеры в нём написаны на языке LISP.

[119] *Oliver J. J.* Decision Graphs — an extension of decision trees // 4^{th} International Conference on Artificial Intelligence and Statistics. 1993. P. 343–350.
Описание графов принятия решений.

[120] *Oliver J. J., Dowe D. L., Wallace C. S.* Inferring Decision Graphs using the Minimum Message Length Principle // 5^{th} Australian Joint Conference on Artificial Intelligence. 1992. P. 361–367.
Графы принятия решений связаны с принципом минимальной длины описания.

[121] *Oliver J. J., Wallace C. S.* Inferring Decision Graphs // International Joint Conference on Artificial Intelligence, Workshop 8. 1991.
Введены графы принятия решений.

[122] *Peng J., Williams R. J.* Incremental Multi-Step Q–learning // Machine Learning. 1996. Vol. 22. P. 283–290.
Вариация алгоритма Q–обучения.

[123] *Peter T.* Myths and Legends of the Baldwin Effect // Proceedings of the Workshop on Evolutionary Computation and Machine Learning at the 13th International Conference on Machine Learning. 1996. P. 135–142.
В статье даётся общая идеология эффекта Болдуина и развеиваются некоторые мифы о нём, которые к 1996 году существовали среди специалистов по машинному обучению.

[124] *Peters J., Schaal S.* Applying the Episodic Natural Actor-Critic Architecture to Motor Primitive Learning // Proceedings of the 2007 European Symposium on Artificial Neural Networks (ESANN). 2007.
Об алгоритме Episodic Natural Actor-Critic.

[125] *Peters J., Schaal S.* Learning to Control in Operational Space // The International Journal of Robotics Research. 2008. Vol. 27, N. 2. P. 197–212.
О методах обучения с подкреплением в роботике.

[126] *Peters J., Schaal S.* Natural Actor-Critic // Neurocomputing. 2008. Vol. 71, N. 7–9. P. 1180–1190.
Об алгоритме Natural Actor-Critic.

[127] *Peters J., Vijayakumar S., Schaal S.* Reinforcement Learning for Humanoid Robotics // Humanoids2003, 3rd IEEE-RAS International Conference on Humanoid Robots. 2003.
О том, как обучение с подкреплением применяется в роботике.

[128] *Peters J., Vijayakumar S., Schaal S.* Scaling Reinforcement Learning Paradigms for Motor Learning // Proceedings of the 10th Joint Symposium on Neural Computation (JSNC 2003). 2003.
О том, как адаптировать методы обучения с подкреплением к пространству большой размерности для задач роботики.

[129] *Pevzner P. A.* Computational Molecular Biology: An Algorithmic Approach. MIT Press, 2000.
Книга об алгоритмах молекулярной биологии.

[130] *Quinlan J. R.* Induction of decision trees // Machine Learning. 1986. Vol. 1, N. 1. P. 81–106.
Статья, в которой впервые появился алгоритм ID3.

[131] *Raileanu L. E., Stoffel K.* Theoretical Comparison between the Gini Index and Information Gain Criteria // Ann. Math. Artif. Intell. 2004. Vol. 41, N. 1. P. 77–93.
Создана теоретическая база для сравнения критериев выбора оптимального критерия в деревьях принятия решений и сходных аппаратах. Эта база применена для сравнения GINI Index и Gain Ratio.

[132] *Ring M.* Continual Learning in Reinforcement Environments. 1994.
Диссертация содержит подробный обзор методов обучения с подкреплением.

[133] *Robert C. P., Casella G.* Monte Carlo Statistical Methods. New York: Springer-Ferlag, 2004.
Монография о методах Монте-Карло в анализе данных.

[134] *Rohas R.* Neural Networks — A Systematic Introduction. Springer-Verlag, Berlin, New-York, 1996.
Хорошая книга о нейронных сетях; в частности, об ассоциативной памяти и сетях Хопфилда.

[135] *Rummery G.* Problem Solving With Reinforcement Learning. 1995.
О применениях обучения с подкреплением.

[136] *Salton G., Buckley C.* Term-weighting approaches in automatic text retrieval // Information Processing and Management. 1988. Vol. 24, N. 5. P. 513–523.
Обзор нескольких методов взвешивания слов в анализе текстов.

[137] *Salton G., McGill M. J.* Introduction to Modern Information Retrieval. McGraw-Hill, 1983.
Классический источник об information retrieval.

[138] *Saygin A. P., Cicekli I., Akman V.* Turing Test: 50 years later // Minds and Machines. 2000. Vol. 10. P. 463–518.
Отличный обзор о тесте Тьюринга, деталях его формулировки и работе, которая была над ним проведена за первые 50 лет его существования.

[139] *Schaal S.* Learning Robot Control // The Handbook of Brain Theory and Neural Networks, 2nd edition. MIT Press, 2002. P. 983–987.
Глава из книги, посвящённая методам современной роботики.

[140] *Schröder B. S. W.* Ordered Sets: An Introduction. Boston: Birkhauser, 2003.
Книга о теории упорядоченных множеств.

[141] *Sebastiani F.* Machine learning in automated text categorization // ACM Computing Surveys. 2002. Vol. 34, N. 1. P. 1—47.
Подробный и очень интересный обзор о том, как методы машинного обучения помогают для классификации текстовых документов.

[142] *Seeger M.* PAC-Bayesian generalization bounds for gaussian processes // Journal of Machine Learning Research. 2002. Vol. 3. P. 233–269.
Оценки на сходимость гауссовских процессов; обобщение [103].

[143] *Shannon C. E.* A Symbolic Analysis of Relay and Switching Circuits: Ph.D. thesis / Massachusetts Institute of Technology, Dept. of Electrical Engineering. 1940.
Дипломная работа Клода Шеннона, заложившая основы архитектуры современных компьютеров.

[144] *Shannon C. E.* A Mathematical Theory of Communication // Bell System Technical Journal. 1948. Vol. 27, N. 4. P. 379–423, 623–656.
Основополагающая работа Клода Шеннона, в которой он развивает свою теорию информации.

[145] *Spärck Jones K.* A statistical interpretation of term specificity and its application in retrieval // Journal of Documentation. 1972. Vol. 28, N. 1. P. 11–21.
Первоисточник меры похожести документов tf-idf.

[146] *Spector D.* Supersymmetry and the Moebius inversion function // Communications in Mathematical Physics. 1990. Vol. 127. P. 239–252.
О функции Мёбиуса в контексте свободного риманова газа.

[147] *Steele G. L., Gabriel R. P.* The evolution of Lisp // History of programming languages—II. New York, NY, USA: ACM Press, 1996. P. 233–330.
История языка программирования LISP.

[148] *Stigler S. M.* Who Discovered Bayes's Theorem? // The American Statistician. 1983. Vol. 37, N. 4. P. 290–296.

Историк статистики утверждает, что теорему Байеса раньше открыл Николь Саундерсон (Nicholas Saunderson).

[149] *Strehl A. L., Li L., Wiewiora E., Langford J., Littman M. L.* PAC model-free reinforcement learning // ICML '06: Proceedings of the 23rd international conference on Machine learning. 2006. P. 881–888.

Вариант Q-обучения с более хорошими PAC-оценками.

[150] *Sutton R. S., Barto A. G.* Reinforcement Learning: An Introduction. Cambridge, MA, USA: MIT Press, 1998.

Книга об обучении с подкреплением.

[151] *Suzuki R., Arita T.* The Baldwin Effect Revisited: Three Steps Characterized by the Quantitative Evolution of Phenotypic Plasticity // Seventh European Conference on Artificial Life. 2003. P. 395–404.

Конкретное описание построения генетического алгоритма, элементы популяции которого — нейронные сети, а обучение использует эффект Болдуина.

[152] *Tan P. J., Dowe D. L.* MML Inference of Decision Graphs with Multi-Way Joins and Dynamic Attributes // 16^{th} Australian Joint Conference on Artificial Intelligence. 2003. P. 269–281.

Графы принятия решений ещё более расширяются по сравнению с первоначальным вариантом [119].

[153] *Tan P. J., Dowe D. L.* MML Inference of Oblique Decision Trees // Lecture Notes in Artificial Intelligence. 2004. Vol. 3339. P. 1082–1088.

Графы принятия решений, в листьях которых проводится дальнейшая классификация при помощи support vector machines.

[154] *Tesauro G.* Practical issues in temporal difference learning // Machine Learning. 1992. Vol. 8. P. 257–277.

Подробная работа о базовых принципах TD-Gammon.

[155] *Tesauro G.* TD-Gammon, a self-teaching backgammon program, achieves masterlevel play // Neural Computation. 1994. Vol. 6, N. 2. P. 215–219.

Краткое сообщение о TD-Gammon — программе-чемпионе по игре в нарды.

[156] *Tesauro G.* Temporal difference learning and TD-Gammon // Communications of the ACM. 1995. Vol. 38, N. 3. P. 58–67.

Более подробный рассказ о TD-Gammon.

[157] *Theodoridis S., Koutroumbas K.* Pattern recognition. Elsevier, 2006.

Книга о методах распознавания образов.

[158] *Touretzky D. S.* Common LISP: a gentle introduction to symbolic computation. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1990.

Хорошее введение в язык программирования LISP.

[159] *Turing A.* Computing Machinery and Intelligence // Mind. 1950. Vol. 59(236). P. 433–460.
Статья, в которой Алан Тьюринг вводит основные положения искусственного интеллекта, в том числе тест Тьюринга и основные идеи генетического программирования.

[160] *Wasserman S., Faust K.* Social Networks Analysis: Methods and Applications. Cambridge: Cambridge University Press, 1994.
Подробная книга об анализе социальных сетей.

[161] *Waterman M. S.* Introduction to Computational Biology: Sequences, Maps and Genomes. CRC Press, 1995.
Учебник по биоинформатике.

[162] *Weiss S. M., Kulikowski C. A.* Computer Systems That Learn. San Mateo, CA: Morgan Kauffman, 1990.
Популярная и хорошая книга о машинном обучении.

[163] *Williams R. J., Baird L. C. I.* Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems: Tech. Rep. NU-CCS-93-11: Boston: Northeastern University, College of Computer Science, 1993.
Немного обобщённый анализ обучения с подкреплением.

[164] *Witten I. H., Frank E.* Data Mining: practical machine learning tools and techniques. Morgan Kaufmann, 2005.
Книга о методах машинного обучения, применяемых в анализе данных.

[165] *Xu R., Wunsch D. I. I.* Survey of clustering algorithms // IEEE Transactions on Neural Networks. 2005. Vol. 16, N. 3. P. 645–678.
Подробный обзор алгоритмов кластеризации.

[166] *Zhang H.* The Optimality of Naive Bayes // FLAIRS Conference / Ed. by V. Barr, Z. Markov. 2004.
О том, как наивный байесовский классификатор оказывается оптимальным в не самых сильных предположениях.

[167] *Боровиков В. П.* Искусство анализа данных. СПб., ПИТЕР, 2005.
Книга о data mining по-русски.

[168] *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы: Учебное пособие. 2-е изд. М.: Физматлит, 2006.
Учебное пособие, в котором полно и интересно изложена теория генетических алгоритмов.

[169] *Грэм Р., Кнут Д., Паташник О.* Конкретная математика. Основание информатики. М.: "Мир 1998.

Одна из самых интересных книг о дискретной математике; читается не хуже любой беллетристики. Приводятся выходные данные русского перевода.

[170] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М., МЦНМО, 2004.

[171] *Мартин Н., Ингленд Д.* Математическая теория энтропии. М.: Мир, 1988. 350 с.

Книга, в которой последовательно излагается теория энтропии с точки зрения теории вероятностей, в том числе непрерывной.

[172] О положении в биологической науке. Стенографический отчёт сессии Всесоюзной Академии Сельскохозяйственных Наук им. В. И. Ленина / Под ред. В. Н. Столетов, А. М. Сиротин, Г. К. Объедков. Государственное издательство сельскохозяйственной литературы, 1948.

Сборник стенограмм, среди которых и знаменитая речь Лысенко.

[173] *Пенроуз Р.* Новый ум короля. М., Едиториал УРСС, 2003. 384 с.

Популярная книга об искусственном интеллекте, теории относительности, квантовых вычислениях и многом другом.

[174] *Самсонов Б. Б., Плохов Е. М., Филоненков А. И., Кречет Т. В.* Теория информации и кодирование. Феникс, 2002.

Введение в теорию информации и теорию кодирования.

[175] *Стенли Р.* Перечислительная комбинаторика. М.: Мир, 1990. 440 с.

Классический источник по комбинаторике; рассказывает о многих интересных областях, которые редко попадают в стандартные курсы комбинаторики и дискретной математики.

[176] *Тулупьев А. Л., Николенко С. И., Сироткин А. В.* Байесовские сети: логико-вероятностный подход. СПб.: Наука, 2006. 608 с.

Книга подробно рассматривает байесовские сети доверия и алгебраические байесовские сети, алгоритмы вывода на них, сравнивает два аппарата.

[177] *Уткин Л. В.* Анализ риска и принятие решений при неполной информации. СПб.: Наука, 2007.

Монография о методах анализа рисков. В книге, в частности, даётся очень интересный подход к изложению теории Демпстера–Шефера на языке случайных множеств.

[178] *Феллер В.* Введение в теорию вероятностей и её приложения. М.: Мир, 1984.

Один из наиболее полных трудов по теории вероятностей.

[179] *Ширяев А. Н.* Вероятность. М.: МЦНМО, 2007. 552 с.

Классический русскоязычный источник по теории вероятностей.

Научное издание

Утверждено к печати Учёным советом
Санкт-Петербургского института информатики и автоматизации РАН

Сергей Игоревич Николенко
Александр Львович Тулупьев

Самообучающиеся системы

Оригинал-макет подготовлен в издательской системе
LaTeX2ε

Художник:	*Ефремова Н. С.*
Редактор:	*Коноваленко Е. А.*
Компьютерная вёрстка:	*Николенко С. И.*

Подписано в печать 20.04.09 г. Формат 60×90/16. Бумага офсетная.
Печать офсетная. 18 печ. л. Тираж 1000 экз. Зак. № .

Издательство Московского центра непрерывного математического образования. 119002, Москва, Большой Власьевский пер., 11.
Тел. (499) 241-74-83.

Отпечатано с готовых диапозитивов в ППП «Типография “Наука”».
121099, Москва, Шубинский пер., 6.

Книги издательства МЦНМО можно приобрести в магазине «Математическая книга», Москва, Большой Власьевский пер., 11.
Тел. (499) 241-72-85. E-mail: biblio@mccme.ru