

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Методические указания
к практическим занятиям по дисциплине
«ОБРАБОТКА ДАННЫХ И МАШИННОЕ ОБУЧЕНИЕ»
для магистров направления
«Прикладная математика и информатика»
профиль
«Математическое моделирование»

Ставрополь, 2026

Обработка данных и машинное обучение: Методические указания к практическим занятиям по дисциплине. - Ставрополь: СКФУ, 2023. - 113 с.

Пособие подготовлено в соответствии с Федеральным государственным образовательным стандартом высшего образования. В учебном пособии изложены методические рекомендации по выполнению 9 практических занятий.

Пособие предназначено для изучения дисциплины «Обработка данных и машинное обучение» для магистров направления подготовки «Прикладная математика и информатика».

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	4
1 ПРИМЕНЕНИЕ PYTHON В ЗАДАЧАХ МАШИННОГО ОБУЧЕНИЯ	5
2 КЛАССИФИКАЦИЯ	25
3 ЛИНЕЙНАЯ РЕГРЕССИЯ.....	44
4 ДЕРЕВЬЯ РЕШЕНИЙ	48
5 СЛУЧАЙНЫЙ ЛЕС	63
6 ГРАДИЕНТНЫЙ БУСТИНГ ДЕРЕВЬЕВ РЕГРЕССИИ	70
7 НЕЙРОННЫЕ СЕТИ	74
8 РАЗБИЕНИЕ ДАННЫХ НА ВЫБОРКИ.....	88
9 АНАЛИЗ ГЛАВНЫХ КОМПОНЕНТ .Ошибка! Закладка не определена.	
ЛИТЕРАТУРА	97

ВВЕДЕНИЕ

Основная цель учебной дисциплины - формирование знаний в области машинного обучения и развития навыков алгоритмического построения статистических моделей.

Задачей курса является изучение основ алгоритмов машинного обучения на основе регрессивных моделей, нейронных сетей и др.

Освоение дисциплины «Высокопроизводительные вычисления» позволит будущему магистру по направлению подготовки «Прикладная математика и информатика» полноценно осуществлять свою профессиональную деятельность.

Аппаратура и материалы

Практикум выполняется в штатном компьютерном классе, оборудованном IBM-совместимыми персональными компьютерами под управлением операционной системы Windows. Необходимым является наличие инструментальной среды языка программирования Python.

Указания по технике безопасности

Указания по технике безопасности в штатном компьютерном классе разработаны и утверждены администрацией и доводятся каждому студенту под роспись. Дополнительных требований нет.

1 ПРИМЕНЕНИЕ PYTHON В ЗАДАЧАХ МАШИННОГО ОБУЧЕНИЯ

Цель: Изучить средства языка программирования Python, используемые в задачах машинного обучения.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Машинное обучение заключается в извлечении знаний из данных. Это научная область, находящаяся на пересечении статистики, искусственного интеллекта и компьютерных наук и также известная как прогнозная аналитика или статистическое обучение. В последние годы применение методов машинного обучения в повседневной жизни стало обыденным явлением. Многие современные веб-сайты и устройства используют алгоритмы машинного обучения, начиная с автоматических рекомендаций по просмотру фильмов, заказа еды или покупки продуктов, и заканчивая персонализированными онлайн-радиотрансляциями и распознаванием друзей на фотографиях. Когда вы видите сложный сайт типа Facebook, Amazon или Netflix, то весьма вероятно, что каждый раздел сайта содержит несколько моделей машинного обучения.

Выйдя за пределы коммерческих приложений, машинное обучение уже оказало огромное влияние на научные исследования, управляемые данными. Инструменты, представленные в этой книге, использовались для решения различных научных задач (исследование звезд, поиск далеких планет, открытие новых частиц, анализ последовательностей ДНК, а также разработка персонализированных методов лечения рака).

Для извлечения прибыли с помощью машинного обучения совсем не обязательно, чтобы ваши задачи были столь же крупномасштабными или меняющими мир, как представленные примеры. В этой главе мы объясним, почему машинное обучение стало таким популярным, и обсудим, какие задачи могут быть решены с помощью него. Затем мы покажем вам, как построить свою первую модель машинного обучения, попутно знакомя вас с важными принципами машинного обучения.

На заре появления «интеллектуальных» приложений многие системы использовали жесткие правила «if» и «else» для обработки данных или корректировки информации, введенной пользователем. Вспомните о спам-филт্রে, чья работа состоит в том, чтобы переместить соответствующие входящие сообщения электронной почты в папку «Спам». Вы можете составить черный список слов, которые будут идентифицировать письмо как спам. Это пример использования системы экспертных правил для разработки «интеллектуального» приложения. Разработка правил принятия решений в ручном режиме допустимо в некоторых задачах, особенно в тех, где люди четко понимают процесс моделирования. Однако, использование жестких решающих правил имеет два основных недостатка:

- Логика, необходимая для принятия решения, относится исключительно к одной конкретной области и задачи. Даже незначительное изменение

задачи может повлечь за собой переписывание всей системы.

- Разработка правил требует глубокого понимания процесса принятия решения.

Один из примеров, где этот жесткий подход потерпит неудачу – это распознавание лиц на изображениях. На сегодняшний день каждый смартфон может распознать лицо на изображении. Тем не менее, распознавание лиц была нерешенной проблемой, по крайней мере, до 2001 года. Основная проблема заключается в том, что способ, с помощью которого компьютер «воспринимает» пиксели, формирующие изображение на компьютере, очень сильно отличается от человеческого восприятия лица. Эта разница в принципе не позволяет человеку сформулировать подходящий набор правил, описывающих лицо с точки зрения цифрового изображения.

Однако, благодаря машинному обучению, простого предъявления большого количества изображений с лицами будет достаточно для того, чтобы алгоритм определил, какие признаки необходимы для идентификации лица.

Наиболее успешные алгоритмы машинного обучения – это те, которые автоматизируют процессы принятия решений путем обобщения известных примеров. В этих методах, известных как обучение с учителем или контролируемое обучение (supervised learning), пользователь предоставляет алгоритму пары объект-ответ, а алгоритм находит способ получения ответа по объекту. В частности, алгоритм способен выдать ответ для объекта, которого он никогда не видел раньше, без какой-либо помощи человека. Если вернуться к примеру классификации спама с использованием машинного обучения, пользователь предъявляет алгоритму большое количество писем (объекты) вместе с информацией о том, является ли письмо спамом или нет (ответы). Для нового электронного письма алгоритм вычислит вероятность, с которой это письмо можно отнести к спаму.

Алгоритмы машинного обучения, которые учатся на парах объект-ответ, называются алгоритмами обучения с учителем, так как «учитель» показывает алгоритму ответ в каждом наблюдении, по которому происходит обучение. Несмотря на то, что создание набора с объектами и ответами – это часто трудоемкий процесс, осуществляемый вручную, алгоритмы обучения с учителем интерпретируемы и качество их работы легко измерить. Если вашу задачу можно сформулировать в виде задачи обучения с учителем, и вы можете создать набор данных, который включает в себя ответы, вероятно, машинное обучение решит вашу проблему.

1. Примеры задач машинного обучения с учителем:

1.1. Определение почтового индекса по рукописным цифрам на конверте

Здесь объектом будет сканированное изображение почерка, а ответ – фактические цифры почтового индекса. Чтобы создать набор данных для построения модели машинного обучения, вам нужно собрать большое количество конвертов. Затем вы можете самостоятельно прочитать почтовые индексы

сы и сохранить цифры в виде ответов.

1.2. Определение доброкачественности опухоли на основе медицинских изображений

Здесь объектом будет изображение, а ответом – диагноз, является ли опухоль доброкачественной или нет. Чтобы создать набор данных для построения модели, вам нужна база медицинских изображений. Кроме того, необходимо мнение эксперта, поэтому врач должен просмотреть все изображения и решить, какие опухоли являются доброкачественными, а какие – нет. Помимо анализа изображения может понадобиться дополнительная диагностика для определения доброкачественности опухоли.

1.3. Обнаружение мошеннической деятельности в сделках по кредитным картам

Здесь объект – запись о транзакции по кредитной карте, а ответ – информация о том, является ли транзакция мошеннической или нет. Предположим, вы – учреждение, выдающее кредитные карты, сбор данных подразумевает сохранение всех транзакций и запись сообщений клиентов о мошеннических транзакциях.

Приведя эти примеры, интересно отметить что, хотя объекты и ответы выглядят достаточно просто, процесс сбора данных для этих трех задач существенно отличается. Несмотря на то что чтение конвертов является трудоёмким занятием, этот процесс прост и дешев. Получение медицинских изображений и проведение диагностики требует не только дорогостоящего оборудования, но и редких, высокооплачиваемых экспертных знаний, не говоря уже об этических проблемах и вопросах конфиденциальности. В примере обнаружения мошенничества с кредитными картами сбор данных осуществляется намного проще. Ваши клиенты сами предоставят вам ответы, сообщая о мошенничестве. Все, что вам нужно сделать для получения объектов и ответов, связанных с мошеннической активностью, – это подождать.

Алгоритмы обучения без учителя или неконтролируемого обучения (unsupervised algorithms) – это еще один вид алгоритмов, который мы рассмотрим в этой книге. В алгоритмах обучения без учителя известны только объекты, а ответов нет. Хотя есть много успешных сфер применения этих методов, их, как правило, труднее интерпретировать и оценить.

2. Примеры задач машинного обучения без учителя:

2.1. Определение тем в наборе постов

Если у вас есть большая коллекция текстовых данных, вы можете агрегировать их и найти распространённые темы. У вас нет предварительной информации о том, какие темы там затрагиваются и сколько их. Таким образом, нет никаких известных ответов.

Сегментирование клиентов на группы с похожими предпочтениями. Имея набор записей о клиентах, вы можете определить группы клиентов со схожими предпочтениями. Для торгового сайта такими группами могут

быть «родители», «книгочеи» или «геймеры». Поскольку вы не знаете заранее о существовании этих групп и их количестве, у вас нет ответов.

2.2.Обнаружение паттернов аномального поведения на веб-сайте

Чтобы выявить злоупотребления или ошибки, часто бывает полезно найти паттерны поведения, которые отличаются от нормы. Паттерны аномального поведения могут быть разными, и, возможно, у вас не будет зарегистрированных случаев аномального поведения. Поскольку в этом примере вы наблюдаете лишь трафик, и вы не знаете, что представляет собой нормальное и ненормальное поведение, речь идет о задаче обучения без учителя.

Решая задачи машинного обучения с учителем и без, важно представить ваши входные данные в формате, понятном компьютеру. Часто данные представляют в виде таблицы. Каждая точка данных, которую вы хотите исследовать (каждое электронное письмо, каждый клиент, каждая транзакция) является строкой, а каждое свойство, которое описывает эту точку данных (скажем, возраст клиента, сумма или место совершения транзакции), является столбцом. Вы можете описать пользователей по возрасту, полу, дате создания учетной записи и частоте покупок в вашем интернет-магазине. Вы можете описать изображение опухоли с помощью градаций серого цвета для каждого пикселя или с помощью размера, формы и цвета опухоли.

В машинном обучении каждый объект или строка называются примером (sample) или точкой данных (data point), а столбцы-свойства, которые описывают эти примеры, называются характеристиками или признаками (features).

Позже в этой книге мы более детально остановимся на теме подготовки данных, которая называется выделение признаков (feature extraction) или конструирование признаков (feature engineering). Однако, вы должны иметь в виду, что ни один алгоритм машинного обучения не сможет сделать прогноз по данным, которые не содержат никакой полезной информации. Например, если единственный признак пациента – это его фамилия, алгоритм не сможет предсказать его пол. Этой информации просто нет в данных. Если добавить еще один признак – имя пациента, то дело уже будет обстоять лучше, поскольку часто, зная имя человека, можно судить о его поле.

Вполне возможно, что самая важная часть процесса машинного обучения – это интерпретация данных, с которыми вы работаете, и применимость этих данных к задаче, которую вы хотите решить. Выбрать случайным образом алгоритм и скормить ему свои данные – неэффективное решение. Прежде чем приступить к построению модели, необходимо понять, что представляет собой ваш набор данных. Каждый алгоритм отличается с точки зрения типа обрабатываемых данных и вида решаемых задач. Создавая модель машинного обучения, вы должны ответить, или, по крайней мере, задуматься над следующими вопросами:

- На какой вопрос(ы) я пытаюсь ответить? Собранные данные могут ответить на этот вопрос?
- Как лучше всего сформулировать свой вопрос(ы) с точки зрения задач

машинного обучения?

- У меня собрано достаточно данных, чтобы составить представление о задаче, которую я хочу решить?
- Какие признаки я извлек и помогут ли они мне получить правильные прогнозы?
- Как я буду измерять эффективность решения задачи?
- Как решение, полученное с помощью машинного обучения, будет взаимодействовать с другими компонентами моего исследования или бизнес-продукта?

В более широком контексте, алгоритмы и методы машинного обучения являются лишь этапом более крупного процесса, призванного решить конкретную задачу, и поэтому необходимо всегда держать схему этого процесса в голове. Многие тратят массу времени, создавая сложные модели машинного обучения решения, затем узнавая, что решают задачу неправильно.

Углубляясь в технические аспекты машинного обучения (что мы и будем делать этой книге), легко упустить из виду конечные цели. Несмотря на то, что мы не будем подробно обсуждать вопросы, перечисленные здесь, мы все же рекомендуем вам вспомнить о них, когда вы начнете строить модели машинного обучения.

Python стал общепринятым языком для многих сфер применения науки о данных (data science). Он сочетает в себе мощь языков программирования с простотой использования предметноориентированных скриптовых языков типа MATLAB или R. В Python есть библиотеки для загрузки данных, визуализации, статистических вычислений, обработки естественного языка, обработки изображений и многого другого. Этот обширный набор инструментов предлагает специалистам по работе с данными (data scientists) большой набор инструментов общего и специального назначения. Одним из основных преимуществ использования Python является возможность напрямую работать с программным кодом с помощью терминала или других инструментов типа Jupyter Notebook, который мы рассмотрим ниже. Машинное обучение и анализ данных – это в основном итерационные процессы, в которых данные задают ход анализа. Крайне важно для этих процессов иметь инструменты, которые позволяют оперативно и легко работать.

В качестве языка программирования общего назначения Python позволяет создавать сложные графические пользовательские интерфейсы (GUI) и веб-сервисы, а также легко интегрироваться в уже существующие системы.

Scikit-learn – проект с открытым исходным кодом, это означает, что его можно свободно использовать и распространять, и любой человек может легко получить исходный код, чтобы увидеть, что происходит «за кулисами». Проект scikit-learn постоянно развивается и совершенствуется, и у него очень активное сообщество пользователей. Он содержит ряд современных алгоритмов машинного обучения, а также полную документацию по каждому алгоритму. scikit-learn – очень популярный инструмент и самая известная питоновская библиотека для машинного обучения. Она широко используется в

промышленности и науке, а в интернете имеется богатый выбор обучающих материалов и примеров программного кода. scikit-learn прекрасно работает с рядом других научных инструментов Python, которые мы обсудим позже в этой главе.

По мере чтения книги мы рекомендуем также ознакомиться с руководством пользователя по scikit-learn и документацией по API для получения дополнительной информации о многочисленных параметрах каждого алгоритма. Онлайн-документация является очень подробной, и эта книга познакомит вас со всеми необходимыми основами машинного обучения, чтобы вы научились детально разбираться в нем.

Scikit-learn требует наличия еще двух пакетов Python – NumPy и SciPy. Для построения графиков и интерактивной работы необходимо также установить matplotlib, IPython и Jupyter Notebook. Мы рекомендуем использовать один из нижеперечисленных дистрибутивов Python, которые уже включают все необходимые **пакеты**:

Anaconda

Дистрибутив Python, предназначенный для крупномасштабной обработки данных, прогнозной аналитики и научных вычислений. Anaconda уже включает NumPy, SciPy, matplotlib, pandas, IPython, Jupyter Notebook и scikit-learn. Есть версии для Mac OS, Windows и Linux. Это очень удобное решение и это тот дистрибутив, который мы рекомендуем пользователям, у которых еще не установлены пакеты Python для научных вычислений. Кроме того, сейчас Anaconda включает в себя коммерческую библиотеку Intel MKL, которой можно пользоваться бесплатно. Использование MKL (это происходит автоматически при установке Anaconda) может дать значительный прирост скорости при выполнении различных алгоритмов в scikitlearn.

Enthought Canopy

Еще один дистрибутив Python для научных вычислений. Он уже содержит NumPy, SciPy, matplotlib, pandas и IPython, но бесплатная версия не включает scikit-learn. Если вы являетесь учебным заведением, вы можете запросить учебную лицензию и получить свободный доступ к платной версии Enthought Canopy. Enthought Canopy доступен для Python 2.7.x и работает на Mac OS, Windows и Linux.

Python (x, y)

Свободный дистрибутив Python для научных вычислений, специально предназначенный для Windows. Python (x, y) включает NumPy, SciPy, matplotlib, pandas, IPython и scikit-learn.

Если у вас уже стоит Python, вы можете использовать pip для установки всех этих пакетов:

```
$ pip install numpy scipy matplotlib ipython  
scikit-learn pandas
```

Понимание принципов работы и использования scikit-learn – важно, но есть несколько других библиотек, которые расширят ваш опыт. scikit-learn базируется на двух питоновских библиотеках для научных вычислений

NumPy и SciPy. Помимо NumPy и SciPy мы будем использовать pandas и matplotlib. Кроме того, мы познакомимся с Jupyter Notebook, который представляет собой интерактивную среду программирования на основе браузера. Если коротко, то ниже приводится информация о перечисленных инструментах, которыми вы должны овладеть, чтобы получить максимальную отдачу от scikitlearn.

Jupyter Notebook представляет собой интерактивную среду для запуска программного кода в браузере. Это отличный инструмент для разведочного анализа данных и широко используется специалистами по анализу данных. Несмотря на то что Jupyter Notebook поддерживает множество языков программирования, нам нужна лишь поддержка Python. Jupyter Notebook позволяет легко интегрировать программный код, текст и изображения, и вся эта книга была фактически написана в формате Jupyter Notebook. Все примеры программного кода, приведенные в этой книге, можно загрузить с GitHub.

NumPy – это один из основных пакетов для научных вычислений в Python. Он содержит функциональные возможности для работы с многомерными массивами, высокоуровневыми математическими функциями (операции линейной алгебры, преобразование Фурье, генератор псевдослучайных чисел).

В scikit-learn массив NumPy – это основная структура данных. scikit-learn принимает данные в виде массивов NumPy. Любые данные, которые вы используете, должны быть преобразованы в массив NumPy. Базовый функционал NumPy – это класс ndarray, многомерный (nмерный) массив. Все элементы массива должны быть одного и того же типа. Массив NumPy выглядит следующим образом:

```
In[2]: import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))
Out[2]: x: [[1 2 3] [4 5 6]]
```

SciPy – это набор функций для научных вычислений в Python. Помимо всего прочего он предлагает продвинутые процедуры линейной алгебры, математическую оптимизацию функций, обработку сигналов, специальные математические функции и статистические функции. scikit-learn использует набор функций SciPy для реализации своих алгоритмов. Для нас наиболее важной частью SciPy является пакет scipy.sparse: с помощью него мы получаем разреженные матрицы (sparse matrices), которые представляют собой еще один формат данных, который используется в scikit-learn. Разреженные матрицы используются всякий раз, когда нам нужно сохранить 2D массив, который содержит в основном нули:

```
In[3]:
from scipy import sparse
# Создаем 2D массив NumPy с единицами по главной
диагонали и нулями в остальных ячейках eye =
np.eye(4)
```

```

print("массив NumPy:\n{}".format(eye))    Out[3]:
массив NumPy: [[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
In[4]:
# Преобразовываем массив NumPy в разреженную
матрицу SciPy в формате CSR
# Сохраняем лишь ненулевые элементы sparse_matrix =
sparse.csr_matrix(eye)
print("\nразреженная матрица SciPy в формате
CSR:\n{}".format(sparse_matrix))    Out[4]:
разреженная матрица SciPy в формате CSR:
(0, 0) 1.0
(1, 1) 1.0
(2, 2) 1.0
(3, 3) 1.0

```

Обычно невозможно плотно записать разреженные данные (поскольку они не уместились бы в памяти), поэтому нам нужно непосредственно создать разреженные матрицы. Ниже приводится способ, который позволяет создать такую же разреженную матрицу, что была приведена выше, но этот раз с использованием формата COO:

```

In[5]: data = np.ones(4) row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices,
col_indices))) print("формат
COO:\n{}".format(eye_coo))    Out[5]: формат COO: (0,
0) 1.0
(1, 1) 1.0
(2, 2) 1.0
(3, 3) 1.0

```

matplotlib – это основная библиотека для построения научных графиков в Python. Она включает функции для создания высококачественных визуализаций типа линейных диаграмм, гистограмм, диаграмм разброса и т.д. Визуализация данных и различных аспектов вашего анализа может дать вам важную информацию, и мы будем использовать matplotlib для всех наших визуализаций. При работе в Jupyter Notebook, вы можете вывести рисунок прямо в браузере с помощью встроенных команд `%matplotlib notebook` и `%matplotlib inline`. Мы рекомендуем использовать `%matplotlib notebook`, который предлагает интерактивное окружение (хотя при написании этой книги мы использовали `%matplotlib inline`). Например, нижеприведенный программный код строит график, изображенный на рис. 1.1:

```

In[6]:
%matplotlib inline
import matplotlib.pyplot as plt

```

```
# Генерируем последовательность чисел от -10 до 10
с 100 шагами x = np.linspace(-10, 10, 100)
#Создаем второй массив с помощью синуса y=np.sin(x)
# Функция создает линейный график на основе двух
массивов plt.plot(x, y, marker="x")
```

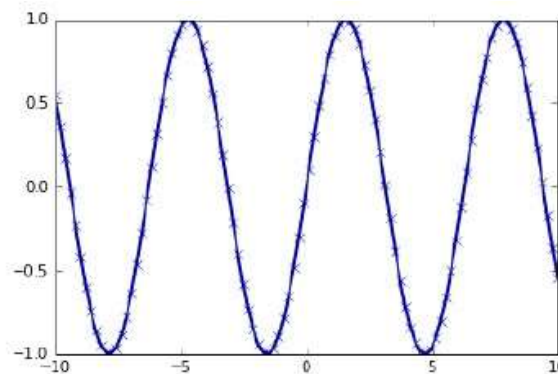


Рисунок 1.1 - Простой линейный график синусоидальной функции с использованием matplotlib

pandas – библиотека Python для обработки и анализа данных. Она построена на основе структуры данных, называемой DataFrame и смоделированной по принципу датафреймов среды статистического программирования R. Проще говоря, DataFrame библиотеки pandas представляет собой таблицу, похожую на электронную таблицу Excel. Библиотека pandas предлагает большой спектр методов по работе с этой таблицей, в частности, она позволяет выполнять SQL-подобные запросы и присоединения таблиц. В отличие от NumPy, который требует, чтобы все записи в массиве были одного и того же типа, в pandas каждый столбец может иметь отдельный тип (например, целые числа, даты, числа с плавающей точкой и строки). Еще одним преимуществом библиотеки pandas является ее способность работать с различными форматами файлов и баз данных, например, с файлами SQL, Excel и CSV. Детальное рассмотрение функционала pandas выходит за рамки этой книги. Ниже приводится небольшой пример создания DataFrame с помощью словаря:

```
In[7]: import pandas as pd
# создаем простой набор данных пользователей
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location': ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33] }
data_pandas = pd.DataFrame(data)
#IPython.display позволяет напечатать датафреймы
# в Jupyter notebook display(data_pandas)
```

Приведенный код генерирует следующий вывод:

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

Существует несколько способов осуществить запрос к таблице. Например:

```
In[8]:
#Выбрать строки, в которых значение столбца age>30
display(data_pandas[data_pandas.Age > 30])
```

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

ПРАКТИЧЕСКАЯ ЧАСТЬ

Рассмотрим простой пример применения машинного обучения и построим нашу первую модель. В процессе изложения материала мы познакомим вас с некоторыми основными принципами и терминами.

Предположим, что ботаник-любитель хочет классифицировать сорта ирисов, которые он собрал. Он измерил в сантиметрах некоторые характеристики ирисов: длину и ширину лепестков, а также длину и ширину чашелистиков (см. рис. 1.2).

Кроме того, у него есть измерения этих же характеристик ирисов, которые ранее позволили опытному эксперту отнести их к сортам *setosa*, *versicolor* и *virginica*. Относительно этих ирисов ботаник-любитель уверенно может сказать, к какому сорту принадлежит каждый ирис. Давайте предположим, что перечисленные сорта являются единственными сортами, которые ботаник-любитель может встретить в дикой природе.

Наша цель заключается в построении модели машинного обучения, которая сможет обучиться на основе характеристик ирисов, уже классифицированных по сортам, и затем предскажет сорт для нового цветка ириса.

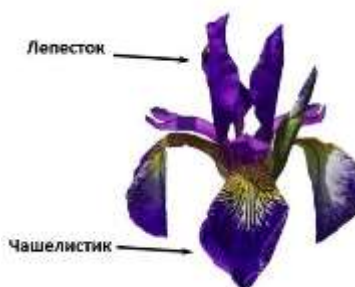


Рис. 1.2 - Структура цветка ириса

Поскольку у нас есть примеры, по которым мы уже знаем правильные сорта ириса, решаемая задача является задачей обучения с учителем. В этой задаче нам нужно спрогнозировать один из сортов ириса. Это пример задачи классификации (classification). Возможные ответы (различные сорта ириса) называются классами (classes). Каждый ирис в наборе данных принадлежит к одному из трех классов, таким образом решаемая задача является задачей трехклассовой классификации.

Ответом для отдельной точки данных (ириса) является тот или иной сорт этого цветка. Сорт, к которому принадлежит цветок (конкретная точка данных), называется меткой (label).

Данные, которые мы будем использовать для этого примера, – это набор данных Iris, классический набор данных в машинном обучении и статистике. Он уже включен в модуль datasets библиотеки scikit-learn. Мы можем загрузить его, вызвав функцию load_iris:

```
In[10]:
from sklearn.datasets import load_iris iris_dataset
= load_iris()
```

Объект iris, возвращаемый load_iris, является объектом Bunch, который очень похож на словарь. Он содержит ключи и значения:

```
In[11]:
print("Ключи iris_dataset:
\n{}".format(iris_dataset.keys()))
```

```
Out[11]:
Ключи iris_dataset:
dict_keys(['target_names', 'feature_names', 'DESCR', 'data', 'target'])
```

Значение ключа DESCR – это краткое описание набора данных. Здесь мы покажем начало описания (оставшуюся часть описания вы можете посмотреть самостоятельно):

```
In[12]: print(iris_dataset['DESCR'][:193] +
"\n...")
```

```
Out[12]:
Iris Plants Database
=====
Notes
----
Data Set Characteristics:
: Number of Instances: 150 (50 in each of three
classes) : Number of Attributes: 4 numeric, predic-
tive att ...
----
```

Значение ключа target_names – это массив строк, содержащий сорта

цветов, которые мы хотим предсказать:

```
In[13]: print("Названия ответов:  
{0}".format(iris_dataset['target_names']))
```

```
Out[13]:
```

```
Названия ответов: ['setosa' 'versicolor' 'virginica']
```

Значение `feature_names` – это список строк с описанием каждого признака:

```
In[14]:  
print("Названия признаков:  
\n{0}".format(iris_dataset['feature_names']))
```

```
Out[14]:
```

```
Названия признаков:
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal  
length (cm)', 'petal width (cm)']
```

Сами данные записаны в массивах `target` и `data`. `data` – массив NumPy, который содержит количественные измерения длины чашелистиков, ширины чашелистиков, длины лепестков и ширины лепестков:

```
In[15]:  
print("Тип массива data:  
{0}".format(type(iris_dataset['data'])))
```

```
Out[15]:
```

```
Тип массива data: <class 'numpy.ndarray'>
```

Строки в массиве `data` соответствуют цветам ириса, а столбцы представляют собой четыре признака, которые были измерены для каждого цветка:

```
In[16]:  
print("Форма массива data:  
{0}".format(iris_dataset['data'].shape))
```

```
Out[16]:
```

```
Форма массива data: (150, 4)
```

Мы видим, что массив содержит измерения для 150 различных цветов по 4 признакам. Вспомним, что в машинном обучении отдельные элементы называются примерами (`samples`), а их свойства – характеристиками или признаками (`feature`). Форма (`shape`) массива данных определяется количеством примеров, умноженным на количество признаков. Это является общепринятым соглашением в `scikit-learn`, и ваши данные всегда будут представлены в этой форме. Ниже приведены значения признаков для первых пяти примеров:

```
In[17]:  
print("Первые пять строк массива data:  
{0}".format(iris_dataset['data'][:5]))
```

```
Out[17]:
```

```
Первые пять строк массива data:
```

```
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]
```

Взглянув на эти данные, мы видим, что все пять цветов имеют ширину лепестка 0.2 см и первый цветок имеет самую большую длину чашелистика, 5.1 см.

Массив `target` содержит сорта уже измеренных цветов, тоже записанные в виде массива NumPy:

```
In[18]: print("Тип массива target:
{}").format(type(iris_dataset['target']))
```

```
Out[18]:
```

```
Тип массива target: <class 'numpy.ndarray'>
```

`target` представляет собой одномерный массив, по одному элементу для каждого цветка:

```
In[19]: print("Форма массива target:
{}").format(iris_dataset['target'].shape)
```

```
Out[19]:
```

```
Форма массива target: (150,)
```

Сорта кодируются как целые числа от 0 до 2:

```
In[20]
```

```
print("Ответы:\n{}").format(iris_dataset['target'])
```

```
Out[20]:
```

```
Ответы:
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2
2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2
2 2]
```

Значения чисел задаются массивом `iris['target_names']`: 0 – *setosa*, 1 – *versicolor*, а 2 – *virginica*.

На основе этих данных нам нужно построить модель машинного обучения, которая предскажет сорта ириса для нового набора измерений. Но прежде, чем мы применим нашу модель к новому набору, мы должны убедиться в том, что модель на самом деле работает и ее прогнозам можно доверять.

К сожалению, для оценки качества модели мы не можем использовать данные, которые были взяты нами для построения модели. Это обусловлено тем, что наша модель просто запомнит весь обучающий набор и поэтому она

всегда будет предсказывать правильную метку для любой точки данных в обучающем наборе. Это «запоминание» ничего не говорит нам об обобщающей способности модели (другими словами, мы не знаем, будет ли эта модель так же хорошо работать на новых данных).

Для оценки эффективности модели, мы предъявляем ей новые размеченные данные (размеченные данные, которые она не видела раньше). Обычно это делается путем разбиения собранных размеченных данных (в данном случае 150 цветов) на две части. Одна часть данных используется для построения нашей модели машинного обучения и называется обучающими данными (training data) или обучающим

набором (training set). Остальные данные будут использованы для оценки качества модели, их называют тестовыми данными (test data),

тестовым набором (test set) или контрольным набором (hold-out set).

В библиотеке scikit-learn есть функция `train_test_split`, которая перемешивает набор данных и разбивает его на две части. Эта функция отбирает в обучающий набор 75% строк данных с соответствующими метками. Оставшиеся 25% данных с метками объявляются тестовым набором. Вопрос о том, сколько данных отбирать в обучающий и тестовый наборы, является дискуссионным, однако использование тестового набора, содержащего 25% данных, является хорошим правилом.

В scikit-learn данные, как правило, обозначаются заглавной X , тогда как метки обозначаются строчной y . Это навеяно стандартной математической формулой $f(x)=y$, где x является аргументом функции, а y – выводом. В соответствии с некоторыми математическими соглашениями мы используем заглавную X , потому что данные представляют собой двумерный массив (матрицу) и строчную y , потому что целевая переменная – это одномерный массив (вектор).

Давайте вызовем функцию `train_test_split` для наших данных и зададим обучающие данные, обучающие метки, тестовые данные, тестовые метки, используя вышеупомянутые буквы:

```
In[21]: from sklearn.model_selection import
train_test_split X_train, X_test, y_train, y_test =
train_test_split( iris_dataset['data'],
iris_dataset['target'], random_state=0)
```

Перед разбиением функция `train_test_split` перемешивает набор данных с помощью генератора псевдослучайных чисел. Если мы просто возьмем последние 25% наблюдений в качестве тестового набора, все точки данных будут иметь метку 2, поскольку все точки данных отсортированы по меткам (смотрите вывод для `iris['target']`, показанный ранее). Используя тестовый набор, содержащий только один из трех классов, вы не сможете объективно судить об обобщающей способности модели, таким образом, мы перемешиваем наши данные, чтобы тестовые данные содержали все три класса.

Чтобы в точности повторно воспроизвести полученный результат, мы

воспользуемся генератором псевдослучайных чисел с фиксированным стартовым значением, которое задается с помощью параметра `random_state`. Это позволит сделать результат воспроизводим, поэтому вышеприведенный программный код будет генерировать один и тот же результат. Мы всегда будем задавать `random_state` при использовании рандомизированных процедур в этой книге.

Выводом функции `train_test_split` являются `X_train`, `X_test`, `y_train` и `y_test`, которые все являются массивами Numpy. `X_train` содержит 75% строк набора данных, а `X_test` содержит оставшиеся 25%:

```
In[22]: print("форма массива X_train:
{}").format(X_train.shape) print("форма массива
y_train: {}".format(y_train.shape))
Out[22]: форма массива X_train: (112, 4) форма
массива y_train: (112,) In[23]:
print("форма массива X_test:
{}").format(X_test.shape) print("форма массива
y_test: {}".format(y_test.shape))
Out[23]: форма массива X_test: (38, 4) форма
массива y_test: (38,)
```

Перед тем как строить модель машинного обучения, неплохо было бы исследовать данные, чтобы понять, можно ли легко решить поставленную задачу без машинного обучения или содержится ли нужная информация в данных.

Кроме того, исследование данных – это хороший способ обнаружить аномалии и особенности. Например, вполне возможно, что некоторые из ваших ирисов измерены в дюймах, а не в сантиметрах. В реальном мире нестыковки в данных и неожиданности очень распространены.

Один из лучших способов исследовать данные – визуализировать их. Это можно сделать, используя диаграмму рассеяния (scatter plot). В диаграмме рассеяния один признак откладывается по оси *x*, а другой признак – по оси *y*, каждому наблюдению соответствует точка. К сожалению, экран компьютера имеют только два измерения, что позволяет разместить на графике только два (или, возможно, три) признака одновременно. Таким образом, трудно разместить на графике наборы данных с более чем тремя признаками. Один из способов решения этой проблемы – построить матрицу диаграмм рассеяния (scatterplot matrix) или парные диаграммы рассеяния (pair plots), на которых будут изображены все возможные пары признаков. Если у вас есть небольшое число признаков, например, четыре, как здесь, то использование матрицы диаграмм рассеяния будет вполне разумным. Однако, вы должны помнить, что матрица диаграмм рассеяния не показывает взаимодействие между всеми признаками сразу, поэтому некоторые интересные аспекты данных не будут выявлены с помощью этих графиков.

Рис. 1.3 представляет собой матрицу диаграмм рассеяния для признаков обучающего набора. Точки данных окрашены в соответствии с сортами

ириса, к которым они относятся. Чтобы построить диаграммы, мы сначала преобразовываем массив NumPy в DataFrame (основный тип данных в библиотеке pandas). В pandas есть функция для создания парных диаграмм рассеяния под названием `scatter_matrix`. По диагонали этой матрицы располагаются гистограммы каждого признака:

In[24]:

```
# создаем dataframe из данных в массиве X_train
# маркируем столбцы, используя строки в
iris_dataset.feature_names iris_dataframe =
pd.DataFrame(X_train, col-
umns=iris_dataset.feature_names) # создаем матрицу
рассеяния из dataframe, цвет точек задаем с помощью
y_train grr = pd.scatter_matrix(iris_dataframe,
c=y_train, figsize=(15, 15), marker='o',
hist_kwds={'bins': 20}, s=60, alpha=.8,
cmap=mglearn.cm3)
```

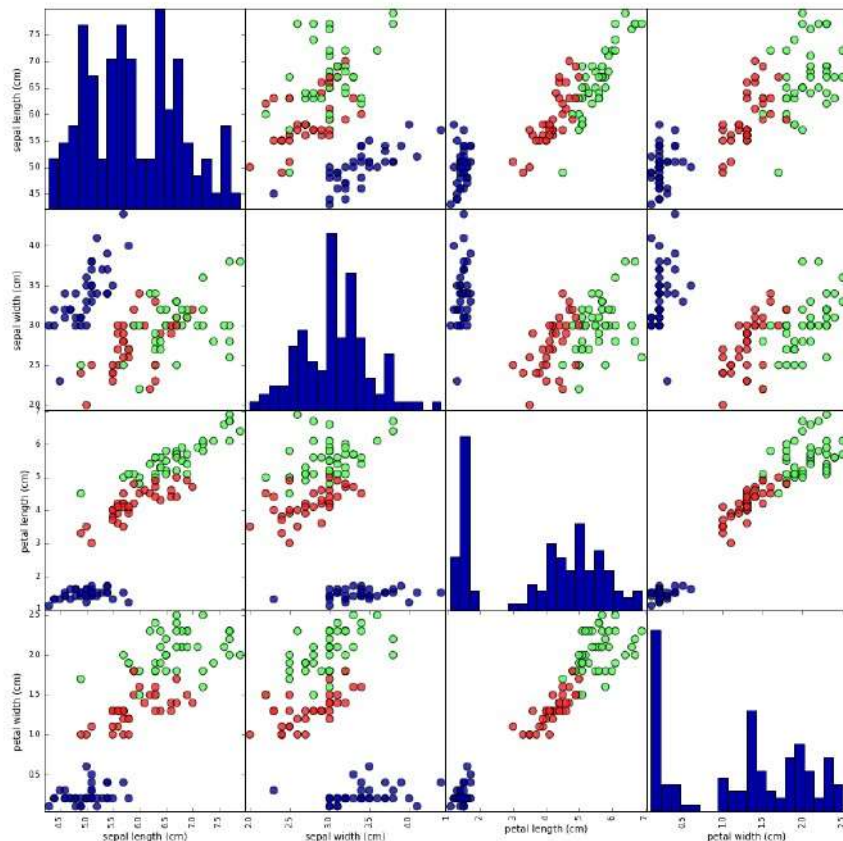


Рис. 1.3 - Матрица диаграмм рассеяния для набора данных Iris, цвет точек данных определяется метками классов

Взглянув на график, мы можем увидеть, что, похоже, измерения чашелистиков и лепестков позволяют относительно хорошо разделить три класса. Это означает, что модель машинного обучения, вероятно, сможет научиться разделять их.

Теперь мы можем начать строить реальную модель машинного обуче-

ния. В библиотеке `scikit-learn` имеется довольно много алгоритмов классификации, которые мы могли бы использовать для построения модели. В данном примере мы будем использовать классификатор на основе метода k ближайших соседей, который легко интерпретировать. Построение этой модели заключается лишь в запоминании обучающего набора. Для того, чтобы сделать прогноз для новой точки данных, алгоритм находит точку в обучающем наборе, которая находится ближе всего к новой точке. Затем он присваивает метку, принадлежащую этой точке обучающего набора, новой точке данных.

k в методе k ближайших соседей означает, что вместо того, чтобы использовать лишь ближайшего соседа новой точки данных, мы в ходе обучения можем рассмотреть любое фиксированное число (k) соседей (например, рассмотреть ближайшие три или пять соседей). Тогда мы можем сделать прогноз для точки данных, используя класс, которому принадлежит большинство ее соседей. Подробнее мы поговорим об этом в главе 2, а в данный момент мы будем использовать только одного соседа.

В `scikit-learn` все модели машинного обучения реализованы в собственных классах, называемых классами `Estimator`. Алгоритм классификации на основе метода k ближайших соседей реализован в классификаторе `KNeighborsClassifier` модуля `neighbors`. Прежде чем использовать эту модель, нам нужно создать объект-экземпляр класса. Это произойдет, когда мы зададим параметры модели. Самым важным параметром `KNeighborsClassifier` является количество соседей, которые мы установим равным 1:

```
In[25]: from sklearn.neighbors
import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

Объект `knn` включает в себя алгоритм, который будет использоваться для построения модели на обучающих данных, а также алгоритм, который сгенерирует прогнозы для новых точек данных. Он также будет содержать информацию, которую алгоритм извлек из обучающих данных. В случае с `KNeighborsClassifier` он будет просто хранить обучающий набор.

Для построения модели на обучающем наборе, мы вызываем метод `fit` объекта `knn`, который принимает в качестве аргументов массив `NumPy` `X_train`, содержащий обучающие данные, и массив `NumPy` `y_train`, соответствующий обучающим меткам:

```
In[26]: knn.fit(X_train, y_train)
Out[26]: KNeighborsClassifier(algorithm='auto',
leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=1, n_neighbors=1, p=2,
weights='uniform')
```

Метод `fit` возвращает сам объект `knn` (и изменяет его), таким образом, мы получаем строковое представление нашего классификатора. Оно показывает нам, какие параметры были использованы при создании модели. Почти

все параметры имеют значения по умолчанию, но вы также можете обнаружить параметр `n_neighbor=1`, заданный нами. Большинство моделей в `scikit-learn` имеют массу параметров, но большая часть из них связана с оптимизацией скорости вычислений или предназначена для особых случаев использования. Вам не нужно беспокоиться о других параметрах, приведенных здесь. Вывод модели в `scikit-learn` может быть очень длинным, но не нужно пугаться его.

Теперь мы можем получить прогнозы, применив эту модель к новым данным, по которым мы еще не знаем правильные метки. Представьте, что мы нашли в дикой природе ирис с длиной чашелистика 5 см, шириной чашелистика 2.9 см, длиной лепестка 1 см и шириной лепестка 0.2 см. К какому сорту ириса нужно отнести этот цветок? Мы можем поместить эти данные в массив NumPy, снова вычисляя форму массива, т.е. количество примеров (1), умноженное на количество признаков (4):

```
In[27]:
X_new = np.array([[5, 2.9, 1, 0.2]])
print("форма массива X_new: {}".format(X_new.shape))
Out[27]:
форма массива X_new: (1, 4)
```

Обратите внимание, что мы записали измерения по одному цветку в двумерный массив NumPy, поскольку `scikit-learn` работает с двумерными массивами данных.

Чтобы сделать прогноз, мы вызываем метод `predict` объекта `knn`:

```
In[28]:
prediction = knn.predict(X_new) print("Прогноз:
{}".format(prediction)) print("Спрогнозированная метка:
{}".format(iris_dataset['target_names'][prediction]))
Out[28]:
Прогноз: [0]
Спрогнозированная метка: ['setosa']
```

Наша модель предсказывает, что этот новый цветок ириса принадлежит к классу 0, что означает сорт `setosa`. Но как узнать, можем ли мы доверять нашей модели? Правильный сорт ириса для этого примера нам неизвестен, а ведь именно получение правильных прогнозов и является главной задачей построения модели!

Это тот самый момент, когда нам понадобится созданный ранее тестовый набор. Эти данные не использовались для построения модели, но мы знаем правильные сорта для каждого ириса в тестовом наборе.

Таким образом, мы можем сделать прогноз для каждого ириса в тестовом наборе и сравнить его с фактической меткой (уже известным сортом). Мы можем оценить качество модели, вычислив правильность (ассигасу) – процент цветов, для которых модель правильно спрогнозировала сорта:

```
In[29]: y_pred = knn.predict(X_test)
print("Прогнозы для тестового набора:\n
```

```
{})".format(y_pred))
```

Out[29]:

Прогнозы для тестового набора:

```
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0
1 1 0 2 1 0 2 2 1 0 2]
```

In[30]:

```
print("Правильность на тестовом наборе:
{:.2f}".format(np.mean(y_pred == y_test)))
```

Out[30]:

Правильность на тестовом наборе: 0.97

Кроме того, мы можем использовать метод `score` объекта `knn`, который вычисляет правильность модели для тестового набора:

```
In[31]: print("Правильность на тестовом наборе:
{:.2f}".format(knn.score(X_test, y_test)))
```

Out[31]:

Правильность на тестовом наборе: 0.97

Правильность этой модели для тестового набора составляет около 0.97, что означает, что мы дали правильный прогноз для 97% ирисов в тестовом наборе. При некоторых математических допущениях это означает, что мы можем ожидать, что наша модель в 97% случаев даст правильный прогноз для новых ирисов. Для нашего ботаника-любителя этот высокий уровень правильности означает, что наша модель может быть достаточно надежной в использовании. В следующих главах мы обсудим, как можно улучшить эффективность модели, и с какими подводными камнями можно столкнуться при настройке модели.

Давайте подытожим то, чему мы научились в этой главе. Мы начали с краткого введения в машинное обучение и сфер его применения, затем обсудили различие между обучением с учителем и обучением без учителя и дали краткий обзор инструментов, которые мы будем использовать в этой книге. Затем мы сформулировали задачу классификации ирисов на основе проведенных измерений характеристик цветов. Мы использовали набор данных, в котором эксперт уже предварительно классифицировал ирисы для построения модели, таким образом, мы решали задачу обучения с учителем. Было три возможных сорта ирисов – *setosa*, *versicolor* и *virginica*, что делало нашу задачу задачей 3-классовой классификации. В задаче классификации возможные сорта ирисов называются классами (*classes*) а сами названия сортов – метками (*labels*).

Набор данных *Iris* состоит из двух массивов NumPy: один содержит данные и в *scikit-learn* обозначается как *X*, другой содержит правильные или нужные ответы и обозначается как *y*. Массив *X* представляет собой двумерный массив признаков, в котором одна строка соответствует одной точке данных, а один столбец – одному признаку. Массив *y* представляет собой одномерный массив, который для каждого примера содержит метку класса, целое число от 0 до 2.

Мы разделили наш набор данных на обучающий набор (training set), чтобы построить нашу модель, а также тестовый набор (test set), чтобы оценить, насколько хорошо наша модель будет классифицировать новые, ранее неизвестные ей данные.

Мы выбрали алгоритм классификации k ближайших соседей, который генерирует прогноз для новой точки данных, рассматривая ее ближайшего соседа(ей) в обучающем наборе. Все это реализовано в классе KNeighborsClassifier, который содержит алгоритм, строящий модель, а также алгоритм, который дает прогнозы, используя построенную модель. Мы создали объект-экземпляр класса, задав параметры. Затем мы построили модель, вызвав метод fit и передав обучающие данные (X_train) и обучающие ответы (y_train) в качестве параметров. Мы оценили качество модели с использованием метода score, который вычисляет правильность модели. Мы применили метод score к тестовым данным и тестовым ответам и обнаружили, что наша модель демонстрирует правильность около 97%. Это означает, что модель выдает правильные прогнозы для 97% наблюдений тестового набора.

Это убедило нас в том, что модель можно применить к новым данным (в нашем примере это измерения характеристик новых цветов), и мы надеемся, что эта модель даст правильные прогнозы в 97% случаев.

Ниже приводится краткое изложение программного кода, необходимого для всей процедуры обучения и оценки модели:

```
In[32]:
X_train, X_test, y_train,
y_test = train_test_split(iris_dataset['data'],
iris_dataset['target'], random_state=0)
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Правильность на тестовом наборе:
{:.2f}".format(knn.score(X_test, y_test)))
```

```
Out[32]:
```

```
Правильность на тестовом наборе: 0.97
```

Этот фрагмент содержит базовый код, необходимый для применения любого алгоритма машинного обучения с помощью scikit-learn. Методы fit, predict и score являются общими для моделей контролируемого обучения в scikit-learn и, используя принципы, приведенные в этой главе, вы можете применить эти модели для решения различных задач машинного обучения. В следующей главе мы подробнее рассмотрим различные модели машинного обучения с учителем, имеющиеся в scikit-learn, и расскажем, как успешно применять их.

2 КЛАССИФИКАЦИЯ

Цель: Углубить и закрепить знания, полученные на лекциях по методам машинного обучения, и изучить возможности языка Python для классификации данных .

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Машинное обучение с учителем является одним из наиболее часто используемых и успешных видов машинного обучения. В этой главе мы более подробно расскажем о машинном обучении с учителем и объясним работу нескольких популярных алгоритмов. Мы уже разбирали применение машинного обучения с учителем в практике 1: классификацию ирисов по нескольким сортам с использованием измерений физических характеристик цветов.

Вспомним, что обучение с учителем используется всякий раз, когда мы хотим предсказать определенный результат (ответ) по данному объекту, и у нас есть пары объект-ответ. Мы строим модель машинного обучения на основе этих пар объект-ответ, которые составляют наш обучающий набор данных. Наша цель состоит в том, чтобы получить точные прогнозы для новых, никогда ранее не встречавшихся данных. Машинное обучение с учителем часто требует вмешательства человека, чтобы получить обучающий набор данных, но потом оно автоматизирует и часто ускоряет решение трудоемких или неосуществимых задач.

Есть два основные задачи машинного обучения с учителем: классификация (classification) и регрессия (regression).

Цель классификации состоит в том, чтобы спрогнозировать метку класса (class label), которая представляет собой выбор из заранее определенного списка возможных вариантов. В главе 1 мы использовали пример классификации ирисов, когда относили цветок к одному из трех возможных сортов. Классификация иногда разделяется на бинарную классификацию (binary classification), которая является частным случаем деления на два класса, и мультиклассовую классификацию (multiclass classification), когда в классификации участвует более двух классов. Бинарную классификацию можно представить как попытку ответить на поставленный вопрос в формате «да/нет». Классификация электронных писем на спам и не-спам является примером бинарной классификации. В данной задаче бинарной классификации ответ «да/нет» дается на вопрос «является ли это электронное письмо спамом?»

В бинарной классификации мы часто говорим о том, что один класс является положительным (positive) классом, а другой класс является отрицательным (negative) классом. При этом «положительный» означает здесь не наличие выгоды (ценности), а объект исследования. Таким образом, при поиске спама, положительным классом может быть класс «спам». Вопрос о том, какой из этих двух классов будет положительным, часто субъективен и зависит от предметной области исследования.

С другой стороны, пример классификации ирисов является примером

мультиклассовой классификации. Еще один пример – прогнозирование языка веб-сайта. Классами здесь будет заранее определённый список возможных языков.

Цель регрессии состоит в том, чтобы спрогнозировать непрерывное число или число с плавающей точкой (floating-point number), если использовать термины программирования, или вещественное число (real number), если говорить языком математических терминов. Прогнозирование годового дохода человека в зависимости от его образования, возраста и места жительства является примером регрессионной задачи. Прогнозируемое значение дохода представляет собой сумму (amount) и может быть любым числом в заданном диапазоне. Другой пример регрессионной задачи – прогнозирование объема урожая зерна на ферме в зависимости от таких атрибутов, как объем предыдущего урожая, погода, и количество сотрудников, работающих на ферме. И снова объем урожая может быть любым числом.

Самый простой способ отличить классификацию от регрессии – спросить, заложена ли в полученном ответе определённая непрерывность (преемственность). Если полученные результаты непрерывно связаны друг с другом, то решаемая задача является задачей регрессии. Возьмём прогнозирование годового дохода. Здесь ясно видна непрерывность ответа. Разница между годовым доходом в 40000\$ или 40001\$ не существенна, хотя речь идёт о разных денежных суммах. Если наш алгоритм предсказывает 39999\$ или 40001\$, в то время как он должен предсказать 40000\$ (реальное значение годового дохода), мы не будем настаивать на том, что разница существенна. Наоборот, в задаче распознавания языка веб-сайта (задаче классификации) ответы чётко определены. Контент сайта может быть написан либо на одном конкретном языке, либо на другом. Между языками нет непрерывной связи, не существует языка, находящегося между английским и французским.

В машинном обучении с учителем нам нужно построить модель на обучающих данных, а затем получить точные прогнозы для новых, еще не встречавшихся нам данных, которые имеют те же самые характеристики, что и использованный нами обучающий набор. Если модель может выдавать точные прогнозы на ранее не встречавшихся данных, мы говорим, что модель обладает способностью обобщать (generalize) результат на тестовые данные. Нам необходимо построить модель, которая будет обладать максимальной обобщающей способностью.

Обычно мы строим модель таким образом, чтобы она давала точные прогнозы на обучающем наборе. Если обучающий и тестовый наборы имеют много общего между собой, можно ожидать, что модель будет точной и на тестовом наборе. Однако в некоторых случаях этого не происходит. Например, если мы строим очень сложные модели, необходимо помнить, что на обучающей выборке можно получить произвольную правильность.

Давайте взглянем на выдуманный пример, чтобы проиллюстрировать этот тезис. Скажем, начинающий специалист по анализу данных хочет спрогнозировать покупку клиентом лодки на основе записей о клиентах, которые

ранее приобрели лодку, и клиентах, которые не заинтересованы в покупке лодки. Цель состоит в том, чтобы отправить рекламные письма клиентам, которые, вероятно, хотят совершить покупку, и не беспокоить клиентов, не заинтересованных в покупке.

Предположим, у нас есть записи о клиентах, приведённые в таблице 2.1.

Таблица 2.1 Пример данных о клиентах

Возраст	Количество автомобилей в собственности	Есть собственный дом	Количество детей	Семейное положение	Есть собака	Купил лодку
66	1	да	2	вдовец	нет	да
52	2	да	3	женат	нет	да
22	0	нет	0	женат	да	нет
25	1	нет	1	холост	нет	нет
44	0	нет	2	разведен	да	нет
39	1	да	2	женат	да	нет
26	1	нет	2	холост	нет	нет
40	3	да	1	женат	да	нет
53	2	да	2	разведен	нет	да
64	2	да	3	разведен	нет	нет
58	2	да	2	женат	да	да
33	1	нет	1	холост	нет	нет

Поработав с данными некоторое время, наш начинающий специалист формулирует следующее правило «если клиент старше 45 лет, у него менее трех детей, либо у него трое и он женат, то он скорее всего купит лодку». Если спросить, насколько хорошо работает это правило, наш специалист воскликнет «оно дает 100%-ную правильность!» И в самом деле, для данных, приведенных в таблице, это правило демонстрирует идеальную правильность. Мы могли бы сформулировать массу правил, объясняющих покупку лодки. Значения возраста появляются в наборе данных лишь один раз, таким образом, мы могли бы сказать, что люди в возрасте 66, 52, 53 и 58 лет хотят купить лодку, тогда как все остальные не собираются ее покупать. Несмотря на то что можно сформулировать множество правил, которые хорошо работают для этих наблюдений, следует помнить о том, что нам не интересны прогнозы для этого набора данных, мы уже знаем ответы для этих клиентов. Мы хотим знать, могут ли новые клиенты купить лодку. Поэтому нам нужно правило, которое будет хорошо работать для новых клиентов, и достижение 100%-ной правильности на обучающей выборке не поможет нам в этом. Нельзя ожидать, что правило, сформулированное нашим специалистом, будет так же хорошо работать и для новых клиентов. Похоже, с этим у нас сложность, ведь у нас мало данных. Например, часть правила «либо трое и он

женат» сформулировано по одному клиенту.

Единственный показатель качества работы алгоритма на новых данных – это использование тестового набора. Однако интуитивно мы ожидаем, что простые модели должны лучше обобщать результат на новые данные. Если бы правило звучало «люди старше 50 лет хотят купить лодку» и оно объясняло бы поведение всех клиентов, мы доверяли бы ему больше, чем правилу, которое помимо возраста включало бы количество детей и семейное положение. Поэтому нам всегда нужно искать самую простую модель. Построение модели, которая слишком сложна для имеющегося у нас объема информации (что и сделал наш начинающий специалист по анализу данных), называется переобучением (*overfitting*). Переобучение происходит, когда ваша модель слишком точно подстраивается под особенности обучающего набора и вы получаете модель, которая хорошо работает на обучающем наборе, но не умеет обобщать результат на новые данные. С другой стороны, если ваша модель слишком проста, скажем, вы сформулировали правило «все, у кого есть собственный дом, покупает лодку», вы, возможно, не смогли охватить все многообразие и изменчивость данных, и ваша модель будет плохо работать даже на обучающем наборе. Выбор слишком простой модели называется недополучением (*underfitting*).

Чем сложнее модель, тем лучше она будет работать на обучающих данных. Однако, если наша модель становится слишком сложной, мы начинаем уделять слишком много внимания каждой отдельной точке данных в нашем обучающем наборе, и эта модель не будет хорошо обобщать результат на новые данные.

Существует оптимальная точка, которая позволяет получить наилучшую обобщающую способность. Собственно, это и есть модель, которую нам нужно найти.

Компромисс между переобучением и недообучением показан на рис. 2.1.

Важно отметить, что сложность модели тесно связана с изменчивостью входных данных, содержащихся в вашем обучающем наборе: чем больше разнообразие точек данных в вашем наборе, тем более сложную модель можно использовать, не беспокоясь о переобучении. Обычно больший объем данных дает большее разнообразие, таким образом, большие наборы данных позволяют строить более сложные модели. Однако простое дублирование одних и тех же точек данных или сбор очень похожих данных здесь не поможет.

Возвращаясь к продажам лодок, можно сказать, что если бы у нас было более 10000 строк данных о клиентах и все они подчинялись бы правилу «если клиент старше 45 лет, у него менее трех детей, либо трое и он женат, то он скорее всего купит лодку», мы бы с гораздо большей вероятностью поверили в это правило, чем если бы оно было сформулировано лишь по 12 строкам таблицы 2.1.

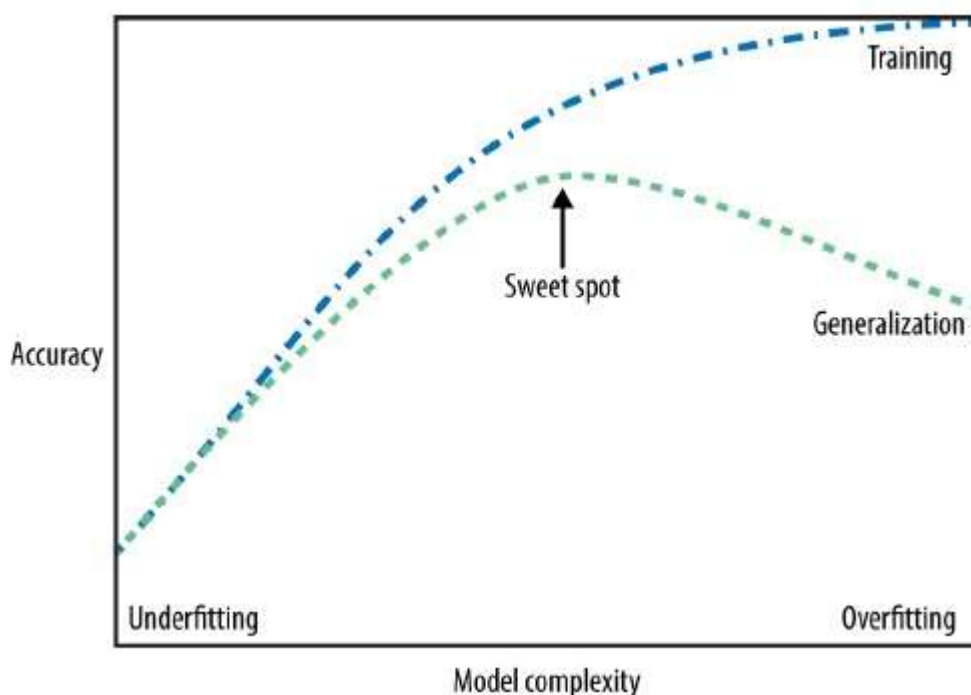


Рис. 2.1 - Компромисс между сложностью модели и правильностью на обучающей и тестовой выборках

Увеличение объема данных и построение более сложных моделей часто творят чудеса при решении задач машинного обучения с учителем. В этой книге мы сосредоточимся на работе с данными фиксированного размера. В действительности вы, как правило, сами можете определить объем собираемых данных, и это может оказаться более полезным, чем корректировка и настройка вашей модели. Никогда не стоит недооценивать преимущества увеличения объема данных.

Теперь мы рассмотрим наиболее популярные алгоритмы машинного обучения и объясним, как они обучаются на основе данных и как вычисляют прогнозы. Кроме того, мы расскажем о том, как принцип сложности реализуется для каждой из этих моделей, и покажем, как тот или иной алгоритм строит модель. Мы рассмотрим преимущества и недостатки каждого алгоритма, а также расскажем о том, применительно к каким данным лучше всего использовать тот или иной алгоритм. Мы также объясним значение наиболее важных параметров и опций. Многие алгоритмы имеют опции классификации и регрессии, поэтому мы опишем обе опции.

Необязательно детально вчитываться в описание каждого алгоритма, но понимание модели даст вам лучшее представление о различных способах работы алгоритмов машинного обучения. Кроме того, эту главу можно использовать в качестве справочного руководства, и вы можете вернуться к ней, если не знаете, как работает тот или иной алгоритм.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Для иллюстрации различных алгоритмов мы будем использовать не-

сколько наборов данных. Некоторые наборы данных будут небольшими и синтетическими (то есть вымышленными), призванными подчеркнуть отдельные аспекты алгоритмов. Другие наборы данных будут большими, реальными примерами.

Примером синтетического набора данных для двухклассовой классификации является набор данных `forge`, который содержит два признака. Программный код, приведённый ниже, создаёт диаграмму рассеяния (рис. 2.2), визуализируя все точки данных в этом наборе. На графике первый признак отложен на оси *x*, а второй – по оси *y*. Как это всегда бывает в диаграммах рассеяния, каждая точка данных представлена в виде одного маркера. Цвет и форма маркера указывает на класс, к которому принадлежит точка:

```
In[2]:  
# генерируем набор данных  
X, y = mglearn.datasets.make_forge()  
# строим график для набора данных  
%matplotlib inline  
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)  
plt.legend(["Класс 0", "Класс 1"], loc=4)  
plt.xlabel("Первый признак")  
plt.ylabel("Второй признак")  
print("форма массива X: {}".format(X.shape))  
Out[2]: форма массива X: (26, 2)
```

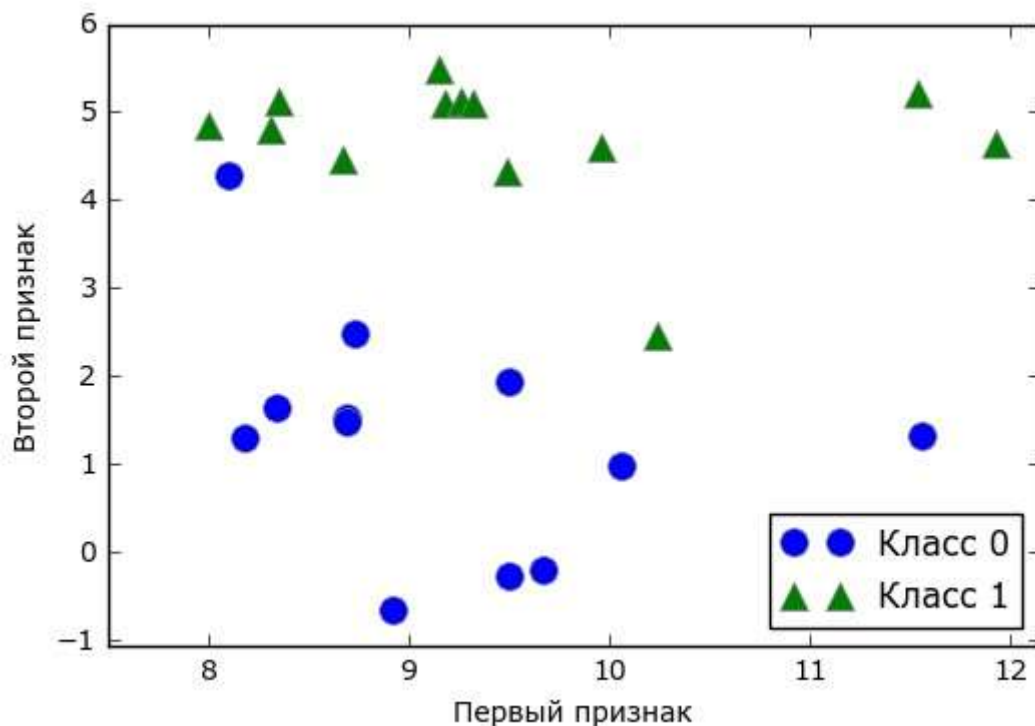


Рис. 2.2 - Диаграмма рассеяния для набора данных `forge`

Как видно из сводки по массиву `X`, этот набор состоит из 26 точек данных и 2 признаков. Для иллюстрации алгоритмов регрессии воспользуемся синтетическим набором `wave`. Набор данных имеет единственный входной

признак и непрерывную целевую переменную или отклик (response), который мы хотим смоделировать. На рисунке, построенном здесь (рис. 2.3), по оси x располагается единственный признак, а по оси y – целевая переменная (ответ).

In[3]:

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o') plt.ylim(-3, 3)
plt.xlabel("Признак")
plt.ylabel("Целевая переменная")
```

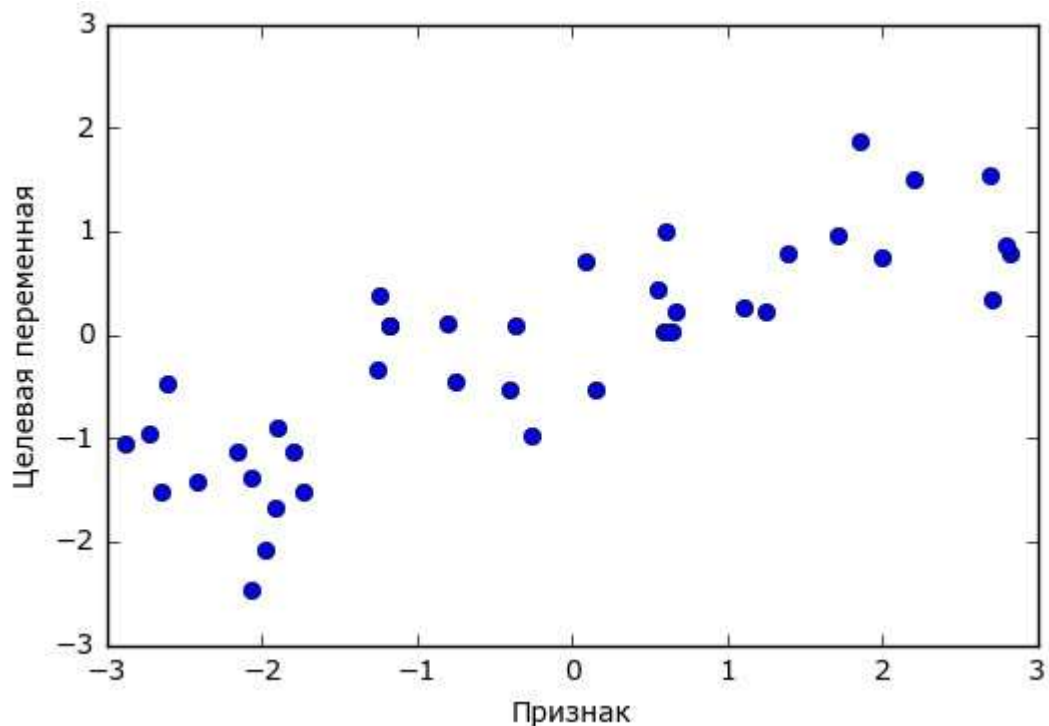


Рис. 2.3 - График для набора данных wave, по оси x отложен признак, по оси y – целевая переменная

Мы используем эти очень простые, низкоразмерные наборы данных, потому что их легко визуализировать – печатная страница имеет два измерения, и данные, которые содержат более двух признаков, графически представить трудно. Вывод, полученный для набора с небольшим числом признаков или низкоразмерном (low-dimensional) наборе, возможно, не подтвердится для набора данных с большим количеством признаков или высокоразмерного (high-dimensional) набора. Если вы помните об этом, проверка алгоритма на низкоразмерном наборе данных может оказаться очень полезной.

Мы дополним эти небольшие синтетические наборы данных двумя реальными наборами, которые включены в scikit-learn. Один из них – набор данных по раку молочной железы Университета Висконсин (cancer для краткости), в котором записаны клинические измерения опухолей молочной железы. Каждая опухоль обозначается как «benign» («доброкачественная», для неагрессивных опухолей) или malignant («злокачественная», для раковых

опухолей), и задача состоит в том, чтобы на основании измерений ткани дать прогноз, является ли опухоль злокачественной.

Данные можно загрузить из scikit-learn с помощью функции `load_breast_cancer`:

```
In[4]: from sklearn.datasets import
load_breast_cancer cancer = load_breast_cancer()
print("Ключи cancer(): \n{}".format(cancer.keys()))
Out[4]: Ключи cancer():
dict_keys(['feature_names', 'data', 'DESCR', 'target', 'target_names'])
```

Набор данных включает 569 точек данных и 30 признаков.

```
In[5]:
print("Форма массива data для набора cancer:
{}".format(cancer.data.shape))
Out[5]:
Форма массива data для набора cancer: (569, 30)
```

Из 569 точек данных 212 помечены как злокачественные, а 357 как доброкачественные.

```
In[6]:
print("Количество примеров для каждого
класса:\n{}".format(
    {n: v for n, v in zip(cancer.target_names,
np.bincount(cancer.target))}))
Out[6]:
Количество примеров для каждого класса:
{'benign': 357, 'malignant': 212}
```

Чтобы получить содержательное описание каждого признака, взглянем на атрибут `feature_names`:

```
In[7]: print("Имена
признаков:\n{}".format(cancer.feature_names))
Out[7]:
Имена признаков:
['mean radius' 'mean texture' 'mean perimeter'
'mean area'
'mean smoothness' 'mean compactness' 'mean concavity'
'mean concave points' 'mean symmetry' 'mean fractal dimension'
'radius error' 'texture error' 'perimeter error'
'area error'
'smoothness error' 'compactness error' 'concavity error'
'concave points error' 'symmetry error' 'fractal dimension error'
'worst radius' 'worst texture' 'worst perimeter']
```

```
'worst area'
'worst smoothness' 'worst compactness' 'worst con-
cavity'
'worst concave points' 'worst symmetry' 'worst
fractal dimension']
```

Если вам интересно, то более подробную информацию о данных можно получить, прочитав `cancer.DESCR`.

Кроме того, для задач регрессии мы будем использовать реальный набор данных – набор данных Boston Housing. Задача, связанная с этим набором данных, заключается в том, чтобы спрогнозировать медианную стоимость домов в нескольких районах Бостона в 1970-е годы на основе такой информации, как уровень преступности, близость к Charles River, удаленность от радиальных магистралей и т.д. Набор данных содержит 506 точек данных и 13 признаков:

```
In[8]: from sklearn.datasets import load_boston boston
= load_boston()
print("форма массива data для набора boston:
{}".format(boston.data.shape)) Out[8]: форма массива
data для набора boston: (506, 13)
```

Опять же, вы можете получить более подробную информацию о наборе данных, прочитав атрибут `boston.DESCR`. В данном случае мы более детально проанализируем набор данных, учитывая не только 13 измерений в качестве входных признаков, но и приняв во внимание все взаимодействия (interactions) между признаками. Иными словами, мы будем учитывать в качестве признаков не только уровень преступности и удаленность от радиальных магистралей по отдельности, но и взаимодействие уровень преступности–удаленность от радиальных магистралей. Включение производных признаков называется конструированием признаков (feature engineering), которое мы рассмотрим более подробно в главе 4. Набор данных с производными признаками можно загрузить с помощью функции `load_extended_boston`:

```
In[9]:
X, y = mglearn.datasets.load_extended_boston()
print("форма массива X: {}".format(X.shape))
Out[9]: форма массива X: (506, 104)
```

Полученные 104 признака – 13 исходных признаков плюс 91 производный признак.

Мы будем использовать эти наборы данных, чтобы объяснить и проиллюстрировать свойства различных алгоритмов машинного обучения. Однако сейчас давайте перейдем к самим алгоритмам. Впервые, мы вернемся к алгоритму k ближайших соседей, который рассматривали в предыдущей практике.

Алгоритм k ближайших соседей, возможно, является самым простым алгоритмом машинного обучения. Построение модели заключается в запо-

минании обучающего набора данных. Для того, чтобы сделать прогноз для новой точки данных, алгоритм находит ближайшие к ней точки обучающего набора, то есть находит «ближайших соседей».

В простейшем варианте алгоритм k ближайших соседей рассматривает лишь одного ближайшего соседа – точку обучающего набора, ближе всего расположенную к точке, для которой мы хотим получить прогноз. Прогнозом является ответ, уже известный для данной точки обучающего набора. На рис. 2.4 показано решение задачи классификации для набора данных *forge*:

```
In[10]:  
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

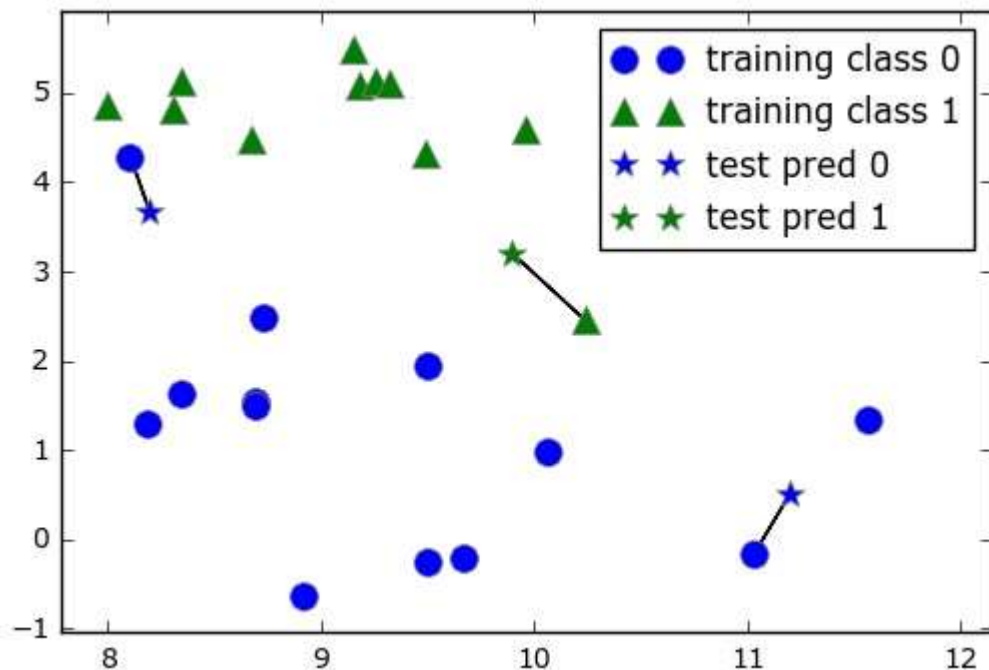


Рис. 2.4 - Прогнозы, полученные для набора данных *forge* с помощью модели одного ближайшего соседа

Здесь мы добавили три новые точки данных, показанные в виде звездочек. Для каждой мы отметили ближайшую точку обучающего набора. Прогноз, который дает алгоритм одного ближайшего соседа – метка этой точки (показана цветом маркера).

Вместо того, чтобы учитывать лишь одного ближайшего соседа, мы можем рассмотреть произвольное количество (k) соседей. Отсюда и происходит название алгоритма k ближайших соседей. Когда мы рассматриваем более одного соседа, для присвоения метки используется голосование (voting). Это означает, что для каждой точки тестового набора мы подсчитываем количество соседей, относящихся к классу 0, и количество соседей, относящихся к классу 1. Затем мы присваиваем точке тестового набора наиболее часто встречающийся класс: другими словами, мы выбираем класс, набравший большинство среди k ближайших соседей. В примере, приведенном ниже (рис. 2.5), используются три ближайших соседа:

```
In[11]: mglearn.plots.plot_
```

```
knn_classification(n_neighbors=3)
```

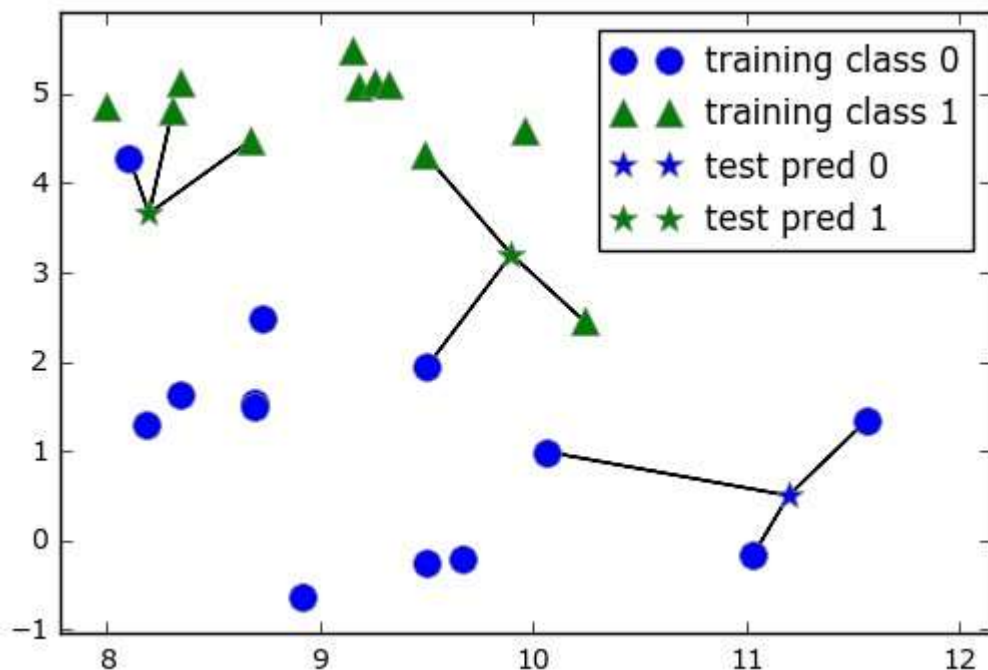


Рис. 2.5 - Прогнозы, полученные для набора данных `forge` с помощью модели трех ближайших соседей

И снова прогнозы переданы цветом маркера. Видно, что прогноз для новой точки данных в верхнем левом углу отличается от прогноза, полученного при использовании одного ближайшего соседа.

Хотя данный рисунок иллюстрирует задачу бинарной классификации, этот метод можно применить к наборам данных с любым количеством классов. В случае мультиклассовой классификации мы подсчитываем количество соседей, принадлежащих к каждому классу, и снова прогнозируем наиболее часто встречающийся класс.

Теперь давайте посмотрим, как можно применить алгоритм `k` ближайших соседей, используя `scikit-learn`. Во-первых, мы разделим наши данные на обучающий и тестовый наборы, чтобы оценить обобщающую способность модели, рассмотренную в практике 1:

```
In[12]: from sklearn.model_selection import
train_test_split
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test =
train_test_split(X, y, random_state=0)
```

Далее выполняем импорт и создаем объект-экземпляр класса, задавая параметры, например, количество соседей, которое будем использовать для классификации. В данном случае мы устанавливаем его равным 3:

```
In[13]: from sklearn.neighbors
```

```
import KNeighborsClassifier
```

```
clf = KNeighborsClassifier(n_neighbors=3)
```

Затем подгоняем классификатор, используя обучающий набор. Для KNeighborsClassifier это означает запоминание набора данных, таким образом, мы можем вычислить соседей в ходе прогнозирования:

```
In[14]: clf.fit(X_train, y_train)
```

Чтобы получить прогнозы для тестовых данных, мы вызываем метод predict. Для каждой точки тестового набора он вычисляет ее ближайших соседей в обучающем наборе и находит среди них наиболее часто встречающийся класс:

```
In[15]:
```

```
print("Прогнозы на тестовом наборе:  
{ }".format(clf.predict(X_test)))
```

```
Out[15]:
```

```
Прогнозы на тестовом наборе: [1 0 1 0 1 0 0]
```

Для оценки обобщающей способности модели мы вызываем метод score с тестовыми данными и тестовыми метками:

```
In[16]: print("Правильность на тестовом наборе:  
{:.2f}".format(clf.score(X_test, y_test)))
```

```
Out[16]:
```

```
Правильность на тестовом наборе: 0.86
```

Мы видим, что наша модель имеет правильность 86%, то есть модель правильно предсказала класс для 86% примеров тестового набора.

Кроме того, для двумерных массивов данных мы можем показать прогнозы для всех возможных точек тестового набора, разместив в плоскости ху. Мы зададим цвет плоскости в соответствии с тем классом, который будет присвоен точке в этой области. Это позволит нам сформировать границу принятия решений (decision boundary), которая разбивает плоскость на две области: область, где алгоритм присваивает класс 0, и область, где алгоритм присваивает класс 1.

Программный код, приведенный ниже, визуализирует границы принятия решений для одного, трех и девяти соседей (показаны на рис. 2.6):

```
In[17]: fig, axes = plt.subplots(1, 3, figsize=(10, 3))  
for n_neighbors, ax in zip([1, 3, 9], axes):  
# создаем объект-классификатор и подгоняем в одной  
строке  
clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)  
mglearn.plots.plot_2d_separator(clf, X, fill=True,  
eps=0.5, ax=ax, alpha=.4)  
mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)  
ax.set_title("количество  
соседей: {}".format(n_neighbors))
```



Рис. 2.6 - Границы принятия решений, созданные моделью ближайших соседей для различных значений `n_neighbors`

На рисунке слева можно увидеть, что использование модели одного ближайшего соседа дает границу принятия решений, которая очень хорошо согласуется с обучающими данными. Увеличение числа соседей приводит к сглаживанию границы принятия решений. Более гладкая граница соответствует более простой модели. Другими словами, использование нескольких соседей соответствует высокой сложности модели (как показано в правой части рис. 2.1), а использование большого количества соседей соответствует низкой сложности модели (как показано в левой части рис. 2.1). Если взять крайний случай, когда количество соседей будет равно количеству точек данных обучающего набора, каждая точка тестового набора будет иметь одних и тех же соседей (соседями будет все точки обучающего набора) и все прогнозы будут одинаковыми: будет выбран класс, который является наиболее часто встречающимся в обучающем наборе.

Давайте выясним, существует ли взаимосвязь между сложностью модели и обобщающей способностью, о которой мы говорили ранее. Мы сделаем это с помощью реального набора данных Breast Cancer. Начнем с того, что разобьем данные на обучающий и тестовый наборы. Затем мы оценим качество работы модели на обучающем и тестовом наборах с использованием разного количества соседей. Результаты показаны на рис. 2.7:

```
In[18]: from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target,
    random_state=66)
training_accuracy = [] test_accuracy = []
# пробуем n_neighbors от 1 до 10
neighbors_settings = range(1, 11)
for n_neighbors in neighbors_settings:
    # строим модель
```

```

clf = KNeighborsClassifier(n_neighbors=n_neighbors)
clf.fit(X_train, y_train)
    записываем правильность на обучающем наборе
training_accuracy.append(clf.score(X_train, y_train))
# записываем правильность на тестовом наборе
test_accuracy.append(clf.score(X_test, y_test))
plt.plot(neighbors_settings, training_accuracy,
label="правильность на обучающем наборе")
plt.plot(neighbors_settings, test_accuracy,
label="правильность на тестовом наборе")
plt.ylabel("Правильность")
plt.xlabel("количество соседей") plt.legend()

```

На этом графике по оси y отложена правильность на обучающем наборе и правильность на тестовом наборе, а по оси x – количество соседей. В реальности подобные графики редко бывают гладкими, мы попрежнему можем увидеть некоторые признаки переобучения и недообучения (обратите внимание, что поскольку использование небольшого количества соседей соответствует более сложной модели, график представляет собой изображение рис. 2.1, зеркально отраженное по горизонтали). При использовании модели одного ближайшего соседа правильность на обучающем наборе идеальна. Однако при использовании большего количества соседей модель становится все проще и правильность на обучающем наборе падает. Правильность на тестовом наборе в случае использования одного соседа ниже, чем при использовании нескольких соседей. Это указывает на то, что использование одного ближайшего соседа приводит к построению слишком сложной модели. С другой стороны, когда используются 10 соседей, модель становится слишком простой и она работает еще хуже. Оптимальное качество работы модели наблюдается где-то посередине, когда используются шесть соседей. Однако посмотрим на шкалу y . Худшая по качеству модель дает правильность на тестовом наборе около 88%, что по-прежнему может быть приемлемым результатом.

Существует также регрессионный вариант алгоритма k ближайших соседей. Опять же, давайте начнем с рассмотрения одного ближайшего соседа, на этот раз воспользуемся набором данных `wave`. Мы добавили три точки тестового набора в виде зеленых звездочек по оси x . Прогноз с использованием одного соседа – это целевое значение ближайшего соседа. На рис. 2.8 прогнозы показаны в виде синих звездочек:

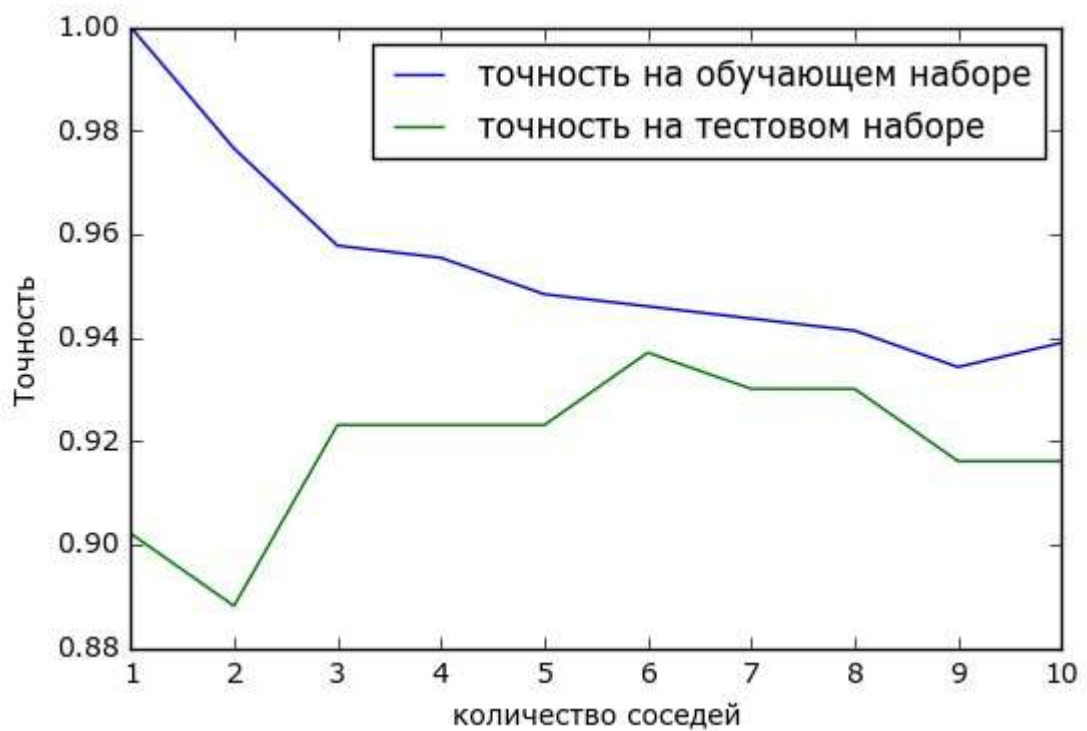


Рис. 2.7 - Сравнение правильности на обучающем и тестовом наборах как функции от количества соседей

In[19]:

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

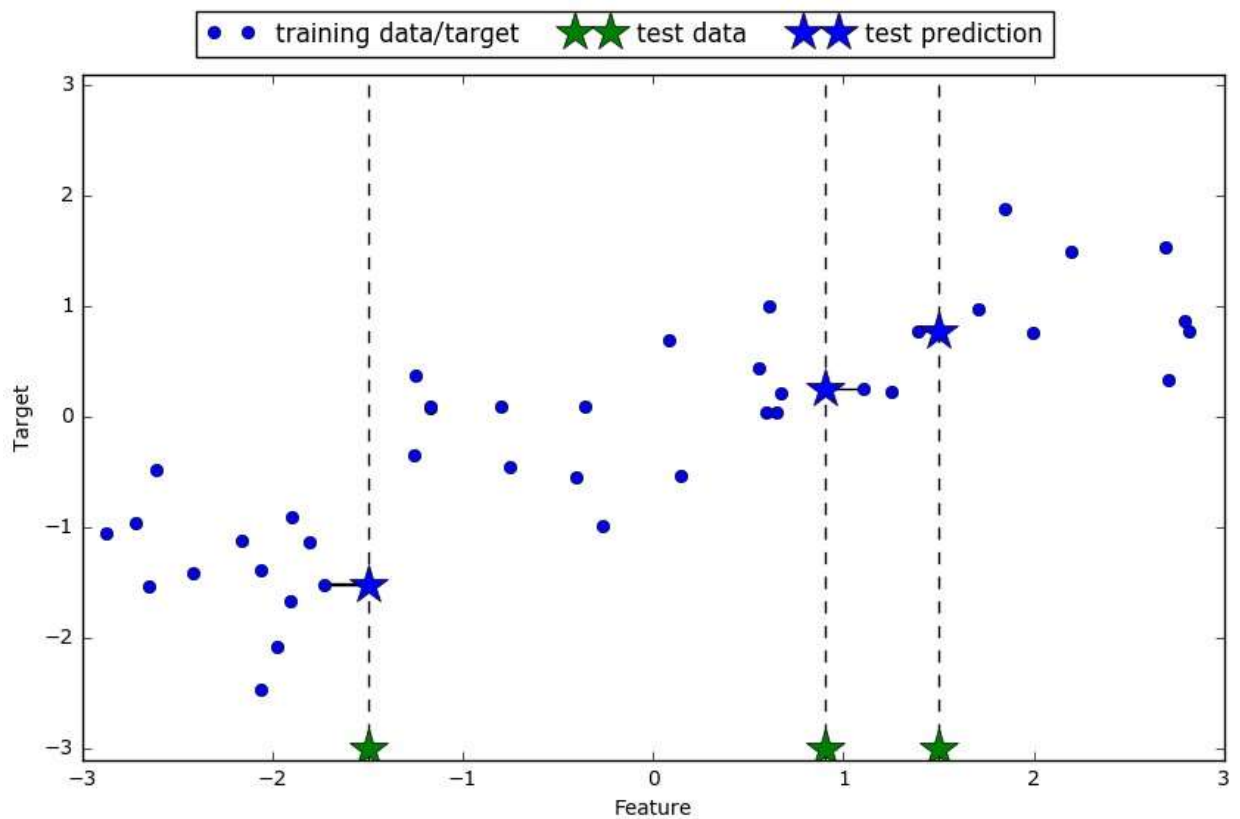


Рис. 2.8 - Прогнозы, полученные с помощью регрессионной модели одного ближайшего соседа для набора данных wave

И снова для регрессии мы можем использовать большее количество ближайших соседей. При использовании нескольких ближайших соседей прогнозом становится среднее значение соответствующих соседей (рис. 2.9):

```
In[20]:
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

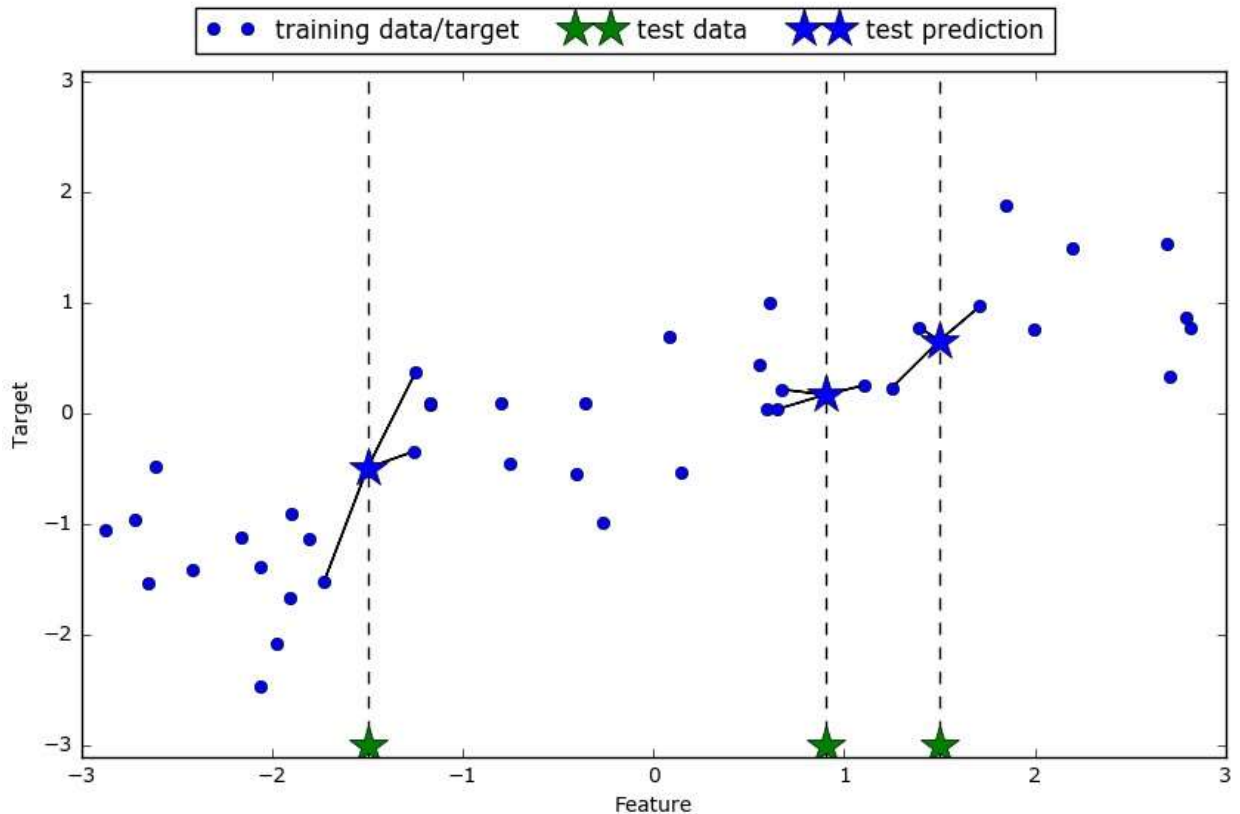


Рис. 2.8 - Прогнозы, полученные с помощью регрессионной модели трех ближайших соседей для набора данных wave

Алгоритм регрессии k ближайших соседей реализован в классе `KNeighborsRegressor`. Он используется точно так же, как `KNeighborsClassifier`:

```
In[21]: from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
# разбиваем набор данных wave на обучающую и
# тестовую выборки
X_train, X_test, y_train,
y_test = train_test_split(X, y, random_state=0)
# создаем экземпляр модели и устанавливаем количество
# соседей равным 3
reg = KNeighborsRegressor(n_neighbors=3)
# подгоняем модель с использованием обучающих данных и
# обучающих ответов
reg.fit(X_train, y_train)
```

А теперь получим прогнозы для тестового набора.

```
In[22]:
print("Прогнозы для тестового
набора:\n{}".format(reg.predict(X_test)))
Out[22]:
Прогнозы для тестового набора:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912
-0.447 -1.139]
```

Кроме того, мы можем оценить качество модели с помощью метода `score`, который для регрессионных моделей возвращает значение R^2 . R^2 , также известный как коэффициент детерминации, является показателем качества регрессионной модели и принимает значения от 0 до 1. Значение 1 соответствует идеальной прогнозирующей способности, а значение 0 соответствует константе модели, которая лишь предсказывает среднее значение ответов в обучающем наборе, `y_train`:

```
In[23]: print("R^2 на тестовом наборе:
{: .2f}".format(reg.score(X_test, y_test)))
Out[23]:
R^2 на тестовом наборе: 0.83
```

В данном случае значение R^2 составляет 0.83, что указывает на относительно хорошее качество подгонки модели.

Применительно к нашему одномерному массиву данных мы можем увидеть прогнозы для всех возможных значений признаков (рис. 2.10). Для этого мы создаём тестовый набор данных и визуализируем полученные линии прогнозов:

```
In[24]: fig, axes = plt.subplots(1, 3, figsize=(15,
4))
# создаем 1000 точек данных, равномерно
распределенных между -3 и 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    # получаем прогнозы, используя 1, 3, и 9 соседей
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mglern.cm2(0),
    markersize=8)
    ax.plot(X_test, y_test, 'v', c=mglern.cm2(1),
    markersize=8)
    ax.set_title("{} neighbor(s)\n train score: {:.2f}
test score: {:.2f}".format(n_neighbors,
reg.score(X_train, y_train),
```

```

reg.score(X_test, y_test)))
ax.set_xlabel("Признак")
ax.set_ylabel("Целевая переменная")
axes[0].legend(["Прогнозы модели",
"Обучающие данные/ответы",
"Тестовые данные/ответы"], loc="best")

```

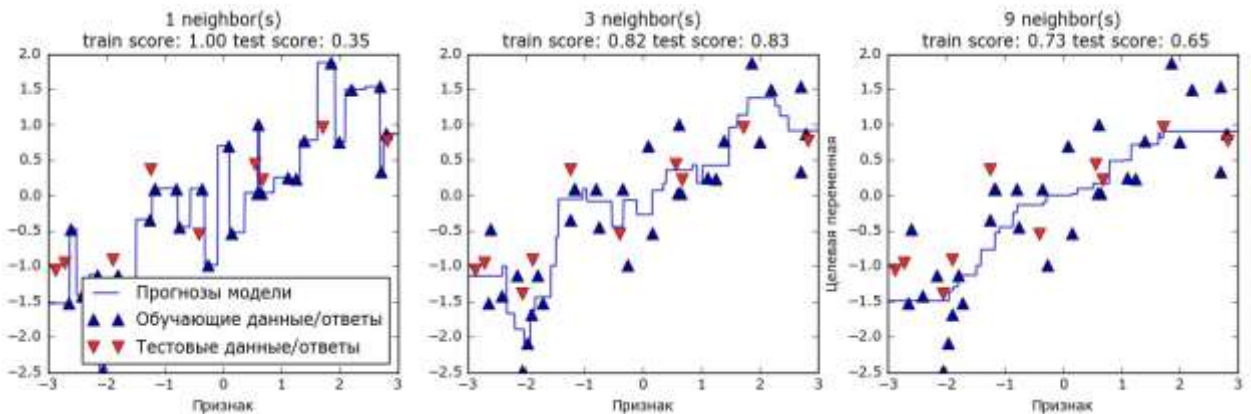


Рис. 2.10 - Сравнение прогнозов, полученных с помощью регрессии ближайших соседей для различных значений `n_neighbors`

Как видно на графике, при использовании лишь одного соседа каждая точка обучающего набора имеет очевидное влияние на прогнозы, и предсказанные значения проходят через все точки данных. Это приводит к очень неустойчивым прогнозам. Увеличение числа соседей приводит к получению более сглаженных прогнозов, но при этом снижается правильность подгонки к обучающим данным.

В принципе, в классификаторе `KNeighbors` есть два важных параметра: количество соседей и мера расстояния между точками данных. На практике использование небольшого числа соседей (например, 3-5) часто работает хорошо, но вы, конечно, можете самостоятельно настроить этот параметр. Вопрос, связанный с выбором правильной меры расстояния, выходит за рамки этой книги. По умолчанию используется евклидово расстояние, которое хорошо работает во многих ситуациях.

Одним из преимуществ метода ближайших соседей является то, что эту модель очень легко интерпретировать и, как правило, этот метод дает приемлемое качество без необходимости использования большого количества настроек. Он является хорошим базовым алгоритмом, который нужно попробовать в первую очередь, прежде чем рассматривать более сложные методы. Как правило, построение модели ближайших соседей происходит очень быстро, но, когда ваш обучающий набор очень большой (с точки зрения количества характеристик или количества наблюдений) получение прогнозов может занять некоторое время. При использовании алгоритма ближайших соседей важно выполнить предварительную обработку данных (смотрите главу 3). Данный метод не так хорошо работает, когда речь идет о наборах

данных с большим количеством признаков (сотни и более), и особенно плохо работает в ситуации, когда подавляющее число признаков в большей части наблюдений имеют нулевые значения (так называемые разреженные наборы данных или *sparse datasets*).

Таким образом, несмотря на то что алгоритм ближайших соседей легко интерпретировать, на практике он не часто используется из-за скорости вычислений и его неспособности обрабатывать большое количество признаков. Метод, который мы обсудим ниже, лишён этих недостатков.

3 ЛИНЕЙНАЯ РЕГРЕССИЯ

Цель: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для обработки данных при помощи линейной регрессии.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Линейные модели представляют собой класс моделей, которые широко используются на практике и были предметом детального изучения в течение последних нескольких десятилетий, а их история насчитывает более ста лет. Линейные модели дают прогноз, используя линейную функцию (linear function) входных признаков.

Для регрессии общая прогнозная формула линейной модели выглядит следующим образом:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Здесь $x[0]$ по $x[p]$ обозначают признаки (в данном примере число характеристик равно $p+1$) для отдельной точки данных, w и b – параметры модели, оцениваемые в ходе обучения, и $\hat{y}$ – прогноз, выдаваемый моделью. Для набора данных с одним признаком эта формула имеет вид:

$$\hat{y} = w[0] * x[0] + b$$

Возможно из курса математики вы вспомните, что эта формула – уравнение прямой. Здесь $x[0]$ является наклоном, а b – сдвигом по оси y . Когда используется несколько признаков, регрессионное уравнение содержит параметры наклона для каждого признака. Как вариант, прогнозируемый ответ можно представить в виде взвешенной суммы входных признаков, где веса (которые могут быть отрицательными) задаются элементами w .

ПРАКТИЧЕСКАЯ ЧАСТЬ

Попробуем вычислить параметры $w[0]$ и b для нашего одномерного набора данных `wave` с помощью следующей строки программного кода (см. рис. 2.11):

```
In[25]:  
mglearn.plots.plot_linear_regression_wave()  
Out[25]:  
w[0]: 0.393906 b: -0.031804
```

Существует различные виды линейных моделей для регрессии. Различие между этими моделями заключается в способе оценивания параметров модели w и b по обучающим данным и контроле сложности модели. Теперь мы рассмотрим наиболее популярные линейные модели для регрессии.

Линейная регрессия или обычный метод наименьших квадратов (ordinary least squares, OLS) – это самый простой и наиболее традиционный метод регрессии. Линейная регрессия находит параметры

w и b , которые минимизируют среднеквадратическую ошибку (mean squared error) между спрогнозированными и фактическими ответами y в обучающем наборе. Среднеквадратичная ошибка равна сумме квадратов разностей между спрогнозированными и фактическими значениями. Линейная регрессия проста, что является преимуществом, но в то же время у неё нет инструментов, позволяющих контролировать сложность модели.

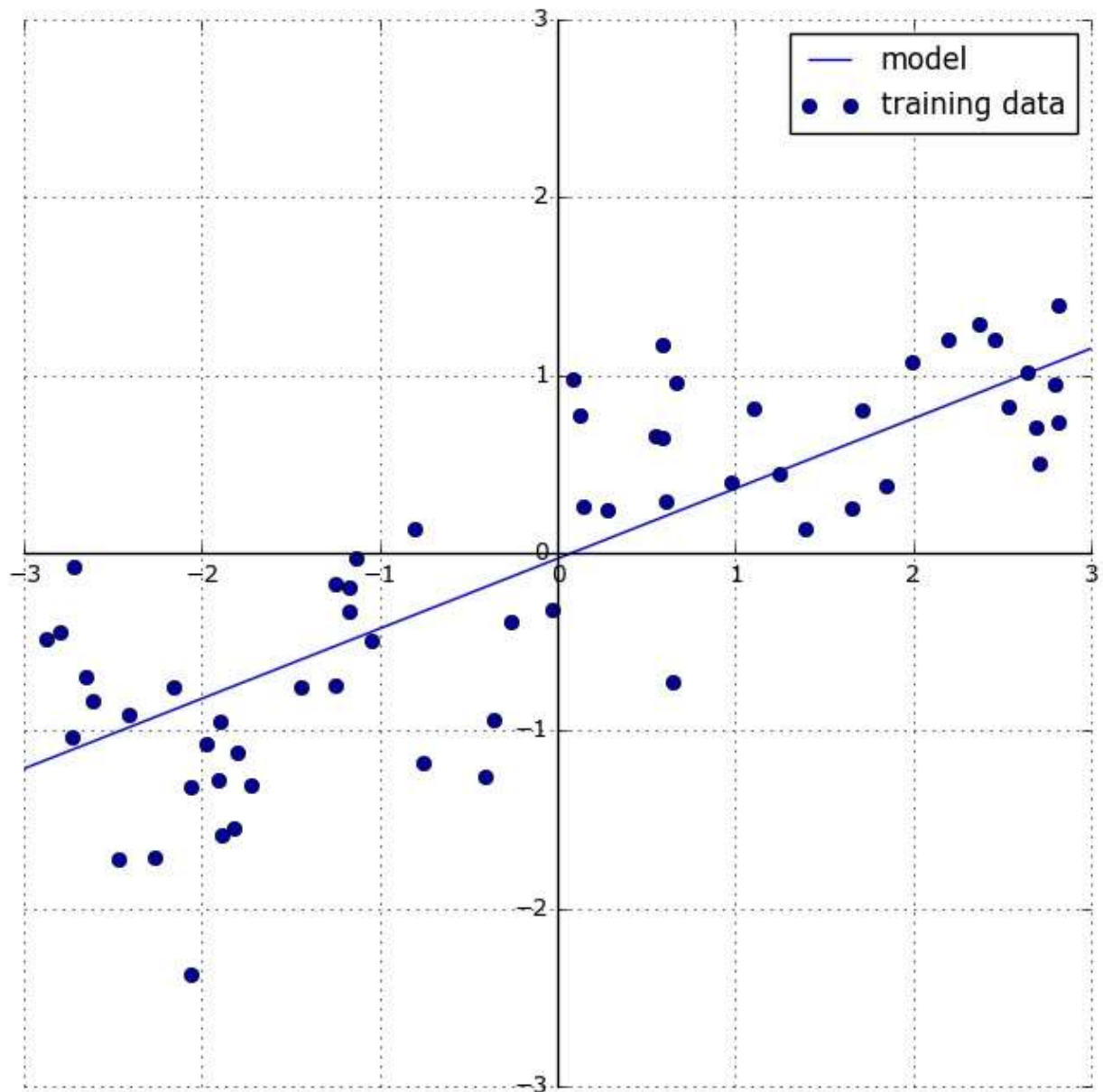


Рис. 2.11 - Прогнозы линейной модели для набора данных wave

Ниже приводится программный код, который строит модель, приведённую на рис. 2.11:

```
In[26]:  
from sklearn.linear_model import LinearRegression  
X, y = mglearn.datasets.make_wave(n_samples=60)  
X_train, X_test, y_train, y_test =
```

```
train_test_split(X, y, random_state=42)
lr = LinearRegression().fit(X_train, y_train)
```

Параметры «наклона» (w), также называемые весами или коэффициентами (coefficients), хранятся в атрибуте `coef_`, тогда как сдвиг (offset) или константа (intercept), обозначаемая как b , хранится в атрибуте `intercept_`:

```
In[27]:
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
Out[27]: lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```

Атрибут `intercept_` - это всегда отдельное число с плавающей точкой, тогда как атрибут `coef_` - это массив NumPy, в котором каждому элементу соответствует входной признак. Поскольку в наборе данных `wave` используется только один входной признак, `lr.coef_` содержит только один элемент.

Давайте посмотрим правильность модели на обучающем и тестовом наборах:

```
In[28]: print("Правильность на обучающем наборе:
{:0.2f}".format(lr.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:0.2f}".format(lr.score(X_test, y_test)))
Out[28]:
```

```
Правильность на обучающем наборе: 0.67
Правильность на тестовом наборе: 0.66
```

Значение R^2 в районе 0.66 указывает на не очень хорошее качество модели, однако можно увидеть, что результаты на обучающем и тестовом наборах очень схожи между собой. Возможно, это указывает на недообучение, а не переобучение. Для этого одномерного массива данных опасность переобучения невелика, поскольку модель очень проста (или строга). Однако для высокоразмерных наборов данных (наборов данных с большим количеством признаков) линейные модели становятся более сложными и существует более высокая вероятность переобучения.

Давайте посмотрим, как `LinearRegression` работает на более сложном наборе данных, например, на наборе `Boston Housing`. Вспомним, что этот набор данных имеет 506 примеров (наблюдений) и 105 производных признаков. Во-первых, мы загрузим набор данных и разобьём его на обучающий и тестовый наборы. Затем построим модель линейной регрессии:

```
In[29]:
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train,
y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

При сравнении правильности на обучающем и тестовом наборах

выясняется, что мы очень точно предсказываем на обучающем наборе, однако R^2 на тестовом наборе имеет довольно низкое значение:

```
In[30]: print("Правильность на обучающем наборе:  
{:.2f}".format(lr.score(X_train, y_train)))
```

```
print("Правильность на тестовом наборе:  
{:.2f}".format(lr.score(X_test, y_test)))
```

```
Out[30]:
```

```
Правильность на обучающем наборе: 0.95
```

```
Правильность на тестовом наборе: 0.61
```

Это несоответствие между правильностью на обучающем наборе и правильностью на тестовом наборе является явным признаком переобучения и поэтому мы должны попытаться найти модель, которая позволит нам контролировать сложность. Одна из наиболее часто используемых альтернатив стандартной линейной регрессии – гребневая регрессия, которую мы рассмотрим ниже.

4 ДЕРЕВЬЯ РЕШЕНИЙ

Цель занятия: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для обработки данных при помощи алгоритма дерева решений.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Деревья решений являются моделями, широко используемыми для решения задач классификации и регрессии. По сути, они задают вопросы и выстраивают иерархию правил «если... то», приводящую к решению.

Эти вопросы похожи на вопросы, которые вы можете спросить в игре «20 Questions». Представьте, вам нужно научиться отличать друг от друга четыре вида животных: медведей, ястребов, пингвинов и дельфинов. Ваша цель состоит в том, чтобы получить правильный ответ, задав несколько вопросов. Вы могли бы начать с вопроса, есть ли у этих видов животных перья, вопроса, который сужает количество возможных видов животных до двух. Если получен ответ «да», вы можете задать еще один вопрос, который может помочь вам различать ястребов и пингвинов. Например, вы могли бы спросить, может ли данный вид животных летать. Если у этого вида животных нет перьев, ваши возможные варианты – дельфины и медведи, и вам нужно задать вопрос, чтобы провести различие между этими двумя видами животных, например, спросить, есть ли плавники у этого вида животных.

Эти вопросы можно выразить в виде дерева решений, как это показано на рис. 2.22.

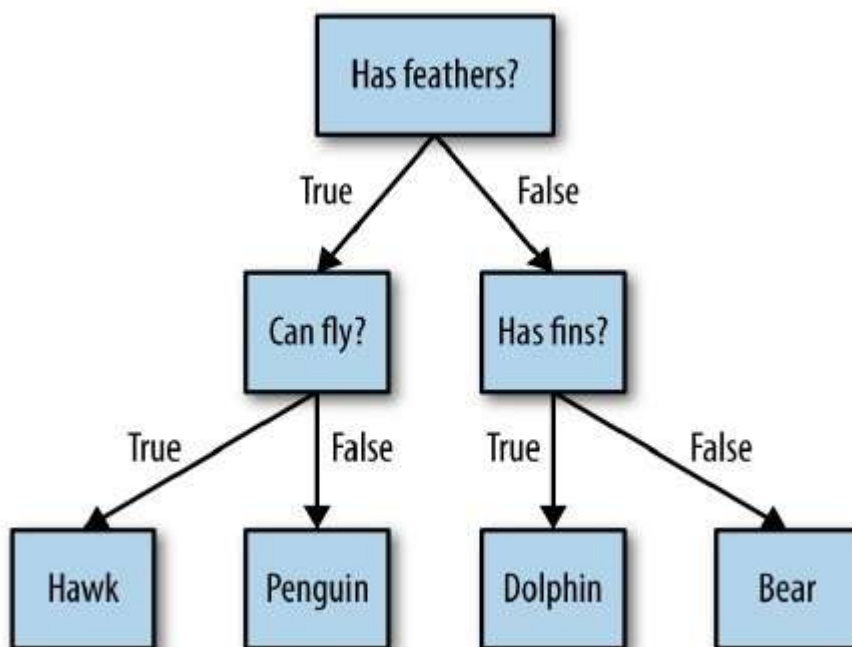


Рис. 2.22 - Дерево решений, различающее несколько видов животных

ПРАКТИЧЕСКАЯ ЧАСТЬ

In[56]:

```
mglearn.plots.plot_animal_tree()
```

На этом рисунке каждый узел дерева либо представляет собой либо вопрос, либо терминальный узел (его еще называют листом или leaf), который содержит ответ. Ребра соединяют вышестоящие узлы с нижестоящими.

Говоря языком машинного обучения, мы построили модель, различающую четыре класса животных (ястребов, пингвинов, дельфинов и медведей), используя три признака «есть перья», «может летать» и «имеет плавники». Вместо того, чтобы строить эти модели вручную, мы можем построить их с помощью контролируемого обучения.

Давайте рассмотрим процесс построения дерева решений для двумерного классификационного набора данных, показанного на рис. 2.23. Набор данных состоит из точек, обозначаемых маркерами двух типов. Каждому типу маркера соответствует свой класс, на каждый класс приходится по 75 точек данных. Назовем этот набор данных `two_moons`.

Построение дерева решений означает построение последовательности правил «если... то...», которая приводит нас к истинному ответу максимально коротким путем. В машинном обучении эти правила называются тестами (tests). Не путайте их с тестовым набором, который мы используем для проверки обобщающей способности нашей модели. Как правило, данные бывают представлены не только в виде бинарных признаков да/нет, как в примере с животными, но и в виде непрерывных признаков, как в двумерном наборе данных, показанном на рис. 2.23. Тесты, которые используются для непрерывных данных имеют вид «Признак i больше значения a ?»

In[57]:

```
mglearn.plots.plot_tree_progressive()
```

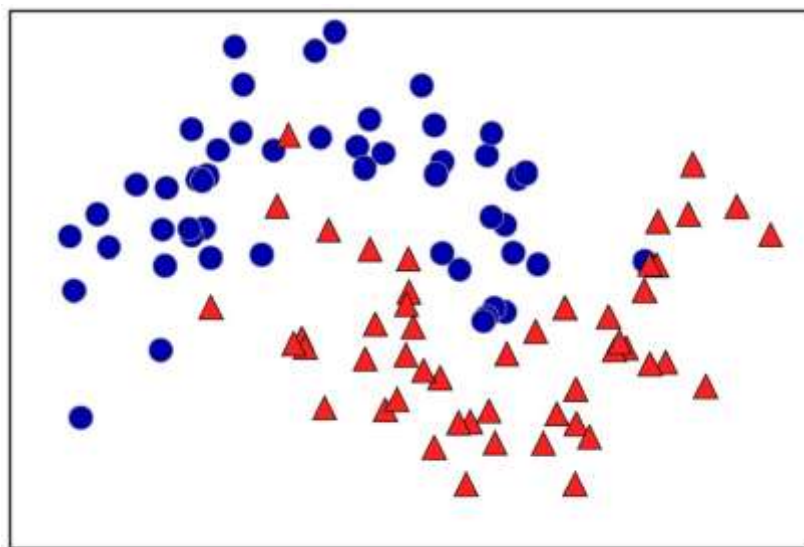


Рис. 2.23 - Набор данных `two_moons`, по которому будет построено дерево решений

Чтобы построить дерево, алгоритм перебирает все возможные тесты и находит тот, который является наиболее информативным с точки зрения прогнозирования значений целевой переменной. Рис. 2.24 показывает первый выбранный тест. Разделение набора данных по горизонтали в точке $x[1]=0.0596$ даёт наиболее полную информацию. Оно лучше всего разделяет точки класса 0 от точек класса 1. Верхний узел, также называемый корнем (root), представляет собой весь набор данных, состоящий из 50 точек, принадлежащих к классу 0, и 50 точек, принадлежащих к классу 1. Разделение выполняется путем тестирования $x[1] \leq 0.0596$, обозначенного черной линией. Если тест верен, точка назначается левому узлу, который содержит 2 точки, принадлежащие классу 0, и 32 точки, принадлежащие классу 1. В противном случае точка будет присвоена правому узлу, который содержит 48 точек, принадлежащих классу 0, и 18 точек, принадлежащих классу 1. Эти два узла соответствуют верхней и нижней областям, показанным на рис. 2.24. Несмотря на то что первое разбиение довольно хорошо разделило два класса, нижняя область по-прежнему содержит точки, принадлежащие к классу 0, а верхняя область по-прежнему содержит точки, принадлежащие к классу 1. Мы можем построить более точную модель, повторяя процесс поиска наилучшего теста в обеих областях. Рис. 2.25 показывает, что следующее наиболее информативное разбиение для левой и правой областей основывается на $x[0]$.

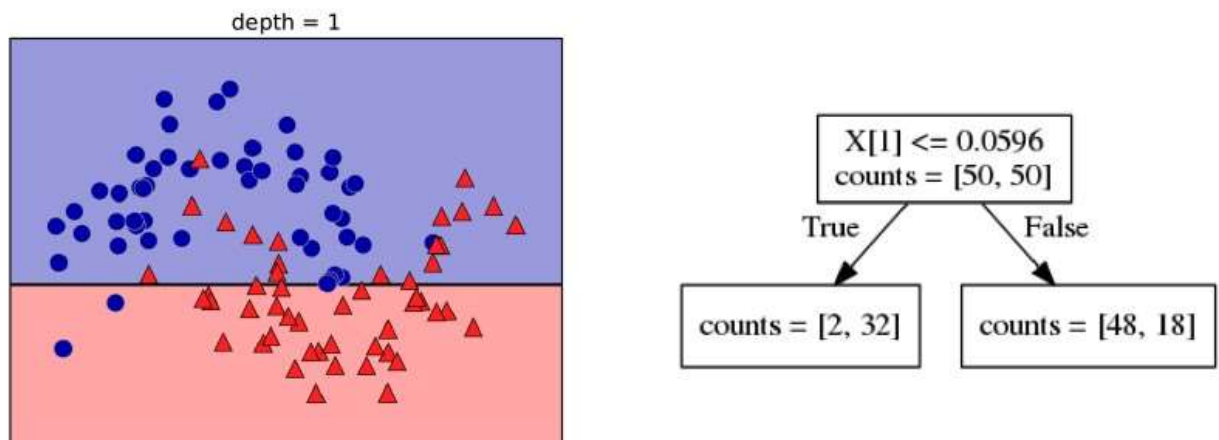


Рис. 2.24 Граница принятия решений, полученная с помощью дерева глубиной 1 (слева) и соответствующее дерево решений (справа)

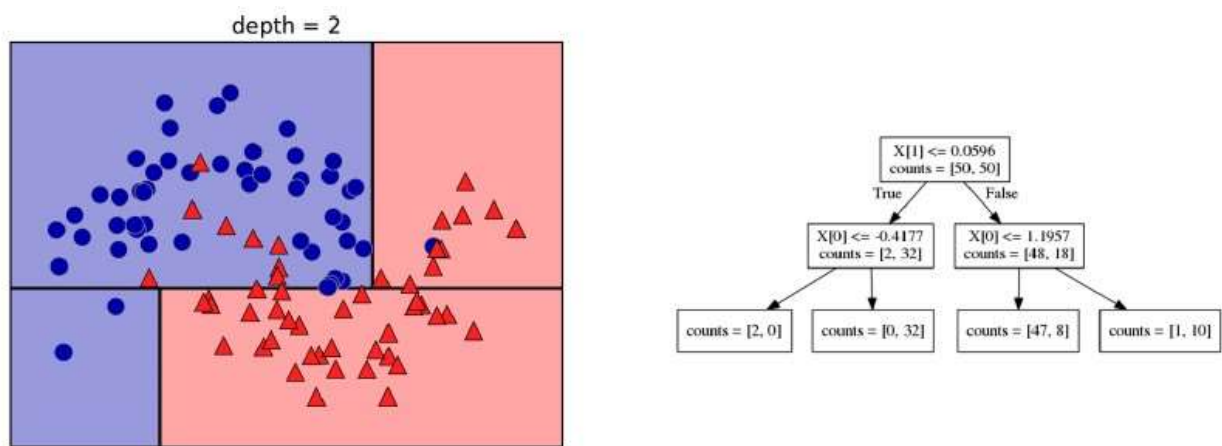


Рис. 2.25 Граница принятия решений, полученная с помощью дерева глубиной 2 (слева) и соответствующее дерево решений (справа)

Этот рекурсивный процесс строит в итоге бинарное дерево решений, в котором каждый узел соответствует определённому тесту. Кроме того, вы можете интерпретировать тест как разбиение части данных, рассматриваемое в данном случае вдоль одной оси. Это позволяет составить представление об алгоритме как способе выстроить иерархию разбиений. Поскольку каждый тест рассматривает только один признак, области, получающиеся в результате разбиения, всегда имеют границы, параллельные осям.

Рекурсивное разбиение данных повторяется до тех пор, пока все точки данных в каждой области разбиения (каждом листе дерева решений) не будут принадлежать одному и тому же значению целевой переменной (классу или количественному значению). Лист дерева, который содержит точки данных, относящиеся к одному и тому же значению целевой переменной, называется чистым (pure). Итоговое разбиение для нашего набора данных показано на рис. 2.26.

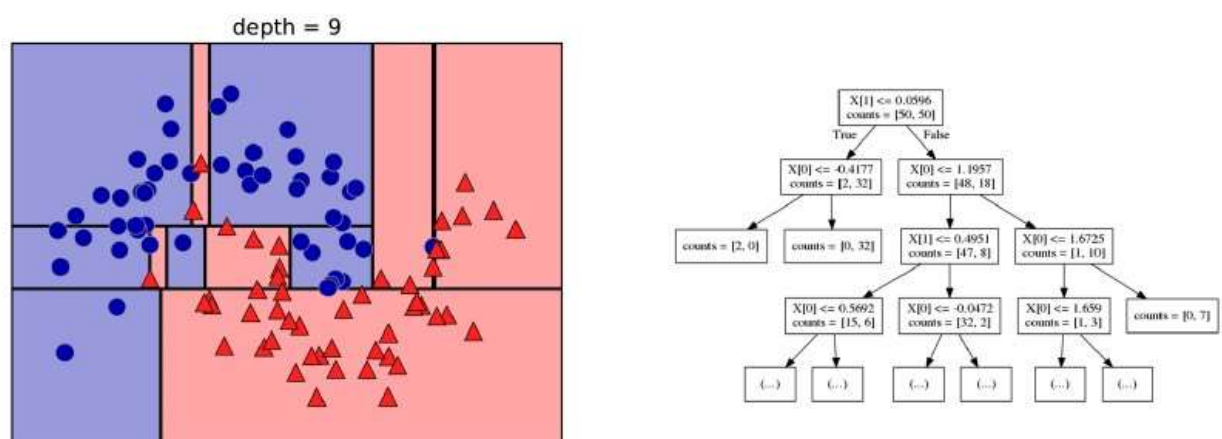


Рис. 2.26 Граница принятия решений, полученная с помощью дерева глубиной 9 (слева) и фрагмент соответствующего дерева (справа), полное дерево имеет довольно большой размер и его сложно визуализировать

Прогноз для новой точки данных получают следующим образом: сна-

чала выясняют, в какой области разбиения пространства признаков находится данная точка, а затем определяют класс, к которому относится большинство точек в этой области (либо единственный класс в области, если лист является чистым). Область может быть найдена с помощью обхода дерева, начиная с корневого узла и путём перемещения влево или вправо, в зависимости от того, выполняется ли тест или нет.

Кроме того, можно использовать деревья для решения задач регрессии, используя точно такой же подход. Для получения прогноза мы обходим дерево на основе тестов в каждом узле и находим лист, в который попадает новая точка данных. Выходом для этой точки данных будет значение целевой переменной, усреднённое по всем обучающим точкам в этом листе.

Как правило, построение дерева, описанное здесь и продолжающееся до тех пор, пока все листья не станут чистыми, приводит к получению моделей, которые являются очень сложными и характеризуются сильным переобучением на обучающих данных. Наличие чистых листьев означает, что дерево имеет 100%-ную правильность на обучающей выборке. Каждая точка обучающего набора находится в листе, который имеет правильный мажоритарный класс. Переобучение можно увидеть в левой части рис. 2.26. Видно, что точки, определяемые как точки класса 1, находятся среди точек, принадлежащих к классу 0. С другой стороны, мы видим ряд точек, спрогнозированных как класс 1, вокруг точки, отнесенной к классу 0. Это не та граница принятия решений, которую мы могли бы себе представить. Здесь граница принятия решений фокусируется больше на отдельных точках-выбросах, которые находятся слишком далеко от остальных точек данного класса.

Есть две общераспространенные стратегии, позволяющие предотвратить переобучение. Первая стратегия – ранняя остановка построения дерева, называемая предварительной обрезкой (pre-pruning). Вторая стратегия – построение дерева с последующим удалением или сокращением малоинформативных узлов, называемое пост-обрезкой (post-pruning) или просто обрезкой (pruning). Возможные критерии предварительной обрезки включают в себя ограничение максимальной глубины дерева, ограничение максимального количества листьев или минимальное количество наблюдений в узле, необходимое для разбиения.

В библиотеке `scikit-learn` деревья решений реализованы в классах `DecisionTreeRegressor` и `DecisionTreeClassifier`. Обратите внимание, в `scikit-learn` реализована лишь предварительная обрезка.

Давайте более детально посмотрим, как работает предварительная обрезка на примере набора данных `Breast Cancer`. Как всегда, мы импортируем набор данных и разбиваем его на обучающую и тестовую части. Затем мы строим модель, используя настройки по умолчанию для построения полного дерева (выращиваем дерево до тех пор, пока все листья не станут чистыми). Зафиксируем `random_state` для воспроизводимости результатов:

```
In[58]:
```

```
from sklearn.tree import DecisionTreeClassifier
```

```

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test =
train_test_split(
cancer.data, cancer.target, stratify=cancer.target,
random_state=42) tree = DecisionTreeClassifier(random_state=0) tree.fit(X_train, y_train)
print("Правильность на обучающем наборе:
{:.3f}".format(tree.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(tree.score(X_test, y_test)))

```

Out[58]:

Правильность на обучающем наборе: 1.000

Правильность на тестовом наборе: 0.937

Как и следовало ожидать, правильность на обучающем наборе составляет 100%, поскольку листья являются чистыми. Дерево имеет глубину, как раз достаточную для того, чтобы прекрасно запомнить все метки обучающих данных. Правильность на тестовом наборе немного хуже, чем при использовании ранее рассмотренных линейных моделей, правильность которых составляла около 95%.

Если не ограничить глубину, дерево может быть сколь угодно глубоким и сложным. Поэтому необрезанные деревья склонны к переобучению и плохо обобщают результат на новые данные. Теперь давайте применим к дереву предварительную обрезку, которая остановит процесс построения дерева до того, как мы идеально подгоним модель к обучающим данным. Один из вариантов – остановка процесса построения дерева по достижении определенной глубины. Здесь мы установим `max_depth=4`, то есть можно задать только четыре последовательных вопроса (см. рис. 2.24 и 2.26). Ограничение глубины дерева уменьшает переобучение. Это приводит к более низкой правильности на обучающем наборе, но улучшает правильность на тестовом наборе:

```

In[59]: tree = DecisionTreeClassifier(max_depth=4,
random_state=0) tree.fit(X_train, y_train)
print("Правильность на обучающем наборе:
{:.3f}".format(tree.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(tree.score(X_test, y_test)))

```

Out[59]:

Правильность на обучающем наборе: 0.988

Правильность на тестовом наборе: 0.951

Мы можем визуализировать дерево, используя функцию `export_graphviz` из модуля `tree`. Она записывает файл в формате `.dot`, который является форматом текстового файла, предназначенным для описания графиков. Мы можем задать цвет узлам, чтобы выделить класс, набравший большинство в каждом узле, и передать имена классов и признаков, чтобы

дерево было правильно размечено:

```
In[60]:
```

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot",
class_names=["malignant", "benign"],
feature_names=cancer.feature_names, impurity=False,
filled=True)
```

Мы можем прочитать этот файл и визуализировать его, как показано на рис. 2.27, используя модуль graphviz (или любую другую программу, которая может читать файлы с расширением .dot):

```
In[61]: import graphviz
with open("tree.dot") as f:
dot_graph = f.read()
```

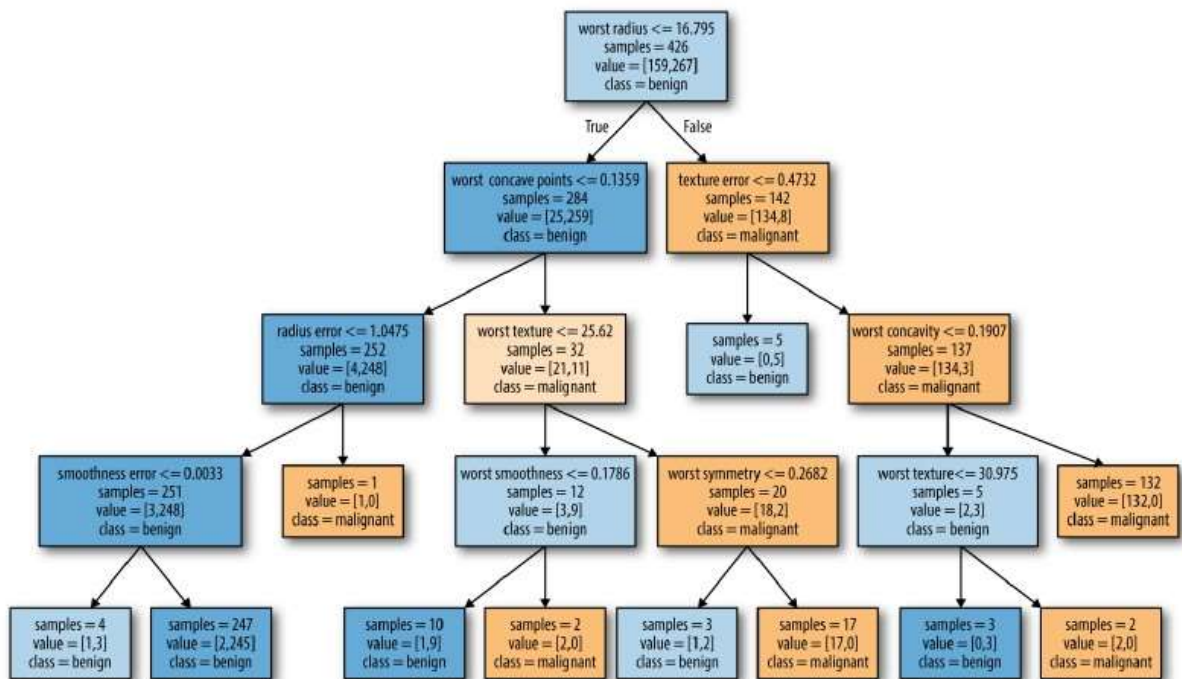


Рис. 2.27 Визуализация дерева решений, построенного на наборе данных Breast Cancer

Как вариант, можно построить диаграмму дерева и записать ее в файл .pdf. Дополнительно нам потребуется модуль pydotplus.

```
import numpy as np import matplotlib.pyplot as plt
import pandas as pd import mglearn %matplotlib inline
from sklearn.model_selection import
train_test_split from sklearn.datasets import
load_breast_cancer from sklearn import tree
from sklearn.tree import export_graphviz
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test =
train_test_split(
```

```
cancer.data, cancer.target, stratify=cancer.target,
random_state=42) clf =
tree.DecisionTreeClassifier(max_depth=4, ran-
dom_state=0) clf = clf.fit(X_train, y_train)
import pydotplus
dot_data = tree.export_graphviz(clf, out_file=None)
pip install graphviz
```

Затем в переменной окружения PATH необходимо прописать полный путь к установленной папке graphviz. В Windows 7 для этого нажмите кнопку Пуск, выберите Панель управления. Дважды нажмите на Система, затем выберите Дополнительные параметры системы. Во вкладке Дополнительно нажмите на Переменные среды. Выберите Path и нажмите на Изменить. В поле Значение переменной введите путь к папке graphviz (например, C:\Anaconda3\Library\bin\graphviz).

Можно построить визуализацию дерева с помощью функции Image интерактивной оболочки IPython:

```
from IPython.display import Image
dot_data = tree.export_graphviz(clf, out_file=None,
feature_names=cancer.feature_names,
class_names=cancer.target_names,
filled=True, rounded=True,
special_characters=True) graph = pydot-
plus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

Визуализация дерева дает более глубокое представление о том, как алгоритм делает прогнозы и является хорошим примером алгоритма машинного обучения, который легко объяснить неспециалистам. Однако, как показано здесь, даже при глубине 4 дерево может стать немного громоздким. Деревья с большим значением глубины (деревья глубиной 10 – не редкость) ещё труднее понять. Один из полезных способов исследования дерева заключается в том, чтобы выяснить, какие узлы содержат наибольшее количество данных. Параметр `samples`, выводимый в каждом узле на рис. 2.27, показывает общее количество примеров в узле, тогда как параметр `value` показывает количество примеров в каждом классе. Проследовав по правой ветви, отходящей от корневого узла, мы видим, что правилу `worst radius > 16.795` соответствует узел, который содержит 134 случая злокачественной опухоли и лишь 8 случаев доброкачественной опухоли. Далее дерево выполняет серию более точных разбиений оставшихся 142 случаев. Из 142 случаев, которые при первоначальном разбиении были записаны в правый узел, почти все (132) в итоге попали в правый лист (для удобства выделен красной рамкой).

Проследовав по левой ветви, отходящей от корневого узла, мы видим, что правилу `worst radius <= 16.795` соответствует узел, который содержит 25 случаев злокачественной опухоли и 259 случаев доброкачественной опухоли. Почти все случаи доброкачественной опухоли попадают во второй лист

справа (для удобства выделен синей рамкой), остальные случаи распределяются по нескольким листьям, содержащим очень мало наблюдений.

Вместо того, чтобы просматривать все дерево, что может быть обременительно, есть некоторые полезные параметры, которые мы можем использовать как итоговые показатели работы дерева. Наиболее часто используемым показателем является важность признаков (feature importance), которая оценивает, насколько важен каждый признак с точки зрения получения решений. Это число варьирует в диапазоне от 0 до 1 для каждого признака, где 0 означает «не используется вообще», а 1 означает, что «отлично предсказывает целевую переменную». Важности признаков в сумме всегда дают 1:

```
In[62]: print("Важности признаков:\n{}".format(tree.feature_importances_))
Out[62]:
Важности признаков
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.01019737 0.04839825
 0.  0.  0.0024156  0.  0.  0.  0.  0.  0.72682851
 0.0458159  0.  0.  0.0141577  0.  0.018188
 0.1221132  0.01188548  0.]
```

Приведённая сводка не совсем удобна, поскольку мы не знаем, каким именно признакам соответствуют приведённые важности. Чтобы исправить это, воспользуемся программным кодом, приведённым ниже:

```
for name, score in zip(cancer["feature_names"],
tree.feature_importances_):
print(name, score)
mean radius 0.0 mean texture 0.0 mean perimeter 0.0
mean area 0.0 mean smoothness 0.0 mean compactness
0.0 mean concavity 0.0 mean concave points 0.0 mean
symmetry 0.0 mean fractal dimension 0.0 radius er-
ror 0.0101973682021 texture error 0.0483982536186
perimeter error 0.0 area error 0.0
smoothness error 0.00241559508532
compactness error 0.0 concavity error 0.0 concave
points error 0.0 symmetry error 0.0 fractal dimen-
sion error 0.0 worst radius 0.72682850946 worst
texture 0.0458158970889 worst perimeter 0.0 worst
area 0.0
worst smoothness 0.0141577021047 worst compactness
0.0 worst concavity 0.0181879968645 worst concave
points 0.122113199265 worst symmetry
0.0118854783101 worst fractal dimension 0.0
```

Мы можем визуализировать важности признаков аналогично тому, как мы визуализируем коэффициенты линейной модели (рис. 2.28):

```
In[63]:
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features),
model.feature_importances_, align='center')
plt.yticks(np.arange(n_features), can-
cer.feature_names)      plt.xlabel("Важность
признака")      plt.ylabel("Признак")
plot_feature_importances_cancer(tree)
```

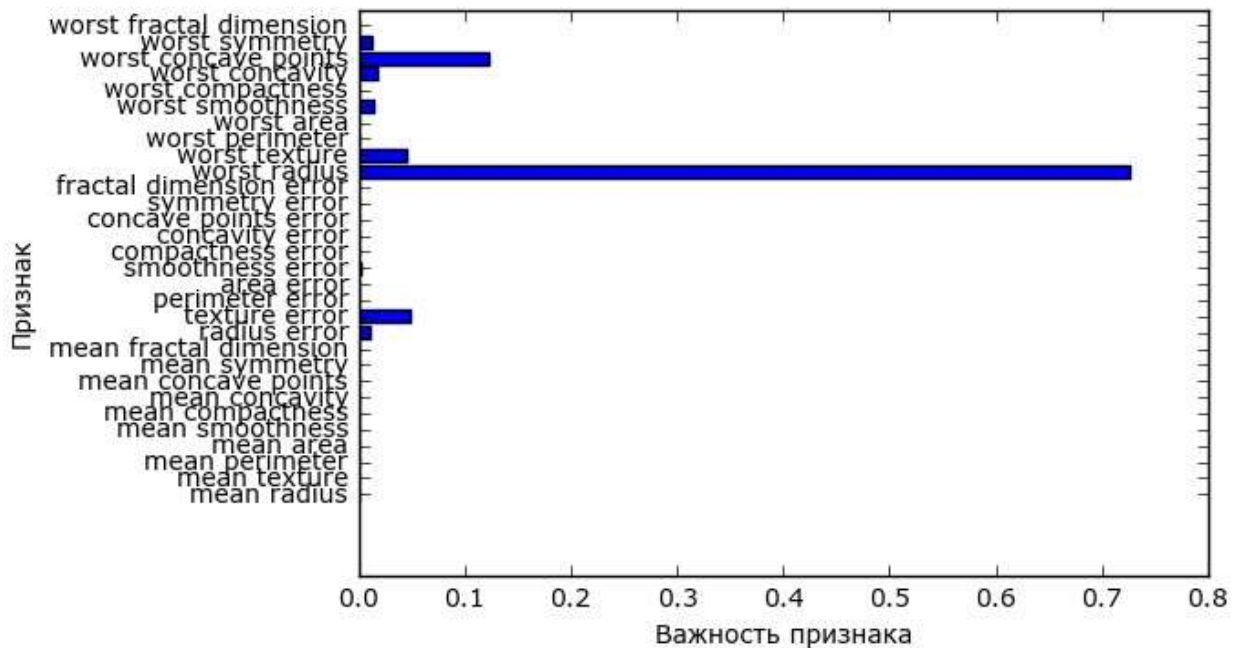


Рис. 2.28 - Важности признаков, вычисленные с помощью дерева решений для набора данных Breast Cancer

Здесь мы видим, что признак, использованный в самом верхнем разбиении («worst radius»), на данный момент является наиболее важным. Это подтверждает наш вывод о том, что уже на первом уровне два класса достаточно хорошо разделены.

Однако, если признак имеет низкое значение `feature_importance_`, это не значит, что он неинформативен. Это означает только то, что данный признак не был выбран деревом, поскольку, вероятно, другой признак содержит ту же самую информацию.

В отличие от коэффициентов линейных моделей важности признаков всегда положительны и они не указывают на взаимосвязь с каким-то конкретным классом. Важности признаков говорят нам, что «worst radius» важен, но мы не знаем, является ли высокое значение радиуса признаком доброкачественной или злокачественной опухоли. На самом деле, найти такую очевидную взаимосвязь между признаками и классом невозможно, что можно проиллюстрировать на следующем примере (рис. 2.29 и 2.30):

```

In[64]:
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
Out[64]:
Feature importances: [ 0.  1.]

```

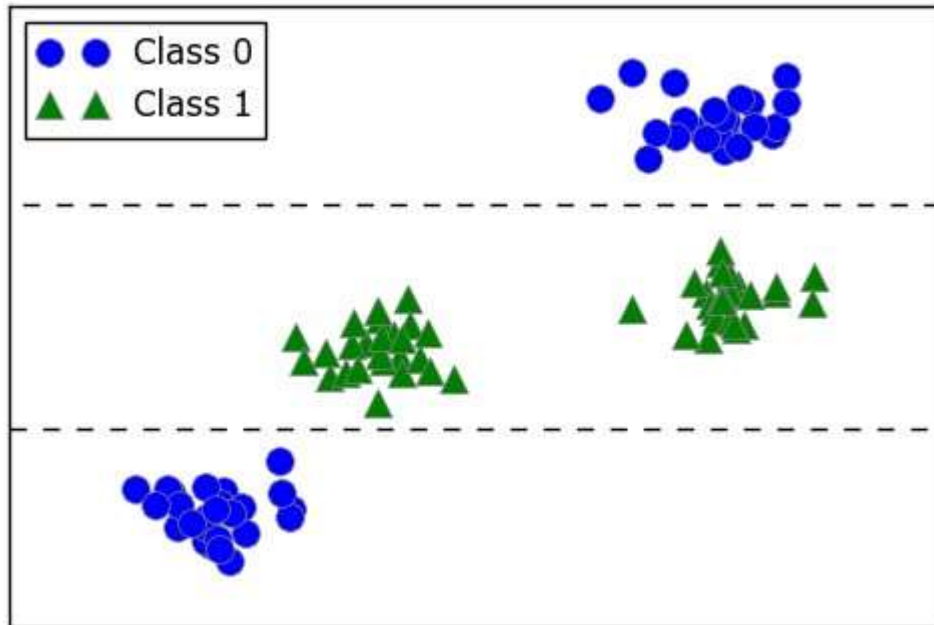


Рис. 2.29 - Двумерный массив данных, в котором признак имеет немонотонную взаимосвязь с меткой класса, и границы принятия решений, найденные с помощью дерева

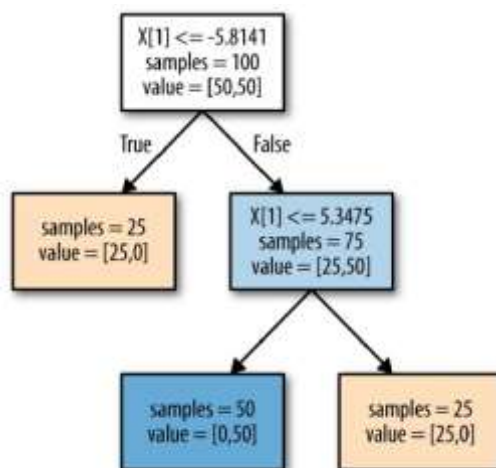


Рис. 2.30 - Дерево решений для набора данных, показанном на рис. 2.29

График показывает набор данных с двумя признаками и двумя классами. Здесь вся информация содержится в $X[1]$, а $X[0]$ не используется вообще. Но взаимосвязь между $X[1]$ и целевым классом не является монотонной, то есть мы не можем сказать, что «высокое значение $X[0]$ означает класс 0, а

низкое значение означает класс 1» (или наоборот).

Несмотря на то, что мы сосредоточились здесь на деревьях классификации, все вышесказанное верно и для деревьев регрессии, которые реализованы в `DecisionTreeRegressor`. Применение и анализ деревьев регрессии очень схожи с применением и анализом деревьев классификации. Однако существует одна особенность использования деревьев регрессии, на которую нужно указать. `DecisionTreeRegressor` (и все остальные регрессионные модели на основе дерева) не умеет экстраполировать или делать прогнозы вне диапазона значений обучающих данных.

Давайте детальнее рассмотрим это, воспользовавшись набором данных RAM Price (содержит исторические данные о ценах на компьютерную память). Рис. 2.31 визуализирует этот набор данных, дата отложена по оси x, а цена одного мегабайта оперативной памяти в соответствующем году – по оси y:

```
In[65]: import pandas as pd
ram_prices = pd.read_csv("C:/Data/ram_price.csv")
plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Год") plt.ylabel("Цена $/Мбайт")
```

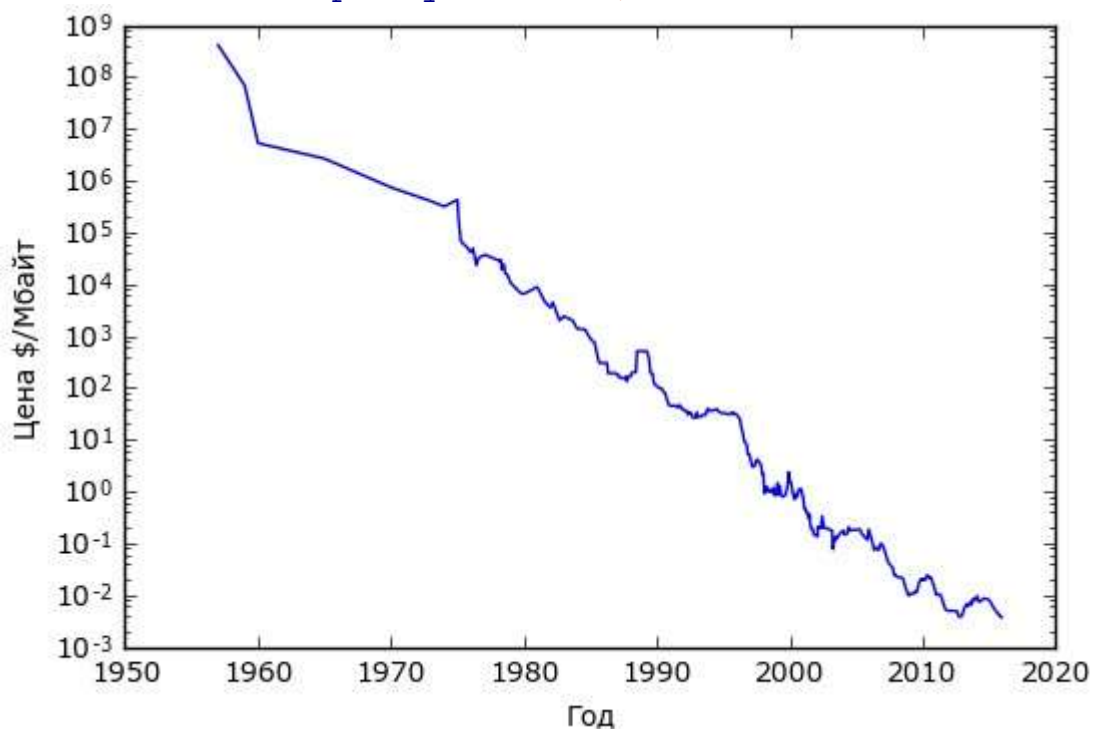


Рис. 2.31 - Историческое развитие цен на RAM по логарифмической шкале

Обратите внимание на логарифмическую шкалу оси y. При логарифмическом преобразовании взаимосвязь выглядит вполне линейной и таким образом становится легко прогнозируемой, за исключением некоторых всплесков.

Мы будем прогнозировать цены на период после 2000 года, используя исторические данные до этого момента, единственным признаком будут даты. Мы сравним две простые модели: `DecisionTreeRegressor` и

LinearRegression. Мы отмасштабируем цены, используя логарифм, таким образом, взаимосвязь будет относительно линейной. Это несущественно для DecisionTreeRegressor, однако существенно для LinearRegression (мы рассмотрим ее более подробно в главе 4). После обучения модели и получения прогнозов мы применим экспоненцирование, чтобы обратить логарифмическое преобразование. Мы получим и визуализируем прогнозы для всего набора данных, но для количественной оценки мы будем рассматривать только тестовый набор:

```
In[66]: from sklearn.tree import Decision-
TreeRegressor
# используем исторические данные для
прогнозирования цен после 2000 года
data_train = ram_prices[ram_prices.date < 2000] da-
ta_test = ram_prices[ram_prices.date >= 2000]
# прогнозируем цены по датам
X_train = data_train.date[:, np.newaxis]
# мы используем логпреобразование, что получить
простую взаимосвязь между данными и откликом
y_train = np.log(data_train.price)
tree = DecisionTreeRegressor().fit(X_train,
y_train) linear_reg = LinearRegres-
sion().fit(X_train, y_train)
# прогнозируем по всем данным
X_all = ram_prices.date[:, np.newaxis]
pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)
# экспоненцируем, чтобы обратить логарифмическое
преобразование price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

Рис. 2.32, созданный здесь, сравнивает прогнозы дерева решений и линейной регрессии с реальными.

```
In[67]:
plt.semilogy(data_train.date, data_train.price, la-
bel="Обучающие данные")
plt.semilogy(data_test.date, data_test.price, la-
bel="Тестовые данные")
plt.semilogy(ram_prices.date, price_tree, la-
bel="Прогнозы дерева")
plt.semilogy(ram_prices.date, price_lr, la-
bel="Прогнозы линейной регрессии") plt.legend()
```

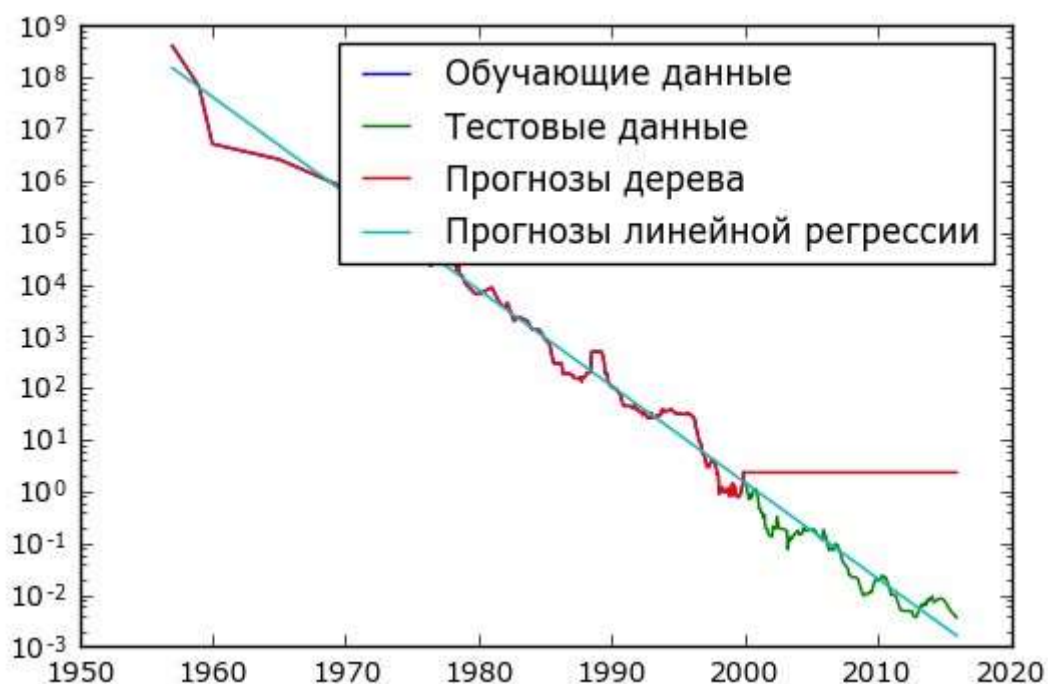


Рис. 2.32 - Сравнение прогнозов линейной модели и прогнозов дерева регрессии для набора данных RAM price

Разница между моделями получилась весьма впечатляющая. Линейная модель аппроксимирует данные с помощью уже известной нам прямой линии. Эта линия даёт достаточно хороший прогноз для тестовых данных (период после 2000 года), при этом сглаживая некоторые всплески в обучающих и тестовых данных. С другой стороны, модель дерева прекрасно прогнозирует на обучающих данных. Здесь мы не ограничивали сложность дерева, поэтому она полностью запомнила весь набор данных. Однако, как только мы выходим из диапазона значений, известных модели, модель просто продолжает предсказывать последнюю известную точку. Дерево не способно генерировать «новые» ответы, выходящие за пределы значений обучающих данных. Этот недостаток относится ко всем моделям на основе деревьев решений.

Как уже говорилось выше, параметры, которые контролируют сложность модели в деревьях решений – это параметрами предварительной обрезки дерева, которые останавливают построение дерева, прежде чем оно достигнет максимального размера. Обычно, чтобы предотвратить переобучение, достаточно выбрать одну из стратегий предварительной обрезки – настроить `max_depth`, `max_leaf_nodes` или `min_samples_leaf`.

По сравнению со многими алгоритмами, обсуждавшимися до сих пор, деревья решений обладают двумя преимуществами: полученная модель может быть легко визуализирована и понята неспециалистами (по крайней мере это верно для небольших деревьев) и деревья не требуют масштабирования данных. Поскольку каждый признак обрабатывается отдельно, а возможные разбиения данных не зависят от масштабирования, алгоритмы деревьев ре-

шений не нуждаются в таких процедурах предварительной обработки, как нормализация или стандартизация признаков. Деревья решений хорошо работают, когда у вас есть признаки, измеренные в совершенно разных шкалах, или когда ваши данные представляют смесь бинарных и непрерывных признаков.

Основным недостатком деревьев решений является то, что даже при использовании предварительной обрезки, они склонны к переобучению и имеют низкую обобщающую способность. Поэтому в большинстве случаев, как правило, вместо одиночного дерева решений используются ансамбли деревьев, которые мы обсудим далее.

5 СЛУЧАЙНЫЙ ЛЕС

Цель занятия: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для обработки данных при помощи алгоритма случайный лес.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Ансамбли (ensembles) – это методы, которые сочетают в себе множество моделей машинного обучения, чтобы в итоге получить более мощную модель. Существует много моделей машинного обучения, которые принадлежат к этой категории, но есть две ансамблевых модели, которые доказали свою эффективность на самых различных наборах данных для задач классификации и регрессии, обе используют деревья решений в качестве строительных блоков: случайный лес деревьев решений и градиентный бустинг деревьев решений.

Как мы только что отметили, основным недостатком деревьев решений является их склонность к переобучению. Случайный лес является одним из способов решения этой проблемы. По сути случайный лес – это набор деревьев решений, где каждое дерево немного отличается от остальных. Идея случайного леса заключается в том, что каждое дерево может довольно хорошо прогнозировать, но скорее всего переобучается на части данных. Если мы построим много деревьев, которые хорошо работают и переобучаются с разной степенью, мы можем уменьшить переобучение путем усреднения их результатов. Уменьшение переобучения при сохранении прогнозной силы деревьев можно проиллюстрировать с помощью строгой математики.

Для реализации вышеизложенной стратегии нам нужно построить большое количество деревьев решений. Каждое дерево должно на приемлемом уровне прогнозировать целевую переменную и должно отличаться от других деревьев. Случайные леса получили свое название из-за того, что в процесс построения деревьев была внесена случайность, призванная обеспечить уникальность каждого дерева. Существует две техники, позволяющие получить рандомизированные деревья в рамках случайного леса: сначала выбираем точки данных (наблюдения), которые будут использоваться для построения дерева, а затем отбираем признаки в каждом разбиении. Давайте разберем этот процесс более подробно.

Для построения модели случайных лесов необходимо определиться с количеством деревьев (параметр `n_estimators` для `RandomForestRegressor` или `RandomForestClassifier`). Допустим, мы хотим построить 10 деревьев. Эти деревья будут построены совершенно независимо друг от друга, и алгоритм будет случайным образом отбирать признаки для построения каждого дерева, чтобы получить непохожие друг на друга деревья. Для построения дерева мы сначала сформируем бутстреп-выборку (bootstrap sample) наших данных. То есть из `n_samples` примеров мы случайным образом выбираем пример с

возвращением `n_samples` раз (поскольку отбор с возвращением, то один и тот же пример может быть выбран несколько раз). Мы получаем выборку, которая имеет такой же размер, что и исходный набор данных, однако некоторые примеры будут отсутствовать в нем (примерно одна треть), а некоторые попадут в него несколько раз.

Чтобы проиллюстрировать это, предположим, что мы хотим создать бутстреп-выборку списка ['a', 'b', 'c', 'd']. Возможная бутстрепвыборка может выглядеть как ['b', 'd', 'd', 'c']. Другой возможной бутстреп-выборкой может быть ['d', 'a', 'd', 'a'].

Далее на основе этой сформированной бутстреп-выборки строится дерево решений. Однако алгоритм, который мы описывали для дерева решений, теперь слегка изменен. Вместо поиска наилучшего теста для каждого узла, алгоритм для разбиения узла случайным образом отбирает подмножество признаков и затем находит наилучший тест, используя один из этих признаков. Количество отбираемых признаков контролируется параметром `max_features`. Отбор подмножества признаков повторяется отдельно для каждого узла, поэтому в каждом узле дерева может быть принято решение с использованием «своего» подмножества признаков.

Использование бутстрепа приводит к тому, что деревья решений в случайном лесе строятся на немного отличающихся между собой бутстреп-выборках. Из-за случайного отбора признаков в каждом узле все расщепления в деревьях будут основано на отличающихся подмножествах признаков. Вместе эти два механизма приводят к тому, что все деревья в случайном лесе отличаются друг от друга.

Критическим параметром в этом процессе является `max_features`. Если мы установим `max_features` равным `n_features`, это будет означать, что в каждом разбиении могут участвовать все признаки набора данных, и в отбор признаков не будет привнесена случайность (впрочем, случайность в силу использования бутстрепа остается). Если мы установим `max_features` равным 1, это означает, что при разбиении не будет никакого отбора признаков для тестирования вообще, будет осуществляться поиск с учетом различных пороговых значений для случайно выбранного признака. Таким образом, высокое значение `max_features` означает, что деревья в случайном лесе будут весьма схожи между собой и они смогут легко аппроксимировать данные, используя наиболее дискриминирующие признаки. Низкое значение `max_features` означает, что деревья в случайном лесе будут сильно отличаться друг от друга и, возможно, каждое дерево будет иметь очень большую глубину, чтобы хорошо соответствовать данным.

Чтобы дать прогноз для случайного леса, алгоритм сначала дает прогноз для каждого дерева в лесе. Для регрессии мы можем усреднить эти результаты, чтобы получить наш окончательный прогноз. Для классификации используется стратегия «мягкого голосования». Это означает, что каждый алгоритм дает «мягкий» прогноз, вычисляя вероятности для каждого класса. Эти вероятности усредняются по всем деревьям и прогнозируется класс с

наибольшей вероятностью.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Давайте применим случайный лес, состоящий из пяти деревьев, к набору данных `two_moons`, который мы изучали ранее:

```
In[68]:
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
random_state=42)
forest = RandomForestClassifier(n_estimators=5,
random_state=2) forest.fit(X_train, y_train)
```

Деревья, которые строятся в рамках случайного леса, сохраняются в атрибуте `estimator_`. Давайте визуализируем границы принятия решений, полученные каждым деревом, а затем выведем агрегированный прогноз, выданный лесом (рис. 2.33):

```
In[69]: fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Дерево {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
alpha=.4) axes[-1, -1].set_title("Случайный лес")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

На рисунках отчетливо видно, что границы принятия решений, полученные с помощью пяти деревьев, существенно различаются между собой. Каждое дерево совершает ряд ошибок, поскольку из-за бутстрепа некоторые точки исходного обучающего набора фактически не были включены в обучающие наборы, по которым строились деревья.

В отличие от отдельных деревьев случайный лес переобучается в меньшей степени и дает гораздо более чувствительную (гибкую) границу принятия решений. В реальных примерах используется гораздо большее количество деревьев (часто сотни или тысячи), что приводит к получению еще более чувствительной границы.

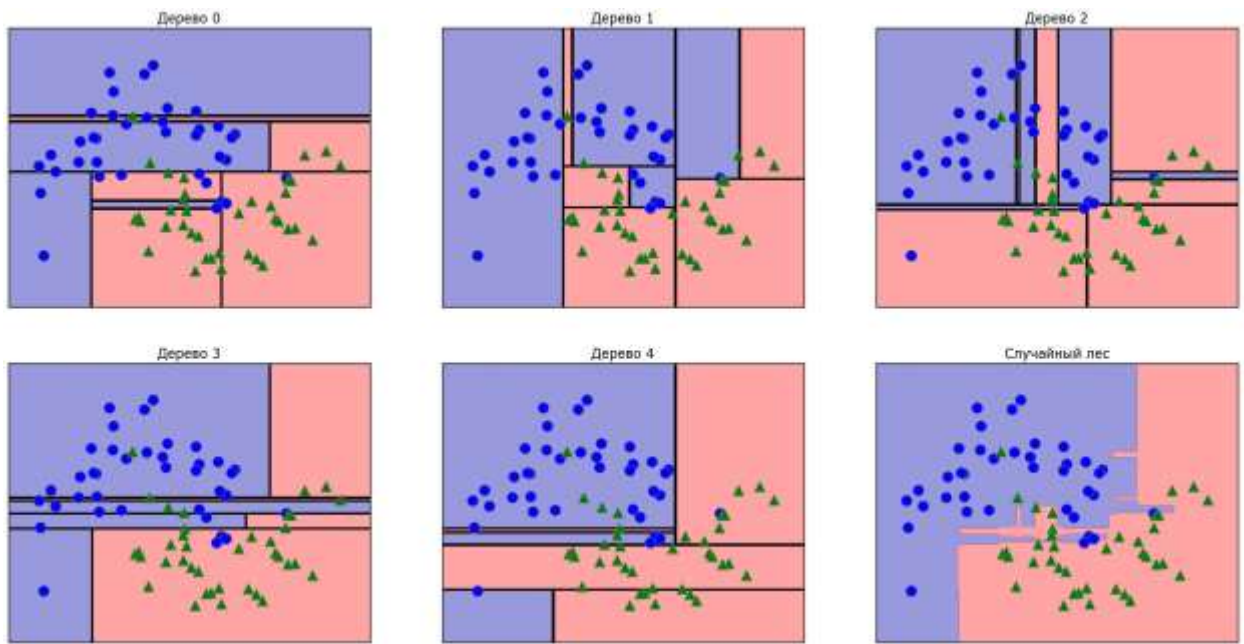


Рис. 2.33 - Границы принятия решений, найденные пятью рандомизированными деревьями решений, и граница принятия решений, полученная путем усреднения их спрогнозированных вероятностей

В качестве еще одного примера давайте построим случайный лес, состоящий из 100 деревьев, на наборе данных Breast Cancer:

```
In[70]:
X_train, X_test, y_train, y_test =
train_test_split(cancer.data, cancer.target,
random_state=0)
forest = RandomForestClassifier(n_estimators=100,
random_state=0) forest.fit(X_train, y_train)
print("Правильность на обучающем наборе:
{:.3f}".format(forest.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(forest.score(X_test, y_test)))
Out[70]:
Правильность на обучающем наборе: 1.000
Правильность на тестовом наборе: 0.972
```

Без настройки каких-либо параметров случайный лес дает нам правильность 97%, это лучше результата линейных моделей или одиночного дерева решений. Мы могли бы отрегулировать настройку `max_features` или применить предварительную обрезку, как это делали для одиночного дерева решений. Однако часто параметры случайного леса, выставленные по умолчанию, работают уже сами по себе достаточно хорошо.

Как и дерево решений, случайный лес позволят вычислить важности признаков, которые рассчитываются путем агрегирования значений важности

по всем деревьям леса. Как правило, важности признаков, вычисленные случайным лесом, являются более надежным показателем, чем важности, вычисленные одним деревом. Посмотрите на рис. 2.34.

```
In[71]: def
plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.
feature_importances_, align='center')
plt.yticks(np.arange(n_features), cancer.
feature_names)
plt.xlabel("Важность признака")
plt.ylabel("Признак")
plot_feature_importances_cancer(forest)
```

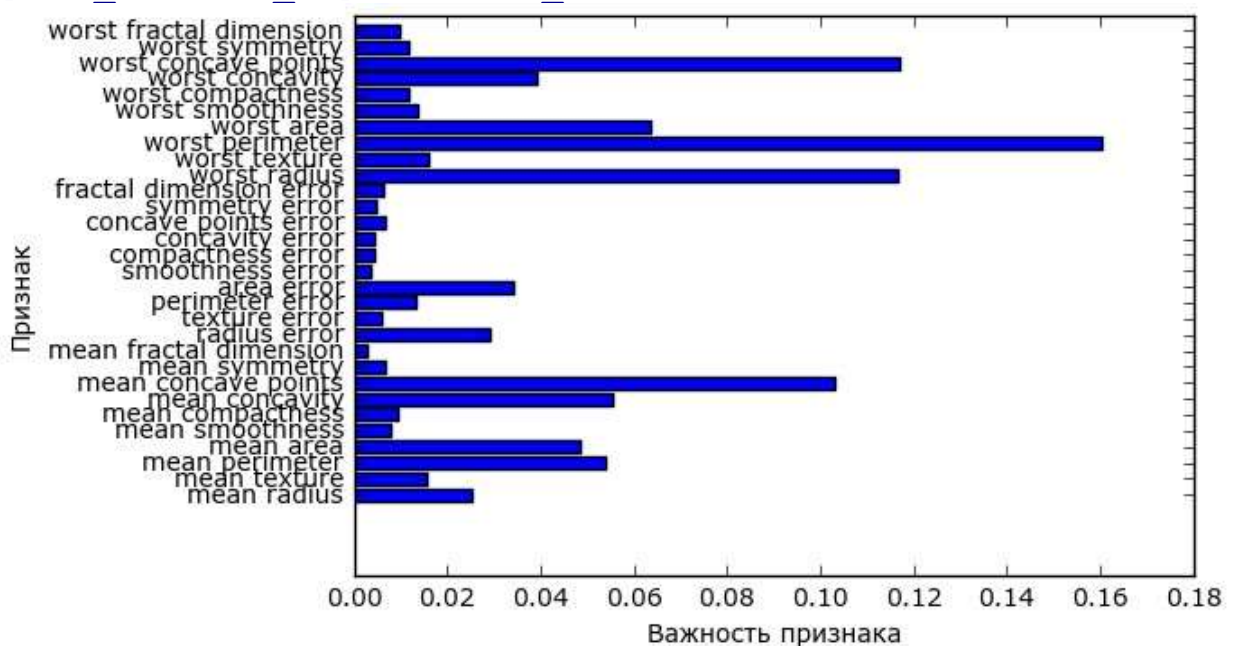


Рис. 2.34 - Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

На рисунке видно, что в отличие от одиночного дерева решения случайный лес вычисляет ненулевые значения важностей для гораздо большего числа признаков. Как и дерево решений, случайный лес также присваивает высокое значение важности признаку «worst radius», однако в качестве наиболее информативного признака выбирает «worst perimeter». Случайность, лежащая в основе случайного леса, заставляет алгоритм рассматривать множество возможных интерпретаций. Это приводит к тому, что случайный лес дает гораздо более широкую картину данных, чем одиночное дерево.

В настоящее время случайные леса регрессии и классификации являются одним из наиболее широко используемых методов машинного обучения. Они обладают высокой прогнозной силой, часто дают хорошее качество модели без утомительной настройки параметров и не требуют масштабирования данных.

По сути случайные леса обладают всеми преимуществами деревьев решений, хотя и не лишены некоторых их недостатков. Одна из причин, в силу которой деревья решений еще используются до сих пор, – это компактное представление процесса принятия решений. Детальная интерпретация десятков или сотен деревьев невозможна в принципе, и, как правило, деревья в случайном лесу получаются более глубокими по сравнению с одиночными деревьями решений (из-за использования подмножеств признаков). Поэтому, если вам нужно в сжатом виде визуализировать процесс принятия решений для неспециалистов, одиночное дерево решений может быть оптимальным выбором. Несмотря на то, что построение случайных лесов на больших наборах данных может занимать определенное время, его можно легко распараллелить между несколькими ядрами процессора в компьютере. Если ваш компьютер оснащен многоядерным процессором (как почти все современные компьютеры), вы можете использовать параметр `n_jobs` для настройки количества используемых ядер. Использование большего количества процессорных ядер приведет к линейному росту скорости (при использовании двух ядер обучение случайного леса будет осуществляться в два раза быстрее), однако установка значения `n_jobs`, превышающего количество ядер, не поможет. Вы можете установить `n_jobs=-1`, чтобы использовать все ядра вашего процессора.

Вы должны помнить, что случайный лес по своей природе является рандомизированным алгоритмом и установка различных стартовых значений генератора псевдослучайных чисел (или вообще отказ от использования `random_state`) может кардинально изменить построение модели. Чем больше деревьев в лесу, тем более устойчивым он будет к изменению стартового значения. Если вы хотите получить результаты, которые потом нужно будет воспроизвести, важно зафиксировать `random_state`.

Случайный лес плохо работает на данных очень высокой размерности, разреженных данных, например, на текстовых данных. Для подобного рода данных линейные модели подходят больше. Случайный лес, как правило, хорошо работает даже на очень больших наборах данных, и обучение могут легко распараллелить между многочисленными процессорными ядрами в рамках мощного компьютера. Однако случайный лес требует больше памяти и медленнее обучается и прогнозирует, чем линейные модели. Если время и память имеют важное значение, имеет смысл вместо случайного леса использовать линейную модель.

Важными параметрами настройки являются `n_estimators`, `max_features` и опции предварительной обрезки деревьев, например, `max_depth`. Что касается `n_estimators`, большее значение всегда дает лучший результат. Усреднение результатов по большому количеству деревьев позволит получить более устойчивый ансамбль за счет снижения переобучения. Однако обратная сторона увеличения числа деревьев заключается в том, что с ростом количества деревьев требуется больше памяти и больше времени для обучения. Общее правило заключается в том, чтобы построить «столько, сколько позволяет

ваше время/память».

Как было описано ранее, `max_features` случайным образом определяет признаки, использующиеся при разбиении в каждом дереве, а меньшее значение `max_features` уменьшает переобучение. В общем, лучше взять за правило использовать значения, выставленные по умолчанию: `max_features=sqrt(n_features)` для классификации и `max_features=n_features` для регрессии. Увеличение значений `max_features` или `max_leaf_nodes` иногда может повысить качество модели. Кроме того, оно может резко снизить требования к пространству на диске и времени вычислений в ходе обучения и прогнозирования.

6 ГРАДИЕНТНЫЙ БУСТИНГ ДЕРЕВЬЕВ РЕГРЕССИИ

Цель занятия: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для обработки данных при помощи алгоритма - градиентный бустинг деревьев регрессии.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Градиентный бустинг деревьев регрессии – еще один ансамблевый метод, который объединяет в себе множество деревьев для создания более мощной модели. Несмотря на слово «регрессия» в названии, эти модели можно использовать для регрессии и классификации. В отличие от случайного леса, градиентный бустинг строит последовательность деревьев, в которой каждое дерево пытается исправить ошибки предыдущего. По умолчанию в градиентном бустинге деревьев регрессии отсутствует случайность, вместо этого используется строгая предварительная обрезка. В градиентном бустинге деревьев часто используются деревья небольшой глубины, от одного до пяти уровней, что делает модель меньше с точки зрения памяти и ускоряет вычисление прогнозов.

Основная идея градиентного бустинга заключается в объединении множества простых моделей (в данном контексте известных под названием слабые ученики или *weak learners*), деревьев небольшой глубины. Каждое дерево может дать хорошие прогнозы только для части данных и таким образом для итеративного улучшения качества добавляется все большее количество деревьев.

Градиентный бустинг деревьев часто занимает первые строчки в соревнованиях по машинному обучению, а также широко используется в коммерческих сферах. В отличие от случайного леса он, как правило, немного более чувствителен к настройке параметров, однако при правильно заданных параметрах может дать более высокое значение правильности.

Помимо предварительной обрезки и числа деревьев в ансамбле, еще один важный параметр градиентного бустинга – это *learning_rate*, который контролирует, насколько сильно каждое дерево будет пытаться исправить ошибки предыдущих деревьев. Более высокая скорость обучения означает, что каждое дерево может внести более сильные корректировки и это позволяет получить более сложную модель. Добавление большего количества деревьев в ансамбль, осуществляемое за счет увеличения значения *n_estimators*, также увеличивает сложность модели, поскольку модель имеет больше шансов исправить ошибки на обучающем наборе.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Ниже приведен пример использования GradientBoostingClassifier на наборе данных Breast Cancer. По умолчанию используются 100 деревьев с максимальной глубиной 3 и скорости обучения 0.1:

```
In[72]:
from sklearn.ensemble import GradientBoostingClassifier
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("Правильность на обучающем наборе:
{:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(gbrt.score(X_test, y_test)))
Out[72]:
```

Правильность на обучающем наборе: 1.000

Правильность на тестовом наборе: 0.958

Поскольку правильность на обучающем наборе составляет 100%, мы, вероятно, столкнулись с переобучением. Для уменьшения переобучения мы можем либо применить более сильную предварительную обрезку, ограничив максимальную глубину, либо снизить скорость обучения:

```
In[73]: gbrt = GradientBoostingClassifi-
er(random_state=0, max_depth=1) gbrt.fit(X_train,
y_train)
print("Правильность на обучающем наборе:
{:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(gbrt.score(X_test, y_test)))
Out[73]:
```

Правильность на обучающем наборе: 0.991

Правильность на тестовом наборе: 0.972 In[74]: gbrt =

```
GradientBoostingClassifier(random_state=0, learn-
ing_rate=0.01) gbrt.fit(X_train, y_train)
```

```
print("Правильность на обучающем наборе:
{:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(gbrt.score(X_test, y_test)))
Out[74]:
```

Правильность на обучающем наборе: 0.988

Правильность на тестовом наборе: 0.965

Как и ожидалось, эти методы, направленные на уменьшение сложности модели, снижают правильность на обучающем наборе. В данном случае снижение максимальной глубины деревьев значительно улучшило модель, тогда как скорость обучения лишь незначительно повысило обобщающую способ-

ность.

И вновь, как и в случае с остальными моделями на основе деревьев, мы можем визуализировать важности признаков, чтобы получить более глубокое представление о нашей модели (рис. 2.35). Поскольку мы использовали 100 деревьев, вряд ли целесообразно проверять все деревья, даже если все они имеют глубину 1:

```
In[75]: gbrt = GradientBoostingClassifier(random_state=0, max_depth=1) gbrt.fit(X_train,
y_train)
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Важность признака")
    plt.ylabel("Признак")
plot_feature_importances_cancer(gbrt)
```

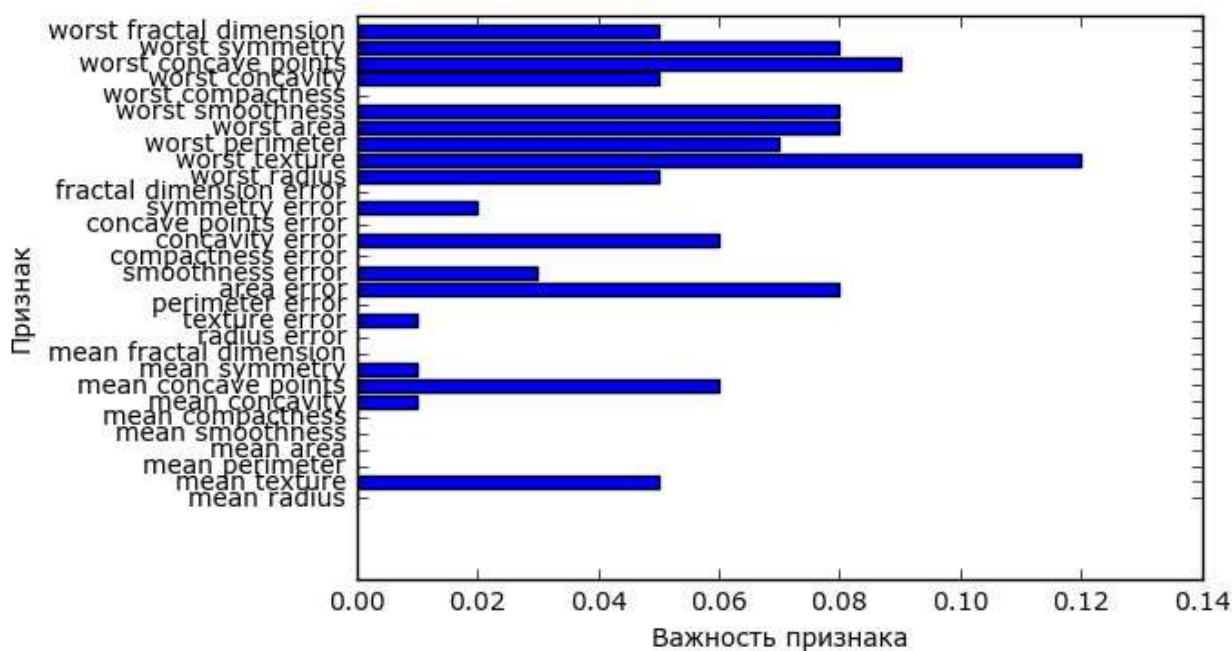


Рис. 2.35 - Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

На рисунке видно, что важности признаков, вычисленные градиентным бустингом деревьев, в какой-то степени схожи с важностями признаков, полученными с помощью случайного леса, хотя градиентный бустинг полностью проигнорировал некоторые признаки.

Поскольку и градиентный бустинг и случайный лес хорошо работают на одних и тех же данных, общераспространенный подход заключается в том, чтобы сначала попытаться построить случайный лес, который дает вполне устойчивые результаты. Если случайный лес дает хорошее качество

модели, однако время, отводимое на прогнозирование, на вес золота или важно выжать из модели максимальное значение правильности, выбор в пользу градиентного бустинга часто помогает решить эти задачи.

Если вы хотите применить градиентный бустинг для решения крупномасштабной задачи, возможно стоит обратиться к пакету `xgboost` и его Python-интерфейсу, который на многих наборах данных работает быстрее (а иногда и проще настраивается), чем реализация градиентного бустинга в `scikit-learn`.

Градиентный бустинг деревьев решений – одна из самых мощных и широко используемых моделей обучения с учителем. Его основной недостаток заключается в том, что он требует тщательной настройки параметров и для обучения может потребоваться много времени. Как и другие модели на основе дерева, алгоритм хорошо работает на данных, представляющих смесь бинарных и непрерывных признаков, не требуя масштабирования. Как и остальные модели на основе дерева, он также плохо работает на высокоразмерных разреженных данных.

Основные параметры градиентного бустинга деревьев – это количество деревьев (`n_estimators`) и скорость обучения (`learning_rate`), контролирующая степень вклада каждого дерева в устранение ошибок предыдущих деревьев. Эти два параметра тесно взаимосвязаны между собой, поскольку более низкое значение `learning_rate` означает, что для построения модели аналогичной сложности необходимо большее количество деревьев. В отличие от случайного леса, в котором более высокое значение `n_estimators` всегда дает лучшее качество, увеличение значения `n_estimators` в градиентном бустинге дает более сложную модель, что может привести к переобучению. Общепринятая практика – подгонять `n_estimators` в зависимости от бюджета времени и памяти, а затем подбирать различные значения `learning_rate`.

Другим важным параметром является параметр `max_depth` (или, как альтернатива, `max_leaf_nodes`), направленный на уменьшение сложности каждого дерева. Обычно для моделей градиентного бустинга значение `max_depth` устанавливается очень низким, как правило, не больше пяти уровней.

7 НЕЙРОННЫЕ СЕТИ

Цель занятия: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для обработки данных при помощи алгоритма – нейронные сети.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Семейство алгоритмов, известное как нейронные сети, недавно пережило свое возрождение под названием «глубокое обучение». Несмотря на то что глубокое обучение сулит большие перспективы в различных сферах применения машинного обучения, алгоритмы глубокого обучения, как правило, жестко привязаны к конкретным случаям использования. В данном разделе мы рассмотрим лишь некоторые относительно простые методы, а именно многослойные персептроны для классификации и регрессии, которые могут служить отправной точкой в изучении более сложных методов машинного обучения. Многослойные персептроны (MLP) также называют простыми (vanilla) нейронными сетями прямого распространения, а иногда и просто нейронными сетями.

MLP можно рассматривать как обобщение линейных моделей, которое прежде чем прийти к решению выполняет несколько этапов обработки данных.

Вспомним, что в линейной регрессии прогноз получают с помощью следующей формулы:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b$$

Говоря простым языком, $\hat{y}$ – это взвешенная сумма входных признаков $x[0] \dots x[p]$. Входные признаки взвешены по вычисленным в ходе обучения коэффициентам $w[0] \dots w[p]$. Мы можем представить их графически, как показано на рис. 2.44:

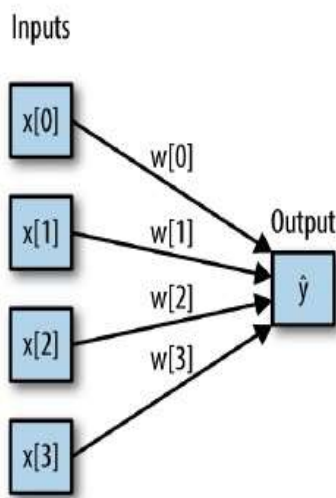


Рис. 2.44 - Визуализация логистической регрессии, в которой входные признаки и прогнозы показаны в виде узлов, а коэффициенты – в виде

соединений между узлами

```
In[89]:  
dis-  
play(mglearn.plots.plot_logistic_regression_graph()  
)
```

Здесь каждый узел, показанный слева, представляет собой входной признак, соединительные линии – коэффициенты, а узел справа – это выход, который является взвешенной суммой входов.

В MLP процесс вычисления взвешенных сумм повторяется несколько раз. Сначала вычисляются скрытые элементы (hidden units), которые представляют собой промежуточный этап обработки. Они вновь объединяются с помощью взвешенных сумм для получения конечного результата (рис. 2.45):

```
In[90]:  
display(mglearn.plots.plot_single_hidden_layer_graph())
```

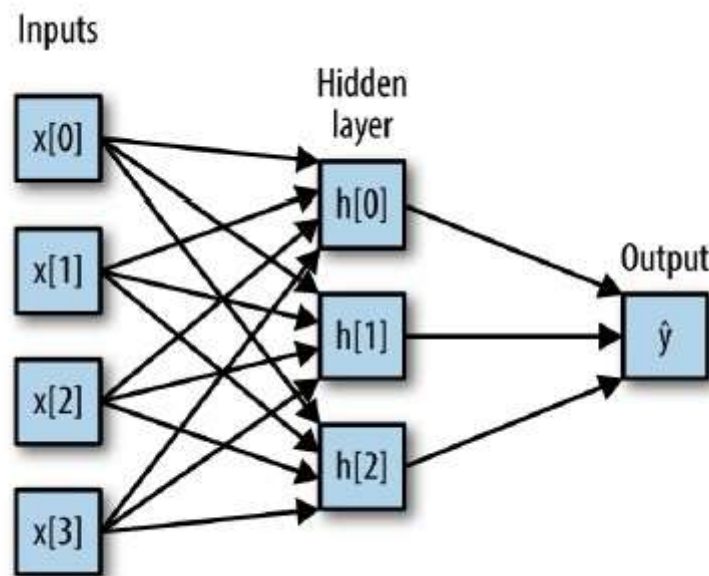


Рис. 2.45 - Иллюстрация многослойного персептрона с одним скрытым слоем

У этой модели гораздо больше вычисляемых коэффициентов (также называемых весами): коэффициент между каждым входом и каждым скрытым элементом (которые образуют скрытый слой или hidden layer) и коэффициент между каждым элементом скрытого слоя и выходом. С математической точки зрения вычисление серии взвешенных сумм – это то же самое, что вычисление лишь одной взвешенной суммы, таким образом, чтобы эта модель обладала более мощной прогнозной силой, чем линейная модель, нам нужен один дополнительный трюк. Поясним его на примере нейронной сети с одним скрытым слоем. Входной слой просто передает входы скрытому слою сети, либо без преобразования, либо выполнив сначала стандартизацию входов. Затем происходит вычисление взвешенной суммы входов для каждого элемента скрытого слоя, к ней применяется функция активации – обычно

используются нелинейные функции выпрямленный линейный элемент (rectified linear unit или relu) или гиперболический тангенс (hyperbolic tangent или tanh). В итоге получаем выходы нейронов скрытого слоя. Эти промежуточные выходы могут считаться нелинейными преобразованиями и комбинациями первоначальных входов. Они становятся входами выходного слоя. Снова вычисляем взвешенную сумму входов, применяем функцию активации и получаем итоговые значения целевой переменной. Функции активации relu и tanh показаны на рис. 2.46. Relu отсекает значения ниже нуля, в то время как tanh принимает значения от -1 до 1 (соответственно для минимального и максимального значений входов). Любая из этих двух нелинейных функций позволяет нейронной сети в отличие от линейной модели вычислять гораздо более сложные зависимости.

```
In[91]: line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best") plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```

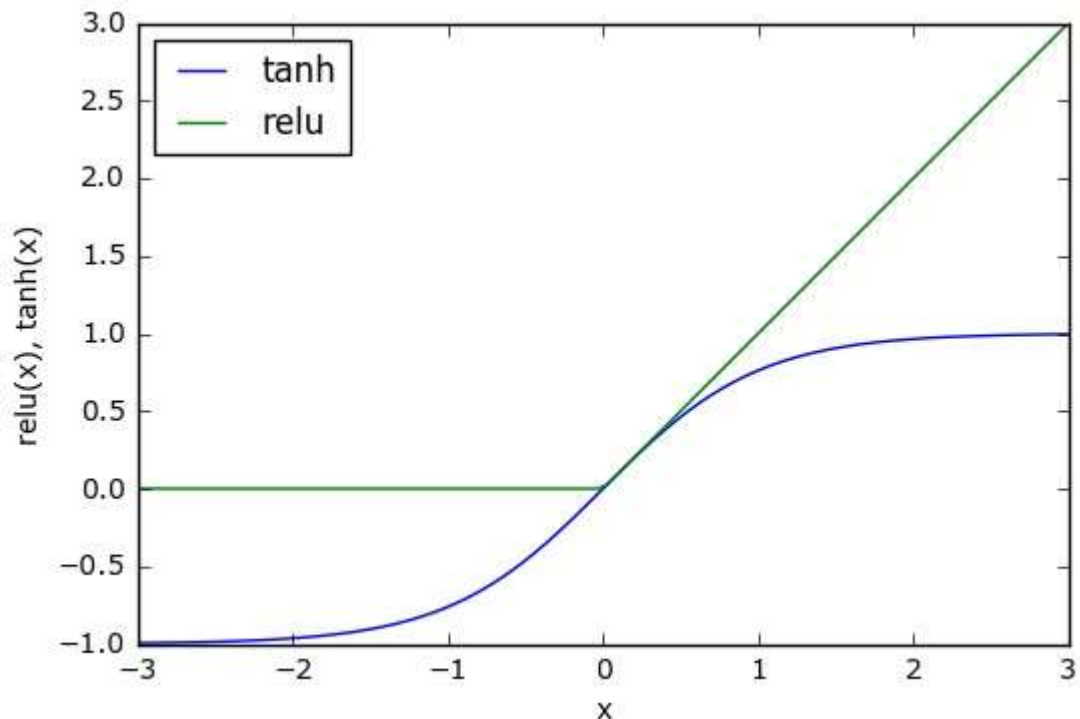


Рис. 2.46 - Функция активации гиперболический тангенс и функция активации выпрямленного линейного элемента

Для небольшой нейронной сети, изображенной на рис. 2.45, полная формула вычисления y в случае регрессии будет выглядеть так (при использовании tanh):

$$\begin{aligned}
h[0] &= \tanh(w[0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3]) \\
h[1] &= \tanh(w[0,1] * x[0] + w[1,1] * x[1] + w[2,1] * x[2] + w[3,1] * x[3]) \\
h[2] &= \tanh(w[0,2] * x[0] + w[1,2] * x[1] + w[2,2] * x[2] + w[3,2] * x[3]) \\
\hat{y} &= v[0] * h[0] + v[1] * h[1] + v[2] * h[2]
\end{aligned}$$

Здесь w – веса между входом x и скрытым слоем h , а v – весовые коэффициенты между скрытым слоем h и выходом y . Веса v и w вычисляются по данным, x являются входными признаками, y – вычисленный выход, а h – промежуточные вычисления. Важный параметр, который должен задать пользователь – количество узлов в скрытом слое. Его значение может быть маленьким, например, 10 для очень маленьких или простых наборов данных или же большим, например, 10000 для очень сложных данных. Кроме того, можно добавить дополнительные скрытые слои, как показано на рис. 2.47:

In [92] :

```
mglearn.plots.plot_two_hidden_layer_graph()
```

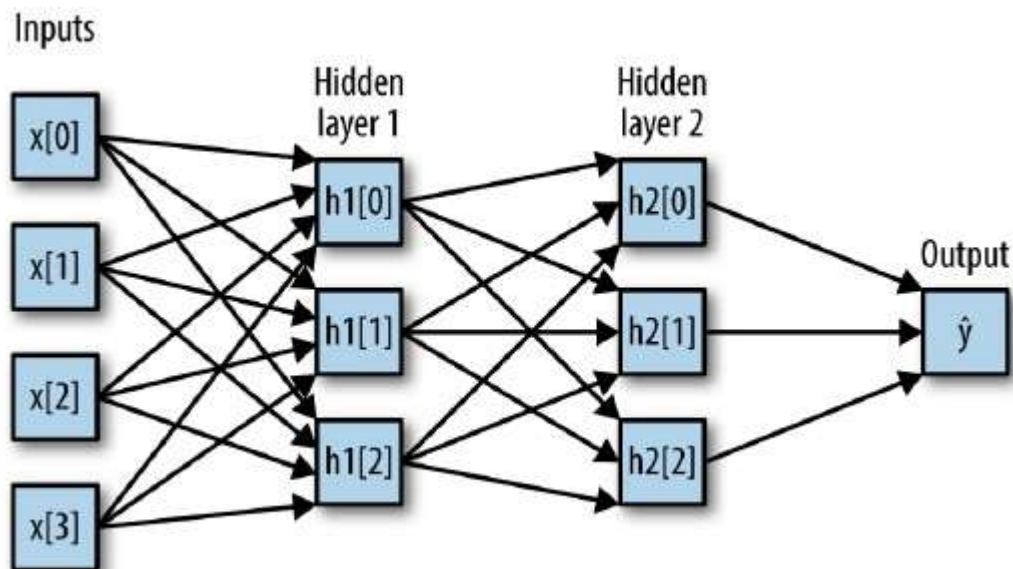


Рис. 2.47 - Многослойный персептрон с двумя скрытыми слоями

Построение больших нейронных сетей, состоящих из множества слоев вычислений, вдохновило специалистов ввести в обиход термин «глубокое обучение» («deep learning»).

ПРАКТИЧЕСКАЯ ЧАСТЬ

Давайте посмотрим, как работает MLP, применив MLPClassifier к набору данных two_moons, который мы использовали ранее в этой главе. Результаты показаны на рис. 2.48:

In [93] :

```
from sklearn.neural_network import MLPClassifier from
sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, ran-
dom_state=3)
```

```

X_train, X_test, y_train,
y_test = train_test_split(X, y, stratify=y,
random_state=42)
mlp = MLPClassifier(solver='lbfgs',
random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True,
alpha=.3) mglearn.discrete_scatter(X_train[:, 0],
X_train[:, 1], y_train) plt.xlabel("Признак 0")
plt.ylabel("Признак 1")

```

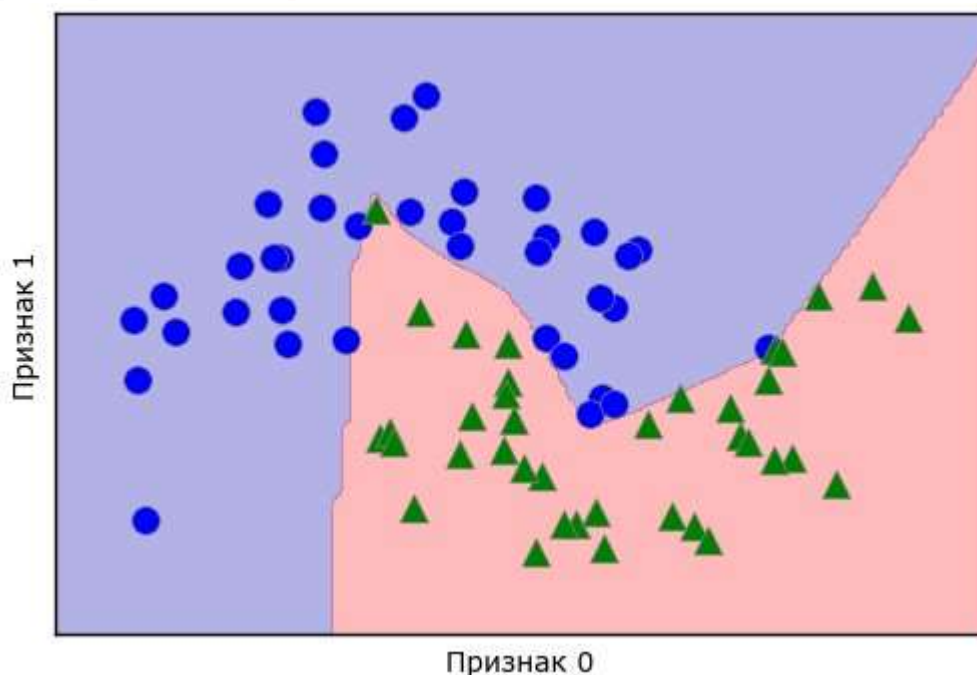


Рис. 2.48 - Граница принятия решений, построенная нейронной сетью со 100 скрытыми элементами на наборе данных two_moons

Как видно из рис., нейронная сеть построила нелинейную, но относительно гладкую границу принятия решений. Мы использовали `solver='lbfgs'`, который рассмотрим позднее.

По умолчанию MLP использует 100 скрытых узлов, что довольно много для этого небольшого набора данных. Мы можем уменьшить число (что снизит сложность модели) и снова получить хороший результат (рис. 2.49):

In[94]:

```

mlp = MLPClassifier(solver='lbfgs', random_state=0,
hidden_layer_sizes=[10]) mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train,
fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:,
1], y_train) plt.xlabel("Признак 0")
plt.ylabel("Признак 1")

```

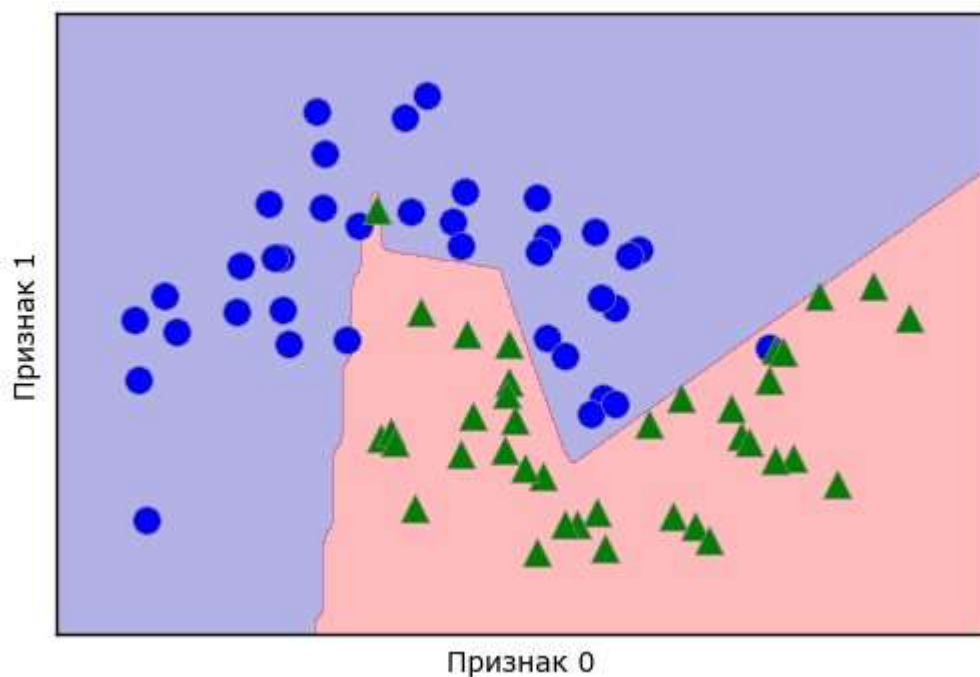


Рис. 2.49 - Граница принятия решений, построенная нейронной сетью с 10 скрытыми элементами на наборе данных two_moons

При использовании лишь 10 скрытых элементов граница принятия решений становится более неровной. По умолчанию используется функция активации `relu`, показанная на рис. 2.46. При использовании одного скрытого слоя решающая функция будет состоять из 10 прямолинейных отрезков. Если необходимо получить более гладкую решающую границу, можно добавить большее количество скрытых элементов (как показано на рис. 2.49), добавить второй скрытый слой (рис. 2.50), или использовать функцию активации `tanh` (рис. 2.51):

```
In[95]:
# использование двух скрытых слоев по 10 элементов
в каждом
mlp = MLPClassifier(solver='lbfgs', random_state=0,
hidden_layer_sizes=[10, 10]) mlp.fit(X_train,
y_train)
mglearn.plots.plot_2d_separator(mlp, X_train,
fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0],
X_train[:, 1], y_train)
plt.xlabel("Признак 0") plt.ylabel("Признак 1")
```

```
In[96]:
# использование двух скрытых слоев по 10 элементов
в каждом, на этот раз с функцией tanh
mlp = MLPClassifier(solver='lbfgs',
```

```

activation='tanh', random_state=0,
hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train,
fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0],
X_train[:, 1], y_train) plt.xlabel("Признак 0")
plt.ylabel("Признак 1")

```

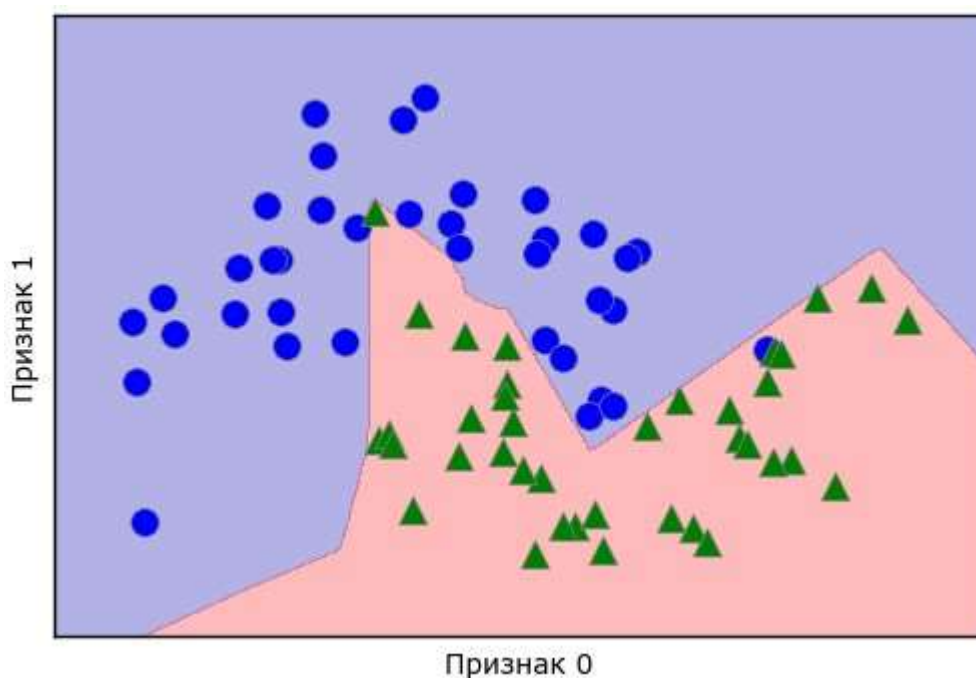


Рис. 2.50 - Граница принятия решений, построенная нейронной сетью с 2 скрытыми слоями по 10 элементов в каждом и функцией активации выпрямленный линейный элемент

И, наконец, мы можем дополнительно настроить сложность нейронной сети с помощью l2 штрафа, чтобы сжать весовые коэффициенты до близких к нулю значений, как мы это делали в гребневой регрессии и линейных классификаторов. В MLPClassifier за это отвечает параметр `alpha` (как и в моделях линейной регрессии), и по умолчанию установлено очень низкое значение (небольшая регуляризация). На рис. 2.52 показаны результаты применения к набору данных `two_moons` различных значений `alpha` с использованием двух скрытых слоев с 10 или 100 элементами в каждом:

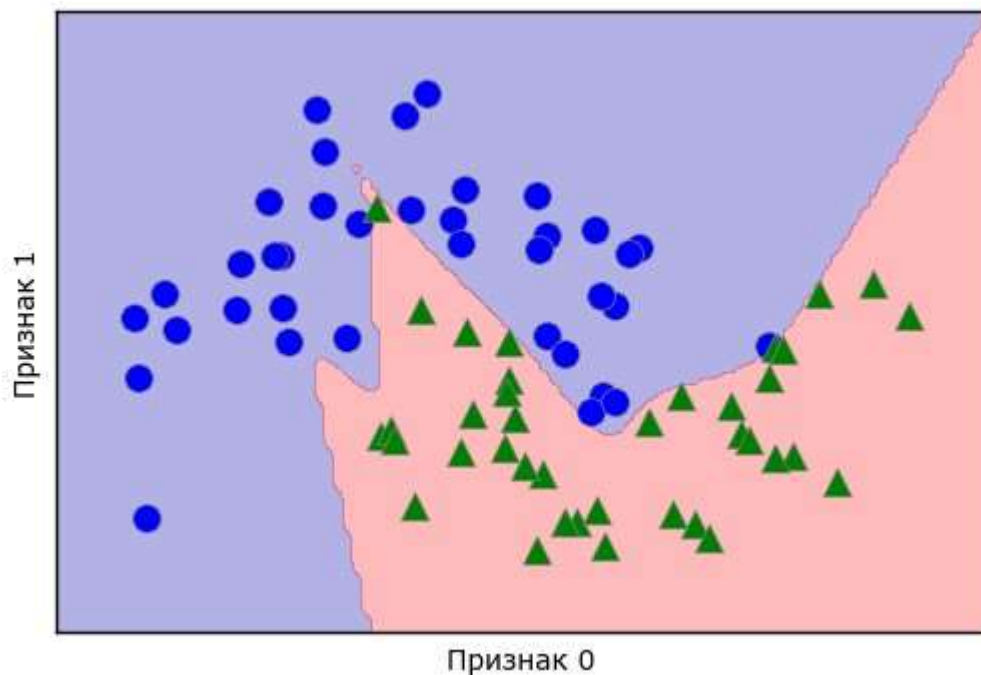


Рис. 2.51 Граница принятия решений, построенная нейронной сетью с 2 скрытыми слоями по 10 элементов в каждом и функцией активации гиперболический тангенс

```
In[97]:
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
            hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True,
            alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1],
            y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
            n_hidden_nodes, n_hidden_nodes, alpha))
```

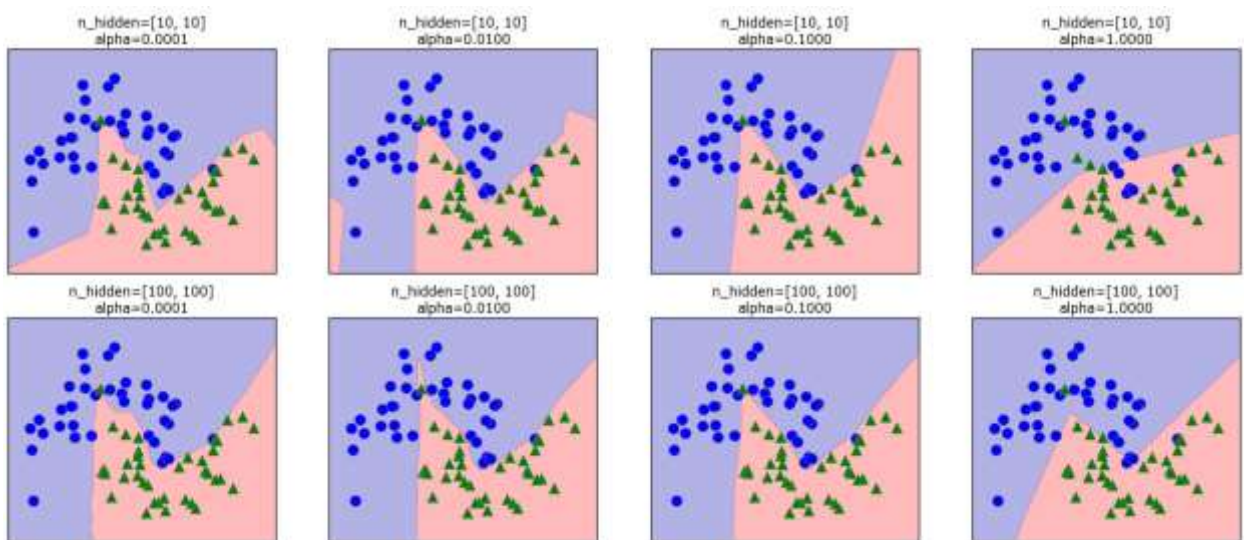


Рис. 2.52 - Границы принятия решений для различного количества скрытых элементов и разных значений параметра α

Как вы, наверное, уже поняли, существуют различные способы регулировать сложность нейронной сети: количество скрытых слоев, количество элементов в каждом скрытом слое и регуляризация (α). На самом деле их гораздо больше, но мы не будем здесь вдаваться в подробности.

Важным свойством нейронных сетей является то, что их веса задаются случайным образом перед началом обучения и случайная инициализация влияет на процесс обучения модели. Это означает, что даже при использовании одних и тех же параметров мы можем получить очень разные модели, задавая разные стартовые значения генератора псевдослучайных чисел. При условии, что сеть имеет большой размер и ее сложность настроена правильно, данный факт не должен сильно влиять на правильность, однако о нем стоит помнить (особенно при работе с небольшими сетями). На рис. 2-53 представлены графики нескольких моделей, обученных с использованием тех же самых значений параметров:

```
In[98]:
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True,
                                    alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0],
                              X_train[:, 1], y_train, ax=ax)
```

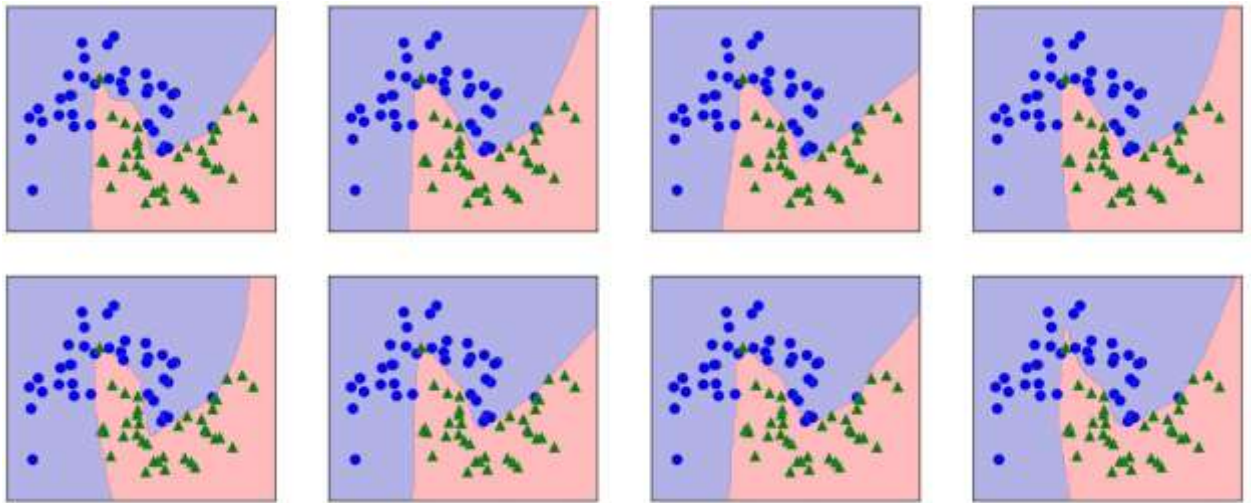


Рис. 2.53 - Границы принятия решений, полученные с использованием тех же самых параметров, но разных стартовых значений

Чтобы лучше понять, как нейронная сеть работает на реальных данных, давайте применим `MLPClassifier` к набору данных `Breast Cancer`. Мы начнем с параметров по умолчанию:

```
In[99]:
print("Максимальные значения характеристик:\n{}".
      format(cancer.data.max(axis=0)))
Out[99]:
Максимальные значения характеристик:
[28.110  39.280 188.500 2501.00  0.163    0.345    0.427
 0.201  0.304  0.097  2.873  4.885  21.980 542.200    0.031
 0.135  0.396  0.053  0.079  0.030  36.040  49.540 251.200
4254.0  0.223  1.058  1.252  0.291  0.664   0.207]
```

```
In[100]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
mlp = MLPClassifier(random_state=42) mlp.fit(X_train,
y_train)

print("Правильность на обучающем наборе:
{:.2f}".format(mlp.score(X_train, y_train)))
print("Правильности на тестовом наборе:
{:.2f}".format(mlp.score(X_test, y_test)))
Out[100]:
Правильность на обучающем наборе: 0.92
Правильность на тестовом наборе: 0.90
```

MLP демонстрирует довольно неплохую правильность, однако не столь хорошую, если сравнивать с другими моделями. Как и в предыдущем примере с SVC, это, вероятно, обусловлено масштабом данных. Нейронные сети также требуют того, чтобы все входные признаки были измерены в одном и том же масштабе, в идеале они должны иметь среднее 0 и дисперсию 1. Мы должны отмасштабировать наши данные так, чтобы они отвечали этим требованиям. Опять же, мы будем делать это вручную, однако в главе 3 расскажем, как это делать автоматически с помощью `StandardScaler`.

```
In[101]:
# вычисляем среднее для каждого признака обучающего
набора
mean_on_train = X_train.mean(axis=0)
# вычисляем стандартное отклонение для каждого признака
обучающего набора
std_on_train = X_train.std(axis=0)
# вычитаем среднее и затем умножаем на обратную величину
стандартного отклонения
# mean=0 и std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# используем ТО ЖЕ САМОЕ преобразование (используем сред-
нее и стандартное отклонение
# обучающего набора) для тестового набора
X_test_scaled = (X_test - mean_on_train) / std_on_train
mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".
      format(mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".
      format(mlp.score(X_test_scaled, y_test)))
Out[101]:
Правильность на обучающем наборе: 0.991
Правильность на тестовом наборе: 0.965

ConvergenceWarning:
Stochastic Optimizer: Maximum iterations reached and the
optimization      hasn't converged yet.
```

После масштабирования результаты стали намного лучше и теперь уже вполне могут конкурировать с результатами остальных моделей. Впрочем, мы получили предупреждение о том, что достигнуто максимальное число итераций. Оно является неотъемлемой частью алгоритма `adam` и сообщает нам о том, что мы должны увеличить число итераций:

```
In[102]:
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Правильность на обучающем наборе:
{:.3f}".format(mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе:
{:.3f}".format(mlp.score(X_test_scaled, y_test)))
Out[102]:
Правильность на обучающем наборе: 0.995
Правильность на тестовом наборе: 0.965
```

Увеличение количества итераций повысило правильность лишь на обучающем наборе. Тем не менее модель имеет достаточно высокое значение правильности. Поскольку существует определенный разрыв между правильностью на обучающем наборе и правильностью на тестовом наборе, мы можем попытаться уменьшить сложность модели, чтобы улучшить обобщающую способность. В данном случае мы увеличим параметр `alpha` (довольно сильно с 0.0001 до 1), чтобы применить к весам более строгую регуляризацию:

```
In[103]: mlp = MLPClassifier(max_iter=1000, alpha=1,
random_state=0) mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".
format(mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".
format(mlp.score(X_test_scaled, y_test)))
Out[103]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.972
```

Это дает правильность, сопоставимую с правильностью лучших моделей.

Несмотря на то что анализ нейронной сети возможен, он, как правило, гораздо сложнее анализа линейной модели или модели на основе дерева. Один из способов анализа нейронной сети заключается в том, чтобы исследовать веса модели. Образец такого анализа вы можете увидеть в галерее примеров `scikit-learn`. Применительно к набору данных `Breast Cancer` такой анализ может быть немного сложен. Следующий график (рис. 2.54) показывает весовые коэффициенты, которые были вычислены при подключении входного слоя к первому скрытому слою. Строки в этом графике соответствуют 30 входным признакам, а столбцы – 100 скрытым элементам. Светлые цвета соответствуют высоким положительным значениям, в то время как темные цвета соответствуют отри-

цательным значениям:

```
In[104]: plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none',
cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Столбцы матрицы весов")
plt.ylabel("Входная характеристика")
plt.colorbar()
```

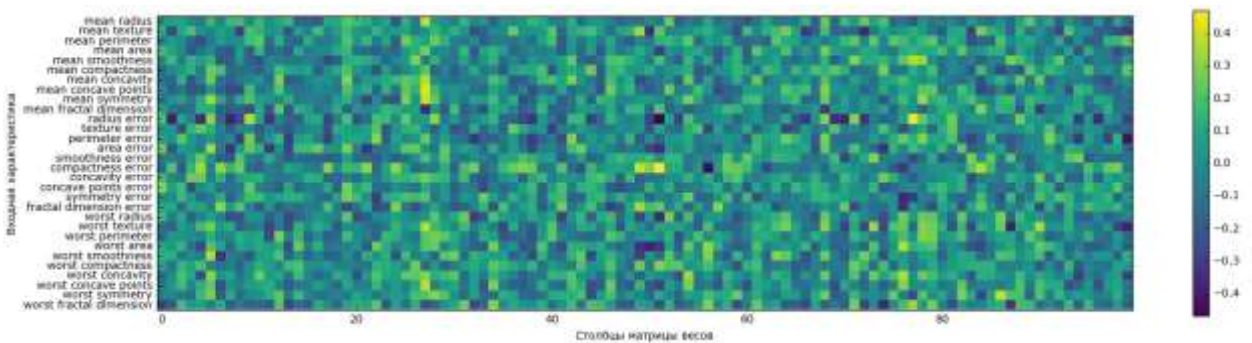


Рис. 2.54 - Теплокарта для весов первого слоя нейронной сети, обученной на наборе данных Breast Cancer

Один из возможных выводов, который мы можем сделать, заключается в том, что признаки с небольшими весами скрытых элементов «менее важны» в модели. Мы можем увидеть, что «mean smoothness» и «mean compactness» наряду с признаками, расположенными между «smoothness error» и «fractal dimension error», имеют относительно низкие веса по сравнению с другими признаками. Это может означать, что эти признаки являются менее важными или, возможно, мы не преобразовали их таким способом, чтобы их могла использовать нейронная сеть.

Кроме того, мы можем визуализировать веса, соединяющие скрытый слой с выходным слоем, но их еще труднее интерпретировать.

Несмотря на то что для наиболее распространенных архитектур нейронных сетей MLPClassifier и MLPRegressor предлагают легкий в использовании интерфейс, они представляют лишь небольшой набор возможных средств, позволяющих строить нейронные сети. Если вас интересует работа с более гибкими или более масштабными моделями, мы рекомендуем вам не ограничиваться возможностями библиотеки scikit-learn и обратиться к фантастическим по своим возможностям библиотекам глубокого обучения. Для пользователей Python наиболее устоявшимися являются keras, lasagna и tensorflow. lasagna построена на основе библиотеки theano, тогда как keras может использовать либо tensorflow, либо theano. Эти библиотеки предлагают гораздо более гибкий интерфейс для построения нейронных сетей и обновляются в соответствии с последними достижениями в области глубокого обучения. Кроме того, все популярные библиотеки глубокого обучения позволяют использовать высокопроизводительные графические процессоры

(GPU), которые в `scikit-learn` не поддерживаются. Использование графических процессоров позволяет ускорить вычисления от 10 до 100 раз, и они имеют важное значение для применения методов глубокого обучения для крупномасштабных наборов данных.

Нейронные сети вновь «вышли на сцену» во многих сферах применения машинного обучения в качестве передовых методов. Одно из их главных преимуществ заключается в том, что они способны обрабатывать информацию, содержащуюся в больших объемах данных, и строить невероятно сложные модели. При наличии достаточного времени вычислений, данных и тщательной настройки параметров нейронные сети часто превосходят другие алгоритмы машинного обучения (для задач классификации и регрессии).

Это дает и свои минусы. Нейронные сети, особенно крупные нейронные сети, как правило, требуют длительного времени обучения. Как мы видели здесь, они также требуют тщательной предварительной обработки данных. Аналогично SVM, нейронные сети лучше всего работают с «однородными» данными, где все признаки измерены в одном и том же масштабе. Что касается данных, в которых признаки имеют разный масштаб, модели на основе дерева могут дать лучший результат. Кроме того, настройка параметров нейронной сети – это само по себе искусство. В наших экспериментах мы едва коснулись возможных способов настройки и обучения нейросетевых моделей.

8 РАЗБИЕНИЕ ДАННЫХ НА ВЫБОРКИ

Цель занятия: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для разбиения на выборки.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Неконтролируемые преобразования (unsupervised transformations) – это алгоритмы, создающие новое представление данных, которое в отличие от исходного представления человеку или алгоритму машинного обучения будет обработать легче. Общераспространенное применение неконтролируемых преобразований – сокращение размерности. Мы берем высокоразмерное представление данных, состоящее из множества признаков, и находим новый способ представления этих данных, обобщая основные характеристики и получая меньшее количество признаков. Общераспространенное применение сокращения размерности – получение двумерного пространства в целях визуализации.

Еще одно применение неконтролируемых преобразований – поиск компонент, из которых «состоят» данные. Примером такого преобразования является выделение тем из коллекций текстовых документов. Здесь задача состоит в том, чтобы найти неизвестные темы, обсуждаемые в коллекции документов, а также выяснить, какие темы встречаются в каждом документе. Это может быть полезно для отслеживания в социальных сетях обсуждений таких тем, как выборы, контроль огнестрельного оружия или жизнь поп-звезд.

С другой стороны, алгоритмы кластеризации (clustering algorithms) разбивают данные на отдельные группы схожих между собой элементов. Рассмотрим пример загрузки фотографий в социальной сети. Часто вы формируете запросы типа «покажите мне все фотографии, на которых изображен Иван Петров». Для выполнения подобных запросов, администрация сайта, возможно, захочет сгруппировать фотографии, на которых изображен один и тот же человек. Однако при этом неизвестно, на каких загружаемых фотографиях кто показан, и неизвестно, какое количество различных пользователей присутствует на ваших фотографиях. Разумный подход заключался бы в том, чтобы извлечь все лица и разделить их на группы лиц, которые схожи между собой. Будем надеяться, что они соответствуют одному и тому же человеку и изображения в сгруппированном виде будут предъявлены вам.

Главная проблема машинного обучения без учителя – оценка полезности информации, извлеченной алгоритмом. Алгоритмы машинного обучения без учителя, как правило, применяются к данным, которые не содержат никаких меток, таким образом, мы не знаем, каким должен быть правильный ответ. Поэтому очень трудно судить о качестве работы модели. Например, наш гипотетический алгоритм кластеризации мог бы сгруппировать вместе все фотографии лиц в профиль и все фотографии лиц в анфас. Перед нами, несо-

мненно, один из способов разбить коллекцию фотографий лиц на группы, но это совсем не то, что нам нужно. Тем не менее у нас нет никакой возможности «рассказать» алгоритму, что мы ищем, и часто единственный способ оценить результат работы алгоритма машинного обучения без учителя – ручная проверка этого результата.

Как следствие, алгоритмы машинного обучения без учителя часто используются в разведочных целях, когда специалист хочет лучше изучить сами данные. Еще одно общераспространенное применение алгоритмов машинного обучения без учителя заключается в том, что они служат этапом предварительной обработки данных для алгоритмов машинного обучения с учителем. Изучение нового представления данных иногда может повысить правильность алгоритмов машинного обучения с учителем или может привести к снижению времени вычислений и потребления объема памяти.

Прежде чем начать знакомство с «реальными» алгоритмами машинного обучения без учителя, мы кратко рассмотрим некоторые простые методы предварительной обработки данных, которые часто могут пригодиться. Хотя предварительная обработка данных и масштабирование часто применяются вместе с алгоритмами контролируемого обучения, методы масштабирования не используют учителя, что делает их методами неконтролируемого обучения.

В предыдущих практиках мы видели, что некоторые алгоритмы, например, нейронные сети и SVM, очень чувствительны к масштабированию данных. Поэтому обычной практикой является преобразование признаков с тем, чтобы итоговое представление данных было более подходящим для использования вышеупомянутых алгоритмов. Часто достаточно простого масштабирования признаков и корректировки данных. Программный код (рис. 3.1) показывает простой пример:

```
In[2]: mglearn.plots.plot_scaling()
```

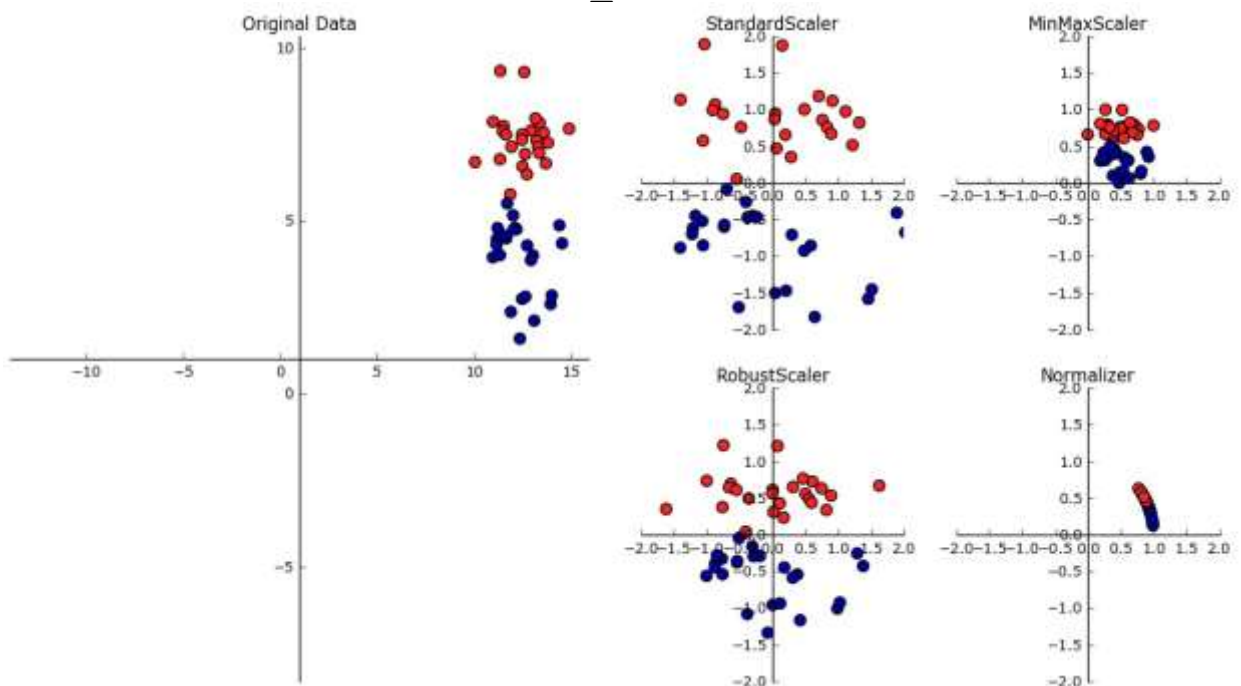


Рис. 3.1 - Различные способы масштабирования и предварительной обработки данных

ПРАКТИЧЕСКАЯ ЧАСТЬ

Первый график на рис. 3.1 соответствует синтетическому двухклассовому набору данных с двумя признаками. Первый признак (ось x) принимает значения в диапазоне от 10 до 15. Второй признак (ось y) принимает значения примерно в диапазоне от 1 до 9.

Следующие четыре графика показывают четыре различных способа преобразования данных, которые дают более стандартные диапазоны значений. Применение `StandardScaler` в `scikit-learn` гарантирует, что для каждого признака среднее будет равно 0, а дисперсия будет равна 1, в результате чего все признаки будут иметь один и тот же масштаб. Однако это масштабирование не гарантирует получение каких-то конкретных минимальных и максимальных значений признаков.

`RobustScaler` аналогичен `StandardScaler` в том плане, что в результате его применения признаки будут иметь один и тот же масштаб. Однако `RobustScaler` вместо среднего и дисперсии использует медиану и квартили. Это позволяет `RobustScaler` игнорировать точки данных, которые сильно отличаются от остальных (например, ошибки измерений). Эти странные точки данных еще называются выбросами (`outliers`) и могут стать проблемой для остальных методов масштабирования.

С другой стороны, `MinMaxScaler` сдвигает данные таким образом, что все признаки находились строго в диапазоне от 0 до 1. Для двумерного набора данных это означает, что все данные помещаются в прямоугольник, образованный осью x с диапазоном значений от 0 и 1 и осью y с диапазоном значений от 0 и 1.

И, наконец, `Normalizer` осуществляет совершенно иной вид масштабирования. Он масштабирует каждую точку данных таким образом, чтобы вектор признаков имел евклидову длину 1. Другими словами, он проецирует точку данных на окружность с радиусом 1 (или сферу в случае большого числа измерений). Вектор умножается на инверсию своей длины. Подобная нормализация используется тогда, когда важным является направление (но не длина) вектора признаков.

Теперь, когда мы увидели, что делают различные виды преобразований, давайте применим их, воспользовавшись `scikit-learn`. Мы будем использовать набор данных `cancer`. Методы предварительной обработки обычно применяются перед использованием алгоритма машинного обучения с учителем. Допустим, в качестве примера нам нужно применить ядерный SVM (`SVC`) к набору данных `cancer` и использовать `MinMaxScaler` для предварительной обработки данных. Мы начнем с того, что загрузим наш набор данных и разобьем его на тренировочный и тестовый наборы (обучающий и тестовый наборы нам нужны для оценки качества модели, которую мы построим с помощью алгоритма контролируемого обучения после предварительной обработки):

```

In[3]:
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import
train_test_split cancer = load_breast_cancer()

X_train, X_test, y_train, y_test =
train_test_split(cancer.data, cancer.target,
random_state=1) print(X_train.shape)
print(X_test.shape)
Out[3]:
(426, 30)
(143, 30)

```

Напомним, что набор содержит 569 точек данных, каждая из которых представлена 30 признаками. Мы разбиваем набор данных на 426 примеров в обучающей выборке и 143 примера в тестовой выборке.

Как и в случае с моделями контролируемого обучения, построенными ранее, мы сначала импортируем класс, который осуществляет предварительную обработку, а затем создаем его экземпляр:

```

In[4]: from sklearn.preprocessing import
MinMaxScaler
scaler = MinMaxScaler()

```

Затем с помощью метода `fit` мы подгоняем `scaler` на обучающих данных. Для `MinMaxScaler` метод `fit` вычисляет минимальное и максимальное значения каждого признака на обучающем наборе. В отличие от классификаторов и регрессоров, описанных в главе 2, при вызове метода `fit` `scaler` работает с данными (`X_train`), а ответы (`y_train`) не используются:

```

In[5]: scaler.fit(X_train)
Out[5]:
MinMaxScaler(copy=True, feature_range=(0, 1))

```

Чтобы применить преобразование, которое мы только что подогнали, то есть фактически отмасштабировать (`scale`) обучающие данные, мы воспользуемся методом `transform`. Метод `transform` используется в `scikit-learn`, когда модель возвращает новое представление данных:

```

In[6]:
# преобразовываем данные
X_train_scaled = scaler.transform(X_train)
# печатаем значения признаков в обучающем наборе до и

```

после масштабирования

```
print("форма преобразованного массива: {}".format(X_train_scaled.shape))
print("min значение признака до масштабирования:\n {}".format(X_train.min(axis=0)))
print("max значение признака до масштабирования:\n {}".format(X_train.max(axis=0)))
print("min знач-ние признака после масшт-ния:\n {}".format(X_train_scaled.min(axis=0)))
print("max значение признака после масшт-ния:\n {}".format(X_train_scaled.max(axis=0)))
Out[6]:
форма преобразованного массива:
(426, 30) min значение признака до масштабирования:
[6.98 9.71 43.79 143.50 0.05 0.02 0. 0. 0.11 0.05 0.12
 0.36 0.76 6.80 0. 0. 0. 0. 0.0 0. 7.93 12.02 50.41
 185.20 0.07 0.03 0. 0. 0.16 0.06]
max значение признака до масштабирования:
[28.11 39.28 188.5 2501.0 0.16 0.29 0.43 0.2 0.300 0.100
 2.87 4.88 21.98 542.20 0.03 0.14 0.400 0.050 0.06 0.03
 36.04 49.54 251.20 4254.00 0.220 0.940 1.17 0.29 0.58
 0.15]
min значение признака после масштабирования:
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
max значение признака после масштабирования:
[ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Преобразованные данные имеют такую же форму, что и исходные данные – признаки просто смещены и масштабированы. Видно, что теперь все признаки принимают значения в диапазоне от 0 до 1, как нам и требовалось.

Чтобы применить SVM к масштабированным данным, мы должны преобразовать еще тестовый набор. Это снова делается с помощью вызова метода `transform`, на этот раз для `X_test`:

```
In[7]:
# преобразовываем тестовые данные
X_test_scaled = scaler.transform(X_test)
# печатаем значения признаков в тестовом наборе после
масштабирования print("min значение признака после
масштабирования:\n{}".format(X_test_scaled.min(axis
=0))) print("max значение признака после
масштабирования:\n{}".format(X_test_scaled.max(axis
=0))) Out[7]:
```

```

min значение признака после масштабирования:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.  0.
 0.154  -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.  0.
-0.032  0.007  0.027  0.058  0.02  0.009  0.109
 0.026  0.  0.  -0.  -0.002] max значение
признака после масштабирования:
[ 0.958  0.815  0.956  0.894  0.811  1.22  0.88
 0.933  0.932  1.037
 0.427  0.498  0.441  0.284  0.487  0.739  0.767
 0.629  1.337  0.391
 0.896  0.793  0.849  0.745  0.915  1.132  1.07
 0.924  1.205  1.631]

```

Возможно, полученные результаты несколько удивят вас: после масштабирования минимальные и максимальные значения признаков в тестовом наборе не равны 0 и 1. Некоторые признаки даже выходят за пределами диапазона 0-1! Объяснить это можно тем, что MinMaxScaler (и все остальные типы масштабирования) всегда применяют одинаковое преобразование к обучающему и тестовому наборам. Это означает, что метод transform всегда вычитает минимальное значение, вычисленное для обучающего набора, и делит на ширину диапазона, вычисленную также для обучающего набора. Минимальное значение и ширина диапазона для обучающего набора могут отличаться от минимального значения и ширины диапазона для тестового набора.

9 АНАЛИЗ ГЛАВНЫХ КОМПОНЕНТ

Цель занятия: Углубить и закрепить знания, полученные на лекциях по алгоритмам машинного обучения и изучить возможности языка Python для проектирования данных – анализ главных компонент.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Преобразование данных с помощью неконтролируемого обучения может быть обусловлено многими причинами. Наиболее распространенные причины – визуализация, сжатие данных, а также поиск такого представления данных, которое даст больше информации в ходе дальнейшей обработки.

Одним из самых простых и наиболее широко используемых алгоритмов контролируемого обучения является анализ главных компонент (principal component analysis, PCA). Кроме того, мы рассмотрим еще два алгоритма: факторизацию неотрицательных матриц (non-negative matrix factorization, NMF), которая обычно используется для выделения признаков, и стохастическое вложение соседей с распределением Стьюдента (t-distributed stochastic neighbor embedding, t-SNE), которое обычно используется для визуализации с использованием двумерных диаграмм рассеяния.

Анализ главных компонент представляет собой метод, который осуществляет вращение данных с тем, чтобы преобразованные признаки не коррелировали между собой. Часто это вращение сопровождается выбором подмножества новых признаков в зависимости от их важности с точки зрения интерпретации данных. Следующий пример (рис. 3.3) иллюстрирует результат применения PCA к синтетическому двумерному массиву данных:

In[13]:

```
mglearn.plots.plot_pca_illustration()
```

Первый рис. (вверху слева) показывает исходные точки данных, выделенные цветом для лучшей дискриминации. Алгоритм начинает работу с того, что сначала находит направление максимальной дисперсии, помеченное как «компонента 1». Речь идет о направлении (или векторе) данных, который содержит большую часть информации, или другими словами, направление, вдоль которого признаки коррелируют друг с другом сильнее всего. Затем алгоритм находит направление, которое содержит наибольшее количество информации, и при этом ортогонально (расположено под прямым углом) первому направлению. В двумерном пространстве существует только одна возможная ориентация, расположенная под прямым углом, но в пространствах большей размерности может быть (бесконечно) много ортогональных направлений. Хотя эти две компоненты изображаются в виде стрелок, на самом деле не имеет значения, где начало, а где конец, мы могли бы нарисовать первую компоненту, выходящую из центра в верхний левый угол, а не в нижний правый. Направления, найденные с помощью этого алгоритма, называются главными компонентами (principal components), поскольку они яв-

ляются основными направлениями дисперсии данных. В целом максимально возможное количество главных компонент равно количеству исходных признаков.

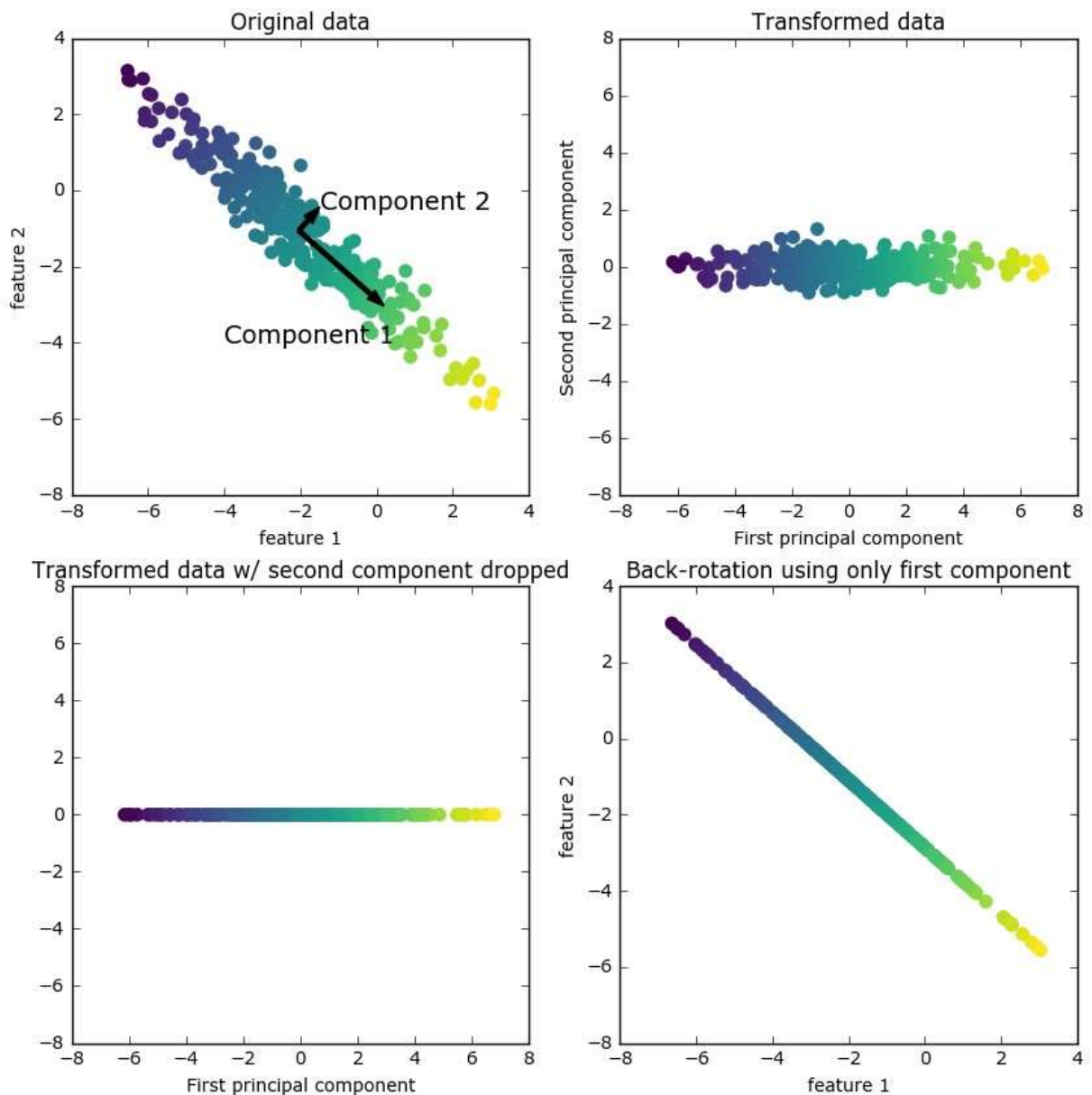


Рис. 3.3 Преобразование данных с помощью PCA

Второй график (вверху справа) показывает те же самые данные, но теперь повернутые таким образом, что первая главная компонента совпадает с осью x , а вторая главная компонента совпадает с осью y . Перед вращением из каждого значения данных вычитается среднее, таким образом, преобразованные данные центрированы около нуля. В новом представлении данных, найденном с помощью PCA, две оси становятся некоррелированными. Это означает, что в новом представлении все элементы корреляционной матрицы данных, кроме диагональных, будут равны нулю.

Мы можем использовать PCA для уменьшения размерности, сохранив

лишь несколько главных компонент. В данном примере мы можем оставить лишь первую главную компоненту, как показано на третьем графике рис. 3.3 (внизу слева). Это уменьшит размерность данных: из двумерного массива данных получаем одномерный массив данных. Однако следует отметить, что вместо того, чтобы оставить лишь один из исходных признаков, мы находим наиболее интересное направление (выходящее из верхнего левого угла в нижний правый на первом графике) и оставляем это направление, т.е. первую главную компоненту.

И, наконец, мы можем отменить вращение и добавить обратно среднее значение к значениям данных. В итоге получим данные, показанные на последнем графике рис. 3.3. Эти точки располагаются в пространстве исходных признаков, но мы оставили лишь информацию, содержащуюся в первой главной компоненте. Это преобразование иногда используется, чтобы удалить эффект шума из данных или показать, какая часть информации сохраняется при использовании главных компонент.

ЛИТЕРАТУРА

1. Манро, Р. Машинное обучение с участием человека / Монарх Р. Манро; перевод В. И. Бахур. — Москва: ДМК Пресс, 2022. — 498 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/125122.html>
2. Горбаченко, В. И. Машинное обучение: учебное пособие / В. И. Горбаченко, К. Е. Савенков, М. А. Малахов. — Москва: Ай Пи Ар Медиа, 2023. — 217 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/125886.html>
3. Теория и практика машинного обучения: учебное пособие / В. В. Воронина, А. В. Михеев, Н. Г. Ярушкина, К. В. Святков. — Ульяновск: Ульяновский государственный технический университет, 2017. — 291 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/106120.html>
4. Шалев-Шварц, Ш. Идеи машинного обучения: от теории к алгоритмам / Ш. Шалев-Шварц, Ш. Бен-Давид; перевод А. А. Слинкин. — Москва: ДМК Пресс, 2019. — 436 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/125114.html>
5. Сопов, Е. А. Многокритериальные нейроэволюционные системы в задачах машинного обучения и человеко-машинного взаимодействия: монография / Е. А. Сопов, И. А. Иванов. — Красноярск: Сибирский федеральный университет, 2019. — 160 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/100054.html>
6. Хапке, Х. Разработка конвейеров машинного обучения. Автоматизация жизненных циклов модели с помощью TensorFlow / Х. Хапке, К. Нельсон; перевод Н. Б. Желнова. — Москва: ДМК Пресс, 2021. — 345 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/125138.html>
7. Шарден, Б. Крупномасштабное машинное обучение вместе с Python / Б. Шарден, Л. Массарон, А. Боскетти; перевод А. В. Логунов. — Москва: ДМК Пресс, 2018. — 358 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/124994.html>
8. Ракитский, А. А. Методы машинного обучения: учебно-методическое пособие / А. А. Ракитский. — Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2018. — 32 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/90591.html>
9. Сараев, П. В. Методы машинного обучения: методические указания и задания к лабораторным работам по курсу / П. В. Сараев. — Липецк:

- Липецкий государственный технический университет, ЭБС АСВ, 2017. — 48 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/83183.html>
10. Неделько, В. М. Основы статистических методов машинного обучения: учебное пособие / В. М. Неделько. — Новосибирск: Новосибирский государственный технический университет, 2010. — 72 с. — Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. — URL: <https://www.iprbookshop.ru/45418.html>