O'REILLY®

Машинное обучение

Карманный справочник

Краткое руководство по методам структурированного машинного обучения на Python

Мэтт Харрисон

Машинное обучение

Карманный справочник

Machine Learning
Pocket Reference

Working with Structured Data in Python

Matt Harrison

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Машинное обучение

Карманный справочник

Краткое руководство по методам структурированного машинного обучения на Python

Мэтт Харрисон

ДИАЛЕКТИКА

Москва · Санкт-Петербург
2020

ББК 32.973.26-018.2.75
X21
УДК 681.3.07

ООО "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *В.А. Коваленко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Харрисон, Мэтт.

X21 Машинное обучение: карманный справочник. Краткое руководство по методам структурированного машинного обучения на Python. : Пер. с англ. — СПб. : ООО "Диалектика", 2020 — 320 с. : ил. — Парал. тит. англ.

ISBN 978-5-907203-17-4 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Мэтт Харрисон

Машинное обучение: карманный справочник
Краткое руководство по методам структурированного машинного обучения на Python

Подписано в печать 06.04.2020.
Формат 84x108/32. Гарнитура Times.
Усл. печ. л. 16,8. Уч.-изд. л. 8,8.
Тираж 200 экз. Заказ № 2368.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург,
Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-17-4 (рус.)

ISBN 978-1-492-04754-4 (англ.)

Оглавление

Введение	**13**
Глава 1. Введение в машинное обучение	**19**
Глава 2. Обзор процесса машинного обучения	**27**
Глава 3. Пошаговая классификация: набор данных Titanic	**29**
Глава 4. Пропущенные данные	**59**
Глава 5. Очистка данных	**67**
Глава 6. Исследование	**71**
Глава 7. Предварительная обработка данных	**91**
Глава 8. Выбор признаков	**105**
Глава 9. Несбалансированные классы	**115**
Глава 10. Классификация	**121**
Глава 11. Выбор модели	**169**
Глава 12. Метрики и оценка классификации	**173**
Глава 13. Объяснение моделей	**191**
Глава 14. Регрессия	**205**
Глава 15. Метрики и регрессионная оценка	**239**
Глава 16. Объяснение регрессионных моделей	**247**
Глава 17. Уменьшение размерности	**253**
Глава 18. Кластеризация	**285**
Глава 19. Конвейер	**301**
Предметный указатель	**307**

Содержание

Об авторе 11
Колофон 11

Введение **13**
Чего ожидать 14
Для кого написана эта книга 14
Соглашения, принятые в этой книге 14
Использование примеров кода 15
Посвящение 16
Ждем ваших отзывов! 17

Глава 1. Введение в машинное обучение **19**
Установка с использованием pip 23
Установка с помощью conda 24

Глава 2. Обзор процесса машинного обучения **27**

Глава 3. Пошаговая классификация: набор данных Titanic **29**
Соображения о плане проекта 29
Импорт 30
Задать вопрос 31
Условия для данных 31
Сбор данных 33
Очистка данных 34
Создание признаков 41
Выборка данных 43
Замещение данных 44
Нормализация данных 45
Рефакторинг 46

Простая модель 47
Разные семейства 48
Стекирование 49
Создание модели 50
Оценка модели 51
Оптимизация модели 52
Матрица неточностей 53
Кривая ROC 55
Кривая обучения 56
Развертывание модели 57

Глава 4. Пропущенные данные 59
Изучение пропущенных данных 60
Отбрасывание пропущенных данных 63
Замещение данных 64
Добавление индикаторных столбцов 65

Глава 5. Очистка данных 67
Имена столбцов 67
Замена пропущенных значений 68

Глава 6. Исследование 71
Размер данных 71
Сводная статистика 72
Гистограмма 73
Диаграмма рассеяния 74
Объединенный график 75
Парная сетка 77
Диаграмма размаха и скрипичная диаграмма размаха 78
Сравнение двух порядковых значений 80
Корреляция 82
RadViz 86
Параллельные координаты 88

Глава 7. Предварительная обработка данных 91
Стандартизация 91
Масштабирование до диапазона 93
Фиктивные переменные 94
Меточное кодирование 95

Частотное кодирование 96
Извлечение категорий из строк 97
Другие категориальные кодирования 99
Конструирование признаков данных 101
Добавление признака col_na 102
Конструирование признаков вручную 103

Глава 8. Выбор признаков 105
Коллинеарные столбцы 106
Регрессия лассо 108
Удаление рекурсивных признаков 110
Взаимная информация 112
Анализ основных компонентов 112
Важность признака 113

Глава 9. Несбалансированные классы 115
Использование другой метрики 115
Алгоритмы и ансамбли на основе дерева 115
Штрафующие модели 116
Повышающая дискретизация миноритарного класса 116
Генерация данных миноритарного класса 117
Понижающая дискретизация мажоритарного класса 118
Повышающая дискретизация, затем понижающая 120

Глава 10. Классификация 121
Логистическая регрессия 122
Наивный байесовский классификатор 127
Метод опорных векторов 129
К-ближайшие соседи 133
Дерево решений 136
Случайный лес 143
XGBoost 149
Градиентный бустинг с LightGBM 159
TPOT 165

Глава 11. Выбор модели 169
Кривая валидации 169
Кривая обучения 171

Глава 12. Метрики и оценка классификации **173**
Матрица неточностей 173
Метрики 176
Корректность 178
Отзыв 178
Точность 179
F1 179
Отчет о классификации 179
ROC 180
Кривая "точность–отзыв" 181
График кумулятивного усиления 183
Кривая подъема 185
Баланс классов 186
Ошибка прогнозирования класса 187
Порог дискриминации 188

Глава 13. Объяснение моделей **191**
Коэффициенты регрессии 191
Важность признака 192
LIME 192
Интерпретация дерева 194
Графики частичной зависимости 195
Суррогатные модели 198
Shapley 199

Глава 14. Регрессия **205**
Базовая модель 207
Линейная регрессия 208
SVM 212
К-ближайшие соседи 214
Древо решений 216
Случайный лес 223
Регрессия XGBoost 226
Регрессия LightGBM 232

Глава 15. Метрики и регрессионная оценка **239**
Метрики 239
График остатков 242

Гетероскедастичность 242
Нормальные остатки 244
График ошибки прогноза 245

Глава 16. Объяснение регрессионных моделей 247
Shapley 247

Глава 17. Уменьшение размерности 253
PCA 253
UMAP 271
t-SNE 277
PHATE 281

Глава 18. Кластеризация 285
Метод k-средних 285
Агломерационная (иерархическая) кластеризация 292
Понятие кластеров 295

Глава 19. Конвейер 301
Классификационный конвейер 301
Конвейер регрессии 303
Конвейер PCA 304

Предметный указатель 307

Об авторе

Мэтт Харрисон руководит компанией *MetaSnake*, занимающейся обучением языку Python и науке о данных, а также оказывающей консалтинговые услуги. Он использует язык Python с 2000 года в самых разных областях: в науке о данных, бизнес-аналитике, хранении, тестировании и автоматизации, управлении стеками программ с открытым исходным кодом, финансах и поиске.

Колофон

Животное на обложке — это северный гребенчатый тритон (*Triturus cristatus*), амфибия, водящаяся возле стоячей воды в Британии, на востоке континентальной Европы и Западной России.

Этот тритон имеет серо-коричневую спину с темными пятнами и желто-оранжевое брюхо с белыми крапинками. Во время брачного периода у самцов появляются большие зубчатые гребни, а у самок на хвостах — оранжевая полоса.

В зимние месяцы северный гребенчатый тритон не впадает в спячку в грязи или под скалами. Он охотится на других тритонов, головастиков, молодых лягушек, личинок насекомых и водяных улиток в воде, а также на насекомых, червей и других беспозвоночных на суше. Эти тритоны живут до 27 лет и могут достигать 7 дюймов (17,78 см) в длину.

Хотя в настоящее время статус сохранности северного гребенчатого тритона обозначен как наименее опасный, многие животные на обложках *O'Reilly* находятся под угрозой исчезновения. Все они важны для этого мира.

Иллюстрация на обложке сделана Кареном Монтгомери (Karen Montgomery) на основании черно-белой гравюры от *Meyers Kleines Lexicon*.

Введение

Машинное обучение и наука о данных сейчас очень популярны и являются сложными темами. Я работал с языком Python и данными бо́льшую часть своей профессиональной деятельности и хотел бы получить бумажную книгу, в которой можно было бы ознакомиться с общими методами, которые я использовал в деле и преподавал на семинарах по решению задач структурированного машинного обучения.

Я считаю, что эта книга — наилучший сборник ресурсов и примеров для решения задач прогнозирующего моделирования, если у вас есть структурированные данные. Есть много библиотек, которые выполняют часть требуемых задач, и я попытался включить в книгу те из них, которые мне показались полезными, поскольку я применял их на практике.

Многие могут посетовать на отсутствие методов глубокого обучения. Для это есть отдельные книги. Я также предпочитаю более простые методы, и другие специалисты в отрасли, кажется, со мной согласны. Глубокое обучение предназначено для неструктурированных данных (видео, аудио, изображений), а для структурированных есть такие мощные инструменты, как XGBoost.

Я надеюсь, что эта книга послужит вам полезным справочным материалом для решения насущных проблем.

Чего ожидать

В этой книге приведены подробные примеры решения общих задач структурированных данных. В ней рассматриваются различные библиотеки и модели, их компромиссы, настройка и интерпретация.

Приведенные фрагменты кода имеют такой размер, чтобы их можно было использовать и адаптировать в ваших собственных проектах.

Для кого написана эта книга

Если вы только изучаете машинное обучение или работали с ним не один год, эта книга станет для вас ценным справочным материалом. Она предполагает некоторое знание языка Python и совсем не углубляется в синтаксис. Скорее, она демонстрирует, как использовать различные библиотеки для решения реальных проблем.

Она не заменит углубленный курс, но поможет вам ориентироваться в том, что может охватывать прикладной курс машинного обучения. (Примечание: автор использует ее в качестве справочного материала для курсов по анализу данных и машинному обучению, который он преподает.)

Соглашения, принятые в этой книге

Здесь используются соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом*. Чтобы обратить внимание читателя на отдельные фрагменты текста, также применяется *курсив*.
- Текст программ, функций, переменных, URL веб-страниц и другой код представлены моноширинным шрифтом.

- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным** шрифтом.
- Знакоместо в описаниях синтаксиса выделено *курсивом*. Это указывает на необходимость заменить знакоместо фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте: `BINDSIZE=(`*максимальная ширина колонки*`) * (`*номер колонки*`)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).

СОВЕТ

Этот элемент обозначает совет или предложение.

НА ЗАМЕТКУ

Этот элемент обозначает общее примечание.

ВНИМАНИЕ

Этот элемент содержит предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т.д.) доступен по адресу `https://github.com/mattharrison/ml_pocket_reference`.

Эта книга поможет вам выполнить свою работу. В общем, если в ней предлагается пример кода, можете использовать его в своих программах и документах. Не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, для написания программы, в которой используется несколько фрагментов кода из этой книги, разрешение не требуется. Для продажи или распространения

CD-ROM с примерами из книг издательства *O'Reilly* необходимо иметь разрешение. Чтобы ответить на вопрос, сославшись на эту книгу и приведя пример кода, разрешение получать не нужно. Чтобы включить значительное количество примеров кода из этой книги в документацию своего продукта, требуется разрешение.

Мы ценим библиографические ссылки, но не требуем их. Обычно они включают название книги, автора, издателя и ISBN, например "*Machine Learning Pocket Reference* by Matt Harrison (O'Reilly). Copyright © 2019 Matt Harrison, 978-1-492-04754-4".

Если вы считаете, что использование примеров кода выходит за рамки добросовестного применения или требует указанного выше разрешения, свяжитесь с нами по адресу `permissions@oreilly.com`.

Посвящение

Благодарю мою жену и семью за поддержку. Я благодарен сообществу Python за обеспечение замечательного языка и набора инструментов для работы. С Николь Таш (Nicole Tache) было приятно работать, и она обеспечила отличные отзывы. Мои технические рецензенты, Микио Браун (Mikio Braun), Наталино Буса (Natalino Busa) и Джастин Фрэнсис (Justin Francis), поддержали меня. Честно. Спасибо!

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Актуальность ссылок не гарантируется.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

ГЛАВА 1

Введение в машинное обучение

Это не столько учебное пособие, сколько заметки, таблицы и примеры для машинного обучения. Книга была создана автором как дополнительный ресурс во время обучения, предназначенный для публикации в виде физического издания. Читатели, предпочитающие бумажные книги, могут добавить на полях собственные заметки и мысли и получить ценную ссылку на кураторские примеры.

Мы рассмотрим классификацию со структурированными данными. Другая распространенная область применения машинного обучения подразумевает прогнозирование непрерывного значения (регрессия), создание кластеров и попытки уменьшить размерность, среди прочего. В этой книге не обсуждаются методы глубокого обучения; хотя методы из этой книги хорошо работают с неструктурированными данными, большинство из них можно рекомендовать и для структурированных данных.

Мы предполагаем знание языка Python или знакомство с ним. Полезно научиться манипулировать данными с помощью *библиотеки pandas*. У нас много примеров использования библиотеки pandas, и это отличный инструмент для работы со структурированными данными. Но некоторые операции индексирования могут привести к путанице, если вы не знакомы с numpy. Полное освещение библиотеки pandas достойно отдельной книги.

В этой книге используется много библиотек. Это может быть и хорошо, и плохо. Некоторые из этих библиотек могут быть сложными для установки или конфликтовать с другими версиями библиотек. Не думайте, что вам нужно установить их все. Используйте "JIT-инсталляцию" и устанавливайте только те библиотеки, которые хотите использовать по мере необходимости.

```
>>> import autosklearn, catboost,
category_encoders, dtreeviz, eli5, fancyimpute,
fastai, featuretools, glmnet_py, graphviz,
hdbscan, imblearn, janitor, lime, matplotlib,
missingno, mlxtend, numpy, pandas, pdpbox, phate,
pydotplus, rfpimp, scikitplot, scipy, seaborn,
shap, sklearn, statsmodels, tpot, treeinterpreter,
umap, xgbfir, xgboost, yellowbrick

>>> for lib in [
...     autosklearn,
...     catboost,
...     category_encoders,
...     dtreeviz,
...     eli5,
...     fancyimpute,
...     fastai,
...     featuretools,
...     glmnet_py,
...     graphviz,
...     hdbscan,
...     imblearn,
...     lime,
...     janitor,
...     matplotlib,
...     missingno,
...     mlxtend,
...     numpy,
...     pandas,
...     pandas_profiling,
...     pdpbox,
...     phate,
...     pydotplus,
...     rfpimp,
```

```
...     scikitplot,
...     scipy,
...     seaborn,
...     shap,
...     sklearn,
...     statsmodels,
...     tpot,
...     treeinterpreter,
...     umap,
...     xgbfir,
...     xgboost,
...     yellowbrick,
... ]:
...     try:
...         print(lib.__name__, lib.__version__)[1]
...     except:
...         print("Missing", lib.__name__)
catboost 0.11.1
category_encoders 2.0.0
Missing dtreeviz
eli5 0.8.2
fancyimpute 0.4.2
fastai 1.0.28
featuretools 0.4.0
Missing glmnet_py
graphviz 0.10.1
hdbscan 0.8.22
imblearn 0.4.3
janitor 0.16.6
Missing lime
matplotlib 2.2.3
missingno 0.4.1
mlxtend 0.14.0
numpy 1.15.2
pandas 0.23.4
Missing pandas_profiling
pdpbox 0.2.0
phate 0.4.2
Missing pydotplus
rfpimp
```

[1] Обратите внимание, идентификаторы `__name__` и `__version__` начинаются и заканчиваются двумя символами подчеркивания.

```
scikitplot 0.3.7
scipy 1.1.0
seaborn 0.9.0
shap 0.25.2
sklearn 0.21.1
statsmodels 0.9.0
tpot 0.9.5
treeinterpreter 0.1.0
umap 0.3.8
xgboost 0.81
yellowbrick 0.9
```

NOTE

Большинство из этих библиотек легко устанавливаются с помощью `pip` или `conda`. С `fastai` мне нужно использовать `pip install --no-deps fastai`. Библиотека `umap` устанавливается с использованием `pip install umap-learn`. Библиотека `janitor` устанавливается с использованием `pip install pyjanitor`. Библиотека `autosklearn` устанавливается с использованием `pip install auto-sklearn`.

Для анализа я обычно применяю Jupyter. Вы также можете использовать другие инструменты Notebook. Обратите внимание, что некоторые из них, например Google Colab, предварительно устанавливают множество библиотек (хотя они могут иметь устаревшие версии).

Существует два основных варианта установки библиотек в язык Python. Одним из них является использование `pip` (сокращение от "Pip Installs Python"), инструмента, поставляемого с языком Python. Другой вариант — использовать *Anaconda*. Мы представим оба.

Установка с использованием pip

Перед использованием `pip` мы создадим среду *песочницы* (sandbox) для установки наших библиотек. Это называется виртуальной средой по имени `env`:

```
$ python -m venv env
```

НА ЗАМЕТКУ

На Macintosh и Linux используйте `python`, на Windows — `python3`. Если Windows не распознает это из командной строки, вам может потребоваться переустановка или исправление установки. Убедитесь, что флажок Add Python to my PATH (Добавить Python в мой путь) установлен.

Затем вы активизируете среду, чтобы библиотеки при установке помещались в среду песочницы, а не в глобальную установку Python. Поскольку многие из этих библиотек изменяются и обновляются, лучше фиксировать версии для каждого проекта, чтобы знать, что код будет работать.

Вот как мы активизируем виртуальную среду на Linux и Macintosh:

```
$ source env/bin/activate
```

Вы заметите, что приглашение обновилось, указав, что используется виртуальная среда:

```
(env) $ which python
env/bin/python
```

В Windows нужно активизировать среду, выполнив следующую команду:

```
C:> env\Scripts\activate.bat
```

Опять же, вы заметите, что приглашение обновляется, указав, что используется виртуальная среда:

```
(env) C:> where python
env\Scripts\python.exe
```

На всех платформах можно устанавливать пакеты, используя `pip`. Чтобы установить библиотеку pandas, введите

```
(env) $ pip install pandas
```

Некоторые из имен пакетов отличаются от имен библиотек. Вы можете искать пакеты с помощью команды

```
(env) $ pip search libraryname
```

Установив свои пакеты, можете создать файл со всеми версиями пакетов, используя `pip`:

```
(env) $ pip freeze > requirements.txt
```

С помощью этого файла `requirements.txt` можно легко установить пакеты в новую виртуальную среду:

```
(other_env) $ pip install -r requirements.txt
```

Установка с помощью conda

Инструмент `conda` поставляется с Anaconda и позволяет создавать среды и устанавливать пакеты.

Чтобы создать среду по имени `env`, выполните команду

```
$ conda create --name env python=3.6
```

Чтобы активизировать эту среду, выполните команду

```
$ conda activate env
```

Это обновит приглашение в системах Unix и Windows. Теперь вы можете искать пакеты, используя команду

```
(env) $ conda search libraryname
```

Чтобы установить пакет, например, pandas, выполните команду

```
(env) $ conda install pandas
```

Чтобы создать файл с требованиями к пакету, выполните команду

```
(env) $ conda env export > environment.yml
```

Чтобы установить эти требования в новой среде, выполните команду

```
(other_env) $ conda create -f environment.yml
```

ВНИМАНИЕ

Некоторые из библиотек, упомянутых в этой книге, недоступны для установки из хранилища Anaconda. Не волнуйтесь. Оказывается, можно использовать `pip` внутри среды `conda` (создавать новую виртуальную среду не нужно) и устанавливать их с помощью `pip`.

ГЛАВА 2

Обзор процесса машинного обучения

Межотраслевой стандартный процесс для исследования данных (Cross-Industry Standard Process for Data Mining — CRISP-DM) — это процесс *интеллектуального анализа данных* (data mining). Он состоит из нескольких этапов, которые можно выполнять для постоянного улучшения.

- Понимание предметной области
- Понимание данных
- Подготовка данных
- Моделирование
- Оценка
- Развертывание

На рис. 2.1 демонстрируется мой рабочий процесс для создания прогнозирующей модели, которая расширяет методологию CRISP-DM. Пошаговое руководство в следующей главе будет охватывать следующие основные этапы.

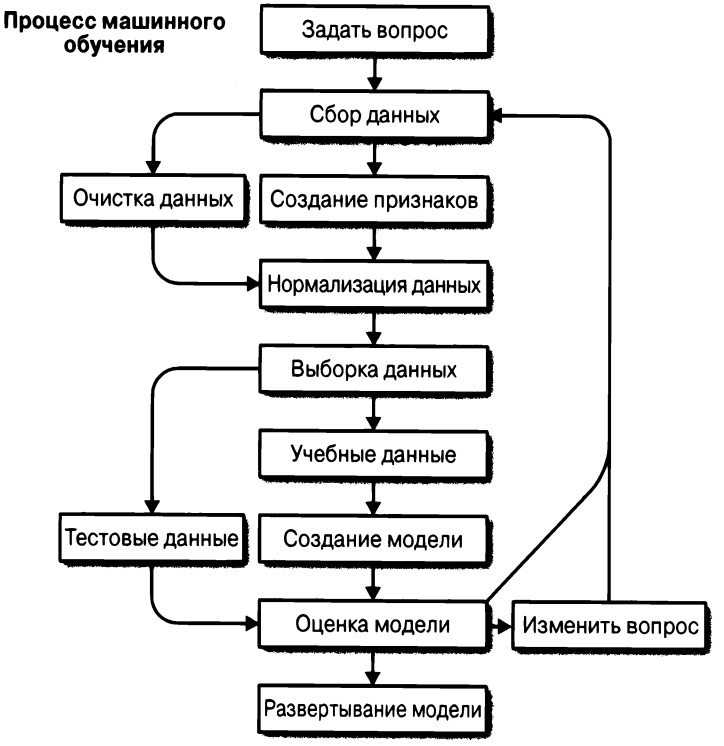


Рис. 2.1. *Общий рабочий процесс машинного обучения*

ГЛАВА 3

Пошаговая классификация: набор данных Titanic

В этой главе рассматриваются общие вопросы классификации с использованием *набора данных Titanic*. В последующих главах подробно рассмотрены общие этапы, выполняемые во время анализа.

Соображения о плане проекта

Отличным инструментом для проведения разведочного анализа данных является *Jupyter* — оболочка с открытым исходным кодом, которая поддерживает язык Python и другие языки. Она позволяет создавать *ячейки* кода или содержимое Markdown.

Я склонен использовать Jupyter в двух режимах: один — для анализа данных и быстрого опробования, другой — более гибкий стиль, в котором я формирую отчет с использованием ячеек Markdown и вставляю ячейки кода, чтобы проиллюстрировать важные моменты или открытия. Если вы не будете осторожны, вашей оболочке могут потребоваться *рефакторинг* (refactoring) и применение методов разработки программного обеспечения (удаление глобальных переменных, использование признаков, классов и т.д.).

Пакет научных данных cookiecutter предлагает план создания проекта анализа, который позволяет легко воспроизводить код и обмениваться им.

Импорт

Этот пример основан главным образом на библиотеках *pandas*, *Scikit-learn* и *Yellowbrick*. Библиотека pandas предоставляет инструменты, облегчающие сбор данных. Библиотека Scikit-learn имеет отличные прогнозирующие модели, а Yellowbrick — библиотека визуализации для оценки моделей:

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> from sklearn import (
...     ensemble,
...     preprocessing,
...     tree,
... )
>>> from sklearn.metrics import (
...     auc,
...     confusion_matrix,
...     roc_auc_score,
...     roc_curve,
... )
>>> from sklearn.model_selection import (
...     train_test_split,
...     StratifiedKFold,
... )
>>> from yellowbrick.classifier import (
...     ConfusionMatrix,
...     ROCAUC,
... )
>>> from yellowbrick.model_selection import (
...     LearningCurve,
... )
```

ВНИМАНИЕ

В Интернете вы можете найти документацию и примеры, в которых синтаксис импорта включает звездочку, например:

```
from pandas import *
```

Воздержитесь от использования синтаксиса импорта со звездочкой. Ясность упростит понимание вашего кода.

Задать вопрос

В этом примере будет создана прогностическая модель для ответа на вопрос. Модель будет классифицировать, выживет ли человек при катастрофе на корабле Титаник, исходя из индивидуальных признаков и характеристик. Это учебный пример, но он служит педагогическим инструментом для демонстрации многих этапов моделирования. Наша модель должна быть способна принимать информацию о пассажирах и прогнозировать, выживет ли пассажир на Титанике.

Поскольку мы прогнозируем метку выживания (либо пассажир выжил, либо нет), это вопрос классификации.

Условия для данных

Обычно мы обучаем модель на матрице данных. (Я предпочитаю использовать pandas DataFrame, поскольку очень хорошо иметь метки столбцов да и массивы numpy хорошо работают.)

При обучении с учителем, таком как регрессия или классификация, наша цель заключается в получении функции, которая преобразует признаки в метку. Если бы мы написали это как алгебраическую формулу, то она выглядела бы так:

```
y = f(X)
```

X — это матрица. Каждая строка представляет *выборку* (sample) данных или информацию о человеке. Каждый столбец в X — это *признак* (feature). Результатом нашей функции y является вектор, который содержит метки (для классификации) или значения (для регрессии) (рис. 3.1).

Рис. 3.1. *Формат структурированных данных*

Это стандартная процедура присвоения имен данным и выводу. Если вы читаете научные статьи или даже просматриваете документацию для библиотек, вы уже заметили, что они следуют этому соглашению. В языке Python мы используем переменную по имени X для хранения данных выборки, даже если использование заглавных букв является нарушением стандартных соглашений об именах (PEP 8). Не волнуйтесь, все это делают, и если бы вы назвали свою переменную x, это могло бы выглядеть смешно. Переменная y хранит метки или цели.

В табл. 3.1 демонстрируется простой набор данных с двумя выборками и тремя признаками для каждой выборки.

Таблица 3.1. Выборки (строки) и признаки (столбцы)

pclass	age	sibsp
1	29	0
1	2	1

Сбор данных

Мы собираемся загрузить файл Excel (убедитесь, что у вас установлены pandas и xlrd[1]) с признаками Titanic. В нем много столбцов, в том числе столбец survived (выжил), который содержит метку о том, что стало с человеком:

```
>>> url = (
...     "http://biostat.mc.vanderbilt.edu/"
...     "wiki/pub/Main/DataSets/titanic3.xls"
... )
>>> df = pd.read_excel(url)
>>> orig_df = df
```

В набор данных включены следующие столбцы.

- `pclass` — класс пассажира (1 — первый, 2 — второй, 3 — третий)
- `survival` — выживание (0 — нет, 1 — да)
- `name` — имя
- `sex` — пол
- `age` — возраст
- `sibsp` — количество братьев и сестер или супругов на борту
- `parch` — количество родителей или детей на борту
- `ticket` — номер билета
- `fare` — тариф пассажира
- `cabin` — каюта

[1] Хотя мы не вызываем эту библиотеку непосредственно, когда загружаем файл Excel, библиотека pandas использует ее внутренне.

- `embarked` — точка посадки (C — Шербур, Q — Квинстаун, S — Саутгемптон)
- `boat` — спасательная шлюпка
- `body` — идентификационный номер тела
- `home.dest` — дом или место назначения

Библиотека pandas может сама прочитать эту таблицу и преобразовать ее во фрейм данных. Нам нужно будет выборочно проверить данные и убедиться, что они подходят для проведения анализа.

Очистка данных

Получив данные, следует убедиться, что они представлены в формате, который можно использовать для создания модели. Большинство моделей Scikit-learn требуют, чтобы наши признаки были числовыми (целочисленными или с плавающей запятой). Кроме того, многие модели терпят неудачу, если им передаются данные с пропущенными значениями (`NaN` в pandas или numpy). Некоторые модели работают лучше, если данные *стандартизированы* (со средним значением 0 и стандартным отклонением 1). Мы будем решать эти проблемы, используя библиотеки pandas или Scikit-learn. Кроме того, в наборе данных Titanic есть утечки (*leaky*).

Признаки утечки (leaky feature) — это переменные, содержащие информацию о будущем или цели. Нет ничего плохого в том, чтобы иметь данные о цели, и мы часто имеем эти данные во время создания модели. Но если эти переменные недоступны, когда мы выполняем прогнозирование для новой выборки, необходимо удалить их из модели, поскольку они пропускают данные из будущего.

Очистка данных может отнять немного времени. При этом полезно обратиться к *эксперту в предметной области* (Subject Matter Expert — SME), который может предоставить рекомендации по работе с выбросами или пропущенными данными.

```
>>> df.dtypes
pclass        int64
survived      int64
name         object
sex          object
age         float64
sibsp         int64
parch         int64
ticket       object
fare        float64
cabin        object
embarked     object
boat         object
body        float64
home.dest    object
dtype: object
```

Обычно мы видим `int64`, `float64`, `datetime64[ns]` или `object`. Это типы, которые библиотека pandas использует для хранения столбца данных. Типы `int64` и `float64` являются числовыми. Тип `datetime64[ns]` содержит данные о дате и времени. Тип `object` обычно означает, что столбец содержит строковые данные, хотя это может быть комбинация строковых и других типов.

При чтении из файлов CSV библиотека pandas попытается привести данные к соответствующему типу, но если не получится, будет считать их типом `object`. Чтение данных из электронных таблиц, баз данных или других систем и лучшие типы может обеспечить DataFrame. В любом случае стоит просмотреть данные и убедиться, что типы имеют смысл.

Целочисленные типы обычно в порядке. Типы с плавающей запятой могут иметь некоторые пропущенные значения. Типы даты и строк необходимо преобразовать или использовать для создания числовых типов. Строковые типы с малым количеством элементов называют *категориальными столбцами* (categorical column), и может быть целесообразным создать из них *фиктивные столбцы* (dummy column) (об этом позаботится функция `pd.get_dummies`).

НА ЗАМЕТКУ

До pandas 0.23 тип `int64` гарантировал отсутствие пропущенных значений. Значения типа `float64` могут быть всеми числами с плавающей запятой, но могут также представлять собой целочисленные значения с пропущенными значениями. Библиотека pandas преобразует целочисленные значения с пропущенными числами в числа с плавающей точкой, поскольку этот тип поддерживает пропущенные значения. Тип `object` обычно означает строковые типы (или строковые и числовые).

Начиная с pandas 0.24, появился новый тип `Int64` (обратите внимание на заглавные буквы). Стандартно это не целочисленный тип, но вы можете осуществить приведение к этому типу и получить поддержку для пропущенных чисел.

Библиотека `pandas_profiling` включает в себя профильный отчет. Вы можете создать этот отчет в оболочке. Он резюмирует типы столбцов и позволит вам просмотреть подробную информацию о квантильной статистике, описательной статистике, гистограмме, общих значениях и экстремальных значениях (рис. 3.2 и 3.3):

```
>>> import pandas_profiling
>>> pandas_profiling.ProfileReport(df)
```

Для проверки количества строк и столбцов используйте атрибут `.shape` объекта DataFrame:

```
>>> df.shape
(1309, 14)
```

Для получения сводной статистики, а также для просмотра количества ненулевых данных используйте метод `.describe`. Стандартное поведение этого метода — создавать отчеты только по числовым столбцам. Здесь вывод усекается, чтобы показать только первые два столбца:

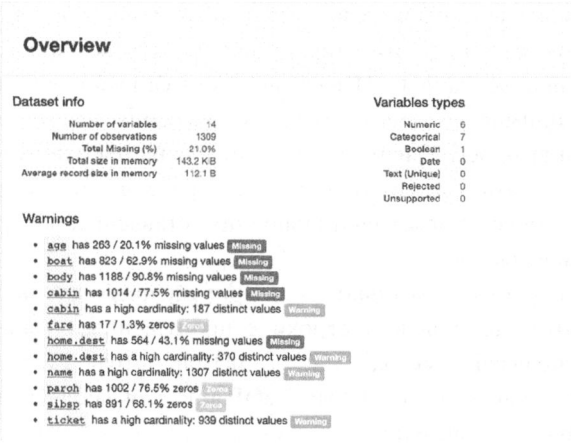


Рис. 3.2. *Резюме* `pandas_profiling`

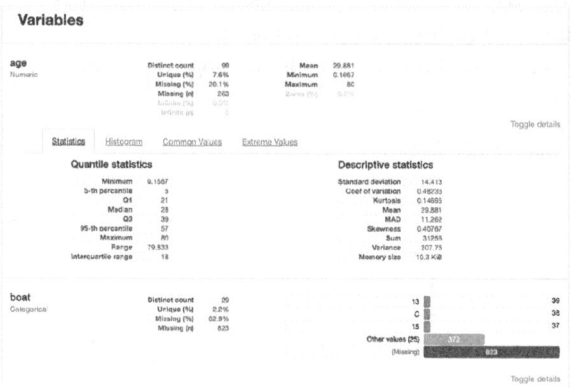


Рис. 3.3. *Подробности о переменных* `pandas_profiling`

```
>>> df.describe().iloc[:, :2]
            pclass     survived
count  1309.000000  1309.000000
mean      2.294882     0.381971
std       0.837836     0.486055
min       1.000000     0.000000
25%       2.000000     0.000000
50%       3.000000     0.000000
75%       3.000000     1.000000
max       3.000000     1.000000
```

Статистика подсчета включает только значения, которые не являются NaN, поэтому полезно проверить, нет ли в столбце пропущенных данных. Имеет также смысл выборочно проверить минимальные и максимальные значения, чтобы увидеть, есть ли выбросы. Сводная статистика является одним из способов сделать это. Построение гистограммы или диаграммы размаха обеспечивает визуальное представление, которое мы увидим позже.

Нужно будет разобраться в пропущенных данных. Чтобы найти столбцы или строки с пропущенными значениями, используйте метод .isnull. Вызов метода .isnull для DataFrame возвращает новый DataFrame, каждая ячейка которого содержит значение True или False. В Python эти значения рассматриваются как 1 и 0 соответственно. Это позволяет нам суммировать их или даже вычислять процент пропущенных значений (вычислив среднее значение).

Код выводит количество пропущенных данных в каждом столбце:

```
>>> df.isnull().sum()
pclass          0
survived        0
name            0
sex             0
age           263
sibsp           0
parch           0
ticket          0
fare            1
cabin        1014
embarked        2
boat          823
body         1188
home.dest     564
dtype: int64
```

Чтобы получить процент от нулевых значений, замените .sum на .mean. Стандартно вызов этих методов приведет к выполнению операции вдоль оси 0, которая проходит вдоль

индекса. Если вы хотите получить количество пропущенных признаков для каждой выборки, примените это по оси 1 (вдоль столбцов):

```
>>> df.isnull().sum(axis=1).loc[:10]
0     1
1     1
2     2
3     1
4     2
5     1
6     1
7     2
8     1
9     2
dtype: int64
```

SME может помочь определить, что делать с пропущенными данными. Столбец `age` может быть полезен, поэтому его сохранение и интерполяция значений могут дать некоторый сигнал для модели. Столбцы, в которых пропущено большинство значений (`cabin`, `boat` и `body`), как правило, имеют мало смысла и могут быть отброшены.

Столбец `body` (идентификационный номер тела) для многих строк отсутствует. Мы должны отбросить этот столбец в любом случае, поскольку данные в нем пропущены. Этот столбец указывает, что пассажир не выжил; при необходимости наша модель может использовать его для обмана. Мы удалим это. (Если мы создаем модель для прогнозирования смертности пассажиров, то информация о том, что у него есть идентификационный номер тела, априори сообщит нам, что он уже мертв. Мы хотим, чтобы наша модель не знала об этом и делала прогноз на основе сведений из других столбцов.) Аналогично столбец `boat` создает утечку обратной информации (что пассажир выжил).

Давайте посмотрим на некоторые строки с пропущенными данными. Мы можем создать логический массив (набор значений `True` или `False`, чтобы указать, есть ли в строке

недостающие данные) и использовать его для проверки строк, в которых данные пропущены:

```
>>> mask = df.isnull().any(axis=1)
>>> mask.head() # rows
0    True
1    True
2    True
3    True
4    True
dtype: bool

>>> df[mask].body.head()
0      NaN
1      NaN
2      NaN
3    135.0
4      NaN
Name: body, dtype: float64
```

Позже мы будем добавлять (или вычислять) пропущенные значения для столбца `age`.

Столбцы с типом `object` обычно категориальные (но они могут также быть строковыми данными с большим количеством элементов или комбинацией типов столбцов). Для проверки количества значений столбцов типа `object`, которые мы считаем категориальными, используйте метод `.value_counts`:

```
>>> df.sex.value_counts(dropna=False)
male      843
female    466
Name: sex, dtype: int64
```

Помните, что pandas обычно игнорирует значения `null` и `NaN`. Если вы хотите включить их, используйте `dropna=False`, чтобы показать счетчики и для `NaN`:

```
>>> df.embarked.value_counts(dropna=False)
S      914
C      270
Q      123
NaN      2
Name: embarked, dtype: int64
```

У нас есть несколько вариантов действий при пропущенных начальных значениях. Использование `S` может показаться логичным, поскольку это наиболее распространенное значение. Мы могли бы покопаться в данных и попытаться определить, не лучше ли другой вариант. Мы также можем отбросить эти два значения. Или, поскольку они категориальные, мы можем их игнорировать и использовать pandas для создания фиктивных столбцов, если в этих двух выборках будет просто 0 записей для каждого варианта. Для данного признака мы будем использовать этот последний вариант.

Создание признаков

Мы можем отбросить столбцы, которые не имеют дисперсии или сигнала. В этом наборе данных нет таких признаков, но если бы был столбец по имени "is human" (человек ли), в котором была бы единица для каждого случая, этот столбец не предоставил бы никакой информации.[2]

В качестве альтернативы, если мы не используем NLP или не извлекаем данные из текстовых столбцов, в которых каждое значение различается, модель не сможет воспользоваться этим столбцом. Примером этого является имя столбца. Некоторые извлекают из названия заголовок `t` и считают его категориальным.

Мы также хотим удалить столбцы с утечкой информации. Столбцы `boat` и `body` обеспечивают утечку информации о том, выжил ли пассажир.

Метод `.drop` библиотеки pandas может отбрасывать строки или столбцы:

```
>>> name = df.name
>>> name.head(3)
0      Allen, Miss. Elisabeth Walton
1     Allison, Master. Hudson Trevor
```

[2] А как же собаки и кошки? Их не было на борту или их судьба никого не интересует? — *Примеч. ред.*

```
2       Allison, Miss. Helen Loraine
Name: name, dtype: object

>>> df = df.drop(
...     columns=[
...         "name",
...         "ticket",
...         "home.dest",
...         "boat",
...         "body",
...         "cabin",
...     ]
... )
```

Из строковых столбцов нужно создать фиктивные столбцы. Это даст новые столбцы — `sex` и `embarked`. Для этого pandas имеет удобную функцию `get_dummies`:

```
>>> df = pd.get_dummies(df)

>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp',
       'parch', 'fare', 'sex_female', 'sex_male',
       'embarked_C', 'embarked_Q', 'embarked_S'],
    dtype='object')
```

На данный момент столбцы `sex_male` и `sex_female` имеют обратную корреляцию. Обычно мы удаляем любые столбцы с идеальной или очень высокой положительной или отрицательной корреляцией. В некоторых моделях мультиколлинеарность может влиять на интерпретацию важности признаков и коэффициентов. Вот код для удаления столбца `sex_male`:

```
>>> df = df.drop(columns="sex_male")
```

В качестве альтернативы к вызову `get_dummies` мы можем добавить параметр `drop_first=True`:

```
>>> df = pd.get_dummies(df, drop_first=True)

>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp',
    'parch', 'fare', 'sex_male',
```

```
    'embarked_Q', 'embarked_S'],
    dtype='object')
```

Создайте объект DataFrame (`X`) с элементами и серию (`y`) с метками. Мы также можем использовать массивы numpy, но тогда у нас не будет имен столбцов:

```
>>> y = df.survived
>>> X = df.drop(columns="survived")
```

СОВЕТ

Для замены двух последних строк мы можем использовать *библиотеку pyjanitor*:

```
>>> import janitor as jn
>>> X, y = jn.get_features_targets(
...     df, target_columns="survived"
... )
```

Выборка данных

Мы всегда хотим обучаться и тестироваться на разных данных. В противном случае невозможно узнать, насколько хорошо модель обобщает данные, которых она не видела раньше. Мы будем использовать библиотеку Scikit-learn, чтобы получить 30% данных для тестирования (используя `random_state=42`, чтобы удалить элемент случайности, если мы начнем сравнивать разные модели):

```
>>> X_train, X_test, y_train, y_test = model_selec
tion.train_test_split(
...     X, y, test_size=0.3, random_state=42
... )
```

Замещение данных

В столбце age пропущены значения. Нам нужно подставить возраст из числовых значений. Мы только хотим вписать их в учебный набор, а затем использовать этот *заменитель* (imputer) для заполнения дат тестового набора. В противном случае мы допускаем утечку данных (обманываем, подставляя в модель информацию о будущем).

Теперь, когда у нас есть тестовые и обучающие данные, мы можем заменить пропущенные значения в обучающем наборе и использовать обученные заменители для заполнения тестового набора данных. *Библиотека fancyimpute* имеет много алгоритмов, которые она реализует. К сожалению, большинство из этих алгоритмов не реализованы в *индуктивной* манере. Это означает, что вы не можете вызвать сначала .fit, а затем — .transform, а значит, вы не можете рассчитывать на новые данные, основанные на том, как была обучена модель.

Класс IterativeImputer (который ранее был в fancyimpute, но был перенесен в Scikit-learn) поддерживает индуктивный режим. Чтобы использовать его, нам нужно добавить специальный экспериментальный импорт (начиная с версии Scikit-learn 0.21.2):

```
>>> from sklearn.experimental import (
...     enable_iterative_imputer,
... )
>>> from sklearn import impute
>>> num_cols = [
...     "pclass",
...     "age",
...     "sibsp",
...     "parch",
...     "fare",
...     "sex_female",
... ]

>>> imputer = impute.IterativeImputer()
>>> imputed = imputer.fit_transform(
...     X_train[num_cols]
```

```
... )
>>> X_train.loc[:, num_cols] = imputed
>>> imputed = imputer.transform(X_test[num_cols])
>>> X_test.loc[:, num_cols] = imputed
```

Чтобы дописать медиану, можно использовать код pandas:

```
>>> meds = X_train.median()
>>> X_train = X_train.fillna(meds)
>>> X_test = X_test.fillna(meds)
```

Нормализация данных

Нормализация или предварительная обработка данных поможет многим моделям работать лучше. Особенно тем, которые зависят от метрики расстояния для определения сходства. (Обратите внимание, что древовидные модели, которые обрабатывают каждый признак отдельно, не имеют этого требования.)

Мы собираемся стандартизировать данные для предварительной обработки. Стандартизация преобразует данные так, чтобы они имели нулевое среднее значение и стандартное отклонение, равное единице. Таким образом, модели не рассматривают переменные с большими масштабами как более важные, чем переменные с меньшими масштабами. Я собираюсь вставить результат (массив numpy) обратно в DataFrame pandas для облегчения манипуляции (и сохранения имен столбцов).

Я также обычно не стандартизирую фиктивные столбцы, поэтому буду их игнорировать:

```
>>> cols = "pclass,age,sibsp,fare".split(",")
>>> sca = preprocessing.StandardScaler()
>>> X_train = sca.fit_transform(X_train)
>>> X_train = pd.DataFrame(X_train, columns=cols)
>>> X_test = sca.transform(X_test)
>>> X_test = pd.DataFrame(X_test, columns=cols)
```

Рефакторинг

На данный момент мне нравится рефакторинг моего кода. Обычно я создаю две функции: одну — для общей очистки, а другую — для разделения на обучающий и тестовый наборы данных, а также для выполнения мутаций, которые должны происходить в этих наборах по-разному:

```
>>> def tweak_titanic(df):
...     df = df.drop(
...         columns=[
...             "name",
...             "ticket",
...             "home.dest",
...             "boat",
...             "body",
...             "cabin",
...     ]
... ).pipe(pd.get_dummies, drop_first=True)
... return df

>>> def get_train_test_X_y(
...     df, y_col, size=0.3, std_cols=None
... ):
... y = df[y_col]
... X = df.drop(columns=y_col)
... X_train, X_test, y_train, y_test =
model_selection.train_test_split(
...         X, y, test_size=size, random_state=42
... )
... cols = X.columns
... num_cols = [
...     "pclass",
...     "age",
...     "sibsp",
...     "parch",
...     "fare",
... ]
... fi = impute.IterativeImputer()
... X_train.loc[
...     :, num_cols
... ] = fi.fit_transform(X_train[num_cols])
```

```
... X_test.loc[:, num_cols] = fi.transform(
...     X_test[num_cols]
... )
...
... if std_cols:
...     std = preprocessing.StandardScaler()
...     X_train.loc[
...         :, std_cols
... ] = std.fit_transform(
...         X_train[std_cols]
... )
... X_test.loc[
...     :, std_cols
... ] = std.transform(X_test[std_cols])
...
... return X_train, X_test, y_train, y_test

>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test =
get_train_test_X_y(
...     ti_df, "survived", std_cols=std_cols
... )
```

Простая модель

Создание простой модели, которая делает действительно что-то простое, может дать нам кое-что для сравнения с нашей моделью. Обратите внимание, что использование стандартного результата `.score` дает нам точность, которая может вводить в заблуждение. Модель задачи, в которой позитивный случай равен 1 на 10000, может легко дать точность более 99%, всегда прогнозируя негативный результат:

```
>>> from sklearn.dummy import DummyClassifier
>>> bm = DummyClassifier()
>>> bm.fit(X_train, y_train)
>>> bm.score(X_test, y_test) # точность
0.5292620865139949
```

```
>>> from sklearn import metrics
>>> metrics.precision_score(
...     y_test, bm.predict(X_test)
... )
0.4027777777777778
```

Разные семейства

Этот код опробует различные семейства алгоритмов. Теорема "бесплатных обедов не бывает" гласит, что ни один алгоритм не работает хорошо на всех данных. Однако для некого конечного набора данных вполне может существовать алгоритм, который хорошо работает именно с этим набором. (Популярный выбор для структурированного обучения в наши дни — это алгоритм с древовидной структурой, такой как XGBoost.)

Здесь мы используем несколько разных семейств, а также сравниваем оценку AUC и стандартное отклонение, используя k-блочную перекрестную проверку. Алгоритм, имеющий немного меньшую среднюю оценку, но более жесткое стандартное отклонение, может быть лучшим выбором.

Поскольку мы используем k-блочную перекрестную проверку, мы передадим модели все X и y:

```
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
>>> from sklearn import model_selection
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.linear_model import (
...     LogisticRegression,
... )
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import (
...     KNeighborsClassifier,
... )
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import (
...     RandomForestClassifier,
```

```
>>> import xgboost

>>> for model in [
...     DummyClassifier,
...     LogisticRegression,
...     DecisionTreeClassifier,
...     KNeighborsClassifier,
...     GaussianNB,
...     SVC,
...     RandomForestClassifier,
...     xgboost.XGBClassifier,
... ]:
...     cls = model()
...     kfold = model_selection.KFold(
...         n_splits=10, random_state=42
... )
... s = model_selection.cross_val_score(
...     cls, X, y, scoring="roc_auc", cv=kfold
... )
... print(
...     f"{model.__name__:22} AUC: "
...     f"{s.mean():.3f} STD: {s.std():.2f}"
... )
DummyClassifier        AUC: 0.511  STD: 0.04
LogisticRegression     AUC: 0.843  STD: 0.03
DecisionTreeClassifier AUC: 0.761  STD: 0.03
KNeighborsClassifier   AUC: 0.829  STD: 0.05
GaussianNB             AUC: 0.818  STD: 0.04
SVC                    AUC: 0.838  STD: 0.05
RandomForestClassifier AUC: 0.829  STD: 0.04
XGBClassifier          AUC: 0.864  STD: 0.04
```

Стекирование

Если вы следовали по маршруту Kaggle (или хотите получить максимальную производительность за счет интерпретируемости), можете использовать *стекирование* (stacking). Стековый классификатор берет другие модели и использует их вывод для прогнозирования цели или метки. Мы будем

использовать выводы предыдущих моделей и объединять их, чтобы увидеть, может ли стековый классификатор работать лучше:

```
>>> from mlxtend.classifier import (
...     StackingClassifier,
... )
>>> clfs = [
...     x()
...     for x in [
...         LogisticRegression,
...         DecisionTreeClassifier,
...         KNeighborsClassifier,
...         GaussianNB,
...         SVC,
...         RandomForestClassifier,
...     ]
... ]
>>> stack = StackingClassifier(
...     classifiers=clfs,
...     meta_classifier=LogisticRegression(),
... )
>>> kfold = model_selection.KFold(
...     n_splits=10, random_state=42
... )
>>> s = model_selection.cross_val_score(
...     stack, X, y, scoring="roc_auc", cv=kfold
... )
>>> print(
...     f"{stack.__class__.__name__} "
...     f"AUC: {s.mean():.3f} STD: {s.std():.2f}"
... )
StackingClassifier   AUC: 0.804   STD: 0.06
```

В данном случае похоже, что немного снизились производительность и стандартное отклонение.

Создание модели

Для создания модели я собираюсь использовать классификатор случайного леса. Это гибкая модель, которая обычно

дает приличные результаты. Не забудьте обучить его (вызывая `.fit`) на учебных данных, которые мы ранее разбили на учебный набор и тестовый:

```
>>> rf = ensemble.RandomForestClassifier(
... n_estimators=100, random_state=42
... )
>>> rf.fit(X_train, y_train)
RandomForestClassifier(bootstrap=True,
   class_weight=None, criterion='gini',
   max_depth=None, max_features='auto',
   max_leaf_nodes=None,
   min_impurity_decrease=0.0,
   min_impurity_split=None,
   min_samples_leaf=1, min_samples_split=2,
   min_weight_fraction_leaf=0.0, n_estimators=10,
   n_jobs=1, oob_score=False, random_state=42,
   verbose=0, warm_start=False)
```

Оценка модели

Теперь, когда у нас есть модель, мы можем использовать тестовые данные, чтобы увидеть, насколько хорошо модель обобщает данные, которых она не видела раньше. Метод `.score` классификатора возвращает среднее значение точности прогнозирования. Мы хотим убедиться, что вызываем метод `.score` с тестовыми данными (предположительно с учебными данными он должен работать лучше):

```
>>> rf.score(X_test, y_test)
0.7964376590330788
```

Мы также можем рассмотреть и другие показатели, такие как точность:

```
>>> metrics.precision_score(
... y_test, rf.predict(X_test)
... )
0.8013698630136986
```

Большим преимуществом древовидных моделей является то, что вы можете проверить важность признака. Важность

признака говорит о том, насколько он влияет на модель. Обратите внимание, что удаление признака не означает, что оценка будет соответственно снижаться, поскольку другие признаки могут быть коллинеарными (в этом случае мы могли бы удалить столбец sex_male или sex_female, поскольку он имеет идеальную отрицательную корреляцию):

```
>>> for col, val in sorted(
...     zip(
...         X_train.columns,
...         rf.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
age             0.277
fare            0.265
sex_female      0.240
pclass          0.092
sibsp           0.048
```

Важность признака рассчитывается с учетом увеличения ошибки. Если удаление признака увеличивает ошибку в модели, этот признак важен.

Мне действительно нравится библиотека SHAP, применяемая и для изучения признаков, которые модель считает важными, и для объяснения прогнозов. Эта библиотека работает с моделями черного ящика, и мы покажем ее позже.

Оптимизация модели

Модели имеют *гиперпараметры* (hyperparameter), которые контролируют их поведение. Изменяя значения этих параметров, мы изменяем их производительность. Библиотека Scikit-learn имеет класс сеточного поиска для оценки модели с различными комбинациями параметров и получения наилучшего результата. Мы можем использовать эти параметры для создания экземпляра класса модели:

```
>>> rf4 = ensemble.RandomForestClassifier()
>>> params = {
...     "max_features": [0.4, "auto"],
...     "n_estimators": [15, 200],
...     "min_samples_leaf": [1, 0.1],
...     "random_state": [42],
... }
>>> cv = model_selection.GridSearchCV(
...     rf4, params, n_jobs=-1
... ).fit(X_train, y_train)
>>> print(cv.best_params_)
{'max_features': 'auto', 'min_samples_leaf': 0.1,
'n_estimators': 200, 'random_state': 42}

>>> rf5 = ensemble.RandomForestClassifier(
...     **{
...         "max_features": "auto",
...         "min_samples_leaf": 0.1,
...         "n_estimators": 200,
...         "random_state": 42,
...     }
... )
>>> rf5.fit(X_train, y_train)
>>> rf5.score(X_test, y_test)
0.7888040712468194
```

Для оптимизации различных метрик мы можем передать в `GridSearchCV` параметр `scoring`. Список метрик и их значений приведен в главе 12.

Матрица неточностей

Матрица неточностей (confusion matrix) позволяет увидеть правильные классификации, а также ложно позитивные и ложно негативные случаи. Может случиться так, что мы хотим оптимизировать в сторону ложно позитивных или ложно негативных случаев, а различные модели и параметры могут изменить это. Мы можем использовать библиотеку Scikit-learn для получения текстовой версии или библиотеку Yellowbrick для построения графика (рис. 3.4):

```
>>> from sklearn.metrics import confusion_matrix
>>> y_pred = rf5.predict(X_test)
>>> confusion_matrix(y_test, y_pred)
array([[196, 28],
       [ 55, 114]])

>>> mapping = {0: "died", 1: "survived"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
...     rf5,
...     classes=["died", "survived"],
...     label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig(
...     "images/mlpr_0304.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

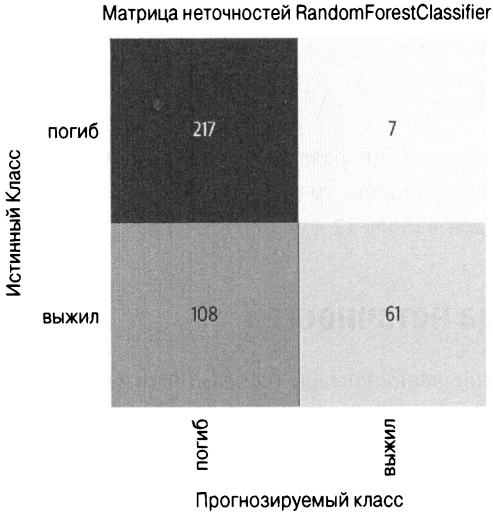


Рис. 3.4. *Матрица неточностей Yellowbrick. Это полезный инструмент оценки, который представляет прогнозируемый класс по горизонтали и истинный класс по вертикали. Хороший классификатор будет иметь все значения по диагонали и нули в других ячейках*

Кривая ROC

Кривая рабочей характеристики приемника (Receiver Operating Characteristic — ROC) является распространенным инструментом, используемым для оценки классификаторов. Измеряя *площадь под кривой* (Area Under The Curve — AUC), мы можем получить метрику для сравнения различных классификаторов (рис. 3.5). Он отображает соотношение истинно позитивных и ложно позитивных случаев. Для расчета AUC мы можем использовать Scikit-learn:

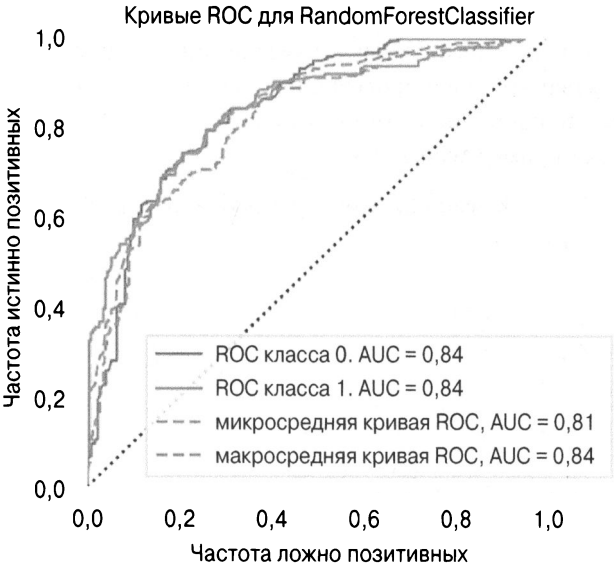


Рис. 3.5. *Кривая ROC демонстрирует соотношение между истинно позитивными случаями и ложно позитивными. В общем, чем дальше, тем лучше. Измерение AUC дает для оценки одно число. Чем ближе к единице, тем лучше. Ниже 0,5 — модель плохая*

```
>>> y_pred = rf5.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

или Yellowbrick — для визуализации графика:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(rf5)
>>> roc_viz.score(X_test, y_test)
0.8279691030696217
>>> roc_viz.poof()
>>> fig.savefig("images/mlpr_0305.png")
```

Кривая обучения

Кривая обучения (learning curve) указывает, достаточно ли у нас данных для обучения. Она обучает модель на последовательных порциях данных и измеряет оценку (рис. 3.6). Если оценка перекрестной проверки продолжает расти, то нам, возможно, придется инвестировать в сбор дополнительных данных. Вот пример Yellowbrick:

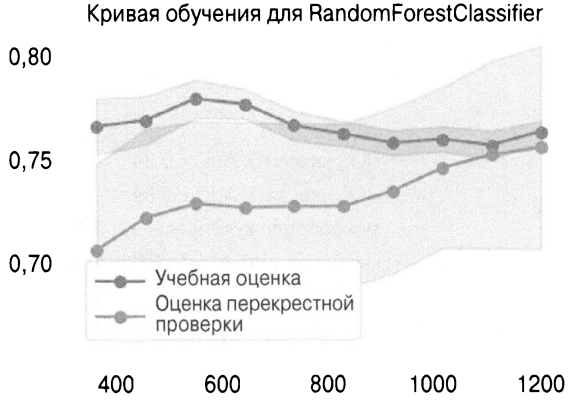


Рис. 3.6. *Эта кривая обучения показывает, что по мере добавления обучающих выборок наши оценки перекрестной проверки (тестирования), по-видимому, улучшаются*

```
>>> import numpy as np
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> cv = StratifiedKFold(12)
>>> sizes = np.linspace(0.3, 1.0, 10)
>>> lc_viz = LearningCurve(
```

```
...     rf5,
...     cv=cv,
...     train_sizes=sizes,
...     scoring="f1_weighted",
...     n_jobs=4,
...     ax=ax,
... )
>>> lc_viz.fit(X, y)
>>> lc_viz.poof()
>>> fig.savefig("images/mlpr_0306.png")
```

Развертывание модели

Используя модуль `pickle` в Python, мы можем сохранять и загружать модели. Получив модель, мы вызываем метод `.predict`, чтобы получить результат классификации или регрессии:

```
>>> import pickle
>>> pic = pickle.dumps(rf5)
>>> rf6 = pickle.loads(pic)
>>> y_pred = rf6.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

Использование *Flask* для развертывания прогнозирующей веб-службы очень распространено. В настоящее время появляются другие коммерческие продукты с открытым исходным кодом, которые поддерживают развертывание. Среди них — Clipper, Pipeline и Google Cloud Machine Learning Engine.

ГЛАВА 4

Пропущенные данные

В этой главе речь пойдет о том, что делать при пропуске данных. Пример был приведен в предыдущей главе, а здесь мы углубимся в эту тему немного больше. Большинство алгоритмов не будут работать, если данных нет. Заметным исключением являются недавние бустинговые библиотеки: XGBoost, Cat-Boost и LightGBM.

Как и во многих случаях в машинном обучении, нет точных рекомендаций по поводу того, как поступать при отсутствии данных. Кроме того, пропуск в данных может свидетельствовать о различных ситуациях. Представьте, что в данных переписи пропущены возрастные данные. Это потому, что опрашиваемый не захотел раскрывать свой возраст? Он не знал своего возраста? Тот, кто задавал вопросы, забыл даже спросить о возрасте? Есть ли какой-то шаблон для пропущенных возрастов? Коррелирует ли это с другим признаком? Или это произошло совершенно случайно?

Есть несколько способов справиться с пропуском данных.

- Удалить все строки с пропущенными данными
- Удалить любой столбец с пропущенными данными
- Дополнить пропущенные значения
- Создать столбец индикатора, чтобы указать, что данные пропущены

Изучение пропущенных данных

Давайте вернемся к данным о Титанике. Поскольку Python рассматривает значения `True` и `False` как `1` и `0` соответственно, мы можем использовать этот трюк в pandas, чтобы получить процент пропущенных данных:

```
>>> df.isnull().mean() * 100
pclass        0.000000
survived      0.000000
name          0.000000
sex           0.000000
age          20.091673
sibsp         0.000000
parch         0.000000
ticket        0.000000
fare          0.076394
cabin        77.463713
embarked      0.152788
boat         62.872422
body         90.756303
home.dest    43.086325
dtype: float64
```

Для визуализации шаблонов в пропущенных данных используйте *библиотеку missingno*. Она полезна для просмотра непрерывных областей пропущенных данных, что указывает на то, что пропуск данных не является случайным (рис. 4.1). Функция `matrix` включает *спарклайн* (sparkline) справа. Шаблоны здесь также указывают на неслучайность пропуска данных. Чтобы увидеть шаблоны, возможно, придется ограничить количество выборок:

```
>>> import missingno as msno
>>> ax = msno.matrix(orig_df.sample(500))
>>> ax.get_figure().savefig("images/mlpr_0401.png")
```

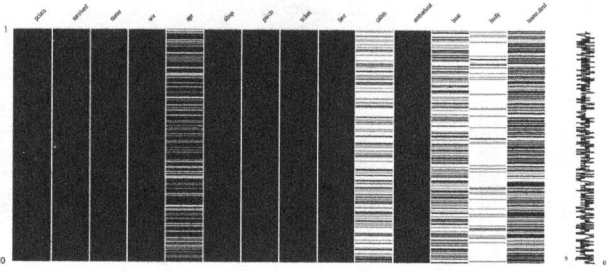


Рис. 4.1. *Где пропущены данные. Автору четкие шаблоны не очевидны*

С помощью pandas можно создать гистограмму пропущенных данных (рис. 4.2):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> (1 - df.isnull().mean()).abs().plot.bar(ax=ax)
>>> fig.savefig("images/mlpr_0402.png", dpi=300)
```

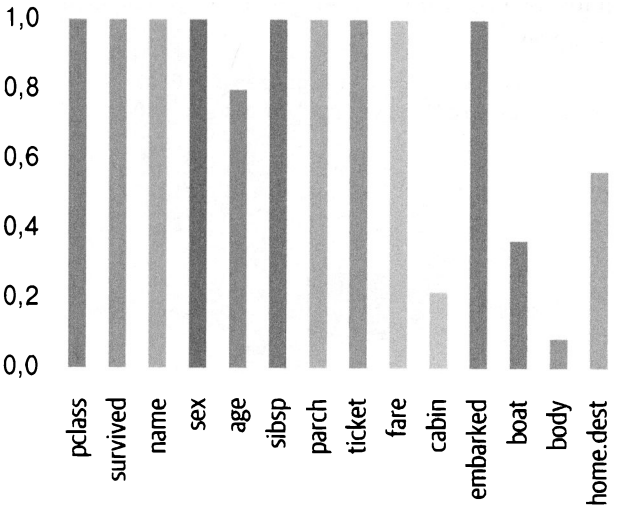


Рис. 4.2. *Процент присутствующих данных. В столбцах* `boat` *и* `body` *утечка, поэтому их следует игнорировать. Интересно, что пропущены некоторые данные столбца* `age` *(возраст)*

Для создания того же графика можно использовать библиотеку missingno (рис. 4.3):

```
>>> ax = msno.bar(orig_df.sample(500))
>>> ax.get_figure().savefig("images/mlpr_0403.png")
```

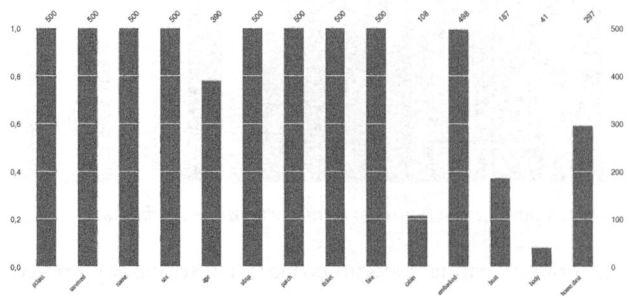


Рис. 4.3. *Процент присутствующих и пропущенных данных*

Кроме того, можно создать *тепловую карту* (heat map), показывающую, есть ли корреляции в пропуске данных (рис. 4.4). В данном случае не похоже, что места, где пропущены данные, коррелируют:

```
>>> ax = msno.heatmap(df, figsize=(6, 6))
>>> ax.get_figure().savefig("/tmp/mlpr_0404.png")
```

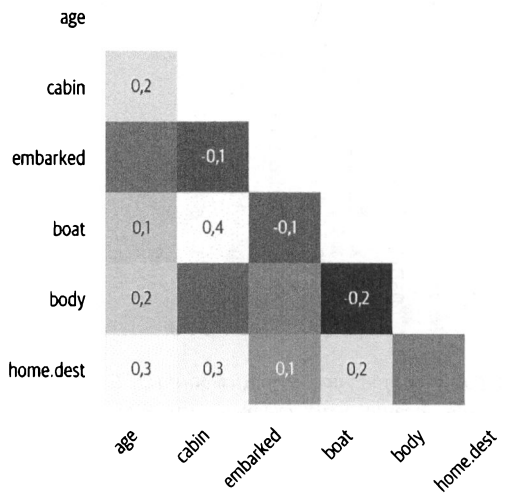


Рис. 4.4. *Корреляции присутствующих данных с пропущенными*

Мы можем создать *дендрограмму* (dendrogram), демонстрирующую кластеры, в которых пропущены данные (рис. 4.5). Листья, находящиеся на одном уровне, прогнозируют присутствие друг друга (пусто или заполнено). Вертикальные линии используются для обозначения различных кластеров. Короткие линии означают, что ветви схожи:

```
>>> ax = msno.dendrogram(df)
>>> ax.get_figure().savefig("images/mlpr_0405.png")
```

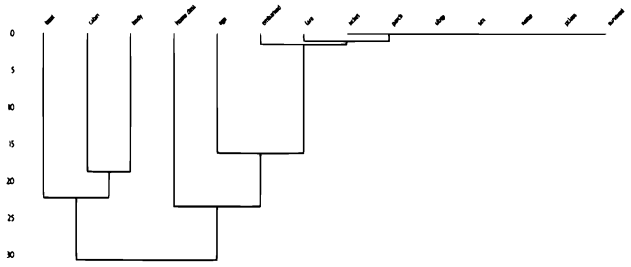


Рис. 4.5. *Дендрограмма присутствующих и пропущенных данных. В правом верхнем углу видны столбцы без пропущенных данных*

Отбрасывание пропущенных данных

С помощью метода `.dropna` библиотеки pandas можно отбросить все строки с пропущенными данными:

```
>>> df1 = df.dropna()
```

Чтобы удалить столбцы, можно отметить, какие столбцы пропущены, и использовать метод `.drop`. Можно передать список имен столбцов или одно имя столбца:

```
>>> df1 = df.drop(columns="cabin")
```

В качестве альтернативы можно использовать метод `.dropna` и установить `axis = 1` (отбрасывание вдоль оси столбца):

```
>>> df1 = df.dropna(axis=1)
```

Будьте осторожны при отбрасывании данных! Я обычно рассматриваю это как последний вариант.

Замещение данных

Если у вас есть инструмент прогнозирования данных, можете использовать его для прогнозирования пропущенных данных. Общая задача определения значений для пропущенных значений называется *замещением* (imputation).

Если вы подставляете данные, создайте конвейер и используйте ту же логику замещения во время создания модели и прогнозирования. Класс `SimpleImputer` библиотеки Scikit-learn будет вычислять среднее значение, медиану и наиболее частое значение признака.

Стандартное поведение класса — вычисление среднего значения:

```
>>> from sklearn.impute import SimpleImputer
>>> num_cols = df.select_dtypes(
...     include="number"
... ).columns
>>> im = SimpleImputer() # среднее
>>> imputed = im.fit_transform(df[num_cols])
```

Чтобы изменить подставляемое значение на медиану или наиболее распространенное значение, задайте `strategy='median'` или `strategy='most_frequent'` соответственно. Если вы хотите заполнять постоянным значением, скажем, `-1`, используйте `strategy='constant'` в сочетании с `fill_value=-1`.

СОВЕТ

Чтобы вычислять пропущенные значения, можно также использовать метод `.fillna` библиотеки pandas. Убедитесь, что вы не пропускаете данные. Если вы заполняете средним значением, убедитесь, что используется одно и то же среднее значение как во время создания модели, так и при прогнозировании.

Стратегии постоянного и наиболее частого значений могут использоваться с числовыми или строковыми данными. Среднее значение и медиана требуют числовых данных.

Библиотека fancyimpute реализует множество алгоритмов и следует интерфейсу Scikit-learn. К сожалению, большинство алгоритмов являются *трансдуктивными* (transductive), а это означает, что вы не можете вызвать метод `.transform` сам по себе после подгонки алгоритма. Метод `IterativeImputer` является *индуктивным* (inductive) (он был переведен из fancyimpute в Scikit-learn) и поддерживает преобразование после подгонки.

Добавление индикаторных столбцов

Пропуск данных сам по себе способен дать модели некий сигнал. Библиотека pandas может добавить новый столбец, чтобы указать, что значение отсутствует:

```
>>> def add_indicator(col):
...     def wrapper(df):
...         return df[col].isna().astype(int)
...
...     return wrapper

>>> df1 = df.assign(
...     cabin_missing=add_indicator("cabin")
... )
```

ГЛАВА 5

Очистка данных

Для очистки данных можно использовать как общие инструменты, такие как pandas, так и специализированные, такие как pyjanitor.

Имена столбцов

При использовании pandas наличие дружественных к Python имен столбцов делает возможным доступ к атрибутам. Функция `clean_names` библиотеки pyjanitor возвращает объект DataFrame со столбцами в нижнем регистре и пробелами, замененными подчеркиванием:

```
>>> import janitor as jn
>>> Xbad = pd.DataFrame(
...     {
...         "A": [1, None, 3],
...         " sales numbers ": [20.0, 30.0, None],
...     }
... )
>>> jn.clean_names(Xbad)
     a  _sales_numbers_
0  1.0              20.0
1  NaN              30.0
2  3.0               NaN
```

СОВЕТ

Я рекомендую обновлять столбцы с помощью присвоения индекса, метода `.assign` и присвоения `.loc` или `.iloc`. Я также рекомендую не использовать присвоение атрибутов для обновления столбцов в pandas. Из-за риска перезаписи существующих методов с теми же именами, что и у столбца, назначение атрибутов не гарантируется.

Библиотека pyjanitor удобна, но не допускает использование в столбцах пробелов. Для более детального управления переименованием столбцов мы можем использовать библиотеку pandas:

```
>>> def clean_col(name):
...     return (
...         name.strip().lower().replace(" ", "_")
...     )

>>> Xbad.rename(columns=clean_col)
     a  sales_numbers
0  1.0           20.0
1  NaN           30.0
2  3.0            NaN
```

Замена пропущенных значений

Функция `coalesce` из pyjanitor получает объект DataFrame и список столбцов для рассмотрения. Это похоже на функцию в базах данных Excel и SQL. Она возвращает для каждой строки первое ненулевое значение:

```
>>> jn.coalesce(
...     Xbad,
...     columns=["A", " sales numbers "],
...     new_column_name="val",
... )
```

```
    val
0   1.0
1  30.0
2   3.0
```

Для того чтобы заполнить пропущенные значения определенным значением, можно использовать метод `.fillna` объекта DataFrame:

```
>>> Xbad.fillna(10)
      A    sales numbers
0   1.0              20.0
1  10.0              30.0
2   3.0              10.0
```

или функцию `fill_empty` библиотеки pyjanitor:

```
>>> jn.fill_empty(
...     Xbad,
...     columns=["A", " sales numbers "],
...     value=10,
... )
      A   sales numbers
0   1.0             20.0
1  10.0             30.0
2   3.0             10.0
```

Зачастую для замены нулевым значением в каждом столбце мы будем использовать более тонкие замещения в pandas, Scikit-learn или fancyimpute.

В качестве "санитарной" проверки перед созданием моделей вы можете использовать библиотеку pandas, чтобы убедиться, что вы справились со всеми пропущенными значениями. Следующий код возвращает единственное логическое значение, если в объекте DataFrame отсутствует какая-либо ячейка:

```
>>> df.isna().any().any()
True
```

ГЛАВА 6

Исследование

Говорят, что проще взять эксперта в предметной области и обучить его науке о данных, чем наоборот. Я не уверен, что согласен с этим на все 100%, но правда в том, что данные имеют нюансы, и эксперт может помочь в них разобраться. Понимая предметную область и данные, они могут создавать лучшие модели и оказывать лучшее влияние на свой бизнес.

Прежде чем создать модель, я делаю некий предварительный анализ данных. Это не только дает мне представление о данных, но и является отличным поводом для встреч и обсуждения проблем с организациями, контролирующими эти данные.

Размер данных

Здесь мы снова используем набор данных Titanic. Свойство `.shape` библиотеки pandas вернет кортеж из количества строк и столбцов:

```
>>> X.shape
(1309, 13)
```

Как мы можем видеть, этот набор данных имеет 1309 строк и 13 столбцов.

Сводная статистика

Чтобы получить сводную статистику для наших данных, можно использовать библиотеку pandas. Метод `.describe` даст также количество значений, отличных от `NaN`. Давайте посмотрим на результаты для первого и последнего столбцов:

```
>>> X.describe().iloc[:, [0, -1]]
            pclass       embarked_S
count  1309.000000      1309.000000
mean     -0.012831         0.698243
std       0.995822         0.459196
min      -1.551881         0.000000
25%      -0.363317         0.000000
50%       0.825248         1.000000
75%       0.825248         1.000000
max       0.825248         1.000000
```

Итоговая строка говорит нам, что оба эти столбца заполнены. Пропущенных значений нет. У нас также есть среднее значение, стандартное отклонение, минимальное, максимальное и квартильное значения.

НА ЗАМЕТКУ

Объект DataFrame библиотеки pandas имеет атрибут `iloc`, с которым можно выполнять операции индексации. Он позволяет выбирать строки и столбцы по положению индекса. Мы передаем позиции строк как скаляр, список или срез массива, а затем можем добавить запятую и передать позиции столбцов как скаляр, список или срез массива.

Здесь мы передаем вторую и пятую строки, а также три последних столбца:

```
>>> X.iloc[[1, 4], -3:]
     sex_male  embarked_Q  embarked_S
677       1.0           0           1
864       0.0           0           1
```

Существует также атрибут `.loc`, и мы можем выводить строки и столбцы на основе имени (а не позиции). Вот та же часть DataFrame:

```
>>> X.loc[[677, 864], "sex_male":]
     sex_male  embarked_Q  embarked_S
677       1.0           0           1
864       0.0           0           1
```

Гистограмма

Гистограмма (histogram) — это отличный инструмент для визуализации числовых данных. Вы можете увидеть, сколько режимов существует, а также посмотреть распределение (рис. 6.1). В библиотеке pandas есть метод `.plot` для отображения гистограмм:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.fare.plot(kind="hist", ax=ax)
>>> fig.savefig("images/mlpr_0601.png", dpi=300)
```

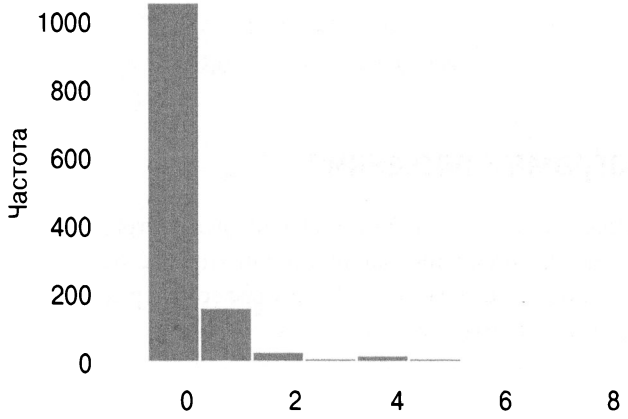


Рис. 6.1. *Гистограмма pandas*

Используя *библиотеку seaborn*, мы можем построить гистограмму непрерывных значений относительно цели (рис. 6.2):

```
fig, ax = plt.subplots(figsize=(12, 8))
mask = y_train == 1
ax = sns.distplot(X_train[mask].fare, label='survived')
ax = sns.distplot(X_train[~mask].fare, label='died')
ax.set_xlim(-1.5, 1.5)
ax.legend()
fig.savefig('images/mlpr_0602.png', dpi=300,
bbox_inches='tight')
```

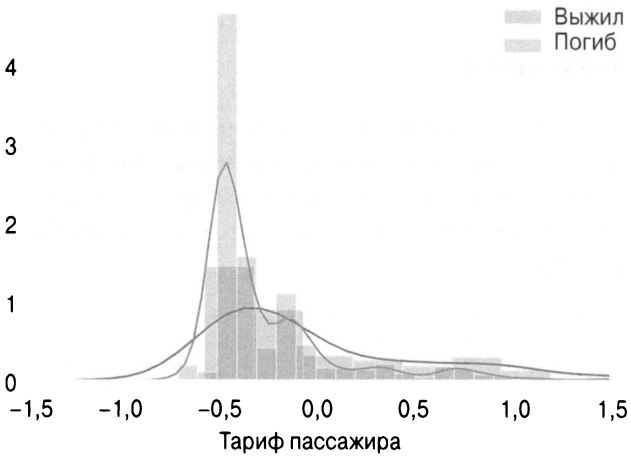


Рис. 6.2. *Гистограмма seaborn*

Диаграмма рассеяния

Диаграмма рассеяния (scatter plot) демонстрирует взаимосвязь между двумя числовыми столбцами (рис. 6.3). С библиотекой pandas это тоже легко. Если у вас есть перекрывающиеся данные, настройте параметр `alpha`:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.plot.scatter(
...     x="age", y="fare", ax=ax, alpha=0.3
... )
>>> fig.savefig("images/mlpr_0603.png", dpi=300)
```

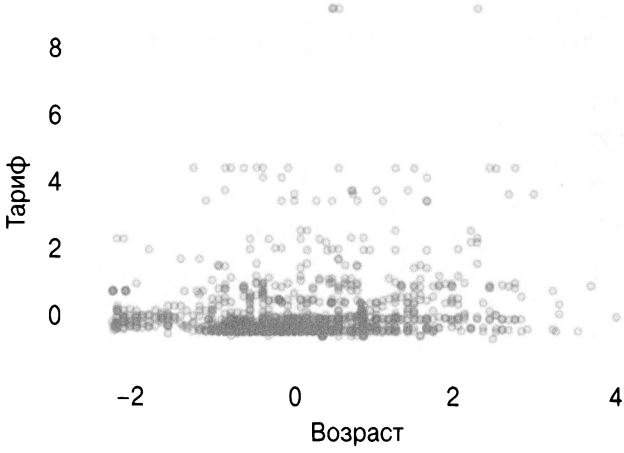


Рис. 6.3. *Диаграмма рассеяния pandas*

Похоже, между этими двумя признаками нет большой корреляции. Чтобы определить корреляцию, мы можем использовать *корреляцию Пирсона* (Pearson correlation) между двумя столбцами с помощью метода `.corr` (библиотека pandas):

```
>>> X.age.corr(X.fare)
0.17818151568062093
```

Объединенный график

В библиотеке Yellowbrick есть причудливый график рассеяния, по краям которого содержатся гистограммы, а также линия регрессии — это *объединенный график* (joint plot) (рис. 6.4):

```
>>> from yellowbrick.features import (
...     JointPlotVisualizer,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> jpv = JointPlotVisualizer(
...     feature="age", target="fare"
... )
>>> jpv.fit(X["age"], X["fare"])
```

```
>>> jpv.poof()
>>> fig.savefig("images/mlpr_0604.png", dpi=300)
```

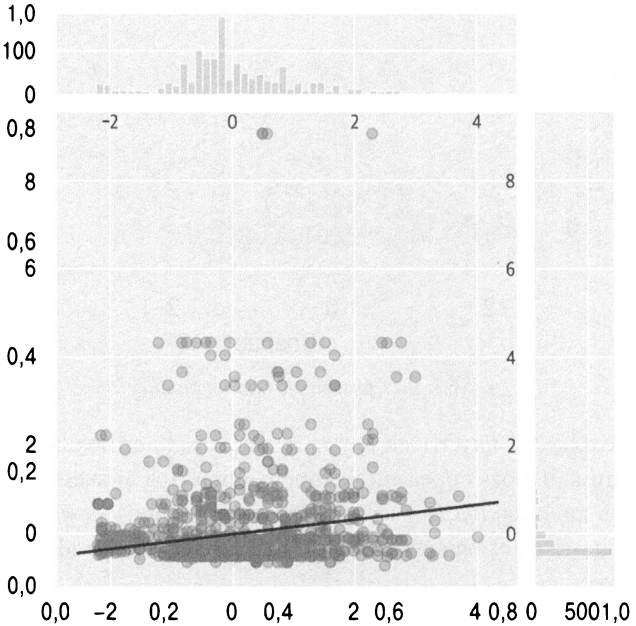


Рис. 6.4. *Объединенный график Yellowbrick*

ВНИМАНИЕ

`X` и `y` в этом методе `.fit` ссылаются на каждый столбец. Обычно `X` — это объект DataFrame, а не серия.

Для создания объединенного графика (рис. 6.5) также можно использовать библиотеку *seaborn*:

```
>>> from seaborn import jointplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> p = jointplot(
...     "age", "fare", data=new_df, kind="reg"
```

```
... )
>>> p.savefig("images/mlpr_0605.png", dpi=300)
```

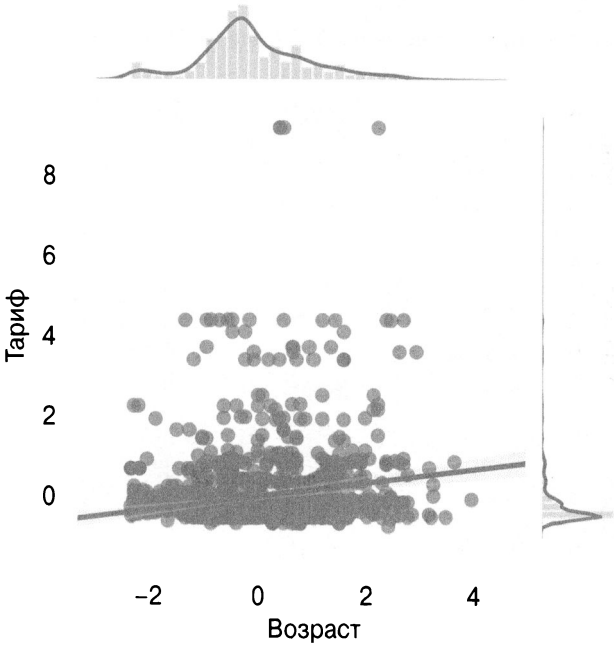


Рис. 6.5. *Объединенный график seaborn*

Парная сетка

Библиотека seaborn позволяет создать *парную сетку* (pair grid) (рис. 6.6). Этот график представляет собой матрицу столбцов и *ядерных оценок плотности* (kernel density estimate). Чтобы закрасить столбец из DataFrame, используем параметр hue. Раскрасив цели, можно увидеть, влияют ли объекты на цель по-разному:

```
>>> from seaborn import pairplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
```

```
>>> vars = ["pclass", "age", "fare"]
>>> p = pairplot(
...     new_df, vars=vars, hue="target", kind="reg"
... )
>>> p.savefig("images/mlpr_0606.png", dpi=300)
```

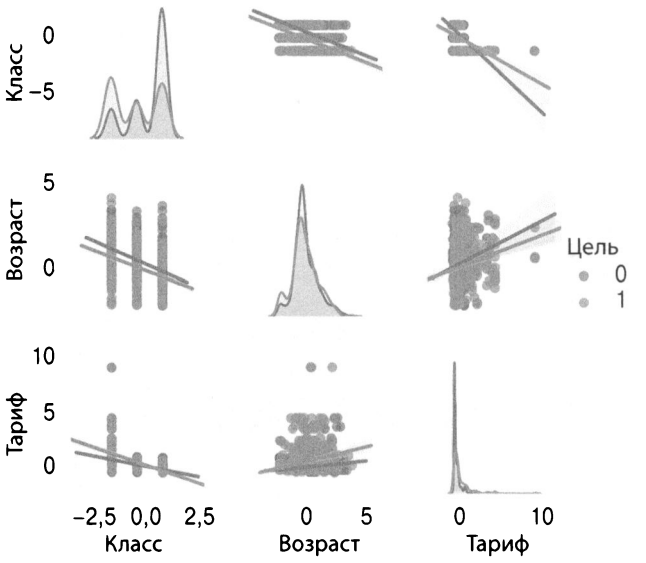


Рис. 6.6. *Парная сетка seaborn*

Диаграмма размаха и скрипичная диаграмма размаха

Библиотека seaborn имеет различные диаграммы для визуализации распределений. Мы рассмотрим примеры *диаграммы размаха* (box plot) и *скрипичной диаграммы размаха* (violin plot) (рис. 6.7 и 6.8). Эти графики могут визуализировать отношение признака и цели:

```
>>> from seaborn import box plot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> boxplot(x="target", y="age", data=new_df)
>>> fig.savefig("images/mlpr_0607.png", dpi=300)
```

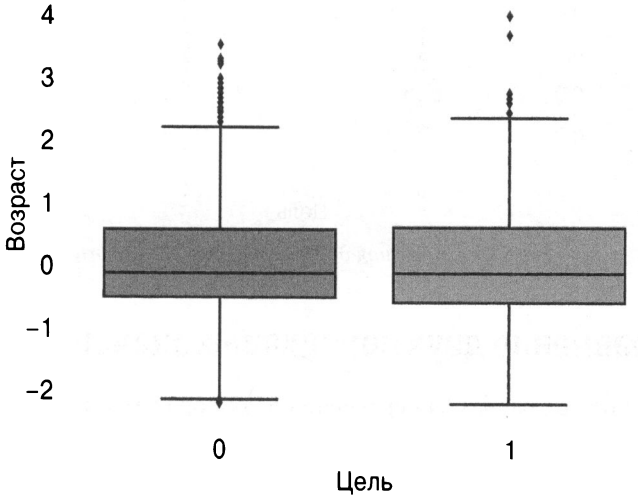


Рис. 6.7. *Диаграмма размаха seaborn*

Скрипичная диаграмма размаха может помочь с визуализацией распределения:

```
>>> from seaborn import violinplot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> violinplot(
...     x="target", y="sex_male", data=new_df
... )
>>> fig.savefig("images/mlpr_0608.png", dpi=300)
```

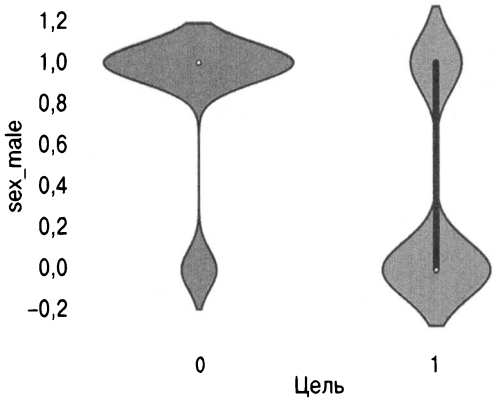


Рис. 6.8. *Скрипичная диаграмма размаха seaborn*

Сравнение двух порядковых значений

Вот код pandas для сравнения двух порядковых категорий. Я имитирую это, разделив `age` (возраст) на десять квантилей и `pclass` (класс пассажира) — на три. График нормализован и поэтому заполняет всю вертикальную область. Это позволяет легко увидеть, что в квантиле 40% большинство билетов было в 3-м классе (рис. 6.9):

```
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> (
...     X.assign(
...         age_bin=pd.qcut(
...             X.age, q=10, labels=False
...         ),
...         class_bin=pd.cut(
...             X.pclass, bins=3, labels=False
...         ),
...     )
...     .groupby(["age_bin", "class_bin"])
...     .size()
...     .unstack()
...     .pipe(lambda df: df.div(df.sum(1), axis=0))
...     .plot.bar(
```

```
...         stacked=True,
...         width=1,
...         ax=ax,
...         cmap="viridis",
...     )
...     .legend(bbox_to_anchor=(1, 1))
... )
>>> fig.savefig(
...     "image/mlpr_0609.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

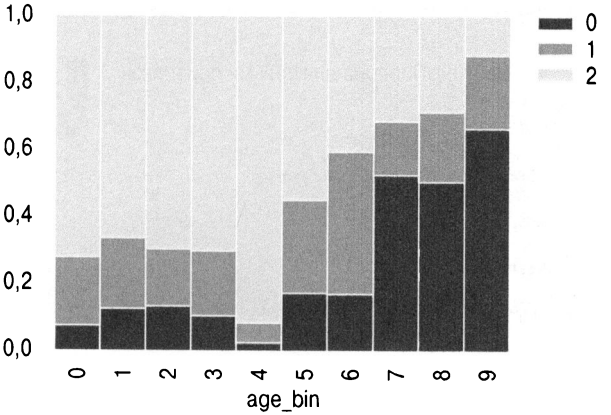


Рис. 6.9. *Сравнение порядковых значений*

НА ЗАМЕТКУ

Строки

```
.groupby(["age_bin", "class_bin"])
.size()
.unstack()
```

можно заменить строками

```
.pipe(lambda df: pd.crosstab(
      df.age_bin, df.class_bin)
)
```

В библиотеке pandas зачастую есть несколько способов сделать что-то и доступны некоторые вспомогательные функции, которые формируют другие функции, такие как `pd.crosstab`.

Корреляция

Библиотека Yellowbrick может осуществлять попарные сравнения между объектами (рис. 6.10). На этом графике показана корреляция Пирсона (параметр `algorithm` может также иметь значения `'spearman'` и `'covariance'`):

Рис. 6.10. *Ковариационная корреляция, созданная с помощью Yellowbrick*

```
>>> from yellowbrick.features import Rank2D
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pcv = Rank2D(
...     features=X.columns, algorithm="pearson"
... )
```

```
>>> pcv.fit(X, y)
>>> pcv.transform(X)
>>> pcv.poof()
>>> fig.savefig(
...     "images/mlpr_0610.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Аналогичный график, тепловая карта, доступен в библиотеке seaborn (рис. 6.11). Нам нужно передать корреляцию DataFrame в качестве данных. К сожалению, цветовая шкала не распространяется от –1 до 1, если только это не делают значения в матрице или мы не добавляем параметры `vmin` и `vmax`:

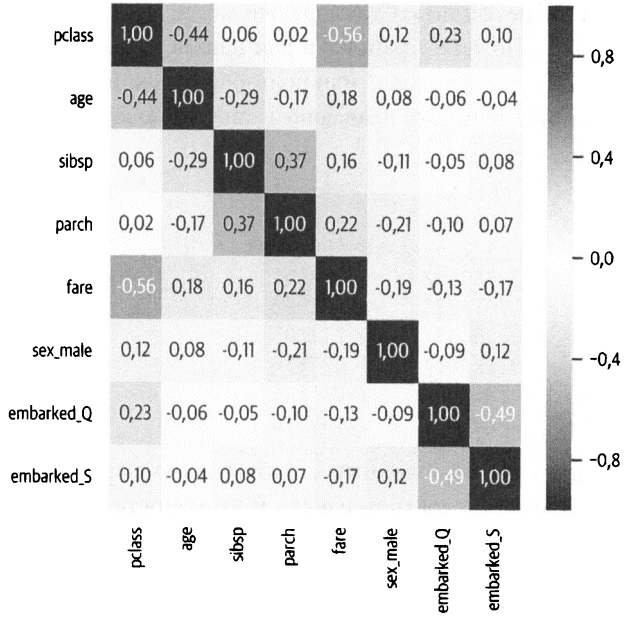


Рис. 6.11. *Тепловая карта seaborn*

```
>>> from seaborn import heatmap
>>> fig, ax = plt.subplots(figsize=(8, 8))
>>> ax = heatmap(
...     X.corr(),
```

```
...     fmt=".2f",
...     annot=True,
...     ax=ax,
...     cmap="RdBu_r",
...     vmin=-1,
...     vmax=1,
... )
>>> fig.savefig(
...     "images/mlpr_0611.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Библиотека pandas также может отобразить корреляцию между столбцами DataFrame. Мы показываем только первые два столбца результата. Стандартный метод — это `'pearson'`, но вы также можете установить для параметра `method` значения `'kendall'`, `'spearman'` или пользовательский вызов, который возвращает число с плавающей запятой по двум столбцам:

```
>>> X.corr().iloc[:, :2]
              pclass       age
pclass      1.000000 -0.440769
age        -0.440769  1.000000
sibsp       0.060832 -0.292051
parch       0.018322 -0.174992
fare       -0.558831  0.177205
sex_male    0.124617  0.077636
embarked_Q  0.230491 -0.061146
embarked_S  0.096335 -0.041315
```

Сильно коррелированные столбцы не имеют ценности и могут снизить важность признака и интерпретацию коэффициента регрессии. Ниже приведен код для поиска коррелированных столбцов. По нашим данным, ни один из столбцов не имеет высокой корреляции (помните, что мы удалили столбец `sex_male`).

Если бы у нас были коррелированные столбцы, мы могли бы удалить из данных столбцы от `level_0` или `level_1`:

```
>>> def correlated_columns(df, threshold=0.95):
...     return (
```

```
...         df.corr()
...         .pipe(
...             lambda df1: pd.DataFrame(
...                 np.tril(df1, k=-1),
...                 columns=df.columns,
...                 index=df.columns,
...             )
...         )
...         .stack()
...         .rename("pearson")
...         .pipe(
...             lambda s: s[
...                 s.abs() > threshold
...             ].reset_index()
...         )
...         .query("level_0 not in level_1")
...     )

>>> correlated_columns(X)
Empty DataFrame
Columns: [level_0, level_1, pearson]
Index: []
```

Использовав набор данных с *б*ольшим количеством столбцов, мы увидим, что многие из них имеют корреляции:

```
>>> c_df = correlated_columns(agg_df)
>>> c_df.style.format({"pearson": "{:.2f}"})
       level_0     level_1   pearson
3   pclass_mean      pclass  1.00
4   pclass_mean  pclass_min  1.00
5   pclass_mean  pclass_max  1.00
6    sibsp_mean   sibsp_max  0.97
7   parch_mean   parch_min  0.95
8   parch_mean   parch_max  0.96
9    fare_mean        fare  0.95
10   fare_mean    fare_max  0.98
12   body_mean    body_min  1.00
13   body_mean    body_max  1.00
14    sex_male  sex_female -1.00
15  embarked_S  embarked_C -0.95
```

RadViz

График RadViz демонстрирует каждую выборку на окружности с признаками на периферии (рис. 6.12). Значения нормализованы, и вы можете представить, что у каждой фигуры есть пружина, которая выталкивает из нее выборки на основании значения.

Это один из методов визуализации разделений между целями.

Это поможет сделать библиотека Yellowbrick:

```
>>> from yellowbrick.features import RadViz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> rv = RadViz(
...     classes=["died", "survived"],
...     features=X.columns,
... )
>>> rv.fit(X, y)
>>> _ = rv.transform(X)
>>> rv.poof()
>>> fig.savefig("images/mlpr_0612.png", dpi=300)
```

Библиотека pandas также может составлять графики RadViz (рис. 6.13):

```
>>> from pandas.plotting import radviz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> radviz(
...     new_df, "target", ax=ax, colormap="PiYG"
... )
>>> fig.savefig("images/mlpr_0613.png", dpi=300)
```

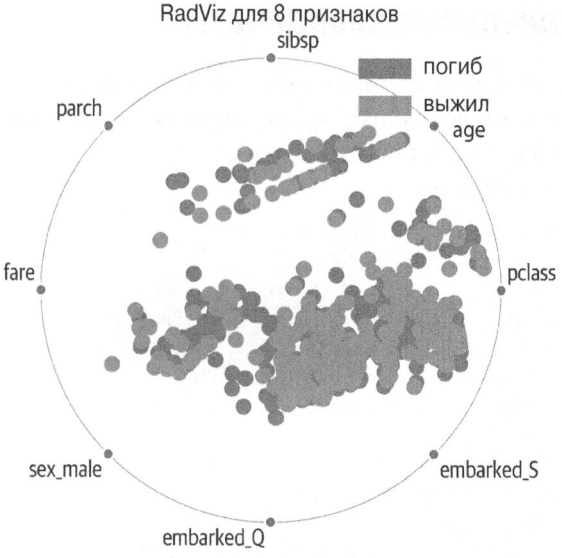


Рис. 6.12. *График RadViz Yellowbrick*

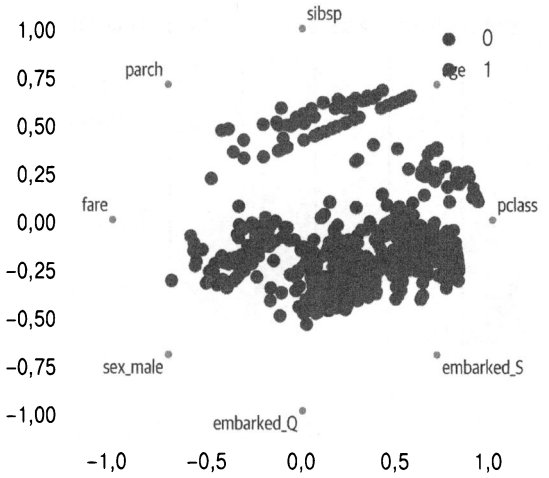


Рис. 6.13. *График RadViz pandas*

Параллельные координаты

Для многомерных данных вы можете использовать график параллельных координат, чтобы визуально представить кластеризацию (рис. 6.14 и 6.15).

Версия Yellowbrick:

```
>>> from yellowbrick.features import (
...     ParallelCoordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pc = ParallelCoordinates(
...     classes=["died", "survived"],
...     features=X.columns,
... )
>>> pc.fit(X, y)
>>> pc.transform(X)
>>> ax.set_xticklabels(
...     ax.get_xticklabels(), rotation=45
... )
>>> pc.poof()
>>> fig.savefig("images/mlpr_0614.png", dpi=300)
```

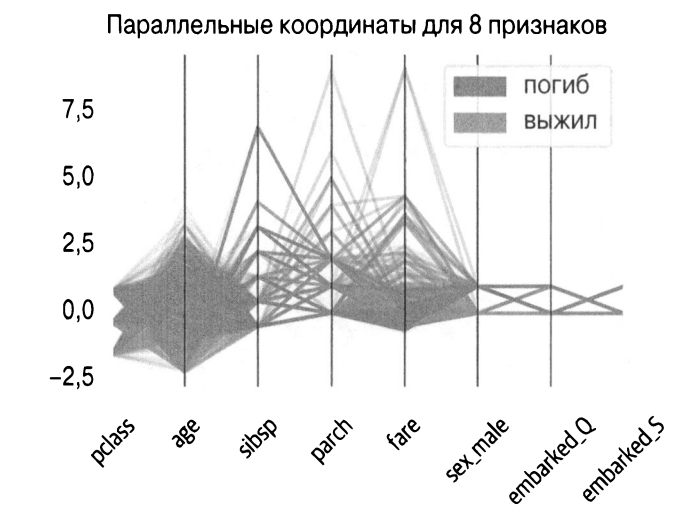


Рис. 6.14. *График параллельных координат Yellowbrick*

И версия pandas:

```
>>> from pandas.plotting import (
...     parallel_coordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> parallel_coordinates(
...     new_df,
...     "target",
...     ax=ax,
...     colormap="viridis",
...     alpha=0.5,
... )
>>> ax.set_xticklabels(
...     ax.get_xticklabels(), rotation=45
... )
>>> fig.savefig(
...     "images/mlpr_0615.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

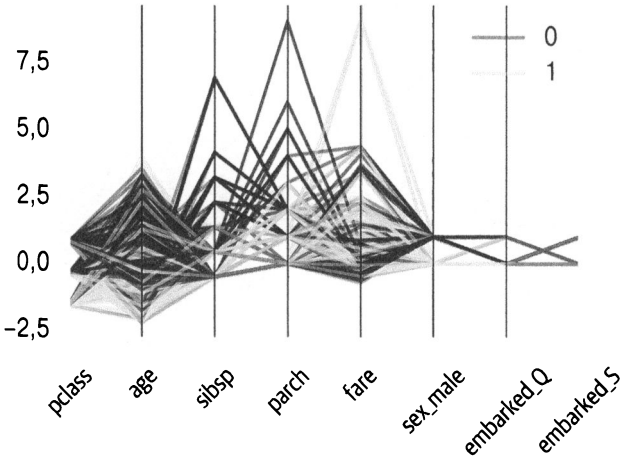


Рис. 6.15. *График параллельных координат pandas*

ГЛАВА 7

Предварительная обработка данных

В этой главе рассмотрены общие этапы предварительной обработки с использованием следующих данных:

```
>>> X2 = pd.DataFrame(
...     {
...         "a": range(5),
...         "b": [-100, -50, 0, 200, 1000],
...     }
... )
>>> X2
   a     b
0  0  -100
1  1   -50
2  2     0
3  3   200
4  4  1000
```

Стандартизация

Некоторые алгоритмы, такие как SVM (Support Vector Machine — метод опорных векторов), работают лучше, когда данные стандартизированы. Каждый столбец должен иметь среднее значение 0 и стандартное отклонение 1. Библиотека Scikit-learn предоставляет метод `.fit_transform`, объединяющий методы `.fit` и `.transform`:

```
>>> from sklearn import preprocessing
>>> std = preprocessing.StandardScaler()
>>> std.fit_transform(X2)
array([[-1.41421356, -0.75995002],
       [-0.70710678, -0.63737744],
       [ 0.        , -0.51480485],
       [ 0.70710678, -0.02451452],
       [ 1.41421356, 1.93664683]])
```

После подгонки (fitting) мы можем проверять различные атрибуты:

```
>>> std.scale_
array([  1.41421356, 407.92156109])
>>> std.mean_
array([  2., 210.])
>>> std.var_
array([2.000e+00, 1.664e+05])
```

Вот версия pandas. Помните, что вам нужно отслеживать исходное среднее и стандартное отклонения, если вы используете их для предварительной обработки. Любая выборка, которую вы будете использовать для прогнозирования позже, должна быть стандартизирована с теми же значениями:

```
>>> X_std = (X2 - X2.mean()) / X2.std()
>>> X_std
          a         b
0 -1.264911  -0.679720
1 -0.632456  -0.570088
2   0.000000 -0.460455
3   0.632456 -0.021926
4   1.264911  1.732190

>>> X_std.mean()
a    4.440892e-17
b    0.000000e+00
dtype: float64

>>> X_std.std()
a    1.0
b    1.0
dtype: float64
```

Библиотека fastai также реализует это:

```
>>> X3 = X2.copy()
>>> from fastai.structured import scale_vars
>>> scale_vars(X3, mapper=None)
>>> X3.std()
a    1.118034
b    1.118034
dtype: float64
>>> X3.mean()
a    0.000000e+00
b    4.440892e-17
dtype: float64
```

Масштабирование до диапазона

Масштабирование до диапазона преобразует данные так, чтобы они находились в диапазоне от 0 до 1 включительно. Ограничение данных может быть полезным. Но если у вас есть выбросы, вам, вероятно, придется соблюдать осторожность, используя это:

```
>>> from sklearn import preprocessing
>>> mms = preprocessing.MinMaxScaler()
>>> mms.fit(X2)
>>> mms.transform(X2)
array([[0.     , 0.     ],
       [0.25   , 0.04545],
       [0.5    , 0.09091],
       [0.75   , 0.27273],
       [1.     , 1.     ]])
```

Вот версия pandas:

```
>>> (X2 - X2.min()) / (X2.max() - X2.min())
      a         b
0  0.00  0.000000
1  0.25  0.045455
2  0.50  0.090909
3  0.75  0.272727
4  1.00  1.000000
```

Фиктивные переменные

Мы можем использовать библиотеку pandas для создания фиктивных переменных из категориальных данных. Это называется также *унитарным кодированием* (one-hot encoding) или *индикаторным кодированием* (indicator encoding). *Фиктивные переменные* (dummy variable) особенно полезны, если данные являются номинальными (неупорядоченными). Функция `get_dummies` библиотеки pandas создает для категориального столбца несколько столбцов, каждый с 1 или 0, если исходный столбец имел это значение:

```
>>> X_cat = pd.DataFrame(
...     {
...         "name": ["George", "Paul"],
...         "inst": ["Bass", "Guitar"],
...     }
... )
>>> X_cat
     name   inst
0  George   Bass
1    Paul Guitar
```

Вот версия pandas. Обратите внимание, что параметр `drop_first` может использоваться для исключения столбца (один из фиктивных столбцов является линейной комбинацией других столбцов):

```
>>> pd.get_dummies(X_cat, drop_first=True)
   name_Paul  inst_Guitar
0          0            0
1          1            1
```

Библиотека pyjanitor также имеет возможность разделять столбцы с помощью функции `expand_column`:

```
>>> X_cat2 = pd.DataFrame(
...     {
...         "A": [1, None, 3],
...         "names": [
...             "Fred,George",
```

```
...                 "George",
...                 "John,Paul",
...             ],
...         }
... )
>>> jn.expand_column(X_cat2, "names", sep=",")
     A         names  Fred  George  John  Paul
0  1.0  Fred,George     1       1     0     0
1  NaN        George     0       1     0     0
2  3.0     John,Paul     0       0     1     1
```

Если у нас есть большое количество номинальных данных, мы можем использовать *меточное кодирование* (label encoding). Это обсуждается в следующем разделе.

Меточное кодирование

Альтернативой фиктивным переменным является меточное кодирование. Оно получает категориальные данные и назначает каждому значению номер. Это полезно для больших количеств данных. Этот кодер назначает порядковый номер, который может или не может быть желательным. Он может занимать меньше места, чем унитарное кодирование, и некоторые (древовидные) алгоритмы вполне могут справиться с этим кодированием.

Меточный кодировщик может обрабатывать только один столбец за раз:

```
>>> from sklearn import preprocessing
>>> lab = preprocessing.LabelEncoder()
>>> lab.fit_transform(X_cat)
array([0,1])
```

Если у вас есть кодированные значения, с помощью метода `.inverse_transform` можете их декодировать:

```
>>> lab.inverse_transform([1, 1, 0])
array(['Guitar', 'Guitar', 'Bass'], dtype=object)
```

Для меточного кодирования можно использовать и библиотеку pandas. Сначала вы преобразуете тип столбца в категориальный, а затем извлекаете из него числовой код.

Этот код создаст новую серию числовых данных из серии pandas. Чтобы упорядочить категории, мы используем метод `.as_ordered`:

```
>>> X_cat.name.astype(
...     "category"
... ).cat.as_ordered().cat.codes + 1
0    1
1    2
dtype: int8
```

Частотное кодирование

Другим вариантом обработки больших категориальных данных является *частотное кодирование* (frequency encoding). Оно подразумевает замену названия категории номером, который был в учебных данных. Для этого мы будем использовать библиотеку pandas. Вначале используем метод pandas `.value_counts` для создания отображения (серии pandas, которые отображают строки на номера). Имея отображение, мы можем использовать метод `.map` для кодирования:

```
>>> mapping = X_cat.name.value_counts()
>>> X_cat.name.map(mapping)
0    1
1    1
Name: name, dtype: int64
```

Убедитесь, что вы сохранили учебное отображение, чтобы можно было кодировать будущие данные с теми же данными.

Извлечение категорий из строк

Один из способов повысить точность модели Titanic — извлечь заголовки из названий. Быстрый способ найти наиболее распространенные тройки — использовать класс `Counter`:

```
>>> from collections import Counter
>>> c = Counter()
>>> def triples(val):
...   for i in range(len(val)):
...       c[val[i : i + 3]] += 1
>>> df.name.apply(triples)
>>> c.most_common(10)
[(', M', 1282),
 (' Mr', 954),
 ('r. ', 830),
 ('Mr.', 757),
 ('s. ', 460),
 ('n, ', 320),
 (' Mi', 283),
 ('iss', 261),
 ('ss.', 261),
 ('Mis', 260)]
```

Мы видим, что "Mr." и "Miss." встречаются очень часто.

Другой вариант — использовать регулярное выражение для извлечения заглавной буквы, за которой следуют строчные буквы и точка:

```
>>> df.name.str.extract(
...     "([A-Za-z]+)\.", expand=False
... ).head()
0      Miss
1    Master
2      Miss
3        Mr
4       Mrs
Name: name, dtype: object
```

Чтобы увидеть их частоту, мы можем использовать метод `.value_counts`:

```
>>> df.name.str.extract(
... "([A-Za-z]+)\.", expand=False
... ).value_counts()
Mr          757
Miss        260
Mrs         197
Master       61
Dr            8
Rev           8
Col           4
Mlle          2
Ms            2
Major         2
Dona          1
Don           1
Lady          1
Countess      1
Capt          1
Sir           1
Mme           1
Jonkheer      1
Name: name, dtype: int64
```

НА ЗАМЕТКУ

Полное описание регулярных выражений выходит за рамки этой книги. Это выражение получает группу с одним или несколькими буквенными символами. За этой группой следует точка.

Используя эти манипуляции и библиотеку pandas, вы можете создавать фиктивные переменные или объединять столбцы с низким количеством данных в другие категории (или отбрасывать их).

Другие категориальные кодирования

Библиотека categoryor_encoding представляет собой набор преобразователей Scikit-learn, используемых для преобразования категориальных данных в числовые. Приятной особенностью этой библиотеки является то, что она поддерживает вывод DataFrame для pandas (в отличие от Scikit-learn, которая преобразует их в массивы numpy).

Одним из алгоритмов, реализованных в библиотеке, является хеш-кодер. Он полезен, если вы не знаете заранее, сколько у вас категорий, или используете для представления текста набор слов. Он будет хешировать категориальные столбцы в `n_components`. Если вы используете дистанционное обучение (модели, которые можно обновлять), это может быть очень полезно:

```
>>> import category_encoders as ce
>>> he = ce.HashingEncoder(verbose=1)
>>> he.fit_transform(X_cat)
   col_0 col_1 col_2 col_3 col_4 col_5 col_6 col_7
0      0     0     0     1     0     1     0     0
1      0     2     0     0     0     0     0     0
```

Порядковый кодировщик может преобразовывать категориальные столбцы, имеющие порядок, в один столбец чисел. Здесь мы преобразуем размер столбца в порядковые числа. Если в словаре сопоставления отсутствует значение, используется стандартное `-1`:

```
>>> size_df = pd.DataFrame(
...     {
...         "name": ["Fred", "John", "Matt"],
...         "size": ["small", "med", "xxl"],
...     }
... )
>>> ore = ce.OrdinalEncoder(
...     mapping=[
...         {
...             "col": "size",
...             "mapping": {
```

```
...                 "small": 1,
...                 "med": 2,
...                 "lg": 3,
...             },
...         }
...     ]
... )
>>> ore.fit_transform(size_df)
   name  size
0  Fred   1.0
1  John   2.0
2  Matt  -1.0
```

Эта ссылка (https://oreil.ly/JUtYh) объясняет многие алгоритмы библиотеки `categoryor_encoding`.

Если у вас есть данные с большим количеством элементов (большим количеством уникальных значений), рассмотрите возможность использования одного из байесовских кодеров, которые выводят по одному столбцу на каждый категориальный столбец. Это `TargetEncoder`, `LeaveOneOutEncoder`, `WOEEncoder`, `JamesSteinEncoder` и `MEstimateEncoder`.

Например, чтобы преобразовать столбец `survival` набора Titanic в комбинацию апостериорной вероятности цели и априорной вероятности с учетом информации заголовка (категориального), используйте следующий код:

```
>>> def get_title(df):
...     return df.name.str.extract(
...         "([A-Za-z]+)\.", expand=False
...     )
>>> te = ce.TargetEncoder(cols="Title")
>>> te.fit_transform(
...     df.assign(Title=get_title), df.survived
... )["Title"].head()
0    0.676923
1    0.508197
2    0.676923
3    0.162483
4    0.786802
Name: Title, dtype: float64
```

Конструирование признаков данных

Библиотека fastai имеет функцию `add_datepart`, которая будет генерировать столбцы атрибутов даты на основе столбца `datetime`. Это полезно, поскольку большинство алгоритмов машинного обучения не смогут вывести этот тип сигнала из числового представления даты:

```
>>> from fastai.tabular.transform import (
...     add_datepart,
... )
>>> dates = pd.DataFrame(
...     {
...         "A": pd.to_datetime(
...             ["9/17/2001", "Jan 1, 2002"]
...         )
...     }
... )

>>> add_datepart(dates, "A")
>>> dates.T
                            0          1
AYear                    2001       2002
AMonth                      9          1
AWeek                      38          1
ADay                       17          1
ADayofweek                  0          1
ADayofyear                260          1
AIs_month_end           False      False
AIs_month_start         False       True
AIs_quarter_end         False      False
AIs_quarter_start       False       True
AIs_year_end            False      False
AIs_year_start          False       True
AElapsed           1000684800 1009843200
```

ВНИМАНИЕ

Функция `add_datepart` изменяет объект DataFrame, который может создать библиотека pandas, но обычно это не так!

Добавление признака col_na

В библиотеке fastai была функция для создания столбца, чтобы заполнить пропущенные значения (медианой) и указать, что значение отсутствовало. Пропуск значения может быть неким сигналом. Вот копия функции и пример ее использования:

```
>>> from pandas.api.types import is_numeric_dtype
>>> def fix_missing(df, col, name, na_dict):
...     if is_numeric_dtype(col):
...         if pd.isnull(col).sum() or (
...             name in na_dict
...         ):
...             df[name + "_na"] = pd.isnull(col)
...             filler = (
...                 na_dict[name]
...                 if name in na_dict
...                 else col.median()
...             )
...             df[name] = col.fillna(filler)
...             na_dict[name] = filler
...     return na_dict
>>> data = pd.DataFrame({"A": [0, None, 5, 100]})
>>> fix_missing(data, data.A, "A", {})
{'A': 5.0}
>>> data
       A   A_na
0    0.0  False
1    5.0   True
2    5.0  False
3  100.0  False
```

Вот версия pandas:

```
>>> data = pd.DataFrame({"A": [0, None, 5, 100]})
>>> data["A_na"] = data.A.isnull()
>>> data["A"] = data.A.fillna(data.A.median())
```

Конструирование признаков вручную

Для создания новых признаков мы можем использовать библиотеку pandas. Для набора данных Titanic можно добавить совокупные данные о каюте (максимальный возраст на каюту, средний возраст на каюту и т.д.). Чтобы получить сводные данные для каждой каюты и объединить их, используйте для создания данных метод pandas `.groupby`. Затем выровняйте его по исходным данным, используя метод `.merge`:

```
>>> agg = (
...     df.groupby("cabin")
...     .agg("min,max,mean,sum".split(","))
...     .reset_index()
... )
>>> agg.columns = [
...     "_".join(c).strip("_")
...     for c in agg.columns.values
... ]
>>> agg_df = df.merge(agg, on="cabin")
```

Если вы хотите суммировать столбцы "good" и "bad" (хорошие и плохие), можете создать новый столбец, который будет суммой агрегированных столбцов (или другой математической операции). Это что-то вроде искусства и также требует понимания данных.

ГЛАВА 8

Выбор признаков

Мы используем *выбор признаков* (feature selection) для отбора тех признаков, которые полезны для модели. Нерелевантные признаки могут оказать негативное влияние на модель. Коррелированные признаки могут сделать коэффициенты регрессии (или важность признаков в древовидных моделях) нестабильными или трудными для интерпретации.

Проклятие размерности (curse of dimensionality) — это еще одна проблема, которую стоит рассмотреть. По мере увеличения количества размерностей ваших данных они становятся все более и более разреженными. Это может затруднить получение сигнала, если у вас нет больше данных. По мере добавления размерностей вычисления соседей, как правило, теряют свою полезность.

Кроме того, время обучения обычно зависит от количества столбцов (и иногда оно даже хуже линейного). Обеспечив краткость и точность своих столбцов, вы можете получить лучшую модель за меньшее время. Мы рассмотрим несколько примеров, используя набор данных agg_df из предыдущей главы. Помните, что это набор данных Titanic с некоторыми дополнительными столбцами с информацией о каюте. Поскольку этот набор данных агрегирует числовые значения для каждой каюты, он покажет много корреляций. К другим вариантам относятся PCA и поиск древовидного классификатора `.feature_importances_`.

Коллинеарные столбцы

Чтобы найти столбцы с коэффициентом корреляции 0,95 или выше, можно использовать определенный ранее признак `correlated_columns` или запустить следующий код:

```
>>> limit = 0.95
>>> corr = agg_df.corr()
>>> mask = np.triu(
...     np.ones(corr.shape), k=1
... ).astype(bool)
>>> corr_no_diag = corr.where(mask)
>>> coll = [
...     c
...     for c in corr_no_diag.columns
...     if any(abs(corr_no_diag[c]) > threshold)
... ]
>>> coll
['pclass_min', 'pclass_max', 'pclass_mean',
 'sibsp_mean', 'parch_mean', 'fare_mean',
 'body_max', 'body_mean', 'sex_male', 'embarked_S']
```

Показанный ранее визуализатор `Rank2` библиотеки Yellowbrick построит тепловую карту корреляций.

Пакет rfpimp способен визуализировать *мультиколлинеарность* (multicollinearity). Функция `plot_dependence_heatmap` обучает случайный лес по каждому числовому столбцу из других столбцов в обучающем наборе данных. Значение зависимости — это оценка R2 из *оценки OOB* (Out Of Bag — не вошедший в набор) для прогнозирования этого столбца (рис. 8.1).

Предлагаемый способ использования этого графика — найти значения, близкие к 1. Метка на оси X — это признак, который прогнозирует метку на оси Y. Если признак прогнозирует другую метку, вы можете удалить спрогнозированный признак (признак по оси Y). В нашем примере `fare` прогнозирует `pclass`, `sibsp`, `parch` и `embarked_Q`. Мы должны сохранить `fare` и, удалив другие признаки, получить аналогичные показатели:

```
>>> rfpimp.plot_dependence_heatmap(
...     rfpimp.feature_dependence_matrix(X_train),
...     value_fontsize=12,
...     label_fontsize=14,
...     figsize=(8, 8),sn
... )
>>> fig = plt.gcf()
>>> fig.savefig(
...     "images/mlpr_0801.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

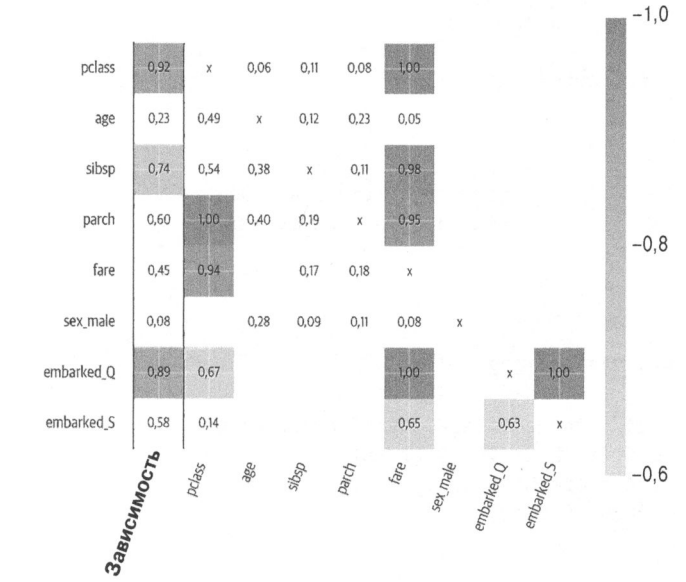


Рис. 8.1. *Тепловая карта зависимости. Признаки pclass, sibsp, parch и embarked_Q можно прогнозировать из fare, поэтому мы можем удалить их*

Вот код, показывающий, что мы получим похожую оценку, если уберем эти столбцы:

```
>>> cols_to_remove = [
...     "pclass",
```

```
...     "sibsp",
...     "parch",
...     "embarked_Q",
... ]
>>> rf3 = RandomForestClassifier(random_state=42)
>>> rf3.fit(
...     X_train[
...         [
...             c
...             for c in X_train.columns
...             if c not in cols_to_remove
...         ]
...     ],
...     y_train,
... )
>>> rf3.score(
...     X_test[
...         [
...             c
...             for c in X_train.columns
...             if c not in cols_to_remove
...         ]
...     ],
...     y_test,
... )
0.7684478371501272

>>> rf4 = RandomForestClassifier(random_state=42)
>>> rf4.fit(X_train, y_train)
>>> rf4.score(X_test, y_test)
0.7659033078880407
```

Регрессия лассо

Если вы используете регрессию лассо[1], можете установить альфа-параметр, который действует, как параметр регуляризации. Когда вы увеличиваете его значение, оно придает меньший вес менее важным признакам. Здесь мы используем

[1] Least Absolute Shrinkage and Selection Operator — LASSO. — *Примеч. ред.*

модель `LassoLarsCV` для итерации по различным значениям альфа и отслеживания коэффициентов признаков (рис. 8.2):

```
>>> from sklearn import linear_model
>>> model = linear_model.LassoLarsCV(
...     cv=10, max_n_alphas=10
... ).fit(X_train, y_train)
>>> fig, ax = plt.subplots(figsize=(12, 8))
>>> cm = iter(
...     plt.get_cmap("tab20")(
...         np.linspace(0, 1, X.shape[1])
...     )
... )
>>> for i in range(X.shape[1]):
...     c = next(cm)
...     ax.plot(
...         model.alphas_,
...         model.coef_path_.T[:, i],
...         c=c,
...         alpha=0.8,
...         label=X.columns[i],
...     )
>>> ax.axvline(
...     model.alpha_,
...     linestyle="-",
...     c="k",
...     label="alphaCV",
... )
>>> plt.ylabel("Regression Coefficients")
>>> ax.legend(X.columns, bbox_to_anchor=(1, 1))
>>> plt.xlabel("alpha")
>>> plt.title(
...     "Regression Coefficients Progression for Lasso
Paths"
... )
>>> fig.savefig(
...     "images/mlpr_0802.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

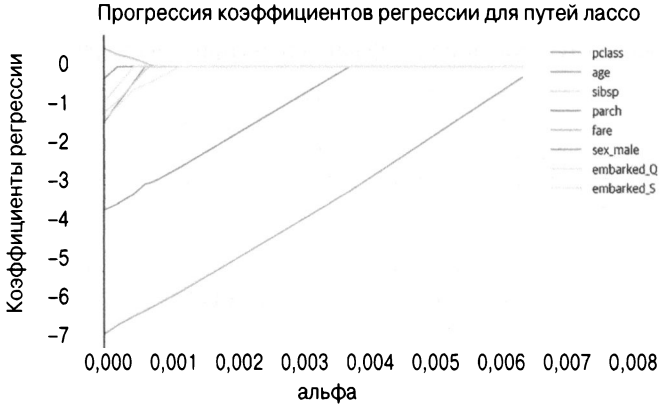


Рис. 8.2. *Коэффициенты признаков при варьировании альфа во время регрессии Лассо*

Удаление рекурсивных признаков

При удалении рекурсивных признаков удаляются самые слабые признаки, а затем модель подгоняется (рис. 8.3). Для этого модель передается Scikit-learn с атрибутом `.coef_` или `.feature_importances_`:

```
>>> from yellowbrick.features import RFECV
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> rfe = RFECV(
...     ensemble.RandomForestClassifier(
...         n_estimators=100
...     ),
...     cv=5,
... )
>>> rfe.fit(X, y)

>>> rfe.rfe_estimator_.ranking_
array([1, 1, 2, 3, 1, 1, 5, 4])

>>> rfe.rfe_estimator_.n_features_
4
>>> rfe.rfe_estimator_.support_
```

```
array([ True,  True, False, False,  True,
        True, False, False])

>>> rfe.poof()
>>> fig.savefig("images/mlpr_0803.png", dpi=300)
```

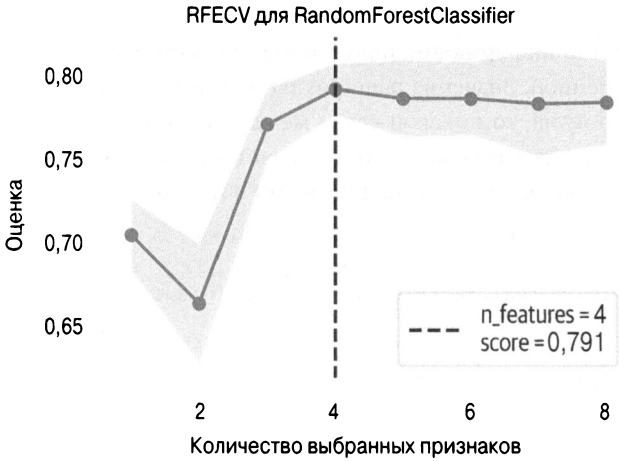


Рис. 8.3. *Удаление рекурсивных признаков*

Мы будем использовать удаление рекурсивных признаков, чтобы найти 10 наиболее важных признаков. (В этом агрегированном наборе данных мы обнаруживаем утечку в столбце survival!)

```
>>> from sklearn.feature_selection import RFE
>>> model = ensemble.RandomForestClassifier(
...     n_estimators=100
... )
>>> rfe = RFE(model, 4)
>>> rfe.fit(X, y)
>>> agg_X.columns[rfe.support_]
Index(['pclass', 'age', 'fare', 'sex_male'],
dtype='object')
```

Взаимная информация

Библиотека Scikit-learn предоставляет непараметрические тесты, использующие метод k-ближайшего соседа для определения *взаимной информации* (mutual information) между признаками и целью. Взаимная информация определяет количество информации, полученной при наблюдении за другой переменной. Значение равно нулю или больше. Если значение равно нулю, то никакой связи между ними нет (рис. 8.4). Это число не ограничено и представляет количество *битов* (bit), совместно используемых признаком и целью:

```
>>> from sklearn import feature_selection

>>> mic = feature_selection.mutual_info_classif(
...     X, y
... )
>>> fig, ax = plt.subplots(figsize=(10, 8))
>>> (
...     pd.DataFrame(
...         {"feature": X.columns, "vimp": mic}
...     )
...     .set_index("feature")
...     .plot.barh(ax=ax)
... )
>>> fig.savefig("images/mlpr_0804.png")
```

Анализ основных компонентов

Другой вариант выбора признака — запуск *анализа основных компонентов* (principal component analysis). Если у вас есть главные основные компоненты (main principal component), изучите признаки, которые вносят наибольший вклад. Это признаки, которые имеют наибольшую вариацию. Обратите внимание, что это алгоритм без учителя, который не учитывает y.

Более подробная информация по этой теме приведена далее в книге.

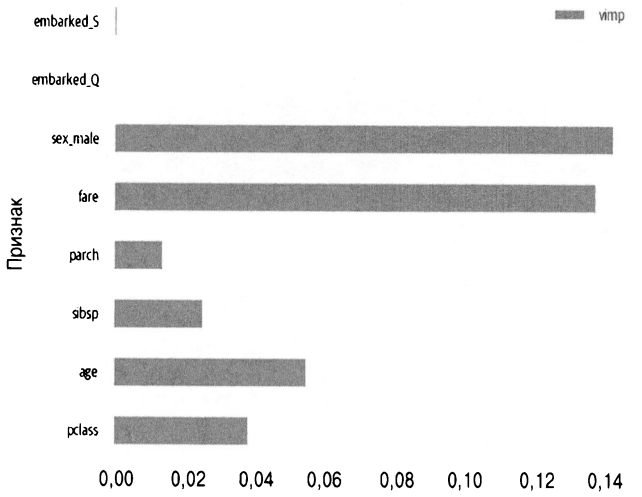


Рис. 8.4. *Диаграмма взаимной информации*

Важность признака

После обучения большинство древовидных моделей предоставляют доступ к атрибуту `.feature_importances_`. Более высокая важность обычно означает, что при удалении признака из модели возникает большая ошибка. Более подробная информация о древовидных моделях приведена в соответствующих главах.

ГЛАВА 9

Несбалансированные классы

Если вы классифицируете данные и классы не являются относительно сбалансированными по размеру, смещение в сторону более популярных классов может перейти в вашу модель. Например, если у вас есть 1 позитивный случай и 99 негативных, можете получить точность 99%, просто классифицируя все как негативное. Существуют различные варианты работы с *несбалансированными классами* (imbalanced class).

Использование другой метрики

Одним из вариантов является использование для калибровки моделей меры, отличной от точности (AUC — хороший выбор). *Точность* (precision) и *отзыв* (recall) — также отличные варианты, когда размеры целей различаются. Но есть и другие варианты для рассмотрения.

Алгоритмы и ансамбли на основе дерева

Древовидные модели могут работать лучше в зависимости от распределения миноритарного класса. Если они имеют кластерный характер, их легче классифицировать.

Ансамбли могут дополнительно помочь в выявлении миноритарных классов. *Бэггинг* (bagging) и *бустинг* (boosting) — это

опции, которые можно найти в древовидных моделях, таких как *случайные леса* (random forest) и *экстремальный градиентный бустинг* (Extreme Gradient Boosting — XGBoost).

Штрафующие модели

Многие модели классификации библиотеки Scikit-learn имеют параметр `class_weight`. Установка для этого параметра значения `'balanced'` приводит к попытке упорядочить миноритарные классы и стимулировать модель к их правильной классификации. В качестве альтернативы вы можете выполнить *сеточный поиск* (grid search) и указать параметры веса, передав класс сопоставления словаря для весов (придайте меньшим классам больший вес).

Библиотека *XGBoost* имеет параметр `max_delta_step`, которому можно присвоить значение в диапазоне от 1 до 10, чтобы сделать шаг обновления более консервативным. Она также имеет параметр `scale_pos_weight`, который устанавливает соотношение негативных и позитивных выборок (для двоичных классов). Кроме того, для классификации параметр `eval_metric` должен быть установлен равным `'auc'`, а не стандартному `'error'`.

Модель KNN имеет параметр `weights`, способный смещать соседей, которые находятся ближе. Если выборки миноритарного класса расположены близко одна к другой, установка для этого параметра значения `'distance'` может улучшить производительность.

Повышающая дискретизация миноритарного класса

Вы можете повысить дискретизацию миноритарного класса несколькими способами. Вот реализация Scikit-learn:

```
>>> from sklearn.utils import resample
>>> mask = df.survived == 1
>>> surv_df = df[mask]
>>> death_df = df[~mask]
>>> df_upsample = resample(
...     surv_df,
...     replace=True,
...     n_samples=len(death_df),
...     random_state=42,
... )
>>> df2 = pd.concat([death_df, df_upsample])

>>> df2.survived.value_counts()
1    809
0    809
Name: survived, dtype: int64
```

Также можно использовать библиотеку imbalanced-learn для случайной выборки с заменой:

```
>>> from imblearn.over_sampling import (
...     RandomOverSampler,
... )
>>> ros = RandomOverSampler(random_state=42)
>>> X_ros, y_ros = ros.fit_sample(X, y)
>>> pd.Series(y_ros).value_counts()
1    809
0    809
dtype: int64
```

Генерация данных миноритарного класса

Библиотека imbalanced-learn может также генерировать новые выборки миноритарных классов с использованием таких алгоритмов выборки, как *стратегия искусственного увеличения экземпляров миноритарного класса* (Synthetic Minority Oversampling Technique — SMOTE) и *адаптивного синтетического сэмплинга* (Adaptive Synthetic Sampling — ADASYN). Алгоритм SMOTE работает, выбирая одного из k-ближайших соседей, соединяя его линией и выбирая точку вдоль этой линии.

Алгоритм ADASYN похож на SMOTE, но генерирует больше выборок из тех, на которых сложнее обучаться. Соответствующие классы библиотеки imbanced-learn — это `over_sampling.SMOTE` и `over_sampling.ADASYN`.

Понижающая дискретизация мажоритарного класса

Другой способ сбалансировать классы — понижающая дискретизация мажоритарного класса. Вот реализации Scikit-learn:

```
>>> from sklearn.utils import resample
>>> mask = df.survived == 1
>>> surv_df = df[mask]
>>> death_df = df[~mask]
>>> df_downsample = resample(
...     death_df,
...     replace=False,
...     n_samples=len(surv_df),
...     random_state=42,
... )
>>> df3 = pd.concat([surv_df, df_downsample])

>>> df3.survived.value_counts()
1    500
0    500
Name: survived, dtype: int64
```

СОВЕТ

Не используйте замену при понижающей дискретизации.

Библиотека imbalanced-learn реализует также различные алгоритмы понижающей дискретизации:

`ClusterCentroids`

Этот класс использует метод k-средних для синтеза данных с центроидами.

`RandomUnderSampler`

Этот класс делает выборки случайным образом.

`NearMiss`

Этот класс использует для понижения дискретизации метод ближайших соседей.

`TomekLink`

Этот класс сокращает выборки, удаляя те из них, которые расположены близко одна к другой.

`EditedNearestNeighbours`

Этот класс удаляет выборки, у которых есть соседи, не принадлежащие к большинству или все принадлежащие к одному и тому же классу.

`RepeatedNearestNeighbours`

Этот класс регулярно вызывает `EditedNearestNeighbours`.

`AllKNN`

Этот класс похож, но увеличивает количество ближайших соседей во время итераций понижающей дискретизации.

`CondensedNearestNeighbour`

Этот класс извлекает одну выборку класса для понижающей дискретизации, затем перебирает другие выборки класса и, если KNN не классифицирует ошибочно, добавляет эту выборку.

`OneSidedSelection`

Этот класс убирает шумные выборки.

`NeighbourhoodCleaningRule`

Этот класс использует результаты `EditedNearestNeighbours` и применяет к нему KNN.

`InstanceHardnessThreshold`

Этот класс обучает модель, а затем удаляет выборки с низкой вероятностью.

Все эти классы поддерживают метод `.fit_sample`.

Повышающая дискретизация, затем понижающая

В библиотеке imbalanced-learn реализованы `SMOTEENN` и `SMOTE Tomek`, которые повышают дискретизацию выборки, а затем понижают для очистки данных.

ГЛАВА 10

Классификация

Классификация (classification) — это механизм *обучения с учителем* (supervised learning), осуществляющий маркировку выборки на основе признаков. Обучение с учителем означает, что у нас есть метки для классификации или числа для регрессии, которые алгоритм должен выучить.

В этой главе мы рассмотрим различные модели классификации. Библиотека Scikit-learn реализует множество общих и полезных моделей. Мы также увидим некоторые из таковых, которые не находятся в библиотеке Scikit-learn, но реализуют ее интерфейс. Поскольку они используют один и тот же интерфейс, легко опробовать разные семейства моделей и посмотреть, насколько хорошо они работают.

В Scikit-learn мы создаем экземпляр модели и вызываем для него метод `.fit` с учебными данными и учебными метками. Теперь, с обученной моделью, мы можем вызвать метод `.predict` (либо методы `.predict_proba` и `.predict_log_proba`). Для оценки модели мы используем метод `.score` с тестовыми данными и тестовыми метками.

Более сложной задачей обычно является упорядочение данных в такой форме, которая будет работать с библиотекой Scikit-learn. Данные (`X`) должны быть массивом (m на n) (или объектом pandas DataFrame) с m строками выборочных данных, каждая из которых имеет n объектов (столбцов). Метка (`y`) представляет собой вектор (или серию pandas) размером m со значением (классом) для каждой выборки.

Метод `.score` возвращает среднюю точность, которая сама по себе может оказаться недостаточной для оценки классификатора. Мы увидим и другие показатели оценки.

Мы рассмотрим многие модели и обсудим их эффективность, методы предварительной обработки, которые им требуются, а также как предотвратить переобучение и поддерживает ли модель интуитивную интерпретацию результатов.

Модели типа Scikit-learn реализуют такие основные методы:

```
fit(X, y[, sample_weight])
```

(обучает модель)

```
predict(X)
```

(прогнозирует классы)

```
predict_log_proba(X)
```

(прогнозирует логарифмическую вероятность)

```
score(X, y[, sample_weight])
```

(выводит точность)

Логистическая регрессия

Логистическая регрессия оценивает вероятности с помощью логистической функции. (Осторожно: несмотря на то что в названии есть слово "регрессия", оно используется для классификации.) Для большинства наук это была стандартная классификационная модель.

Ниже приведены некоторые характеристики, которые мы рассмотрим для каждой модели.

Эффективность выполнения (runtime efficiency)

Можно использовать `n_jobs`, если не используется решатель `'liblinear'`.

Предварительная обработка данных (preprocess data)

Если для `solver` установлено значение `'sag'` или `'saga'`, выполните стандартизацию для сходимости. Может обрабатывать разреженный ввод.

Предотвращение переобучения (prevent overfitting)

Параметр `C` контролирует регуляризацию. (Чем ниже `C`, тем больше регуляризация; чем выше `C`, тем меньше регуляризация.) Для `penalty` можно установить значение `'l1'` или `'l2'` (стандартное).

Интерпретация результатов (interpret results)

Атрибут `.coef_` обученной модели демонстрирует коэффициенты функции принятия решения. Изменение x на одну единицу изменяет логарифм вероятности успешного исхода на коэффициент. Атрибут `.intercept_` — это обратный логарифм вероятности успешного исхода при стандартных условиях.

Вот пример использования этой модели:

```
>>> from sklearn.linear_model import (
...     LogisticRegression,
... )
>>> lr = LogisticRegression(random_state=42)
>>> lr.fit(X_train, y_train)
LogisticRegression(C=1.0, class_weight=None,
    dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100,
    multi_class='ovr', n_jobs=1, penalty='l2',
    random_state=42, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
>>> lr.score(X_test, y_test)
0.8040712468193384

>>> lr.predict(X.iloc[[0]])
array([1])
>>> lr.predict_proba(X.iloc[[0]])
array([[0.08698937, 0.91301063]])
>>> lr.predict_log_proba(X.iloc[[0]])
array([[-2.4419694 , -0.09100775]])
```

```
>>> lr.decision_function(X.iloc[[0]])
array([2.35096164])
```

Параметры экземпляра

```
penalty='l2'
```

Норма штрафа — `'l1'` или `'l2'`.

```
dual=False
```

Использует двойную формулировку (только `'l2'` или `'liblinear'`).

```
C=1.0
```

Положительное число с плавающей запятой. Сила обратной регуляризации. Чем оно меньше, тем сильнее регуляризация.

```
fit_intercept=True
```

Добавляет смещение к функции принятия решения.

```
intercept_scaling=1
```

Если `fit_intercept` и `'liblinear'`, масштабировать отсечение.

```
max_iter=100
```

Максимальное количество итераций.

```
multi_class='ovr'
```

Использовать один против всех для каждого класса или для `'multinomial'`, обучает один класс.

```
class_weight=None
```

Словарь или `'balanced'`.

```
solver='liblinear'
```

`'liblinear'` хорош для небольших данных. `'newton-cg'`, `'sag'`, `'saga'` и `'lbfgs'` предназначены для многоклассовых данных. `'liblinear'` и `'saga'` работают только со штрафом `'l1'`. Остальные работают с `'l2'`.

`tol=0.0001`

Остановка толерантности.

`verbose=0`

Многословность (если не нулевое целое значение).

`warm_start=False`

Если `True`, помнить предыдущую подгонку.

`njobs=1`

Количество используемых процессоров; `-1` — все процессоры. Работает только с `multi_class='over'` и если `solver` не `'liblinear'`.

Атрибуты после подгонки

`coef_`

Коэффициенты функции принятия решения.

`intercept_`

Отсечение функции принятия решения.

`n_iter_`

Количество итераций.

Отсечение (intercept) — это логарифм стандартных условий. Мы можем преобразовать его обратно в проценты точности (пропорции):

```
>>> lr.intercept_
array([-0.62386001])
```

Использовав обратную логит-функцию, мы увидим, что базовая линия для выживания составляет 34%:

```
>>> def inv_logit(p):
...     return np.exp(p) / (1 + np.exp(p))

>>> inv_logit(lr.intercept_)
array([0.34890406])
```

Мы можем проверить коэффициенты. Обратный логит коэффициентов дает долю позитивных случаев. В данном случае, если тариф выше, у нас больше шансов выжить. Если пол мужской, у нас меньше шансов выжить.

```
>>> cols = X.columns
>>> for col, val in sorted(
...     zip(cols, lr.coef_[0]),
...     key=lambda x: x[1],
...     reverse=True,
... ):
...     print(
...         f"{col:10}{val:10.3f} {inv_logit(val):10.3f}"
...     )
fare         0.104    0.526
parch       -0.062    0.485
sibsp       -0.274    0.432
age         -0.296    0.427
embarked_Q  -0.504    0.377
embarked_S  -0.507    0.376
pclass      -0.740    0.323
sex_male    -2.400    0.083
```

Yellowbrick также может визуализировать коэффициенты. Этот визуализатор имеет параметр `relative=True`, который устанавливает наибольшее значение равным 100 (или –100), а остальные являются процентами этого значения (рис. 10.1):

```
>>> from yellowbrick.features.importances import (
...     FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(lr)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1001.png", dpi=300)
```

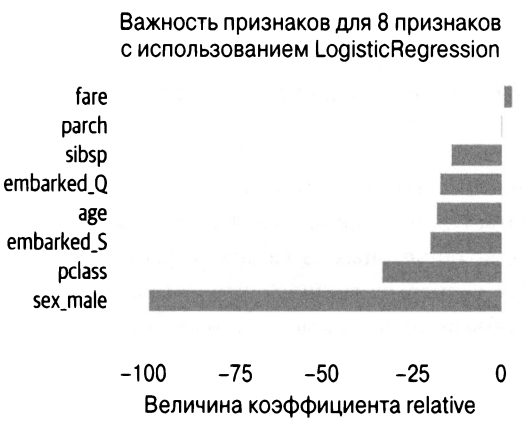


Рис. 10.1. *Важность признака (относительно наибольшего абсолютного коэффициента регрессии)*

Наивный байесовский классификатор

Наивный байесовский классификатор (Naive Bayes) — это вероятностный классификатор, предполагающий независимость между признаками данных. Он популярен в приложениях классификации текста, таких как предотвращение спама. Одним из преимуществ этой модели является способность к обучению на небольшом количестве выборок, поскольку она предполагает независимость признаков. (Недостатком является неспособность фиксировать взаимодействия между признаками.) Эта простая модель также может работать с данными, имеющими много признаков. Как таковая она служит хорошей базовой моделью.

В библиотеке Scikit-learn есть три класса: `GaussianNB`, `MultinomialNB` и `BernoulliNB`. Первый предполагает гауссово распределение (непрерывные признаки с нормальным распределением), второй — для дискретных подсчетов вхождений, а третий — для дискретных булевых признаков.

Эта модель имеет следующие свойства.

Эффективность выполнения

Обучение O(Nd), где N — количество примеров обучения, а d — размерность. Тестирование O(cd), где c — это количество классов.

Предварительная обработка данных

Предполагается, что данные являются независимыми. После удаления коллинеарных столбцов эффективность должна возрасти. Для непрерывных числовых данных, возможно, будет полезно объединить данные. Гауссово распределение предполагает нормальное распределение данных, и вам может потребоваться преобразовать данные, чтобы получить нормальное распределение.

Предотвращение переобучения

Показывает высокое смещение и низкую дисперсию (ансамбли не уменьшат дисперсию).

Интерпретация результатов

Процент — это вероятность того, что выборка принадлежит классу на основе априорных значений.

Вот пример использования этой модели:

```
>>> from sklearn.naive_bayes import GaussianNB
>>> nb = GaussianNB()
>>> nb.fit(X_train, y_train)
GaussianNB(priors=None, var_smoothing=1e-09)
>>> nb.score(X_test, y_test)
0.7837150127226463

>>> nb.predict(X.iloc[[0]])
array([1])
>>> nb.predict_proba(X.iloc[[0]])
array([[2.17472227e-08, 9.99999978e-01]])
>>> nb.predict_log_proba(X.iloc[[0]])
array([[-1.76437798e+01, -2.17472227e-08]])
```

Параметры экземпляра

```
priors=None
```

Априорные вероятности классов.

`var_smoothing=1e-9`

Добавляет дисперсию для стабилизации расчетов.

Атрибуты после подгонки

`class_prior_`

Вероятности классов.

`class_count_`

Подсчет классов.

`theta_`

Среднее значение каждого столбца на класс.

`sigma_`

Дисперсия каждого столбца на класс.

`epsilon_`

Аддитивное значение для каждой дисперсии.

СОВЕТ

Эти модели подвержены *проблеме нулевой вероятности* (zero probability problem). Если вы попытаетесь классифицировать новую выборку, в которой нет обучающих данных, она будет иметь нулевую вероятность. Одним из решений является использование *сглаживания по Лапласу*. Библиотека Scikit-learn контролирует его с помощью параметра `alpha`, который стандартно равен `1` и включает сглаживание в моделях `MultinomialNB` и `BernoulliNB`.

Метод опорных векторов

Метод опорных векторов (Support Vector Machine — SVM) — это алгоритм, который пытается провести линию (или плоскость, или гиперплоскость) между различными классами,

чтобы максимизировать расстояние от линии до точек классов. Таким образом, он пытается найти надежное разделение между классами. *Опорные векторы* (support vector) — это точки на краях разделяющей гиперплоскости.

НА ЗАМЕТКУ

В библиотеке Scikit-learn есть несколько разных реализаций SVM. `SVC` упаковывает библиотеку `libsvm`, в то время как `LinearSVC` упаковывает библиотеку `liblinear`.

Существует также `linear_model.SGDClassifier`, реализующий SVM при использовании стандартного параметра `loss`. В этой главе будет описана первая реализация.

SVM обычно работает хорошо и может поддерживать линейные или нелинейные пространства, используя трюк ядра. *Трюк ядра* (kernel trick) заключается в том, что мы можем создать границу принятия решения в новом измерении, минимизируя формулу, которую вычислить легче, чем фактически сопоставлять точки с новым измерением. Стандартным ядром является *радиальная базисная функция* (Radial Basis Function) (`'rbf'`), которая контролируется параметром `gamma` и может отображать входное пространство в многомерное пространство.

SVM имеют следующие свойства.

Эффективность выполнения

Реализация Scikit-learn — $O(n^4)$, поэтому ее сложно масштабировать до больших размеров. Использование линейного ядра или модели `Line arSVC` может улучшить производительность времени выполнения, возможно, за счет точности. Увеличение параметра `cache_size` может привести к уменьшению значения до $O(n^3)$.

Предварительная обработка данных

Алгоритм не является масштабно-инвариантным. Настоятельно рекомендуется стандартизация данных.

Предотвращение переобучения

Параметр `C` (параметр штрафов) контролирует регуляризацию. Меньшее значение допускает меньший зазор в гиперплоскости. Большее значение `gamma` создаст тенденцию к переобучению на учебных данных. Для поддержки регуляризации модель `LinearSVC` поддерживает параметры `loss` и `penalty`.

Интерпретация результатов

Проверьте `.support_vectors_`, хотя это трудно объяснить. С линейными ядрами вы можете проверить `.coef_`.

Вот пример использования SVM реализации Scikit-learn:

```
>>> from sklearn.svm import SVC
>>> svc = SVC(random_state=42, probability=True)
>>> svc.fit(X_train, y_train)
SVC(C=1.0, cache_size=200, class_weight=None,
  coef0=0.0, decision_function_shape='ovr',
  degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=True, random_state=42,
  shrinking=True, tol=0.001, verbose=False)
>>> svc.score(X_test, y_test)
0.8015267175572519

>>> svc.predict(X.iloc[[0]])
array([1])
>>> svc.predict_proba(X.iloc[[0]])
array([[0.15344656, 0.84655344]])
>>> svc.predict_log_proba(X.iloc[[0]])
array([[-1.87440289, -0.16658195]])
```

Чтобы получить вероятность, используйте `probability=True`, что замедлит подгонку модели.

Это похоже на перцептрон, но находит максимальные края. Если данные не разделяются линейно, это минимизирует ошибку. В качестве альтернативы можно использовать другое ядро.

Параметры экземпляра

`C=1.0`

Параметр штрафа. Чем меньше значение, тем плотнее граница принятия решения (больше переобучение).

`cache_size=200`

Размер кеша (Мбайт). Увеличение этого показателя может сократить время обучения на больших наборах данных.

`class_weight=None`

Словарь или `'balanced'`. Используйте словарь, чтобы установить `C` для каждого класса.

`coef0=0.0`

Независимый член для поли- и сигмовидных ядер.

`decision_function_shape='ovr'`

Используйте один против всех (`'ovr'`) или один против одного.

`degree=3`

Степень для полиномиального ядра.

`gamma='auto'`

Коэффициент ядра. Может быть числом, `'scale'` (стандартно — 0,22, 1 / (`num features * X.std()`)) или `'auto'` (стандартно — 1 / `num features`). Более низкое значение приводит к переобучению на учебных данных.

`kernel='rbf'`

Тип ядра: `'linear'`, `'poly'`, `'rbf'` (стандартно), `'sigmoid'`, `'precomputed'` или функция.

`max_iter=-1`

Максимальное количество итераций для решателя. `-1` — без ограничений.

`probability=False`

Включить оценку вероятности. Замедляет обучение.

`random_state=None`

Случайное начальное число.

`shrinking=True`

Использовать сокращающуюся эвристику.

`tol=0.001`

Остановка толерантности.

`verbose=False`

Многословность.

Атрибуты после подгонки

`support_`

Индексы опорных векторов.

`support_vectors_`

Опорные векторы.

`n_support_vectors_`

Количество опорных векторов для каждого класса.

`coef_`

Коэффициенты (линейного) ядра.

K-ближайшие соседи

Алгоритм *k-ближайших соседей* (k-Nearest Neighbor — KNN) классифицирует на основе дистанции до некоторого числа (k) обучающих выборок. Это семейство алгоритмов называется *обучением на примерах* (instance-based learning), поскольку параметров для изучения нет. Эта модель предполагает, что дистанция достаточна для вывода; в противном случае он не делает никаких предположений о базовых данных или их распределении.

Сложная часть заключается в выборе подходящего значения k. Кроме того, проклятие размерности может препятствовать метрикам расстояния, поскольку при больших размерностях разница между ближайшим и дальним соседями невелика.

Модели ближайших соседей имеют следующие свойства.

Эффективность выполнения

Обучение O(1), но необходимо хранить данные. Тестирование O(Nd), где N — количество обучающих примеров, а d — размерность.

Предварительная обработка данных

Да, расчеты на основе расстояний лучше выполняются при стандартизации.

Предотвращение переобучения

Увеличить `n_neighbors`. Изменить `p` для метрики L1 или L2.

Интерпретация результатов

Интерпретировать k-ближайших соседей к выборке (используя метод `.kneighbors`). Эти соседи объясняют ваш результат (если вы сможете их объяснить).

Вот пример использования модели:

```
>>> from sklearn.neighbors import (
...     KNeighborsClassifier,
... )
>>> knc = KNeighborsClassifier()
>>> knc.fit(X_train, y_train)
KNeighborsClassifier(algorithm='auto',
  leaf_size=30, metric='minkowski',
  metric_params=None, n_jobs=1, n_neighbors=5,
  p=2, weights='uniform')
>>> knc.score(X_test, y_test)
0.7837150127226463

>>> knc.predict(X.iloc[[0]])
array([1])
```

```
>>> knc.predict_proba(X.iloc[[0]])
array([[0., 1.]])
```

Атрибуты

```
algorithm='auto'
```

Может быть `'brute'`, `'ball_tree'` или `'kd_tree'`.

```
leaf_size=30
```

Используется для древовидных алгоритмов.

```
metric='minkowski'
```

Метрика расстояния.

```
metric_params=None
```

Дополнительный словарь параметров для пользовательской метрической функции.

```
n_jobs=1
```

Количество процессоров.

```
n_neighbors=5
```

Количество соседей.

```
p=2
```

Степенной параметр Минковского: 1 — манхэттен (L1), 2 — евклидово (L2).

```
weights='uniform'
```

Может быть `'distance'`, в этом случае более близкие точки имеют большее влияние.

Метрики расстояния включают: `'euclidean'`, `'manhattan'`, `'chebyshev'`, `'minkowski'`, `'wminkowski'`, `'seuclidean'`, `'mahalanobis'`, `'haversine'`, `'hamming'`, `'canberra'`, `'braycurtis'`, `'jaccard'`, `'matching'`, `'dice'`, `'rogerstanimoto'`, `'russellrao'`, `'sokalmichener'`, `'sokalsneath'` или определяемая пользователем.

НА ЗАМЕТКУ

Если k — четное число и соседи разделены, результат зависит от порядка обучающих данных.

Дерево решений

Дерево решений похоже на обращение к врачу, который задает ряд вопросов, чтобы определить причину симптомов. Мы можем использовать процесс создания дерева решений и задать ряд вопросов для прогнозирования целевого класса. К преимуществам этой модели относятся поддержка нечисловых данных (в некоторых реализациях), несложная подготовка данных (нет необходимости в масштабировании), поддержка работы с нелинейными отношениями, раскрытие важности признаков и легкость объяснения.

Стандартным алгоритмом, используемым для создания, является *дерево классификации и регрессии* (Classification And Regression Tree — CART). Для построения решений он использует *коэффициент Джини* (Gini impurity) или *показатель индекса* (index measure). Это осуществляется за счет перебора признаков и поиска такого значения, которое дает наименьшую вероятность ошибочной классификации.

СОВЕТ

Стандартный алгоритм создает полностью выросшее дерево (читай "переобучение"). Чтобы контролировать это, используйте такой механизм, как `max_depth`, и перекрестную проверку.

Деревья решений имеют следующие свойства.

Эффективность выполнения

Для создания переберите все m признаков и отсортируйте все n выборок, O($mn \log n$). Для прогнозирования вы проходите по дереву O(высота).

Предварительная обработка данных

Масштабирование не обязательно. Нужно избавиться от пропущенных значений и преобразовать их в числовые.

Предотвращение переобучения

Установите для `max_depth` меньшее значение, увеличьте `min_impurity_decrease`.

Интерпретация результатов

Можно пройти по дереву выбора. Поскольку существуют этапы, дерево плохо справляется с линейными отношениями (небольшое изменение в числах, и процесс может пойти другим путем). Дерево также сильно зависит от обучающих данных. Небольшое изменение может изменить все дерево.

Вот пример использования библиотеки Scikit-learn:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier(
...     random_state=42, max_depth=3
... )
>>> dt.fit(X_train, y_train)
DecisionTreeClassifier(class_weight=None,
  criterion='gini', max_depth=None,
  max_features=None, max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,
  min_samples_leaf=1, min_samples_split=2,
  min_weight_fraction_leaf=0.0, presort=False,
  random_state=42, splitter='best')

>>> dt.score(X_test, y_test)
0.8142493638676844

>>> dt.predict(X.iloc[[0]])
array([1])
```

```
>>> dt.predict_proba(X.iloc[[0]])
array([[0.02040816, 0.97959184]])
>>> dt.predict_log_proba(X.iloc[[0]])
array([[-3.8918203 , -0.02061929]])
```

Параметры экземпляра

```
class_weight=None
```

Веса для класса в словаре. `'balanced'` установит значения в обратную пропорцию частот класса. Стандартно это значение 1 для каждого класса. Для множества классов нужен список словарей *один против всех* (one-versus-rest — OVR) для каждого класса.

```
criterion='gini'
```

Функция разделения, `'gini'` или `'entropy'`.

```
max_depth=None
```

Глубина дерева. Стандартно дерево будет строиться до тех пор, пока содержимое листьев меньше `min_samples_split`.

```
max_features=None
```

Количество признаков для проверки на разделение. Стандартно — все.

```
max_leaf_nodes=None
```

Предельное количество листьев. Стандартно — не ограничено.

```
min_impurity_decrease=0.0
```

Разделять узел, если разделение уменьшит инородность > = значение (impurity >= value).

```
min_impurity_split=None
```

Нерекомендуемый.

```
min_samples_leaf=1
```

Минимальное количество выборок в каждом листе.

`min_samples_split=2`

Минимальное количество выборок, необходимых для разделения узла.

`min_weight_fraction_leaf=0.0`

Минимальная сумма весов, необходимая для конечных узлов.

`presort=False`

Может ускорить обучение при небольшом наборе данных или ограниченной глубине, если установлено значение `True`.

`random_state=None`

Случайное начальное число.

`splitter='best'`

Используйте `'random'` или `'best'`.

Атрибуты после подгонки

`classes_`

Метки класса.

`feature_importances_`

Массив важности Джини.

`n_classes_`

Количество классов.

`n_features_`

Количество признаков.

`tree_`

Базовый объект дерева.

Дерево демонстрирует следующий код (рис. 10.2):

```
>>> import pydotplus
>>> from io import StringIO
>>> from sklearn.tree import export_graphviz
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dt,
...     out_file=dot_data,
...     feature_names=X.columns,
```

```
...     class_names=["Died", "Survived"],
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1002.png")
```

Для Jupyter используйте

```
from IPython.display import Image
Image(g.create_png())
```

Пакет dtreeviz может помочь понять, как работает дерево решений. Он создает дерево с помеченными гистограммами, что дает ценную информацию (рис. 10.3). Вот пример: в Jupyter мы можем просто отобразить объект `viz` непосредственно. Если мы работаем из сценария, можно вызвать метод `.save` для создания файла PDF, SVG или PNG:

```
>>> viz = dtreeviz.trees.dtreeviz(
...       dt,
...       X,
...       y,
...       target_name="survived",
...       feature_names=X.columns,
...       class_names=["died", "survived"],
... )
>>> viz
```

Важность признака, демонстрируемая важностью Джини (уменьшение ошибки с использованием этого признака):

```
>>> for col, val in sorted(
...     zip(X.columns, dt.feature_importances_),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
sex_male       0.607
pclass         0.248
sibsp          0.052
fare           0.050
age            0.043
```

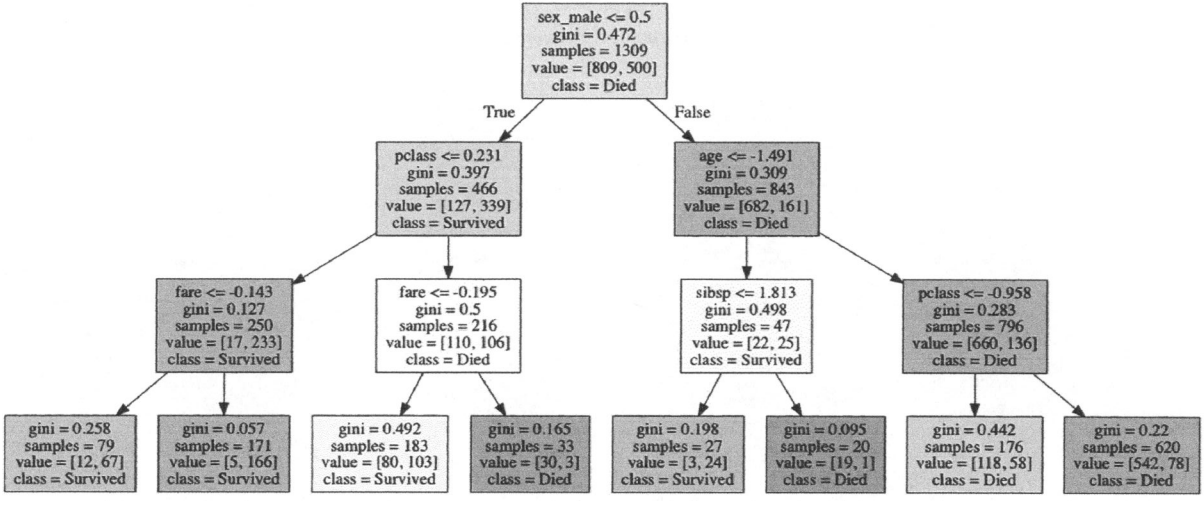


Рис. 10.2. *Дерево решений*

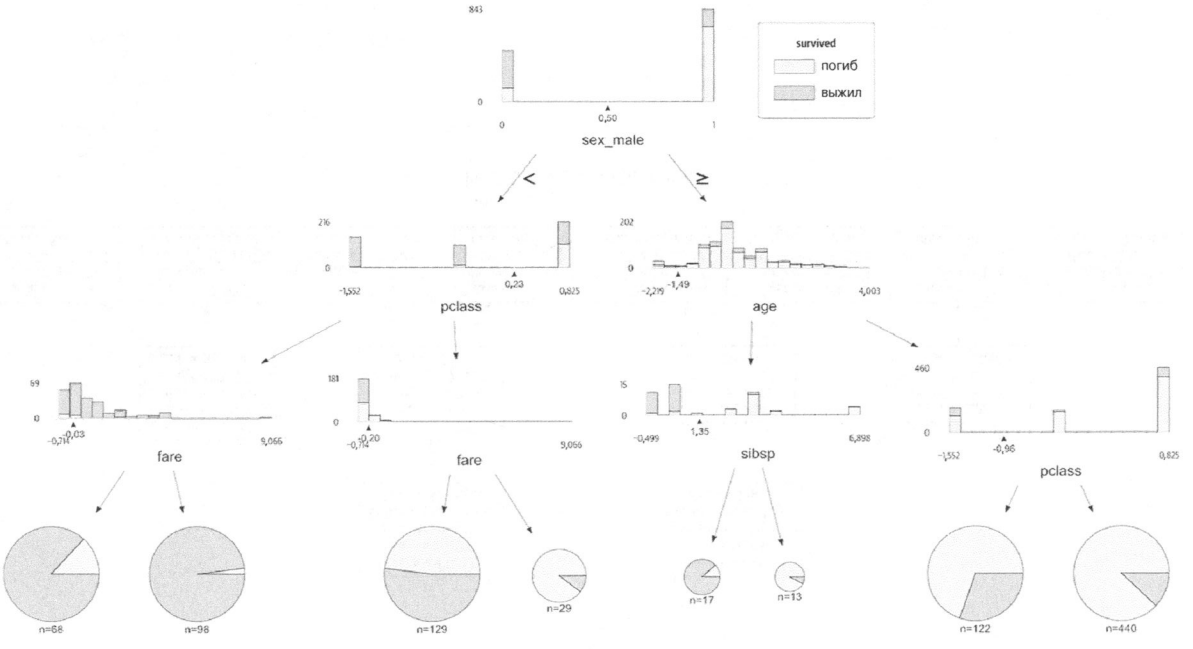


Рис. 10.3. *Вывод dtreeviz*

Для визуализации важности признака вы также можете использовать библиотеку Yellowbrick (рис. 10.4):

```
>>> from yellowbrick.features.importances import (
...     FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(dt)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1004.png", dpi=300)
```

Рис. 10.4. *Важность признака (коэффициент Джини) для дерева решений (нормализована по значимости для мужчин)*

Случайный лес

Случайный лес (random forest) — это совокупность деревьев решений. Для коррекции тенденции деревьев решений к переобучению он использует *бэггинг* (bagging). Дисперсия снижается за счет создания множества деревьев, обученных на случайных подвыборках из выборок и случайных признаках.

Поскольку они обучаются на подвыборках данных, случайные леса могут оценить *ошибку OOB* (OOB error) и производи-

тельность. Они также могут отслеживать важность признаков, усредняя ее по всем деревьям.

Понятие бэггинга основано на эссе маркиза де Кондорсе 1785 года. Его суть в том, что, создавая жюри, вы должны добавить кого-то, имеющего более 50% шансов вынести правильный вердикт, а затем усреднить свои решения. Каждый раз, когда вы добавляете следующего участника (и процесс его выбора не зависит от других), вы получаете лучший результат.

Идея со случайными лесами заключается в том, чтобы создать "лес" деревьев решений, обученных по различным столбцам обучающих данных. Если у каждого дерева больше 50% шансов на правильную классификацию, вы должны включить его прогноз. Случайный лес был отличным инструментом как для классификации, так и для регрессии, хотя в последнее время он потерял популярность из-за деревьев с градиентным бустингом.

Он имеет следующие свойства.

Эффективность выполнения

Необходимо создать j случайных деревьев. Используя `n_jobs`, это можно сделать параллельно. Сложность для каждого дерева — O($mn \log n$), где n — это количество выборок, а m — количество признаков. Для создания осуществите перебор всех m признаков и отсортируйте все n выборок, O($mn \log n$). Для прогнозирования пройдите по дереву O(высота).

Предварительная обработка данных

Не обязательна.

Предотвращение переобучения

Добавьте больше деревьев (`n_estimators`). Используйте меньший `max_depth`.

Интерпретация результатов

Поддерживает важность признаков, но у нас нет единого дерева решений, которое мы могли бы пройти. Можно пройти отдельные деревья из ансамбля.

Вот пример:

```
>>> from sklearn.ensemble import (
...     RandomForestClassifier,
... )
>>> rf = RandomForestClassifier(random_state=42)
>>> rf.fit(X_train, y_train)
RandomForestClassifier(bootstrap=True,
  class_weight=None, criterion='gini',
  max_depth=None, max_features='auto',
  max_leaf_nodes=None, min_impurity_decrease=0.0,
  min_impurity_split=None, min_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0,
  n_estimators=10, n_jobs=1, oob_score=False,
  random_state=42, verbose=0, warm_start=False)
>>> rf.score(X_test, y_test)
0.7862595419847328

>>> rf.predict(X.iloc[[0]])
array([1])
>>> rf.predict_proba(X.iloc[[0]])
array([[0., 1.]])
>>> rf.predict_log_proba(X.iloc[[0]])
array([[-inf,   0.]])
```

Параметры экземпляра (эти параметры отражают дерево решений)

`bootstrap=True`

Начальная загрузка при построении деревьев.

`class_weight=None`

Веса для класса в словаре. `'balanced'` установит значения на обратную пропорцию частот класса. Стандартно — значение 1 для каждого класса. Для множества классов нужен список словарей (OVR) для каждого класса.

`criterion='gini'`

Функция разделения, `'gini'` или `'entropy'`.

`max_depth=None`

Глубина дерева. Стандартно дерево будет строиться до тех пор, пока содержимое листьев меньше `min_samples_split`.

`max_features='auto'`

Количество признаков для проверки на разделение. Стандартно — все.

`max_leaf_nodes=None`

Ограничьте количество листьев. Стандартно — не ограничено.

`min_impurity_decrease=0.0`

Разделять узел, если разделение уменьшит соотношение "инородность > = значение".

`min_impurity_split=None`

Нерекомендуемый.

`min_samples_leaf=1`

Минимальное количество выборок в каждом листе.

`min_samples_split=2`

Минимальное количество выборок, необходимых для разделения узла.

`min_weight_fraction_leaf = 0.0`

Минимальная общая сумма весов, необходимых для конечных узлов.

`n_estimators = 10`

Количество деревьев в лесу.

`n_jobs=1`

Количество заданий для подбора и прогнозирования.

`oob_score=False`

Стоит ли оценивать `oob_score`.

`random_state=None`

Случайное начальное число.

`verbose=0`

Многословность.

`warm_start=False`

Подогнать новый лес или использовать существующий.

Атрибуты после подгонки

`classes_`

Метки класса.

`feature_importances_`

Массив важности Джини.

`n_classes_`

Количество классов.

`n_features_`

Количество признаков.

`oob_score_`

Оценка OOB. Средняя точность для каждого наблюдения, не используемого в деревьях.

Важность признака, демонстрируемая важностью Джини (уменьшение ошибки с использованием этого признака):

```
>>> for col, val in sorted(
...     zip(X.columns, rf.feature_importances_),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
age            0.285
fare           0.268
sex_male       0.232
pclass         0.077
sibsp          0.059
```

СОВЕТ

Классификатор случайных лесов вычисляет важность признака, определяя *среднее снижение инородности* (mean decrease in impurity) для каждого признака (известного также как *индекс важности Джини* (Gini importance)). Признаки, которые уменьшают неопределенность в классификации, получают более высокие оценки.

Эти цифры могут быть искажены, если признаки различаются по масштабу или количеству элементов в категориальных столбцах. Более надежным показателем является *важность перестановки* (permutation importance) (каждый столбец имеет свои переставленные значения и измеряется падение точности). Еще более надежным механизмом является *важность удаления столбца* (drop column importance) (когда каждый столбец отбрасывается, а модель переоценивается), но, к сожалению, для этого необходимо создать новую модель для каждого отброшенного столбца. Рассмотрим функцию `importances` из пакета rfpimp:

```
>>> import rfpimp
>>> rf = RandomForestClassifier(ran
dom_state=42)
>>> rf.fit(X_train, y_train)
>>> rfpimp.importances(
...     rf, X_test, y_test
... ).Importance
Feature
sex_male      0.155216
fare          0.043257
age           0.033079
pclass        0.027990
parch         0.020356
embarked_Q    0.005089
sibsp         0.002545
embarked_S    0.000000
Name: Importance, dtype: float64
```

XGBoost

Хотя в библиотеке Scikit-learn есть класс `GradientBoostedClassifier`, лучше прибегнуть к сторонней реализации, использующей экстремальный бустинг. Она обычно обеспечивает лучшие результаты.

XGBoost — это популярная библиотека, отличная от Scikit-learn. Она создает *слабое дерево* (weak tree), а затем "бустирует" последующие деревья, чтобы уменьшить остаточные ошибки. Она пытается фиксировать и устранять любые шаблоны ошибок до тех пор, пока они не окажутся случайными.

Библиотека XGBoost имеет следующие свойства.

Эффективность выполнения

Библиотека XGBoost параллелизуема. Чтобы указать количество процессоров, используйте параметр `n_jobs`. Для еще лучшей производительности используйте графический процессор.

Предварительная обработка данных

С моделями деревьев масштабирование не требуется. Категориальные данные нужно кодировать.

Предотвращение переобучения

Для остановки обучения может быть установлен параметр `early_stopping_rounds=N`, если после N раундов улучшения не происходит. Регуляризация L1 и L2 контролируется `reg_alpha` и `reg_lambda` соответственно. Более высокие значения дают более консервативные обновления.

Интерпретация результатов

Имеет важность признаков.

В библиотеке XGBoost есть дополнительный параметр для метода `.fit`. Параметр `early_stopping_rounds` можно объединить с параметром `eval_set`, чтобы сообщить XGBoost о прекращении создания деревьев, если показатель оценки не улучшился после стольких циклов бустинга. Для `eval_metric`

также может быть задано одно из следующих значений: 'rmse', 'mae', 'logloss', 'error' (стандартно), 'auc', 'aucpr', а также пользовательская функция.

Вот пример использования библиотеки:

```
>>> import xgboost as xgb
>>> xgb_class = xgb.XGBClassifier(random_state=42)
>>> xgb_class.fit(
...     X_train,
...     y_train,
...     early_stopping_rounds=10,
...     eval_set=[(X_test, y_test)],
... )
XGBClassifier(base_score=0.5, booster='gbtree',
  colsample_bylevel=1, colsample_bytree=1, gamma=0,
  learning_rate=0.1, max_delta_step=0, max_depth=3,
  min_child_weight=1, missing=None,
  n_estimators=100, n_jobs=1, nthread=None,
  objective='binary:logistic', random_state=42,
  reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
  seed=None, silent=True, subsample=1)

>>> xgb_class.score(X_test, y_test)
0.7862595419847328

>>> xgb_class.predict(X.iloc[[0]])
array([1])
>>> xgb_class.predict_proba(X.iloc[[0]])
array([[0.06732017, 0.93267983]], dtype=float32)
```

Параметры экземпляра

```
max_depth=3
```

Максимальная глубина

```
learning_rate=0.1
```

Скорость обучения (или эта) для бустинга (от 0 до 1). После каждого этапа бустинга вновь добавленные веса масштабируются по этому коэффициенту. Чем ниже значение, тем более консервативно обновление, но для схождения потребуется больше деревьев. В вызове .train вы можете передать

параметр `learning_rates`, который представляет собой список частот в каждом раунде (т.е. [0,1] * 100 + [0,05] * 100).

`n_estimators=100`

Количество раундов или расширяемых деревьев.

`silent=True`

Противоположность многословия. Выводит сообщения во время запуска бустинга.

`objective='binary:logistic'`

Задача обучения или вызываемый объект для классификации.

`booster='gbtree'`

Может быть `'gbtree'`, `'gblinear'` или `'dart'`.

`nthread=None`

Нерекомендуемый.

`n_jobs=1`

Количество потоков для использования.

`gamma=0`

Управляет отсечением. Диапазон — от 0 до бесконечности. Для дальнейшего разделения листа необходимо минимальное снижение потерь. Чем выше гамма, тем более консервативно обновление. Если результаты обучения и тестов расходятся, введите бо́льшее число (около 10). Если результаты обучения и тестов близки, используйте меньшее число.

`min_child_weight=1`

Минимальное значение суммы гессиана для листа.

`max_delta_step=0`

Делает обновление более консервативным. Установите равным от 1 до 10 для несбалансированных классов.

`subsample=1`

Доля выборок для использования в следующем раунде.

`colsample_bytree=1`

Доля столбцов, используемых для раунда.

`colsample_bylevel=1`

Доля столбцов, используемых на уровне.

`colsample_bynode=1`

Доля столбцов, используемых для узла.

`reg_alpha=0`

Регуляризация L1 (среднее значение весов) способствует разреженности. Увеличьте, чтобы обновления были более консервативны.

`reg_lambda=1`

Регуляризация L2 (корень весов в квадрате) поощряет малые веса. Увеличьте, чтобы обновления были более консервативны.

`scale_pos_weight=1`

Соотношение отрицательного/положительного веса.

`base_score=.5`

Первоначальный прогноз.

`seed=None`

Нерекомендуемый.

`random_state=0`

Случайное начальное число.

`missing=None`

Значение для интерпретации `missing`. `None` означает `np.nan`.

`importance_type='gain'`

Тип важности признака: `'gain'`, `'weight'`, `'cover'`, `'total_gain'` или `'total_cover'`.

Атрибуты

`coef_`

Коэффициенты для учащихся `gblinear`.

`feature_importances_`

Важности признаков для учащихся `gbtree`.

Важность признака (feature importance) — это среднее усиление по всем узлам, в которых используется признак:

```
>>> for col, val in sorted(
...     zip(
...         X.columns,
...         xgb_class.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
fare            0.420
age             0.309
pclass          0.071
sex_male        0.066
sibsp           0.050
```

Библиотека XGBoost может отобразить важность признака (рис. 10.5). Здесь есть параметр `importance_type`. Его стандартным значением является `"weight"`, т.е. количество раз, когда признак появляется в дереве. Это также может быть `"gain"`, когда показывают среднее усиление при использовании признака, или `"cover"`, когда показывают количество выборок, затронутых разделением:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_importance(xgb_class, ax=ax)
>>> fig.savefig("images/mlpr_1005.png", dpi=300)
```

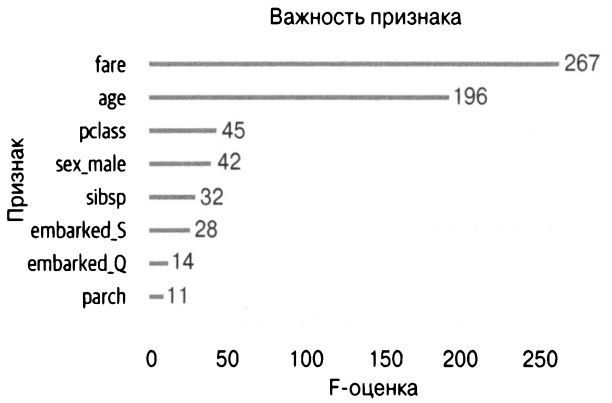


Рис. 10.5. *Важность признака с использованием веса (сколько раз признак встретился в деревьях)*

Мы можем создать график, используя библиотеку Yellowbrick, которая нормализует значения (рис. 10.6):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(xgb_class)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1006.png", dpi=300)
```

Рис. 10.6. *Важность признака Yellowbrick для XGBoost (нормализована до 100)*

Библиотека XGBoost обеспечивает как текстовое представление деревьев, так и графическое. Вот текстовое представление:

```
>>> booster = xgb_class.get_booster()
>>> print(booster.get_dump()[0])
0:[sex_male<0.5] yes=1,no=2,missing=1
  1:[pclass<0.23096] yes=3,no=4,missing=3
    3:[fare<-0.142866] yes=7,no=8,missing=7
      7:leaf=0.132530
      8:leaf=0.184
    4:[fare<-0.19542] yes=9,no=10,missing=9
      9:leaf=0.024598
      10:leaf=-0.1459
  2:[age<-1.4911] yes=5,no=6,missing=5
    5:[sibsp<1.81278] yes=11,no=12,missing=11
    11:leaf=0.13548
    12:leaf=-0.15000
  6:[pclass<-0.95759] yes=13,no=14,missing=13
    13:leaf=-0.06666
    14:leaf=-0.1487
```

Значение листа — это оценка для класса 1. Его можно преобразовать в вероятность с помощью логистической функции. Если решения опустились до листа 7, вероятность класса 1 составляет 53%. Это оценка с одного дерева. Если бы наша модель имела 100 деревьев, мы суммировали бы каждое значение листа и получили бы вероятность с помощью логистической функции:

```
>>> # score from first tree leaf 7
>>> 1 / (1 + np.exp(-1 * 0.1238))
0.5309105310475829
```

Вот графическая версия первого дерева в модели (рис. 10.7):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_tree(xgb_class, ax=ax, num_trees=0)
>>> fig.savefig("images/mlpr_1007.png", dpi=300)
```

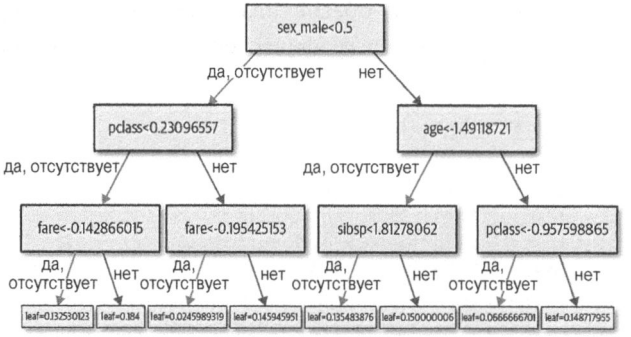


Рис. 10.7. *Дерево XGBoost*

Пакет xgbfir — это библиотека, созданная на основе библиотеки XGBoost. Она предлагает различные меры для важности признака. Уникальным является то, что она предоставляет эти меры для столбцов, а также для пар столбцов, чтобы вы могли видеть взаимодействия. Кроме того, вы можете получить информацию о триплетных (трехстолбцовых) взаимодействиях.

Принимаемые меры таковы.

`Gain`

Общее усиление для каждого признака или признака взаимодействия.

`FScore`

Количество возможных разделений, сделанных для признака или признака взаимодействия.

`wFScore`

Количество возможных разделений, сделанных для признака или признака взаимодействия, взвешенных по вероятности того, что разделение произойдет.

`Average wFScore`

`wFScore`, деленное на `FScore`.

`Average Gain`

`Gain`, деленное на `FScore`.

```
Expected Gain
```

Общее усиление каждого признака или признака взаимодействия, взвешенное по вероятности получения усиления.

Интерфейс (interface) — это просто экспорт в электронную таблицу, поэтому мы будем использовать библиотеку pandas, чтобы прочитать их обратно. Вот важность столбца:

```
>>> import xgbfir
>>> xgbfir.saveXgbFI(
...     xgb_class,
...     feature_names=X.columns,
...     OutputXlsxFile="fir.xlsx",
... )
>>> pd.read_excel("/tmp/surv-fir.xlsx").head(3).T
                          0         1         2
Interaction         sex_male    pclass      fare
Gain                1311.44   585.794   544.884
FScore                   42        45       267
wFScore              39.2892   21.5038    128.33
Average wFScore     0.935458  0.477861  0.480636
Average Gain         31.2247   13.0177   2.04076
Expected Gain        1307.43   229.565   236.738
Gain Rank                  1         2         3
FScore Rank                4         3         1
wFScore Rank               3         4         1
Avg wFScore Rank           1         5         4
Avg Gain Rank              1         2         4
Expected Gain Rank         1         3         2
Average Rank         1.83333   3.16667       2.5
Average Tree Index   32.2381   20.9778   51.9101
Average Tree Depth  0.142857   1.13333   1.50562
```

Из этой таблицы мы видим, что `sex_male` занимает высокие позиции по Gain, средние — по `wFScore`, `Average Gain` и `Expected Gain`, в то время как у fare значения выше, чем у `FScore` и `wFScore`.

Давайте посмотрим на пары столбцов взаимодействий:

```
>>> pd.read_excel(
...     "fir.xlsx",
...     sheet_name="Interaction Depth 1",
```

```
... ).head(2).T
Interaction        pclass|sex_male  age|sex_male
Gain                       2090.27       964.046
FScore                          35             18
wFScore                    14.3608        9.65915
Average wFScore           0.410308       0.536619
Average Gain                59.722        53.5581
Expected Gain               827.49         616.17
Gain Rank                        1              2
FScore Rank                      5             10
wFScore Rank                     4              8
Avg wFScore Rank                 8              5
Avg Gain Rank                    1              2
Expected Gain Rank               1              2
Average Rank               3.33333        4.83333
Average Tree Index         18.9714        38.1111
Average Tree Depth               1        1.11111
```

Здесь верхние два взаимодействия включают столбец `sex_male` в сочетании с `pclass` и `age`. Если бы вы могли создать модель только с двумя признаками, вы, вероятно, захотели бы выбрать `pclass` и `sex_male`.

Наконец, давайте посмотрим триплеты:

```
>>> pd.read_excel(
...     "fir.xlsx",
...     sheet_name="Interaction Depth 2",
... ).head(1).T
                                        0
Interaction         fare|pclass|sex_male
Gain                              2973.16
FScore                                 44
wFScore                           8.92572
Average wFScore                  0.202857
Average                    Gain 67.5719
Expected Gain                     549.145
Gain Rank                               1
FScore Rank                             1
wFScore Rank                            4
Avg wFScore Rank                       21
Avg Gain Rank                           3
Expected Gain Rank                      2
```

```
Average Rank                    5.33333
Average Tree Index              16.6591
Average Tree Depth                    2
```

Это только первый триплет из-за нехватки места, но в таблице есть еще много триплетов:

```
>>> pd.read_excel(
...     "/tmp/surv-fir.xlsx",
...     sheet_name="Interaction Depth 2",
... )[["Interaction", "Gain"]].head()
            Interaction         Gain
0  fare|pclass|sex_male  2973.162529
1   age|pclass|sex_male  1621.945151
2    age|sex_male|sibsp  1042.320428
3     age|fare|sex_male   366.860828
4    fare|fare|sex_male   196.224791
```

Градиентный бустинг с LightGBM

LightGBM — это реализация от Microsoft. Для обработки непрерывных значений LightGBM использует механизм выборки. Это позволяет создавать деревья быстрее (чем, скажем, XGBoost) и уменьшает использование памяти.

LightGBM увеличивает также глубину деревьев (*по листьям* (leaf-wise), а не *по уровню* (level-wise)). Поэтому для контроля переобучения вместо `max_depth` используйте `num_leaves` (где это значение < 2^(`max_depth`)).

НА ЗАМЕТКУ

Установка этой библиотеки в настоящее время требует наличия компилятора и оказывается немного сложнее, чем просто `pip install`.

Свойства таковы.

Эффективность выполнения

Может использовать несколько процессоров. Используя группировку, LightGBM может быть в 15 раз быстрее, чем XGBoost.

Предварительная обработка данных

Имеет некоторую поддержку кодирования категориальных столбцов в виде целых чисел (или тип `Categorical` библиотеки pandas), но AUC, по-видимому, отстает по сравнению с унитарным кодированием.

Предотвращение переобучения

Снизьте `num_leaves`. Увеличьте `min_data_in_leaf`. Используйте `min_gain_to_split` с `lambda_l1` или `lambda_l2`.

Интерпретация результатов

Важность признака доступна. Отдельные деревья слабы и их трудно интерпретировать.

Вот пример использования библиотеки:

```
>>> import lightgbm as lgb
>>> lgbm_class = lgb.LGBMClassifier(
...     random_state=42
... )
>>> lgbm_class.fit(X_train, y_train)
LGBMClassifier(boosting_type='gbdt',
  class_weight=None, colsample_bytree=1.0,
  learning_rate=0.1, max_depth=-1,
  min_child_samples=20, min_child_weight=0.001,
  min_split_gain=0.0, n_estimators=100,
  n_jobs=-1, num_leaves=31, objective=None,
  random_state=42, reg_alpha=0.0, reg_lambda=0.0,
  silent=True, subsample=1.0,
  subsample_for_bin=200000, subsample_freq=0)

>>> lgbm_class.score(X_test, y_test)
0.7964376590330788

>>> lgbm_class.predict(X.iloc[[0]])
array([1])
```

```
>>> lgbm_class.predict_proba(X.iloc[[0]])
array([[0.01637168, 0.98362832]])
```

Параметры экземпляра

```
boosting_type='gbdt'
```

Может быть: `'gbdt'` (gradient boosting — градиентный бустинг), `'rf'` (random forest — случайный лес), `'dart'` (dropouts meet multiple additive regression trees — отбрасывание соответствует множеству аддитивных деревьев регрессии) или `'goss'` (gradient-based, one-sided sampling — односторонняя выборка на основе градиента).

```
class_weight=None
```

Словарь или `'balanced'`. Чтобы установить вес для каждой метки класса при выполнении многоклассовых задач, используйте словарь. Для бинарных задач используйте `is_unbalance` или `scale_pos_weight`.

```
colsample_bytree=1.0
```

Диапазон (0, 1,0]. Выберите процент признака для каждого раунда бустинга.

```
importance_type='split'
```

Как рассчитывать важность признака. `'split'` означает количество раз использования признака. `'gain'` — это общее усиление разделений для признака.

```
learning_rate=0.1
```

Диапазон (0; 1,0]. Скорость обучения для бустинга. Меньшее значение замедляет переобучение, поскольку раунды бустинга оказывают меньшее влияние. Меньшее значение должно дать лучшую производительность, но потребует большее `num_iterations`.

```
max_depth=-1
```

Максимальная глубина дерева. `-1` — не ограничено. Большие глубины ведут к переобучению.

`min_child_samples=20`

Количество выборок, необходимых для листа. Меньшие значения ведут к переобучению.

`min_child_weight=0.001`

Сумма гессиан веса, необходимая для листа.

`min_split_gain=0.0`

Требуется уменьшение потерь для разделения листа.

`n_estimators=100`

Количество деревьев или раундов бустинга.

`n_jobs=-1`

Количество потоков.

`num_leaves=31`

Максимум листьев дерева.

`objective=None`

`None` — это `'binary'` или `'multiclass'` для классификатора. Может быть функцией или строкой.

`random_state=42`

Случайное начальное число.

`reg_alpha=0.0`

Регуляризация L1 (среднее значение весов). Увеличьте, чтобы обновления были более консервативны.

`reg_lambda=0.0`

Регуляризация L2 (корень квадратов весов). Увеличьте, чтобы обновления были более консервативны.

`silent=True`

Режим подробного вывода сообщений.

`subsample=1.0`

Доля выборок для использования в следующем раунде.

`subsample_for_bin=200000`

Выборки, необходимые для создания групп.

`subsample_freq=0`

Частота подвыборки. Чтобы включить, измените на `1`.

Важность признака основана на `'splits'` (количество раз использования продукта):

```
>>> for col, val in sorted(
...     zip(cols, lgbm_class.feature_importances_),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
fare         1272.000
age          1182.000
sibsp         118.000
pclass        115.000
sex_male      110.000
```

Библиотека LightGBM поддерживает создание графика важности признаков (рис. 10.8). Стандартное значение основано на `'splits'`, количестве раз, когда используется признак. Вы можете указать `'importance_type'`, если хотите изменить его на `'gain'`:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_importance(lgbm_class, ax=ax)
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1008.png", dpi=300)
```

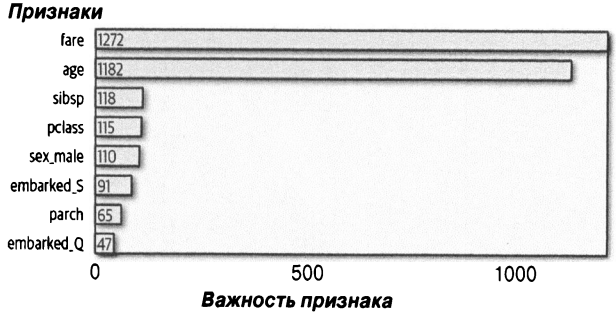


Рис. 10.8. *Важность признака, разделенная для LightGBM*

ВНИМАНИЕ

Начиная с версии 0.9 Yellowbrick не работает с LightGBM при создании графиков важности признаков.

Мы также можем создать дерево решений (рис. 10.9):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_tree(lgbm_class, tree_index=0, ax=ax)
>>> fig.savefig("images/mlpr_1009.png", dpi=300)
```

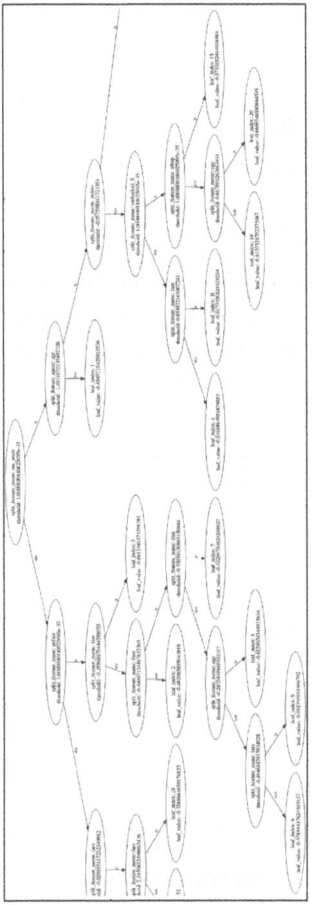


Рис. 10.9. *Дерево LightGBM*

СОВЕТ

В Jupyter используйте следующую команду для просмотра дерева:

```
lgb.create_tree_digraph (lgbm_class)
```

TPOT

TPOT использует генетический алгоритм для опробования различных моделей и ансамблей. Это может занять от нескольких часов до нескольких дней, поскольку учитывает несколько моделей и этапы предварительной обработки, а также гиперпараметры для указанных моделей и параметры ансамблей. На типичной машине запуск может занять пять или более минут.

Свойства таковы.

Эффективность выполнения

Может занять часы или дни. Чтобы использовать все процессоры, установите `n_jobs=-1`.

Предварительная обработка данных

Необходимо удалить `NaN` и категориальные данные.

Предотвращение переобучения

В идеале результаты должны использовать перекрестную проверку для минимизации переобучения.

Интерпретация результатов

Зависит от результатов.

Вот пример использования библиотеки:

```
>>> from tpot import TPOTClassifier
>>> tc = TPOTClassifier(generations=2)
>>> tc.fit(X_train, y_train)
>>> tc.score(X_test, y_test)
0.7888040712468194

>>> tc.predict(X.iloc[[0]])
array([1])
```

```
>>> tc.predict_proba(X.iloc[[0]])
array([[0.07449919, 0.92550081]])
```

Параметры экземпляра

`generations=100`

Итерации для запуска.

`population_size=100`

Размер популяции для генетического программирования. Больший размер обычно работает лучше, но требует больше памяти и времени.

`offspring_size=None`

Потомство для каждого поколения. Стандартно это `population_size`.

`mutation_rate=.9`

Коэффициент мутации для алгоритма [0, 1]. Стандартно — 0,9.

`crossover_rate=.1`

Показатель скрещивания (сколько конвейеров нужно размножить в поколении). Диапазон [0, 1]. Стандартно — 0,1.

`scoring='accuracy'`

Механизм оценки. Использует строки Scikit-learn.

`cv=5`

Перекрестная проверка блоков.

`subsample=1`

Подвыборка учебных экземпляров. Диапазон [0, 1]. Стандартно — 1.

`n_jobs=1`

Количество используемых процессоров, для всех ядер — `-1`.

`max_time_mins=None`

Максимальное количество минут выполнения.

`max_eval_time_mins=5`

Максимальное количество минут для оценки одного конвейера.

`random_state=None`

Случайное начальное число.

`config_dict`

Варианты конфигурации для оптимизации.

`warm_start=False`

Повторное использование предыдущих вызовов `.fit`.

`memory=None`

Может кешировать конвейеры. `'auto'` или путь будет сохранен в каталоге.

`use_dask=False`

Использовать dask.

`periodic_checkpoint_folder=None`

Путь к папке, в которой лучший конвейер будет периодически сохраняться.

`early_stop=None`

Остановитесь после запуска этого поколения без каких-либо улучшений.

`verbosity=0`

0 — нет, 1 — минимально, 2 — высоко или 3 — все. 2 и выше показывает индикатор выполнения.

`disable_update_check=False`

Отключить проверку версии.

Атрибуты

`evaluated_individuals_`

Словарь со всеми конвейерами, которые были оценены.

```
fitted_pipeline_
```

Лучший конвейер.

После этого вы можете экспортировать конвейер:

```
>>> tc.export("tpot_exported_pipeline.py")
```

Результат может выглядеть так:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import \
    train_test_split
from sklearn.pipeline import make_pipeline, \
    make_union
from sklearn.preprocessing import Normalizer
from tpot.builtins import StackingEstimator

# ПРИМЕЧАНИЕ: убедитесь, что класс помечен как
# 'target' в файле данных
tpot_data = pd.read_csv('PATH/TO/DATA/FILE',
    sep='COLUMN_SEPARATOR', dtype=np.float64)
features = tpot_data.drop('target', axis=1).values
training_features, testing_features, \
    training_target, testing_target = \
    train_test_split(features,
        tpot_data['target'].values, random_state=42)

# Оценка на учебном наборе была:0.8122535043953432
exported_pipeline = make_pipeline(
    Normalizer(norm="max"),
    StackingEstimator(
      estimator=ExtraTreesClassifier(bootstrap=True,
        criterion="gini", max_features=0.85,
        min_samples_leaf=2, min_samples_split=19,
        n_estimators=100)),
    ExtraTreesClassifier(bootstrap=False,
      criterion="entropy", max_features=0.3,
      min_samples_leaf=13, min_samples_split=9,
      n_estimators=100)
)

exported_pipeline.fit(training_features,
  training_target)
results = exported_pipeline.predict(
  testing_features)
```

ГЛАВА 11

Выбор модели

В этой главе будет обсуждаться оптимизация гиперпараметров. Также будет рассмотрен вопрос о том, требуется ли модели больше данных для лучшей работы.

Кривая валидации

Создание кривой валидации является одним из способов определения подходящего значения для гиперпараметра. *Кривая валидации* (validation curve) — это график, показывающий, как производительность модели реагирует на изменения значения гиперпараметра (рис. 11.1). Диаграмма демонстрирует как учебные, так и проверочные данные. *Оценки валидации* (validation score) позволяют определить, как модель будет реагировать на новые данные. Как правило, мы выбираем гиперпараметр, который максимизирует оценку валидации.

В следующем примере мы будем использовать библиотеку Yellowbrick, чтобы увидеть, меняет ли значение гиперпараметра `max_depth` производительность модели случайного леса. Вы можете предоставить набор параметров `scoring` метрике модели Scikit-learn (стандартно для классификации используется `'accuracy'`).

СОВЕТ

Чтобы воспользоваться преимуществом наличия нескольких процессоров и работать быстрее, используйте параметр `n_jobs`. Если вы установите его равным `-1`, будут использованы все процессоры.

```
>>> from yellowbrick.model_selection import (
...     ValidationCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> vc_viz = ValidationCurve(
...     RandomForestClassifier(n_estimators=100),
...     param_name="max_depth",
...     param_range=np.arange(1, 11),
...     cv=10,
...     n_jobs=-1,
... )
>>> vc_viz.fit(X, y)
>>> vc_viz.poof()
>>> fig.savefig("images/mlpr_1101.png", dpi=300)
```

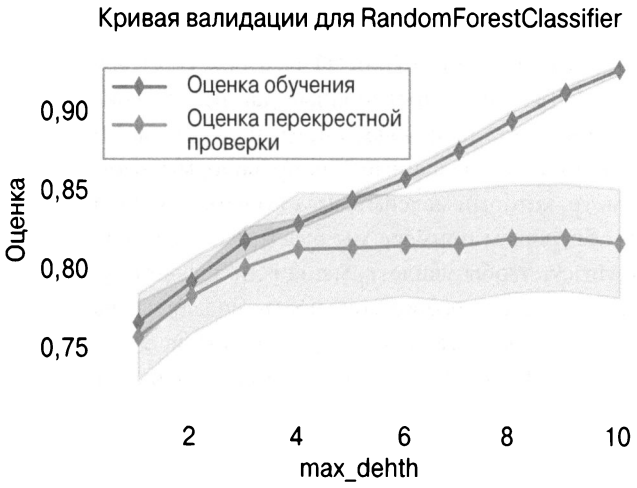


Рис. 11.1. *Отчет кривой валидации*

Класс `ValidationCurve` имеет параметр `scoring`. Параметр может быть пользовательской функцией или одним из следующих значений, в зависимости от задачи.

Параметр `scoring` для классификации: `'accuracy'`, `'average_precision'`, `'f1'`, `'f1_micro'`, `'f1_macro'`, `'f1_weighted'`, `'f1_samples'`, `'neg_log_loss'`, `'precision'`, `'recall'` и `'roc_auc'`.

Параметр `scoring` для кластеризации: `'adjusted_mutual_info_score'`, `'adjusted_rand_score'`, `'completeness_score'`, `'fowlkesmallows_score'`, `'homogeneity_score'`, `'mutual_info_score'`, `'normalized_mutual_info_score'` и `'v_measure_score'`.

Параметр `scoring` для регрессии: `'explained_variance'`, `'neg_mean_absolute_error'`, `'neg_mean_squared_error'`, `'neg_mean_squared_log_error'`, `'neg_median_absolute_error'` и `'r2'`.

Кривая обучения

Сколько данных вам нужно, чтобы выбрать наилучшую модель для своего проекта? Кривая обучения может помочь ответить на этот вопрос. Она отображает результаты обучения и перекрестной проверки при создании моделей с большим количеством выборок. Например, если оценка перекрестной проверки продолжает расти, это может указывать на то, что увеличение количества данных поможет модели работать лучше.

Следующая визуализация показывает кривую валидации, а также помогает исследовать смещение и дисперсию в нашей модели (рис. 11.2). Если в оценке обучения есть изменчивость (большая заштрихованная область), то модель страдает от ошибки смещения и является слишком простой (недообученной). Если изменчивость есть в перекрестной проверке, то модель страдает от ошибки дисперсии и является слишком сложной (переобучение). Другим признаком переобучения модели

является то, что производительность на проверочном наборе намного хуже, чем на обучающем наборе.

Вот пример создания кривой обучения с использованием Yellowbrick:

```
>>> from yellowbrick.model_selection import (
...     LearningCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lc3_viz = LearningCurve(
...     RandomForestClassifier(n_estimators=100),
...     cv=10,
... )
>>> lc3_viz.fit(X, y)
>>> lc3_viz.poof()
>>> fig.savefig("images/mlpr_1102.png", dpi=300)
```

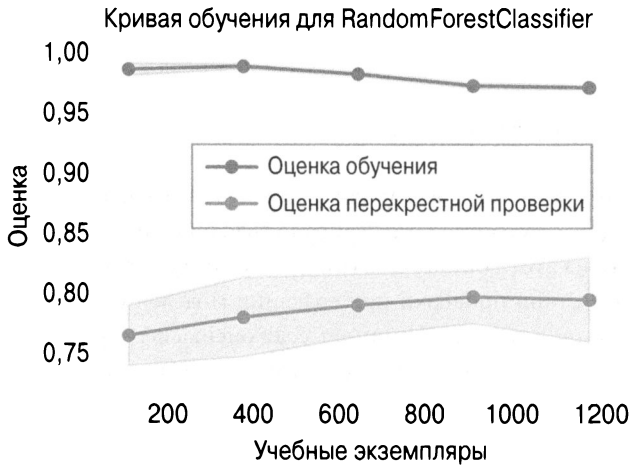


Рис. 11.2. *График кривой обучения. Плато в оценке валидации указывает, что добавление большего количества данных не улучшит эту модель*

Изменив параметры оценки, эту визуализацию можно также использовать для регрессии или кластеризации.

ГЛАВА 12

Метрики и оценка классификации

В этой главе мы рассмотрим следующие метрики и инструменты оценки: матрицы неточностей, различные метрики, отчет о классификации и некоторые визуализации.

Оцениваться будет модель дерева решений, прогнозирующая выживание на Титанике.

Матрица неточностей

Матрица неточностей (confusion matrix) может помочь понять, как работает классификатор.

Двоичный классификатор может иметь четыре результата классификации: истинно позитивные (true positive — TP), истинно негативные (true negative — TN), ложно позитивные (false positive — FP) и ложно негативные (false negative — FN). Первые два являются правильными классификациями.

Вот типичный пример для запоминания других результатов. Если предположить, что "позитивный" — означает "беременность", а "негативный" — "не беременность", то ложно позитивный результат — это как утверждение, что мужчина беременен. Ложно негативное утверждение — что женщина не беременна (когда это явно не так) (рис. 12.1). Эти две последние ошибки называются ошибками *типа 1* и *типа 2* соответственно (табл. 12.1).

Еще один способ запомнить это — P (для ложно позитивных) имеет одну прямую линию (ошибка типа 1), а N (для ложно негативных) имеет две вертикальные линии.

Рис. 12.1. *Ошибки классификации*

Таблица 12.1. Результаты двоичной классификации по матрице неточностей

Факт	Прогноз негативный	Прогноз позитивный
Фактически негативный	Истинно негативный	Ложно позитивный (тип 1)
Фактически позитивный	Ложно негативный (тип 2)	Истинно позитивный

Вот код pandas для расчета результатов классификации. Комментарии показывают результаты. Мы будем использовать эти переменные для расчета других метрик:

```
>>> y_predict = dt.predict(X_test)
>>> tp = (
...     (y_test == 1) & (y_test == y_predict)
... ).sum() # 123
>>> tn = (
...     (y_test == 0) & (y_test == y_predict)
... ).sum() # 199
>>> fp = (
...     (y_test == 0) & (y_test != y_predict)
... ).sum() # 25
```

```
>>> fn = (
...     (y_test == 1) & (y_test != y_predict)
... ).sum() # 46
```

Хорошие классификаторы в идеале имеют высокий счет по истинной диагонали. Мы можем создать объект DataFrame с помощью функции Scikit-learn `confusion_matrix`:

```
>>> from sklearn.metrics import confusion_matrix
>>> y_predict = dt.predict(X_test)
>>> pd.DataFrame(
...     confusion_matrix(y_test, y_predict),
...     columns=[
...         "Predict died",
...         "Predict Survive",
...     ],
...     index=["True Death", "True Survive"],
... )
              Predict died  Predict Survive
True Death             199               25
True Survive            46              123
```

В Yellowbrick есть график для матрицы неточностей (рис. 12.2):

```
>>> import matplotlib.pyplot as plt
>>> from yellowbrick.classifier import (
...     ConfusionMatrix,
... )
>>> mapping = {0: "died", 1: "survived"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
...     dt,
...     classes=["died", "survived"],
...     label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig("images/mlpr_1202.png", dpi=300)
```

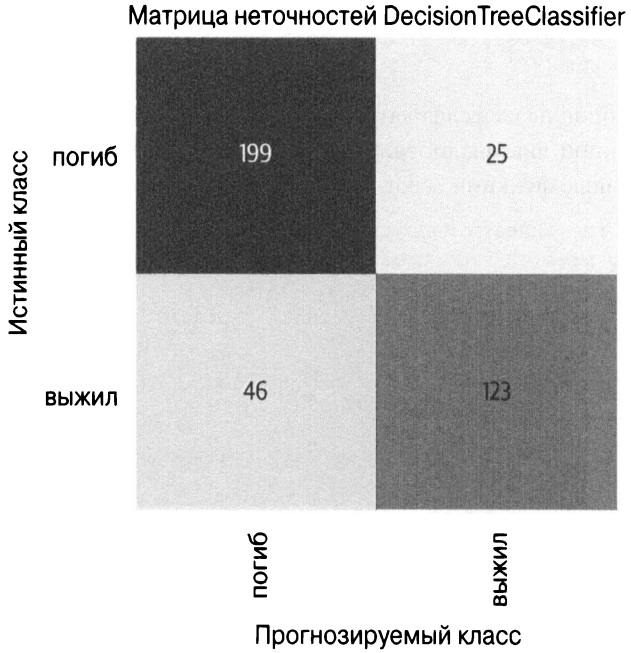


Рис. 12.2. *Матрица неточностей. Вверху слева и внизу справа указаны правильные классификации. Внизу слева — ложно негативная. В верхнем правом углу — ложно позитивная*

Метрики

Модуль `sklearn.metrics` реализует множество общих метрик классификации.

`'accuracy'`

Процент правильных прогнозов.

`'average_precision'`

Сводка кривой точного отзыва.

`'f1'`

Гармоническое среднее точности и отзыва.

`'neg_log_loss'`

Логистическая или кросс-энтропийная потеря (модель должна поддерживать `predict_proba`).

`'precision'`

Возможность поиска только релевантных выборок (не помечать негатив как позитив).

`'recall'`

Возможность найти все позитивные выборки.

`'roc_auc'`

Площадь под кривой рабочей характеристики приемника.

Эти строки можно использовать в качестве параметра `scoring` при выполнении сеточного поиска, или вы можете использовать функции из модуля `sklearn.metrics`, которые имеют те же имена, что и строки, но оканчиваются на `_score`.

НА ЗАМЕТКУ

`'f1'`, `'precision'` и `'recall'` поддерживают следующие суффиксы для многоклассовых классификаторов.

`'_micro'`

Глобальное средневзвешенное значение метрики.

`'_macro'`

Невзвешенное среднее метрики.

`'_weighted'`

Многоклассовое взвешенное среднее метрики.

`'_samples'`

Метрика по каждой выборке.

Корректность

Корректность (accuracy) — это процент правильных классификаций:

```
>>> (tp + tn) / (tp + tn + fp + fn)
0.8142493638676844
```

Что такое хорошая корректность? Все зависит от обстоятельств. Прогнозируя мошенничество (которое обычно является редким событием, скажем 1 на 10000), я могу получить очень высокую корректность, всегда прогнозируя не мошенничество. Но эта модель не очень полезна. Анализ других метрик и стоимости прогнозирования ложно позитивных и ложно негативных может помочь нам определить, хороша ли модель.

Для расчета мы можем использовать библиотеку Scikit-learn:

```
>>> from sklearn.metrics import accuracy_score
>>> y_predict = dt.predict(X_test)
>>> accuracy_score(y_test, y_predict)
0.8142493638676844
```

Отзыв

Отзыв (recall) (или *чувствительность* (sensitivity)) — это процент позитивных значений, классифицированных правильно. (Количество возвращаемых релевантных результатов?)

```
>>> tp / (tp + fn)
0.7159763313609467

>>> from sklearn.metrics import recall_score
>>> y_predict = dt.predict(X_test)
>>> recall_score(y_test, y_predict)
0.7159763313609467
```

Точность

Точность (precision) — это процент позитивных прогнозов, классифицированных правильно (TP, деленное на (TP + FP)). (Насколько релевантны результаты?)

```
>>> tp / (tp + fp)
0.8287671232876712
>>> from sklearn.metrics import precision_score
>>> y_predict = dt.predict(X_test)
>>> precision_score(y_test, y_predict)
0.8287671232876712
```

F1

F1 — это гармоническое среднее отзыва и точности:

```
>>> pre = tp / (tp + fp)
>>> rec = tp / (tp + fn)
>>> (2 * pre * rec) / (pre + rec)
0.7682539682539683

>>> from sklearn.metrics import f1_score
>>> y_predict = dt.predict(X_test)
>>> f1_score(y_test, y_predict)
0.7682539682539683
```

Отчет о классификации

В библиотеке Yellowbrick есть отчет о классификации, демонстрирующий оценки точности, отзыва и f1 как для позитивных, так и для негативных значений (рис. 12.3). Цвет здесь имеет значение; чем ячейка краснее (ближе к единице), тем лучше оценка:

```
>>> import matplotlib.pyplot as plt
>>> from yellowbrick.classifier import (
...     ClassificationReport,
... )
```

```
>>> fig, ax = plt.subplots(figsize=(6, 3))
>>> cm_viz = ClassificationReport(
...     dt,
...     classes=["died", "survived"],
...     label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig("images/mlpr_1203.png", dpi=300)
```

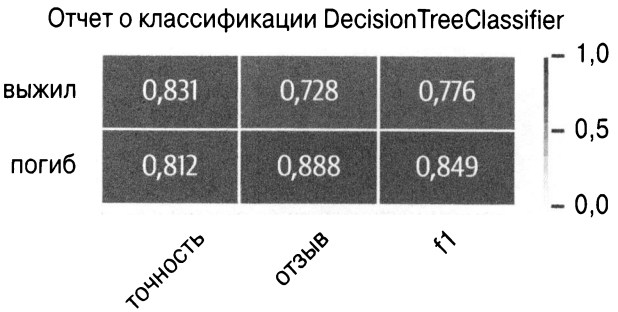


Рис. 12.3. *Отчет о классификации*

ROC

Кривая ROC иллюстрирует работу классификатора, отслеживая частоту истинно позитивных результатов (отзыв/чувствительность) при изменении частоты ложно позитивных (инверсная специфичность) (рис. 12.4).

Эмпирическое правило: график должен подниматься в верхнему левом углу. График, который находится слева и выше другого графика, указывает на лучшую производительность. Диагональ на этом графике указывает на поведение классификатора случайного угадывания. AUC дает метрику оценки производительности:

```
>>> from sklearn.metrics import roc_auc_score
>>> y_predict = dt.predict(X_test)
>>> roc_auc_score(y_test, y_predict)
0.8706304346418559
```

Его можно построить, используя Yellowbrick:

```
>>> from yellowbrick.classifier import ROCAUC
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(dt)
>>> roc_viz.score(X_test, y_test)
0.8706304346418559
>>> roc_viz.poof()
>>> fig.savefig("images/mlpr_1204.png", dpi=300)
```

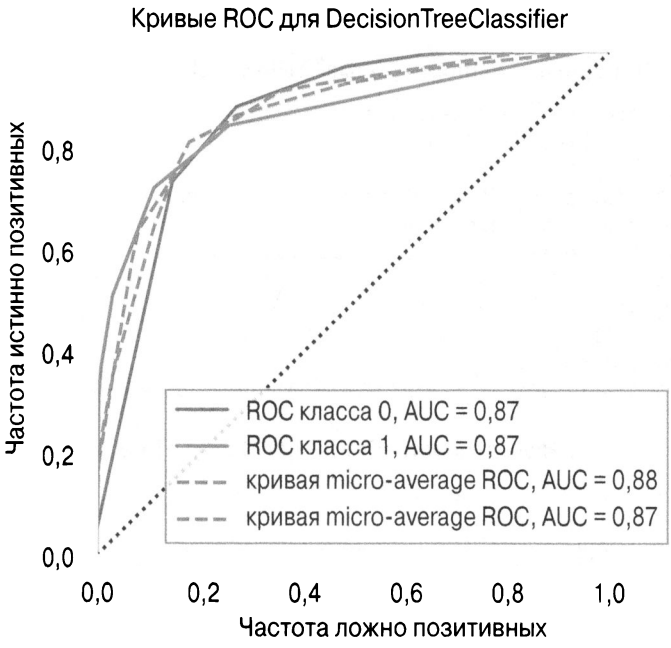


Рис. 12.4. *Кривая ROC*

Кривая "точность–отзыв"

Кривая ROC может быть чрезмерно оптимистичной для несбалансированных классов. Другим вариантом оценки классификаторов является использование кривой "точность–отзыв" (рис. 12.5). *Классификация* (classification) — это

уравновешивающий процесс поиска всего, что вам нужно (отзыв), и в то же время ограничения нежелательных результатов (точность). Обычно это компромисс. По мере роста отзыва точность обычно падает, и наоборот.

```
>>> from sklearn.metrics import (
...     average_precision_score,
... )
>>> y_predict = dt.predict(X_test)
>>> average_precision_score(y_test, y_predict)
0.7155150490642249
```

Вот кривая "точность–отзыв" Yellowbrick:

```
>>> from yellowbrick.classifier import (
...     PrecisionRecallCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> viz = PrecisionRecallCurve(
...     DecisionTreeClassifier(max_depth=3)
... )
>>> viz.fit(X_train, y_train)
>>> print(viz.score(X_test, y_test))
>>> viz.poof()
>>> fig.savefig("images/mlpr_1205.png", dpi=300)
```

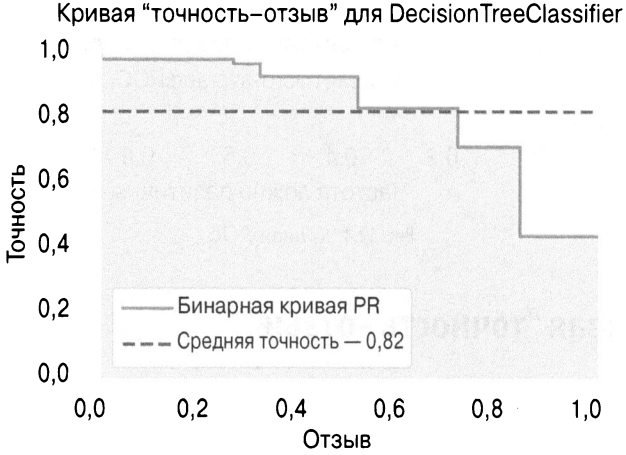


Рис. 12.5. *Кривая "точность–отзыв"*

График кумулятивного усиления

График кумулятивного усиления (cumulative gains plot) может быть использован для оценки двоичного классификатора. Он моделирует *долю истинно позитивных* (true positive rate) (чувствительность) по отношению к *доле поддержки* (support rate) (доля позитивных прогнозов). Принцип, лежащий в основе этого графика, заключается в сортировке всех классификаций по прогнозируемой вероятности. В идеале это должен быть чистый срез, который делит позитивные и негативные выборки. Если первые 10% прогнозов имеют 30% позитивных выборок, вы должны нанести точку от (0, 0) до (0,1, 0,3). Вы продолжаете этот процесс по всем выборкам (рис. 12.6).

Его обычное использование — определение реакции клиента. Кривая кумулятивного усиления показывает уровень поддержки или прогнозируемый позитивный курс по оси X. Наш график помечает это как "Процент выборки" (Percentage of sample). Он отображает чувствительность или долю истинно позитивных по оси Y. На нашем графике это обозначено как "Усиление" (Gain).

Если вам нужен контакт с 90% клиентов, которые ответят (чувствительность), вы можете проследить от 0,9 по оси Y вправо, пока не достигнете этой кривой. Ось X в этой точке будет указывать, к скольким клиентам необходимо обратиться (поддержка), чтобы получить 90%.

В этом случае мы не связываемся с клиентами, которые примут участие и ответят на опрос, а прогнозируем выживание на Титанике. Если бы мы упорядочили всех пассажиров на Титанике в соответствии с вероятностью их выживания согласно нашей модели и взяли бы первые 65% из них, у нас было бы 90% выживших. Если у вас есть связь цены за контакт и дохода за ответ, вы можете рассчитать, какое число лучше.

В общем, модель, которая находится левее и выше другой модели, является лучшей. Лучшими моделями являются линии, идущие вверх (если 10% выборок позитивны, они попадают в

точку (0,1; 1)), а затем непосредственно вправо. Если график находится ниже базовой линии, нам лучше назначать метки случайным образом, чем использовать нашу модель.

Библиотека scikit-plot может создать график кумулятивного усиления:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> y_probas = dt.predict_proba(X_test)
>>> scikitplot.metrics.plot_cumulative_gain(
...     y_test, y_probas, ax=ax
... )
>>> fig.savefig(
...     "images/mlpr_1206.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

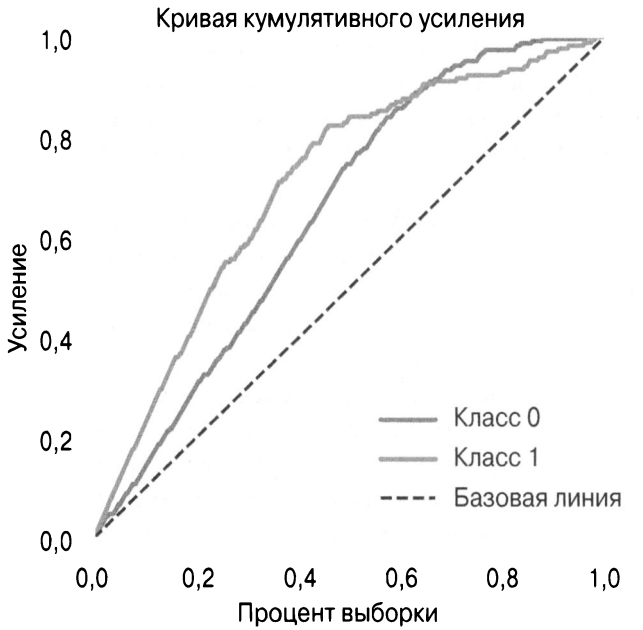


Рис. 12.6. *Кривая кумулятивного усиления. Если бы мы упорядочили людей на Титанике в соответствии с нашей моделью, рассмотрев 20% из них, мы получили бы 40% выживших*

Кривая подъема

Кривая подъема (lift curve) — это еще одно средство, чтобы взглянуть на информацию на графике кумулятивного усиления. Он демонстрирует, насколько мы лучше, чем базовая модель. На нашем графике ниже мы можем видеть, что если бы мы отсортировали наших пассажиров Титаника по вероятности выживания и взяли первые 20% из них, то наш *подъем* (lift) составил бы около 2,2 раза (усиление, деленное на процент выборки) лучше, чем случайный выбор выживших (рис. 12.7). (Мы получили бы в 2,2 раза больше выживших.)

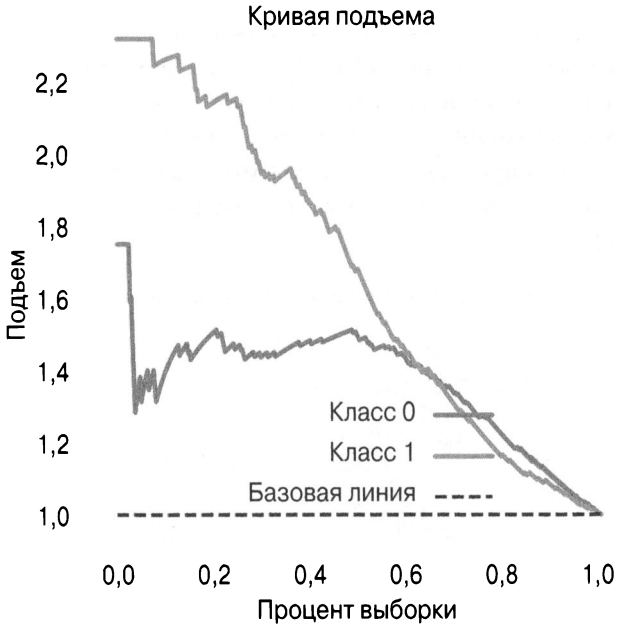


Рис. 12.7. *Кривая подъема*

Библиотека scikit-plot может создать кривую подъема:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> y_probas = dt.predict_proba(X_test)
```

```
>>> scikitplot.metrics.plot_lift_curve(
...     y_test, y_probas, ax=ax
... )
>>> fig.savefig(
...     "images/mlpr_1207.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Баланс классов

В библиотеке Yellowbrick есть простой линейный график для просмотра размеров классов. Когда относительные размеры классов различаются, точность не является хорошим показателем оценки (рис. 12.8). При разделении данных на обучающие и тестовые наборы используйте *стратифицированную выборку* (stratified sampling), чтобы наборы сохраняли относительную пропорцию классов. (Это делает функция `test_train_split`, когда вы устанавливаете параметр `stratify` для меток.)

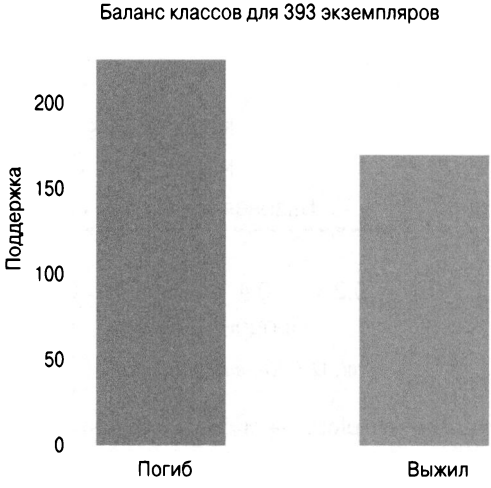


Рис. 12.8. *Небольшой дисбаланс классов*

```
>>> from yellowbrick.classifier import ClassBalance
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cb_viz = ClassBalance(
...     labels=["Died", "Survived"]
... )
>>> cb_viz.fit(y_test)
>>> cb_viz.poof()
>>> fig.savefig("images/mlpr_1208.png", dpi=300)
```

Ошибка прогнозирования класса

График ошибки прогнозирования класса (class prediction error plot) от Yellowbrick — это гистограмма, визуализирующая матрицу неточностей (рис. 12.9):

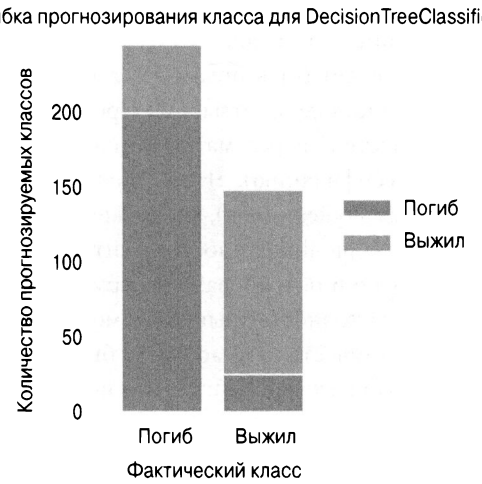


Рис. 12.9. *Ошибка прогнозирования класса. В верхней части левого столбца находятся люди, которые погибли, но мы спрогнозировали, что они выжили (ложно позитивные). В нижней части правого столбца находятся люди, которые выжили, но модель спрогнозировала обратное (ложно негативные)*

```
>>> from yellowbrick.classifier import (
...     ClassPredictionError,
... )
```

```
>>> fig, ax = plt.subplots(figsize=(6, 3))
>>> cpe_viz = ClassPredictionError(
...     dt, classes=["died", "survived"]
... )
>>> cpe_viz.score(X_test, y_test)
>>> cpe_viz.poof()
>>> fig.savefig("images/mlpr_1209.png", dpi=300)
```

Порог дискриминации

Большинство бинарных классификаторов, прогнозирующих вероятность, имеют порог дискриминации 50%. Если прогнозируемая вероятность выше 50%, классификатор назначает позитивную метку. На рис. 12.10 это пороговое значение расположено между 0 и 100 и показаны влияние на точность, отзыв, f1 и коэффициент очереди.

Этот график может быть полезен для просмотра компромисса между точностью и отзывом. Предположим, что мы ищем мошенничество (и рассматриваем мошенничество как позитивную классификацию). Чтобы получить высокий отзыв (поймать всех мошенников), мы можем просто классифицировать все как мошенничество. Но в ситуации с банком это было бы не выгодно и потребовало бы армии рабочих. Чтобы получить высокую точность (выявлять мошенничество только в случае мошенничества), у нас может быть модель, которая срабатывает только в случаях экстремального мошенничества. Но это упустит большую часть мошенничеств, которые могут быть не столь очевидны. Здесь есть компромисс.

Коэффициент очереди (queue rate) — это процент прогнозов выше порога. Вы можете считать это процентом дел, которые нужно рассмотреть, если вы имеете дело с мошенничеством.

Если у вас есть затраты на позитивные, негативные и ошибочные вычисления, можете определить, какой порог вам удобен.

Следующий график полезен, чтобы увидеть, какой порог дискриминации будет максимизировать оценку f1, или скорректировать точность или отзыв в сочетании со скоростью очереди.

Этот визуализатор предоставляет библиотека Yellowbrick. Стандартно он перемешивает данные и запускает 50 испытаний, выделяя 10% для проверки:

```
>>> from yellowbrick.classifier import (
...     DiscriminationThreshold,
... )
>>> fig, ax = plt.subplots(figsize=(6, 5))
>>> dt_viz = DiscriminationThreshold(dt)
>>> dt_viz.fit(X, y)
>>> dt_viz.poof()
>>> fig.savefig("images/mlpr_1210.png", dpi=300)
```

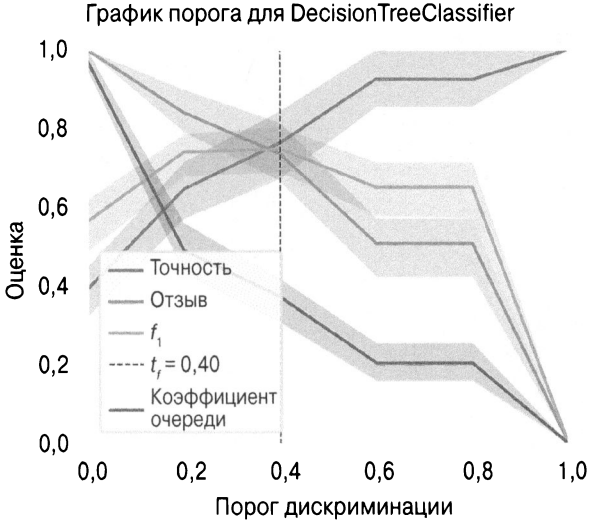


Рис. 12.10. *Порог дискриминации*

ГЛАВА 13

Объяснение моделей

Прогнозирующие модели имеют разные свойства. Одни из них предназначены для обработки линейных данных. Другие способны справляться с более сложным вводом. Одни модели можно интерпретировать очень легко, другие похожи на черные ящики и не дают особого представления о том, как делается прогноз.

В этой главе мы рассмотрим интерпретацию разных моделей. Рассмотрим примеры с использованием набора данных Titanic.

```
>>> dt = DecisionTreeClassifier(
...      random_state=42, max_depth=3
... )
>>> dt.fit(X_train, y_train)
```

Коэффициенты регрессии

Коэффициенты отсечения и регрессии объясняют ожидаемое значение и то, как признаки влияют на прогноз. Положительный коэффициент указывает, что по мере увеличения значения признака прогноз также увеличивается.

Важность признака

Древовидные модели в библиотеке Scikit-learn включают атрибут `.feature_importances_`, предназначенный для проверки того, как признаки набора данных влияют на модель. Мы можем простроить их или отобразить на графике.

LIME

LIME помогает объяснить модели черного ящика. Он выполняет *локальную* (local) интерпретацию, а не общую. Это поможет объяснить одну выборку.

Для данной точки данных или выборки LIME указывает, какие признаки были важны при определении результата. Для этого создается вариация рассматриваемой выборки и осуществляется подгонка к ней линейной модели. Линейная модель аппроксимирует модель ближе к выборке (рис. 13.1).

Вот пример, объясняющий последнюю выборку из обучающих данных (которая, по прогнозам нашего дерева решений, выживет):

```
>>> from lime import lime_tabular
>>> explainer = lime_tabular.LimeTabularExplainer(
...     X_train.values,
...     feature_names=X.columns,
...     class_names=["died", "survived"],
... )
>>> exp = explainer.explain_instance(
...     X_train.iloc[-1].values, dt.predict_proba
... )
```

LIME не любит использовать объекты DataFrame в качестве входных данных. Обратите внимание, что мы преобразовали данные в массивы numpy, используя `.values`.

СОВЕТ

Если вы делаете это в Jupyter, используйте следующий код:

```
exp.show_in_notebook()
```

Он создаст HTML-версию объяснения.

Если нужно экспортировать объяснение (или не использовать Jupyter), можно использовать matplotlib:

```
>>> fig = exp.as_pyplot_figure()
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1301.png")
```

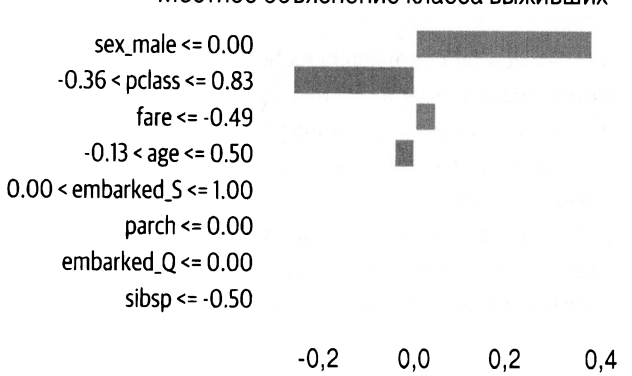


Рис. 13.1. *LIME-объяснение набора данных Titanic. Признаки выборки подталкивают прогноз вправо (выжившие) или влево (погибшие)*

Поэкспериментируйте с этим и обратите внимание, что если вы измените пол, это повлияет на результаты. Ниже мы берем второй, и последний, ряд в учебных данных. Прогноз для этого ряда — 48% погибших и 52% выживших. Изменив пол, мы обнаружим, что прогноз смещается в сторону 88% погибших:

```
>>> data = X_train.iloc[-2].values.copy()
>>> dt.predict_proba(
...     [data]
... )   # прогнозирует, что женщина выживет
[[0.48062016 0.51937984]]
>>> data[5] = 1     # изменить пол на мужской
>>> dt.predict_proba([data])
array([[0.87954545, 0.12045455]])
```

НА ЗАМЕТКУ

Метод `.predict_proba` возвращает вероятность для каждой метки.

Интерпретация дерева

Для моделей на основе дерева Scikit-learn (дерево решений, случайный лес и дополнительные древовидные модели) вы можете использовать пакет treeinterpreter. Это рассчитает смещение и вклад каждого признака. Смещение — это среднее значение учебного набора.

Каждый вклад указывает, какой вклад он вносит в каждую из меток. (Смещение плюс вклад должны суммироваться с прогнозом.) Поскольку это двоичная классификация, их всего два. Мы видим, что sex_male является наиболее важным признаком, а затем — возраст и тариф:

```
>>> from treeinterpreter import (
...     treeinterpreter as ti,
... )
>>> instances = X.iloc[:2]
>>> prediction, bias, contribs = ti.predict(
...     rf5, instances
... )
>>> i = 0
>>> print("Instance", i)
>>> print("Prediction", prediction[i])
>>> print("Bias (trainset mean)", bias[i])
>>> print("Feature contributions:")
```

```
>>> for c, feature in zip(
...     contribs[i], instances.columns
... ):
...     print(" {} {}".format(feature, c))
Instance 0
Prediction [0.98571429 0.01428571]
Bias (trainset mean) [0.63984716 0.36015284]
Feature contributions:
  pclass [ 0.03588478 -0.03588478]
  age [ 0.08569306 -0.08569306]
  sibsp [ 0.01024538 -0.01024538]
  parch [ 0.0100742 -0.0100742]
  fare [ 0.06850243 -0.06850243]
  sex_male [ 0.12000073 -0.12000073]
  embarked_Q [ 0.0026364 -0.0026364]
  embarked_S [ 0.01283015 -0.01283015]
```

НА ЗАМЕТКУ

Это пример классификации, но поддерживается и регрессия.

Графики частичной зависимости

Благодаря важности признаков в деревьях мы знаем, что признак влияет на результат, но мы не знаем, как меняется влияние при изменении значения признака. Графики частичной зависимости позволяют визуализировать связь между изменениями только одного признака и результата. Мы будем использовать pdpbox для визуализации того, как возраст влияет на выживаемость (рис. 13.2).

В этом примере используется модель случайного леса:

```
>>> rf5 = ensemble.RandomForestClassifier(
...     **{
...             "max_features": "auto",
...             "min_samples_leaf": 0.1,
...             "n_estimators": 200,
...             "random_state": 42,
```

```
...     }
... )
>>> rf5.fit(X_train, y_train)

>>> from pdpbox import pdp
>>> feat_name = "age"
>>> p = pdp.pdp_isolate(
...     rf5, X, X.columns, feat_name
... )
>>> fig, _ = pdp.pdp_plot(
...     p, feat_name, plot_lines=True
... )
>>> fig.savefig("images/mlpr_1302.png", dpi=300)
182 |
```

Мы также можем визуализировать взаимодействия между двумя признаками (рис. 13.3):

```
>>> features = ["fare", "sex_male"]
>>> p = pdp.pdp_interact(
...     rf5, X, X.columns, features
... )
>>> fig, _ = pdp.pdp_interact_plot(p, features)
>>> fig.savefig("images/mlpr_1303.png", dpi=300)
```

НА ЗАМЕТКУ

График частичной зависимости фиксирует значение признака по выборкам, а затем усредняет результат. (Будьте осторожны с выбросами и средним.) Кроме того, этот график предполагает, что признаки независимы. (Это не всегда так; например, устойчивое увеличение ширины чашелистика, вероятно, повлияет на высоту.[1]) Библиотека pdpbox выводит также на экран отдельные условные ожидания, чтобы лучше визуализировать эти отношения.

---

[1] Вероятно, имеется в виду еще один учебный набор данных, Iris, из библиотеки Scikit-learn. — *Примеч. ред.*

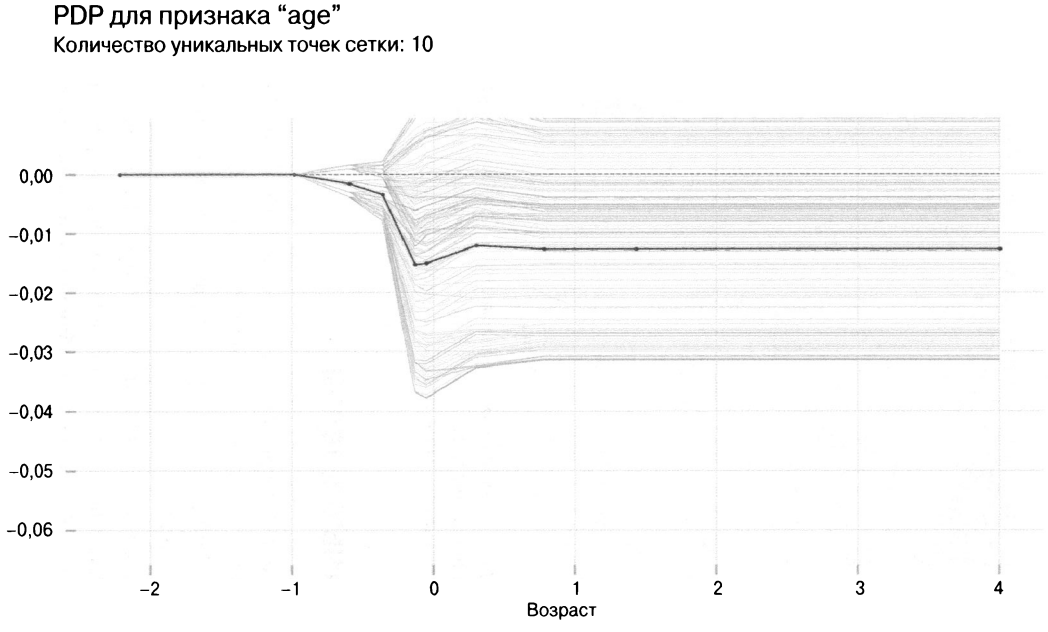


Рис. 13.2. *График частичной зависимости, показывающий, что происходит с целью при изменении возраста*

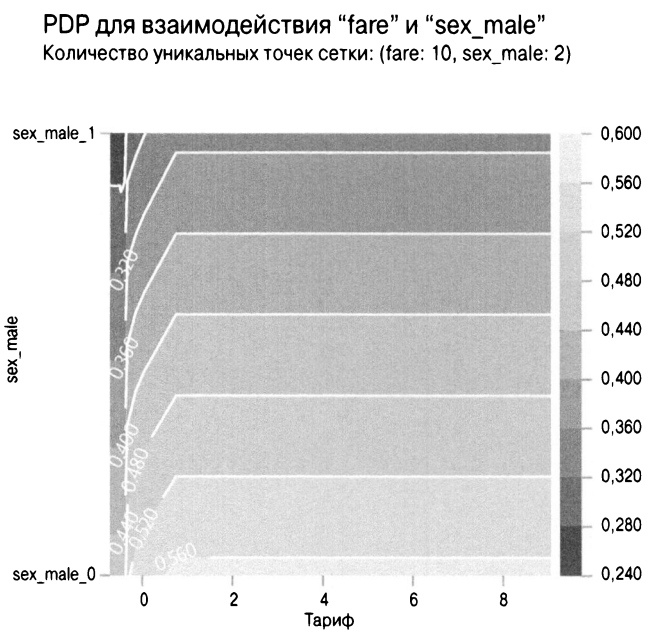


Рис. 13.3. *График частичной зависимости с двумя признаками. При повышении тарифа и смене пола с мужского на женский вероятность выживания увеличивается*

Суррогатные модели

Если у вас неинтерпретируемая модель (SVM или нейронная сеть), можете подогнать к ней интерпретируемую модель (дерево решений). Используя суррогат, вы можете изучить важность признаков.

Здесь мы создаем *классификатор опорных векторов* (Support Vector Classifier — SVC), но обучаем дерево решений (без ограничения глубины для переобучения и выяснения того, что происходит в этой модели), чтобы объяснить это:

```
>>> from sklearn import svm
>>> sv = svm.SVC()
>>> sv.fit(X_train, y_train)
```

```
>>> sur_dt = tree.DecisionTreeClassifier()
>>> sur_dt.fit(X_test, sv.predict(X_test))
>>> for col, val in sorted(
...     zip(
...         X_test.columns,
...         sur_dt.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:7]:
...     print(f"{col:10}{val:10.3f}")
sex_male       0.723
pclass         0.076
sibsp          0.061
age            0.056
embarked_S     0.050
fare           0.028
parch          0.005
```

Shapley

Пакет SHapley Additive exPlanations (SHAP) может визуализировать признаки любой модели. Это действительно хороший пакет, поскольку он не только работает с большинством моделей, он также может объяснять индивидуальные прогнозы и вклад глобальных признаков.

SHAP работает как для классификации, так и для регрессии. Он генерирует значения "SHAP". Для классификационных моделей значение SHAP суммирует логарифмы отношений шансов для двоичной классификации. Для регрессии значения SHAP суммируются с целевым прогнозом.

Для интерактивности на некоторых своих графиках этой библиотеке требуется Jupyter (JavaScript). (Статические изображения можно отображать с помощью matplotlib.) Вот пример для выборки 20, которая прогнозирует гибель:

```
>>> rf5.predict_proba(X_test.iloc[[20]])
array([[0.59223553, 0.40776447]])
```

На графике силы для выборки 20 вы можете увидеть "базовое значение". Это женщина, которая, по прогнозу, погибнет (рис. 13.4). Мы будем использовать индекс выживания (1), поскольку хотим, чтобы правая часть графика была выживанием. Признаки толкают его вправо или влево. Чем больше признак, тем больше его влияние. В этом случае низкие тарифы и третий класс подталкивают к гибели (выходное значение ниже 0,5):

```
>>> import shap
>>> s = shap.TreeExplainer(rf5)
>>> shap_vals = s.shap_values(X_test)
>>> target_idx = 1
>>> shap.force_plot(
...     s.expected_value[target_idx],
...     shap_vals[target_idx][20, :],
...     feature_names=X_test.columns,
... )
```

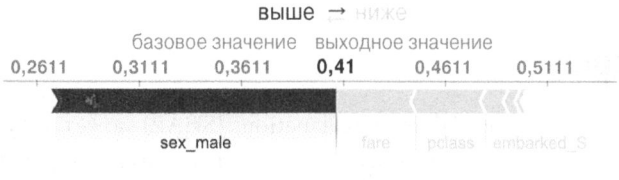


Рис. 13.4. *Вклады признаков Shapley для выборки 20. На этом графике показаны базовое значение и признаки, способствующие гибели*

Вы также можете визуализировать объяснения для всего набора данных (повернув их на 90° и нанеся вдоль оси x) (рис. 13.5):

```
>>> shap.force_plot(
...     s.expected_value[1],
...     shap_vals[1],
...     feature_names=X_test.columns,
... )
```

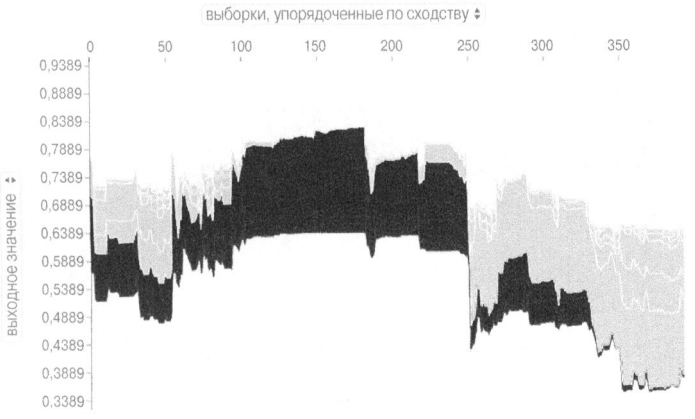


Рис. 13.5. *Вклад признака Shapley в набор данных*

Библиотека SHAP может также генерировать графики зависимости. На следующем графике (рис. 13.6) представлена взаимосвязь между возрастом и значением SHAP (оно окрашено в цвет pclass, который SHAP выбирает автоматически; укажите имя столбца в качестве параметра `interaction_index`, чтобы выбрать собственный):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> res = shap.dependence_plot(
...     "age",
...     shap_vals[target_idx],
...     X_test,
...     feature_names=X_test.columns,
...     alpha=0.7,
... )
>>> fig.savefig(
...     "images/mlpr_1306.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

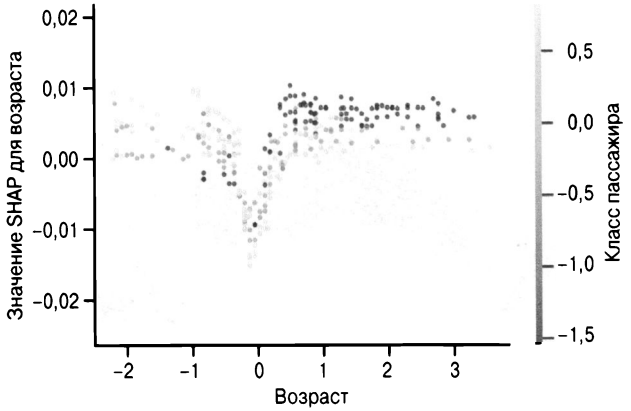


Рис. 13.6. *График зависимости Shapley от возраста. Молодые и старые имеют более высокий уровень выживания. С возрастом у пассажиров более низкого класса появляется больше шансов на выживание*

СОВЕТ

Вы можете получить график зависимости, который имеет вертикальные линии. Установка для параметра `x_jitter` значения 1 полезна, если вы просматриваете порядковые категориальные признаки.

Кроме того, мы можем обобщить все признаки. Это очень мощный график для понимания. Он демонстрирует глобальное воздействие, а также индивидуальное воздействие. Признаки ранжируются по важности. Самые важные признаки находятся вверху.

Признаки также окрашены в соответствии с их значениями. Мы можем видеть, что низкий показатель `sex_male` (женщина) дает сильный толчок к выживанию, в то время как высокий показатель имеет менее сильный толчок к гибели. Интерпретировать возрастную черту немного сложнее. Это связано с тем, что молодые и старые имеют тенденцию к выживанию, а средние — к гибели.

Объединив сводный график с графиком зависимости, вы можете получить хорошее представление о поведении модели (рис. 13.7):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.summary_plot(shap_vals[0], X_test)
>>> fig.savefig("images/mlpr_1307.png", dpi=300)
```

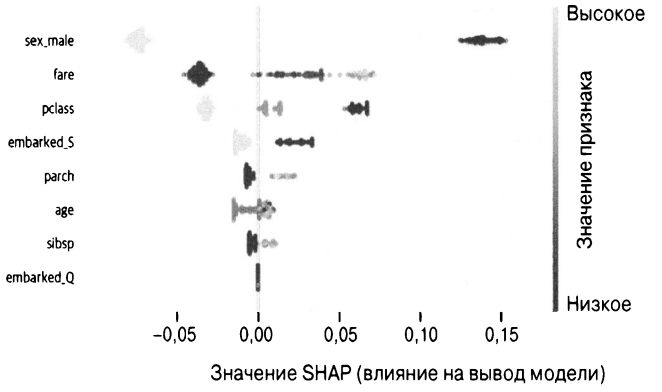


Рис. 13.7. *Сводный график Shapley, демонстрирующий наиболее важные признаки в верхней части. Цветом показано, как значения признака влияют на цель*

ГЛАВА 14

Регрессия

Регрессия (regression) — это процесс машинного обучения с учителем. Это похоже на классификацию, но вместо метки мы пытаемся прогнозировать непрерывное значение. Если вы пытаетесь прогнозировать число, используйте регрессию.

Оказывается, что библиотека Scikit-learn поддерживает множество одинаковых моделей классификации для задач регрессии. На самом деле это одинаковый интерфейс API, вызова методов `.fit`, `.score` и `.predict`. Это также относится к библиотекам бустинга следующего поколения, XGBoost и LightGBM.

Хотя здесь есть сходства с классификационными моделями и гиперпараметрами, показатели оценки у регрессии различны. В этой главе будет рассмотрено множество типов регрессионных моделей. Для их исследования мы будем использовать набор данных Boston, о жилье в Бостоне.

Здесь мы загружаем данные, создаем раздельную версию для обучения и тестирования, а также еще одну раздельную версию со стандартизованными данными:

```
>>> import pandas as pd
>>> from sklearn.datasets import load_boston
>>> from sklearn import (
...     model_selection,
...     preprocessing,
... )
>>> b = load_boston()
>>> bos_X = pd.DataFrame(
```

```
...     b.data, columns=b.feature_names
... )
>>> bos_y = b.target

>>> bos_X_train, bos_X_test, bos_y_train,
bos_y_test = model_selection.train_test_split(
...     bos_X,
...     bos_y,
...     test_size=0.3,
...     random_state=42,
... )

>>> bos_sX = preprocessing.Stand
ardScaler().fit_transform(
...     bos_X
... )
>>> bos_sX_train, bos_sX_test, bos_sy_train,
bos_sy_test = model_selection.train_test_split(
...     bos_sX,
...     bos_y,
...     test_size=0.3,
...     random_state=42,
... )
```

Вот описания признаков набора данных жилья, взятые из самого набора.

CRIM

Уровень преступности на душу населения по городам.

ZN

Доля жилой земли, разбитой на участки площадью более 25 000 квадратных футов (2 322,576 м²).

INDUS

Доля не подлежащих розничной продаже бизнес-акров (акр — 4046,86 м²) на город.

CHAS

Фиктивная переменная реки Чарльз (1, если граничит с рекой, 0 — в противном случае).

NOX

Концентрация оксидов азота (частей на 10 миллионов).

RM

Среднее количество комнат в доме.

AGE

Доля единиц, построенных до 1940 года.

DIS

Взвешенные расстояния до пяти бостонских центров занятости.

RAD

Индекс доступности к радиальным магистралям.

TAX

Полная налоговая ставка на имущество стоимостью более 10 000 долларов США.

PTRATIO

Соотношение учеников и учителей по городам.

B

$1000(Bk - 0{,}63)^2$, где Bk — это доля чернокожих по городам (на 1978 год).

LSTAT

Процент населения с низким состоянием.

MEDV

Средняя стоимость занимаемых домов, с шагом в 1000 долларов.

Базовая модель

Базовая модель регрессии дает нам нечто для сравнения с другими нашими моделями. В библиотеке Scikit-learn результатом стандартного метода `.score` является *коэффициент детерминации* (coefficient of determination) (r^2 или R^2). Это число

объясняет процент вариации исходных данных, которые фиксирует прогноз. Значение обычно составляет от 0 до 1, но может быть даже отрицательным в случае особенно плохих моделей.

Стандартная стратегия `DummyRegressor` заключается в прогнозировании среднего значения обучающего набора. Мы видим, что эта модель не очень хорошо работает:

```
>>> from sklearn.dummy import DummyRegressor
>>> dr = DummyRegressor()
>>> dr.fit(bos_X_train, bos_y_train)
>>> dr.score(bos_X_test, bos_y_test)
-0.03469753992352409
```

Линейная регрессия

Простая линейная регрессия преподается на начальных курсах математики и статистики. Она пытается подогнать формулу $y = mx + b$ к имеющейся форме, минимизируя квадрат ошибок. Когда решение получено, у нас есть отсечение и коэффициент. Отсечение дает базовое значение для прогноза, модифицированного в ходе добавления произведения коэффициента и входных данных.

Эта форма может быть обобщена на более высокие измерения. В этом случае каждый признак имеет коэффициент. Чем больше абсолютное значение коэффициента, тем больше влияние признака на цель.

Эта модель предполагает, что прогноз представляет собой линейную комбинацию входных данных. Для некоторых наборов данных это недостаточно гибко. Сложность может быть добавлена за счет преобразования признаков (преобразователь Scikit-learn `preprocessing.PolynomialFeatures` способен создавать полиномиальные комбинации признаков). Если это приводит к переобучению, то для регуляризации оценщика можно использовать регрессию лассо или гребневую.

Эта модель подвержена также *гетероскедастичности* (heteroscedasticity). Идея в том, что при изменении входных зна-

чений размер ошибки прогноза (или остатки) также часто меняется. Если вы отобразите входные данные относительно остатков, то увидите форму лопасти вентилятора или конуса. Мы рассмотрим примеры позже.

Другая проблема, о которой следует знать, — это *мультиколлинеарность* (multicollinearity). Если столбцы имеют высокую корреляцию, это может затруднить интерпретацию коэффициентов. На модель это обычно не влияет, влияет только на смысл коэффициента.

Модель линейной регрессии обладает следующими свойствами.

Эффективность выполнения

Для улучшения производительности используйте `n_jobs`.

Предварительная обработка данных

Стандартизируйте данные перед обучением модели.

Предотвращение переобучения

Вы можете упростить модель, не используя или не добавляя полиномиальные элементы.

Интерпретация результатов

Может интерпретировать результаты как веса вклада признаков, но предполагает нормальность и независимость признаков. Возможно, вы захотите удалить коллинеарные признаки для улучшения интерпретации. R2 укажет вам, какое количество общей дисперсии результата объясняется моделью.

Вот пример запуска со стандартными данными:

```
>>> from sklearn.linear_model import (
...     LinearRegression,
... )
>>> lr = LinearRegression()
>>> lr.fit(bos_X_train, bos_y_train)
LinearRegression(copy_X=True, fit_intercept=True,
  n_jobs=1, normalize=False)
>>> lr.score(bos_X_test, bos_y_test)
0.7109203586326287
>>> lr.coef_
```

```
array([-1.32774155e-01,  3.57812335e-02,
        4.99454423e-02,  3.12127706e+00,
       -1.54698463e+01,  4.04872721e+00,
       -1.07515901e-02, -1.38699758e+00,
        2.42353741e-01, -8.69095363e-03,
       -9.11917342e-01,  1.19435253e-02,
       -5.48080157e-01])
```

Параметры экземпляра

`n_jobs=None`

Количество используемых процессоров, для всех ядер — `-1`.

Атрибуты после подгонки

`coef_`

Коэффициенты линейной регрессии.

`intercept_`

Отсечение линейной модели.

Значение `.intercept_` является ожидаемым средним значением. Вы можете увидеть, как масштабирование данных влияет на коэффициенты. Знак коэффициентов объясняет направление связи между признаком и целью. Положительный знак означает, что при увеличении признака метка увеличивается. Отрицательный знак означает, что при увеличении признака метка уменьшается. Чем больше абсолютное значение коэффициента, тем большее влияние он оказывает:

```
>>> lr2 = LinearRegression()
>>> lr2.fit(bos_sX_train, bos_sy_train)
LinearRegression(copy_X=True, fit_intercept=True,
  n_jobs=1, normalize=False)
>>> lr2.score(bos_sX_test, bos_sy_test)
0.7109203586326278
>>> lr2.intercept_
22.50945471291039
>>> lr2.coef_
array([-1.14030209, 0.83368112,  0.34230461,
        0.792002,  -1.7908376,   2.84189278, -0.30234582,
       -2.91772744, 2.10815064, -1.46330017,
       -1.97229956, 1.08930453, -3.91000474])
```

Для визуализации коэффициентов вы можете использовать библиотеку Yellowbrick (рис. 14.1). Поскольку масштабированные данные Boston — это просто массив данных, а не объект pandas DataFrame, нам нужно передать параметр `labels`, если мы хотим использовать имена столбцов:

```
>>> from yellowbrick.features import (
...     FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(
...     lr2, labels=bos_X.columns
... )
>>> fi_viz.fit(bos_sX, bos_sy)
>>> fi_viz.poof()
>>> fig.savefig(
...     "images/mlpr_1401.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

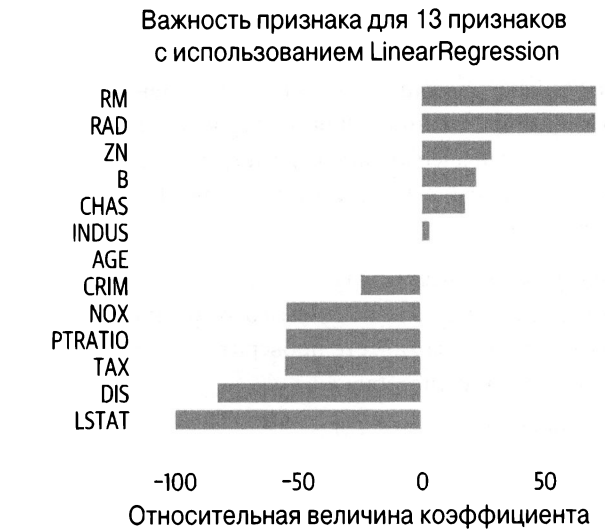


Рис. 14.1. *Важность признака. Указывает, что RM (количество комнат) увеличивает цену, возраст на самом деле не имеет значения, а LSTAT (процент населения с низким состоянием) снижает цену*

SVM

Машины опорных векторов также могут осуществлять регрессию.

SVM имеют следующие свойства.

Эффективность выполнения

Реализация Scikit-learn — $O(n^4)$, поэтому ее сложно масштабировать до больших размеров. Использование линейного ядра или модели `LinearSVC` может улучшить производительность времени выполнения, возможно, за счет точности. Увеличение параметра `cache_size` может привести к уменьшению значения до $O(n^3)$.

Предварительная обработка данных

Алгоритм не является масштабно-инвариантным. Настоятельно рекомендуется стандартизация данных.

Предотвращение переобучения

Параметр `C` (параметр штрафов) контролирует регуляризацию. Меньшее значение допускает меньший зазор в гиперплоскости. Большее значение `gamma` создает тенденцию к переобучению на учебных данных. Для поддержки регуляризации модель `LinearSVC` поддерживает параметры `loss` и `penalty`. Параметр `epsilon` может быть увеличен (при 0 следует ожидать переобучения).

Интерпретация результатов

Проверьте `.support_vectors_`, хотя это трудно объяснить. С линейными ядрами вы можете проверить `.coef_`. Вот пример использования библиотеки:

```
>>> from sklearn.svm import SVR
>>> svr = SVR()
>>> svr.fit(bos_sX_train, bos_sy_train)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3,
  epsilon=0.1, gamma='auto', kernel='rbf',
  max_iter=-1, shrinking=True, tol=0.001,
  verbose=False)
```

```
>>> svr.score(bos_sX_test, bos_sy_test)
0.6555356362002485
```

Параметры экземпляра

`C=1.0`

Параметр штрафа. Чем меньше значение, тем плотнее граница принятия решения (больше переобучение).

`cache_size=200`

Размер кеша (МБ). Увеличение этого показателя может сократить время обучения на больших наборах данных.

`coef0=0.0`

Независимый член для полиномиальных и сигмовидных ядер.

`epsilon=0.1`

Определяет допустимое отклонение, при котором штраф за ошибки не назначается. Для больших наборов данных должно быть меньше.

`degree=3`

Степень для полиномиального ядра.

`gamma='auto'`

Коэффициент ядра. Может быть числом, `'scale'` (стандартно 0,22; 1 / (`num features * X.std()`)) или `'auto'` (стандартно 1 / `num features`). Более низкое значение приводит к переобучению на учебных данных.

`kernel='rbf'`

Тип ядра: `'linear'`, `'poly'`, `'rbf'` (стандартно), `'sigmoid'`, `'precomputed'` или функция.

`max_iter=-1`

Максимальное количество итераций для решателя. `-1` — без ограничений.

`probability=False`

Включить оценку вероятности. Замедляет обучение.

`random_state=None`

Случайное начальное число.

`shrinking=True`

Использовать сокращающуюся эвристику.

`tol=0.001`

Остановка толерантности.

`verbose=False`

Многословность.

Атрибуты после подгонки

`support_`

Индексы опорных векторов.

`support_vectors_`

Опорные векторы.

`coef_`

Коэффициенты (линейного) ядра.

`intercept_`

Константа для функции принятия решения.

K-ближайшие соседи

Модель KNN поддерживает также регрессию, находя k соседних целей для той выборки, для которой создается прогноз. Для регрессии, чтобы определить прогноз, эта модель усредняет все цели вместе.

Модели ближайших соседей имеют следующие свойства.

Эффективность выполнения

Обучение O(1), но есть компромисс, поскольку выборки данных необходимо хранить. Время выполнения тестирования — O(Nd), где N — количество обучающих примеров, а d — размерность.

Предварительная обработка данных

Да, расчеты на основе расстояний лучше выполняются при стандартизации.

Предотвращение переобучения

Увеличить `n_neighbors`. Изменить `p` для метрики L1 или L2.

Интерпретация результатов

Интерпретировать *k*-ближайших соседей к выборке (используя метод `.kneighbors`). Эти соседи объясняют ваш результат (если вы сможете их объяснить).

Вот пример использования модели:

```
>>> from sklearn.neighbors import (
...     KNeighborsRegressor,
... )
>>> knr = KNeighborsRegressor()
>>> knr.fit(bos_sX_train, bos_sy_train)
KNeighborsRegressor(algorithm='auto',
  leaf_size=30, metric='minkowski',
  metric_params=None, n_jobs=1, n_neighbors=5,
  p=2, weights='uniform')

>>> knr.score(bos_sX_test, bos_sy_test)
0.747112767457727
```

Атрибуты

```
algorithm='auto'
```

Может быть `'brute'`, `'ball_tree'` или `'kd_tree'`.

```
leaf_size=30
```

Используется для древовидных алгоритмов.

```
metric='minkowski'
```

Метрика расстояния.

`metric_params=None`

Дополнительный словарь параметров для пользовательской метрической функции.

`n_jobs=1`

Количество процессоров.

`n_neighbors=5`

Количество соседей.

`p=2`

Степенной параметр Минковского: 1 — манхэттен (L1), 2 — евклидово (L2).

`weights='uniform'`

Может быть `'distance'`, в этом случае более близкие точки имеют большее влияние.

Древо решений

Деревья решений поддерживают классификацию и регрессию. На каждом уровне дерева оцениваются различные разделения по признакам. Выбирается разделение, дающее самую низкую ошибку (инородность). Для определения метрики инородности может быть скорректирован параметр `criterion`.

Деревья решений имеют следующие свойства.

Эффективность выполнения

Для создания переберите все m признаков и отсортируйте все n выборок, O(mn log n). Для прогнозирования вы проходите по дереву, O(высота).

Предварительная обработка данных

Масштабирование не обязательно. Нужно избавиться от пропущенных значений и преобразовать их в числовые.

Предотвращение переобучения

Установите для `max_depth` меньшее значение, увеличьте `min_impurity_decrease`.

Интерпретация результатов

Можно пройти по дереву выбора. Поскольку существуют этапы, дерево плохо справляется с линейными отношениями (небольшое изменение в числах, и процесс может пойти другим путем). Дерево также сильно зависит от обучающих данных. Небольшое изменение может изменить все дерево.

Вот пример использования библиотеки Scikit-learn:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> dtr = DecisionTreeRegressor(random_state=42)
>>> dtr.fit(bos_X_train, bos_y_train)
DecisionTreeRegressor(criterion='mse',
  max_depth=None, max_features=None,
  max_leaf_nodes=None, min_impurity_decrease=0.0,
  min_impurity_split=None, min_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0, presort=False,
  random_state=42, splitter='best')

>>> dtr.score(bos_X_test, bos_y_test)
0.8426751288675483
```

Параметры экземпляра

`class_weight=None`

Веса для класса в словаре. `'balanced'` установит значения в обратную пропорцию частот класса. Стандартно это значение `1` для каждого класса. Для множества классов нужен список словарей *один против всех* (one-versus-rest — OVR) для каждого класса.

`criterion='gini'`

Функция разделения, `'gini'` или `'entropy'`.

`criterion='mse'`

Функция разделения. Стандартно — среднеквадратическая ошибка (потеря L2). `'friedman_mse'` или `'mae'` (потери L1).

`max_depth=None`

Глубина дерева. Стандартно дерево будет строиться до тех пор, пока содержимое листьев меньше `min_samples_split`.

`max_features=None`

Количество признаков для проверки на разделение. Стандартно — все.

`max_leaf_nodes=None`

Предельное количество листьев. Стандартно — не ограничено.

`min_impurity_decrease=0.0`

Разделять узел, если разделение уменьшит инородность > = значение.

`min_impurity_split=None`

Нерекомендуемый.

`min_samples_leaf=1`

Минимальное количество выборок в каждом листе.

`min_samples_split=2`

Минимальное количество выборок, необходимых для разделения узла.

`min_weight_fraction_leaf=0.0`

Минимальная сумма весов, необходимая для конечных узлов.

`presort=False`

Может ускорить обучение при небольшом наборе данных или ограниченной глубине, если установлено значение `True`.

`random_state=None`

Случайное начальное число.

`splitter='best'`

Используйте `'random'` или `'best'`.

Атрибуты после подгонки

`feature_importances_`

Массив важности Джини.

`max_features_`

Вычисленное значение `max_features`.

`n_outputs_`

Количество выводов.

`n_features_`

Количество признаков.

`tree_`

Базовый объект дерева.

Рассмотрим дерево (рис. 14.2):

```
>>> import pydotplus
>>> from io import StringIO
>>> from sklearn.tree import export_graphviz
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dtr,
...     out_file=dot_data,
...     feature_names=bos_X.columns,
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1402.png")
```

Для Jupyter используйте:

```
from IPython.display import Image
Image(g.create_png())
```

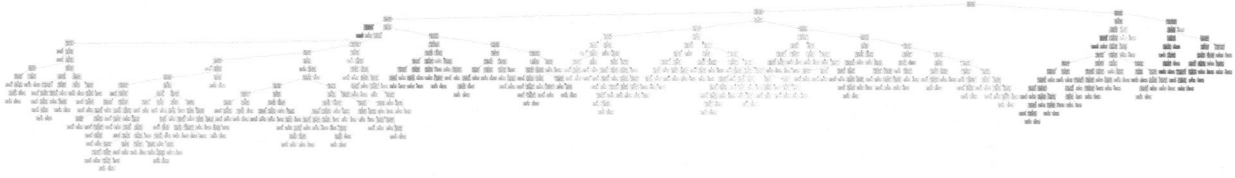


Рис. 14.2. *Дерево решений*

Этот график немного широковат. На компьютере вы можете увеличить его части. Вы также можете ограничить его глубину (рис. 14.3). (Оказывается, что наиболее важные функции обычно находятся в верхней части дерева.) Для этого мы будем использовать параметр `max_depth`:

```
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dtr,
...     max_depth=2,
...     out_file=dot_data,
...     feature_names=bos_X.columns,
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1403.png")
```

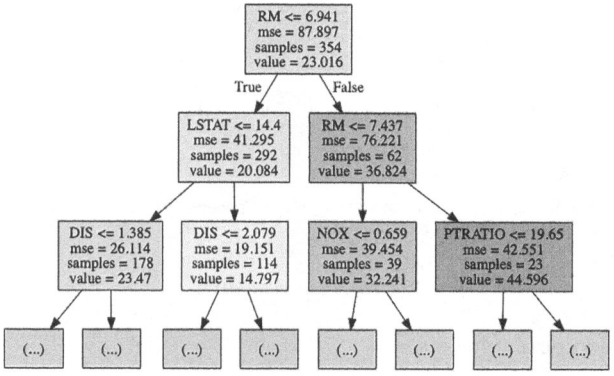


Рис. 14.3. *Первые два слоя дерева решений*

Мы также можем использовать пакет dtreeviz для просмотра графика рассеяния в каждом из узлов дерева (рис. 14.4). Чтобы видеть детали, мы будем использовать дерево, ограниченное глубиной "два":

```
>>> dtr3 = DecisionTreeRegressor(max_depth=2)
>>> dtr3.fit(bos_X_train, bos_y_train)
>>> viz = dtreeviz.trees.dtreeviz(
```

```
...     dtr3,
...     bos_X,
...     bos_y,
...     target_name="price",
...     feature_names=bos_X.columns,
... )
>>> viz
```

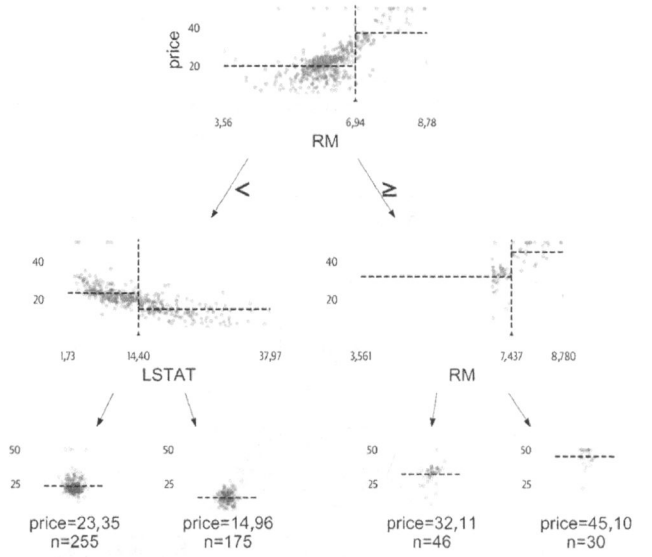


Рис. 14.4. *Регрессия dtviz*

Важность признака:

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, dtr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
RM             0.574
LSTAT          0.191
```

```
DIS             0.110
CRIM            0.061
RAD             0.018
```

Случайный лес

Деревья решений хороши тем, что они объяснимы, но имеют тенденцию к переобучению. Случайный лес меняет часть объяснимости на модель, которая лучше обобщает. Эта модель также может быть использована для регрессии.

Случайные леса имеют следующие свойства.

Эффективность выполнения

Необходимо создать j случайных деревьев. Используя `n_jobs`, это можно сделать параллельно. Сложность для каждого дерева — $O(mn \log n)$, где n — это количество выборок, а m — количество признаков. Для создания осуществите перебор всех m признаков и отсортируйте все n выборок, $O(mn \log n)$. Для прогнозирования пройдите по дереву O(высота).

Предварительная обработка данных

Необязательна, если ввод числовой и нет пропущенных значений.

Предотвращение переобучения

Добавьте больше деревьев (`n_estimators`). Используйте меньший `max_depth`.

Интерпретация результатов

Поддерживает важность признаков, но у нас нет единого дерева решений, которое мы могли бы пройти. Можно пройти отдельные деревья из ансамбля.

Вот пример использования модели:

```
>>> from sklearn.ensemble import (
...     RandomForestRegressor,
... )
>>> rfr = RandomForestRegressor(
```

```
...     random_state=42, n_estimators=100
... )
>>> rfr.fit(bos_X_train, bos_y_train)
RandomForestRegressor(bootstrap=True,
  criterion='mse', max_depth=None,
  max_features='auto', max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0,
  n_estimators=100, n_jobs=1,
  oob_score=False, random_state=42,
  verbose=0, warm_start=False)

>>> rfr.score(bos_X_test, bos_y_test)
0.8641887615545837
```

Параметры экземпляра (отражают дерево решений)

`bootstrap=True`

Начальная загрузка при построении деревьев.

`criterion='mse'`

Функция разделения, `'mse'`.

`max_depth=None`

Глубина дерева. Стандартно дерево будет строиться до тех пор, пока содержимое листьев меньше `min_samples_split`.

`max_features='auto'`

Количество признаков для проверки на разделение. Стандартно — все.

`max_leaf_nodes=None`

Ограничить количество листьев. Стандартно — не ограничено.

`min_impurity_decrease=0.0`

Разделять узел, если разделение уменьшит инородность на это значение или более.

`min_impurity_split=None`

Нерекомендуемый.

`min_samples_leaf=1`

Минимальное количество выборок на каждом листе.

`min_samples_split=2`

Минимальное количество выборок, необходимых для разделения узла.

`min_weight_fraction_leaf=0.0`

Минимальная сумма весов, необходимая для конечных узлов.

`n_estimators=10`

Количество деревьев в лесу.

`n_jobs=None`

Количество заданий для подбора и прогнозирования. (`None` означает "1".)

`oob_score=False`

Использовать ли выборки OOB для оценки по новым данным.

`random_state=None`

Случайное начальное число.

`verbose=0`

Многословность.

`warm_start=False`

Подогнать новый лес или использовать существующий.

Атрибуты после подгонки

`estimators_`

Коллекция деревьев.

`feature_importances_`

Массив важности Джини.

`n_classes_`

Количество классов.

`n_features_`

Количество признаков.

`oob_score_`

Оценка набора обучающих данных с использованием оценки OOB.

Важность признака:

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, rfr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
RM             0.505
LSTAT          0.283
DIS            0.115
CRIM           0.029
PTRATIO        0.016
```

Регрессия XGBoost

Библиотека XGBoost также поддерживает регрессию. Она строит простое (simple) дерево решений, а затем "бустирует" его, добавляя последующие деревья. Каждое дерево пытается исправить остатки предыдущего вывода. На практике это хорошо работает со структурированными данными.

Она имеет следующие свойства.

Эффективность выполнения

Библиотека XGBoost параллелизуема. Чтобы указать количество процессоров, используйте параметр `n_jobs`. Для еще лучшей производительности используйте графический процессор.

Предварительная обработка данных

С моделями деревьев масштабирование не требуется. Категориальные данные нужно кодировать. Есть поддержка для пропущенных данных!

Предотвращение переобучения

Для остановки обучения может быть установлен параметр `early_stopping_rounds=N`, если после *N* раундов улучшения не происходит. Регуляризация L1 и L2 контролируется `reg_alpha` и `reg_lambda` соответственно. Более высокие значения дают более консервативные обновления.

Интерпретация результатов

Имеет важность признаков.

Вот пример использования библиотеки:

```
>>> xgr = xgb.XGBRegressor(random_state=42)
>>> xgr.fit(bos_X_train, bos_y_train)
XGBRegressor(base_score=0.5, booster='gbtree',
  colsample_bylevel=1, colsample_bytree=1,
  gamma=0, learning_rate=0.1, max_delta_step=0,
  max_depth=3, min_child_weight=1, missing=None,
  n_estimators=100, n_jobs=1, nthread=None,
  objective='reg:linear', random_state=42,
  reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
  seed=None, silent=True, subsample=1)

>>> xgr.score(bos_X_test, bos_y_test)
0.871679473122472

>>> xgr.predict(bos_X.iloc[[0]])
array([27.013563], dtype=float32)
```

Параметры экземпляра

`max_depth=3`

Максимальная глубина.

`learning_rate=0.1`

Скорость обучения (или эта) для бустинга (от 0 до 1). После каждого этапа бустинга вновь добавленные веса масштабиру-

ются по этому коэффициенту. Чем ниже значение, тем более консервативно обновление, но для схождения потребуется больше деревьев. В вызове `.train` вы можете передать параметр `learning_rates`, который представляет собой список частот в каждом раунде (т.е. [0,1] * 100 + [0,05] * 100).

`n_estimators=100`

Количество раундов или расширяемых деревьев.

`silent=True`

Выводить сообщения во время запуска бустинга.

`objective="reg:linear"`

Задача обучения или вызываемый объект для классификации.

`booster='gbtree'`

Может быть `'gbtree'`, `'gblinear'` или `'dart'`. Значение `'dart'` добавляет отбрасывание (отбрасывает случайные деревья, чтобы предотвратить переобучение). Значение `'gbtree'` создает регуляризованную линейную модель (не дерево, а похожее на регрессию лассо).

`nthread=None`

Нерекомендуемый.

`n_jobs=1`

Количество потоков для использования.

`gamma=0`

Минимальное снижение потерь, необходимое для дальнейшего разделения листа.

`min_child_weight=1`

Минимальное значение суммы гессиана для листа.

`max_delta_step=0`

Делает обновление более консервативным. Установите от 1 до 10 для несбалансированных классов.

`subsample=1`

Доля выборок для использования в следующем раунде.

`colsample_bytree=1`

Доля столбцов, используемых для раунда.

`colsample_bylevel=1`

Доля столбцов, используемых на уровне.

`colsample_bynode=1`

Доля столбцов, используемых для узла.

`reg_alpha=0`

Регуляризация L1 (среднее значение весов). Увеличьте, чтобы обновления были более консервативны.

`reg_lambda=1`

Регуляризация L2 (корень весов в квадрате). Увеличьте, чтобы обновления были более консервативны.

`base_score=.5`

Первоначальный прогноз.

`seed=None`

Нерекомендуемый.

`random_state=0`

Случайное начальное число.

`missing=None`

Значение для интерпретации `missing`. `None` означает `np.nan`.

`importance_type='gain'`

Тип важности признака: `'gain'`, `'weight'`, `'cover'`, `'total_gain'` или `'total_cover'`.

Атрибуты

`coef_`

Коэффициенты для учащихся `gblinear` (`booster = 'gblinear'`).

`intercept_`

Отсечение для учеников.

`feature_importances_`

Важности признаков для учащихся `gbtree`.

Важность признака — это среднее усиление по всем узлам, в которых используется признак:

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, xgr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
DIS            0.187
CRIM           0.137
RM             0.137
LSTAT          0.134
AGE            0.110
```

Библиотека XGBoost имеет графические объекты для важности признака. Обратите внимание, что параметр `priority_type` изменяет значения в этом графике (рис. 14.5). Стандартно для определения важности признака используется вес:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_importance(xgr, ax=ax)
>>> fig.savefig("images/mlpr_1405.png", dpi=300)
```

Использование библиотеки Yellowbrick для построения графика важности признаков (она нормализует атрибут `feature_importances_`) (рис. 14.6):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(xgr)
```

```
>>> fi_viz.fit(bos_X_train, bos_y_train)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1406.png", dpi=300)
```

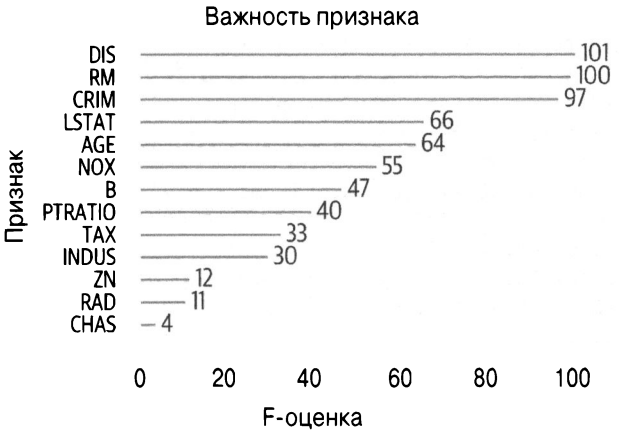


Рис. 14.5. *Важность признака с использованием веса (сколько раз признак встретился в деревьях)*

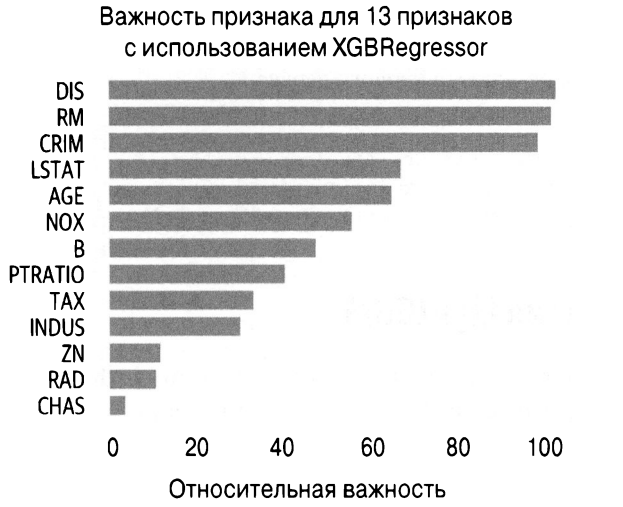


Рис. 14.6. *Важность признака с использованием относительной важности усиления (процент важности наиболее важного признака)*

Библиотека XGBoost обеспечивает как текстовое представление деревьев, так и графическое. Вот текстовое представление:

```
>>> booster = xgr.get_booster()
>>> print(booster.get_dump()[0])
0:[LSTAT<9.72500038] yes=1,no=2,missing=1
 1:[RM<6.94099998] yes=3,no=4,missing=3
  3:[DIS<1.48494995] yes=7,no=8,missing=7
   7:leaf=3.9599998
   8:leaf=2.40158272
  4:[RM<7.43700027] yes=9,no=10,missing=9
   9:leaf=3.22561002
   10:leaf=4.31580687
 2:[LSTAT<16.0849991] yes=5,no=6,missing=5
  5:[B<116.024994] yes=11,no=12,missing=11
   11:leaf=1.1825
   12:leaf=1.99701393
  6:[NOX<0.603000045] yes=13,no=14,missing=13
   13:leaf=1.6868
   14:leaf=1.18572915
```

Значения листа могут быть интерпретированы как сумма `base_score` и листа. (Чтобы проверить это, вызовите `.predict` с параметром `ntree_limit=1` для ограничения модели использованием результата первого дерева.)

Вот графическая версия дерева (рис. 14.7):

```
fig, ax = plt.subplots(figsize=(6, 4))
xgb.plot_tree(xgr, ax=ax, num_trees=0)
fig.savefig('images/mlpr_1407.png', dpi=300)
```

Регрессия LightGBM

Библиотека дерева градиентного бустинга LightGBM также поддерживает регрессию. Как упомянуто в главе о классификации, она может быть быстрее, чем XGBoost, из-за механизма выборки, используемого для определения разделения узла.

Кроме того, помните, что глубина деревьев сначала растет, поэтому ограничение глубины может повредить модели. Он имеет следующие свойства.

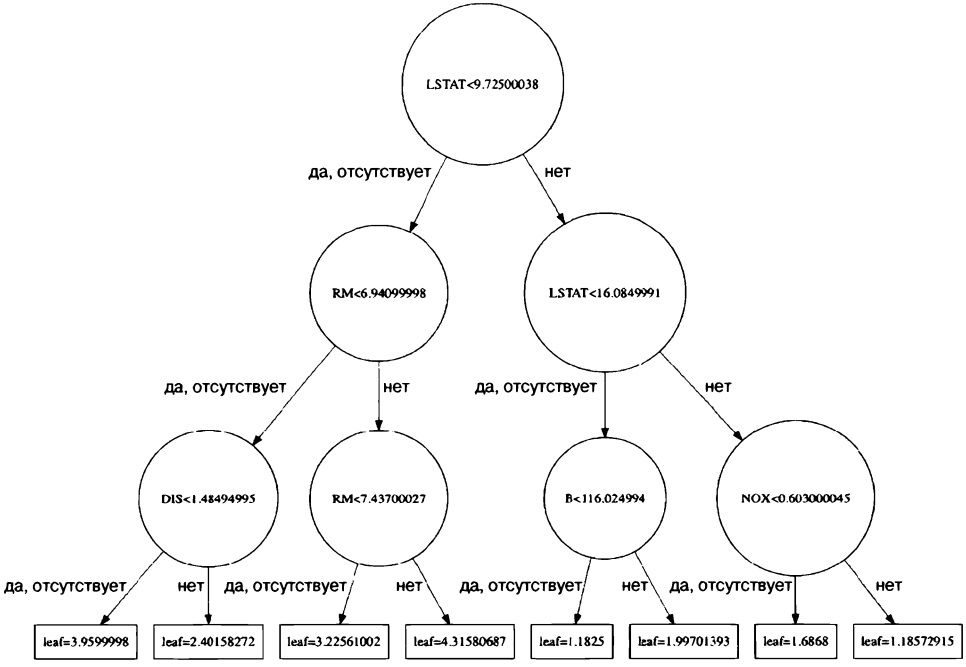


Рис. 14.7. *Дерево XGBoost*

Эффективность выполнения

Может использовать несколько процессоров. Используя группировку, она может быть в 15 раз быстрее, чем XGBoost.

Предварительная обработка данных

Имеет некоторую поддержку кодирования категориальных столбцов в виде целых чисел (или тип `Categorical` библиотеки pandas), но AUC, по-видимому, отстает по сравнению с унитарным кодированием.

Предотвращение переобучения

Снизьте `num_leaves`. Увеличьте `min_data_in_leaf`. Используйте `min_gain_to_split` с `lambda_l1` или `lambda_l2`.

Интерпретация результатов

Важность признака доступна. Отдельные деревья слабы, и их трудно интерпретировать.

Вот пример использования модели:

```
>>> import lightgbm as lgb
>>> lgr = lgb.LGBMRegressor(random_state=42)
>>> lgr.fit(bos_X_train, bos_y_train)
LGBMRegressor(boosting_type='gbdt',
  class_weight=None, colsample_bytree=1.0,
  learning_rate=0.1, max_depth=-1,
  min_child_samples=20, min_child_weight=0.001,
  min_split_gain=0.0, n_estimators=100,
  n_jobs=-1, num_leaves=31, objective=None,
  random_state=42, reg_alpha=0.0,
  reg_lambda=0.0, silent=True, subsample=1.0,
  subsample_for_bin=200000, subsample_freq=0)

>>> lgr.score(bos_X_test, bos_y_test)
0.847729219534575

>>> lgr.predict(bos_X.iloc[[0]])
array([30.31689569])
```

Параметры экземпляра

`boosting_type='gbdt'`

Может быть `'gbdt'` (gradient boosting — градиентный бустинг), `'rf'` (random forest — случайный лес), `'dart'` (dropouts meet multiple additive regression trees — отбрасывание соответствует множеству аддитивных деревьев регрессии) или `'goss'` (gradient-based, one-sided sampling — односторонняя выборка на основе градиента).

`num_leaves=31`

Максимум листьев дерева.

`max_depth=-1`

Максимальная глубина дерева; `-1` — не ограничено. Большие глубины ведут к переобучению.

`learning_rate=0.1`

Диапазон (0; 1,0]. Скорость обучения для бустинга. Меньшее значение замедляет переобучение, поскольку раунды бустинга оказывают меньшее влияние. Меньшее значение должно дать лучшую производительность, но потребует большего `num_iterations`.

`n_estimators=100`

Количество деревьев или раундов бустинга.

`subsample_for_bin=200000`

Выборки, необходимые для создания групп.

`objective=None`

`None` — стандартная регрессия. Может быть функцией или строкой.

`min_split_gain=0.0`

Требуется уменьшение потерь для разделения листа.

`min_child_weight=0.001`

Сумма гессиан веса, необходимая для листа. Чем больше, тем более консервативны обновления.

`min_child_samples=20`

Количество выборок, необходимых для листа. Меньшие цифры означают большее переобучение.

`subsample=1.0`

Доля выборок для использования в следующем раунде.

`subsample_freq=0`

Частота подвыборки. Чтобы включить, измените на `1`.

`colsample_bytree=1.0`

Диапазон (0, 1,0]. Выберите процент признака для каждого раунда бустинга.

`reg_alpha=0.0`

Регуляризация L1 (среднее значение весов). Увеличьте, чтобы обновления были более консервативны.

`reg_lambda=0.0`

Регуляризация L2 (корень квадратов весов). Увеличьте, чтобы обновления были более консервативны.

`random_state=42`

Случайное начальное число.

`n_jobs=-1`

Количество потоков.

`silent=True`

Детальный режим.

`importance_type='split'`

Как рассчитывать важность признака. `'split'` означает количество раз использования признака. `'gain'` — это общее усиление от разделения, когда был использован признак.

Библиотека LightGBM поддерживает важность признака. Ее расчет определяет параметр `importance_type` (стандартное значение зависит от того, сколько раз был использован признак):

```
>>> for col, val in sorted(
...     zip(
...         bos_X.columns, lgr.feature_importances_
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
LSTAT          226.000
RM             199.000
DIS            172.000
AGE            130.000
B              121.000
```

График важности признака, показывающий, сколько раз был использован признак (рис. 14.8):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_importance(lgr, ax=ax)
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1408.png", dpi=300)
```

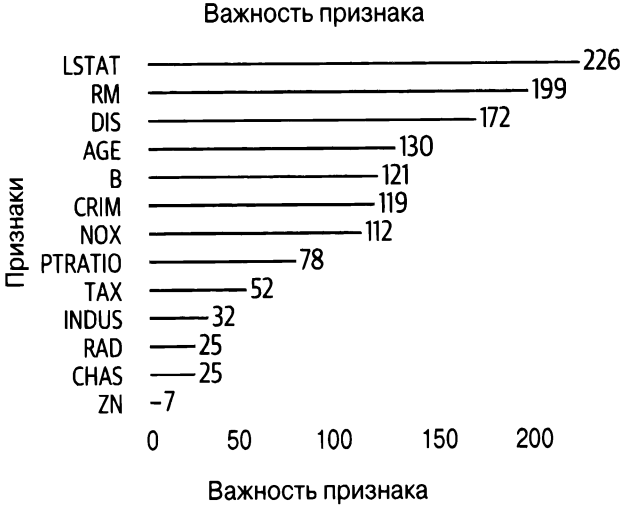


Рис. 14.8. *Важность признаков, показывающая, сколько раз был использован признак*

СОВЕТ

В Jupyter для просмотра дерева используйте следующую команду:

```
lgb.create_tree_digraph(lgbr)
```

Метрики и регрессионная оценка

В этой главе будут оцениваться результаты регрессора случайного леса, обученного на данных о жилье в Бостоне:

```
>>> rfr = RandomForestRegressor(
...     random_state=42, n_estimators=100
... )
>>> rfr.fit(bos_X_train, bos_y_train)
```

Метрики

Метрики для оценки регрессионных моделей содержат модуль `sklearn.metrics`. Метрические функции заканчивают работу при минимуме `loss` или `error` (потерь или ошибок). Функции могут заканчивать работу на максимуме `score` (оценка).

Общей метрикой регрессии является *коэффициент детерминации* (coefficient of determination) (r^2). Обычно это значение находится в диапазоне от 0 до 1. Оно представляет процент дисперсии цели, которую вносят признаки. Чем выше значения, тем лучше, но в целом сложно оценить модель только по этой метрике. Означает ли значение 0,7 хороший результат? В зависимости от обстоятельств. Для одного набора данных хорошим показателем может быть 0,5, а для другого набора данных и 0,9 может быть плохим. Обычно мы используем это

число для оценки модели в сочетании с другими показателями или визуализациями.

Например, легко создать модель, которая прогнозирует цены акций на следующий день с r^2, равным 0,99. Но я бы не обменял свои деньги на эту модель. Она может оказаться немного ниже или выше, что может нанести ущерб финансам.

Метрика r^2 является стандартной метрикой, используемой во время сеточного поиска. Используя параметр `scoring`, вы можете указать другие метрики.

Для регрессионных моделей его рассчитывает метод `.score`:

```
>>> from sklearn import metrics
>>> rfr.score(bos_X_test, bos_y_test)
0.8721182042634867

>>> metrics.r2_score(bos_y_test, bos_y_test_pred)
0.8721182042634867
```

НА ЗАМЕТКУ

Существует также метрика *объяснимой дисперсии* (explained variance) (`'explained_variance'` в сеточном поиске). Если среднее значение *остатков* (residual) (ошибок в прогнозах) равно 0 (в моделях *обычных наименьших квадратов* (Ordinary Least Squares — OLS)), то объяснимая дисперсия совпадает с коэффициентом детерминации:

```
>>> metrics.explained_variance_score(
... bos_y_test, bos_y_test_pred
... )
0.8724890451227875
```

Средняя абсолютная ошибка (mean absolute error) (`'neg_mean_absolute_error'` при использовании в сеточном поиске) выражает среднюю абсолютную ошибку прогнозирования модели. Идеальная модель получила бы оценку 0, но эта метрика не имеет верхних границ в отличие от коэффициента

детерминации. Но поскольку она выражена в единицах измерения цели, она более понятна. Если вы хотите игнорировать выбросы, это хороший показатель для использования.

Эта метрика не может указать, насколько модель плоха, но может использоваться для сравнения двух моделей. Если у вас есть две модели, модель с более низкой оценкой лучше.

Это число говорит нам, что средняя ошибка примерно вдвое выше или ниже реального значения:

```
>>> metrics.mean_absolute_error(
...     bos_y_test, bos_y_test_pred
... )
2.0839802631578945
```

Среднеквадратичная ошибка (root mean squared error) (`'neg_mean_squared_error'` в сеточном поиске) также измеряет ошибку модели в терминах цели. Но поскольку она усредняет квадрат ошибок до получения квадратного корня, она штрафует большие ошибки. Если вы хотите штрафовать большие ошибки, это хорошая метрика для использования. Например, если нечто хуже в восемь раз, то нечто, худшее в четыре раза, лучше.

Как и в случае средней абсолютной ошибки, эта метрика не может указывать, насколько плоха модель, но может использоваться для сравнения двух моделей. Если вы предполагаете, что ошибки распределяются нормально, это хороший выбор.

Результат говорит нам, что, если мы возведем ошибки в квадрат и усредним их, результат будет порядка 9,5:

```
>>> metrics.mean_squared_error(
...     bos_y_test, bos_y_test_pred
... )
9.52886846710526
```

Среднеквадратичная логарифмическая ошибка (mean squared logarithmic error) (в сеточном поиске — `'neg_mean_squared_log_error'`) штрафует недостаточный прогноз больше, чем чрезмерный. Если у вас есть цели с экспоненциальным ростом (население, запасы и т.д.), это хороший показатель.

Если вы берете логарифм ошибки, а затем возводите ее в квадрат, среднее значение этих результатов будет 0,021:

```
>>> metrics.mean_squared_log_error(
...     bos_y_test, bos_y_test_pred
... )
0.02128263061776433
```

График остатков

Хорошие модели (с соответствующими оценками R2) будут демонстрировать *гомоскедастичность* (homoscedasticity). Это означает, что дисперсия одинакова для всех значений целей независимо от входных данных. На графике остатков это выглядит как случайно распределенные значения. Если есть шаблоны, модель или данные являются проблематичными.

Графики остатков показывают также выбросы, способные оказать большое влияние на подгонку модели (рис. 15.1).

Библиотека Yellowbrick способна визуализировать графики остатков:

```
>>> from yellowbrick.regressor import ResidualsPlot
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> rpv = ResidualsPlot(rfr)
>>> rpv.fit(bos_X_train, bos_y_train)
>>> rpv.score(bos_X_test, bos_y_test)
>>> rpv.poof()
>>> fig.savefig("images/mlpr_1501.png", dpi=300)
```

Гетероскедастичность

Библиотека statsmodel включает *тест Бройша–Пагана* (Breusch–Pagan test) на гетероскедастичность. Это означает, что дисперсия остатков варьируется в зависимости от прогнозируемых значений. В тесте Бройша–Пагана, если *p*-значения значимы (`p-value` меньше 0,05), нулевая гипотеза гомоскеда-

стичности отвергается. Это указывает на то, что остатки гетероскедастичны, а прогнозы смещены.

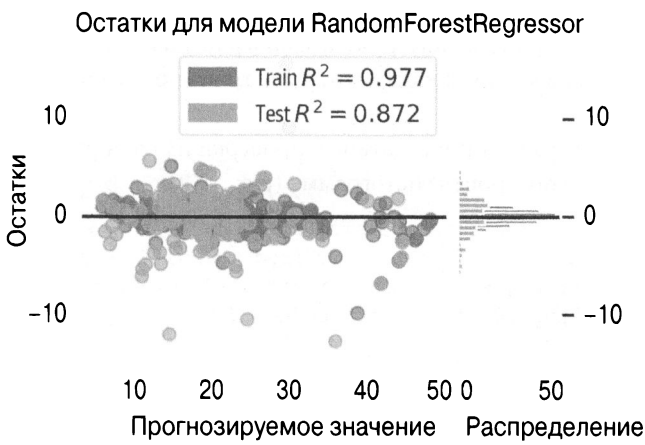


Рис. 15.1. *График остатков. Дальнейшее тестирование покажет, что они гетероскедастичны*

Тест подтверждает гетероскедастичность:

```
>>> import statsmodels.stats.api as sms
>>> hb = sms.het_breuschpagan(resids, bos_X_test)
>>> labels = [
...     "Lagrange multiplier statistic",
...     "p-value",
...     "f-value",
...     "f p-value",
... ]
>>> for name, num in zip(name, hb):
...     print(f"{name}: {num:.2}")

Lagrange multiplier statistic: 3.6e+01
p-value: 0.00036
f-value: 3.3
f p-value: 0.00022
```

Нормальные остатки

Библиотека scipy включает в себя *график распределения вероятностей* (probability plot) и *критерий Колмогорова–Смирнова*, оба из которых измеряют, являются ли остатки нормальными.

Для визуализации остатков и проверки их на нормальность мы можем построить гистограмму (рис. 15.2):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> resids = bos_y_test - rfr.predict(bos_X_test)
>>> pd.Series(resids, name="residuals").plot.hist(
...     bins=20, ax=ax, title="Residual Histogram"
... )
>>> fig.savefig("images/mlpr_1502.png", dpi=300)
```

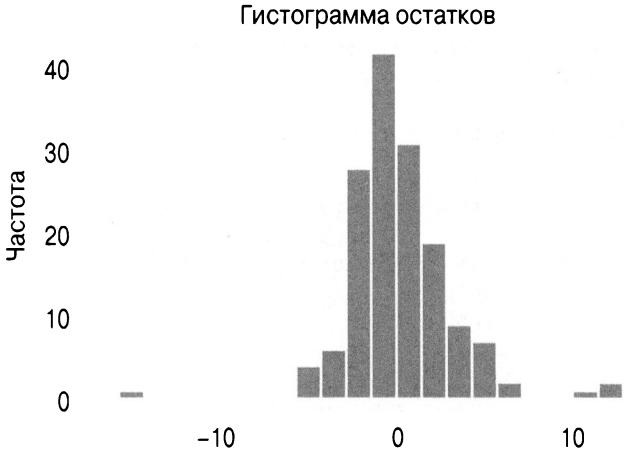


Рис. 15.2. *Гистограмма остатков*

На рис. 15.3 демонстрируется график вероятности. Если выборки, представленные по отношению к квантилям, выстраиваются в линию, остатки нормальные. Мы можем видеть, что в данном случае это не так:

```
>>> from scipy import stats
>>> fig, ax = plt.subplots(figsize=(6, 4))
```

```
>>> _ = stats.probplot(resids, plot=ax)
>>> fig.savefig("images/mlpr_1503.png", dpi=300)
```

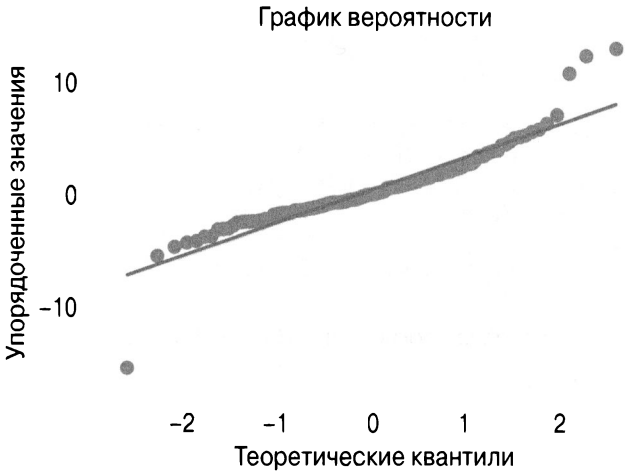


Рис. 15.3. *График вероятности остатков*

Критерий Колмогорова–Смирнова позволяет оценить, является ли распределение нормальным. Если p-значение является значимым (<0,05), то значения не являются нормальными.

В данном случае это не так, что говорит нам, что остатки не являются нормальными:

```
>>> stats.kstest(resids, cdf="norm")
KstestResult(statistic=0.1962230021010155,
pvalue=1.3283596864921421e-05)
```

График ошибки прогноза

График ошибки прогноза позволяет сравнивать реальные цели с прогнозируемыми значениями. У идеальной модели эти точки выстраиваются в линию под углом 45 градусов.

Поскольку наша модель, по-видимому, прогнозирует более низкие значения для верхнего предела y, модель имеет

некоторые проблемы с производительностью. Это также очевидно на графике остатков (рис. 15.4).

Вот версия Yellowbrick:

```
>>> from yellowbrick.regressor import (
...     PredictionError,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pev = PredictionError(rfr)
>>> pev.fit(bos_X_train, bos_y_train)
>>> pev.score(bos_X_test, bos_y_test)
>>> pev.poof()
>>> fig.savefig("images/mlpr_1504.png", dpi=300)
```

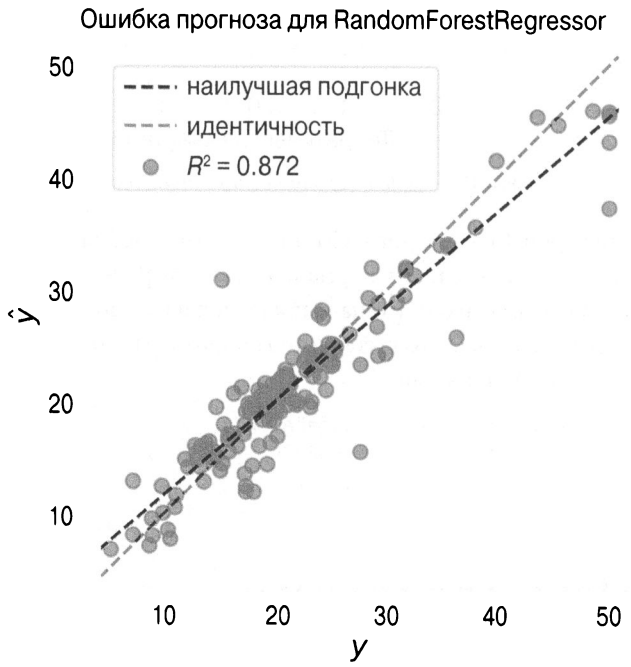


Рис. 15.4. *Ошибка прогноза. Графики прогнозируемого y по сравнению с фактическим y*

Объяснение регрессионных моделей

Большинство методов, используемых для объяснения классификационных моделей, применимы к регрессионным моделям. В этой главе я покажу, как использовать библиотеку SHAP для интерпретации регрессионных моделей.

Мы интерпретируем модель XGBoost для набора данных Boston:

```
>>> import xgboost as xgb
>>> xgr = xgb.XGBRegressor(
...     random_state=42, base_score=0.5
... )
>>> xgr.fit(bos_X_train, bos_y_train)
```

Shapley

Я — большой поклонник пакета Shapley, поскольку он не зависит от модели. Эта библиотека дает также глобальное понимание нашей модели и помогает объяснить отдельные прогнозы. Я считаю модель черного ящика очень полезной.

Сначала рассмотрим прогноз для индекса 5. Наша модель прогнозирует значение 27,26:

```
>>> sample_idx = 5
>>> xgr.predict(bos_X.iloc[[sample_idx]])
array([27.269186], dtype=float32)
```

Чтобы использовать модель, необходимо создать `TreeExplainer` из нашей модели и оценить значения SHAP для наших выборок. Если мы хотим использовать Jupyter и иметь интерактивный интерфейс, нам также нужно вызвать функцию `initjs`:

```
>>> import shap
>>> shap.initjs()

>>> exp = shap.TreeExplainer(xgr)
>>> vals = exp.shap_values(bos_X)
```

Используя *блок объяснения* (explainer) и значения SHAP, мы можем создать график силы, чтобы объяснить прогноз (рис. 16.1). Он сообщает нам, что базовый прогноз равен 23 и что статус населения (LSTAT) и ставка налога на имущество (TAX) толкают цену вверх, а количество комнат (RM) — вниз:

```
>>> shap.force_plot(
...     exp.expected_value,
...     vals[sample_idx],
...     bos_X.iloc[sample_idx],
... )
```

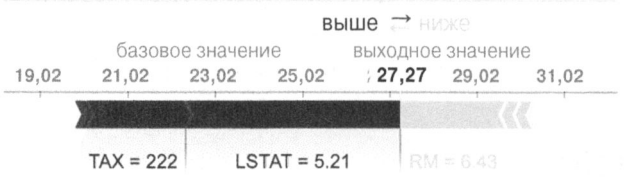


Рис. 16.1. *График силы для регрессии. Ожидаемое значение увеличено с 23 до 27 из-за статуса населения и налоговой ставки*

Чтобы получить общее представление о поведении, мы также можем просмотреть график силы для всех выборок. Если мы используем интерактивный режим JavaScript на Jupyter, мы можем навести указатель мыши на выборки и посмотреть, какие признаки влияют на результат (рис. 16.2):

```
>>> shap.force_plot(
...     exp.expected_value, vals, bos_X
... )
```

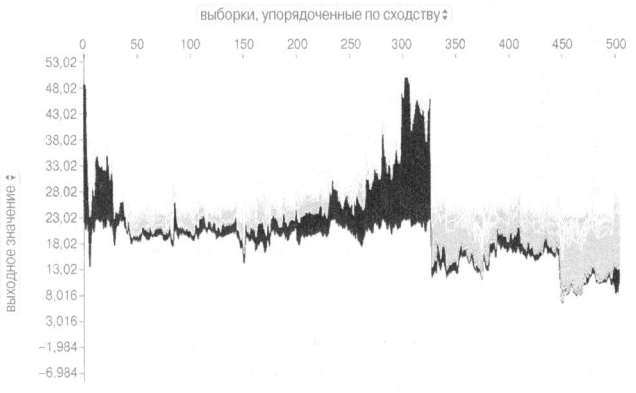


Рис. 16.2. *График силы для регрессии по всем выборкам*

Из графика силы по всем выборкам мы видим, что признак LSTAT оказал наибольшее влияние. Чтобы визуализировать, как LSTAT влияет на результат, мы можем создать график зависимости. Библиотека автоматически выберет признак для окрашивания (чтобы установить свой признак, можете указать параметр `interaction_index`).

На графике зависимости для LSTAT (рис. 16.3) можно видеть, что при увеличении LSTAT (процент населения с низким статусом) значение SHAP уменьшается (опускаясь до цели). Очень низкое значение LSTAT увеличивает SHAP. Из просмотра окраски TAX (ставка налога на имущество) видно, что по мере снижения ставки (более синего цвета) значение SHAP увеличивается:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.dependence_plot("LSTAT", vals, bos_X)
>>> fig.savefig(
...     "images/mlpr_1603.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

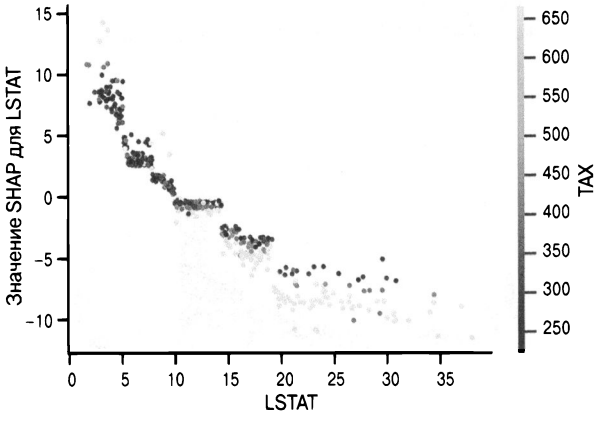


Рис. 16.3. *График зависимости для LSTAT. По мере увеличения LSTAT прогнозируемое значение уменьшается*

Вот еще один график зависимости (рис. 16.4) для изучения DIS (расстояние до центров занятости). Похоже, что этот признак малоэффективен, если он не слишком мал:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.dependence_plot(
...     "DIS", vals, bos_X, interaction_index="RM"
... )
>>> fig.savefig(
...     "images/mlpr_1604.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

Наконец, мы рассмотрим глобальный эффект признака, используя сводный график (рис. 16.5). Признаки в верхней части имеют наибольшее влияние на модель. Из этого представления вы можете видеть, что большие значения RM (количество комнат) сильно увеличивают цель, в то время как средние и меньшие значения немного ее понижают:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.summary_plot(vals, bos_X)
>>> fig.savefig(
```

```
...     "images/mlpr_1605.png",
...     bbox_inches="tight",
...     dpi=300,
... )
```

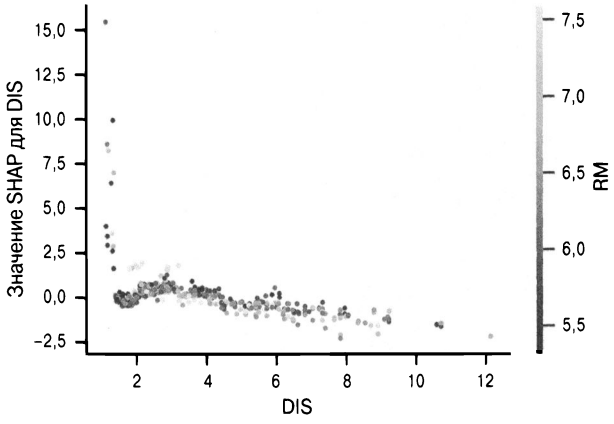


Рис. 16.4. *График зависимости для DIS. Если DIS не очень мало, график SHAP остается относительно плоским*

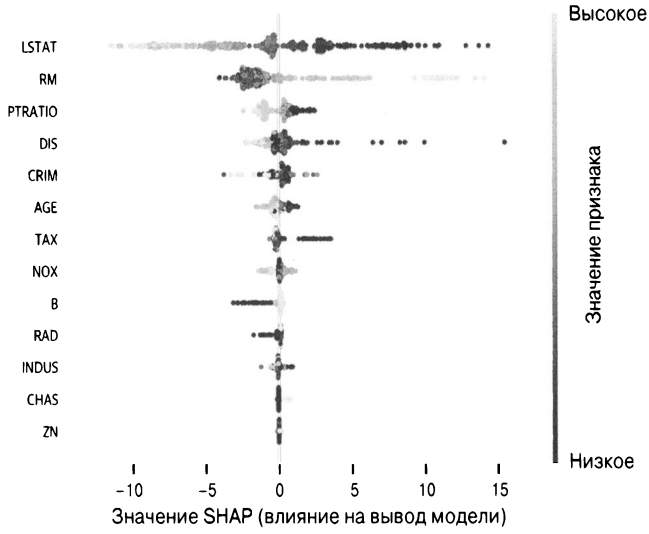


Рис. 16.5. *Сводный график. Самые важные признаки находятся вверху*

Библиотека SHAP — отличный инструмент в вашем наборе инструментов. Он помогает понять глобальное влияние признака, а также объяснить индивидуальные прогнозы.

ГЛАВА 17

Уменьшение размерности

Существует множество методов для разложения признаков на более мелкие подмножества. Это может быть полезно для разведочного анализа данных, визуализации, создания прогнозирующих моделей или кластеризации.

В этой главе мы рассмотрим набор данных Titanic, используя различные методы. Мы рассмотрим PCA, UMAP, t-SNE и PHATE.

Вот данные:

```
>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test =
get_train_test_X_y(
...     ti_df, "survived", std_cols=std_cols
... )
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
```

PCA

Анализ основных компонентов (Principal Component Analysis — PCA) использует матрицу (X) строк (выборок) и столбцов (признаков). PCA возвращает новую матрицу, в которой столбцы представляют собой линейные комбинации исходных столбцов. Эти линейные комбинации максимизируют дисперсию.

Каждый столбец ортогонален (под прямым углом) к другим столбцам. Столбцы отсортированы в порядке убывания дисперсии.

Библиотека Scikit-learn имеет реализацию этой модели. Лучше стандартизировать данные до запуска алгоритма. После вызова метода `.fit` у вас будет доступ к атрибуту `.explained_variance_ratio_`, в котором указан процент отклонения в каждом столбце.

PCA полезен для визуализации данных в двух (или трех) измерениях. Он также используется в качестве этапа предварительной обработки для фильтрации случайных шумов в данных. Это хорошо работает с линейными данными и годится для поиска глобальных структур, но не локальных.

В этом примере мы собираемся запустить PCA на признаках набора Titanic. Класс PCA в библиотеке Scikit-learn является *трансформером* (transformer); вы вызываете метод `.fit`, чтобы обучить его получению основных компонентов, а затем вызываете метод `.transform`, чтобы преобразовать матрицу в матрицу основных компонентов:

```
>>> from sklearn.decomposition import PCA
>>> from sklearn.preprocessing import (
...     StandardScaler,
... )
>>> pca = PCA(random_state=42)
>>> X_pca = pca.fit_transform(
...     StandardScaler().fit_transform(X)
... )
>>> pca.explained_variance_ratio_
array([0.23917891, 0.21623078, 0.19265028,
       0.10460882, 0.08170342, 0.07229959,
       0.05133752, 0.04199068])

>>> pca.components_[0]
arrayarray([-0.63368693, 0.39682566,
             0.00614498, 0.11488415, 0.58075352,
            -0.19046812, -0.21190808, -0.09631388])
```

Параметры экземпляра

`n_components=None`

Количество компонентов для генерации. Если `None`, вернуть то же число, что и количество столбцов. Может быть числом с плавающей запятой в диапазоне (0, 1) и создать столько компонентов, сколько необходимо для получения этого коэффициента дисперсии.

`copy=True`

Изменит данные в `.fit`, если `True`.

`whiten=False`

Превратит данные в белый шум после преобразования, чтобы избавиться от корреляции компонентов.

`svd_solver='auto'`

`'auto'` запускает `'randomized'` SVD, если `n_components` составляет менее 80% от наименьшего измерения (быстро, но приблизительно). В противном случае работает `'full'`.

`tol=0.0`

Толерантность для сингулярных значений.

`iterated_power='auto'`

Количество итераций для `'randomized'` `svd_solver`.

`random_state=None`

Случайное состояние для `'randomized'` `svd_solver`.

Атрибуты

`components_`

Основные компоненты (столбцы линейных комбинационных весов для оригинальных признаков).

`explained_variance_`

Величина дисперсии для каждого компонента.

`explained_variance_ratio_`

Нормализованная величина дисперсии для каждого компонента (сумма до 1).

`singular_values_`

Сингулярные значения для каждого компонента.

`mean_`

Среднее значение каждого признака.

`n_components_`

Размер компонентов, когда `n_components` — это число с плавающей запятой.

`noise_variance_`

Ожидаемая ковариация шума.

График кумулятивной суммы объясненного коэффициента дисперсии называется *графиком собственных значений* (scree plot) (рис. 17.1). Он показывает, сколько информации хранится в компонентах. Вы можете использовать *метод локтей* (elbow method), чтобы увидеть, изгибается ли график, и чтобы определить, сколько компонентов использовать:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> ax.plot(pca.explained_variance_ratio_)
>>> ax.set(
...     xlabel="Component",
...     ylabel="Percent of Explained variance",
...     title="Scree Plot",
...     ylim=(0, 1),
... )
>>> fig.savefig(
...     "images/mlpr_1701.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Другой способ просмотреть эти данные — использовать *кумулятивный график* (cumulative plot) (рис. 17.2). Наши исходные данные имели 8 столбцов, но из графика видно, что мы сохраняем около 90% дисперсии, если используем только 4 компонента PCA:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> ax.plot(
...     np.cumsum(pca.explained_variance_ratio_)
```

```
... )
>>> ax.set(
...     xlabel="Component",
...     ylabel="Percent of Explained variance",
...     title="Cumulative Variance",
...     ylim=(0, 1),
... )
>>> fig.savefig("images/mlpr_1702.png", dpi=300)
```

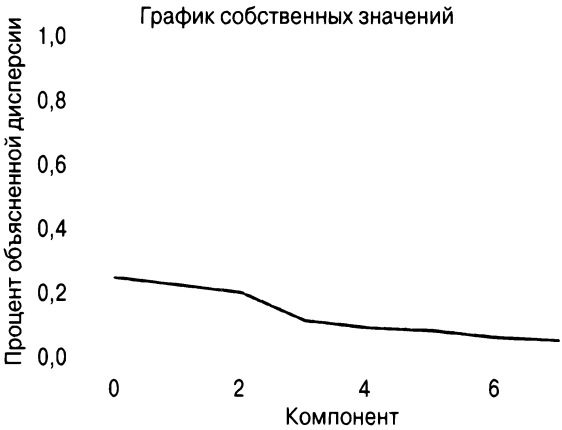


Рис. 17.1. *График PCA собственных значений*

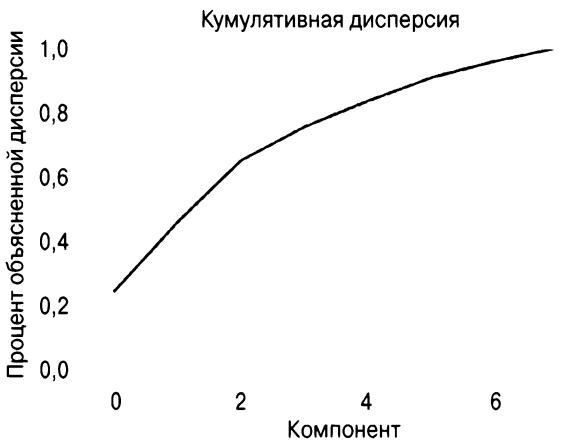


Рис. 17.2. *Кумулятивная объясненная дисперсия PCA*

Насколько признаки влияют на компоненты? Используйте функцию matplotlib `imshow`, чтобы нанести компоненты вдоль оси x и исходные признаки вдоль оси y (рис. 17.3). Чем темнее цвет, тем больше вклад исходного столбца в компонент.

Похоже, что на первый компонент сильно влияют столбцы pclass, age и fare. (Использование спектральной цветовой карты (`cmap`) подчеркивает ненулевые значения, а предоставление `vmin` и `vmax` добавляет ограничения к цветовой легенде.)

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> plt.imshow(
...     pca.components_.T,
...     cmap="Spectral",
...     vmin=-1,
...     vmax=1,
... )
>>> plt.yticks(range(len(X.columns)), X.columns)
>>> plt.xticks(range(8), range(1, 9))
>>> plt.xlabel("Principal Component")
>>> plt.ylabel("Contribution")
>>> plt.title(
...     "Contribution of Features to Components"
... )
>>> plt.colorbar()
>>> fig.savefig("images/mlpr_1703.png", dpi=300)
```

Альтернатива — взглянуть на гистограмму (рис. 17.4). Каждый компонент показан с вкладами из исходных данных:

```
>>> fig, ax = plt.subplots(figsize=(8, 4))
>>> pd.DataFrame(
...     pca.components_, columns=X.columns
... ).plot(kind="bar", ax=ax).legend(
...     bbox_to_anchor=(1, 1)
... )
>>> fig.savefig("images/mlpr_1704.png", dpi=300)
```

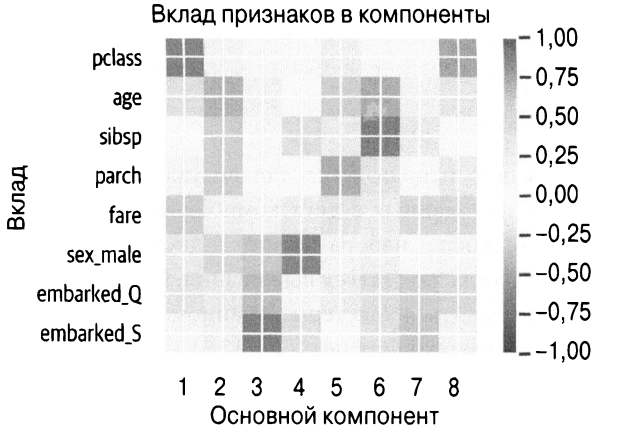


Рис. 17.3. *Признаки в компонентах PCA*

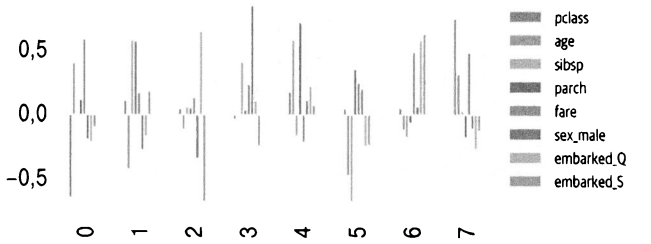


Рис. 17.4. *Признаки в компонентах PCA*

Если у нас много признаков, мы можем захотеть ограничить графики выше, показывая только те признаки, которые соответствуют минимальному весу. Вот код, чтобы найти все признаки в первых двух компонентах, которые имеют абсолютные значения по крайней мере 0,5:

```
>>> comps = pd.DataFrame(
...     pca.components_, columns=X.columns
... )
>>> min_val = 0.5
>>> num_components = 2
>>> pca_cols = set()
>>> for i in range(num_components):
...     parts = comps.iloc[i][
```

```
...             comps.iloc[i].abs() > min_val
...         ]
...         pca_cols.update(set(parts.index))
>>> pca_cols
{'fare', 'parch', 'pclass', 'sibsp'}
```

PCA обычно используется для визуализации высокоразмерных наборов данных в двух компонентах. Здесь мы визуализируем признаки набора features в двух измерениях. Они раскрашены согласно статусу выживания. Иногда в визуализации могут появляться кластеры. В данном случае, похоже, что кластеризации выживших нет (рис. 17.5).

Мы создаем эту визуализацию, используя Yellowbrick:

```
>>> from yellowbrick.features.pca import (
...     PCADecomposition,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> colors = ["rg"[j] for j in y]
>>> pca_viz = PCADecomposition(color=colors)
>>> pca_viz.fit_transform(X, y)
>>> pca_viz.poof()
>>> fig.savefig("images/mlpr_1705.png", dpi=300)
```

Рис. 17.5. *График PCA Yellowbrick*

Если вы хотите окрасить график рассеяния цветом столбца и добавить легенду (а не цветовую шкалу), вам нужно зациклить каждый цвет и нанести на график эту группу индивидуально в pandas или matplotlib (или использовать seaborn). Ниже мы устанавливаем также соотношение сторон согласно соотношению объясненных отклонений для рассматриваемых компонентов (рис. 17.6). Поскольку второй компонент имеет только 90% первого компонента, он немного короче.

Вот версия seaborn:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
...     X_pca,
...     columns=[
...         f"PC{i+1}"
...         for i in range(X_pca.shape[1])
...     ],
... )
>>> pca_df["status"] = [
...     ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> ax.set_aspect(evr[1] / evr[0])
>>> sns.scatterplot(
...     x="PC1",
...     y="PC2",
...     hue="status",
...     data=pca_df,
...     alpha=0.5,
...     ax=ax,
... )
>>> fig.savefig(
...     "images/mlpr_1706.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

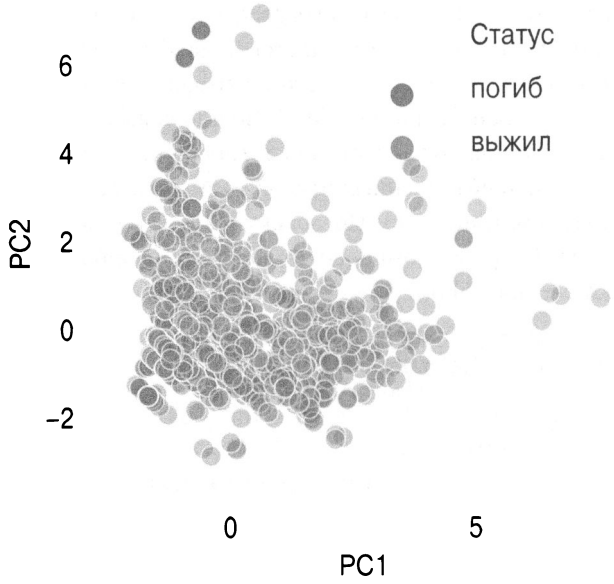


Рис. 17.6. *График PCA seaborn с легендой и относительным соотношением*

Ниже мы увеличиваем график рассеяния, показывая поверх него *график нагрузки* (loading plot). Этот график называют также двойным, поскольку он объединяет график рассеяния и нагрузки (рис. 17.7). График нагрузки показывает, насколько сильны признаки и как они соотносятся. Если их углы близки, они, вероятно, коррелируют. Если углы под 90 градусов, они, вероятно, не коррелируют. Наконец, если угол между ними близок к 180 градусам, они имеют отрицательную корреляцию:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
...     X_pca,
...     columns=[
...         f"PC{i+1}"
...         for i in range(X_pca.shape[1])
...     ],
... )
>>> pca_df["status"] = [
```

```
...     ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> x_idx = 0  # x_pc
>>> y_idx = 1  # y_pc
>>> ax.set_aspect(evr[y_idx] / evr[x_idx])
>>> x_col = pca_df.columns[x_idx]
>>> y_col = pca_df.columns[y_idx]
>>> sns.scatterplot(
...     x=x_col,
...     y=y_col,
...     hue="status",
...     data=pca_df,
...     alpha=0.5,
...     ax=ax,
... )
>>> scale = 8
>>> comps = pd.DataFrame(
...     pca.components_, columns=X.columns
... )
>>> for idx, s in comps.T.iterrows():
...     plt.arrow(
...         0,
...         0,
...         s[x_idx] * scale,
...         s[y_idx] * scale,
...         color="k",
...     )
...     plt.text(
...         s[x_idx] * scale,
...         s[y_idx] * scale,
...         idx,
...         weight="bold",
...     )
>>> fig.savefig(
...     "images/mlpr_1707.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

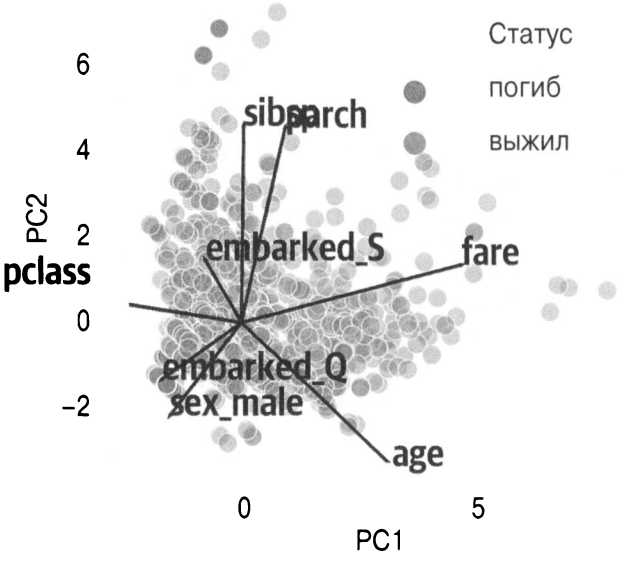


Рис. 17.7. *Двойной график seaborn с графиками рассеяния и нагрузки*

Из предыдущих древовидных моделей мы знаем, что для определения выживания пассажира важны возраст, тариф и пол. Первый основной компонент зависит от класса, возраста и тарифа, а четвертый — от пола. Давайте рассмотрим эти компоненты относительно друг друга.

Опять же, этот график масштабирует соотношение сторон, основываясь на коэффициентах дисперсии компонентов (рис. 17.8).

Этот график, кажется, более точно разделяет выживших:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
...     X_pca,
...     columns=[
...         f"PC{i+1}"
...         for i in range(X_pca.shape[1])
...     ],
... )
>>> pca_df["status"] = [
```

```
...     ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> ax.set_aspect(evr[3] / evr[0])
>>> sns.scatterplot(
...     x="PC1",
...     y="PC4",
...     hue="status",
...     data=pca_df,
...     alpha=0.5,
...     ax=ax,
... )
>>> fig.savefig(
...     "images/mlpr_1708.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

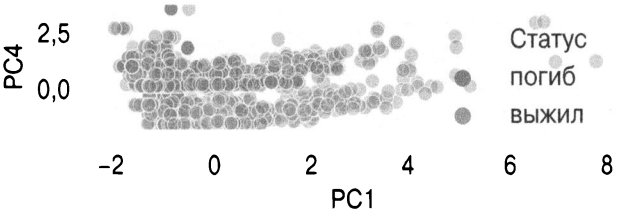


Рис. 17.8. *График PCA, показывающий соотношение компонентов 1 и 4*

Библиотека matplotlib может создавать симпатичные графики, но она менее полезна для интерактивных графиков. При выполнении PCA обычно полезно просматривать данные графиков рассеяния. Я включил функцию, которая использует библиотеку Bokeh для взаимодействия с графиками рассеяния (рис. 17.9). Это хорошо работает в Jupyter:

```
>>> from bokeh.io import output_notebook
>>> from bokeh import models, palettes, transform
>>> from bokeh.plotting import figure, show
>>>
>>> def bokeh_scatter(
...     x,
...     y,
```

```
...     data,
...     hue=None,
...     label_cols=None,
...     size=None,
...     legend=None,
...     alpha=0.5,
... ):
...     """
...     x - x column name to plot
...     y - y column name to plot
...     data - pandas DataFrame
...     hue - column name to color by (numeric)
...     legend - column name to label by
...     label_cols - columns to use in tooltip
...                  (None all in DataFrame)
...     size - size of points in screen space unigs
...     alpha - transparency
...     """
...     output_notebook()
...     circle_kwargs = {}
...     if legend:
...         circle_kwargs["legend"] = legend
...     if size:
...         circle_kwargs["size"] = size
...     if hue:
...        color_seq = data[hue]
...        mapper = models.LinearColorMapper(
...            palette=palettes.viridis(256),
...            low=min(color_seq),
...            high=max(color_seq),
...        )
...        circle_kwargs[
...            "fill_color"
...        ] = transform.transform(hue, mapper)
...     ds = models.ColumnDataSource(data)
...     if label_cols is None:
...       label_cols = data.columns
...     tool_tips = sorted(
...         [
...             (x, "@{}".format(x))
...             for x in label_cols
...         ],
```

```
...             key=lambda tup: tup[0],
...         )
...         hover = models.HoverTool(
...             tooltips=tool_tips
...         )
...         fig = figure(
...             tools=[
...                 hover,
...                 "pan",
...                 "zoom_in",
...                 "zoom_out",
...                 "reset",
...             ],
...             toolbar_location="below",
...         )
...
...         fig.circle(
...             x,
...             y,
...             source=ds,
...             alpha=alpha,
...             **circle_kwargs
...         )
...         show(fig)
...         return fig
>>> res = bokeh_scatter(
...         "PC1",
...         "PC2",
...         data=pca_df.assign(
...             surv=y.reset_index(drop=True)
...         ),
...         hue="surv",
...         size=10,
...         legend="surv",
... )
```

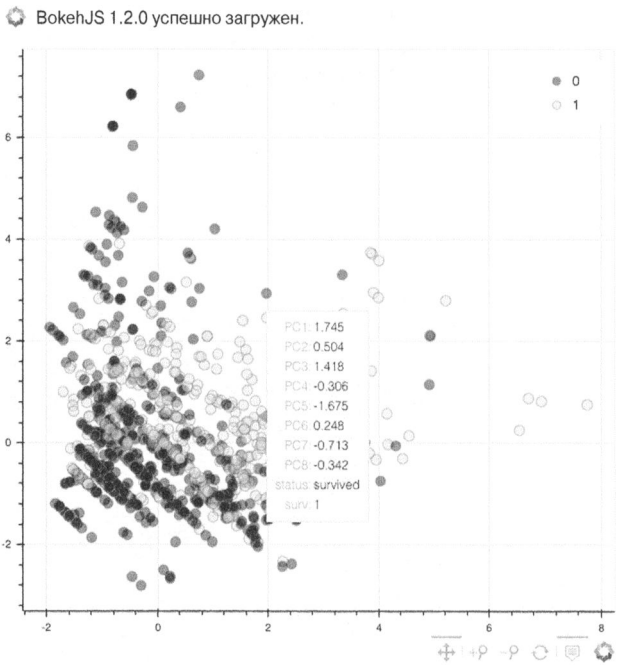


Рис. 17.9. *График рассеяния Bokeh со всплывающими подсказками*

Библиотека Yellowbrick может также строить график в трех измерениях (рис. 17.10):

```
>>> from yellowbrick.features.pca import (
...     PCADecomposition,
... )
>>> colors = ["rg"[j] for j in y]
>>> pca3_viz = PCADecomposition(
...     proj_dim=3, color=colors
... )
>>> pca3_viz.fit_transform(X, y)
>>> pca3_viz.finalize()
>>> fig = plt.gcf()
>>> plt.tight_layout()
>>> fig.savefig(
...     "images/mlpr_1710.png",
...     dpi=300,
```

```
...     bbox_inches="tight",
... )
```

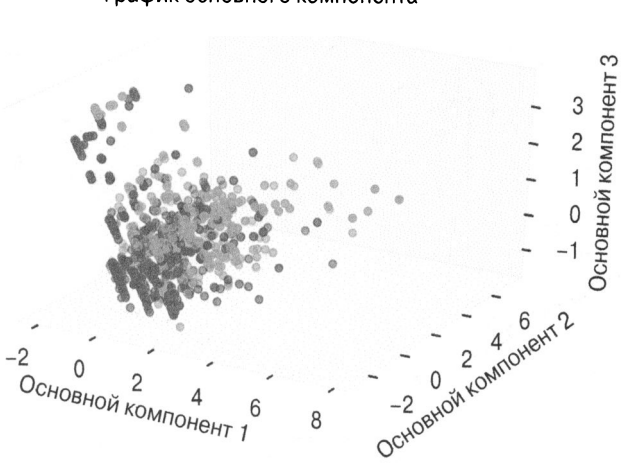


Рис. 17.10. *Трехмерный график Yellowbrick PCA*

Библиотека scprep (зависящая от библиотеки PHATE, о которой мы вскоре поговорим) имеет полезную функцию построения графиков. Функция `rotate_scatter3d` может генерировать график, который будет анимироваться в Jupyter (рис. 17.11). Это облегчает понимание трехмерных графиков.

Вы можете использовать эту библиотеку для визуализации любых трехмерных данных, а не только PHATE:

```
>>> import scprep
>>> scprep.plot.rotate_scatter3d(
...     X_pca[:, :3],
...     c=y,
...     cmap="Spectral",
...     figsize=(8, 6),
...     label_prefix="Principal Component",
... )
```

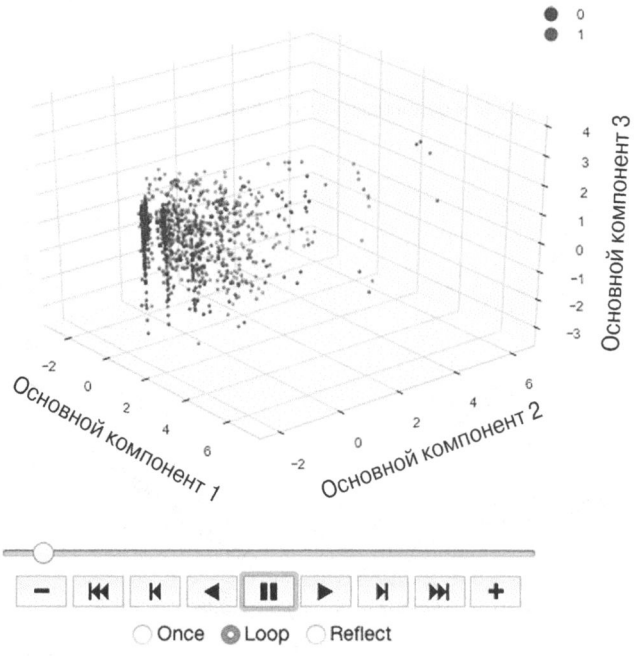


Рис. 17.11. *3D-анимация scprep PCA*

Если в Jupyter вы измените магический режим ячейки matplotlib на `notebook`, вы можете получить интерактивный трехмерный график из matplotlib (рис. 17.12):

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure(figsize=(6, 4))
>>> ax = fig.add_subplot(111, projection="3d")
>>> ax.scatter(
...     xs=X_pca[:, 0],
...     ys=X_pca[:, 1],
...     zs=X_pca[:, 2],
...     c=y,
...     cmap="viridis",
... )
>>> ax.set_xlabel("PC 1")
>>> ax.set_ylabel("PC 2")
>>> ax.set_zlabel("PC 3")
```

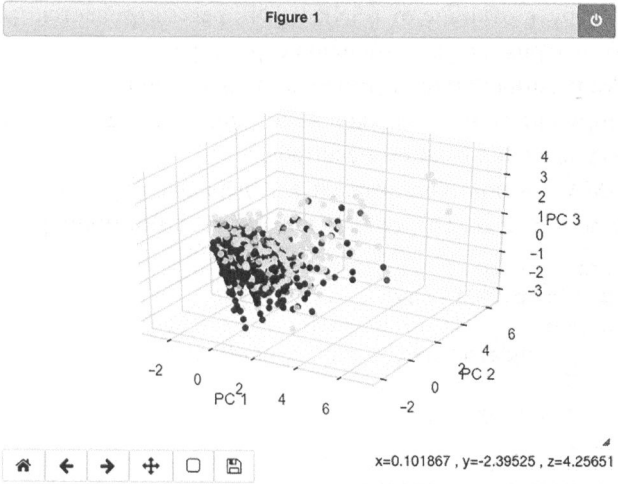


Рис. 17.12. *Интерактивный трехмерный график matplotlib с режимом* `notebook` *PCA*

ВНИМАНИЕ

Обратите внимание, что переключение в Jupyter магии ячеек для matplotlib с

```
% matplotlib inline
```

на

```
% matplotlib notebook
```

может иногда заставить Jupyter перестать отвечать. Будьте осторожны!

UMAP

UMAP (Uniform Manifold Approximation and Projection) — это метод уменьшения размерности, использующий *обучение многообразием* (manifold learning). Как таковой он имеет тенденцию объединять подобные элементы топологически. Он пытается сохранить как их глобальную, так и локальную

структуру, в отличие от t-SNE (t-SNE рассматривается далее), что благоприятствует локальной структуре.

Реализация Python не имеет многоядерной поддержки.

Нормализация признаков — это хорошая идея, чтобы получить значения в одном масштабе.

UMAP очень чувствителен к гиперпараметрам (`n_neighbors`, `min_dist`, `n_components` или `metric`). Вот пример:

```
>>> import umap
>>> u = umap.UMAP(random_state=42)
>>> X_umap = u.fit_transform(
...     StandardScaler().fit_transform(X)
... )
>>> X_umap.shape
(1309, 2)
```

Параметры экземпляра

`n_neighbors=15`

Размер окрестности. Большие значения означают использование глобального представления, меньшие — более локального.

`n_components=2`

Количество размерностей для вложения.

`metric='euclidean'`

Используемая метрика расстояния. Может быть функцией, которая получает два одномерных массива и возвращает число с плавающей запятой.

`n_epochs=None`

Количество эпох обучения. Стандартно — будет 200 или 500 (в зависимости от размера данных).

`learning_rate=1.0`

Скорость обучения для встроенной оптимизации.

`init='spectral'`

Тип инициализации. Стандартно используется спектральное вложение. Может быть `'random'` или массивом numpy локаций.

`min_dist=0.1`

От 0 до 1. Минимальное расстояние между вложенными точками. "Меньше" означает большую плотность, "больше" — разреженность.

`spread=1.0`

Определяет расстояние вложенных точек.

`set_op_mix_ratio=1.0`

От 0 до 1: нечеткое объединение (1) или нечеткое пересечение (0).

`local_connectivity=1.0`

Количество соседей для локального подключения. По мере того как это происходит, создается больше локальных соединений.

`repulsion_strength=1.0`

Сила отталкивания. Более высокие значения дают больший вес негативным выборкам.

`negative_sample_rate=5`

Соотношение негативных и позитивных выборок. Более высокое значение дает большее отталкивание, больше затрат на оптимизацию и лучшую точность.

`transform_queue_size=4.0`

Агрессивность при поиске ближайших соседей. Чем выше значение, тем ниже производительность, но выше точность.

`a=None`

Параметр для контроля вложения. Если равно `None`, UMAP определяет их из `min_dist` и `spread`.

`b=None`

Параметр для контроля вложения. Если равно `None`, UMAP определяет их из `min_dist` и `spread`.

`random_state=None`

Случайное начальное число.

`metric_kwds=None`

Словарь метрик для дополнительных параметров, если признак используется для `metric`. Также возможна параметризация `minkowsi` (и другие метрики).

`angular_rp_forest=False`

Используйте угловую случайную проекцию.

`target_n_neighbors=-1`

Количество соседей установлено для простоты.

`target_metric='categorical'`

Для использования сокращения с учителем. Также может быть `'L1'` или `'L2'`, или функция, получающая в качестве входных данных два массива из `X` и возвращающая значение расстояния между ними.

`target_metric_kwds=None`

Словарь метрик для использования, если функция используется для `target_metric`.

`target_weight=0.5`

Весовой коэффициент. От 0,0 до 1,0, где 0 означает "только на основе данных", а 1 — "только на основе цели".

`transform_seed=42`

Случайное начальное число для операций преобразования.

`verbose=False`

Многословность.

Атрибуты

`embedding_`

Результаты вложения

Давайте визуализируем результаты UMAP для набора данных Titanic стандартно (рис. 17.13):

```
>>> fig, ax = plt.subplots(figsize=(8, 4))
>>> pd.DataFrame(X_umap).plot(
...     kind="scatter",
...     x=0,
...     y=1,
...     ax=ax,
...     c=y,
...     alpha=0.2,
...     cmap="Spectral",
... )
>>> fig.savefig("images/mlpr_1713.png", dpi=300)
```

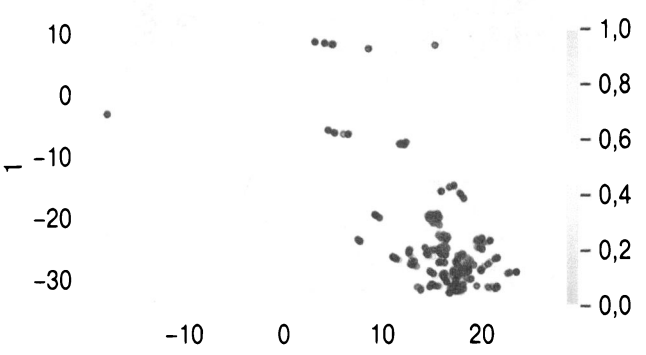


Рис. 17.13. *Результаты UMAP*

Чтобы скорректировать результаты UMAP, сначала сконцентрируйтесь на гиперпараметрах `n_neighbors` и `min_dist`. Вот иллюстрации изменения этих значений (рис. 17.14 и 17.15):

```
>>> X_std = StandardScaler().fit_transform(X)
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate([2, 5, 10, 50]):
...     ax = axes[i]
...     u = umap.UMAP(
...         random_state=42, n_neighbors=n
...     )
...     X_umap = u.fit_transform(X_std)
...
```

```
...     pd.DataFrame(X_umap).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"nn={n}")
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1714.png", dpi=300)
```

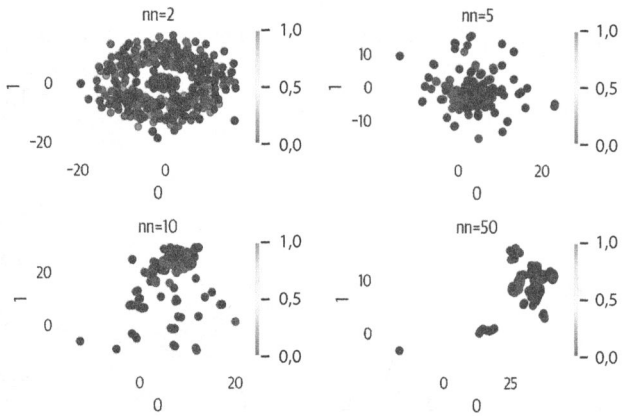


Рис. 17.14. *Результаты настройки* `n_neighbors` *UMAP*

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate([0, 0.33, 0.66, 0.99]):
...     ax = axes[i]
...     u = umap.UMAP(random_state=42, min_dist=n)
...     X_umap = u.fit_transform(X_std)
...     pd.DataFrame(X_umap).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
```

```
...     )
...     ax.set_title(f"min_dist={n}")
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1715.png", dpi=300)
```

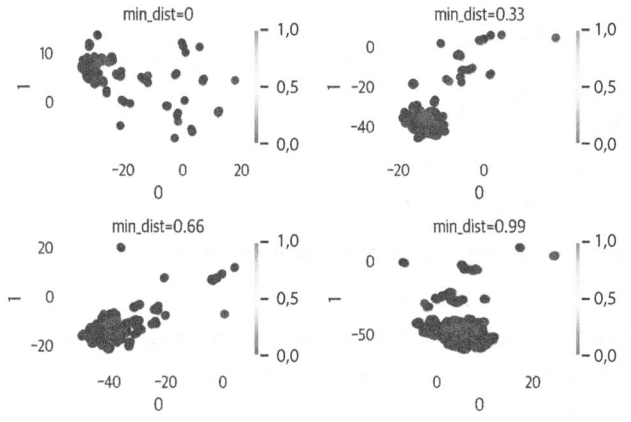


Рис. 17.15. *Корректировка результатов* `min_dist` *UMAP*

Иногда PCA выполняется перед UMAP, чтобы уменьшить размеры и ускорить вычисления.

t-SNE

Метод *стохастического вложения соседей с t-распределением* (t-SNE — t-Distributed Stochastic Neighboring Embedding) — это метод визуализации и уменьшения размерности. Он использует распределения входных данных и низкоразмерных вложений, а также минимизирует совместные вероятности между ними. Поскольку это требует значительных вычислительных ресурсов, вы не сможете использовать эту технику с большим набором данных.

Одной из характеристик t-SNE является то, что он довольно чувствителен к гиперпараметрам. Кроме того, хотя он хорошо сохраняет локальные кластеры, глобальная информация не сохраняется. Таким образом, расстояние между кластерами не

имеет смысла. Наконец, это не детерминированный алгоритм и может не сходиться.

Перед использованием этого метода имеет смысл стандартизировать данные:

```
>>> from sklearn.manifold import TSNE
>>> X_std = StandardScaler().fit_transform(X)
>>> ts = TSNE()
>>> X_tsne = ts.fit_transform(X_std)
```

Параметры экземпляра

`n_components=2`

Количество размерностей для вложения.

`perplexity=30.0`

Предлагаемые значения находятся в диапазоне от 5 до 50. Меньшие значения означают большую плотность.

`early_exaggeration=12.0`

Контролирует плотность кластеров и расстояние между ними. Большие значения означают больший интервал.

`learning_rate=200.0`

Обычно от 10 до 1000. Если данные выглядят, как шарик, уменьшайте. Если данные выглядят сжатыми, увеличивайте.

`n_iter=1000`

Количество итераций.

`n_iter_without_progress=300`

Прервать, если нет прогресса после этого количества итераций.

`min_grad_norm=1e-07`

Оптимизация прекращается, если норма градиента ниже этого значения.

`metric='euclidean'`

Метрика расстояния от `scipy.spatial.distance.pdist`, `pairwise.PAIRWISE_DISTANCE_METRIC` или функция.

`init='random'`

Инициализация вложения.

`verbose=0`

Многословность.

`random_state=None`

Случайное начальное число.

`method='barnes_hut'`

Алгоритм расчета градиента.

`angle=0.5`

Для расчета градиента. Менее чем 0,2 увеличивает время выполнения. Больше 0,8 увеличивает ошибку.

Атрибуты

`embedding_`

Вложение векторов

`kl_divergence_`

Расстояние Кульбака-Лейблера

`n_iter_`

Количество итераций

Вот визуализация результатов t-SNE с использованием matplotlib (рис. 17.16):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> colors = ["rg"[j] for j in y]
>>> scat = ax.scatter(
...     X_tsne[:, 0],
...     X_tsne[:, 1],
...     c=colors,
...     alpha=0.5,
... )
>>> ax.set_xlabel("Embedding 1")
>>> ax.set_ylabel("Embedding 2")
>>> fig.savefig("images/mlpr_1716.png", dpi=300)
```

Рис. 17.16. *Результат t-SNE с matplotlib*

Изменение значения `perplexity` может оказать большое влияние на график (рис. 17.17). Вот несколько разных значений:

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate((2, 30, 50, 100)):
...     ax = axes[i]
...     t = TSNE(random_state=42, perplexity=n)
...     X_tsne = t.fit_transform(X)
...     pd.DataFrame(X_tsne).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"perplexity={n}")
...     plt.tight_layout()
...     fig.savefig("images/mlpr_1717.png", dpi=300)
```

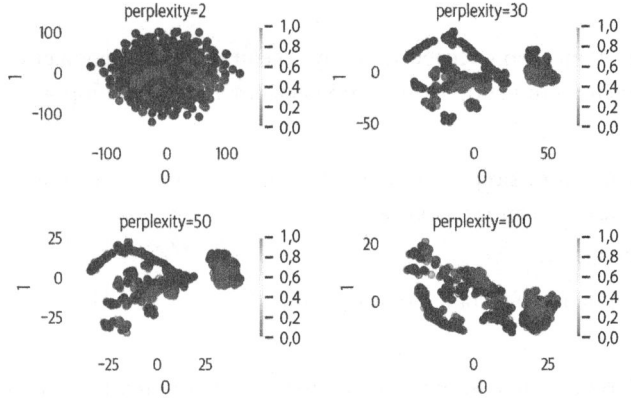


Рис. 17.17. *Изменение* `perplexity` *для t-SNE*

PHATE

PHATE (Potential of Heat-diffusion for Affinity-based Trajectory Embedding) является инструментом для визуализации многомерных данных. Обычно он сохраняет как глобальную структуру (например, PCA), так и локальную (например, t-SNE).

Сначала PHATE кодирует локальную информацию (близкие одна к другой точки должны оставаться близкими). Для обнаружения глобальных данных он использует "диффузию", а затем уменьшает размерность:

```
>>> import phate
>>> p = phate.PHATE(random_state=42)
>>> X_phate = p.fit_transform(X)
>>> X_phate.shape
```

Параметры экземпляра

```
n_components=2
```

Количество измерений.

`knn=5`

Количество соседей по ядру. Увеличьте, если вложение отключено или набор данных превышает 100 000 выборок.

`decay=40`

Скорость затухания ядра. Понижение этого значения увеличивает связность графа.

`n_landmark=2000`

Ориентиры для использования.

`t='auto'`

Степень диффузии. Для данных выполняется сглаживание. Увеличить, если вложение не имеет структуры. Уменьшить, если структура плотная и компактная.

`gamma=1`

Логарифм потенциала (от –1 до 1). Если вложения сосредоточены вокруг одной точки, попробуйте установить значение 0.

`n_pca=100`

Количество основных компонентов для расчета окрестности.

`knn_dist='euclidean'`

Метрика KNN.

`mds_dist='euclidean'`

Метрика многомерного масштабирования (Multidimensional scaling — MDS).

`mds='metric'`

Алгоритм MDS для уменьшения размерности.

`n_jobs=1`

Количество процессоров для использования.

`random_state=None`

Случайное начальное число.

`verbose=1`

Многословность.

Атрибуты (обратите внимание, что за ними не следует _)

`X`

Входные данные.

`embedding`

Вложение пространства.

`diff_op`

Оператор диффузии.

`graph`

Граф KNN, построенный из ввода.

Вот пример использования PHATE (рис. 17.18):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> phate.plot.scatter2d(p, c=y, ax=ax, alpha=0.5)
>>> fig.savefig("images/mlpr_1718.png", dpi=300)
```

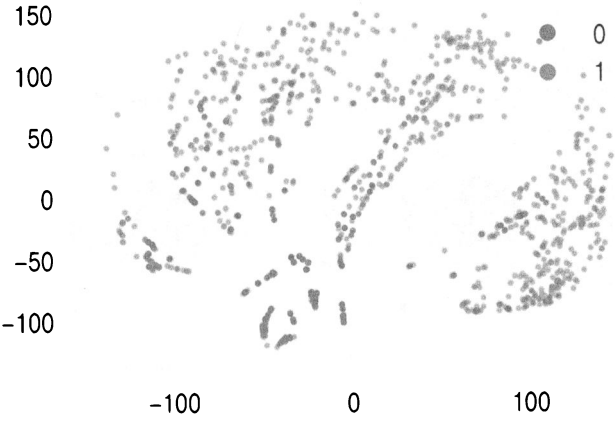


Рис. 17.18. *Результаты PHATE*

Как отмечалось выше, в параметрах экземпляра, есть несколько параметров, которые мы можем настроить так, чтобы изменить поведение модели. Ниже приведен пример настройки параметра `knn` (рис. 17.19). Обратите внимание, что, если мы используем метод `.set_params`, это ускорит вычисления,

так как будут использованы предварительно вычисленный граф и оператор диффузии:

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> p = phate.PHATE(random_state=42, n_jobs=-1)

>>> for i, n in enumerate((2, 5, 20, 100)):
...     ax = axes[i]
...     p.set_params(knn=n)
...     X_phate = p.fit_transform(X)
...     pd.DataFrame(X_phate).plot(
...         kind="scatter",
...         x=0,
...         y=1,
...         ax=ax,
...         c=y,
...         cmap="Spectral",
...         alpha=0.5,
...     )
...     ax.set_title(f"knn={n}")
... plt.tight_layout()
... fig.savefig("images/mlpr_1719.png", dpi=300)
```

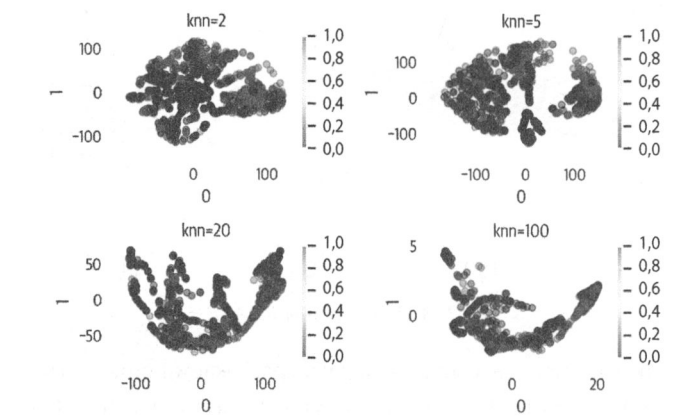


Рис. 17.19. *Изменение параметра* knn *для PHATE*

ГЛАВА 18

Кластеризация

Кластеризация (clustering) — это техника машинного обучения без учителя, используемая для разделения группы на когорты. Поскольку это обучение без учителя, мы не даем модели метки; она просто проверяет признаки и определяет, какие выборки схожи и относятся к кластеру. В этой главе мы рассмотрим методы k-средних и иерархической кластеризации. Мы также снова исследуем набор данных Titanic, используя различные методы.

Метод k-средних

Алгоритм k-средних требует, чтобы пользователь выбрал количество кластеров или *k*. Затем он случайным образом выбирает *k* центроидов и назначает каждую выборку кластеру на основе метрики расстояния от центроида. После назначения он пересчитывает центроиды на основе центра каждой выборки, назначенного метке. Затем он повторяет назначение выборок кластерам на основе новых центроидов. После нескольких итераций должно наступить схождение.

Поскольку для определения схожих выборок кластеризация использует метрики расстояния, в зависимости от масштаба данных поведение может изменяться. Вы можете стандартизировать данные и представить все признаки в одном масштабе. Некоторые полагают, что SME может не рекомендовать

стандартизацию, если есть намеки на то, что некоторые признаки имеют большее значение. В этом примере мы будем стандартизировать данные.

В данном примере мы сгруппируем пассажиров Титаника. Чтобы увидеть, может ли кластеризация отражать выживание, мы начнем с двух кластеров (мы не допустим утечки данных о выживании в кластеризацию и будем использовать только X, а не y).

Алгоритмы без учителя имеют методы `.fit` и `.predict`. В `.fit` мы передаем только X:

```
>>> from sklearn.cluster import KMeans
>>> X_std = preprocessing.Stand
ardScaler().fit_transform(
...     X
... )
>>> km = KMeans(2, random_state=42)
>>> km.fit(X_std)
KMeans(algorithm='auto', copy_x=True,
  init='k-means', max_iter=300,
  n_clusters=2, n_init=10, n_jobs=1,
  precompute_distances='auto',
  random_state=42, tol=0.0001, verbose=0)
```

После обучения модели мы можем вызвать метод `.predict`, чтобы назначить кластеру новые выборки:

```
>>> X_km = km.predict(X)
>>> X_km
array([1, 1, 1, ..., 1, 1, 1], dtype=int32)
```

Параметры экземпляра

```
n_clusters=8
```

Количество кластеров для создания.

```
init='kmeans++'
```

Метод инициализации.

```
n_init=10
```

Количество запусков алгоритма с разными центроидами. Победит наилучший результат.

`max_iter=300`

Количество итераций за запуск.

`tol=0.0001`

Допуск до схождения.

`precompute_distances='auto'`

Предварительное вычисление расстояний (занимает больше памяти, но быстрее). `auto` выполняет предварительные вычисления, если `n_samples * n_clusters` меньше или равно 12 миллионам.

`verbose=0`

Многословность.

`random_state=None`

Случайное начальное число.

`copy_x=True`

Копирование данных перед вычислением.

`n_jobs=1`

Количество процессоров для использования.

`algorithm='auto'`

Алгоритм k-средних. `'full'` работает с разреженными данными, но `'elkan'` эффективнее. `'auto'` использует `'elkan'` с плотными данными.

Атрибуты

`cluster_centers_`

Координаты центров.

`labels_`

Метки для выборок.

`inertia_`

Сумма квадратов расстояния до центроида кластера.

`n_iter_`

Количество итераций.

Если вы не знаете заранее, сколько кластеров вам нужно, запустите алгоритм с различными размерами и оцените различные метрики. Это может быть сложно.

Вы можете построить собственный *локтевой график* (elbow plot), используя расчет `.inertia_`. Посмотрите, где изгибается кривая, поскольку это потенциально хороший выбор для количества кластеров. В данном случае кривая плавная, но после восьми улучшение не кажется значительным (рис. 18.1).

Для участков без изгиба у нас есть несколько вариантов. Мы можем использовать другие метрики, некоторые из которых показаны ниже. Мы также можем проверить визуализацию кластеризации и посмотреть, видны ли кластеры. Мы можем добавить признаки к данным и посмотреть, поможет ли это с кластеризацией.

Вот код локтевого графика:

```
>>> inertias = []
>>> sizes = range(2, 12)
>>> for k in sizes:
...     k2 = KMeans(random_state=42, n_clusters=k)
...     k2.fit(X)
...     inertias.append(k2.inertia_)
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pd.Series(inertias, index=sizes).plot(ax=ax)
>>> ax.set_xlabel("K")
>>> ax.set_ylabel("Inertia")
>>> fig.savefig("images/mlpr_1801.png", dpi=300)
```

Библиотека Scikit-learn имеет и другие метрики кластеризации, когда основные метки истинности неизвестны. Мы можем рассчитать и отобразить их. *Коэффициент силуэта* (silhouette coefficient) — это значение от -1 до 1. Чем выше оценка, тем лучше. 1 означает узкие кластеры, а 0 — перекрывающиеся кластеры. Исходя из этого показателя, два кластера дают нам лучший результат.

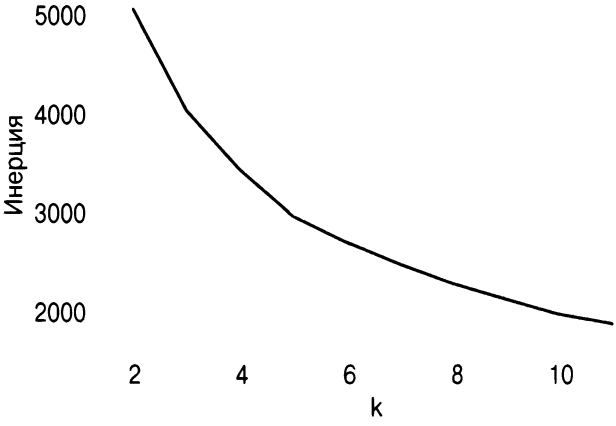


Рис. 18.1. *Локтевой график выглядит довольно гладко*

Индекс Калинского–Харабаза (Calinski–Harabasz Index) представляет собой отношение дисперсии между кластерами и внутри кластера. Чем выше оценка, тем лучше. По этой метрике два кластера дают лучший результат.

Индекс Дэвиса–Болдина (Davis–Bouldin Index) — это среднее сходство между каждым кластером и ближайшим кластером. Оценки варьируются от 0 и выше. 0 указывает на лучшую кластеризацию.

Здесь мы построим график инерции, коэффициента силуэта, индекса Калинского–Харабаза и индекса Дэвиса–Болдина для диапазона размеров кластеров, чтобы увидеть, есть ли для данных четкие размеры кластеров (рис. 18.2). Похоже, что большинство этих метрик согласуются на двух кластерах:

```
>>> from sklearn import metrics
>>> inertias = []
>>> sils = []
>>> chs = []
>>> dbs = []
>>> sizes = range(2, 12)
>>> for k in sizes:
...     k2 = KMeans(random_state=42, n_clusters=k)
...     k2.fit(X_std)
```

```
...     inertias.append(k2.inertia_)
...     sils.append(
...         metrics.silhouette_score(X, k2.labels_)
...     )
...     chs.append(
...         metrics.calinski_harabasz_score(
...             X, k2.labels_
...         )
...     )
...     dbs.append(
...         metrics.davies_bouldin_score(
...             X, k2.labels_
...         )
...      )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> (
...     pd.DataFrame(
...         {
...             "inertia": inertias,
...             "silhouette": sils,
...             "calinski": chs,
...             "davis": dbs,
...             "k": sizes,
...         }
...     )
...     .set_index("k")
...     .plot(ax=ax, subplots=True, layout=(2, 2))
... )
>>> fig.savefig("images/mlpr_1802.png", dpi=300)
```

Другой метод определения кластеров — визуализация силуэтов для каждого кластера. Для этого у библиотеки Yellowbrick есть визуализатор (рис. 18.3).

Вертикальная красная пунктирная линия на этом графике является средней оценкой. Один из способов ее интерпретации — убедиться, что каждый кластер выходит за рамки среднего значения, а оценки кластеров выглядят прилично. Убедитесь, что вы используете одни и те же ограничения x (`ax.set_xlim`). Я бы выбрал два кластера из этих графиков:

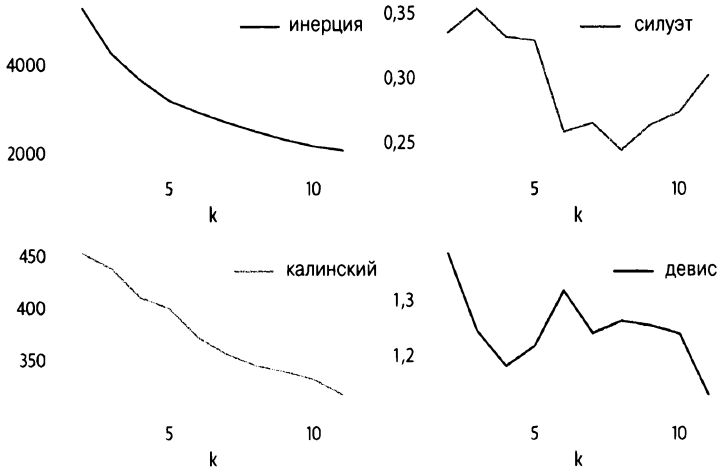


Рис. 18.2. *Метрики кластеризации. Эти метрики в основном согласны на два кластера*

```
>>> from yellowbrick.cluster.silhouette import (
...     SilhouetteVisualizer,
... )
>>> fig, axes = plt.subplots(2, 2, figsize=(12, 8))
>>> axes = axes.reshape(4)
>>> for i, k in enumerate(range(2, 6)):
...     ax = axes[i]
...     sil = SilhouetteVisualizer(
...         KMeans(n_clusters=k, random_state=42),
...         ax=ax,
...     )
...     sil.fit(X_std)
...     sil.finalize()
...     ax.set_xlim(-0.2, 0.8)
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1803.png", dpi=300)
```

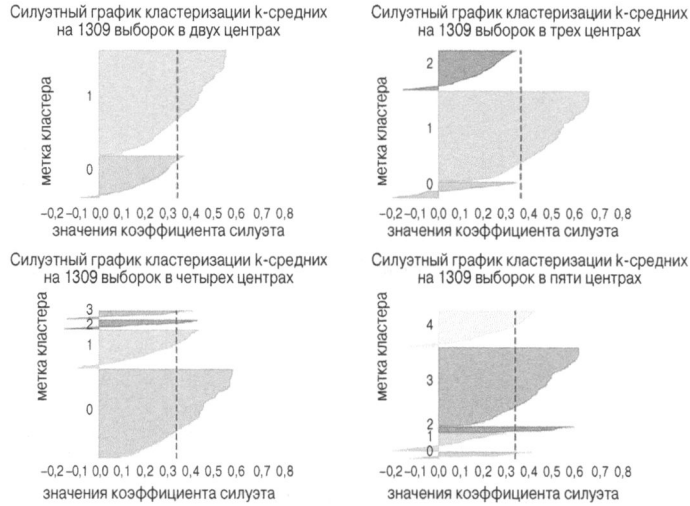


Рис. 18.3. *Визуализатор силуэта Yellowbrick*

Агломерационная (иерархическая) кластеризация

Агломерационная кластеризация (agglomerative clustering) является еще одной методологией. Вы начинаете с каждой выборки в собственном кластере. Затем вы объединяете "ближайшие" кластеры. Повторяйте до завершения, сохраняя при этом ближайшие размеры.

По завершении у вас будет *дендрограмма* (dendrogram), или дерево, отслеживающее, когда были созданы кластеры и какова была метрика расстояния. Для визуализации дендрограммы можете использовать библиотеку scipy.

Мы можем использовать scipy и для создания дендрограммы (рис. 18.4). Как видите, если у вас много выборок, листовые узлы трудно читать:

```
>>> from scipy.cluster import hierarchy
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> dend = hierarchy.dendrogram(
```

```
...     hierarchy.linkage(X_std, method="ward")
... )
>>> fig.savefig("images/mlpr_1804.png", dpi=300)
```

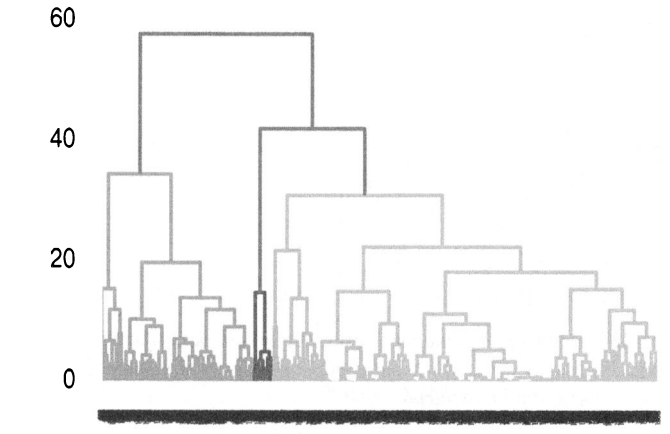


Рис. 18.4. *Дендрограмма иерархической кластеризации scipy*

Если у вас есть дендрограмма, у вас есть все кластеры (от одного до размера выборки). Высота представляет, насколько схожи кластеры, когда они объединены. Чтобы узнать, сколько кластеров в данных, нужно провести горизонтальную линию, пересекающую самые высокие линии.

В данном случае, похоже, что, когда вы выполняете это пересечение, у вас будет три кластера.

Предыдущий график со всеми выборками был немного шумным. Вы также можете использовать параметр `truncate_mode`, чтобы объединять листья в один узел (рис. 18.5):

```
>>> from scipy.cluster import hierarchy
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> dend = hierarchy.dendrogram(
...     hierarchy.linkage(X_std, method="ward"),
...     truncate_mode="lastp",
...     p=20,
...     show_contracted=True,
... )
>>> fig.savefig("images/mlpr_1805.png", dpi=300)
```

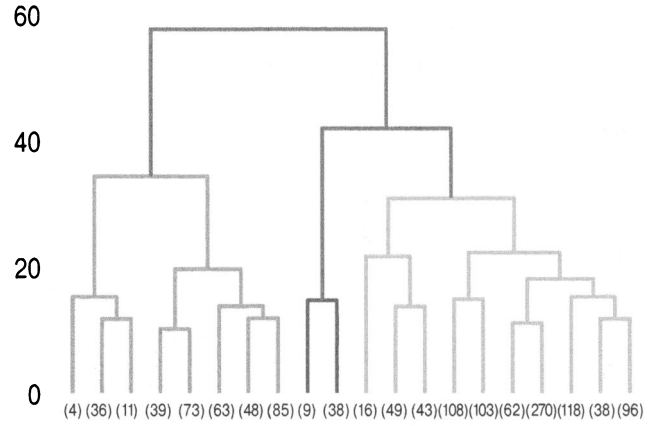


Рис. 18.5. *Усеченная дендрограмма иерархической кластеризации. Если мы пересечем самые большие вертикальные линии, мы получим три кластера*

Как только мы узнаем, сколько кластеров нужно, мы сможем использовать библиотеку Scikit-learn для создания модели:

```
>>> from sklearn.cluster import (
...     AgglomerativeClustering,
... )
>>> ag = AgglomerativeClustering(
...     n_clusters=4,
...     affinity="euclidean",
...     linkage="ward",
... )
>>> ag.fit(X)
```

НА ЗАМЕТКУ

Пакет fastcluster предоставляет оптимизированный метод агломерационной кластеризации, если реализация Scikit-learn слишком медленна.

Понятие кластеров

Используя метод k-средних для набора данных Titanic, мы можем создать два кластера. Можно использовать функции группировки pandas для изучения различий в кластерах. Приведенный ниже код проверяет среднее значение и дисперсию для каждого признака. Похоже, что среднее значение для pclass меняется незначительно.

Я возвращаю данные о выживании, чтобы посмотреть, была ли с этим связана кластеризация:

```
>>> km = KMeans(n_clusters=2)
>>> km.fit(X_std)
>>> labels = km.predict(X_std)
>>> (
...     X.assign(cluster=labels, survived=y)
...     .groupby("cluster")
...     .agg(["mean", "var"])
...     .T
... )
cluster                   0          1
pclass      mean   0.526538  -1.423831
            var    0.266089   0.136175
age         mean  -0.280471   0.921668
            var    0.653027   1.145303
sibsp       mean  -0.010464  -0.107849
            var    1.163848   0.303881
parch       mean   0.387540   0.378453
            var    0.829570   0.540587
fare        mean  -0.349335   0.886400
            var    0.056321   2.225399
sex_male    mean   0.678986   0.552486
            var    0.218194   0.247930
embarked_Q  mean   0.123548   0.016575
            var    0.108398   0.016345
embarked_S  mean   0.741288   0.585635
            var    0.191983   0.243339
survived    mean   0.596685   0.299894
            var    0.241319   0.210180
```

НА ЗАМЕТКУ

В Jupyter вы можете добавить следующий код к DataFrame, и он выделит верхнее и нижнее значения каждой строки. Это полезно для визуального просмотра того, какие значения выделяются в приведенной выше сводке кластера:

```
.style.background_gradient(cmap='RdBu', axis=1)
```

На рис. 18.6 представлена гистограмма средних для каждого кластера:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
... (
...     X.assign(cluster=labels, survived=y)
...     .groupby("cluster")
...     .mean()
...     .T.plot.bar(ax=ax)
... )
>>> fig.savefig(
...     "images/mlpr_1806.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Мне также нравится рисовать компоненты PCA раскрашенными по меткам кластера (рис. 18.7). Здесь мы используем для этого Seaborn. Также интересно изменить значения оттенка, чтобы погрузиться в признаки, характерные для кластеров.

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> sns.scatterplot(
...     "PC1",
...     "PC2",
...         data=X.assign(
...         PC1=X_pca[:, 0],
...         PC2=X_pca[:, 1],
...         cluster=labels,
...     ),
...     hue="cluster",
...     alpha=0.5,
...     ax=ax,
```

```
... )
>>> fig.savefig(
...     "images/mlpr_1807.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

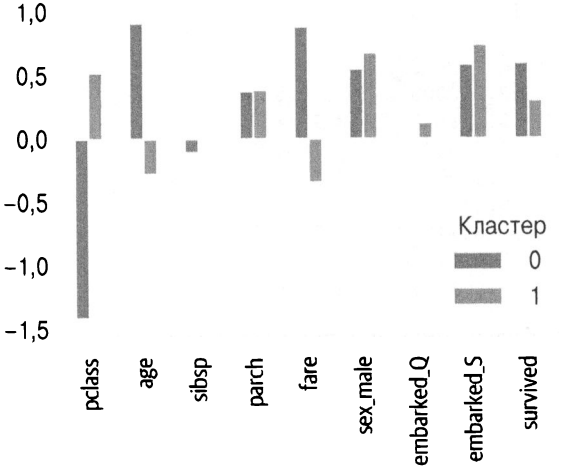


Рис. 18.6. *Средние значения каждого кластера*

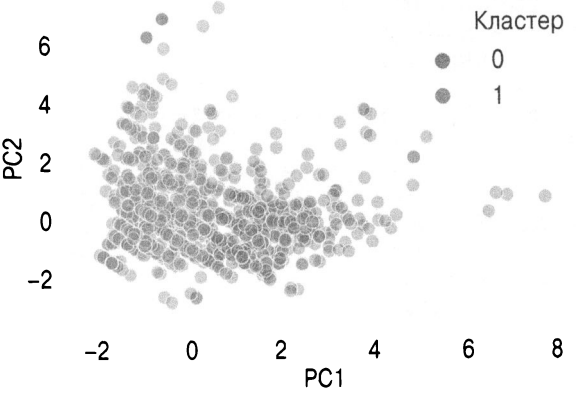


Рис. 18.7. *График кластеров PCA*

Чтобы изучить один признак, можно использовать метод pandas .describe:

```
>>> (
...     X.assign(cluster=label)
...     .groupby("cluster")
...     .age.describe()
...     .T
... )
cluster          0           1
count   362.000000  947.000000
mean      0.921668   -0.280471
std       1.070188    0.808101
min      -2.160126   -2.218578
25%       0.184415   -0.672870
50%       0.867467   -0.283195
75%       1.665179    0.106480
max       4.003228    3.535618
```

Для объяснения кластеров мы также можем создать суррогатную модель. Здесь мы используем дерево решений, чтобы объяснить их. Это также демонстрирует, что pclass (имеющий большое различие в среднем) очень важен:

```
>>> dt = tree.DecisionTreeClassifier()
>>> dt.fit(X, labels)
>>> for col, val in sorted(
...     zip(X.columns, dt.feature_importances_),
...     key=lambda col_val: col_val[1],
...     reverse=True,
... ):
...     print(f"{col:10}{val:10.3f}")
pclass         0.902
age            0.074
sex_male       0.016
embarked_S     0.003
fare           0.003
parch          0.003
sibsp          0.000
embarked_Q     0.000
```

И мы можем визуализировать решения, как на рис. 18.8, демонстрирующем, что pclass — это первый признак, на который суррогат обращает внимание, чтобы принять решение:

```
>>> dot_data = StringIO()
>>> tree.export_graphviz(
...     dt,
...     out_file=dot_data,
...     feature_names=X.columns,
...     class_names=["0", "1"],
...     max_depth=2,
...     filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
...     dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1808.png")
```

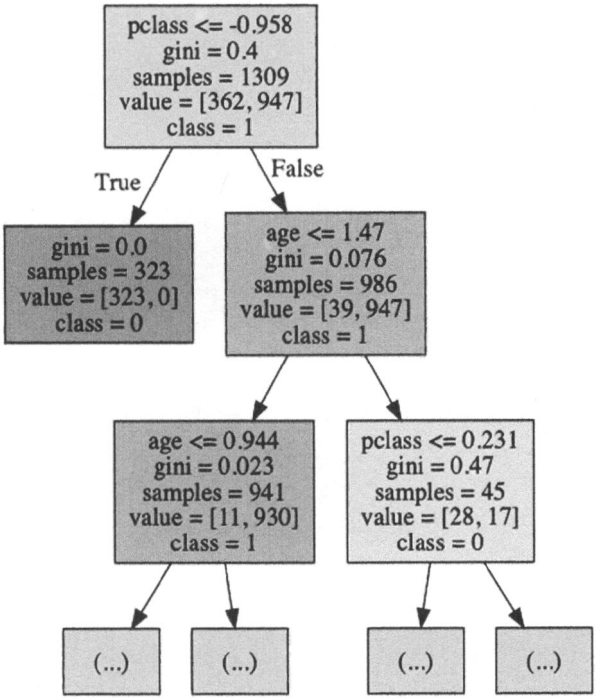


Рис. 18.8. *Дерево решений, объясняющее кластеризацию*

ГЛАВА 19

Конвейер

Библиотека Scikit-learn использует понятие *конвейера* (pipeline). Используя класс `Pipeline`, вы можете связать воедино трансформеры и модели, а весь процесс можете рассматривать как модель Scikit-learn. Вы даже можете вставить собственную логику.

Классификационный конвейер

Вот пример использования функции `tweak_titanic` внутри конвейера:

```
>>> from sklearn.base import (
...     BaseEstimator,
...     TransformerMixin,
... )
>>> from sklearn.pipeline import Pipeline

>>> def tweak_titanic(df):
...     df = df.drop(
...     columns=[
...         "name",
...         "ticket",
...         "home.dest",
...         "boat",
...         "body",
...         "cabin",
...         ]
```

```
...     ).pipe(pd.get_dummies, drop_first=True)
...     return df

>>> class TitanicTransformer(
...     BaseEstimator, TransformerMixin
... ):
...     def transform(self, X):
...         # предполагается, что X получаем
...         # из чтения файла Excel
...         X = tweak_titanic(X)
...         X = X.drop(column="survived")
...         return X
...
...     def fit(self, X, y):
...         return self

>>> pipe = Pipeline(
...     [
...         ("titan", TitanicTransformer()),
...         ("impute", impute.IterativeImputer()),
...         (
...             "std",
...             preprocessing.StandardScaler(),
...         ),
...         ("rf", RandomForestClassifier()),
...     ]
... )
```

Имея конвейер, мы можем вызвать методы `.fit` и `.score`:

```
>>> from sklearn.model_selection import (
...     train_test_split,
... )
>>> X_train2, X_test2, y_train2, y_test2 =
train_test_split(
...     orig_df,
...     orig_df.survived,
...     test_size=0.3,
...     random_state=42,
... )
>>> pipe.fit(X_train2, y_train2)
>>> pipe.score(X_test2, y_test2)
0.7913486005089059
```

Конвейеры могут быть использованы при сеточном поиске. Наша функция `param_grid` должна иметь параметры с префиксом имени этапа конвейера, за которым следуют два подчеркивания. В приведенном ниже примере мы добавляем некоторые параметры для этапа случайного леса:

```
>>> params = {
...     "rf__max_features": [0.4, "auto"],
...     "rf__n_estimators": [15, 200],
... }

>>> grid = model_selection.GridSearchCV(
...     pipe, cv=3, param_grid=params
... )
>>> grid.fit(orig_df, orig_df.survived)
```

Теперь мы можем извлечь лучшие параметры и обучить окончательную модель. (В данном случае случайный лес не улучшается после сеточного поиска.)

```
>>> grid.best_params_
{'rf__max_features': 0.4, 'rf__n_estimators': 15}
>>> pipe.set_params(**grid.best_params_)
>>> pipe.fit(X_train2, y_train2)
>>> pipe.score(X_test2, y_test2)
0.7913486005089059
```

Мы можем использовать конвейер, в котором присутствуют модели Scikit-learn:

```
>>> metrics.roc_auc_score(
...     y_test2, pipe.predict(X_test2)
... )
0.7813688715131023
```

Конвейер регрессии

Вот пример конвейера, выполняющего линейную регрессию для набора данных Boston:

```
>>> from sklearn.pipeline import Pipeline

>>> reg_pipe = Pipeline(
...     [
...         (
...             "std",
...             preprocessing.StandardScaler(),
...         ),
...         ("lr", LinearRegression()),
...     ]
... )
>>> reg_pipe.fit(bos_X_train, bos_y_train)
>>> reg_pipe.score(bos_X_test, bos_y_test)
0.7112260057484934
```

Извлечь из конвейера части, чтобы проверить их свойства, можно с помощью атрибута `.named_steps`:

```
>>> reg_pipe.named_steps["lr"].intercept_
23.01581920903956
>>> reg_pipe.named_steps["lr"].coef_
array([-1.10834602,  0.80843998, 0.34313466,
        0.81386426, -1.79804295, 2.913858 ,
       -0.29893918, -2.94251148, 2.09419303,
       -1.44706731, -2.05232232, 1.02375187,
       -3.88579002])_
```

Мы также можем использовать конвейер при вычислениях метрик:

```
>>> from sklearn import metrics
>>> metrics.mean_squared_error(
...     bos_y_test, reg_pipe.predict(bos_X_test)
... )
21.517444231177205
```

Конвейер PCA

Конвейеры Scikit-learn могут быть также использованы для PCA.

Здесь мы стандартизируем набор данных Titanic и выполняем для него PCA:

```
>>> pca_pipe = Pipeline(
...     [
...         (
...             "std",
...             preprocessing.StandardScaler(),
...         ),
...         ("pca", PCA()),
...     ]
... )
>>> X_pca = pca_pipe.fit_transform(X)
```

Используя атрибут `.named_steps`, мы можем извлечь свойства из части PCA конвейера:

```
>>> pca_pipe.named_steps[
... "pca"
... ].explained_variance_ratio_
array([0.23917891, 0.21623078, 0.19265028,
       0.10460882, 0.08170342, 0.07229959,
       0.05133752, 0.04199068])
>>> pca_pipe.named_steps["pca"].components_[0]
array([-0.63368693,  0.39682566,  0.00614498,
        0.11488415,  0.58075352, -0.19046812,
       -0.21190808, -0.09631388])
```

Предметный указатель

A

Accuracy, 178
ADASYN, 117
Agglomerative clustering, 292
Area Under The Curve, 55
AUC, 55

B

Bagging, 115; 143
Boosting, 115
Box plot, 78
Breusch–Pagan test, 242

C

Calinski–Harabasz Index, 289
CART, 136
Classification, 121
  And Regression Tree, 136
Class prediction error, 187
Clustering, 285
Coefficient of determination, 207; 239
Confusion matrix, 53; 173
CRISP-DM, 27
Cumulative
  gains plot, 183
  plot, 256
Curse of dimensionality, 105

D

Data mining, 27
Davis–Bouldin Index, 289
Dendrogram, 63; 292
Dummy variable, 94

E

Elbow
  method, 256
  plot, 288
Explained variance, 240
Extreme Gradient Boosting, 116

F

Feature, 32
  importance, 153
  selection, 105
Frequency encoding, 96

G

Gini
  importance, 148
  impurity, 136
Grid search, 116

H

Heat map, 62
Histogram, 73
Homoscedasticity, 242
Hyperparameter, 52

I

Imbalanced class, 115
Imputation, 64
Index measure, 136
Indicator encoding, 94
Instance-based learning, 133
Intercept, 125
Interface, 157

J

Joint plot, 75
Jupyter, 29

K

Kernel
  density estimate, 77
  trick, 130
K-Nearest Neighbor, 133
KNN, 133
К-ближайшие соседи, 133

L

Label encoding, 95
Leaky feature, 34
Learning curve, 56
Lift curve, 185
Loading plot, 262

M

Manifold learning, 271
Mean
  absolute error, 240
  squared logarithmic error, 241
Multicollinearity, 106; 209
Mutual information, 112

N

Naive Bayes, 127

O

OLS, 240
One-hot encoding, 94
One-versus-rest, 138
OOB error, 143
Ordinary Least Squares, 240
Out Of Bag, 106
OVR, 138

P

Pair grid, 77
PCA, 253
Pearson correlation, 75
Permutation importance, 148
PHATE, 281
Pipeline, 301
Precision, 115; 179
Principal Component Analysis, 253

Q

Queue rate, 188

R

Radial Basis Function, 130
Random forest, 116; 143
Recall, 115; 178
Receiver Operating Characteristic, 55
Regression, 205
Residual, 240
ROC, 55
Root mean squared error, 241

S

Sample, 32
Scatter plot, 74
Scree plot, 256
Sensitivity, 178
Silhouette coefficient, 288
SMOTE, 117

Stratified sampling, 186
Supervised learning, 121
Support vector, 130
  Classifier, 198
  Machine, 91; 129
SVC, 198
SVM, 91; 129

T

t-SNE, 277

U

UMAP, 271

V

Validation
  curve, 169
  score, 169
Violin plot, 78

W

Weak tree, 149

X

XGBoost, 116

Z

Zero probability problem, 129

А

Агломерационная кластеризация, 292
Алгоритм k-средних, 285
Анализ основных компонентов, 112; 253

Б

Библиотека
  Bokeh, 265
  categoryor_encoding, 99
  fancyimpute, 44; 65
  LightGBM, 159
  matplotlib, 193
  missingno, 60
  pandas, 19; 30
  pandas_profiling, 36
  pyjanitor, 43
  Scikit-learn, 30
  scikit-plot, 184
  scipy, 244
  seaborn, 73
  statsmodel, 242
  TPOT, 165
  xgbfir, 156
  XGBoost, 116; 149
  Yellowbrick, 30
Бустинг, 115
Бэггинг, 115; 143

В

Важность
  перестановки, 148
  признака, 52; 153; 230
  удаления столбца, 148
Взаимная информация, 112
Выборка, 32
Выбор признаков, 105

Г

Гетероскедастичность, 208
Гиперпараметр, 52
Гистограмма, 73
Гомоскедастичность, 242

График
RadViz, 86
кумулятивного усиления, 183
нагрузки, 262
распределения вероятностей, 244
собственных значений, 256

Д

Дендрограмма, 63; 292
Дерево классификации и регрессии, 136
Диаграмма
размаха, 78
рассеяния, 74

З

Заменитель, 44
Замещение, 64

И

Индекс
важности Джини, 148
Дэвиса–Болдина, 289
Калинского–Харабаза, 289
Индикаторное кодирование, 94
Интеллектуальный анализ данных, 27
Интерфейс, 157

К

Категориальный столбец, 35
Классификатор опорных векторов, 198
Классификация, 121; 181
Кластеризация, 285
Конвейер, 301
Корректность, 178
Корреляция, 82
Пирсона, 75
Коэффициент
детерминации, 207; 239
Джини, 136
очереди, 188
силуэта, 288
Кривая
ROC, 180
валидации, 169
обучения, 56
подъема, 185
рабочей характеристики приемника, 55
точность–отзыв, 181
Критерий Колмогорова–Смирнова, 244
Кумулятивный график, 256

Л

Локтевой график, 288

М

Матрица неточностей, 53; 173
Метод
локтей, 256
опорных векторов, 91; 129
Меточное кодирование, 95
Метрика, 239
Модель KNN, 214
Мультиколлинеарность, 106; 209

Н

Набор данных Titanic, 29
Наивный байесовский
  классификатор, 127
Несбалансированный
  класс, 115

О

Обучение
  многообразием, 271
  на примерах, 133
  с учителем, 121
Объединенный график, 75
Объяснимая дисперсия, 240
Один против всех, 138
Опорный вектор, 130
Остаток, 240
Отзыв, 115; 178
Отсечение, 125
Оценка
  OOB, 106
  валидации, 169
Очистка данных, 34
Ошибка
  OOB, 143
  прогнозирования класса, 187

П

Пакет
  cookiecutter, 30
  dtreeviz, 140; 221
  fastcluster, 294
  rfpimp, 106
  SHAP, 199
  Shapley, 247
  treeinterpreter, 194
  xgbfir, 156
Парная сетка, 77
Площадь под кривой, 55
Показатель индекса, 136
Признак, 32
  утечки, 34
Проблема нулевой
  вероятности, 129
Проклятие размерности, 105

Р

Рабочий процесс машинного
  обучения, 27
Радиальная базисная
  функция, 130
Регрессия, 205
  лассо, 108
Рефакторинг, 29

С

Сглаживание по Лапласу, 129
Сеточный поиск, 116
Скрипичная диаграмма
  размаха, 78
Слабое дерево, 149
Случайный лес, 116; 143
Среднее снижение
  инородности, 148
Среднеквадратичная
  логарифмическая ошибка, 241
  ошибка, 241
Средняя абсолютная
  ошибка, 240
Стандартизация, 34; 91
Стекирование, 49
Стохастическое
  вложение соседей с
  t-распределением, 277
Стратифицированная
  выборка, 186

Т

Тепловая карта, 62
Тест Бройша–Пагана, 242
Точность, 115; 179
Трюк ядра, 130

У

Унитарное кодирование, 94
Установка
  conda, 24
  pip, 23

Ф

Фиктивная переменная, 94
Фиктивный столбец, 35

Ч

Частотное кодирование, 96
Чувствительность, 178

Э

Экстремальный градиентный бустинг, 116

Я

Ядерная оценка плотности, 77

ПРИКЛАДНОЕ МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ SCIKIT-LEARN, KERAS И TENSORFLOW

2-е ПОЛНОЦВЕТНОЕ ИЗДАНИЕ

Орельен Жерон

www.williamspublishing.com

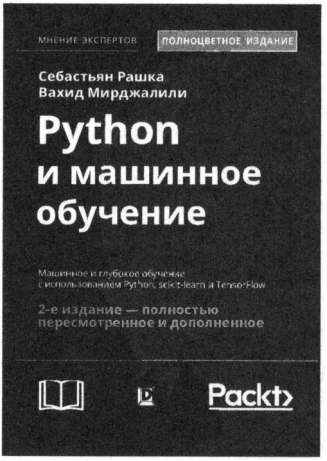
Благодаря серии выдающихся достижений глубокое обучение значительно усилило всю область машинного обучения. В наше время даже программисты, почти ничего не знающие об этой технологии, могут использовать простые и эффективные инструменты для реализации программ, которые способны обучаться на основе данных.

За счет применения конкретных примеров, минимума теории и фреймворков Python производственного уровня обновленное издание этой ставшей бестселлером книги поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

ISBN 978-5-907203-33-4 **в продаже**

PYTHON И МАШИННОЕ ОБУЧЕНИЕ
МАШИННОЕ И ГЛУБОКОЕ ОБУЧЕНИЕ С ИСПОЛЬЗОВАНИЕМ PYTHON, SCIKIT-LEARN И TENSORFLOW, 2-Е ИЗДАНИЕ

Себастьян Рашка
и Вахид Мирджалили



www.dialektika.com

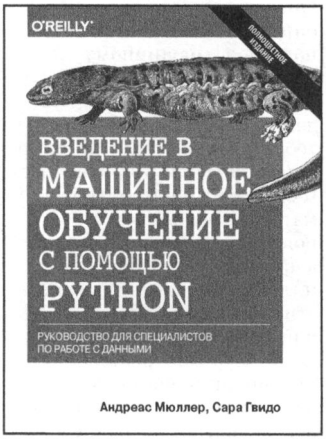
Машинное обучение поглощает мир программного обеспечения, и теперь глубокое обучение расширяет машинное обучение. Освойте и работайте с передовыми технологиями машинного обучения, нейронных сетей и глубокого обучения с помощью 2-го издания бестселлера Себастьяна Рашки. Будучи основательно обновленной с учетом самых последних библиотек Python с открытым кодом, эта книга предлагает практические знания и приемы, которые необходимы для создания и содействия машинному обучению, глубокому обучению и современному анализу данных. Если вы читали 1-е издание книги, то вам доставит удовольствие найти новый баланс классических идей и современных знаний в машинном обучении. Каждая глава была серьезно обновлена, и появились новые главы по ключевым технологиям. У вас будет возможность изучить и поработать с TensorFlow более вдумчиво, нежели ранее, а также получить важнейший охват библиотеки для нейронных сетей Keras наряду с самыми свежими обновлениями библиотеки scikit-learn.

ISBN 978-5-907114-52-4 **в продаже**

ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON

РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

Андреас Мюллер
Сара Гвидо



www.williamspublishing.com

Эта полноцветная книга — отличный источник информации для каждого, кто собирается использовать машинное обучение на практике. Ныне машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, и не следует думать, что эта область — прерогатива исключительно крупных компаний с мощными командами аналитиков. Эта книга научит вас практическим способам построения систем МО, даже если вы еще новичок в этой области. В ней подробно объясняются все этапы, необходимые для создания успешного проекта машинного обучения, с использованием языка Python и библиотек scikit-learn, NumPy и matplotlib. Авторы сосредоточили свое внимание исключительно на практических аспектах применения алгоритмов машинного обучения, оставив за рамками книги их математическое обоснование. Данная книга адресована специалистам, решающим реальные задачи, а поскольку область применения методов МО практически безгранична, прочитав эту книгу, вы сможете собственными силами построить действующую систему машинного обучения в любой научной или коммерческой сфере.

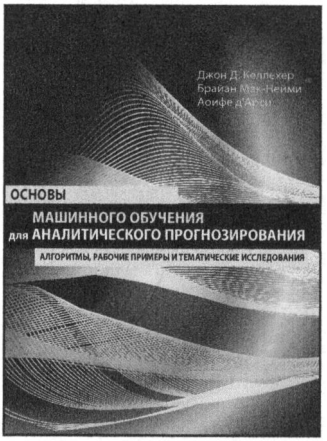
ISBN 978-5-9908910-8-1

в продаже

ОСНОВЫ МАШИННОГО ОБУЧЕНИЯ ДЛЯ АНАЛИТИЧЕСКОГО ПРОГНОЗИРОВАНИЯ

АЛГОРИТМЫ, РАБОЧИЕ ПРИМЕРЫ И ТЕМАТИЧЕСКИЕ ИССЛЕДОВАНИЯ

Джон Д. Келлехер
Брайан Мак-Нейми
Аоифе д'Арси



www.williamspublishing.com

Книга представляет собой учебник по машинному обучению с акцентом на коммерческие приложения. Она предлагает подробное описание наиболее важных подходов к машинному обучению, используемых в интеллектуальном анализе данных, охватывающих как теоретические концепции, так и практические приложения. Формальный математический материал дополняется пояснительными примерами, а примеры исследований иллюстрируют применение этих моделей в более широком контексте бизнеса. В книге рассмотрены информационное обучение, обучение на основе сходства, вероятностное обучение и обучение на основе ошибок. Описанию каждого из этих подходов предшествует объяснение основополагающей концепции, за которой следуют математические модели и алгоритмы, иллюстрированные подробными рабочими примерами.

ISBN 978-5-6040044-9-4

в продаже

ГЛУБОКОЕ ОБУЧЕНИЕ
ГОТОВЫЕ РЕШЕНИЯ

Давид Осинга

www.williamspublishing.com

Благодаря готовым примерам, приведенным в книге, вы научитесь решать задачи, связанные с классификацией и генерированием текста, изображений и музыки. В каждой главе описывается несколько решений, объединяемых в единый проект, например приложение, реализующее тренировку музыкальной рекомендательной системы. Также имеется глава с описанием методик, которые в случае необходимости помогут выполнить отладку нейронной сети. Основные темы книги:

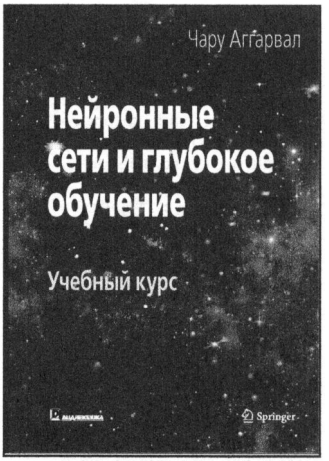
- использование векторных представлений слов для вычисления схожести текстов;
- построение рекомендательной системы фильмов на основе ссылок в Википедии;
- визуализация внутренних состояний нейронной сети;
- создание модели, рекомендующей эмодзи для фрагментов текста;
- повторное использование предварительно обученных сетей для создания службы обратного поиска изображений;
- генерирование пиктограмм с помощью генеративно-состязательных сетей (GAN), автокодировщиков и рекуррентных сетей (RNN);
- распознавание музыкальных жанров и индексирование коллекций песен.

ISBN: 978-5-907144-50-7

в продаже

НЕЙРОННЫЕ СЕТИ И ГЛУБОКОЕ ОБУЧЕНИЕ
УЧЕБНЫЙ КУРС

Чару Аггарвал



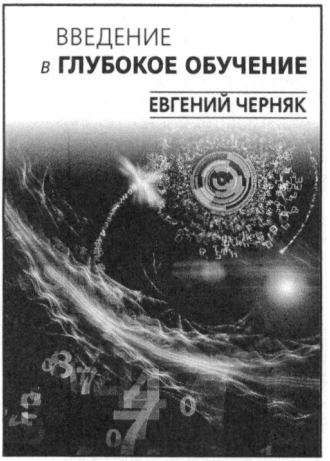
www.dialektika.com

В книге рассматриваются как классические, так и современные модели глубокого обучения. В первых двух главах основной упор сделан на понимании взаимосвязи традиционного машинного обучения и нейронных сетей. Главы 3 и 4 посвящены подробному обсуждению процессов тренировки и регуляризации нейронных сетей. В главах 5 и 6 рассмотрены сети радиально-базисных функций (RBF) и ограниченные машины Больцмана. В главах 7 и 8 обсуждаются рекуррентные и сверточные нейронные сети. Главы 9 и 10 посвящены более сложным темам, таким как глубокое обучение с подкреплением, нейронные машины Тьюринга, самоорганизующиеся карты Кохонена и генеративно-состязательные сети. Книга предназначена для студентов старших курсов, исследователей и специалистов-практиков.

ISBN: 978-5-907203-01-3 **в продаже**

ВВЕДЕНИЕ В ГЛУБОКОЕ ОБУЧЕНИЕ

Евгений Черняк



www.dialektika.com

Автор книги — давний исследователь искусственного интеллекта, специализирующийся на обработке естественного языка, революцию в котором сделало глубокое обучение. К сожалению, ему потребовалось много времени, чтобы это понять. Можно сказать, что нейронные сети угрожают революцией уже третий раз, а отнюдь не первый. Тем не менее автор внезапно оказался далеко позади и изо всех сил пытался наверстать упущенное. Именно поэтому он сделал то, что сделал бы на его месте любой уважающий себя профессор: запланировал преподавание материала и начал ускоренный курс, просматривая веб-страницы. Этим объясняется несколько выдающихся особенностей этой книги. Во-первых, краткость. Во-вторых, она сильно зависит от проекта. Автор считает, что материал по информатике лучше изучать при написании программ, поэтому книга во многом отражает его привычки в преподавании. Эта книга, в первую очередь, задумана как учебник для курса по глубокому обучению. Курс, который автор преподает в Брауне, предназначен как для выпускников, так и для остальных студентов, и охватывает весь материал.

ISBN 978-5-907203-10-5

в продаже

ГЛУБОКОЕ ОБУЧЕНИЕ И TENSORFLOW
ДЛЯ ПРОФЕССИОНАЛОВ

Сантану Паттанаяк

www.williamspublishing.com

Данная книга представляет собой углубленное практическое руководство, которое позволит читателям освоить методы глубокого обучения на уровне, достаточном для развертывания готовых решений. Прочитав книгу, вы сможете быстро приступить к работе с библиотекой TensorFlow и заняться оптимизацией архитектур глубокого обучения. Весь программный код доступен в виде блокнотов iPython и сценариев, позволяющих с легкостью воспроизводить примеры и экспериментировать с ними.

Благодаря этой книге вы:

- овладеете полным стеком технологий глубокого обучения с использованием TensorFlow и получите необходимую для этого математическую подготовку;
- научитесь развертывать сложные приложения глубокого обучения в производственной среде с помощью TensorFlow;
- сможете проводить исследования в области глубокого обучения и выполнять самостоятельные эксперименты в TensorFlow.

ISBN: 978-5-907144-25-5

в продаже

O'REILLY®

Машинное обучение: карманный справочник

Этот карманный справочник, содержащий подробные комментарии, таблицы и примеры, поможет вам ориентироваться в основах структурированного машинного обучения. Автор, Мэтт Харрисон, предлагает ценный учебник, который вы можете использовать как дополнительное пособие при обучении и как удобный ресурс для работы над своим следующим проектом машинного обучения.

В этой книге, идеально подходящей для программистов, аналитиков данных и инженеров искусственного интеллекта, также содержатся обзор процесса машинного обучения и классификация структурированных данных. Кроме всего прочего, с ее помощью вы изучите методы кластеризации, регрессии и уменьшения размерности.

Основные темы книги

- Классификация с использованием набора данных Titanic
- Как очистить данные и справиться с их недостатком
- Разведочный анализ данных
- Общие этапы предварительной обработки с использованием выборки данных
- Выбор признаков, полезных для модели
- Выбор модели
- Оценка метрики и классификации
- Примеры регрессии с использованием нескольких методов машинного обучения
- Метрики для оценки регрессии
- Кластеризация
- Уменьшение размерности
- Конвейеры Scikit-learn

Искусственный интеллект/Моделирование данных

ISBN 978-5-907203-17-4

http://www.williamspublishing.com

www.oreilly.com