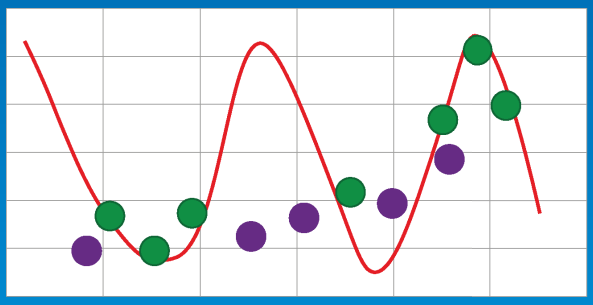
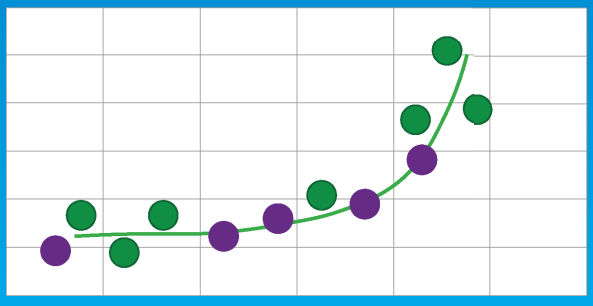
Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина
Институт радиоэлектроники и информационных технологий — РТФ

А.Ю. ДОЛГАНОВ
М.В. РОНКИН
А.В. СОЗЫКИН

БАЗОВЫЕ АЛГОРИТМЫ МАШИННОГО ОБУЧЕНИЯ НА ЯЗЫКЕ PYTHON

Учебно-методическое пособие





Министерство науки и высшего образования
Российской Федерации

Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

А. Ю. Долганов, М. В. Ронкин, А. В. Созыкин

БАЗОВЫЕ АЛГОРИТМЫ МАШИННОГО ОБУЧЕНИЯ НА ЯЗЫКЕ PYTHON

Учебно-методическое пособие

Рекомендовано методическим советом
Уральского федерального университета
для студентов вуза, обучающихся по направлениям подготовки
09.03.01, 09.04.01 — Информатика и вычислительная техника,
09.03.03, 09.04.03 — Прикладная информатика,
09.03.04, 09.04.04 — Программная инженерия,
09.04.02 — Информационные системы и технологии

Екатеринбург
Издательство Уральского университета
2023

УДК 004.85(075.8)
ББК 32.813я73
Д64

Рецензенты:
канд. физ.-мат. наук, заведующий отделом вычислительных систем Института математики и механики им. Н. Н. Красовского УрО РАН *А. М. Григорьев*;
канд. физ.-мат. наук, руководитель исследовательского центра ООО «Сайберлимфа» *Ю. Ю. Чернышов*

Научный редактор — д-р физ.-мат. наук, проф. *Д. Б. Берг*

Долганов, Антон Юрьевич.
Д64 Базовые алгоритмы машинного обучения на языке Python : учебно-методическое пособие / А. Ю. Долганов, М. В. Ронкин, А. В. Созыкин ; М-во науки и высшего образования РФ. — Екатеринбург : Изд-во Урал. ун-та, 2023. — 124 с.

ISBN 978-5-7996-3632-6

Учебно-методическое пособие посвящено изучению основ анализа данных и реализации базовых алгоритмов машинного обучения на языке Python. Целью данного пособия является формирование у студентов теоретических знаний и практических навыков в области базовых алгоритмов машинного обучения, овладение инструментарием, моделями и методами машинного обучения.

Для успешного освоения курса необходимо базовое знание языка программирования Python и высшей математики.

УДК 004.85(075.8)
ББК 32.813я73

ISBN 978-5-7996-3632-6

Оглавление

Предисловие

Целью данного учебно-методического пособия является формирование у студентов теоретических знаний, умений и практических навыков по базовым алгоритмам машинного обучения, овладение студентами инструментарием, моделями и методами машинного обучения.

Книга имеет следующую структуру.

Глава 1 посвящена знакомству с ключевыми понятиями машинного обучения, типами данных, а также с базовыми понятиями линейной алгебры и математического анализа, которые необходимы для дальнейшего освоения учебно-методического пособия.

В главе 2 обсуждаются вопросы, связанные с визуализацией данных, их предварительной обработкой и генерацией признаков. В частности, рассмотрены библиотеки Pandas и Seaborn.

В главе 3 представлена одна из простейших моделей машинного обучения — линейная регрессия. Рассмотрен подход к нахождению весов линейной регрессии «градиентный спуск». Схожие идеи используются для обучения и более сложных нейронных сетей. В качестве практики пошагово реализуется алгоритм линейной регрессии «своими руками».

В главе 4 рассмотрены особенности задачи классификации и ключевые метрики для оценки качества модели классификации. Обсуждается модель логистической регрессии, которая наследует идеи линейной регрессии. В качестве практики пошагово реализуется алгоритм логистической регрессии «своими руками».

В главе 5 обсуждается классический метод уменьшения размерности — метод главных компонент. В качестве практики реализуется метод главных компонент «своими руками».

Глава 6 посвящена особенностям задачи кластеризации и различным подходам к оценке расстояния между объектами. Рассмотрен ба-

зовый алгоритм кластеризации — метод *k*-средних. В качестве практики пошагово реализуется алгоритм кластеризации *k*-средних «своими руками».

В заключении приводятся рекомендации по дальнейшему изучению методов машинного обучения.

В каждой главе приведены контрольные вопросы для самостоятельной оценки усвоения материала. Практические задания к главам включают не только работу с генерируемыми данными, но и с реальным набором данных продаж автомобилей на вторичном рынке для закрепления изученных алгоритмов. Вторичный рынок автомобилей, по нашему мнению, является достаточно хорошо интерпретируемым и понятным каждому читателю набором данных. При этом, как будет далее показано, используемый набор данных представляется достаточно наглядным в отношении решаемых задач. Кроме того, будет полезным отметить, что методически важными критериями этого набора данных являются возможность решения задач как категориального принятия решений (классификации), так и непрерывного (регрессии), а также достаточный объем для демонстрации различных аспектов изучаемых алгоритмов, что оставляет читателю пространство для практических экспериментов и более полного освоения изучаемого материала.

Набор данных продаж автомобилей, а также полные версии листингов программ, используемых в учебно-методическом пособии, доступны в онлайн-хостинге репозиториев GitHub по ссылке *https://github.com/dayekb/Basic_ML_Alg*

Глава 1.
Машинное обучение: общие сведения и понятия

В этой главе мы рассмотрим основные понятия, связанные с машинным обучением: обсудим типы данных, которые обычно обрабатываются, типовые задачи машинного обучения, а также необходимые для освоения данного пособия понятия линейной алгебры и математического анализа.

Ключевые понятия стоит рассмотреть на конкретном примере данных. В табл. 1.1 представлены данные об успеваемости некоторой группы студентов.

Таблица 1.1

Пример данных успеваемости студентов

id студента	Пол	Возраст	Институт	Общежитие	Работа	Оценка Python	ЕГЭ Инф.	Баллы по МО	Экзамен по МО
0	Ж	24	ФТИ	нет	нет	75	83	54	Отл.
1	М	23	Другой	нет	нет	79	40	98	Уд.
2	М	24	Другой	нет	да	43	59	43	Уд.
3	Ж	24	ИРИТ-РТФ	нет	да	98	83	46	Отл.
4	М	24	ИРИТ-РТФ	да	да	50	65	49	Неуд.
5	М	24	ФТИ	да	да	45	96	90	Неуд.
6	Ж	23	ИРИТ-РТФ	нет	нет	71	98	50	Хор.
7	Ж	23	ИРИТ-РТФ	нет	нет	98	43	55	Уд.
8	Ж	24	ИРИТ-РТФ	да	да	49	61	83	Неуд.
9	Ж	24	ИЕНиМ	да	да	63	46	71	Хор.

Первые ключевые понятия и обозначения, которые стоит ввести:

- x — объект, требующий некоторого предсказания (в табл. 1.1 это строка для отдельного студента, характеризуемого собственным id);

- $\mathbb{X}$ — полный набор объектов (в табл. 1.1 это совокупность студентов, все строки);
- y — цель (target), которая является ожидаемым предсказанием (в табл. 1.1 это две целевые переменные для каждого студента: баллы по курсу «Машинное обучение» и оценка за экзамен по машинному обучению);
- $\mathbb{Y}$ — полный набор целей (в табл. 1.1 это целевые переменные для совокупности студентов, оба столбца);
- *признаки* (features) — характеристики объектов (в табл. 1.1 это пол, возраст, институт, общежитие, работа, баллы ЕГЭ по информатике и оценка за курс по Python).

В связи с этим на высоком уровне типичная задача машинного обучения формулируется следующим образом: искать возможности получать целевые переменные при использовании некого признакового пространства данных.

Типы данных

К слову, о признаках и данных. Существуют различные подходы к классификации признаков, назовем их *микроуровень* и *макроуровень*.

На *микроуровне* признаки можно разделить на *числовые* и *категориальные*.

Числовые признаки — это некоторые количественные оценки объектов. Числовые признаки делят на *дискретные*, которые невозможно измерить, но можно посчитать: например, ученики в классе, пальцы, результат в футболе. Выделяют также *непрерывные* данные, которые не могут быть подсчитаны, но их можно измерить — это, например, температура, напряжение, высота.

Для числовых данных используются следующие обозначения:

- $\mathbb{N}$ — натуральные числа;
- $\mathbb{Z}$ — целые числа;
- $\mathbb{Q}$ — рациональные числа;
- $\mathbb{R}$ — действительные числа;
- $\mathbb{C}$ — комплексные числа.

Категориальные признаки — это характеристики объектов. Обычно категориальные признаки делят на *номинальные* (nominal), кото-

рые отвечают на вопрос о том, какое значение принимает данная характеристика, например цвет, пол, язык, институт и т. д., и *порядковые* (ordinal) — дискретные и упорядоченные величины, например уровень английского, итоговая оценка за курс машинного обучения. *Номинальные* признаки, в которых всего два возможных значения, называют *бинарными* (это ответы на такие вопросы, которые предполагают только «да» или «нет»).

На практике категориальные данные могут быть представлены в виде числовых значений, как это изображено на рис. 1.1.

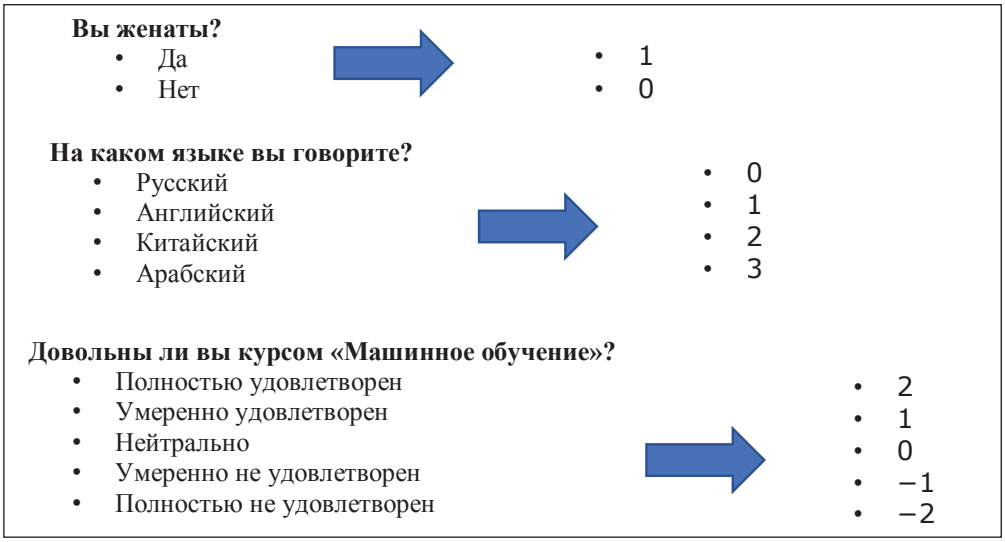


Рис. 1.1. Пример представления категориальных данных в виде числовых значений

Но нужно помнить, что подобные числа не имеют математического значения, как в случае числовых признаков, т. е. их нельзя складывать (например, нельзя сложить английский и китайский и получить арабский) или сравнивать между собой (нельзя сказать, что английский в три раза меньше, чем арабский). Это накладывает определенные ограничения на использование категориальных признаков в моделях машинного обучения.

На *макроуровне* признаки можно разделить на *табличные* и *нетабличные*. Первые — это данные, представленные в виде таблицы (см. табл. 1.1), отображающей совокупность различных числовых и категориальных признаков. В строках представлены различные объекты, в столбцах —

различные признаки. Именно этот тип данных в основном используется, когда речь идет о классических алгоритмах машинного обучения.

К нетабличным данным относятся изображения, временные ряды и естественный язык. Каждый из этих типов данных имеет свои особенности, которые требуют специального подхода. При этом для получения каких-то простых моделей эти данные можно свести к табличным.

Например, можно рассматривать изображение как совокупность интенсивности отдельных пикселей (рис. 1.2). Принято, что большие значения интенсивности соответствуют белому цвету, а небольшие — черному. При этом в зависимости от разрядности изображения можно иметь разные значения оттенков серого между белым и черным. Так, на рис. 1.2 представлено 8-битное изображение (где интенсивность цвета кодируется значением от 0 до 255). Любое изображение можно «спрямить» (flatten), т. е. перейти от двумерного представления к одномерному. Таким образом каждое изображение можно представить в виде большой строки признаков. Например, изображение 10 × 10 пикселей представляется в виде строки из 100 признаков.

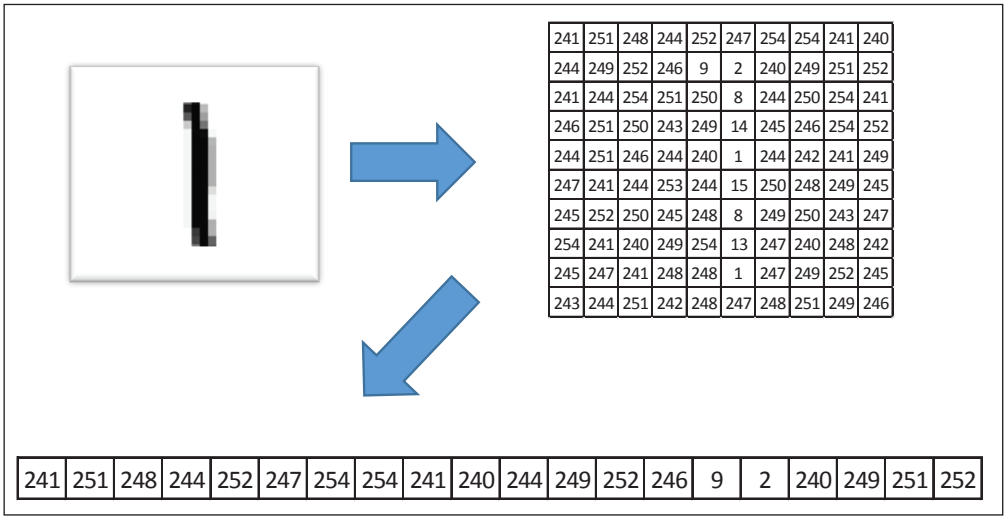


Рис. 1.2. Представление одноканального изображения в виде строки признаков

Однако нужно помнить, что реальные изображения состоят из нескольких цветовых каналов, а не одного, как на рис. 1.2. Наиболее распространена трехканальная цветовая модель RGB: red — красный, green — зеленый, blue — синий. На рисунке 1.3 представлен пример

разложения, пожалуй, самого известного изображения в компьютерном зрении — «Лена» — на три канала. Таким образом, изображение Лены можно представить в виде строки из 512 × 512 × 3 = = 786 432 признаков.

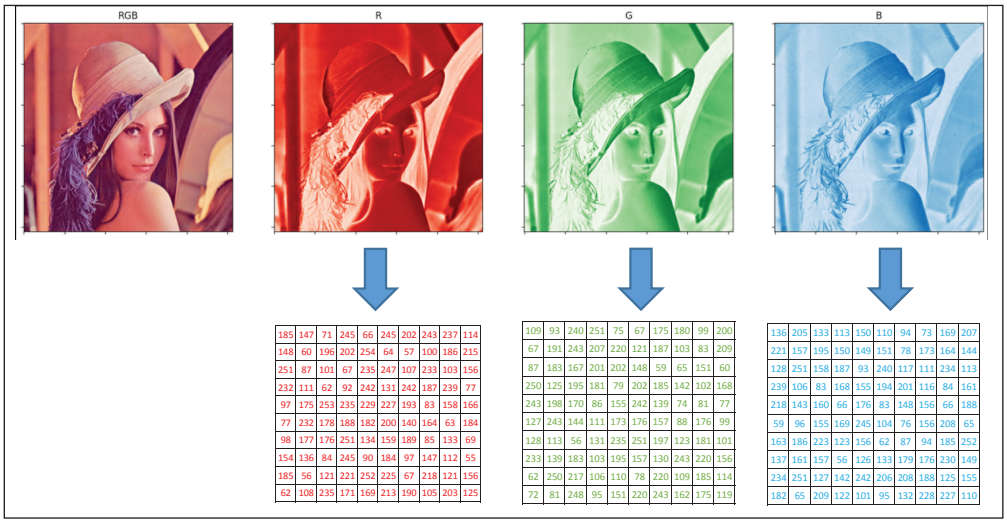


Рис. 1.3. Разложение цветного изображения «Лена» на каналы [1]

Понятно, что такое представление изображения в виде табличных данных не кажется чем-то эффективным. При этом мы должны быть уверены, что в каждом пикселе находится «однотипная» часть изображения: например, если это изображения котов и собак, то для такого подхода они должны быть сняты под одним углом. Поэтому представление изображений в виде таблиц в основном используется только для однотипных данных (например, изображения цифр). А для обработки более сложных изображений в настоящее время самым распространенным подходом являются модели, использующие *сверточные нейронные сети* (Convolutional Neural Networks) [2–4].

В какой-то степени аналогичным способом можно поступать с временными рядами. Временной ряд представляет собой совокупность значений какого-либо измерения в отдельные моменты времени. С одной стороны, можно свести временной ряд к табличным данным. Однако такой подход наивен, если планируется анализировать сигналы разных объектов: мы должны быть уверены в том, что все сигналы «согласованы» по оси времени, но такое редко встречается для реаль-

ных сигналов. Например, сигнал электрокардиографии имеет достаточно сложную структуру (рис. 1.4). При этом информативным является расстояние между последовательными *R*-зубцами.

В этом ключе продуктивным подходом к обработке временных сигналов является расчет признакового пространства. Обработка биомедицинских сигналов и извлечение признакового пространства подробно освещены в учебных пособиях [5] и [6].

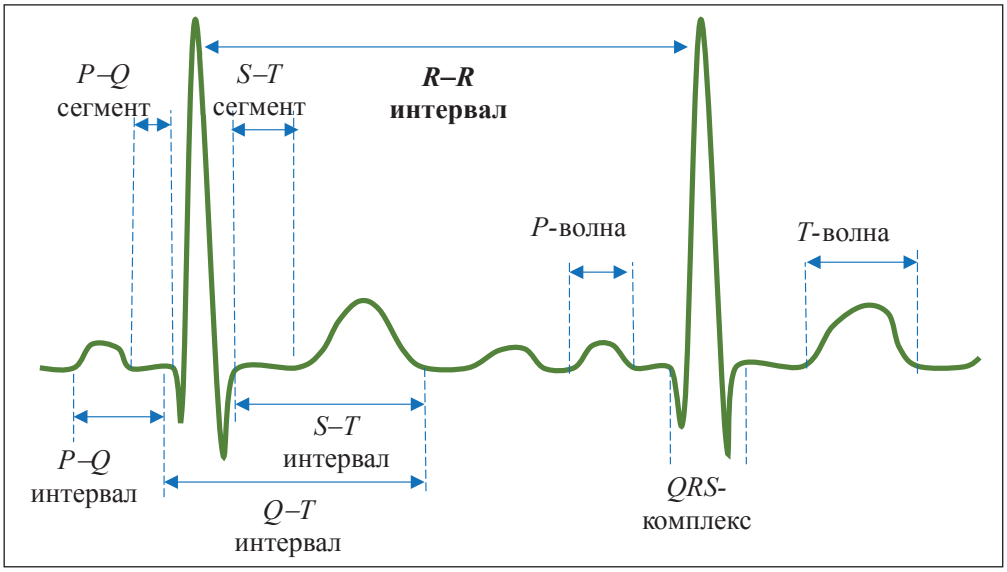


Рис. 1.4. Визуализация сигнала электрокардиограммы

С другой стороны, возможно использовать модели, которые учитывают временную составляющую временных рядов. К таковым относятся модели, использующие *рекуррентные нейронные сети* (Recurrent Neural Network) [7].

Данные естественного языка можно рассматривать в упрощенном виде как совокупность отдельных слов — категориальных признаков. Такие подходы вполне применимы в обобщенных задачах, связанных с оценкой тональности текстов. Другим подходом к решению задач обработки естественного языка является получение векторных представлений слов (Word Embedding) [8; 9]. Однако для более сложных задач, таких как перевод с одного языка на другой, генерация текста, используют так называемые *модели-трансформеры* (Transformers) [10], которые используют механизм *внимание* (Attention) [11].

Обучение модели

В предыдущем разделе мы часто употребляли понятие *модель*. В общем случае модель использует некие операции над признаками для предсказания целевой переменной, т. е. происходит отображение из пространства признаков в пространство целевых предсказаний:

$$a : \mathbb{X} \to \mathbb{Y},$$

где $a \in \mathbb{A}$ — семейство моделей.

Обычно предсказания модели обозначают символом $\hat{y}$. В общем случае справедливо следующее выражение:

$$\widehat{y_i} = a\left(\boldsymbol{x}_i, w, h\right),$$

где x_i — вектор признаков для i-го объекта; w — параметры модели (оптимизируются алгоритмом модели); h — гиперпараметры модели (оптимизируются теми, кто запускает алгоритмы машинного обучения).

Более подробно параметры и гиперпараметры моделей мы рассмотрим в последующих главах.

После того, как мы выбрали модель, ее необходимо обучить. Следующие концепции, которые стоит обсудить, это тренировочная и тестовая выборки. *Тренировочная выборка* — это такой набор данных, для которого нам известны пары «признаки — целевая переменная» для каждого субъекта из выборки. *Тестовая выборка* — это такой набор данных, для которого известны только признаки.

Допустим, у нас есть данные по студентам набора 2021 г. (табл. 1.2), мы знаем все собираемые параметры и целевые переменные. А в 2022 г. поступили новые студенты (табл. 1.3), и мы хотим, используя опыт предыдущего года, попытаться предсказать, у каких студентов могут быть проблемы с курсом «Машинное обучение» и, наоборот, кто из студентов успешно справится, а значит, им можно давать задачи посложнее.

Таблица 1.2

Тренировочная выборка

id студента	Пол	Возраст	Институт	Общежитие	Работа	Оценка Python	ЕГЭ Инф.	Баллы по МО	Экзамен по МО
0	Ж	24	ФТИ	нет	нет	75	83	54	Отл.
1	М	23	Другой	нет	нет	79	40	98	Уд.

Окончание табл. 1.2

id студента	Пол	Возраст	Институт	Общежитие	Работа	Оценка Python	ЕГЭ Инф.	Баллы по МО	Экзамен по МО
2	М	24	Другой	нет	да	43	59	43	Уд.
3	Ж	24	ИРИТ-РТФ	нет	да	98	83	46	Отл.
4	М	24	ИРИТ-РТФ	да	да	50	65	49	Неуд.
5	М	24	ФТИ	да	да	45	96	90	Неуд.
6	Ж	23	ИРИТ-РТФ	нет	нет	71	98	50	Хор.
7	Ж	23	ИРИТ-РТФ	нет	нет	98	43	55	Уд.
8	Ж	24	ИРИТ-РТФ	да	да	49	61	83	Неуд.
9	Ж	24	ИЕНиМ	да	да	63	46	71	Хор.

Таблица 1.3

Тестовая выборка

id студента	Пол	Возраст	Институт	Общежитие	Работа	Оценка Python	ЕГЭ Инф.	Баллы по МО	Экзамен по МО
10	М	23	ИЕНиМ	да	да	51	84	?	?
11	Ж	24	ИЕНиМ	да	нет	90	44	?	?
12	Ж	24	ФТИ	да	нет	60	71	?	?
13	Ж	24	ИЕНиМ	да	нет	55	76	?	?
14	Ж	23	ФТИ	нет	да	40	54	?	?
15	М	24	Другой	нет	да	94	64	?	?
16	М	23	Другой	да	да	65	46	?	?
17	М	24	Другой	нет	нет	50	49	?	?
18	Ж	23	ИРИТ-РТФ	да	да	62	93	?	?
19	М	23	ИЕНиМ	да	да	56	40	?	?

Для того чтобы оценить, насколько плоха или хороша конкретная модель, нам необходимо воспользоваться *функцией потерь* $L\left(y_i, \widehat{y_i}\right)$ для оценки качества модели. В зависимости от типа целевой переменной (числовая или категориальная) функции потери могут отличаться, но в них есть общая суть. Если для конкретного объекта $y_i \sim \widehat{y_i}$, т. е. предсказание модели и реальное значение целевой переменной совпадают, то функция потерь $L\left(y_i, \widehat{y_i}\right)$ принимает небольшие значения; если же предсказание модели и реальное значение разнятся, то функция потерь принимает большие значения.

Используя данные тренировочной выборки, мы можем оценить *функционал потерь* — среднее значение функции потерь для всех объектов из тренировочной выборки:

$$Q(a,\mathbb{X}) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, a\left(x_i, w, h\right)\right).$$

Следовательно, можно сформулировать цель обучения следующим образом:

$$Q(a,\mathbb{X}) \to \min_{a\in\mathbb{A}}.$$

Другими словами, в ходе обучения мы должны подобрать такие параметры и гиперпараметры модели, которые наилучшим образом предсказывают целевые значения в тренировочной выборке.

В общем случае наблюдается закономерность: простые модели (которые содержат ограниченное число признаков, простые зависимости между переменными) обладают большими значениями функционала потерь, в то же время сложные модели (в которых много признаков и существуют сложные зависимости между переменными) могут иметь сколь угодно низкие значения функционала потерь. В общем случае зависимость функционала потерь от сложности модели представлена на рис. 1.5.

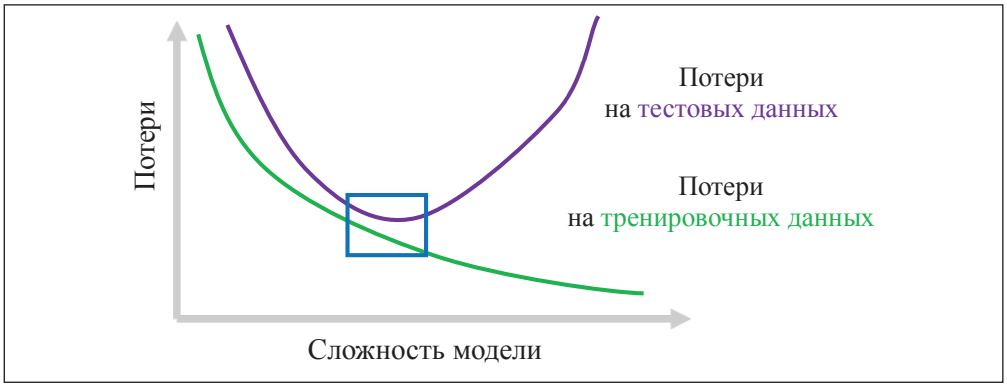


Рис. 1.5. Зависимость потерь от сложности моделей

При усложнении модели потери на тестовых данных, как правило, тоже вначале убывают. Но в определенный момент может возникнуть ситуация, когда потери на тестовой выборке начинают снова расти, а потери на тренировочной выборке продолжают падать. Это явление называется *переобучением*. Мы не находим общие зако-

номерности, а запоминаем тренировочную выборку. Здесь стоит отметить, что практической пользы от модели, которая просто хорошо запоминает тренировочную выборку, немного. Поэтому обычно потери на тренировочных данных рассматривают совместно с потерями на тестовых данных.

Чтобы избежать переобучения, мы должны использовать тестовую выборку для своевременной остановки алгоритма обучения и выбора оптимальных параметров и гиперпараметров модели. Но есть проблема: по умолчанию у нас нет значений целевых переменных для тестовой выборки.

Однако мы можем смоделировать тестовую выборку, используя подход, который называется *валидация*. Мы можем взять «кусочек» тренировочной выборки, отложить его, обучить модель на остальной тренировочной выборке и проверить на отложенном «кусочке». Такой тип валидации называется *использованием отложенной выборки* (Hold-Out Split), рис. 1.6.

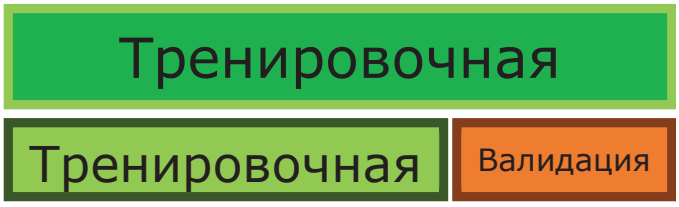


Рис. 1.6. Схема отложенной выборки

Для повышения уверенности в модели можно повторить процесс разбиения выборки на тренировочную и валидационную многократно, по-разному разбивая данные.

Или же можно воспользоваться n-Fold кросс-валидацией (n-Fold Cross-Validation Split). Тренировочная выборка разбивается на n одинаковых по объему частей, которые содержат разные объекты. Производится n итераций. На каждой итерации происходит следующее:

- модель обучается на (n — 1) частях обучающей выборки;
- модель тестируется на части обучающей выборки, которая не участвовала в обучении.

Итоговая оценка функционала потерь усредняется по всем n итерациям. Как правило, $n = 10$ (в случае малого размера выборки — 5). На рисунке 1.7 показан пример с $n = 4$.

Рис. 1.7. Схема 4-Fold кросс-валидации

Такой подход позволяет сымитировать ситуацию, когда модель тестируется на новых, не виденных ранее данных. Поэтому потери на валидационных данных можно использовать для определения оптимальных параметров и гиперпараметров модели (рис. 1.8).

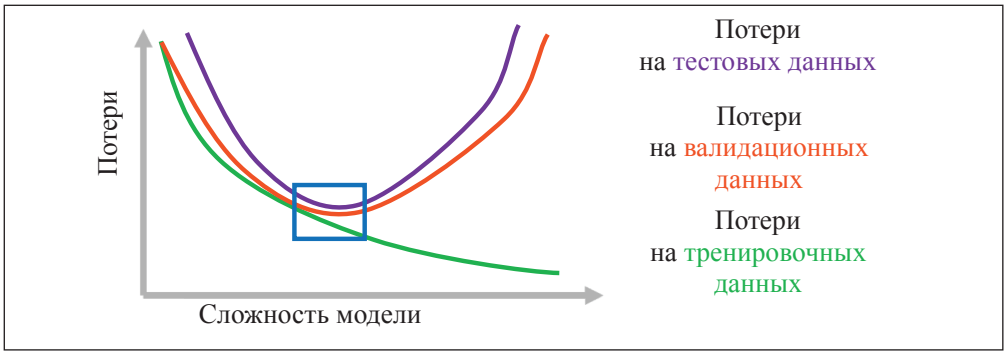


Рис. 1.8. Зависимость потерь от сложности модели
с учетом валидационных данных

Разложение ошибки на смещение и дисперсию

В общем случае функционал потерь можно разложить на следующие составляющие:

$$\text{Loss}(a,\mathbb{X}) \sim \text{Bias}\big(a(\mathbb{X})\big) + \text{Variance}\big(a(\mathbb{X})\big) + \sigma^2,$$

где σ^2 — случайный шум, с которым, к сожалению, ничего нельзя не сделать. В связи с этим нужно иметь в виду, что модели не могут быть совершенными. Однако мы можем уменьшать другие слагаемые в данном разложении.

Ошибка смещения (Bias Error) — это ошибка из-за неверных предположений в алгоритме обучения. Высокая ошибка смещения возникает, если модель слишком проста и не способна отразить закономерности в данных. Ошибку смещения мы можем оценить, используя тренировочные данные.

Дисперсия (Variance) — это ошибка из-за чувствительности к небольшим колебаниям обучающей выборки. Высокая дисперсия возникает, если модель плохо работает на новых данных. Дисперсию модели мы можем оценить с использованием валидационных и тестовых данных.

В зависимости от величины смещения и дисперсии существуют различные ситуации, которые можно описать с помощью мишеней, как указано на рис. 1.9. Здесь близость к центру показывает наименьшие значения функции потерь для конкретного объекта.

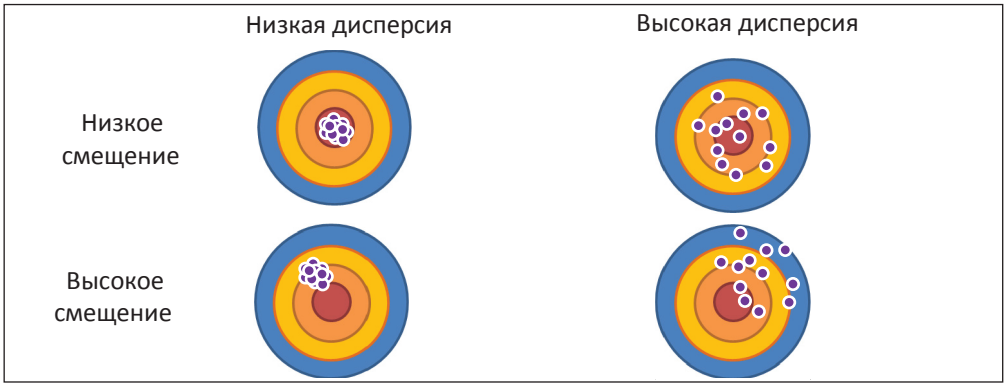


Рис. 1.9. Разложение ошибок модели

Идеальная ситуация — это низкое смещение и низкая дисперсия. Модель хорошо работает в среднем, и при этом для разных данных эта тенденция сохраняется. В случае высокого смещения и низкой дисперсии модель выдает стабильные предсказания как на тренировочной, так и на валидационной выборке, но, к сожалению, эти предсказания далеки от реальных значений. С другой стороны, ситуация низкого

смещения и высокой дисперсии показывает, что модель в среднем работает неплохо, однако существует достаточно большое количество отдельных объектов, на которых предсказания не совпадают с реальными значениями. Наконец, худшая из возможных ситуаций — высокое смещение и высокая дисперсия — говорит о том, что модель совсем не подходит для имеющихся данных.

Идеальные ситуации встречаются редко, и поэтому необходимо находить компромисс между высоким смещением и высокой дисперсией. А это уже зависит от конкретной постановки задачи.

Задачи машинного обучения

Поговорим о типовых задачах машинного обучения. Стандартно задачи машинного обучения разделяют на следующие:

- обучение с учителем (Supervised Learning);
- обучение без учителя (Unsupervised Learning);
- обучение с подкреплением (Reinforcement Learning).

Для *обучения с учителем* характерно использование обучающего набора данных (тренировочной выборки). Для каждого экземпляра из набора данных есть пары «входные данные/признаки — ожидаемый ответ». В этом случае задачей является поиск модели или алгоритма, который предсказывает ожидаемые целевые ответы.

Далее возникают различные ситуации в зависимости от того, на что мы рассчитываем в плане ожидаемых ответов. Если множество возможных ответов конечно, то речь идет о *задаче классификации*:

- $\mathbb{Y} \in \{1, \ldots, K\}$, $K \in \mathbb{Z}$ в случае многих классов;
- $\mathbb{Y} \in \{-1, +1\}$ или $\mathbb{Y} \in \{0, 1\}$ в случае двух классов.

Схематично это представляется следующим образом: есть некоторое пространство признаков, при этом существует заранее известная разметка (раскраска) отдельных объектов. Задача классификации сводится к построению такой разделяющей кривой, которая способна предсказывать метку или класс объекта (рис. 1.10).

В данных из табл. 1.1 оценка за экзамен по машинному обучению является целевой переменной для задачи классификации, т. к. количество возможных ответов конечно.

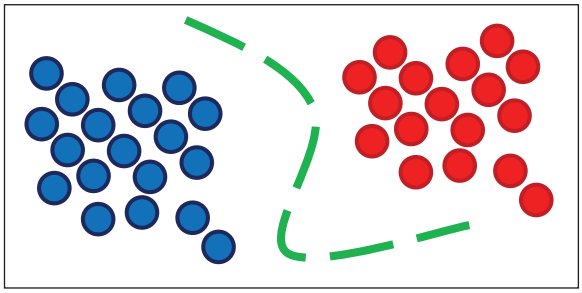


Рис. 1.10. Схематичное представление задачи классификации

С другой стороны, множество возможных ответов может быть почти бесконечным, т. е. $\mathbb{Y} \in \mathbb{R}$. В таком случае решается *задача регрессии* (рис. 1.11). Простой пример: допустим, существует статистика по стоимости квартир и их площадям, но не для всех значений площадей. Используя имеющиеся данные, мы хотим предсказать стоимость квартиры для тех значений площадей, которых нет в тренировочной выборке. Задача регрессии сводится к построению кривой, проходящей через все тренировочные данные.

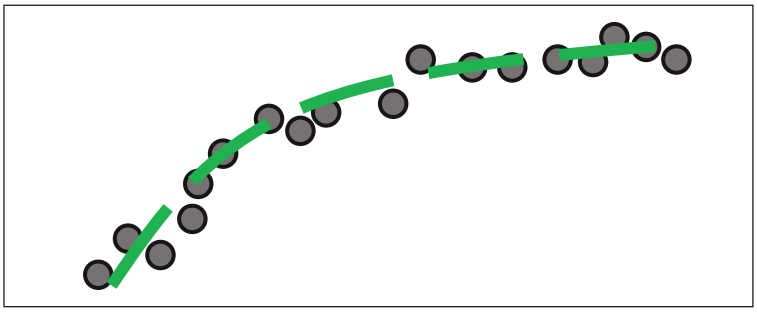


Рис. 1.11. Схематичное представление задачи регресии

В данных из табл. 1.1 балл за текущую успеваемость по машинному обучению может быть целевой переменной для задачи регрессии, т. к. количество возможных ответов бесконечно.

Постановка задачи при *обучении без учителя* затруднена, так как изначально имеются только данные. При этом необходимо найти в этих данных закономерности.

Если требуется разметить принадлежность отдельных объектов к различным кластерам, используя, например, близость или сходство отдельных точек, то это *задача кластеризации* (рис. 1.12).

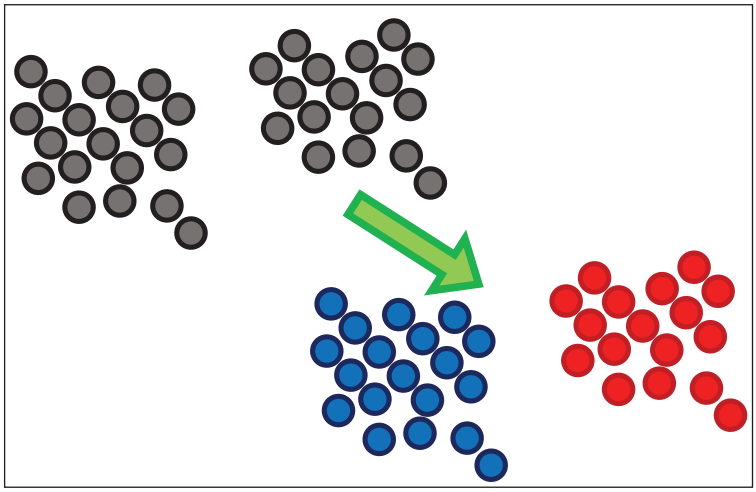


Рис. 1.12. Схематичное представление задачи кластеризации

Стоит отметить, что, в отличие от задачи классификации, в этом случае изначально не известны реальные классы объектов, и алгоритм должен принимать решение исключительно исходя из внутренних закономерностей в данных.

Другой типовой задачей обучения без учителя является *задача снижения размерности*. Пусть имеются табличные данные большой размерности и требуется построить такую новую таблицу данных, которая упрощала бы работу с ними. Тогда решением задачи будет сокращение размерности табличных данных, например, со 100 до 2 или 3, чтобы имеющиеся данные можно было визуализировать.

Обучение с подкреплением предполагает, что существует некая среда и некая система, которая взаимодействует со средой. Задача состоит в том, чтобы сделать это взаимодействие эффективным.

Базовые понятия линейной алгебры

Перед тем как обучать модель и решать задачи, необходимо вспомнить основные понятия из линейной алгебры для манипуляций с данными — в первую очередь *объекты* и *операции*.

Самым простым *объектом* является число, или *скаляр*. Обозначается $x \in \mathbb{R}$. Проще говоря, это одна ячейка в табличных данных.

Совокупность нескольких скаляров называется *вектор*. Обозначается $\boldsymbol{x} \in \mathbb{R}^n$, где n — размерность вектора. В табличных данных выделяют векторы-строки и векторы-столбцы.

Поскольку можно объединить несколько скаляров в вектор, то можно объединить и несколько векторов одинаковой размерности в новый объект, который называется *матрица*. Обозначается как $\boldsymbol{X} = \left[x_{ij} \right]_{m \times n}$ или $\boldsymbol{X} \in \mathbb{R}^{m \times n}$, где m — количество строк, n — количество столбцов. По сути, матрицы и есть таблицы данных.

Наконец, можно объединить и матрицы одинаковой размерности в новые объекты — они называются *тензоры*. Трехмерный тензор обозначается как $\boldsymbol{X} = x_{ijk}$. Мы уже рассматривали в одном из предыдущих разделов данные в виде тензоров — это трехканальное цветное изображение.

В линейной алгебре, по сути, используются те же *операции*, что и в простой школьной алгебре (сложение, вычитание, умножение, деление), но с некоторыми особенностями. Ключевая особенность состоит в том, что нужно внимательно следить за размерностью объектов, над которыми совершаются операции.

Сложение матриц:

$$\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}, \quad \boldsymbol{C} \in \mathbb{R}^{m \times n};$$

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}.$$

Результатом сложения двух матриц, у которых одинаковая размерность, является матрица той же размерности, при этом каждый элемент является суммой соответствующих элементов исходных матриц.

Матрично-скалярное сложение:

$$\boldsymbol{A} \in \mathbb{R}^{m \times n}, \; x \in \mathbb{R}, \quad \boldsymbol{B} \in \mathbb{R}^{m \times n};$$

$$\boldsymbol{B} = \boldsymbol{A} + x.$$

При сложении скаляров и матриц скаляр добавляется к каждому элементу исходной матрицы.

Сложение матрицы и вектора (broadcasting):

$$\boldsymbol{A} \in \mathbf{R}^{m \times n}, \quad \boldsymbol{x} \in \mathbb{R}^n, \quad \boldsymbol{B} \in \mathbb{R}^{m \times n};$$

$$\boldsymbol{B} = \boldsymbol{A} + \boldsymbol{x}.$$

При сложении матрицы и вектора количество столбцов в матрице должно совпадать с размерностью вектора. В данном случае вектор распространяется (от англ. to broadcast) по всем строкам матрицы.

Стоит отметить, что в строгом математическом смысле такой операции нет, однако она реализована в библиотеке NumPy языка Python.

Умножение матрицы на матрицу:

$$\boldsymbol{A} = \left[a_{ij} \right]_{m \times n}, \quad \boldsymbol{B} = \left[b_{ij} \right]_{n \times p}, \quad \boldsymbol{C} = \boldsymbol{AB} = \left[c_{ij} \right]_{m \times p}.$$

Умножение матрицы на матрицу — самое требовательное к размерностям исходных матриц действие. Можно умножать матрицу на матрицу только в том случае, если количество столбцов первой матрицы совпадает с количеством строк второй. При этом размерность итоговой матрицы будет следующей: количество строк будет равно количеству строк первой матрицы, а количество столбцов — количеству столбцов второй матрицы.

При этом каждый элемент итоговой матрицы определяется исходя из следующих соображений: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Кроме того, в общем случае $\boldsymbol{AB} \neq \boldsymbol{BA}$.

Стоит также упомянуть операцию *поэлементного умножения* (произведение Адамара): вычисления производятся по аналогии с поэлементным сложением. Разумеется, размерности исходных матриц должны совпадать:

$$\boldsymbol{A} = \left[a_{ij} \right]_{m \times n}, \quad \boldsymbol{B} = \left[b_{ij} \right]_{m \times n}; \quad \boldsymbol{A} \odot \boldsymbol{B} = \left[a_{ij} b_{ij} \right]_{m \times n}.$$

Транспонирование матрицы:

$$\boldsymbol{A} \in \mathbb{R}^{m \times n}; \quad \boldsymbol{A}^T \in \mathbb{R}^{n \times m}.$$

Иногда возникает необходимость «повернуть» матрицу, т. е. поменять местами столбцы и строки — такая операция называется транспонированием.

Если выполнить операцию транспонирования два раза, мы вернемся к исходной матрице:

$$\boldsymbol{A}^{TT} = \boldsymbol{A}.$$

А вот *деления матриц* не существует. Есть своеобразный аналог деления из школьной алгебры: домножение на обратную матрицу. Обратная матрица соответствует следующему условию:

$$\boldsymbol{A} \cdot \boldsymbol{A}^{-1} = \boldsymbol{A}^{-1} \cdot \boldsymbol{A} = \boldsymbol{I},$$

где $\boldsymbol{I}$ — единичная матрица.

Ключевые понятия математического анализа

Итак, мы вспомнили, что такое объекты и какие базовые операции можно с ними производить. Однако для решения практических задач требуется более сложное взаимодействие между объектами. Для этого понадобятся понятия из математического анализа, которые описывают функции.

Функция в математике — это соответствие между элементами двух множеств, правило, по которому каждому элементу первого множества соответствует один и только один элемент второго множества. Схожая аналогия существует также и в программировании.

Самый простой пример функции — это функция, которая «ничего не делает». Такая функция берет на вход переменную x и возвращает ее.

Функции можно представить по-разному, можно в виде математического выражения: $f(x)=x$, а можно в виде графика (рис. 1.13, *а*). Как видно, данная функция выглядит как прямая линия.

У функции могут быть дополнительные параметры, например входные данные можно умножить на скаляр и добавить скаляр: $f(x)=0.5\cdot x-1$. График такой функции представлен на рис. 1.13, *б*, он все еще выглядит как прямая линия. Подобные функции, выходной аргумент которых пропорционален входному аргументу, называются *линейными*.

Функции бывают также и *нелинейными*. Например, на начальном этапе использования нейронных сетей была достаточно распространена сигмоидная функция, которую также называют логистической, она выражается уравнением $f(x)=\dfrac{1}{1+e^{-x}}$ (рис. 1.13, *в*). Другим примером нелинейной функции из мира нейронных сетей является изображенная на рис. 1.13, *г*, функция ReLU (Rectified Linear Unit — линейный выпрямитель): $f(x)=\max(0,x)$.

Для анализа поведения некоторой функции часто используют другую функцию, которая называется производной. *Производная функции* в точке — это скорость изменения функции в данной точке. Производную можно определить как предел отношения приращения функции к приращению аргумента:

$$f'(x_0)=\lim_{x\to x_0}\frac{f(x)-f(x_0)}{x-x_0}=\lim_{\Delta x\to 0}\frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}=\lim_{\Delta x\to 0}\frac{\Delta f(x)}{\Delta x}.$$

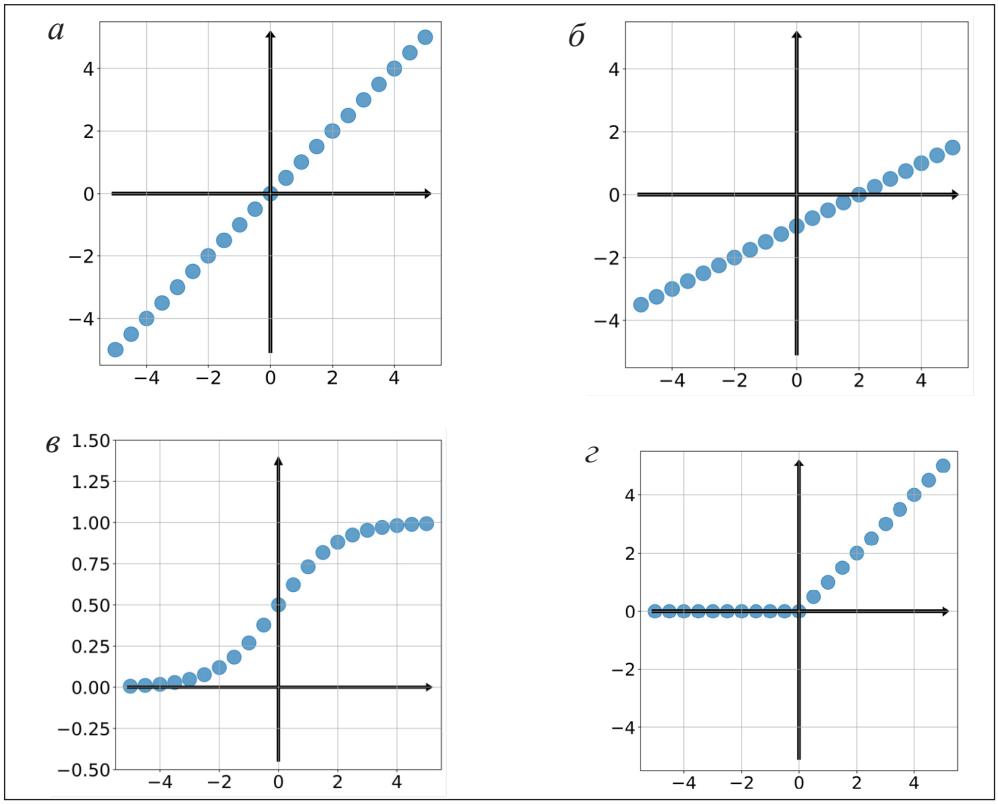


Рис. 1.13. Примеры линейных (*а*, *б*) и нелинейных (*в*, *г*) функций

На рисунке 1.14 представлены примеры производных для линейной функции (*а*), логистической функции (*б*) и ReLU (*в*).

Так, видно, что скорость изменения линейной функции постоянна, логистической функции — зависит от области значений, а для функции ReLU меняется скачкообразно.

Ниже представлены те функции и их производные, которые достаточно знать для успешного усвоения материала данного учебно-методического пособия:

Функция	Производная
c — константа	0
x	1
x^m	$m \cdot x^{m-1}$
$\ln(x)$	$\frac{1}{x}$

Функция	Производная
e^x	e^x
$f(x)\cdot g(x)$	$f'(x)\cdot g(x)+f(x)\cdot g'(x)$
$c\cdot f(x)$	$c\cdot f'(x)$
$y=f(t)$ $t=g(x)$	$\frac{\partial y}{\partial x}=\frac{\partial y}{\partial t}\frac{\partial t}{\partial x}$

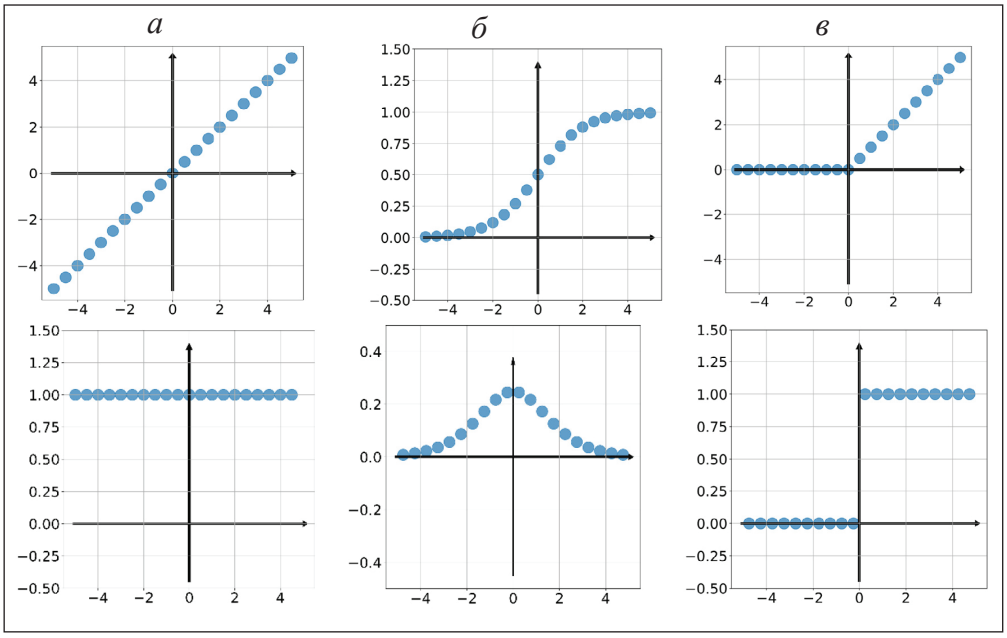


Рис. 1.14. Примеры функций и их производных

Отдельно стоит упомянуть последнюю строку в таблице — так называемое *цепное правило*: если y — это функция от переменной t, а t, в свою очередь, зависит от переменной x, тогда производная y по переменной t будет равна производной y по переменной t, помноженной на производную t по переменной x.

Функции могут зависеть от разных переменных. Например, для функции $f(x,w,b)=wx+b$ можно посчитать производные по трем переменным — x, w, b. Если мы считаем производную $\frac{\partial f}{\partial x}$, то остальные переменные считаются константами.

Производные по отдельным переменным называются *частными производными*. Вектор, составленный из частных производных, назы-

вается *градиент*. Для упомянутой выше функции f градиент будет выглядеть следующим образом:

$$\nabla f(x,w,b)=\left[\frac{\partial f}{\partial x},\frac{\partial f}{\partial w},\frac{\partial f}{\partial b}\right]=[w,x,1].$$

Отдельно стоит рассмотреть производную квадратичной функции $f(x)=x^2$. Напомним, что мы определили: в общем случае цель обучения модели — это минимизация функционала потерь, $Q(a,\mathbb{X})\to\min_{a\in\mathbb{A}}$. В качестве примера функционала потерь достаточно часто используется именно квадратичная функция. Допустим, у этого функционала всего один параметр — w (рис. 1.15).

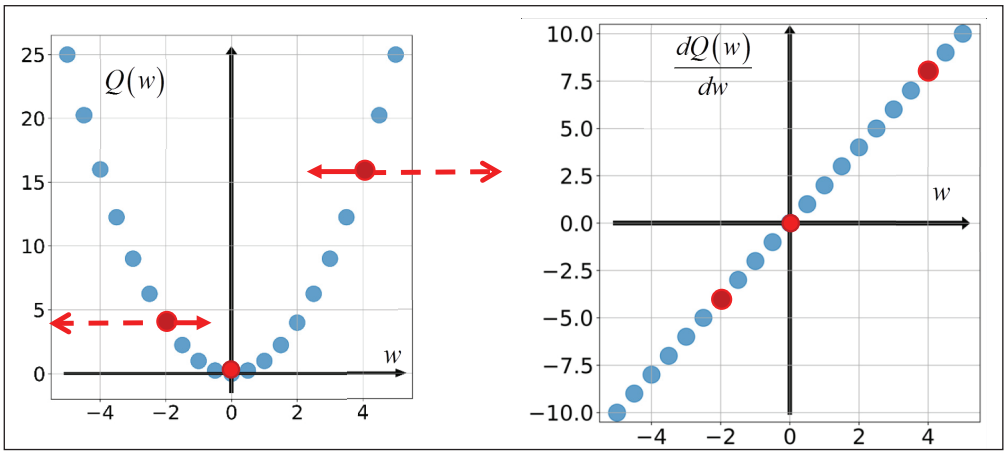


Рис. 1.15. Квадратичная функция и ее производная

Для того чтобы найти минимум этой функции, нужно приравнять производную этой функции к нулю. Значение параметра w_0, при котором производная $\frac{dQ(w=w_0)}{dw}$ равна нулю, будет соответствовать параметру, минимизирующему исходный функционал. Запомним это.

С другой стороны, допустим, мы подобрали значения параметра случайным образом и хотим оценить, насколько хорош этот параметр и можно ли его как-то изменить, используя производную. Пусть $\frac{dQ(w=-2)}{dw}=-4$. Что это значит? В этой точке функция убывает (производная отрицательная), т. е. если продолжать увеличивать параметр,

то мы, скорее всего, придем к минимуму функции. Аналогично рассмотрим $\frac{dQ\left(w=4\right)}{dw}=8$. Производная положительная, а значит, функция возрастает. Причем возрастает быстрее, чем в точке $w = -2$. Это означает, что если мы уменьшим параметр, то приблизимся к минимуму функции. При этом нужно сделать больший шаг, поскольку абсолютная величина производной больше. Эта идея изменения параметров в направлении, обратном знаку производной, на величину, пропорциональную значению производной, легла в основу *алгоритма градиентного спуска*, который мы более подробно рассмотрим в 3-й главе.

Контрольные вопросы

1. Какие числовые признаки называются дискретными, а какие — непрерывными? Приведите собственные примеры.
2. Опишите разницу между обучением с учителем и обучением без учителя.
3. Опишите разницу между задачами классификации и регрессии.
4. Если вас попросят создать программу, которая будет определять кошку или собаку по изображению, к какому типу задач машинного обучения относится эта просьба?
5. Почему в качестве альтернативы простому использованию всех тренировочных данных рекомендуется выполнять проверку модели на валидационных данных?
6. Даны три матрицы *A*, *B*, *C*. Матрица *A* имеет размеры 5 × 4, *B* — размеры 4 × 6, *C* имеет размеры 3 × 5. Укажите все возможные матрицы, которые можно перемножить между собой.
7. Найдите градиент функции $f\left(x,y,z\right)=yx^2+\ln\left(y\right)+e^{-z}$.
8. Найдите частные производные сложной функции $E=(\hat{y}-y)^2$, $\hat{y}=wx+b$ по переменным *w* и *b*.
9. Найдите частные производные сложной функции $E=y\ln\hat{y}+\left(1-y\right)\ln\left(1-\hat{y}\right)$, $\hat{y}=\frac{1}{1+e^{-z}}$ по переменной *z*.

Глава 2.
Исследовательский анализ данных

В этой главе рассмотрим действия, с которых нужно начинать анализ данных, перед тем как применять к ним простые и сложные алгоритмы машинного обучения.

В качестве набора данных будет использоваться набор Cars, он представляет собой статистику параметров автомобилей на вторичном рынке. Набор включает ряд категориальных и численных значений, составляющих одну запись (строку). Каждый столбец в записи — это отдельный признак. Среди указанных признаков приведены целевой для задачи предсказания (регрессии) — цена автомобиля, а также целевой для задачи классификации — тип трансмиссии. Последняя задача может быть рассмотрена как пример задачи на заполнение пропусков (если продавец не указал соответствующий параметр).

Для начального анализа данных рекомендуется использовать библиотеку Pandas [12].

Библиотека Pandas для анализа данных

Для начала работы с любой библиотекой необходимо импортировать ее:

```
import pandas as pd
```

Теперь нужно загрузить данные в структуру Pandas датафрейм (Dataframe). Она представляет собой двумерную структуру данных с именными столбцами потенциально разных типов – ее можно воспринимать как файл Excel, но на языке Python.

Считывание файлов в датафрейм

Для загрузки данных можно использовать функцию `pd.read_csv(path,delimiter)`. Для успешного использования необходимо указать путь (`path`) к файлу. В общем случае путь – это строка, содержащая полный путь к файлу и название файла.

Можно также указать разделитель (`delimiter`), в нашем случае это «,». Но для разных файлов могут использоваться и другие типы разделителей, такие как «;», « » (пробел), «\t» (табуляция). Вы можете проверить используемый в файле разделитель, предварительно открыв его в текстовом редакторе, например Notepad++.

Следующая строка загрузит файл с названием `cars.csv`, который лежит в папке `/content/`, разделитель – запятая:

```
df = pd.read_csv('/content/cars.csv', delimiter = ',')
```

Для того чтобы визуализировать содержание загруженного датафрейма в блокнотах Google Colab, достаточно просто запустить отдельную ячейку, в которой прописана переменная, содержащая датафрейм. В других случаях можно воспользоваться функцией `display`. На рисунке 2.1 представлена визуализация датафрейма набора Cars.

	Make	Model	Year	Style	Distance	Engine_capacity(cm3)	Fuel_type	Transmission	Price(euro)
0	Toyota	Prius	2011	Hatchback	195000.0	1800.0	Hybrid	Automatic	7750.0
1	Renault	Grand Scenic	2014	Universal	135000.0	1500.0	Diesel	Manual	8550.0
2	Volkswagen	Golf	1998	Hatchback	1.0	1400.0	Petrol	Manual	2200.0
3	Renault	Laguna	2012	Universal	110000.0	1500.0	Diesel	Manual	6550.0
4	Opel	Astra	2006	Universal	200000.0	1600.0	Metan/Propan	Manual	4100.0
...	...	...	...	...	...	...	...	...	...
41002	Dacia	Logan Mcv	2015	Universal	89000.0	1500.0	Diesel	Manual	7000.0
41003	Renault	Modus	2009	Hatchback	225.0	1500.0	Diesel	Manual	4500.0
41004	Mercedes	E Class	2016	Sedan	50000.0	1950.0	Diesel	Automatic	29500.0
41005	Mazda	6	2006	Combi	370000.0	2000.0	Diesel	Manual	4000.0
41006	Renault	Grand Scenic	2006	Minivan	300000.0	1500.0	Diesel	Manual	4000.0

41007 rows × 9 columns

Рис. 2.1. Визуализация датафрейма набора Cars

Общая информация о датафрейме

Чтобы получить общую информацию о содержимом датафрейма, можно применить к нему метод `.info()`. Должна появиться следующая информация (рис. 2.2): количество столбцов, их имена, тип данных в каждом столбце, количество пропущенных значений для каждого столбца.

```
RangeIndex: 41007 entries, 0 to 41006
Data columns (total 9 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Make                  41007 non-null  object
 1   Model                 41007 non-null  object
 2   Year                  41007 non-null  int64
 3   Style                 41007 non-null  object
 4   Distance              41007 non-null  float64
 5   Engine_capacity(cm3)  41007 non-null  float64
 6   Fuel_type             41007 non-null  object
 7   Transmission          41007 non-null  object
 8   Price(euro)           41007 non-null  float64
dtypes: float64(3), int64(1), object(5)
memory usage: 2.8+ MB
```

Рис. 2.2. Информация о датафрейме набора Cars

Поиск и удаление дубликатов

Повторяющиеся строки в наборе данных могут возникать по многим причинам: программная ошибка, человеческая ошибка и т. д. При этом никакой пользы для алгоритмов машинного обучения от включения повторных строк не будет.

Вы можете проверить количество повторяющихся строк, применив метод `.duplicated()` к датафрейму. В результате получится столбец, который содержит информацию о том, является ли данная строка повторной или нет. Можно применить метод `.sum()`, чтобы узнать количество повторных строк:

```
df.duplicated().sum()
```

Для датафрейма набора Cars имеется 3743 дубликата, почти 1/10 от всех данных. Чтобы удалить дубликаты, используем следующую строку:

```
DF = df.drop_duplicates().reset_index(drop=True)
```

Она означает, что для оригинального датафрейма `df` мы просим Python использовать метод удаления дубликатов `.drop_duplicates()` и сбросить индексы датафрейма `.reset_index(drop=True)` — в качестве альтернативы можно сохранить исходные индексы датафрейма. Результаты вышеупомянутого преобразования сохраняются в новый датафрейм `DF`.

Хорошей практикой является размещение результата значительного преобразования в отдельном датафрейме, чтобы всегда можно было вернуться к исходному. Вы можете проверить, что все сделано правильно, оценив количество строк дубликатов в новом датафрейме.

Сохранение датафрейма в файл

Для сохранения датафрейма в файл можно воспользоваться методом `.to_csv(path, index)`, которому необходимо указать полный путь и название файла, в который вы хотите сохранить данные (`path`). Кроме того, можно указать, необходимо ли сохранять индексы строк (булева переменная `index`). Так, следующая строка сохранит датафрейм без дубликатов в папку `/content/` с названием `'cars_ no_dup.csv'`:

```
DF.to_csv('/content/cars_no_dup.csv',index=False)
```

Представление части датафрейма

Здесь стоит упомянуть два метода: `.head(n)` и `.tail(n)`. Эти два метода можно применить к датафрейму для визуализации первых `n` или последних `n` строк. Это очень полезно, когда датафрейм довольно большой и не может быть легко визуализирован полностью. Попробуйте сами: покажите первые 6, а затем последние 9 строк датафрейма.

Индексация

Одним из способов получения определенных элементов датафрейма является использование атрибута `.loc`. Приведем несколько примеров:

- взятие одной ячейки: `DF.loc[1437,'Transmission']` — этот код вернет элемент из строки с индексом 1437 в столбце `'Transmission'`;

- взятие одной колонки в формате серий (Series): `DF.loc[:,'Transmission']` — этот код вернет все элементы в столбце `'Transmission'` в виде вектора;
- взятие одной колонки в формате датафрейма: `DF.loc[:,['Transmission']]` — этот код вернет все элементы в столбце `'Transmission'` в виде матрицы;
- взятие нескольких колонок: `DF.loc[:,['Transmission','Year']]` — этот код вернет все элементы в столбцах `'Transmission'` и `'Year'`;
- взятие нескольких колонок подряд (среза столбцов): `DF.loc[:,'Make':'Style']` — этот код вернет все элементы в столбцах, начиная с `'Make'` по `'Style'` (включительно).

Для случаев построчной индексации:

- взятие одной конкретной строки в формате серий: `DF.loc[69,:]` — этот код вернет все элементы из строки с индексом 69;
- взятие одной конкретной строки в формате датафрейма: `DF.loc[69:69,:]`;
- получение среза строк `DF.loc[322:1437,:]` — этот код вернет все элементы в строках, начиная с индекса 322 и заканчивая индексом 1437 (включительно).

Конечно же, можно комбинировать срезы по строкам и срезы по столбцам: `DF.loc[227:229,'Make':'Fuel_type']` — этот код вернет все элементы в строках, начиная с индекса 227 и заканчивая индексом 229 (включительно), и в столбцах с `'Make'` по `'Fuel_type'`.

Альтернативой использования атрибута `.loc` является использование атрибута `.iloc`. Основное различие между ними заключается в следующем: `.loc` работает с названиями столбцов, а `.iloc` использует вместо этого целочисленную нумерацию.

Доступно также логическое индексирование (Boolean Indexing), когда необходимо взять такую часть датафрейма, которая соответствует некому условию. Например, строка `DF[DF['Transmission']=='Manual']` вернет все элементы исходного датафрейма, для которых выполняется условие, что признак в столбце `'Transmission'` равен `'Manual'`.

Некоторые функции не могут работать с датафреймами напрямую, поскольку рассчитаны на то, что данные подаются в формате массивов, например numpy array. Чтобы перейти от датафреймов к массивам, достаточно применить метод `.values` к необходимой части датафрейма.

Сортировка DataFrame

Для сортировки данных в датафрейме можно воспользоваться методом `.sort_values` — нужно указать столбец, по которому необходимо отсортировать. Например, строка `DF.sort_values(by = 'Price(euro)')` отсортирует строки датафрейма по значениям столбца `'Price(euro)'`.

По умолчанию метод `.sort_values` сортирует значения по возрастанию. Кроме того, можно указать «направление» сортировки, используя переменную `ascending`. Например, приведенный ниже код сортирует по убыванию столбца `'Year'`:

```
DF.sort_values(by = 'Year', ascending= False)
```

Визуализация данных

Для визуализации данных можно использовать «стандартную» для Python библиотеку визуализации Matplotlib [13; 14]. Однако при использовании датафреймов Pandas целесообразно пользоваться библиотекой Seaborn [15; 16]:

```
import seaborn as sns
```

Seaborn – это библиотека для создания разнообразной графики на Python, которая тесно интегрирована со структурами данных Pandas.

Один из лучших графиков для ознакомления с данными – это парный график (`pairplot`). Он отображает все возможные попарные комбинации числовых признаков набора данных в виде скаттерограмм, а по диагонали строятся гистограммы распределений. Но если признаков много, то и времени на это уйдет немало – лучше разделить датафрейм на несколько частей. Категориальные признаки можно визуализировать, используя, например, цвет данных. При этом Seaborn позволяет реализовать визуализацию достаточно просто, буквально в одну строку:

```
sns.pairplot(data = DF, hue ='Transmission')
```

Для того чтобы построить график, достаточно указать датафрейм, из которого будут взяты численные признаки в переменную `data`, дополнительно можно указать в качестве переменной `hue` (цвет) один из столбцов с категориальными данными. На рисунке 2.3 представлен `pairplot` для набора данных Cars.

Возможности библиотеки Seaborn для визуализации данных ограничены только воображением пользователя. Ознакомиться с другими визуализациями Seaborn можно по следующей ссылке: [17].

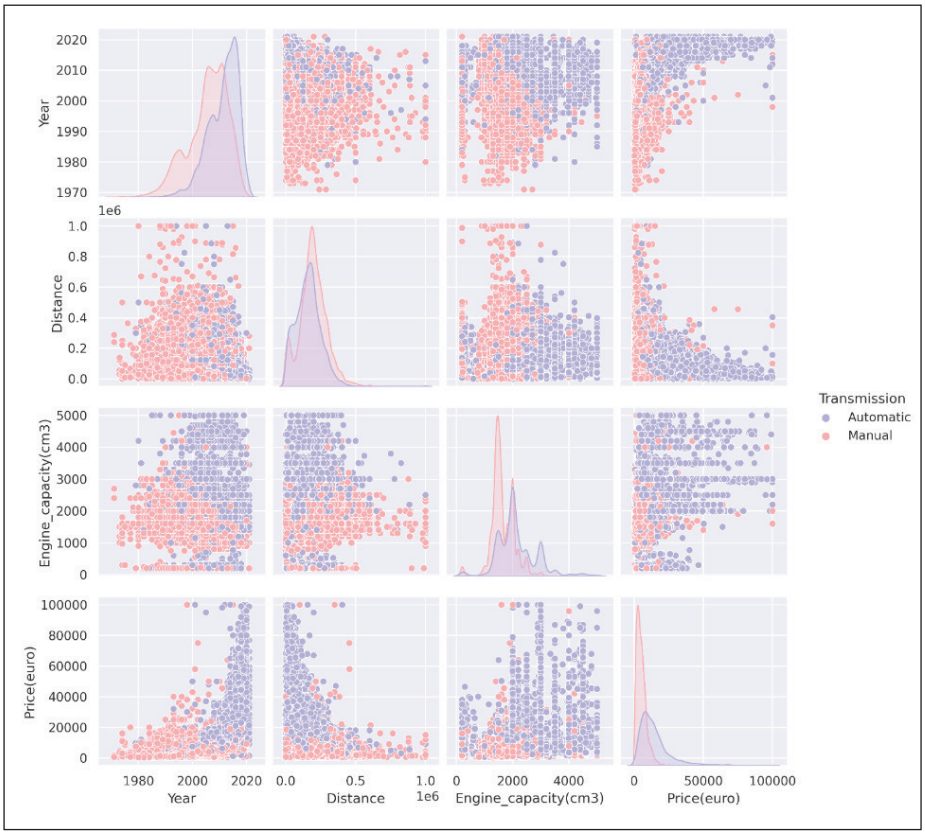


Рис. 2.3. Визуализация `pairplot` для набора данных Cars

Предварительная обработка данных

После ознакомления с данными необходимо выполнить их предварительную обработку. Она зависит от типа данных. Для того чтобы определить типы данных, можно воспользоваться следующим кодом:

```
cat_columns = []
num_columns = []
```

```
for column_name in df.columns:
    if (df[column_name].dtypes == object):
        cat_columns +=[column_name]
    else:
        num_columns +=[column_name]
```

В переменную `cat_columns` попадут названия всех колонок с категориальными признаками, а в `num_columns` — с числовыми признаками. Обсудим методы предварительной обработки, характерные для разных типов данных.

Предварительная обработка числовых данных

Начать анализ числовых данных стоит с оценки статистических показателей. В библиотеке Pandas это можно сделать в одну строку с использованием метода `.describe()`:

```
DF[num_columns].describe()
```

На рисунке 2.4 представлен результат применения метода `.describe()` к набору данных Cars.

	Year	Distance	Engine_capacity(cm3)	Price(euro)
count	37264.000000	3.726400e+04	37264.000000	3.726400e+04
mean	2007.709264	4.758488e+05	1858.932535	9.569387e+03
std	8.295806	4.591520e+06	707.662731	5.283315e+04
min	1900.000000	0.000000e+00	0.000000	1.000000e+00
25%	2004.000000	9.000000e+04	1499.000000	3.300000e+03
50%	2009.000000	1.700000e+05	1800.000000	6.490000e+03
75%	2014.000000	2.300000e+05	2000.000000	1.179900e+04
max	2021.000000	1.000000e+08	9999.000000	1.000000e+07

Рис. 2.4. Результат применения метода `.describe()` к числовым признакам набора данных Cars

В результате демонстрируются следующие оценки: количество данных в столбце (`count`), среднее значение (`mean`), стандартное отклонение (`std`), минимальное значение (`min`), 25, 50 и 75 перцентили (`25%, 50%, 75%`), максимальное значение (`max`).

Сопоставление среднего, минимального и максимального значений, а также перцентилей позволяет оценить, существуют ли в данных аномалии – редко встречающиеся значения. Для этого пригодится и визуализация гистограмм распределения. Иногда, как, например, для колонки `'Price(euro)'`, числовые признаки могут изменяться в большом диапазоне: от 1 до 10^7, хотя среднее значение, 25 и 75 перцентили сконцентрированы в диапазоне $10^3 \ldots 10^4$. Тогда для визуализации гистограммы распределения рекомендуется использовать логарифмический масштаб. Это говорит о наличии в наборе данных редких аномально высоких и аномально низких значений. Если какие-либо признаки встречаются крайне редко, то на таких значениях сложно обучить модель с высокой степенью обобщения. Поэтому, как правило, от редких высоких и низких значений признаков избавляются.

При этом аномалии могут быть вызваны ошибками при заполнении данных. Так, ошибочными выглядят значения столбца `'Price(euro)'` меньше 100 или старые автомобили с общим пробегом меньше, чем 1000 км.

В датафреймах Pandas можно удалить аномальные значения, используя метод `.drop` и логическую индексацию. Ниже представлены рекомендуемые условия по удалению аномальных значений:

```
# Старые автомобили с низким пробегом
question_dist_year = DF[(DF.Year <2021) & (DF.Distance < 1100)]
DF = DF.drop(question_dist_year.index)
# Аномально большой пробег
question_dist = DF[(DF.Distance > 0.5e6)]
DF = DF.drop(question_dist.index)
# Слишком малые значения объема двигателя
question_engine = DF [DF[“Engine_capacity(cm3)”] < 200]
DF = DF.drop(question_engine.index)
# Слишком большие значения объема двигателя
question_engine = DF[DF[“Engine_capacity(cm3)”] > 5000]
DF = DF.drop(question_engine.index)
# Аномально низкие цены
question_price = DF[(DF[“Price(euro)”] < 101]
DF = DF.drop(question_price.index)
# Слишком дорогие автомобили, которых мало
question_price = DF[DF[“Price(euro)”] > 1e5]
DF = DF.drop(question_price.index)
# Слишком старые автомобили, которых мало
question_year = DF[DF.Year < 1971]
DF= DF.drop(question_year.index)
```

Если рассматривать данные в целом, видно, что разные столбцы имеют разный разброс данных: год и объем двигателя измеряются в тысячах, стоимость измеряется в десятках тысяч, а пробег – в сотнях тысяч единиц. Проще сопоставлять данные, если они измеряются в одном диапазоне. Для приведения численных данных к единой шкале существуют различные подходы: стандартизация, нормализация, степенное преобразование.

Стандартизация

Стандартизация, или *z*-нормировка, сводится к вычитанию из матрицы признаков $X \in \mathbb{R}^{n \times p}$ вектора μ_j средних значений для каждого признака и делению полученной разности на вектор σ_j стандартных отклонений для каждого признака:

$$X' = \frac{\left(x_{ij} - \mu_j\right)}{\sigma_j}.$$

Таким образом, у новой матрицы признаков X' будет нулевое среднее и единичная дисперсия. При этом стандартизация – это линейная операция, т. е. распределение признаков не изменится, изменится только масштаб, в рамках которого это изменение происходит.

Чтобы реализовать стандартизацию на датафреймах Pandas, достаточно применить методы `.mean()` и `.std()` для оценки среднего значения `M` и стандартного отклонения `STD` для каждого столбца:

```
M = DF[num_columns].mean()
STD = DF[num_columns].std()
```

Затем происходит поэлементное вычитание из исходного датафрейма и деление:

```
DF_scaled = (DF[num_columns]-M)/STD
```

На рисунке 2.5 представлено распределение признака `Distance` до и после операции стандартизации. Как видно, распределение данных остается неизменным — меняются границы диапазона измерений.

Отдельно стоит отметить, что при выполнении стандартизации (как и других методов предварительной обработки) необходимо сохранять параметры преобразования, поскольку именно эти преобра-

зования необходимо совершать на новых данных, т. е. для тестового набора данных нужно вычитать среднее не тестового набора, а тренировочного.

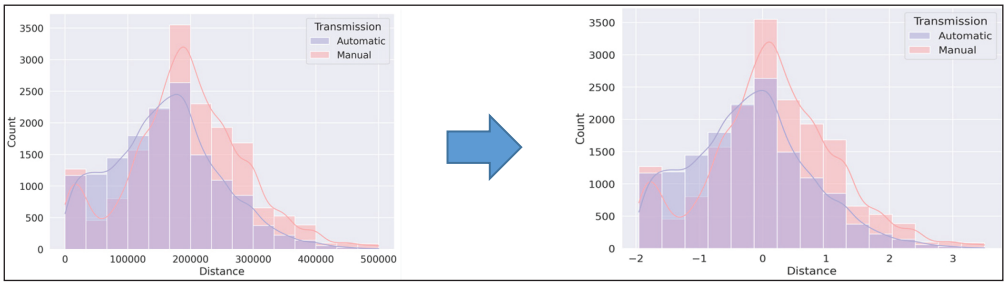


Рис. 2.5. Результат применения стандартизации к столбцу Distance набора данных Cars

Нормализация

Нормализация – это тоже линейное преобразование численных признаков $X \in \mathbb{R}^{n \times p}$. Но в отличие от стандартизации нормализация переводит распределение в интервал от 0 до 1:

$$X' = \frac{\left(x_{ij} - x_{j_{\min}}\right)}{\left(x_{j_{\max}} - x_{j_{\min}}\right)}.$$

Как и стандартизация, нормализация тоже просто реализуется с использованием датафреймов Pandas. Используются методы `.min()` и `.max()` для поиска минимальных и максимальных значений по столбцам:

```
Xmin = DF[num_columns].min()
Xmax = DF[num_columns].max()
```

Затем аналогично из исходного датафрейма происходит поэлементное вычитание и деление:

```
DF_norm = (DF[num_columns]- Xmin)/( Xmax - Xmin)
```

На рисунке 2.6 представлено распределение признака `Distance` до и после операции нормализации. Как видно, распределение данных по-прежнему остается неизменным, меняются только границы диапазона измерений.

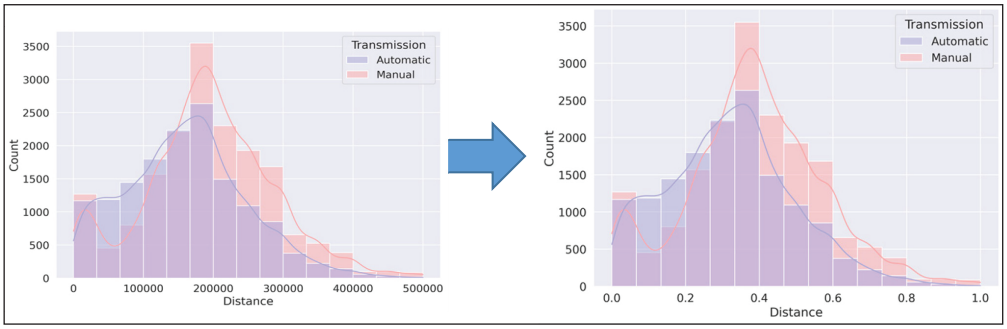


Рис. 2.6. Результат применения нормализации
к столбцу Distance набора данных Cars

Степенное преобразование

Стоит отметить, что корректное применение преобразований стандартизации и нормализации требует нормального распределения числовых данных. Однако такое происходит не всегда. В иных случаях рекомендуется предварительно использовать нелинейные преобразования (Power Transform), и уже потом результат нелинейного преобразования стандартизировать или нормализовать. К нелинейным преобразованиям относят логарифмирование и степенные преобразования (как правило, это корни, т. е. степени меньше единицы).

Так, на рис. 2.7 представлено распределение признака `Price(euro)` до и после операции степенного преобразования. В этом примере мы сначала логарифмировали столбец, а потом применили стандартизацию.

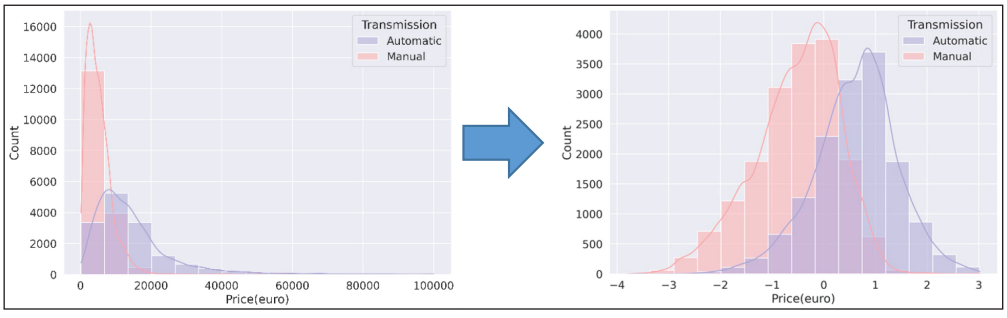


Рис. 2.7. Результат применения степенного преобразования
к столбцу Price(euro) набора данных Cars

Стоит подчеркнуть, что заранее нельзя узнать, какой из типов предварительной обработки данных лучше скажется на предсказаниях мо-

дели. Поэтому тип предварительной обработки можно рассматривать как дополнительный гиперпараметр модели.

Предварительная обработка категориальных данных

Анализ категориальных данных начинается с оценки частоты встречаемости отдельных категорий в рамках столбцов. На первом этапе необходимо воспользоваться методом `.nunique()` и оценить количество уникальных значений для каждой категории:

```
DF[cat_columns].nunique()
```

Для набора данных Cars столбцы `Make` и `Model` имеют 78 и 777 уникальных значений соответственно.

Далее необходимо оценить частоту встречаемости уникальных значений с использованием метода `.value_counts()`:

```
counts = DF.Make.value_counts()
```

Можно заметить, что для столбца `Make` порядка 15 уникальных значений встречаются 10 и меньше раз. Для выборки из ~30000 объектов это слишком мало. Но в отличие от численных признаков, для которых мы удаляли редко встречающиеся значения, для категориальных признаков можно сделать замену на единый класс — «редкий» (`rare`). Для этого воспользуемся логическим индексированием и методом `.replace()`:

```
rare =  counts[(counts.values < 25)]
DF['Make'] = DF['Make'].replace(rare.index.values, 'Rare')
```

Приведение категориальных признаков к числовым

Далее стоит рассмотреть формат хранения категориальных данных. В наборе данных Cars категориальные данные представляют собой тип данных `object` и по сути являются строками. Для уменьшения объема хранимой информации эти данные достаточно часто переводят в числовой формат.

Если уникальных значений не много, то такой перевод удобно реализовать с помощью метода `.map()` и словаря. Так, в строке ниже мы преобразуем столбец `Transmission`, в котором всего 2 уникальных значения:

```
    DF['Transmission'] = DF['Transmission'].map({'Automatic': 1,
'Manual': 0})
```

Однако если уникальных значений много, то такое преобразование может быть затруднено. Поэтому рекомендуется воспользоваться изменением типа данных от `object` к `category`, а затем применить кодирование отдельных столбцов в числовые значения с помощью `.cat.codes`. Полный пример перехода к числовым значениям представлен ниже:

```
DF_ce = df.copy()
DF_ce[cat_columns] = DF_ce[cat_columns].astype('category')

for _, column_name in enumerate(cat_columns):
    DF_ce[column_name] =  DF_ce[column_name].cat.codes
```

Такое кодирование называют *порядковым кодированием* (Ordinal Encoding). Стоит отметить, что порядковое кодирование позволило сократить объем данных примерно на треть: с 2,2 до 1,4 MB.

One-Hot-кодирование

Существуют некоторые алгоритмы, способные работать с категориальными данными в формате порядкового кодирования. Это естественное кодирование порядковых переменных. Для категориальных переменных оно налагает порядковое отношение там, где такого отношения может не быть. В общем случае использование этого кодирования и разрешение модели принимать естественное упорядочение между категориями может привести к снижению производительности или неожиданным результатам (дробные категории).

Использование One-Hot-кодирования (Encoding) достаточно распространено. Суть его состоит в том, что вместо одного вектора $\boldsymbol{x}^{n\times 1}$ категориального признака создается матрица $\Xi^{n\times m}$, где m – количество уникальных категорий в векторе $\boldsymbol{x}$. При этом матрица Ξ состоит только из 0 и 1. Пример перевода категориального признака с использованием One-Hot Encoding представлен на рис. 2.8.

Категориальный признак «цвет» состоит из 4 уникальных категорий. В результате One-Hot-кодирования создается 4 новых столбца, по одному на каждое уникальное значение цвета (красный, зеленый, синий, белый). Столбцы заполнятся в основном 0, а для тех индексов, которые соответ-

ствуют численному значению признака, ставится 1. По сути, One-Hot-кодирование преобразует вектора категориальных признаков в матрицы бинарных признаков. Это позволяет придавать некоторый смысл числовым значениям категориальных признаков. В библиотеке Pandas такое кодирование реализуется с помощью метода `.get_dummies()`. При этом можно преобразовывать полный датафрейм — One-Hot-кодирование применится только к столбцам с категориальными данными:

```
DF_ohe = pd.get_dummies(DF.copy())
```

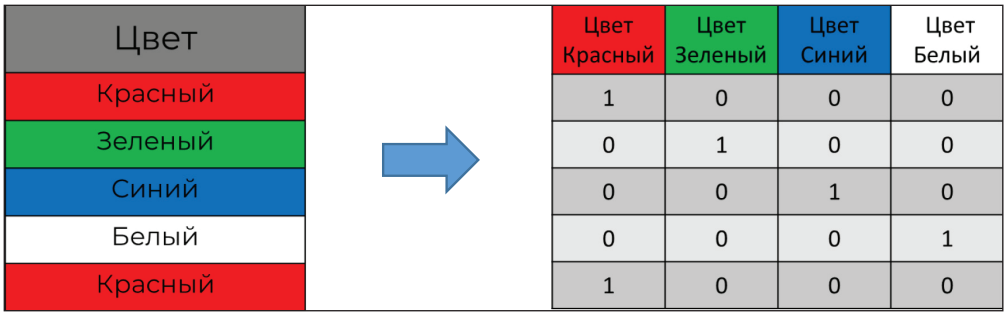


Рис. 2.8. Пример реализации One-Hot-кодирования для признака «цвет»

Стоит отметить, что использование One-Hot-кодирования значительно увеличивает размер матрицы данных. Поэтому для признаков с большим количеством уникальных значений рекомендуется предварительно уменьшить количество категорий, объединив редкие категории в одну, как было показано ранее.

Инженерия признаков

Достаточно часто необходимо выполнить дополнительные преобразования с исходными признаками, чтобы «помочь» модели выявить нужные закономерности. Как правило, базовая инженерия признаков сводится к умной комбинации исходных признаков. Рассмотрим несколько примеров для набора данных Cars.

Есть столбец данных `Year` (год выпуска автомобилей). Но с точки зрения предсказания стоимости автомобиля сам год как таковой име-

ет не очень большое значение. Больший смысл несет «возраст» автомобиля, т. е. количество лет, которое прошло от года выпуска до настоящего времени:

```
DF['Age'] = 2022 - DF.Year
```

Другим важным признаком является средний пробег в год, т. е. насколько интенсивно автомобиль использовался в среднем за год. Это чуть более полезная информация, чем просто общий пробег:

```
DF['km_year'] = DF.Distance/DF.Age
```

Инженерия новых признаков – достаточно творческий процесс, и он иногда требует чуть более глубокого погружения в предметную область. При этом можно проверять различные гипотезы – например, создать категориальный признак, связанный с объемом двигателя: если объем двигателя больше, допустим, 3 литров, то его стоит пометить отдельной категорией. Это связано с тем, что автомобили с бóльшим объемом двигателя меньше востребованы из-за большего расхода и повышенного налогообложения, что тоже сказывается на цене.

При добавлении признаков в набор данных на основе имеющихся важно отслеживать, не добавляем ли мы в модель то, что в ней уже есть. Рекомендуется отслеживать коэффициент корреляции между признаками, и если он близок к единице, то какой-то из признаков рекомендуется удалить.

В библиотеке Pandas вычисление матрицы корреляции делается с помощью метода `.corr()`. Для лучшего восприятия информации добавим «раскраску» полученных результатов:

```
cm = sns.color_palette("vlag", as_cmap=True)
DF.corr().style.background_gradient(cmap=cm, vmin = -1, vmax=1)
```

В результате должен получиться раскрашенный датафрейм, похожий на рис. 2.9.

Видно, что признаки `Age` и `Year` имеют 100%-ю обратную корреляцию. Что неудивительно, ведь один признак — это разность константы и второго признака. С другой стороны, корреляция между новым признаком `km_year` не превышает 0.5 — это говорит о том, что данный признак может нести дополнительную информацию.

	Year	Distance	Engine_capacity(cm3)	Price(euro)	Age	km_year
Year	1.000000	-0.434034	-0.025707	0.551627	-1.000000	0.426025
Distance	-0.434034	1.000000	0.067378	-0.347236	0.434034	0.462777
Engine_capacity(cm3)	-0.025707	0.067378	1.000000	0.382831	0.025707	-0.010386
Price(euro)	0.551627	-0.347236	0.382831	1.000000	-0.551627	0.157024
Age	-1.000000	0.434034	0.025707	-0.551627	1.000000	-0.426025
km_year	0.426025	0.462777	-0.010386	0.157024	-0.426025	1.000000

Рис. 2.9. Матрица корреляция для набора данных Cars

Практические задания

1. Скачайте набор данных Cars (https://github.com/dayekb/Basic_ML_Alg) и используйте функции и методы библиотеки Pandas для загрузки и начальной работы с данными.
2. Выполните визуализацию данных с использованием библиотеки Pandas. Попробуйте построить разные виды графиков для числовых признаков — скаттерограммы, гистограммы и т. д. Для скаттерограмм попробуйте использовать категориальные данные для таких параметров графиков, как оттенок (`hue`), размер маркера (`size`), тип маркера (`style`). Таким образом, вы можете объединить информацию о нескольких признаках в один двумерный график.
3. Попытайтесь добавить в модель дополнительные признаки на основе имеющихся. Проверьте корреляцию новых признаков с добавленными.
4. Выполните предварительную обработку данных. Сохраните результаты разных методов предварительной обработки в разные файлы, чтобы потом была возможность протестировать различные гипотезы.

Контрольные вопросы

1. Допустим, у вас есть файл с данными, который называется `'iris.csv'`. Этот файл находится в папке `'/data/'`. Вы открываете его в текстовом редакторе и видите следующие первые строки:

```
sepal length in cm; sepal width in cm; petal length in
cm; petal width in cm; class
5.1; 3.5; 1.4; 0.2; 0
```

Как должна выглядеть команда для считывания данных в датафрейм Pandas?

2. Для набора данных Cars после удаления дубликатов выберите из полного датафрейма строки с индекса 69 по 322. Отсортируйте полученный датафрейм по колонке `'Distance'` по убыванию. Какое значение колонки `'Style'` у полученного датафрейма во второй строке сверху?
3. Для набора данных Cars оцените количество строк, которые были удалены после анализа гистограмм распределения и удаления аномальных значений.
4. Для набора данных Cars назовите самую распространенную марку автомобилей (столбец `Make`).
5. Визуализируйте скаттерограмму для двух столбцов — `Distance` и `Year` — набора данных Cars с использованием столбца `Transmission` в качестве цвета маркера (`hue`). К какому типу `Transmission` относится точка, которая наиболее близка к координатам (`Year` = 1980, `Distance` = 500 000)?
6. Представим, что вы визуализировали некий набор данных (рис. 2.10).

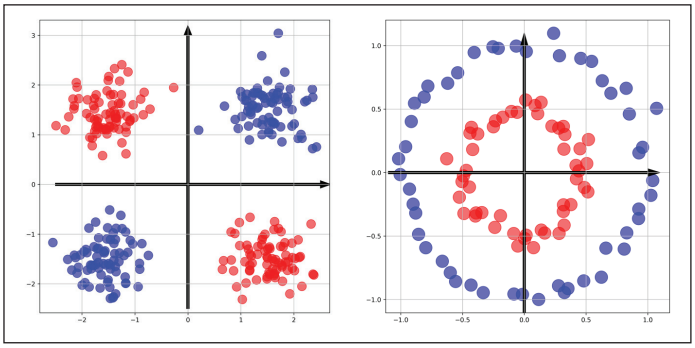


Рис. 2.10. Данные, линейно разделяемые с использованием грамотного конструирования признаков

Какие новые признаки, основанные на имеющихся, необходимо сконструировать, чтобы иметь возможность отделить все красные точки от всех синих точек с помощью прямой линии?

Глава 3.
Линейная регрессия

В этой главе мы рассмотрим, пожалуй, самый простой алгоритм в машинном обучении – линейную регрессию, которая, несмотря на свою простоту, может применяться как стартовая точка при анализе почти любых наборов данных.

Интересный исторический факт. Два выдающихся математика заявили об авторстве алгоритма, и 200 лет спустя вопрос остается нерешенным. В 1805 г. французский математик А. М. Лежандр опубликовал метод подгонки линии к набору точек. Он пытался предсказать местонахождение кометы (для справки: в то время астронавигация была наиболее ценной в мировой торговле наукой, так же как искусственный интеллект – «новое электричество» в наше время).

Четыре года спустя немецкий ученый К. Ф. Гаусс настаивал на том, что он использовал тот же метод с 1795 г., но считал его слишком тривиальным, чтобы о нем писать. Утверждение Гаусса побудило Лежандра анонимно опубликовать приложение, в котором отмечалось, что «один очень знаменитый геометр без колебаний присвоил этот метод».

Дальнейшее развитие раскрыло широкий потенциал алгоритма. В 1922 г. английские статистики Р. Фишер и К. Пирсон показали, как линейная регрессия вписывается в общую статистическую структуру корреляции и распределения, что сделало ее полезной во всех науках. И почти столетие спустя появление компьютеров предоставило данные и вычислительную мощность, позволяющую извлечь из них гораздо больше пользы.

Стоит также отметить, что наиболее распространенным типом нейронов в нейронной сети является модель линейной регрессии, за которой следует нелинейная функция активации, что делает линейную регрессию фундаментальным строительным блоком глубокого обучения.

Генерируемые данные

Перед тем как использовать алгоритм на реальных данных, в учебных целях бывает полезно тестировать его на данных, которые можно менять самостоятельно: чтобы рассмотреть различные ситуации и увидеть, как это влияет на результат.

Реализуем с помощью функции `true_fun` создание одномерных данных, которые могут быть некоторыми функциями от входных данных *X*:

```
    def true_fun(x, a=np.pi, b = 0, f=np.sin):
        x = np.atleast_1d(x)[:]
        a = np.atleast_1d(a)
        if f is None: f = lambda x:x
        x = np.sum([ai*np.power(x, i+1) for i,ai in enumerate(a)],
axis=0)
        return f(x+ b)
```

В этой функции `x` — входные данные, `a` — коэффициент, на который данные умножаются, `b` — константная добавка, `f` — функция, которая применяется к входным данным. Стоит отметить, что для полиномиальных зависимостей достаточно использовать в качестве `a` список коэффициентов. Созданные данные могут быть как линейными, `f = None`, так и гармоническими (`np.sin, np.cos`) или экспоненциальными `np.exp`.

В учебных целях (допускающих не 100%-ю точность) мы также добавим шумы к данным с помощью функции `noises`. Аргументами этой функции являются `shape` (размер массива) и `noise_power` (величина шума):

```
    def noises(shape , noise_power):
       return np.random.randn(*shape) *noise_power
```

Соберем это все в рамках единой функции `dataset`. Новыми аргументами в этой функции становятся `N` – количество точек, которые мы хотим сгенерировать, `x_max` — максимальное значение входных данных и булева переменная `random_x`. Если `random_x = True`, тогда будут случайным образом сгенерированы `N` точек, иначе данные будут распределены равномерно в диапазоне от `0` до `x_max`.

```
    def dataset(a, b, f = None,  N = 250, x_max =1, noise_power
= 0, random_x = True,  seed = 42):
        np.random.seed(seed)

        if random_x:
            x = np.sort(np.random.rand(N))*x_max
        else:
            x = np.linspace(0,x_max,N)

        y_true = np.array([])

        for f_ in np.append([], f):
            y_true=np.append(y_true, true_fun(x, a, b, f_))

        y_true = y_true.reshape(-1,N).T
        y = y_true + noises(y_true.shape , noise_power)

        return y, y_true, np.atleast_2d(x).T
```

Выходными параметрами этой функции являются `y_true` (истинный вариант зависимости), `y` (зашумленный вариант), `x` (входные данные).

Приведем примеры команд для генераций данных и их визуализации:

- линейная зависимость $y = 2x + 2,\ x \in [0,\ \ldots,\ 1]$ (рис. 3.1):

```
y, y_true, x = dataset(a = 2, b = 2, f = None,  N = 100, x_
max =1, noise_power = 0.1, seed = 42)
```

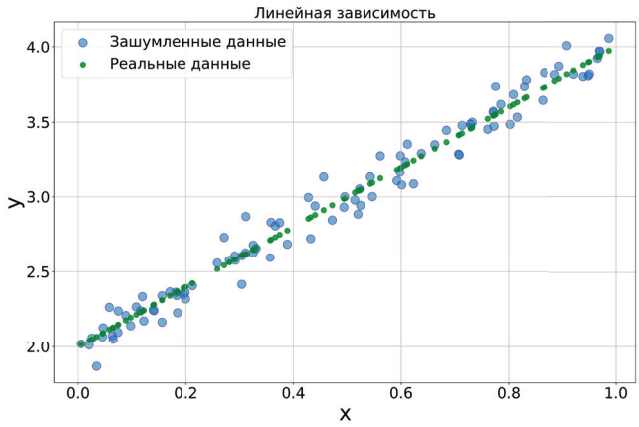


Рис. 3.1. Сгенерированные данные с линейной зависимостью

- полиномиальная зависимость $y = -x + 2x^2 - 2x^3 - 1, x \in [0, ..., 1.25]$ (рис. 3.2):

```
y, y_true, x = dataset(a = [-1,2,-2], b = -1, f = None,  N = 250, x_
max =1.25, noise_power = 0.05, seed = 42)
```

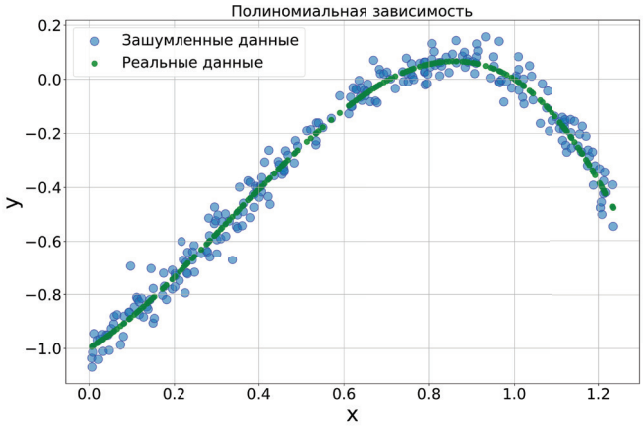


Рис. 3.2. Сгенерированные данные с полиноминальной зависимостью

- гармоническая зависимость $y = \cos(3\pi x + \pi), x \in \left[0, \ldots, \frac{\pi}{4}\right]$ (рис. 3.3):

```
y, y_true, x = dataset(a = 3*np.pi,b = 0, f = np.cos,  N = 25,
x_max =np.pi/4, noise_power = 0.1, seed = 42)
```

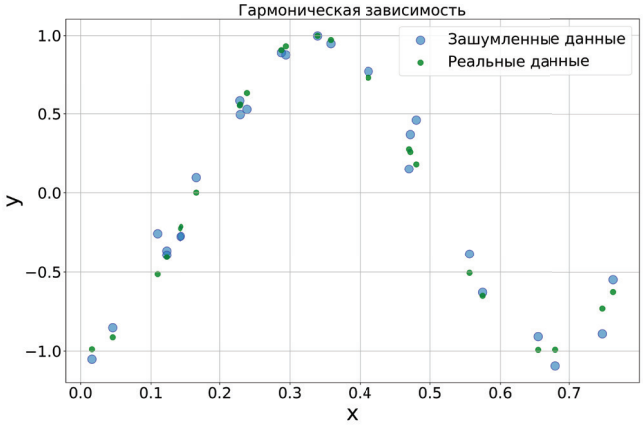


Рис. 3.3. Сгенерированные данные с гармонической зависимостью

Модель линейной регрессии

Для начала построим самую простую зависимость – линейную, она представлена на рис. 3.1. Проведем линейную регрессию, т. е. оценим коэффициенты аппроксимации зашумленных данных функцией вида

$$\hat{\boldsymbol{y}} = \hat{a}\boldsymbol{x} + \hat{b},$$

где $\hat{y}_i$ — это оценки целевых значений y_i; $\hat{a}$ и $\hat{b}$ — коэффициенты модели.

В самом простом случае (для этого набора данных) мы могли бы найти коэффициенты $\hat{a}$ и $\hat{b}$ аналитически. Если бы не было шумов (была бы чистая линия), коэффициенты можно было бы найти классическим решением линейной системы из двух независимых уравнений. При наличии шумов двух уравнений может быть недостаточно, оценки коэффициентов по каждым двум уравнениям могут различаться. Поэтому желательно иметь переопределенную систему (в которой число уравнений больше числа переменных). Такую систему можно решать по-разному.

Известно (теорема Гаусса — Маркова), что если шум в системе имеет нормальное распределение, то оптимальным будет решение методом наименьших квадратов. Для поиска такого решения предположим, что мы производим такие оценки коэффициентов $\hat{a}$ и $\hat{b}$, которые дают результат $\hat{y}$ с ошибкой ε. Ошибку определим как $\varepsilon_i = y_i - \hat{y}_i$. Теперь наша задача – минимизация ошибки ε для каждого значения $\boldsymbol{x}$.

В случае метода наименьших квадратов задача сводится к минимизации одного значения среднего квадрата ошибки. Для линейной регрессии такую задачу можно решить аналитически, однако, как можно будет увидеть ниже, такое решение не всегда рационально.

Для зависимостей более сложных, чем линейная, можно предположить, что указанной выше модели недостаточно. Допустим, нам неизвестно, как лучше всего аппроксимировать зависимость, какую функцию использовать.

В более общем случае можно ввести модель целевой переменной как

$$y_i = \sum_{j=0}^{p} x_{ij} w_j + \varepsilon_i,$$

или

$$y_i = x_{i0} \cdot w_0 + x_{i1} \cdot w_1 + x_{i2} \cdot w_2 + \ldots + x_{ip} \cdot w_p + \varepsilon_i,$$

или

$$y_i = X_i \boldsymbol{w}^T + \varepsilon_i,$$

где y_i — целевой показатель предсказания для i записи в наборе данных; $X_i = \{x_{ij}\}_{j=1}^{p}$ — набор входных параметров для i результата; $w = \{w_j\}_{j=1}^{p}$ — набор весовых параметров, которые необходимо подобрать в модели; ε_i — некоторый набор случайных (не объясняемых моделью, остаточных) значений, будем считать их случайным шумом.

Стоит отметить, что в формуле суммирование происходит по индексу j от 0. Это связано с тем, что, как правило, в модель добавляют «нулевой» столбец, значения в котором для всех i равны 1 — это позволяет оценить независимый коэффициент w_0, который иногда называют смещением (bias). Однако добавлять его в модель необязательно.

Данная модель соответствует как линейной регрессии одной или нескольких переменных, так и полиномиальной или любой другой, где признаки x_{ij} можно считать независимыми составляющими. В качестве примера регрессионная модель одной переменной будет иметь вид

$$\hat{y}_i = \sum_{j=0}^{1} x_{ij} w_j, \text{ или } \hat{y}_i = 1 \cdot w_0 + x_{i1} \cdot w_1, \text{ или } \hat{y}_i = X_i w^T,$$

где $\hat{y}_i$ — результат предсказания для i записи в наборе данных.

Введем функцию расчета (предсказания) значений `predict`. В этой функции мы задали возможность добавлять или не добавлять в модель независимый коэффициент с помощью булевой переменной `add_bias`. Для учета смещения будем добавлять единичный столбец к входным данным:

```
def predict( X, weights, add_bias = True):
    if add_bias:
        X_full = np.column_stack((np.ones(X.shape[0]),X))
    else:
        X_full = X
    return  np.dot(X_full, weights)
```

Теперь рассмотрим решение для обозначенной модели одной переменной. Введем функцию потерь регрессии `loss_func` как квадрат разности между целевыми значениями и их предсказаниями:

$$L\left(\hat{y}_i, y_i\right) = L_i = (\hat{y}_i - y_i)^2 = \left(\sum_{j=0}^{2} x_{ij} w_j - y_i\right)^2 = \left(X_i w^T - y_i\right)^2,$$

где L_i — функция потерь для результата (предсказания) с номером i.

В виде функций Python это реализуется достаточно просто:

```
def loss_func(yhat, y):
    return np.square(yhat - y)
```

Отметим, что для данного случая одной переменной (в вышеприведенных обозначениях) решение могло бы быть найдено как

$$L = \sum_{i=0}^{N-1} (y_i - \hat{y_i})^2 = \sum_{i=0}^{N-1} (y_i - \left(\hat{a} \cdot x_i + \hat{b}\right))^2 \to 0,$$

тогда минимум L будет соответствовать нулям ее производных по a и b:

$$\begin{cases} \dfrac{\partial L}{\partial \hat{a}} = 2\sum_{i=0}^{n-1} (y_i - \left(\hat{a} \cdot x_i + \hat{b}\right)) x_i = 0, \\ \dfrac{\partial L}{\partial \hat{b}} = 2\sum_{i=0}^{n-1} (y_i - \left(\hat{a} \cdot x_i + \hat{b}\right)) = 0. \end{cases}$$

Отсюда решение системы уравнений имеет вид

$$\begin{cases} a = \dfrac{n\sum xy - \sum x \sum y}{n\sum x^2 - (\sum x)^2}, \\ b = \dfrac{\sum y - a\sum x}{n}. \end{cases}$$

В более общем случае можно было бы записать уравнение как

$$L = \sum_{i=0}^{n-1} (y_i - X_i \boldsymbol{w}^T)^2 = 0 \text{ или } (y - X\boldsymbol{w}^T)^2 = 0;$$

тогда

$$\frac{\partial L}{\partial \boldsymbol{w}} = 2\left(y - X\boldsymbol{w}^T\right)X = 0$$

или

$$\boldsymbol{w} = (X^T X)^{-1} X^T y = X^+ y.$$

Однако если массивы X и $\boldsymbol{y}$ достаточно большие, то такое решение оказывается весьма сложным вычислительно. Для больших массивов данных чаще используют численные методы оптимизации. Среди таких методов наиболее зарекомендовал себя метод градиентного спуска.

Метод градиентного спуска позволяет итерационно решить задачу оптимизации. В контексте данного пособия мы будем называть такой процесс оптимизации *обучением*. Каждую итерацию принято называть *эпохой*. Ниже будет показан принцип обучения методом градиентного спуска и его реализация.

Важно отметить, что особенностью итеративных методов обучения является потенциальная ситуация переобучения/недообучения в ходе оптимизации. Проще всего это представить как ситуацию, в которой мы ошибемся в выбранных значениях коэффициентов. Такое явление может происходить если, например, модель будет воспринимать все шумы, помехи и искажения входных данных как важные для точного ответа. Другими словами, для данных, участвующих в обучении (обучающая выборка), наша ошибка будет стремиться к нулю. Однако для данных, отличных от обучающей выборки, точность будет невысокой. Чтобы не допустить этого, на каждом шаге обучения мы будем проверять полученные коэффициенты модели. Для такой проверки будем использовать так называемую *валидационную* выборку. Как правило, валидационная и тренировочная выборки выделяются из одних и тех же данных. В некоторых случаях кроме этих двух выборок может быть также и третья, независимая от них. Такая выборка будет необходима для проверки итоговой точности модели. Итоговую проверочную выборку можно назвать *тестовой* выборкой. По существу тестовая выборка характеризует так называемую обобщающую способность, то есть разность между точностью на тренировочных данных и тех данных, в которых модель должна работать. Разность значений точности должна быть как можно меньше.

Для того чтобы выделить из входных данных тренировочную и тестовую выборки, запишем следующую функцию: `train_test_split`. Функция будет иметь входные аргументы:

- `x, y` — входные данные и целевые значения;
- `train_size` — размер тренировочной части;
- `test_size` — размер тестовой части;
- `random_state` — состояние генератора случайных чисел;
- `shuffle` — необходимость перемешивания данных.

```
    def  train_test_split(x,y,  train_size=None,  test_
size=None, random_state=42, shuffle=True,):
        if random_state: np.random.seed(random_state)
```

```
        size = y.shape[0]
        idxs = np.arange(size)
        if shuffle: np.random.shuffle(idxs)

        if test_size and train_size is None:
            if (test_size<= 1):  train_size = 1 - test_size
            else:  train_size = size - test_size
            test_size = None

         if train_size is None or train_size > size: train_
size = size

        if (train_size<= 1): train_size *= size

        if test_size is not None:
            if test_size<= 1: test_size *= size
            if test_size>size: test_size = size-train_size
        else: test_size = 0

           x_train,  y_train  =  x[idxs[:int(train_size)]],
y[idxs[:int(train_size)]]
       x_val,   y_val   = x[idxs[int(train_size):size - int(test_
size)]], y[idxs[int(train_size):size - int(test_size)]]

        if test_size > 0:
             x_test, y_test = x[idxs[size - int(test_size):]],
y[idxs[size - int(test_size):]]
              return x_train, y_train.squeeze(), x_val, y_val.
squeeze(), x_test, y_test.squeeze()
       return x_train, y_train.squeeze(), x_val, y_val.squeeze()
```

Если есть необходимость разбить данные и на тренировочную, и на валидационную, и на тестовую выборки, тогда нужно указать размер тестовой и тренировочной выборок. В валидационную выборку попадут оставшиеся данные:

```
   x_train, y_train, x_val, y_val, x_test, y_test = train_test_
split(x, y, train_size = 0.5, test_size=0.3, )
```

В этом случае размер тренировочной выборки составит 50 точек, валидационной — 20, тестовой — 30.

Чтобы разбить исходные данные только на тренировочную и тестовую выборки, достаточно указать размер только тестовой выборки:

```
   x_train,  y_train,   x_test,  y_test  =  train_test_
split(x, y,  test_size=0.3, )
```

Тогда исходная выборка из 100 точек разобьется на тренировочную выборку, в которой 70 точек, и тестовую, в которой 30 точек.

Далее функции будут тестироваться для фиксированного `random_state = 42` и разбиения исходных данных только на тренировочную и тестовую выборки (`test_size=0.3`).

Перед началом процедуры обучения модели запишем функцию инициализации весовых параметров (коэффициентов модели) `init_weights`. Данная функция будет создавать случайный массив весовых параметров с нормальным распределением, имеющим среднее, равное 0, и разброс значений $1/\sqrt{\text{w\_shape}}$.

Кроме того, мы будем иметь возможность создавать набор весов с учетом смещения: если `add_bias = True`, то размер выходного массива на 1 больше, чем размер признаков входных данных (в нашем случае входные данные имеют один признак, а параметров будет два: коэффициент смещения и вес при признаке). Значения смещения проинициализируем нулями:

```
    def init_weights(w_shape, add_bias = True, random_
state = 42):
        w_shape = np.atleast_1d(w_shape)
        if random_state:
            np.random.seed(random_state)
        weights = np.random.randn(*list(w_shape))/np.sqrt(np.
sum(w_shape))
        if add_bias:
            weights = np.column_stack((np.zeros(weights.
shape[-1]), weights ))
        return weights.squeeze()
```

Протестируем функции: инициализируем веса и протестируем модель для первой строчки данных:

```
    weights = init_weights(x.shape[1])
    yhat = predict( x_train[0],weights)
    loss = loss_func(yhat, y[0])
```

Если все запущено правильно и везде использован `random_state = 42`, то должен получиться вектор весов $w_0 = 0.0$, $w_1 = 0.49671415$, предсказание модели $\hat{y}_0 = 0.4015424$, реальное значение целевой переменной $y_0 = 2.01974894$ и значение функции потерь $loss\left(\hat{y}_0\right) = 2.6185924$. Поскольку веса сгенерированы случай-

ным образом, неудивительно, что предсказания модели настолько разнятся с реальными значениями.

Попробуем оптимизировать веса, используем для этого метод градиентного спуска. По сути, этот метод сводится к последовательному (итерационному) пересчету значений весовых параметров обратно значениям градиента ошибки (то есть в направлении, обратном направлению роста ошибки):

$$\frac{\partial L_i}{\partial w_j} = 2\left(\hat{y}_i - y_i\right)x_{ij},$$

где $\frac{\partial L_i}{\partial w_j}$ – частная производная функции L_i по параметру w_j.

Тогда по набору всех переменных мы получим производную вида

$$\nabla_W L_i = 2\left\{\left(\hat{y}_i - y_i\right)x_{i0}, \left(\hat{y}_i - y_i\right)x_{i1}, \left(\hat{y}_i - y_i\right)x_{i2}\right\} = 2\left(\hat{y}_i - y_i\right)\times X_i^T,$$

где $\nabla_W L_i$ — градиент, то есть набор частных производных функции L_i по набору $\{w_j\}$.

Реализуем соответствующую функцию на Python с учетом возможности добавления смещения:

```
def grad_loss(y_hat, y, X, add_bias = True):
    if add_bias:
        X_full = np.column_stack((np.ones(X.shape[0]),X))
    else:
        X_full = X
    return 2*np.dot(X_full.T, (y_hat - y)) / y.size
```

Оценим градиент весов для первой точки:

```
grad = grad_loss(yhat, y[0], x[0])
```

Должны получиться следующие значения: [−3.23641307; −0.01787185].

Обозначим номер итерации как *t*, тогда выражение для обновления весовых параметров можно записать как

$$\boldsymbol{w}^t = \boldsymbol{w}^{t-1} - \eta\nabla_{\boldsymbol{w}} L\left(\hat{y}_i, y_i\right) = \boldsymbol{w}^{t-1} - 2\eta\left(\hat{y}_i - y_i\right)\times X_i^T,$$

где η – коэффициент, с которым изменяются значения весовых параметров, так называемая скорость обучения (Learning Rate).

Реализуем это в виде функции `update_weights`:

```
def update_weights(grad, weights, learning_rate):
    return weights - learning_rate*grad
```

Теперь проведем обновление весовых параметров:

```
weights = update_weights(grad, weights, 0.1)
```

После первой итерации обновления весов должны получиться следующие значения: $w_0 = 0.32364131$, $w_1 = 0.49850134$, предсказание модели для нулевой точки $\hat{y}_0 = 0.72662847$, а функция потерь $loss\left(\hat{y}_0\right) = 1.67216056$. Функция потерь уменьшилась по сравнению с нулевым шагом итерации, но надо повторить операцию оценки градиента весов и последующего обновления несколько раз, чтобы значение функции потерь приблизилось к нулю.

Создадим процедуру итерационного обучения. Процедура будет повторять процесс пересчета весов методом градиентного спуска заданное число раз (`epochs`). Функция будет требовать на вход:

- `X` — набор входных значений в формате: число записей × признаки в записи;
- `y` — набор целевых переменных;
- `weights` — начальные значения весовых параметров;
- `learning_rate` — скорость обучения;
- `epochs` — число эпох обучения.

Функция дает на выходе:

- `weights` — набор обученных весовых параметров;
- `cost` — значение функционала потерь на каждой эпохе обучения.

Отметим также, что на практике можно обновлять весовые параметры не для каждого отдельного значения i, а для целого набора таких значений, тогда более верное выражение будет выглядеть как

$$\boldsymbol{w}^t = \boldsymbol{w}^{t-1} - \eta \frac{1}{n} \sum_{i=0}^{n-1} \nabla_W L\left(\hat{y}_i, y_i\right),$$

где n — объем выборки.

```
def fit(X, y, learning_rate, weights = None,  epochs=30):

    if weights is None: weights = init_weights(X.shape[1])
```

```
    cost    = np.zeros(epochs)

    for i in range(epochs):
        yhat    = predict(X,weights)
        grad    = grad_loss(yhat,  y, X)
       weights = update_weights(grad, weights, learning_rate)
        cost[i] = loss_func(yhat,  y).mean()

    return weights, cost
```

Протестируем обучение на 10 эпохах при скорости обучения 0.1:

```
weights, cost = fit(x_train, y_train, learning_rate=0.1,
epochs=25)
```

Можно визуализировать полученные значения функционала потерь для каждой эпохи обучения (рис. 3.4).

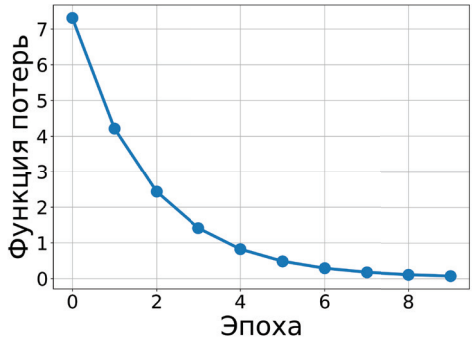


Рис. 3.4. Зависимость функционала потерь от эпохи для модели, обученной с помощью функции `fit`

Как видно, за 10 эпох обучения функционал потерь значительно уменьшился. При этом получены следующие веса: $w_0 = 2.04548417$, $w_1 = 1.55145866$, что достаточно близко к реальным значениям.

Визуализируем предсказания модели (рис. 3.5). Можно сказать, что предсказания модели качественно похожи на истинные значения, но давайте оценим модель количественно. Для этого воспользуемся метрикой *коэффициент детерминации* r^2 (Coefficient of Determination). Метрика соответствует относительной среднеквадратичной ошибке, она может быть рассчитана как

$$r^2 = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2},$$

где $\bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i$ – среднее значение.

```
    def r2_score(yhat, y):
        return 1-(np.square(y-yhat)).sum(axis=0)/(np.square(y-
np.mean(y, axis=0))).sum(axis=0)
```

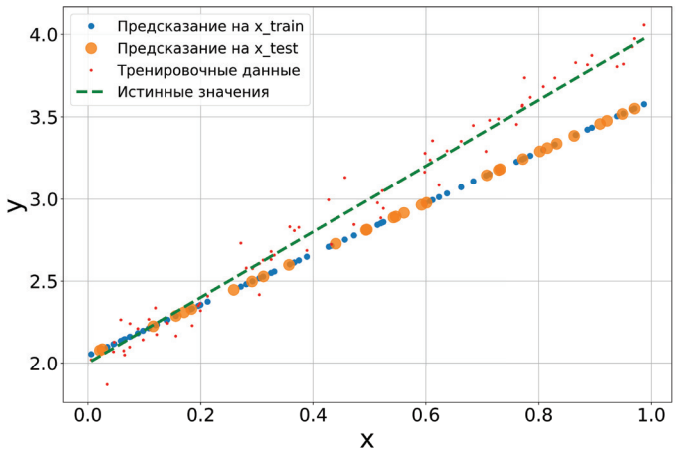


Рис. 3.5. Модель, полученная с помощью функции `fit`

Отметим важное обстоятельство: для расчета градиента мы использовали функцию потерь, однако для оценки качества модели мы пользуемся другой функцией-метрикой. Дело в том, что значения функции потерь, сколь небольшими бы они ни были, очень сложно интерпретировать. Более того, можно ожидать, что для другого метода оптимизации значения могли быть и другими. Таким образом, значения функции потерь не подходят для оценки качества модели. Качество работы модели, как правило, определяется по некоторым метрикам. Такие метрики должны быть интерпретируемыми и едиными для всех сравниваемых оценщиков. В нашем случае метрика соответствует отношению среднего квадрата ошибки к дисперсии целевых значений. Тогда результат работы метрики мы можем интерпретировать как среднюю ошибку предсказания для нашей модели. Чем ниже эта ошибка (для разных моделей), тем лучше.

Помимо коэффициента детерминации для оценки моделей регрессии могут использоваться следующие метрики:

- Mean Square Error – среднеквадратичная ошибка:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2;$$

- Root Mean Square Error – средняя квадратическая ошибка:

$$\mathrm{RMSE} = \sqrt{\mathrm{MSE}};$$

- Mean Average Error – средняя абсолютная ошибка:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|;$$

- Mean Absolute Percentage Error – средняя абсолютная процентная ошибка:

$$\mathrm{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|;$$

- Mean Squared Logarithmic Error – среднеквадратическая логарифмическая ошибка:

$$\mathrm{MSLE} = \frac{1}{n}\sum_{i=1}^{n}\{\ln(1+y_i) - \ln\left(1+\hat{y}_i\right)\}^2;$$

- максимальная ошибка:

$$\mathrm{MaxError} = \max(|y_i - \hat{y}_i|).$$

Применим функцию `r2_score`. Для тренировочных данных должно получиться значение 0.8570755941810686, а для тестовых — 0.818294732486739.

Очевидным способом увеличения качества модели является увеличение количества эпох. На практике, как правило, на каждой эпохе рассматривается не вся выборка, а только некоторая ее часть — так называемый *батч* (мини-пакет). Это позволяет быстрее рассчитывать градиент весов и чаще обновлять веса. Запишем функцию для генерации батчей заданного размера `batch_size`:

```
def load_batch(X,y, batch_size = 100):
    idxs = np.arange(y.size)
    np.random.shuffle(idxs)

    for i_batch in range(0,y.size,batch_size):
        idx_batch = idxs[i_batch:i_batch+batch_size]
```

```
            x_batch   = np.take(X, idx_batch,axis=0)
            y_batch   = np.take(y, idx_batch)
            yield  x_batch, y_batch
```

Обновим функцию fit с учетом разбиения данных на батчи:

```
    def fit_SGD(X, y, lerning_
rate, weights = None,  epochs=30, batch_size = 100, random_
state = 42):
        if random_state: np.random.seed(random_state)

        if weights is None: weights = init_weights(X.shape[1])
        if batch_size is None or batch_size>y.size : batch_
size = y.size
        n_batches = y.size//batch_size

        cost    = np.zeros(epochs)

        for i in range(epochs):
            loss = 0
            for cnt,(x_batch, y_batch) in enumerate(load_
batch(X,y, batch_size)):

                yhat    = predict(x_batch, weights)
                grad    = grad_loss(yhat,  y_batch, x_batch)
                weights = update_weights(grad, weights, lerning_
rate)
                loss   += loss_func(yhat,  y_batch).mean()

                if cnt>= n_batches:
                    break
            cost[i] = loss/n_batches

        return weights, cost
```

Таким образом, на каждой эпохе обучения веса обновляются столько раз, на сколько батчей можно разделить исходную выборку. В предельном случае реализуется стохастический градиентный спуск, при котором обновление происходит при случайном выборе одного элемента.

Визуализируем зависимость функционала потерь от эпох при обучении модели линейной регрессии с использованием функции fit_SGD (рис. 3.6), а также визуализируем предсказания модели fit_SGD на тренировочных и тестовых данных (рис. 3.7).

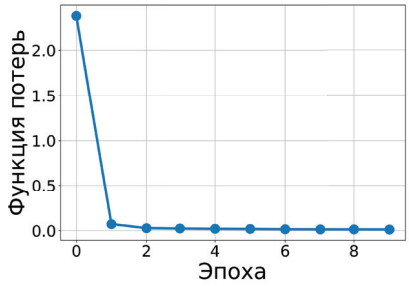


Рис. 3.6. Зависимость функционала потерь от эпохи для модели, обученной с помощью функции `fit_SGD`

При этом получены следующие веса: $w_0 = 2.08073489$, $w_1 = 1.85005883$, что еще ближе к реальным значениям.

Для тренировочных данных должно получиться значение 0.9685919335230442, а для тестовых – 0.9715819769095048.

Мы получили достаточно интересный результат. Количество эпох было одинаковое, в обоих случаях мы использовали одинаковое число данных. Однако если мы предварительно разбиваем выборку и обновляем веса на каждой части отдельно, то результат обучения модели лучше. Такую идею можно применить и в реальной жизни: не обязательно решать большую задачу целиком, можно разбить ее на несколько небольших подзадач.

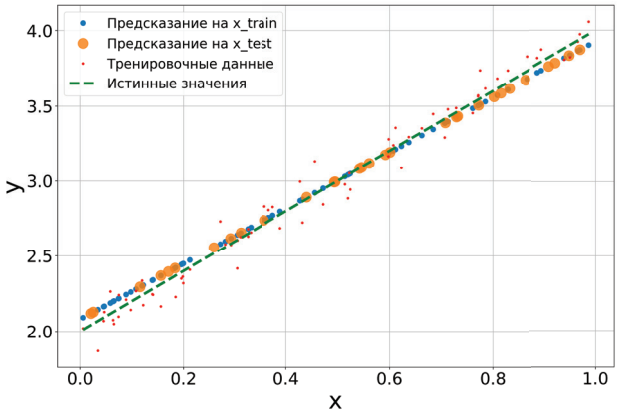


Рис. 3.7. Модель, полученная с помощью функции `fit_SGD`

В приложении 1 представлена реализация описанного нами алгоритма не с помощью функций, как в этой главе, а с помощью единого

класса `LinearRegression`. В основном методы, реализованные в этом классе, совпадают с описанными ранее функциями. Ознакомьтесь с приложением самостоятельно. Подход ООП (объектно-ориентированного программирования) позволит легче модифицировать алгоритм в последующих главах.

Полиномиальная регрессия

Теперь попробуем применить разработанный нами алгоритм для полиномиальной зависимости (рис. 3.2). Сколько бы мы ни добавляли эпох в обучение, наиболее вероятно, что предсказания модели будет выглядеть следующим образом (рис. 3.8).

Видно, что линейная модель слишком проста для такого распределения истинных значений целевой переменной (высокое смещение). Давайте попробуем усложнить модель. Поскольку сгенерировать дополнительные новые признаки неоткуда, будем использовать имеющиеся данные. Как известно, члены полиномов различных целых степеней можно считать независимыми признаками. Поэтому полиномиальная регрессия может быть представлена как многопеременная линейная регрессия.

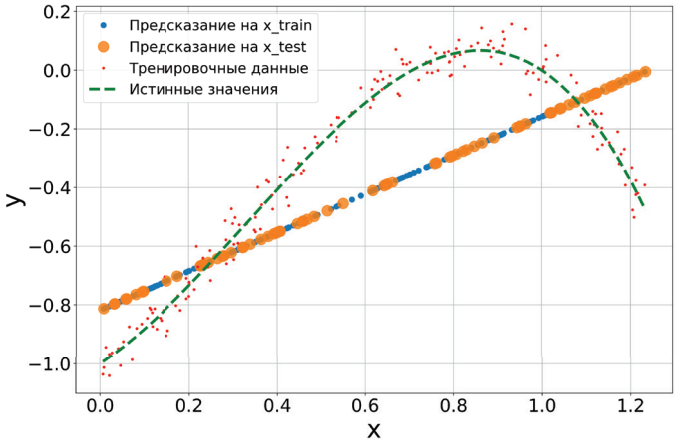


Рис. 3.8. Полученная линейная модель для полиномиальной зависимости

Для реализации запишем функцию `to_polynom`, создающую полином из входных данных `x`. На выходе функция даст массив, имеющий число столбцов, равное степени искомого полинома:

```
def to_polynom(X, order = 2):
    order_range = range(order, order+1,1)
    out = np.copy(X)
    for i in order_range:
        out = np.hstack([out, np.power(X,i)])
    return out
```

Теперь, если предварительно сгенерировать полиномиальные признаки и затем применить к ним алгоритм линейной регрессии, можно получить гораздо лучший результат (рис. 3.9).

```
    x_ = to_polynom(x, order = 5)
    x_train, y_train,  x_test, y_test = train_test_
split(x_, y,  test_size=0.3, )
    regr = LinearRegression(learning_rate=0.1,epochs=100,batch_
size=10, n_batches=None)
    regr.fit(x_train, y_train)
```

Использование полиномов от исходных признаков — достаточно простой, но работающий метод инженерии признаков, который можно использовать на начальном этапе работы с данными. Можно считать, что степень полиномов — тоже гиперпараметр модели линейной регрессии. Однако, как и любой другой инструмент, его нужно использовать с осторожностью. В частности, необходимо проверять, какой степени полинома будет достаточно для конкретных данных.

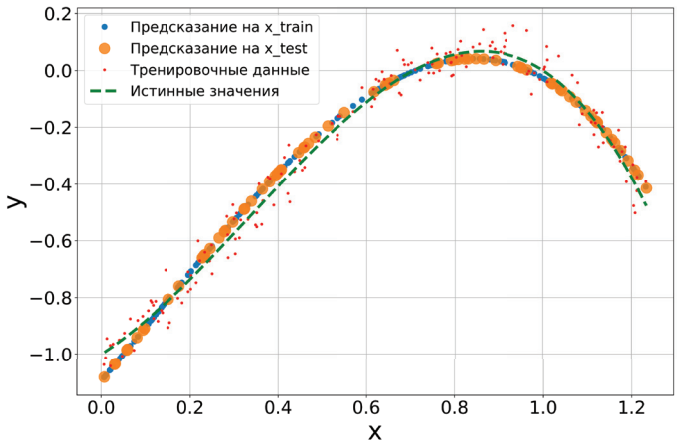


Рис. 3.9. Линейная модель, полученная для полиномиальной зависимости с использованием полиномиальных признаков

Регуляризация линейной регрессии

Регуляризация Тихонова

Как было сказано ранее, бывает так, что обычный градиентный спуск приводит к переобучению модели. Напомним: переобучение — это ситуация, при которой точность на обучающих данных значительно выше, чем на тестовых. В таких случаях также можно сказать, что данные плохо обусловлены, то есть любые небольшие изменения по отношению к тренировочной выборке приведут к большим изменениям в ответе модели. В целом это будет означать, что модель дает очень большой разброс результатов.

Такой разброс может быть снижен при помощи различных техник регуляризации. Смысл использования таких техник сводится к тому, что при обучении модели к выражению обновления весовых параметров добавляется дополнительное условие. Например, можно добавить условие: ограничение суммы квадратов весовых параметров. Такое предположение называется *регуляризацией Тихонова* или *гребневой регуляризацией* (а также L2-регуляризацией).

Технически подобная регуляризация соответствует предположению, что распределение результатов работы модели имеет вид нормального распределения. Такое предположение часто допустимо и оправданно. Функция потерь для регуляризации Тихонова может быть записана в следующей форме:

$$\begin{cases} L\left(\hat{y}_i, y_i\right) \to \min \\ \| \boldsymbol{w} \|_2^2 < \text{const} \end{cases} \to L\left(\hat{y}_i, y_i\right) + \frac{\lambda}{2} \sum_{j=1}^{p} w_j^2 \to \min,$$

где $\| \boldsymbol{w} \|_2^2 = \sum_{j=1}^{p} w_j^2$ — норма Фробениуса для вектора или матрицы; λ — регуляризационный множитель; p — размер вектора весовых параметров.

Отметим также, что смещение не регуляризуется. Как правило, регуляризационный множитель задается в диапазоне от 0 до 0.1, часто проверяются значения в логарифмическом масштабе (0.1; 0.01; 0.001 и т. д).

Оптимизацию с использованием регуляризации можно интуитивно представить в виде следующего графика (рис. 3.10).

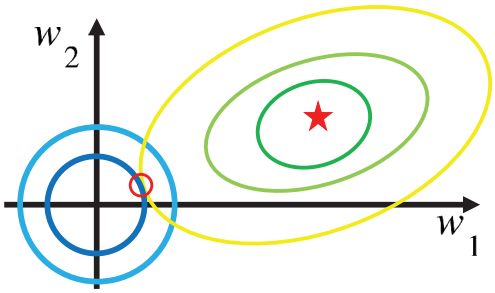


Рис. 3.10. Схематичное изображение L2-регуляризации

Звезда на графике обозначает минимум функции потерь для линейной регрессии. Желтыми и зелеными эллипсами схематично отображаются комбинации весов, при которых функция потерь принимает одно и то же значение. Окружности в центре координат символизируют ограничение, задаваемое регуляризации. Таким образом, при регуляризации находится своеобразный компромисс: необходимо, чтобы веса были на окружности наименьшего радиуса и функция потерь при этом принимала как можно меньшее значение. «Удельный вес» вклада регуляризации и функции потерь задается регуляризационным множителем.

Закон изменения весовых параметров для данной модели можно записать как

$$\boldsymbol{w}^t = \boldsymbol{w}^{t-1} - \eta \frac{1}{n} \sum_{i=0}^{n-1} \nabla_{\boldsymbol{w}} L\left(\hat{y}_i, y_i\right) - \lambda \boldsymbol{w}^{t-1}.$$

Запишем новую версию регрессии `RidgeRegression` на основе уже созданного класса `LinearRegression`. Для этого запишем новый класс, наследующий от уже созданного, и перепишем в нем методы `.loss` и `.update`. Реализация L2-регуляризации представлена в прил. 2.

Регуляризация L1

В ряде случаев, когда разброс данных оказывается очень большим, регуляризации L2 может оказаться бесполезной или даже вредной. Дело в том, что в функции потерь учитываются веса в квадрате и большие колебания весовых параметров в квадрате приведут к большим колебаниям в значениях функции потерь. Часто эта ситуация является недопустимой.

В таких случаях следует выбирать более устойчивые (робастные) методы. Робастные методы могут быть менее точными, однако более стабильными. Одним из наиболее распространенных робастных методов является *L1-регуляризация*, или *регуляризация Лассо*. В этом случае выражение для функции потерь может быть записано следующим образом:

$$\begin{cases} L\left(\hat{y}_i, y_i\right) \to \min \\ \| \boldsymbol{w} \|_1^1 < \text{const} \end{cases} \to L\left(\hat{y}_i, y_i\right) + \lambda \sum_{j=1}^{p} w_j \to \min,$$

где $\| \boldsymbol{w} \|_1^1 = \sum_{j=1}^{p} w_j$ — норма L1 для вектора или матрицы; λ — регуляризационный множитель; p — размер вектора весовых параметров.

Изменение нормы L2 на L1 приводит к достаточно интересному эффекту (рис. 3.11).

Теперь вместо ограничивающей окружности – ограничивающий ромб. Что интересно, такой «угловатый» ограничитель приводит к тому, что для некоторых весов компромисс регуляризации попадет на ось координат. Это, в свою очередь, приводит к тому, что часть весов обнулится, вследствие чего от части признаков модель избавится самостоятельно, что бывает полезно в случае анализа данных с большим числом признаков.

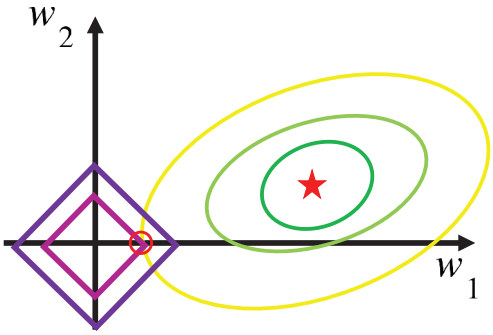


Рис. 3.11. Схематичное изображение L1-регуляризации

Закон изменения весовых параметров для данной модели можно записать как

$$\boldsymbol{w}^t = \boldsymbol{w}^{t-1} - \eta \frac{1}{n} \sum_{i=0}^{n-1} \nabla_w L\left(\hat{y}_i, y_i\right) - \lambda sign\left(\boldsymbol{w}\right).$$

Запишем новую версию регрессии `LassoRegression` на основе уже созданного класса `LinearRegression`. Для этого запишем новый класс,

наследующий от уже созданного, и перепишем в нем методы `.loss` и `.update`. Реализация L1-регуляризации представлена в прил. 3.

Эластичная регуляризация

Отметим, что во многих случаях неизвестно, какая модель регуляризации окажется лучше, поэтому целесообразно использовать обе. Такая модель называется *эластичной регуляризацией*. Запишем новую версию регрессии `ElasticRegression` на основе уже созданного класса `LinearRegression`. Для этого запишем новый класс, наследующий от уже созданного, и перепишем в нем методы `.loss` и `.update`. Эластичная регуляризация представлена в прил. 4.

Как правило, имеет смысл протестировать различные варианты регуляризации, а также различные значения регуляризационных множителей. Все это тоже можно отнести к гиперпараметрам модели линейной регрессии.

Практические задания

1. Используя функцию `dataset`, сгенерируйте линейные зависимости с другими параметрами `a, b, N, noise_power` (см. с. 49) и проверьте модель линейной регрессии на этих данных.
2. Используя функцию `dataset`, сгенерируйте гармонические зависимости с другими параметрами (см. с. 50) и проверьте различные степени полиномов исходных данных и разные типы регуляризации. Посмотрите, что произойдет, если увеличить регуляризационный множитель.
3. Используйте любую из подготовленных вами моделей линейной регрессии для предсказания цены автомобилей в наборе данных Cars. Для оценки качества модели используйте отложенную выборку и несколько метрик регрессии. Сравните результаты модели при использовании только числовых признаков и при добавлении категориальных признаков с помощью One-Hot-кодирования.

4*. Сравните работу реализованных алгоритмов с функциями библиотеки scikit-learn:

- простая линейная регрессия через метод наименьших квадратов `sklearn.linear_model.LinearRegression`;
- простая линейная регрессия через градиентный спуск `sklearn.linear_model.SGDRegressor`;
- регрессия с регуляризацией Тихонова `sklearn.linear_model.Ridge`;
- регрессия с L1-регуляризацией `sklearn.linear_model.Lasso`;
- эластичная регуляризация `sklearn.linear_model.ElasticNet`.

Контрольные вопросы

1. Перечислите возможные гиперпараметры модели линейной регрессии.
2. Может ли коэффициент детерминации быть отрицательным числом?
3. Оцените MSE для следующих данных: реальные значения $\boldsymbol{y}$ {1, 2, 3, 4}, предсказания модели $\hat{y}$ {2, 1, 4, 6}.
4. Предположим, что у вас есть вектор весов $\boldsymbol{w}$ {10, 5, 6}. Вы посчитали градиент функции потерь, который равен {20, −10, 40}. Посчитайте обновленный вектор весов при условии, что скорость обучения составляет 0.1.
5. Перечислите данные, которые вам необходимы для расчета градиента функции потерь.
6. Вы выполнили обучение линейной модели дважды: с регуляризацией и без. У вас есть два вектора весов модели $\boldsymbol{w}_1$ {14.37, 22.80, 32.20} и $\boldsymbol{w}_2$ {0.69, 2.02, 4.20}, но вы не помните, какой вектор весов какой модели соответствует. Как вы считаете, который из приведенных весов соответствует случаю регуляризации?
7. Вы получили веса модели $\boldsymbol{w}$ {3, −2, 2}. В модели не используется смещение. Оцените предсказание модели для следующих значений параметров $\boldsymbol{x}$ {1, 3, 1}.
8. Оцените коэффициент детерминации для следующих данных: реальные значения $\boldsymbol{y}$ {1, 2, 3, 4}, предсказания модели $\hat{y}$ {2, 1, 4, 6}.

Глава 4.
Логистическая регрессия

Один из самых простых методов классификации – это логистическая регрессия. По существу, модель логистической регрессии представляет собой аналог линейной регрессии.

Интересный исторический факт. Был период, когда логистическая регрессия использовалась для классификации задачи выживания: если вы выпьете пузырек с ядом, вас, скорее всего, назовут живым или умершим? Времена изменились. Сегодня вызов экстренных служб дает лучший ответ на этот вопрос, а логистическая регрессия лежит в основе глубокого обучения.

Логистическая функция была изобретена в 1830-е гг. бельгийским статистиком П. Ф. Ферхюльстом для описания динамики населения: со временем первоначальный взрыв экспоненциального роста сглаживается по мере того, как происходит потребление доступных ресурсов, что приводит к характерной логистической кривой. Прошло более века, прежде чем американский статистик Э. Б. Уилсон и его ученица Дж. Вустер разработали логистическую регрессию, чтобы выяснить, какое количество данного опасного вещества может привести к летальному исходу.

Логистическая функция описывает широкий спектр явлений с достаточной точностью, поэтому логистическая регрессия обеспечивает полезные базовые прогнозы во многих ситуациях. В медицине она оценивает смертность и риск заболевания, в политической науке – предсказывает победителей и проигравших на выборах, в экономике она прогнозирует перспективы бизнеса. Что еще более важно, функция управляет частью нейронов, в которых нелинейность является сигмовидной, в самых разных нейронных сетях.

Генерируемые данные

Как и в случае с линейной регрессией, сначала сгенерируем данные. В этом учебно-методическом пособии мы рассмотрим только простой вариант классификации – *бинарную классификацию*. В этот раз мы воспользуемся готовыми функциями, которые реализованы в библиотеке scikit-learn, а именно `make_classification` для линейно разделимых данных, `make_moons` для данных, распределенных в виде знака инь-ян, и `make_circles` для данных, распределенных в виде концентрических кругов.

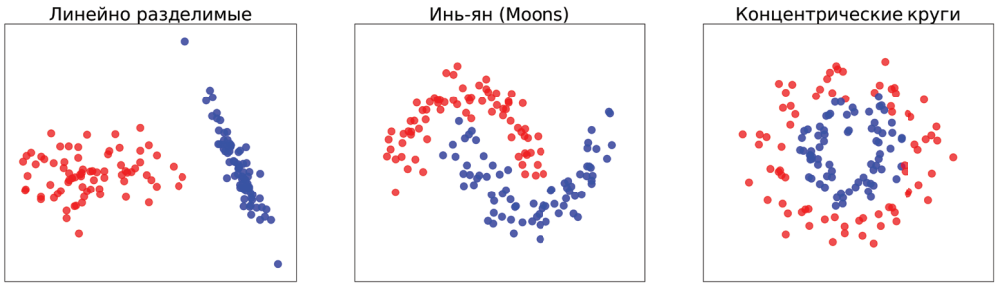


Рис. 4.1. Типы генерируемых данных

Запишем отдельную единую функцию, которая будет генерировать данные. Входными параметрами этой функции будет N – количество точек, method – тип данных, который мы хотим сгенерировать, noises – уровень шума в данных. Выходными параметрами будут признаки – матрица X с размерами $N \times 2$ и вектор y меток классов для каждой точки.

```
    from sklearn.datasets import make_moons, make_
circles, make_classification
    def make_bin_clf(N, method = 'line', noises = 0.15, random_
state = 42):

        if random_state: rng = np.random.
RandomState(seed = random_state)

        if method == 'line' or method is None:
            X, y = make_classification(n_samples=N, n_
features=2, n_redundant=0, n_informative=2, n_clusters_per_
class=1, class_sep=2)
            X += np.random.randn(*X.shape) *noises
```

```
        elif method == 'moons':
            X, y = make_moons(n_samples=N, noise=noises)

        elif method == 'circles':
            X, y = make_circles(n_
samples=N, noise=noises, factor=0.5)

        return X,y
```

Для простоты начнем работу с набором данных, который линейно разделим.

Модель логистической регрессии

В случае линейной регрессии требовалось провести линию через данные таким образом, чтобы в среднем отклонение точек было минимальным. С точки зрения бинарной классификации необходимо предсказывать всего два числа, допустим, 0 и 1. Теоретически можно использовать модель линейной регрессии, но есть фундаментальная загвоздка: в общем случае линейная регрессия выдает ответ в диапазоне $(-\infty, \infty)$. Да, наверняка с помощью градиентного спуска можно подобрать такие веса, которые будут давать необходимый ответ. Но можно и «помочь» модели — обернуть ее в некоторую функцию, которая преобразует данные в нужный диапазон. Примером такой функции является *сигмоида*, или *логистическая функция*, которая упоминалась в главе 1. Эта функция описывается следующим уравнением:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

На рисунке 4.2 представлен график функции сигмоиды на интервале от –5 до 5.

В Python реализовать сигмоиду тоже достаточно просто:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

В итоге мы получили следующую модель, которая представляет собой логистическую функцию, применяемую к модели линейной ре-

грессии. Отсюда название модели — логистическая регрессия, несмотря на то что решается задача классификации. Функционально модель можно описать следующим образом:

$$\hat{y}_i = \sigma\left(\sum_{j=0}^{p} w_j X_{ij}\right) \equiv \sigma\left(\sum_{j=1}^{p} w_j X_{ij} + b\right) = \sigma(z_i),$$

где σ — функция сигмоиды; $\hat{y}_i$ — результат принятия решений, $z_i = \sum_{j=0}^{p} w_j X_{ij}$.

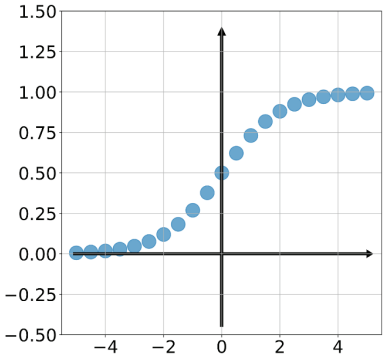


Рис. 4.2. График сигмоиды

Из некоторых статистических выводов известно, что для такой модели необходимо выбрать функцию потерь следующего вида:

$$L\left(y, \hat{y}\right) = -\frac{1}{n}\sum_{i=0}^{n-1}\left[y_i \ln \hat{y}_i + (1 - y_i)\ln\left(1 - \hat{y}_i\right)\right].$$

Эта функция потерь называется *бинарной кросс-энтропией*.

Прежде чем записать выражение для градиента функции потерь, запишем выражение для производной функции активации:

$$\sigma'(z_i) = \frac{\partial \sigma(z_i)}{\partial z_i} = (1 - \sigma(z_i))\sigma(z_i).$$

Градиент функции потерь для одного элемента выборки может быть выражен следующим образом:

$$\nabla_w L_i = -\left(\frac{y_i}{\sigma\left(X_i \boldsymbol{w}^T\right)} - \frac{1 - y_i}{1 - \sigma\left(X_i \boldsymbol{w}^T\right)}\right)\sigma'\left(X_i \boldsymbol{w}^T\right) \times X_i =$$

$$= -\left(y_i - \sigma\left(X_i \boldsymbol{w}^T\right)\right) \times X_i = -\left(y_i - \hat{y}_i\right) \times X_i.$$

Тогда правило обновления весовых параметров может быть записано как

$$w^t = w^{t-1} - \eta \frac{1}{n} \sum_{i=0}^{n-1} \left(\hat{y}_i - y_i \right) \times X_i .$$

Отметим, что данное выражение эквивалентно записанному для линейной регрессии с точностью до коэффициента 2, поэтому мы учтем данный параметр путем замены $\eta \to \eta / 2$.

Как правило, после расчета функции сигмоиды необходимо округлить значения до 0 или до 1, то есть до значения метки одного из классов. Такое округление можно сделать по заданному порогу. Например, можно сказать, что если значение сигмоиды больше 0.5, то пусть класс будет 1, а если меньше, то наоборот. Однако на практике иногда ставят высокий порог, 0.7–0.8.

Запишем функцию определения класса:

```
def to_class(logit, threshold = 0.7):
    return (logit>=threshold)*1
```

В описанном смысле можно говорить о том, что результат применения функции сигмоиды – это вероятность того, что аргумент функции $\sigma(z_i)$ принадлежит одному из классов. Такой аргумент принято называть *логит*. Отметим, что для расчета функции потерь не следует пользоваться округлением до классов.

Теперь запишем функцию потерь. В значениях логарифма мы ввели небольшую константу `eps` с целью исключить ошибку вида «логарифм нуля»:

```
def bce_loss(yhat, y):
    eps = 1e-6
    return -(y*np.log(yhat + eps)+(1-y)*np.log(1-yhat+eps))
.mean()
```

Запишем все в один класс `LogisticRegression`. Однако не будем писать класс с нуля, а создадим его на основе имеющегося класса `ElasticRegression`. Это позволит использовать готовые наработки, что адекватно, поскольку градиентный спуск применяется одинаково в случае и линейной, и логистической регрессии с точки зрения самого алгоритма. Меняются модель и функция потерь. Новый класс `LogisticRegression` представлен в прил. 5.

Дополнительно в классе `LogisticRegression` реализован метод `.plot_desicion_function`, который позволяет визуализировать в двумерном пространстве вероятность принадлежности отдельной точки к конкретному классу – функцию принятия решений. На рисунке 4.3 представлена визуализация функции принятия решений для линейно разделимых данных.

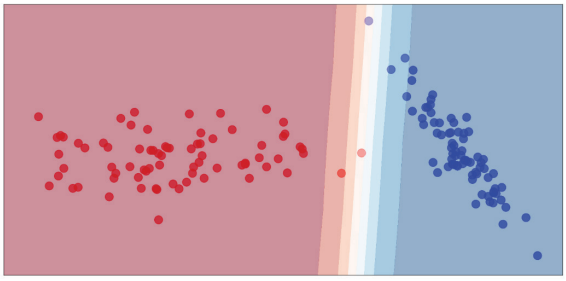


Рис. 4.3. Визуализация функции принятия решений для линейно разделимых данных

Как и ожидалось, логистическая регрессия разделяет классы с помощью линий, которые проводятся таким образом, чтобы минимизировать функцию потерь. Однако это не всегда возможно сделать. Так, для того чтобы классифицировать набор данных в виде концентрических кругов, необходимо предварительно сгенерировать полиномиальные признаки, чтобы усложнить модель. Пример успешной классификации концентрических кругов с помощью логистической регрессии и полиномиальных признаков до второй степени представлен на рис. 4.4.

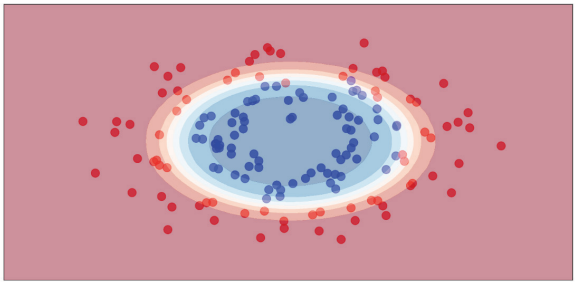


Рис. 4.4. Визуализация функции принятия решений для концентрических кругов

Дополнительно в классе `LogisticRegression` реализован метод `.classification_report`, в котором представлен расчет различных метрик классификации. Давайте остановимся на них отдельно.

Метрики классификации

Рассмотрим пример ошибок классификации (рис. 4.5). У нас есть два класса, красный и зеленый, при этом красный класс – основной. Идеальная модель – зеленая, она отделяет все красные точки от зеленых. Но, допустим, в ходе обучения получилась красная модель. И есть два типа ошибок, которые красная модель совершила. К ошибкам первого рода относят неправильно отнесенные к основному классу объекты – те зеленые, которые модель ошибочно считает красными. К ошибкам второго рода относят пропуски – те красные точки, которые не вошли в предсказании модели.

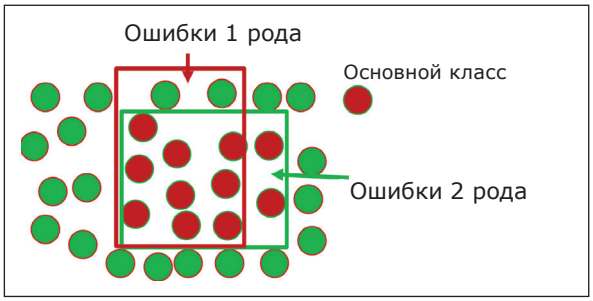


Рис. 4.5. Ошибки классификации

Соотношения между правильными предсказаниями, ошибками первого и второго рода и лежат в основе оценки моделей классификации. Для удобства заполняется *матрица ошибок* (Confusion Matrix), пример которой представлен на рис. 4.6.

	$y = 1$	$y = 0$
$\hat{y} = 1$	True Positive (TP)	False Positive (FP)
$\hat{y} = 0$	False Negative (FN)	True Negative (TN)

Рис. 4.6. Матрица ошибок

В соответствующие ячейки матрицы вставляются количество правильно предсказанных объектов основного класса (True Positive — TP), количество правильно предсказанных объектов неосновного класса (True Negative — TN), количество ошибок 1 рода (False Positive — FP), количество ошибок 2 рода (False Negative).

Затем по матрице ошибок оцениваются различные метрики классификации:

- Accuracy — доля правильных ответов алгоритма:

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}};$$

- Precision — точность:

$$\mathrm{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}};$$

- Recall — полнота:

$$\mathrm{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}};$$

- Specificity — специфичность:

$$\mathrm{specificity} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}};$$

- F_1-мера:

$$F_1 = \frac{2 \times \mathrm{precision} \times \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}.$$

Наиболее очевидная метрика — доля правильных ответов — хорошо работает в случае сбалансированных классов, т. е. когда число объектов разных классов совпадает. Однако в случае несбалансированных классов эта метрика может вводить в заблуждение: допустим, есть два класса, в первом классе 900 объектов, во втором — 100. Если модель будет предсказывать все объекты как первый класс, то доля правильных ответов будет 0.9 или 90 %, что вроде неплохо, но второй класс модель совсем не предсказывает.

Для уточнения предсказательной способности модели вводят дополнительные метрики, которые оценивают долю ошибок первого и второго рода отдельно (точность, полнота, специфичность). Иногда для удобства используют F_1-меру, среднее гармоническое между точностью и полнотой.

В классе `LogisticRegression` для оценки модели классификации реализован метод `.score`, который вычисляет метрику Accuracy.

Практические задания

1. Сгенерируйте линейно разделимые данные с другими параметрами и проверьте модель логистической регрессии на этих данных. Проанализируйте метрики классификации.
2. Сгенерируйте данные, распределенные как знак инь-ян или концентрические круги. Проверьте различные степени полиномов исходных данных и различные типы регуляризации для достижения наилучшего качества классификации. Проанализируйте метрики классификации.
3. Используйте модель логистической регрессии для предсказания типа трансмиссии автомобилей в наборе данных Cars. Для оценки качества модели используйте отложенную выборку и несколько метрик классификации. Сравните результаты модели при использовании только числовых признаков и при добавлении категориальных признаков с помощью One-Hot-кодирования.

4*. Сравните работу реализованных алгоритмов с функцией библиотеки scikit-learn — логистической регрессией `sklearn.linear_model.LogisticRegression`.

Контрольные вопросы

1. Допустим, тест на некое заболевание R дал положительный ответ, хотя на самом деле у испытуемого нет этого заболевания. Какую ошибку допустил тест?
2. Пусть в матрице ошибок TP = 5, TN = 90, FP = 10, FN = 5. Оцените метрики классификации для такой матрицы ошибок.
3. Допустим, есть два классификатора: первый классификатор имеет долю правильных ответов 95 %, чувствительность 99 %, специфичность 50 %; второй классификатор имеет долю пра-

вильных ответов 87 %, чувствительность 84 %, специфичность 94 %. Что вы можете сказать о данных, используемых для классификации? Какой из этих классификаторов надежнее (при условии, что важно определение обоих классов)?

4. Перечислите возможные гиперпараметры модели логистической регрессии.
5. Для набора данных Cars проанализируйте веса моделей при использовании только числовых признаков. Назовите параметр, который в наибольшей степени связан с целевой переменной.
6. Оцените значение функции сигмоиды $\sigma(z)$ для $z = 0.25$.
7. Оцените значение производной функции сигмоиды $\sigma'(z)$ для $z = -3$.
8. Назовите, к какому классу следует отнести результат логистической модели для $z = 0.1$, если порог равен 0.6.
9. Оцените значение функции потерь (бинарной кросс-энтропии) для предсказания модели $\hat{y} = 0.1$ и целевой переменной $y = 1$.

Глава 5.
Уменьшение размерности

В этой главе мы рассмотрим базовый подход к уменьшению размерности, который относится к методу разложения матриц — *метод главных компонент* (Principal Components Analysis). Это метод снижения размерности данных путем преобразования их в такую форму, чтобы оставить только максимально полезную информацию. Под термином *полезная информация* в данном методе понимается набор независимых друг от друга признаков с максимальной дисперсией (с максимальным разбросом значений). При этом мы изначально полагаем, что интенсивность (в некотором смысле) полезной информации преобладает над интенсивностью каких-то случайных искажений или других «неидеальностей» нашего набора данных.

Для выделения полезной информации в методе главных компонент проводится преобразование данных от набора исходных столбцов (исходных признаков), которые могут содержать шумы и быть линейно зависимыми, к набору новых столбцов, которые обладают важным свойством линейной независимости (не коррелируют). Новые столбцы можно изобразить как некоторую систему координат, в которых можно отложить точки — данные. При этом часто оказывается так, что некоторые из координат не нужны: в них почти наверняка нет информации.

Генерируемые данные

Рассмотрим пример на данных, структуру которых мы можем контролировать. Это можно представить так: возьмем двумерную фигуру — эллипс с разными радиусами (рис. 5.1, *а*). При его рассмотрении может оказаться, что вторая ось системы координат не нужна. Таким образом, этот случай можно рассмотреть как одномерный (рис. 5.1, *б*).

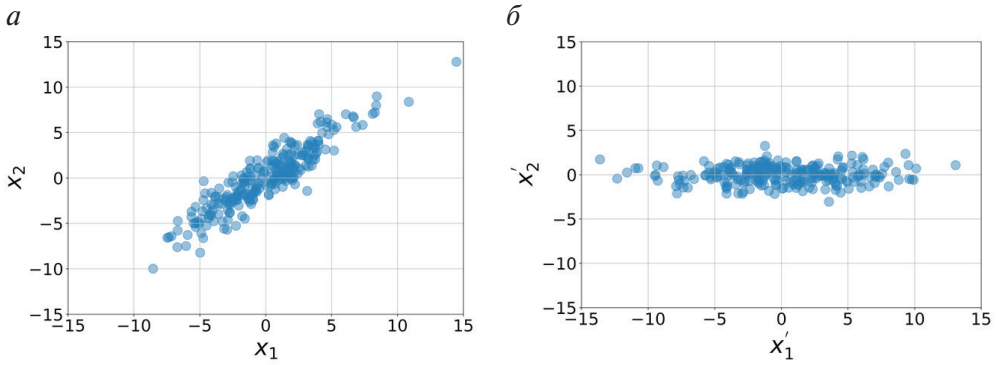


Рис. 5.1. Уменьшение размерности двумерных данных

Сгенерировать данные, распределенные подобным образом, можно с использованием функции `create_elipsoid_data`, где `C1` и `C2` — координаты центра, `S1` и `S2` — радиусы эллипса, `theta` — угол наклона, `N` — количество точек, `random_state` — фиксированный сид случайных чисел (для повторяемости генерируемых данных):

```
    def create_elipsoid_
data(C1 = 0,C2 = 0 ,S1 = 5,S2 =1, theta =45, N = 250, random_
state = 42):
        if random_state: np.random.seed(random_state)

        theta = np.pi*theta/180
        Centers = np.array([C1,C2])
        Sigmas = np.array([S1,S2])

        R = np.array([[np.cos(theta), - np.sin(theta)],
                      [np.sin(theta), np.cos(theta)]])

        return (R @ np.diag(Sigmas) @ np.random.randn(2, N)+np.
diag(Centers)@ np.ones((2, N))).T
```

В нашем примере одна ось, в которой расположена основная часть фигуры, будет главной компонентой. Совокупность главных компонент образует так называемое *собственное подпространство*. Вторая ось останется *шумовым подпространством*. Как видно из примера на рис. 5.1, *б*, размерность фигуры в каждой из обозначенных осей будет соответствовать важности этой оси.

Другими словами, можно сказать, что разброс значений в каждой оси будет соответствовать ее важности. Такой разброс значений по каждой оси будет называться *собственными значениями*. Сортируя

собственные значения по убыванию, мы можем определить те из них, которые следует оставить, и те, которые следует убрать.

В случае данных большей размерности происходят схожие преобразования: находятся оси многомерных эллипсов, в которых данные изменяются в большей степени. При этом уменьшение размерности можно реализовать за счет выбора не всех главных компонент, а только тех, которые имеют наибольшее собственное значение.

Метод главных компонент

Формализуем алгоритм вычисления главных компонент. Для этого нужно ввести дополнительные понятия и определения. *Матрица ковариации* — это матрица, составленная из попарных ковариаций столбцов этой матрицы. *Ковариация* — это мера совместной изменчивости двух случайных величин.

Пусть есть матрица данных $\boldsymbol{X}$, где p — количество признаков, n — количество точек. В общем случае для определения матрицы ковариации необходимо воспользоваться следующей формулой:

$$\Sigma = \operatorname{cov}(\boldsymbol{X}, \boldsymbol{X}) = \mathrm{E}\left[\left(\boldsymbol{X} - \mathrm{E}[\boldsymbol{X}]\right)^T \left(\boldsymbol{X} - \mathrm{E}[\boldsymbol{X}]\right)\right],$$

где E – оператор математического ожидания.

Однако эту формулу можно упростить, предварительно выполнив централизацию матрицы $\boldsymbol{X}$, т. е. вычесть среднее для каждого признака. Тогда вычисление матрицы ковариации просто сведется к матричному умножению транспонированной центрированной матрицы на саму себя. При этом при анализе реальных разнородных данных признаки не просто центрируют, а стандартизируют, добиваясь одинаковой дисперсии признаков. Это оправданно, когда сопоставляются признаки, измеряемые в разных единицах (например, возраст в годах и зарплата в рублях).

В результате получится матрица $\Sigma \in \mathbb{R}^{p \times p}$, т. е. квадратная матрица. Отметим, что собственные вектора и собственные значения для произвольной квадратной матрицы A удовлетворяют следующему уравнению:

$$A\vec{v}_i = \lambda_i \vec{v}_i,$$

где $\vec{v}_i$ – это собственный вектор; λ_i – соответствующее собственное значение. По сути, выражение выше представляет собой решение систе-

мы линейных уравнений с параметром λ_i. Значения данного параметра можно найти из следующего выражения:

$$det\left(A-\lambda_I\right)=0,$$

где *det* — операция поиска определителя матрицы, а λ_I — диагональная матрица с собственными значениями по главной диагонали и нулями в остальных позициях. При раскрытии операции детерминанта по определению данное уравнение может быть сведено к поиску корней полинома.

Цель метода главных компонент — найти и отбросить шумовое подпространство. Классический метод главных компонент состоит из следующих операций:

1) вычисление ковариационной матрицы для набора данных, то есть матрицы дисперсий;
2) вычисление (поиск) собственных векторов и их собственных значений по ковариационной матрице;
3) сортировка собственных значений по убыванию;
4) выделение собственного подпространства;
5) преобразование данных – построение проекции исходного массива на полученные собственные вектора.

Все эти операции реализованы в классе `PCA`, который представлен в прил. 6. Размерность собственного пространства задается входным параметром `n_components`. При этом операции с 1 по 4 реализованы методом `.fit`. Последняя операция реализована методом `.transform`. Кроме того, реализована операция `.inverse_transform`, необходимая для восстановления исходного набора данных. Собственные значения для матрицы данных хранятся в атрибуте `values`, а собственные вектора – в атрибуте `components`. Дополнительно в классе `PCA` реализован метод `.plot_eigvalues` для визуализации собственных значений.

По сути, пространство главных компонент является линейной комбинацией всех исходных признаков, при этом веса в этой линейной комбинации определяются элементами собственных векторов. Соответствующие собственные значения показывают значимость главной компоненты. Анализируя распределение собственных значений, можно делать выводы о том, сколько главных компонент вносят основной вклад в дисперсию исходных данных.

Важно отметить, что на практике такое восстановление может быть неточным, так как, сокращая разность данных, можно удалить оттуда

и часть полезной информации. Для этого в классе `PCA` реализован метод `.score`, который использует метрику *коэффициент детерминации*, который мы рассмотрели в главе 3.

Наконец, обсудим применение метода главных компонент для категориальных признаков. Существует разные взгляды на то, возможно ли это и насколько это корректно. Некоторые работы показывают, что достаточно применить One-Hot-кодирование к категориальным признакам, а затем применить метод главных компонент «как есть». С другой стороны, авторы метода *факторного анализа смешанных данных* (Factorial Analysis of Mixed Data, FAMD) считают, что их подход обобщает метод главных компонент для случая категориальных признаков. Сначала необходимо применить One-Hot-кодирование, затем нормировать каждый столбец закодированных категориальных признаков, чтобы соотнести между собой категориальные и числовые признаки. Для этого нужно поделить каждый столбец на корень из вероятности признака после One-Hot-кодирования (количество единиц в столбце, деленное на количество строк). Деление на квадратный корень из вероятностей можно интерпретировать так: мы придаем больший вес редким признакам, потому что информация «этот признак редко встречается» более значима, чем «этот признак распространен». А затем метод главных компонент применяется как обычно.

Набор данных MNIST

Возможно, поиск осей в двумерном эллипсе не выглядит как что-то впечатляющее. Посмотрим возможности метода главных компонент на более сложной задаче, наборе данных MNIST — это набор рукописных чисел от 0 до 9. Общее число изображений в наборе данных — 70000. Примеры изображений, которые содержатся в этом наборе данных, представлены на рис. 5.2.

Изображения в наборе MNIST – одноканальные, размером 28 × 28 пикселей. Воспользуемся простым подходом и будем рассматривать эти изображения как вектор признаков из 784 параметров. Используем функцию `fetch_openml` для загрузки данных с сайта OpenML, имя набора данных `'mnist_784'`. Помимо набора данных MNIST, сайт

OpenML содержит большое число других наборов данных, которые можно использовать для практики алгоритмов машинного обучения.

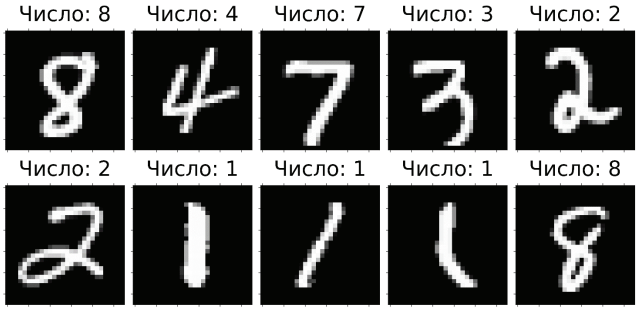


Рис. 5.2. Примеры изображений в наборе данных MNIST

В коде ниже укажем: необходимо, чтобы функция вернула только массив данных и вектор меток (`return_X_y=True`), а не полные данные в формате словаря. При этом добавим, что данные должны быть в формате массивов numpy array, а не датафреймов Pandas (`as_frame = False`).

```
    from sklearn.datasets import fetch_openml
    X, y  = fetch_openml('mnist_784', version=1, return_X_
y=True, as_frame = False)
```

Данные в наборе MNIST хранятся в формате 8 бит (т. е. интенсивность пикселя закодирована числом от 0 до 255). Нормализуем данные, для этого достаточно поделить все значения на 255. При этом данные будут изменяться в диапазоне от 0 до 1, где 0 – это черные пиксели, а 1 — белые. Промежуточные значения – 254 оттенка от черного к белому.

Воспользуемся подготовленным нами классом `PCA` и применим метод главных компонент к набору данных MNIST, указав `n_components` равным 100. На рисунке 5.3 представлена визуализация собственных значений для проведенного преобразования. Видно, что после 100-го собственного значения график асимптотически стремится к 0.

На рисунке 5.4 представлена визуализация скаттерограмм для первых четырех главных компонент. Каждая цифра представлена точкой соответствующего цвета. Несмотря на ограниченную размерность, ряд цифр сгруппированы «по группам». Так, цифры 0 и 1 стоят обособленно от остальных в координатах первой главной компоненты, вторая главная компонента позволяет отделить цифру 2 от остальных, а чет-

вертая главная компонента позволяет отсечь часть представлений цифры 6. При этом перемешанными оказались цифры 3 и 5, а также цифры 4, 7 и 9, что вполне объяснимо из-за схожести в их написании. Тем не менее это достаточно хороший результат для такой простой модели.

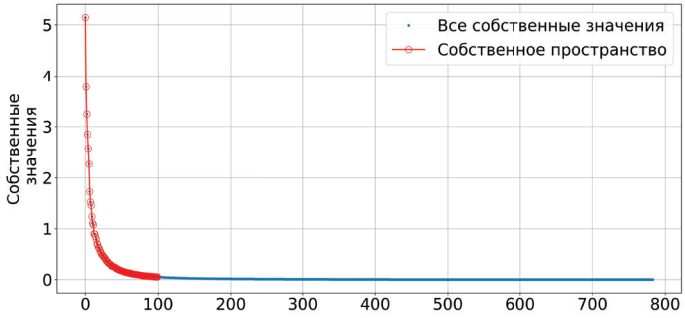


Рис. 5.3. Визуализация собственных значений для набора данных MNIST

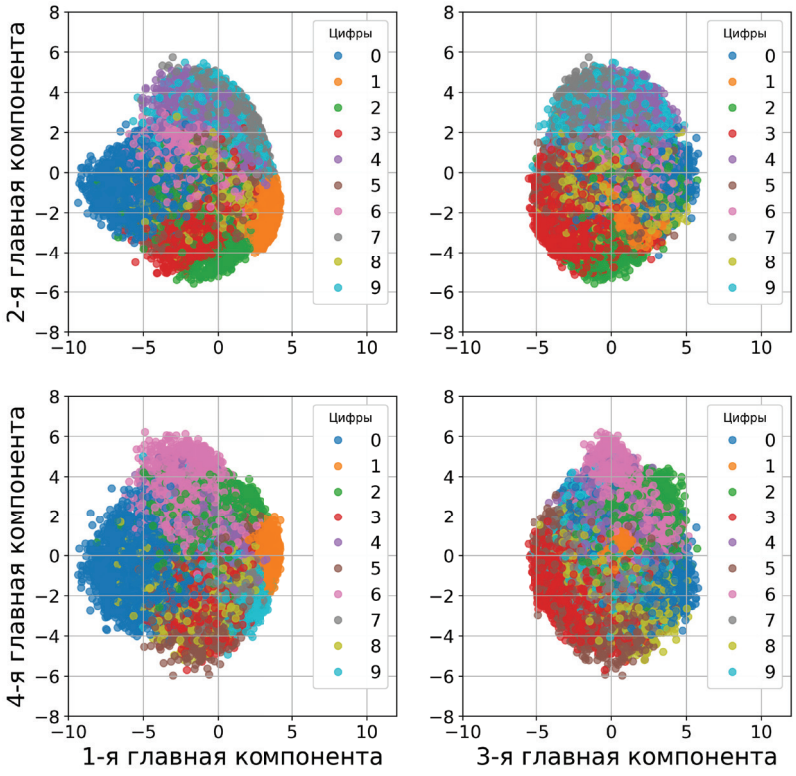


Рис. 5.4. Визуализация пространства четырех главных компонент для набора данных MNIST

Попробуем интерпретировать полученные результаты. Для этого визуализируем собственные вектора. Сгруппируем их таким образом, чтобы можно было составить изображение, аналогичное исходным. На рисунке 5.5 красным цветом обозначены положительные значения, а синим – отрицательные. С учетом того, что главные компоненты – это линейная комбинация исходных признаков, «работу» первой главной компоненты можно интерпретировать следующим образом: берется центральная область изображения с положительными весами и окружающая ее область с отрицательными.

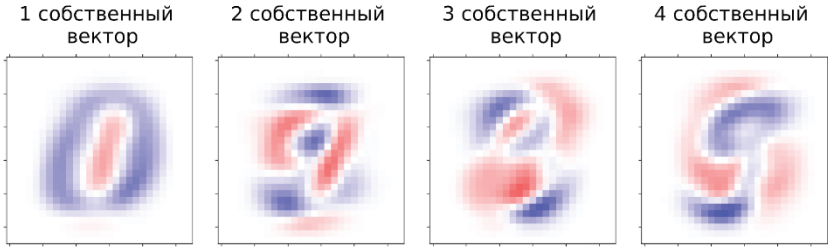


Рис. 5.5. Визуализация четырех собственных векторов для набора данных MNIST

Таким образом, если изображение похоже на цифру 1, то для этой главной компоненты получится положительное число (светлые пиксели исходного изображения будут помножены на положительные веса). Если изображение похоже на цифру 0, то для этой главной компоненты получится отрицательное число (светлые пиксели исходного изображения будут помножены на отрицательные веса). Для других цифр результат зависит от того, сколько светлых пикселей попадут в зоны положительных и отрицательных весов.

Аналогичным образом можно попытаться интерпретировать другие главные компоненты. Однако, к сожалению, такой красивой и однозначной трактовки, как с первой главной компонентой и цифрами 0 и 1, не получится.

Наконец, посмотрим, как будут выглядеть изображения, если их реконструировать из 100 главных компонент. На рисунке 5.6 представлены реконструированные изображения и соответствующие им оригиналы. Видно, что в целом восстановленные изображения соответствуют оригиналам, хотя имеются шумы и артефакты.

Рис. 5.6. Реконструкция изображения с использованием главных компонент

Можно количественно оценить коэффициент сжатия на уровне размерностей. Исходная матрица $n \times p$ восстанавливается из пространства главных компонент размером $n \times p'$ и совокупности собственных векторов общим размером $p \times p'$. Тогда количественно коэффициент сжатия K можно оценить по следующей формуле:

$$K = \frac{p' \cdot (p + n)}{p \cdot n}.$$

Для наших преобразований получим следующие значения:

$$K = \frac{100 \cdot (784 + 70000)}{784 \cdot 70000} \approx 0.13 = 13\%.$$

Однако стоит отметить ограниченность метода главных компонент в общем случае. Так, найденные собственные вектора будут относительно хорошо работать на новых данных, которые «похожи» на те, что были в обучающем наборе. Но если цифры будут наклонены под другим углом, то результаты восстановления могут быть неожиданными. Кроме того, если изображения более сложные, например различные изображения котов, то собственные вектора могут носить случайных характер.

Практические задания

1. Сгенерируйте данные в виде эллипса с разными значениями радиусов и углов наклона. Примените метод главных компонент. Визуализируйте пространство главных компонент, оцените собственные значения и собственные вектора.

2. Загрузите данные MNIST. Поэкспериментируйте с количеством компонент при применении метода главных компонент. Оцените качество восстановления при разных значениях размерности собственного пространства. Визуализируйте разные пространства главных компонент.
3. Примените метод главных компонент для набора данных Cars:
   - выполните визуализацию пространства главных компонент и оцените их связь с исходными признаками;
   - примените пространство главных компонент в качестве входных данных для алгоритмов;
   - сравните результаты модели при использовании только числовых признаков и при добавлении категориальных признаков с использованием One-Hot-кодирования.

4*. Сравните работу реализованных алгоритмов с функцией библиотеки scikit-learn – методом главных компонент `sklearn.decomposition.PCA`.

Контрольные вопросы

1. Как связаны главные компоненты с исходными данными?
2. Сделайте грубую оценку сжатия данных, если исходная матрица имела размерность (4250, 7), а при восстановлении используются три главные компоненты.
3. Сгенерируйте данные в виде эллипса с центром в точке (1.5, –2.5), радиусами (3, 2.5), углом 65 и количеством точек 1100. Оцените собственные вектора, собственные значения, максимальные и минимальные значения в пространстве главных компонент.
4. Для набора данных Cars проанализируйте веса главных компонент при использовании числовых признаков. Какой из параметров вносит наименьший вклад в первую главную компоненту?

Глава 6.
Кластеризация

Задачу кластеризации можно ассоциативно описать с помощью житейской ситуации: если вы стоите рядом с другими людьми на вечеринке, вероятно, у вас есть что-то общее. Эта идея используется в алгоритме кластеризации *k*-средних для разделения точек данных на группы.

Интересный исторический факт. Американский физик С. Ллойд, выпускник знаменитой инновационной фабрики Bell Labs и Манхэттенского проекта, в котором была изобретена атомная бомба, впервые предложил кластеризацию *k*-средних в 1957 г. для распределения информации в цифровых сигналах, но не публиковал алгоритм до 1982 г. Тем временем американский статистик Э. Форги описал аналогичный метод в 1965 г., что привело к его альтернативному названию – *алгоритм Ллойда – Форги*.

В ряде случаев при анализе данных оказывается, что о них ничего не известно, однако требуется понять, насколько они однородны или, например, можно ли их разделить на группы (*кластеры*). Другими словами, нужно найти закономерности в данных как таковых без привязки к тому, какие результаты для них мы хотим получить. Задача разделения на кластеры не требует наличия учителя.

Метод кластеризации *k*-средних, как и многие другие методы кластеризации, опирается на «близость» точек данных друг к другу. Для этого необходимо научиться измерять расстояние между точками в произвольном пространстве.

Метрики расстояния

Для простоты описания используем двумерный случай, однако все описанные метрики будут работать и в случаях пространств большей размерности.

Евклидово расстояние

Первая метрика расстояния знакома нам по школьной программе, хотя не все смотрят на теорему Пифагора как на способ определения расстояния между точками.

Допустим, в двумерном пространстве $a, b \in \mathbb{R}^2$ есть две точки, координаты которых нам известны. Достроим прямоугольный треугольник (рис. 6.1), тогда искомое расстояние будет гипотенузой в этом прямоугольном треугольнике.

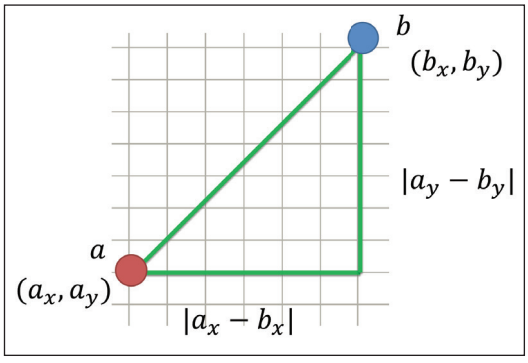


Рис. 6.1. К определению евклидова расстояния

Согласно теореме Пифагора, квадрат гипотенузы равен сумме квадратов катетов. Катеты находятся как модуль разности соответствующих координат. Тогда искомое расстояние будет определено как корень квадратный из суммы квадратов разности координат. В общем случае $a, b \in \mathbb{R}^m$ и расстояние определится по формуле

$$L^2 = d_2(a,b) = \left\{ \sum_{i=1}^{m} |a_i - b_i|^2 \right\}^{\frac{1}{2}}.$$

Эта метрика расстояния носит название *евклидово расстояние*.

Манхэттенское расстояние

Теперь представим ситуацию, что мы едем на машине по городу из пункта *A* в пункт *B* (рис. 6.2). В городской среде мы не можем двигаться напрямую по гипотенузе, как птицы, а вынуждены использовать дороги с их поворотами, перекрестками и т. д. Тогда расстояние между точками *A* и *B* будет определяться просто как сумма длин катетов.

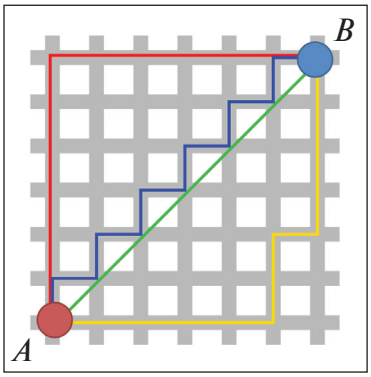


Рис. 6.2. К определению манхэттенского расстояния [18]

В общем случае $a,b \in \mathbb{R}^m$ и расстояние определится по формуле

$$L^1 = d_1(a,b) = \sum_{i=1}^{m} |a_i - b_i|.$$

Эта метрика носит название *манхэттенское расстояние*. Из-за аналогии с движением по городу часто упоминаются названия City Block и Taxicab.

Расстояние Чебышева

Ассоциативным примером для следующей метрики расстояния служат шахматная доска и фигура короля. Из правил шахмат мы помним, что король может двигаться на одну клетку в любом направлении: по горизонтали, вертикали и диагонали. Исходя из этого, можно оценить каждую клетку на доске с точки зрения того, сколько ходов понадобится королю, чтобы до нее дойти (рис. 6.3).

В общем случае $a,b \in \mathbb{R}^m$ и расстояние определится по формуле

$$L^\infty = d_\infty(a,b) = \max\left(|a_i - b_i|\right).$$

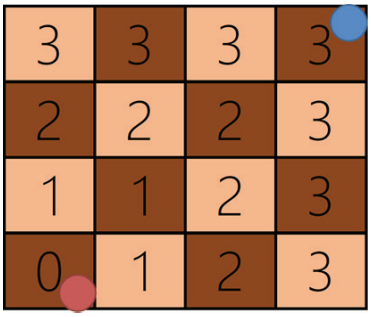


Рис. 6.3. К определению расстояния Чебышева

Эта метрика расстояния носит название *расстояние Чебышева*. Из-за аналогии с шахматной доской часто упоминаются название ChessBoard.

Расстояние Минковского

Нетрудно заметить общую составляющую в метриках расстояния. Используется модуль разности координат, полученные модули разности определенным образом суммируются: либо возводятся в квадрат (вторая степень), либо берутся просто модули (первая степень). Естественным образом метрика расстояния обобщается на произвольную степень $p \geq 1$. В общем случае $a, b \epsilon \mathbb{R}^m$ и расстояние определится по формуле

$$L^p = d_p(a,b) = \left\{ \sum_{i=1}^{m} |a_i - b_i|^p \right\}^{\frac{1}{p}}.$$

Эта метрика расстояния носит название *расстояние Минковского*.

На рисунке 6.4 представлены эквидистантные фигуры для различных метрик расстояний. Все точки, лежащие на прямых одного цвета, находятся на расстоянии 1 от центра координат с точки зрения соответствующей метрики расстояния.

Запишем функцию для определения расстояния:

```
def distance(X1, X2, metric = 'euclidean', p = 2):
    if metric == 'euclidean':
        dist = np.sqrt(np.sum(np.square(X1 - X2).T,axis=0))
    if metric == 'cityblock':
        dist = np.sum(np.abs(X1 - X2).T,axis=0)
    if metric == 'Chebyshev':
```

```
            dist = np.max(np.abs(X1 - X2).T,axis=0)
        if metric == 'Minkowski':
            dist = np.power(np.sum(np.power(np.
abs(X1 - X2),p).T,axis=0),1/p)
        return dist
```

Функция работает как с векторами, так и с матрицами равной размерности.

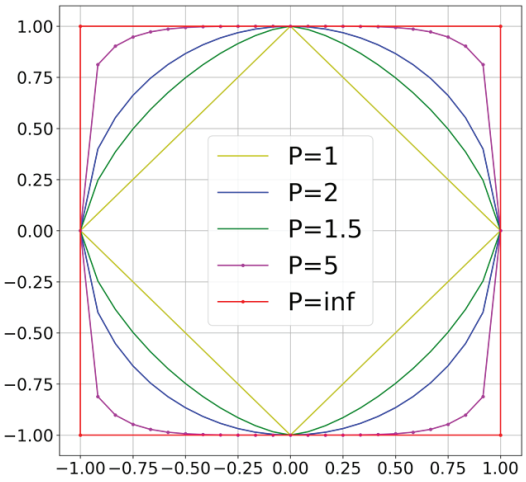


Рис. 6.4. Эквидистантные фигуры для разных метрик расстояния

Как правило, метрикой расстояния по умолчанию является евклидова метрика. Однако в разных задачах, особенно если анализируются категориальные переменные, могут хорошо проявлять себя и другие метрики. Выбор метрики расстояния тоже можно отнести к гиперпараметрам модели.

Кроме того, необходимо помнить о стандартизации или нормализации данных при использовании метрик расстояния. Иначе результаты вычисления метрик могут быть некорректны, если используются данные, измеряющиеся в разных диапазонах.

Алгоритм *k*-средних

Одним из самых простых методов кластеризации является метод *k*-средних. Суть данного метода сводится к тому, чтобы найти заданное

число кластеров (k) и их центры (так называемые *центроиды*) – такие, чтобы расстояние от центроидов до всех точек кластера было минимальным.

Алгоритм k-средних может быть описан следующим образом:

- выбирается k случайных точек – центроидов;
- рассчитывается вектор расстояния между каждой точкой набора данных и каждым центроидом;
- в каждый кластер записываются те точки, для которых оказалось, что до соответствующего центроида расстояние меньше, чем до других;
- новые значение центроидов рассчитываются как среднее значение по всем точкам кластера.

Рассмотрим этот метод на генерируемых данных из главы 4. Начнем с простой модели линейно разделимых данных.

Прежде чем проводить кластеризацию, необходимо проинициализировать кластеры. Для этого выберем случайные индексы среди доступных в наборе данных:

```
    def init_centroids(X, n_clusters):
        centroid_idxs = np.random.randint(0, X.shape[0],
size = n_clusters)
        return X[centroid_idxs,:]
```

Посмотрим, как это работает для двух кластеров. Проведем первую кластеризацию. Для этого возьмем каждый центроид и посчитаем расстояние от него до всех записей набора данных. Индексы значений для каждого кластера выберем как индексы минимальных расстояний до соответствующего центроида. Таким образом, нулевой кластер будет включать те точки набора данных, в которых расстояние до нулевого центроида меньше, чем до первого центроида.

```
    def predict(X, n_clusters, centroids, metric =
'euclidean', p = 2):
        distances = np.zeros((X.shape[0], n_clusters))
        for i,centr in enumerate(centroids):
            distances[:,i] = distance(centr,X, metric, p)
        cluster_label = np.argmin(distances,axis = 1)
        return cluster_label
```

Посмотрим, как распределились результаты кластеризации после случайной инициализации центроидов (рис. 6.5). Получилось доста-

точно интересно: оба центроида оказались относительно близко друг к другу. Тем не менее это нулевая итерация алгоритма.

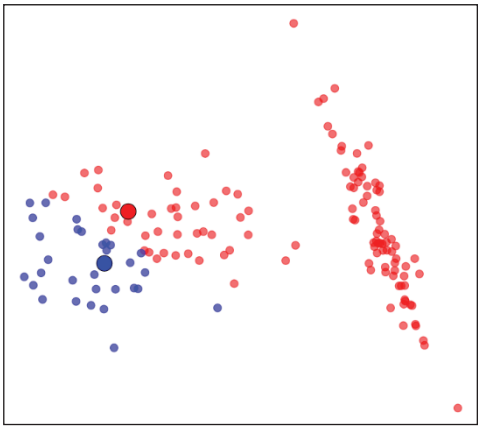


Рис. 6.5. Результаты кластеризации после инициализации центроидов

Теперь выберем новые центроиды для каждого кластера как среднее значение по кластеру. В визуализации будут видны старые центроиды и новый центроид с бóльшим радиусом (рис. 6.6). Видно, что центр для красного кластера значительно сместился.

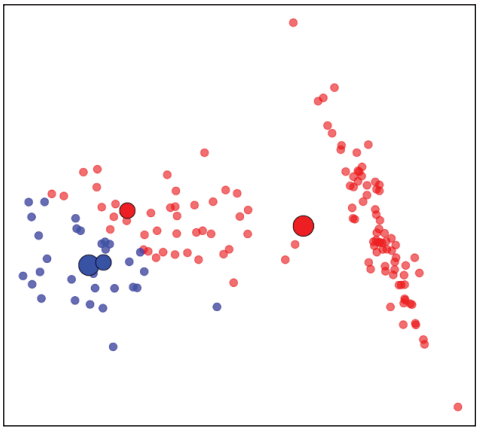


Рис. 6.6. Результаты кластеризации после изменения центроидов

Рассчитаем относительное расстояние между старыми и новыми центроидами. Если расстояние между обновленными центроидами будет сравнительно небольшим, то есть центроиды перестанут менять позицию, то будем считать, что кластеризация закончена.

```
    def delta_centroids(centroids,old_centroids,
metric = 'euclidean', p = 2):
        return (distance(centroids,old_centroids, metric, p)/
distance(old_centroids, np.mean(old_centroids), metric, p)).mean()
```

Для автоматизации процесса реализуем процедуру итерационной кластеризации. Укажем порог относительного изменения центроидов – `tol`. В конце процедуры выведем результирующий номер итерации и расстояние между последним изменением центроидов:

```
    def fit(X, n_clusters, centroids, max_
iter=10, tol=0.01, metric = 'euclidean', p = 2):

        dcentr = np.inf

        for i in range(max_iter):

         old_centroids = np.copy(centroids)
        cluster_label=predict(X, n_clusters, centroids, metric, p)

            for k in range(n_clusters):
                c_idxs = np.flatnonzero(cluster_label==k)
                centroids[k] = X[c_idxs].mean(axis = 0)

          dcentr = delta_centroids(centroids,old_centroids, met-
ric, p)

            if dcentr<=tol:
                break

        print('Мы остановились на итерации:', i,', относитель-
ное изменение центроидов: ',dcentr)

        return cluster_label
```

Проверим и визуализируем результаты. Буквально через пару итераций получим следующее распределение (рис. 6.7).

Теперь объединим все наши наработки в один класс `KMeans` (представлен в прил. 7). Центроиды на последней итерации хранятся в атрибуте `.centroids`. Для того чтобы получить предсказания номера кластера, необходимо воспользоваться методом `.fit_transform`. Кроме того, мы добавили в класс `KMeans` атрибут `.inertia` — сумму квадратов расстояния точек кластеров до соответствующих центроидов.

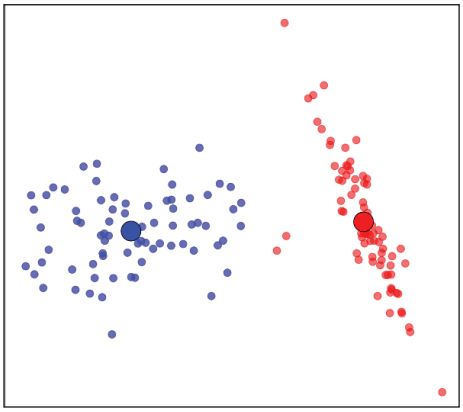


Рис. 6.7. Результаты кластеризации после нескольких итераций

С помощью этого параметра можно попытаться ответить на вопрос, какое число кластеров оптимально для конкретных данных. Для этого можно использовать так называемый *метод локтя* (Elbow Method): проверяются различные количества кластеров и запоминаются конечные значения инерции для каждого числа кластеров, затем визуализируется зависимость (число кластеров, инерция). Как правило, получаются зависимости, подобные рис. 6.8.

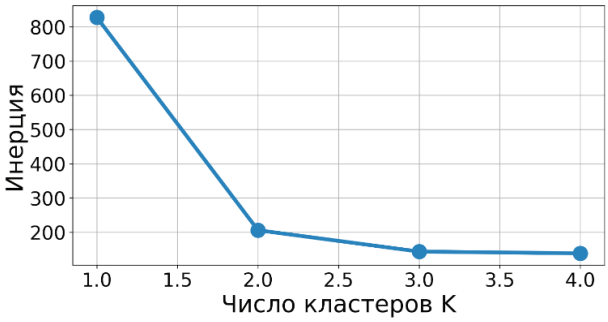


Рис. 6.8. Визуализация метода локтя

Затем ищется такое число кластеров, которое напоминает перегиб согнутого локтя (в нашем случае это число $k = 2$). Большее количество кластеров неуместно: мы начинаем искусственно дробить большие кластеры на малые, нарушая целостную структуру данных (рис. 6.9).

Метод k-средних можно применять не только к двумерным данным, но и к данным большей размерности. При этом рекомендуется

пользоваться методами уменьшения размерности для возможности визуализации результата кластеризации.

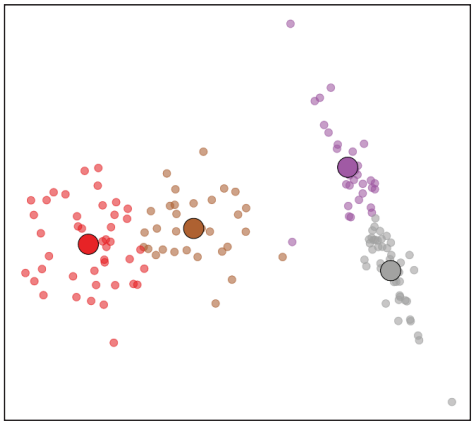


Рис. 6.9. Кластеризация *k*-средних со слишком большим числом кластеров

В общем случае кластеризация применяется в контексте того, что истинные метки кластеров нам неизвестны. Именно поэтому эту задачу машинного обучения относят к задачам обучения без учителя. Однако иногда возникают тренировочные задачи, в которых метки известны или помимо числовых данных есть набор категориальных признаков, и требуется выяснить, связаны ли найденные кластеры с каким-то категориальным признаком.

Для этого можно воспользоваться матрицей, схожей с матрицей ошибок из главы 4 и соответствующими метриками. Только нужно помнить особенность: номера кластеров генерируются случайно и могут совершенно противоречить известным меткам классов. Проверить это можно с помощью перекрестного табулирования с использованием метода `crosstab` библиотеки `Pandas`: здесь `y` — истинные метки классов, а `c_labels` — полученные предсказания кластеров:

```
pd.crosstab(y,c_labels, rownames=['Метки'], colnames = ['Предсказания'])
```

И уже на основе таблицы кросс-табулирования можно оценивать метрики, схожие с метриками классификации, введя следующие обозначения:

- TP — элементы принадлежат одному кластеру и одному классу;

- FP — элементы принадлежат одному кластеру, но разным классам;
- FN — элементы принадлежат разным кластерам, но одному классу;
- TN — элементы принадлежат разным кластерам и разным классам;
- индекс Rand (аналог доли правильных ответов):

$$\text{Rand} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}};$$

- индекс Жаккара:

$$\text{Jaccard} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}};$$

- индекс Фоулкса — Мэллова:

$$\text{FM} = \sqrt{\frac{\text{TP}}{\text{TP} + \text{FP}} \frac{\text{TP}}{\text{TP} + \text{FN}}}.$$

Практические задания

1. Сгенерируйте линейно разделимые данные с другими параметрами и проверьте, как работает алгоритм кластеризации *k*-средних на этих данных. Оцените оптимальное число кластеров по методу локтя.
2. Сгенерируйте данные, распределенные как знак инь-ян или концентрические круги, и проверьте, как работает алгоритм кластеризации *k*-средних на этих данных. Оцените оптимальное число кластеров по методу локтя.
3. Загрузите данные MNIST. Уменьшите размерность данных с использованием метода главных компонент. Примените кластеризацию *k*-средних. Оцените оптимальное число кластеров по методу локтя и связь кластеров с цифрами на изображениях.
4. Выполните кластеризацию для набора данных Cars:
   - выполните кластеризацию для числовых признаков: используйте все числовые признаки, выполнив визуализацию в разных двумерных проекциях;

- оцените оптимальное число кластеров по методу локтя;
- оцените связь кластеров с категориальными признаками;
- сравните результаты модели при использовании данных с применением метода главных компонент.

5*. Сравните работу реализованных алгоритмов с функциями библиотеки scikit-learn — кластеризацией k-средних `sklearn.cluster.KMeans`.

Контрольные вопросы

1. Оцените евклидово расстояние между векторами $\boldsymbol{x}_1$ {2, 5, 3 ,7} и $\boldsymbol{x}_2$ {2, 7, 1, 5}.
2. Оцените расстояние Чебышева между векторами $\boldsymbol{x}_1$ {0, 10, 4, 9} и $\boldsymbol{x}_2$ {3, 7, 0, 2}.
3. Есть три центроида $\boldsymbol{c}_1$ {1, 0, 0}, $\boldsymbol{c}_2$ {0, 1, 1}, $\boldsymbol{c}_3${1, 0, 1} и точка $\boldsymbol{x}$ с координатами {2, 0, 2}. К какому кластеру следует отнести эту точку при использовании евклидовой метрики расстояния?
4. Как называется метод определения оптимального числа k (кластеров) с использованием анализа инерции?

Заключение

Мы прошли достаточно длинный путь в освоении базовых алгоритмов машинного обучения на языке Python.

В 1-й главе мы обсудили ключевые понятия и термины машинного обучения. Рассмотрели основные типы данных, которые обычно анализируются с помощью методов машинного обучения, и конкретные задачи, которые возможно решать с использованием методов машинного обучения. В основном данное учебно-методическое пособие посвящено работе с табличными данными. Кроме того, мы вспомнили некоторые понятия линейной алгебры и математического анализа, необходимые для комфортной работы с алгоритмами машинного обучения. Центральное понятие – производная, благодаря которой обучаются даже самые сложные глубокие нейронные сети.

Во 2-й главе мы рассмотрели возможности библиотеки Pandas для анализа данных и освоили ключевые методы, которые пригодятся в начале работы с данными. Упоминалась также библиотека Seaborn для визуализации данных. (На самом деле возможности этой библиотеки настолько широки, что заслуживают отдельного учебного пособия.) Помимо этого мы обсудили ключевые идеи, которые необходимо использовать для предварительной обработки данных и инженерии признаков.

В 3-й главе мы пошагово реализовали первую модель машинного обучения – линейную регрессию. Реализация была выполнена в двух вариантах: в виде отдельных функций, а также в более высокоуровневом виде, с использованием классов языка Python. Мы обсудили, как с помощью модели, которая может строить только прямые линии, возможно находить нелинейные зависимости и как вводить дополнительные ограничения на модель с использованием регуляризации, которая может не только повысить обобщающую способность модели, но и приводить к отбору значимых признаков.

В 4-й главе мы трансформировали модель линейной регрессии для решения задачи классификации в логистическую регрессию, что позволило унаследовать возможности и сильные стороны модели линейной регрессии для решения новой задачи. Отдельного упоминания стоят метрики классификации, которые основаны на матрице ошибок.

В 5-й главе мы обсудили классический метод уменьшения размерности – метод главных компонент. Этот метод использует дисперсию как критерий важности признаков и позволяет строить новое признаковое пространство как линейную комбинацию исходных данных. Мы применили этот метод к однотипным изображениям – рукописным цифрам – и даже смогли визуализировать полученное уменьшенное пространство признаков.

В 6-й главе мы рассмотрели, как различные метрики расстояния можно использовать в задачах кластеризации. Кластеризация – это своеобразный аналог задачи классификации с определенными особенностями. Пошагово мы реализовали базовый алгоритм кластеризации *k*-средних.

Что делать дальше, после того как все практические задания выполнены, а на контрольные вопросы есть ответы?

Во-первых, рекомендуется более детально ознакомиться с открытой библиотекой машинного обучения scikit-learn. В этом пособии мы уже упоминали ряд функций данной библиотеки для генерации данных, загрузки данных, а также аналоги базовых алгоритмов машинного обучения, реализованных по ходу изложения материала. Но на этом возможности библиотеки scikit-learn не заканчиваются, поскольку включают большое число моделей машинного обучения, которые используют другие базовые идеи [19].

Далее стоит ознакомиться с моделями, которые лучше всего работают при анализе конкретных типов данных. Так, для анализа табличных данных лучше всего подходят модели, основанные на деревьях решений [20]. К таким моделям относят так называемые *методы бустинга*, которые, подобно супергероям из комиксов, объединяющимся для победы над более сильными злодеями, объединяют простые модели таким образом, чтобы лучше решить поставленную задачу. В настоящее время в открытом доступе представлены разные модели [21], в том числе модель CatBoost от компании «Яндекс» [22].

Для обработки изображений в первую очередь стоит ознакомиться с возможностями *сверточных нейронных сетей* [2–4]. Подобные моде-

ли решают задачи не только анализа заданных изображений, но и синтеза новых [23].

Для обработки естественного языка в настоящее время активно применяются *модели-трансформеры*, основанные на *механизме внимания* (Attention), предложенного командой из Google [10; 11]. По своей сути это все те же матричные умножения, реализованные с несколько более сложной архитектурой, чем рассмотренные нами линейные модели. Кстати, в основе нашумевшей нейронной сети ChatGPT тоже лежит механизм внимания.

В целом для успешной работы в сфере науки о данных (Data Science) нужно быть открытым к новой информации и постоянному обучению. Рекомендуется хотя бы раз в неделю просматривать новые статьи и блоги по тематике машинного обучения и искусственного интеллекта. Это направление очень открыто для обмена идеями и их распространения: постоянно публикуются интересные объяснения алгоритмов, визуализации результатов и другие кейсы из практики.

Список библиографических ссылок

1. Lenna [Электронный ресурс] // Википедия : [сайт]. — URL: https://en.wikipedia.org/w/index.php?title=Lenna&oldid=1118923916 (дата обращения: 31.10.2022).
2. Going deeper with convolutions / C. Szegedy [et al.] // Proceedings of the IEEE conference on computer vision and pattern recognition. 2015. P. 1–9.
3. Rethinking the inception architecture for computer vision / C. Szegedy [et al.] // Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. P. 2818–2826.
4. Deep residual learning for image recognition / K. He [et al.] // Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. P. 770–778.
5. Биомедицинские сигналы и изображения в цифровом здравоохранении: хранение, обработка и анализ : учеб. пособие / В. С. Кубланов [и др.]. Екатеринбург : Изд-во Урал. ун-та, 2020. 240 с.
6. Кубланов В. С., Борисов В. И., Долганов А. Ю. Анализ биомедицинских сигналов в среде MATLAB : учеб. пособие. Екатеринбург : Изд-во Урал. ун-та, 2016. 120 с.
7. Hochreiter S., Schmidhuber J. Long short-term memory // Neural Computation. 1997. Vol. 9. № 8. P. 1735–1780.
8. Pennington J., Socher R., Manning C. D. Glove: Global vectors for word representation // Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014. P. 1532–1543.
9. Efficient estimation of word representations in vector space [Электронный ресурс] / T. Mikolov [et al.] // arXiv : [сайт]. URL: https://arxiv.org/abs/1301.3781 (дата обращения: 31.10.2022).
10. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding [Электронный ресурс] / J. Devlin [et al.] // arXiv :

[сайт]. URL: https://arxiv.org/abs/1810.04805 (дата обращения: 31.10.2022).
11. Attention is all you need / A. Vaswani [et al.] // Advances in neural information processing systems. 2017. Vol. 30.
12. Pandas — Python Data Analysis Library [Электронный ресурс]. URL: https://pandas.pydata.org/ (дата обращения: 30.11.2021).
13. Matplotlib — Visualization with Python [Электронный ресурс]. URL: https://matplotlib.org/ (дата обращения: 08.08.2022).
14. Hunter J. D. Matplotlib: A 2D Graphics Environment // Computing in Science & Engineering. 2007. Vol. 9. № 3. P. 90–95.
15. Waskom M. Seaborn: statistical data visualization // Journal of Open Source Software. 2021. Vol. 6. № 60. P. 3021.
16. An introduction to seaborn – seaborn 0.11.2 documentation [Электронный ресурс]. URL: https://seaborn.pydata.org/introduction.html (дата обращения: 08.08.2022).
17. Example gallery — seaborn 0.11.2 documentation [Электронный ресурс]. URL: https://seaborn.pydata.org/examples/index.html (дата обращения: 08.08.2022).
18. Taxicab geometry [Электронный ресурс] // Википедия : [сайт]. — URL: https://en.wikipedia.org/w/index.php?title=Taxicab_geometry&oldid=1116503626 (дата обращения: 31.10.2022).
19. Scikit-learn: Machine learning in Python / F. Pedregosa [et al.] // Journal of machine learning research. 2011. Vol. 12. № 85. P. 2825–2830.
20. Grinsztajn L., Oyallon E., Varoquaux G. Why do tree-based models still outperform deep learning on tabular data? [Электронный ресурс] // arXiv : [сайт]. URL: https://arxiv.org/abs/2207.08815 (дата обращения: 30.11.2021).
21. XGBoost Documentation – xgboost 1.5.1 documentation [Электронный ресурс]. URL: https://xgboost.readthedocs.io/en/stable/ (дата обращения: 30.11.2021).
22. CatBoost – state-of-the-art open-source gradient boosting library with categorical features support [Электронный ресурс]. URL: https://catboost.ai (дата обращения: 30.11.2021).
23. Karras T., Laine S., Aila T. A Style-Based Generator Architecture for Generative Adversarial Networks [Электронный ресурс] // arXiv : [сайт]. URL: https://arxiv.org/abs/1812.04948 (дата обращения: 30.11.2021).

ПРИЛОЖЕНИЯ

1. Класс линейной регрессии

```
import numpy as np
import matplotlib.pyplot as plt

class LinearRegression():
    def __init__(self,
                 learning_rate = 0.5,
                 epochs = 100,
                 weights = None,
                 bias    = None,
                 batch_size   = 1000,
                 n_batches    = None,
                 random_state = 42):
        self.lr      = learning_rate
        self.epochs  = epochs
        self.weights = weights
        self.bias    = bias
        self.seed    = random_state
        self.batch_size = batch_size
        self.cost    = np.zeros(epochs)
        self.n_batches = n_batches
    #---------------------------------
    def forward(self, X):
        # умножаем признаки на веса
        return np.dot(X, self.weights)
    #---------------------------------
    def loss(self,yhat, y):
        # расчет функции потерь
        return np.square(yhat - y).sum()/y.size
    #---------------------------------
```

```
        def grad_step(self,yhat, y, X):
            # расчет градиента
            return 2*np.dot(X.T, (yhat - y)) / y.size
        #---------------------------------
        def update(self):
            # обновление весов
            return self.weights - self.lr*self.grad
        #---------------------------------
        def init(self, weights_size):
            # инициализируем веса
            np.random.seed(self.seed)
           return np.random.randn(weights_size)/np.sqrt(weights_size)
        #---------------------------------
        def predict(self, X):
            # делим предсказание модели
            yhat = self.forward(self.add_bias(X))
            return yhat.squeeze()
        #---------------------------------
        def score(self, X, y):
            # оценка по коэффициенту детерминации
            yhat = self.predict(X)
             return 1-np.sum(np.square(y-yhat))/np.sum(np.square(y-np.
mean(y)))
        #---------------------------------
        def fit(self, X, y):
            # обучение модели с учетом разбиения на батчи
            np.random.seed(self.seed)

            if self.weights is None: # если веса не заданы - задаем
                self.weights = self.init(X.shape[1])

            if self.bias is None: # если смещение не задано - задаем
                self.bias    = self.init(1)

              if self.weights.size == X.shape[1]: # если веса заданы,
но не добавлено смещение - объединяем
                self.weights = np.append(self.bias,self.weights)

            self.grad    = np.zeros(self.weights.shape)
            self.cost    = np.zeros(self.epochs)
```

```
            if self.batch_size is None:    # проверка на согласование
размерности батча и размерности данных
                self.batch_size = y.size

            if self.n_batches is None:
                self.n_batches = y.size//self.batch_size

            for i in range(self.epochs): #циклы обучения, как раньше
                loss = 0
                  for cnt,(x_batch, y_batch) in enumerate(self.load_
batch(X,y)):

                    yhat         = self.forward(x_batch)
                 self.grad    = self.grad_step(yhat,  y_batch, x_batch)
                    self.weights = self.update()
                    loss += self.loss(yhat,  y_batch)

                    if cnt>= self.n_batches:
                        break
                self.cost[i] = loss/self.n_batches

            self.bias = self.weights[0]
        #---------------------------------
        def load_batch(self,X,y):
            # загрузка батча
            idxs = np.arange(y.size)
            np.random.shuffle(idxs)

            for i_batch in range(0,y.size,self.batch_size):
                idx_batch = idxs[i_batch:i_batch+self.batch_size]
                x_batch   = np.take(X, idx_batch,axis=0)
                x_batch   = self.add_bias(x_batch) # тут мы всегда до-
бавляем смещение
                y_batch   = np.take(y, idx_batch)
                yield  x_batch, y_batch
        #---------------------------------
        def add_bias(self, X):
            # добавление смещения
            return np.column_stack((np.ones(X.shape[0]), X))
        #---------------------------------
```

```
    def plot_cost(self, figsize = (12,6), title = ''):
        # отрисовка сразу в методе
        plt.figure(figsize = figsize)
        plt.plot(self.cost)
        plt.grid()
        plt.xlabel('Эпоха', fontsize = 24)
        plt.ylabel('Функция Потерь', fontsize = 24)
        plt.title(title, fontsize = 24)
        plt.show()
    #---------------------------------
    def get_w_and_b(self):
        # «новый» метод - который возвращает веса модели и смещение
        return (self.weights[1:], self.bias)
```

2. Класс регуляризации Тихонова

```
class RidgeRegression(LinearRegression): #унаследуем от класса
LinearRegression
    def __init__(self,
                 learning_rate = 0.5,
                 l2_penalty = 0.001,
                 epochs = 100,
                 weights = None,
                 bias    = None,
                 batch_size   = 1000,
                 n_batches    = None,
                 random_state = 42):

        super().__init__(learning_rate = learning_rate,
                         epochs = epochs,
                         weights = weights,
                         bias    = bias,
                         batch_size   = batch_size,
                         n_batches    = n_batches,
                         random_state = random_state)
        self.l2_penalty = l2_penalty

    #---------------------------------
```

```
        def loss(self,yhat, y):
            # изменяем функцию потерь
                l2_term  =  (self.l2_penalty/2)*np.sum(np.square(self.
weights[1:]))
            return np.square(yhat - y).mean() + l2_term

        #---------------------------------
        def update(self):
            # изменяем правило обновления весов
            l2_term = self.l2_penalty*np.mean(self.weights[1:])
            return self.weights - self.lr*(self.grad + l2_term)
```

3. Класс регуляризации Лассо

```
    class LassoRegression(LinearRegression): # унаследуем  от класса
LinearRegression
        def __init__(self,
                     learning_rate = 0.5,
                     l1_penalty = 0.001,
                     epochs = 100,
                     weights = None,
                     bias    = None,
                     batch_size   = 1000,
                     n_batches    = None,
                     random_state = 42):

            super().__init__(learning_rate = learning_rate,
                             epochs = epochs,
                             weights = weights,
                             bias    = bias,
                             batch_size   = batch_size,
                             n_batches    = n_batches,
                             random_state = random_state)
            self.l1_penalty = l1_penalty

        #---------------------------------
        def loss(self,yhat, y):
            # изменяем функцию потерь
```

```
            l1_term = self.l1_penalty*np.sum(np.abs(self.weights[1:]))
            return np.square(yhat - y).mean() + l1_term

        #---------------------------------
        def update(self):
            # изменяем правило обновления весов
             return self.weights - self.lr*(self.grad + np.sign(self.
weights)*self.l1_penalty)
```

4. Класс эластичной регуляризации

```
    class ElasticRegression(LinearRegression): # унаследуем от класса
LinearRegression
        def __init__(self,
                     learning_rate = 0.5,
                     l1_penalty = 0.0,
                     l2_penalty = 0.0,
                     epochs = 100,
                     weights = None,
                     bias    = None,
                     batch_size   = 1000,
                     n_batches    = None,
                     random_state = 42):

            super().__init__(learning_rate = learning_rate,
                             epochs = epochs,
                             weights = weights,
                             bias    = bias,
                             batch_size   = batch_size,
                             n_batches    = n_batches,
                             random_state = random_state)
            self.l1_penalty = l1_penalty
            self.l2_penalty = l2_penalty

        #---------------------------------
        def loss(self,yhat, y):
            # изменяем функцию потерь
            l1_term = self.l1_penalty*np.sum(np.abs(self.weights[1:]))
```

```
        l2_term = (self.l2_penalty/2)*np.sum(np.square(self.weights[1:]))
            return np.square(yhat - y).mean() + l1_term + l2_term

        #---------------------------------
        def update(self):
            # изменяем правило обновления весов
            l2_term = self.l2_penalty*np.sum(self.weights[1:])
             return self.weights - self.lr*(self.grad +  np.sign(self.
weights)*self.l1_penalty + l2_term)
```

5. Класс классификации логистической регрессии

```
_EPS_ = 1e-6

class LogisticRegression(ElasticRegression):
    # унаследуем от класса ElasticRegression
    def __init__(self,
                 learning_rate = 0.5,
                 l1_penalty    = 0.0,
                 l2_penalty    = 0.0,
                 epochs        = 100,
                 weights       = None,
                 bias          = None,
                 threshold     = 0.5,
                 batch_size    = 1000,
                 n_batches     = None,
                 random_state  = 42):

        super().__init__(learning_rate = learning_rate,
                         epochs = epochs,
                         weights = weights,
                         bias    = bias,
                         batch_size   = batch_size,
                         n_batches    = n_batches,
                         random_state = random_state,
                         l1_penalty = l1_penalty,
                         l2_penalty = l2_penalty)
        self.learning_rate = learning_rate/2
```

```
            self.threshold = threshold
        #----------------------------------
        def loss(self,yhat, y):
            # изменяем функцию потерь - на бинарную кросс-энтропию
            l1_term = self.l1_penalty*np.sum(self.weights[1:])
         l2_term = (self.l2_penalty/2)*np.sum(np.square(self.weights[1:]))
            # добавки от регуляризации остаются прежде
           return -(y*np.log(yhat  + _EPS_)+(1 - y)*np.log(1 - yhat  +
_EPS_)).mean()\
                + l1_term+ l2_term
        #----------------------------------
        def sigmoid(self, z):
            # определение функции сигмоиды
            return 1 / (1 + np.exp(-z))
        #----------------------------------
        def forward(self, X):
            # умножаем признаки на веса и применяем к результату сигмоиду
            return self.sigmoid(np.dot(X, self.weights))
        #----------------------------------
        def to_class(self,logit):
            # классифицируем, сравнивая с порогом
            return (logit>=self.threshold)*1
        #----------------------------------
        def predict(self, X):
            # предсказание модели
            # в этот раз в два этапа
            yhat = self.forward(self.add_bias(X)) # 1 считаем модель
            return self.to_class(yhat) # 2 классифицируем по порогу

        #----------------------------------
        def predict_prob(self, X):
            # предсказание модели, но в «вероятностном виде»
            yhat = self.forward(self.add_bias(X))
            return yhat # для этого просто возвращаем модель
        #----------------------------------
        def score(self, X, y):
            # оценка модели
            yhat  = self.predict(X)
              return sum((yhat==y)*1)/y.size # по количеству совпав-
ших предсказаний - Accuracy
        #----------------------------------
```

```
        def plot_desicion_function(self,X,y,figsize = (12,6),
                                   marker = 'o',colors =("#FF0000", '#0000FF'),
                                   alpha=0.7, s = 150, poly = False, order = 2):
            # отрисовка функции принятия решений
            plt.figure(figsize = figsize) # создаем новое полотно
            cm_bright = ListedColormap(colors) # создаем цветовую карту
            # отрисовываем исходные данные
            plt.scatter(X[:, 0], X[:, 1],marker = marker, c=y, cmap=cm_
bright,s = s, alpha =alpha);

            h =  (X[:, 0].max() - X[:, 0].min())/50 # шаг сетки как 1/50
от разницы между минимумом и максимумом
            x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5 # фик-
сируем минимальные и максимальные значения по горизонтали
            y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5 # фик-
сируем минимальные и максимальные значения по вертикали
            # создаем пары «иксов» и «игреков» (горизонтальных и вер-
тикальных признаков)
             # равномерно распределенных от минимальных до максималь-
ных значений с шагом h
            # т.е. мы разбиваем область значений входных данных на рав-
номерную сетку
            xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                                 np.arange(y_min, y_max, h))
            if poly: # если мы используем полиномиальные признаки
             # то  сетку нужно преобразовать в соответствующие полино-
мы, иначе модель нас не поймет
                # считаем предсказание модели в «вероятностном виде»
                # с помощью метода ravel «выпрямляем» сетку в столбцы
                Z = self.predict_prob(to_polynom(np.c_[xx.ravel(), yy.
ravel()],order))-0.5
                # вычитаем 0.5, чтобы центровать вероятности: если мо-
дель не уверена, то будет 0
            else:
                # считаем предсказание модели в «вероятностном виде»
                # с помощью метода ravel «выпрямляем» сетку в столбцы
              Z = self.predict_prob(np.c_[xx.ravel(), yy.ravel()])-0.5
                # вычитаем 0.5, чтобы центровать вероятности: если мо-
дель не уверена, то будет 0
            cm = plt.cm.RdBu #
```

```
            Z = Z.reshape(xx.shape) # обратно преобразуем строку в сетку
             plt.contourf(xx, yy, Z, cmap=cm, alpha=.5) # отрисовываем
контур вероятности
            plt.xticks([],[])
            plt.yticks([],[])
            plt.tight_layout()
        #---------------------------------
        def classification_report(self, X,y):
            # считаем различные метрики классификации
            tp = 0 # true_positives
            tn = 0 # true_negatives
            fp = 0 # false_positives
            fn = 0 # false_negatives

            yhat  = self.predict(X)
            total = yhat.size
            n= sum(yhat==0)
            p = sum(yhat==1)
            # перебираем все точки и вручную заполняем матрицу ошибок
            for yhati,yi in zip(yhat,y):
                if yi == 1 and yhati == 1:
                    tp += 1
                elif yi == 0 and yhati == 0:
                    tn += 1
                elif yi == 1 and yhati == 0:
                    fn += 1
                elif yi == 0 and yhati == 1:
                    fp += 1
            # пишем все метрики
            print('True Positives:%.4f'%(tp/p), end = '\t')
            print('True Negatives:%.4f'%(tn/n))
            print('False Positives:%.4f'%(fp/p), end = '\t')
            print('False Negatives:%.4f'%(fn/n))
            print('Accuracy:%.4f'% ((tp + tn) / total))
            print('Recall:%.4f'% (tp / (tp + fn)), end = '\t')
            print('Precision:%.4f'%(tp / (tp + fp)))
            print('f1 measure:%.4f'%(tp / (tp + 0.5*(fp+fn))))
```

6. Класс уменьшения размерности методом главных компонент

```
import numpy as np
import matplotlib.pyplot as plt

class PCA():
    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None
        self.values = None
        self.mean = None
    #---------------------------------
    def fit(self, X):
          # обучение - в этом случае сводится к нахождению собственных значений и собственных векторов
        self.mean = np.mean(X, axis=0) # оценка среднего для каждого признака
        # считаем матрицу ковариации, используя функцию библиотеки Numpy
        cov_matrix = np.cov(X - self.mean, rowvar = False) # не забываем вычитать среднее
         # считаем собственные значения и собственные вектора матрицы ковариации
        eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix) # для этого тоже есть функция Numpy
        idx = eigenvalues.argsort()[::-1] #сортируем по возрастанию собственных значений
        # берем первые n собственных векторов
         self.components = eigenvectors[:, idx][:, :self.n_components]
        self.values     = eigenvalues[idx] # отсортированные собственные значения
        return self
    #---------------------------------
    def transform(self, X):
         # преобразование признаков в пространство главных компонент
        X = X - self.mean #вычитаем среднее
```

```
            return np.dot(X, self.components) #находим проекции признаков на собственные вектора (через скалярное произведение)
            #это и будут главные компоненты
        #--------------------------------
        def fit_transform(self, X):
            # 2 в 1: обучаем и преобразуем
            return self.fit(X).transform(X)
        #--------------------------------
        def inverse_transform(self, X_new):
            # обратное преобразование
                # главные компоненты скалярно домножаем на собственные вектора
            return np.dot(X_new, self.components.T) + self.mean # не забываем обратно добавить среднее
        #--------------------------------
        def score(self, X):
            # оцека «качества» восстановления - через коэффициент детерминации
            SStot = np.sum(np.square(X - np.mean(X)))
            SSres = np.sum(np.square(X - self.inverse_transform(self.fit_transform(X))))
            return 1 - SSres/SStot
        #--------------------------------
        def plot_eigvalues(self, figsize=(15,7)):
            # метод для отрисовки собственных значений (объясненной дисперсии)
            plt.figure(figsize=figsize)
            # отдельно мелкими точками визуализируем все собственные значения
            plt.plot(self.values,'.',
                     label='Все собственные значения',
                     linewidth = 3)
            # крупными маркерами - выбранное нами количество главных компонент
            plt.plot(self.values[:self.n_components],'r-o',
                     label='Собственное пространство',
                     markersize = 10, mfc='none',
                     linewidth = 2, alpha = 0.8)
            plt.ylabel('Собственные\n значения', fontsize=25)
            plt.grid();
```

```
        plt.legend(fontsize=25);
        plt.xticks(FontSize = 25); plt.yticks(FontSize = 25);
        plt.tight_layout();
```

7. Класс кластеризации методом *k*-средних

```
import numpy as np

class KMeans():
    def __init__(self,n_clusters = 2, centroids = None,
                 max_iter=10, tol=0.01,
                 metric = 'euclidean', p = 2,
                 random_state = None):
        self.n_clusters = n_clusters
        self.centroids  = centroids
        self.max_iter   = max_iter
        self.tol        = tol
        self.iters      = None
        self.inertia = None
        self.metric = metric
        self.p  = p
        self.random_state  = random_state
    #-------------------------------------
    def distance(self,X1, X2):
        # оценка расстояния
        if self.metric == 'euclidean':
            dist = np.sqrt(np.sum(np.square(X1 - X2).T,axis=0))
        if self.metric == 'cityblock':
            dist = np.sum(np.abs(X1 - X2).T,axis=0)
        if self.metric == 'Chebyshev':
            dist = np.max(np.abs(X1 - X2).T,axis=0)
        if self.metric == 'Minkowski':
                dist = np.power(np.sum(np.power(np.abs(X1 - X2),-
self.p).T,axis=0),1/self.p)
        return dist
    #-------------------------------------
    def init_centroids(self, X):
        # инициализация первых центров кластеров
```

```
        if self.random_state: rng = np.random.seed(self.random_state)
        c_idxs = np.random.randint(0, X.shape[0], size = self.n_clusters)
        return X[c_idxs,:]
    #-------------------------------------
    def predict(self, X):
        # оценка принадлежности точек к кластеру по расстоянию
        distances = np.zeros((X.shape[0], self.n_clusters))

        for i,centr in enumerate(self.centroids):
            distances[:,i] = self.distance(centr,X)
      self.inertia = np.sum(np.power(np.min(distances,axis = 1),2))
        return np.argmin(distances,axis = 1)
    #-------------------------------------
    def transform(self,X):
        # получение предсказаний
        return self.predict(X)
    #-------------------------------------
    def delta_centroids(self,old_centroids):
        # оценка относительного изменения центров кластеров
        return (
                self.distance(self.centroids,old_centroids)/
              self.distance(old_centroids, np.mean(old_centroids))
               ).mean()
    #-------------------------------------
    def fit(self, X):
        # обучение - несколько итераций алгоритма k-средних
        if self.centroids is None: # если центры кластеров не за-
даны - задаем
            self.centroids = self.init_centroids(X)

        d_centrs = np.inf

        for i in range(self.max_iter):

            old_centroids = np.copy(self.centroids)

            cluster_label = self.predict(X)

            for k in range(self.n_clusters):
```

```
                c_idxs = np.flatnonzero(cluster_label==k)

                self.centroids[k] = X[c_idxs].mean(axis = 0)

            d_centrs = self.delta_centroids(old_centroids)

            self.iters = i
            if d_centrs<=self.tol:
                break
        return self
    #-------------------------------------
    def fit_transform(self, X):
        # и обучаем, и сразу выдаем метки кластеров
        return self.fit(X).predict(X)
```

Учебное издание

Долганов Антон Юрьевич
Ронкин Михаил Владимирович
Созыкин Андрей Владимирович

БАЗОВЫЕ АЛГОРИТМЫ МАШИННОГО ОБУЧЕНИЯ НА ЯЗЫКЕ PYTHON

Редактор *Т. Е. Мерц*
Верстка *О. П. Игнатьевой*

Подписано в печать 04.04.2023. Формат 70×100 1/16.
Бумага офсетная. Цифровая печать. Усл. печ. л. 10,0.
Уч.-изд. л. 6,2. Тираж 30 экз. Заказ 25.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
Тел.: 8 (343) 375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: 8 (343) 358-93-06, 350-58-20, 350-90-13
Факс: 8 (343) 358-93-06
http://print.urfu.ru

ДОЛГАНОВ АНТОН ЮРЬЕВИЧ
Кандидат технических наук. Область научных интересов: машинное обучение, обработка биомедицинских сигналов, обработка естественного языка.

РОНКИН МИХАИЛ ВЛАДИМИРОВИЧ
Кандидат технических наук. Область научных интересов: машинное обучение, анализ временных рядов, компьютерное зрение.

СОЗЫКИН АНДРЕЙ ВЛАДИМИРОВИЧ
Кандидат технических наук. Область научных интересов: машинное обучение, разработка систем искусственного интеллекта.