

Артем Груздев



	CUSTOMER LIFETIME VALUE	COVERAGE	EDUCATION	EMPLOYMENT STATUS	GENDER	INCOME	MONTHLY PREMIUM AUTO
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN
2	NaN	NaN	NaN	Employed	F	48767.0	108.0
3	7645.861827	Basic	Bachelor	NaN	NaN	0.0	106.0
4	2813.692575	Basic	Bachelor	NaN	M	43836.0	73.0

Предварительная подготовка данных в Python

Том 2. План, примеры и метрики качества

ИЦ «Гевисста»



В двухтомнике представлены материалы по применению классических методов машинного обучения для различных промышленных задач.

Прочитав второй том, вы научитесь:

- составлять план предварительной подготовки данных;
- конструировать признаки;
- отбирать признаки;
- работать с метриками бинарной классификации и регрессии;
- выполнять байесовскую оптимизацию гиперпараметров;
- создавать контейнеры Docker;
- строить модели с помощью платформы H2O.



Артем Груздев – основатель и директор компании «Гевисста», имеет 10-летний опыт прогнозирования кредитных рисков и 18-летний опыт статистического анализа. В последние годы активно занимается практическим построением прогнозных моделей.

Ведет Telegram-канал, посвященный машинному обучению <https://t.me/Gewissta>.

ИЦ «Гевисста»



Исследовательский центр «Гевисста» с 2009 г. осуществляет разработку, валидацию, внедрение и мониторинг риск-моделей, моделей оттока, моделей отклика на базе IBM SPSS Statistics, IBM SPSS Modeler, SAS Enterprise Miner, SAS Enterprise Guide, R, Python. Осуществляет подготовку специалистов в сфере прогнозного моделирования и анализа данных.

Интернет-магазин:

www.dmkpress.com

Оптовая продажа:

KTK «Галактика»

books@aliants-kniga.ru



ISBN 978-5-93700-177-1



9 785937 001771 >

А. В. Груздев

Предварительная подготовка данных в Python

Том 2

План, примеры и метрики качества



Москва, 2023

УДК 004.04Python

ББК 32.372

Г90

Груздев А. В.

- Г90** Предварительная подготовка данных в Python. Том 2: План, примеры и метрики качества. – М.: ДМК Пресс, 2023. – 814 с.: ил.

ISBN 978-5-93700-177-1

В двухтомнике представлены материалы по применению классических методов машинного обучения в различных промышленных задачах. Во втором томе рассматривается сам процесс предварительной подготовки данных, а также некоторые метрики качества и ряд полезных библиотек и фреймворков (H2O, Dask, Docker, Google Colab).

Издание рассчитано на специалистов по анализу данных, а также может быть полезно широкому кругу специалистов, интересующихся машинным обучением.

УДК 004.04Python

ББК 32.372

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

Оглавление

Введение	7
ЧАСТЬ 3. ПЛАН ПРЕДВАРИТЕЛЬНОЙ ПОДГОТОВКИ ДАННЫХ	8
1. Введение	8
2. Формирование выборки	10
2.1. Генеральная и выборочная совокупности	10
2.2. Характеристики выборки.....	10
2.3. Детерминированные и вероятностные выборки	12
2.4. Виды, методы и способы вероятностного отбора	13
2.5. Подходы к определению необходимого объема выборки	14
3. Определение «окна выборки» и «окна созревания»	28
4. Определение зависимой переменной	32
5. Загрузка данных из CSV-файлов и баз данных SQL.....	33
6. Удаление бесполезных переменных, переменных «из будущего», переменных с юридическим риском.....	39
7. Преобразование типов переменных и знакомство со шкалами переменных.....	41
7.1. Количественные (непрерывные) шкалы.....	41
7.2. Качественные (дискретные) шкалы.....	43
8. Нормализация строковых значений	45
9. Обработка дублирующихся наблюдений.....	61
10. Обработка редких категорий	62
11. Появление новых категорий в новых данных	69
12. Импутация пропусков.....	70
12.1. Способы импутации количественных и бинарных переменных	70

12.2. Способы импутации категориальных переменных	71
12.3. Практика	73
13. Обработка выбросов.....	90
14. Описательные статистики	94
14.1. Пифагорейские средние, медиана и мода	94
14.2. Квантиль	95
14.3. Дисперсия и стандартное отклонение	96
14.4. Корреляция и ковариация	97
14.5. Получение сводки описательных статистик в библиотеке pandas.....	102
15. Нормальное распределение.....	104
15.1. Знакомство с нормальным распределением	104
15.2. Коэффициент острогишинности, коэффициент эксцесса и коэффициент асимметрии	107
15.3. Гистограмма распределения и график квантиль–квантиль.....	111
15.4. Вычисление коэффициента асимметрии и коэффициента эксцесса, построение гистограммы и графика квантиль–квантиль для подбора преобразований, максимизирующих нормальность	112
15.5. Подбор преобразований, максимизирующих нормальность для правосторонней асимметрии	116
15.6. Подбор преобразований, максимизирующих нормальность для левосторонней асимметрии.....	128
15.7. Преобразование Бокса–Кокса	129
16. Конструирование признаков	135
16.1. Статическое конструирование признаков исходя из предметной области	135
16.2. Статическое конструирование признаков исходя из алгоритма	170
16.3. Динамическое конструирование признаков исходя из особенностей алгоритма	290
16.4. Конструирование признаков для временных рядов	297
17. Отбор признаков	433
17.1. Методы-фильтры	436
17.2. Применение метода-фильтра и встроенного метода для отбора признаков (на примере соревнования BNP Paribas Cardif Claims Management с Kaggle)	444
17.3. Комбинирование нескольких методов для отбора признаков (на примере соревнования Porto Seguro's Safe Driver Prediction с Kaggle)	451
18. Стандартизация.....	475
19. Собираем все вместе	486

ЧАСТЬ 4. МЕТРИКИ ДЛЯ ОЦЕНКИ КАЧЕСТВА МОДЕЛИ....514**1. Бинарная классификация.....514**

1.1. Отрицательный и положительный классы, порог отсечения	514
1.2. Матрица ошибок	514
1.3. Доля правильных ответов, правильность (accuracy)	517
1.4. Чувствительность (sensitivity).....	519
1.5. Специфичность (specificity)	521
1.6. $1 - \text{специфичность}$ ($1 - \text{specificity}$)	522
1.7. Сбалансированная правильность.....	523
1.8. Точность (Precision).....	524
1.9. Сравнение точности и чувствительности (полноты)	525
1.10. F-мера (F-score, или F-measure)	526
1.11. Варьирование порога отсечения.....	532
1.12. Коэффициент Мэттьюса (Matthews correlation coefficient или MCC).....	536
1.13. Каппа Коэна (Cohen's kappa).....	540
1.14. ROC-кривая (ROC curve) и площадь под ROC-кривой (AUC-ROC).....	542
1.15. PR-кривая (PR curve) и площадь под PR-кривой (AUC-PR)	603
1.16. Кривая Лоренца (Lorenz curve) и коэффициент Джини (Gini coefficient).....	616
1.17. CAP-кривая (CAP curve).....	620
1.18. Статистика Колмогорова–Смирнова (Kolmogorov–Smirnov statistic) ...	623
1.19. Биномиальный тест (binomial test)	626
1.20. Логистическая функция потерь (logistic loss)	628

2. Регрессия.....634

2.1. R^2 , коэффициент детерминации (R-square, coefficient of determination)	634
2.2. Метрики качества, которые зависят от масштаба данных (RMSE, MSE, MAE, MdAE, RMSLE, MSLE)	643
2.3. Метрики качества на основе процентных ошибок (MAPE, MdAPE, sMAPE, sMdAPE, WAPE, WMAPE, RMSPE, RMdSPE).....	656
2.4. Метрики качества на основе относительных ошибок (MRAE, MdRAE, GMRAE)	689
2.5. Относительные метрики качества (RelMAE, RelRMSE)	697
2.6. Масштабированные ошибки (MASE, MdASE).....	698
2.7. Критерий Диболда–Мариано	705

**ЧАСТЬ 5. ДРУГИЕ ПОЛЕЗНЫЕ БИБЛИОТЕКИ
И ПЛАТФОРМЫ707****1. Библиотеки баевской оптимизации
hyperopt, scikit-optimize и optuna707**

1.1. Недостатки обычного поиска по сетке и случайного поиска по сетке.....	707
1.2. Знакомство с байесовской оптимизацией	708
1.3. Последовательная оптимизация по модели (Sequential model-based optimization – SMBO)	710
1.4. Hyperopt.....	716
1.5. Scikit-Optimize	727
1.6. Optuna	732
2. Docker	742
2.1. Введение	742
2.2. Запуск контейнера Docker.....	743
2.3. Создание контейнера Docker с помощью Dockerfile	744
3. Библиотека H2O	749
3.1. Установка пакета h2o для Python	749
3.2. Запуск кластера H2O	749
3.3. Преобразование данных во фреймы H2O	750
3.4. Знакомство с содержимым фрейма.....	751
3.5. Определение имени зависимой переменной и списка имен признаков	753
3.6. Построение модели машинного обучения.....	753
3.7. Вывод модели	754
3.8. Получение прогнозов	758
3.9. Построение ROC-кривой и вычисление AUC-ROC.....	759
3.10. Поиск оптимальных значений гиперпараметров по сетке	760
3.11. Извлечение наилучшей модели по итогам поиска по сетке.....	762
3.12. Класс H2OAutoML.....	762
3.13. Применение класса H2OAutoML в библиотеке scikit-learn	771
4. Библиотека Dask	783
4.1. Общее знакомство	783
4.2. Машинное обучение с помощью библиотеки dask-ml	792
4.3. Построение конвейера в Dask	800
5. Google Colab.....	804
5.1. Общее знакомство	804
5.2. Регистрация и создание папки проекта	804
5.3. Подготовка блокнота Colab	809

Введение

Настоящая книга является коллекцией избранных материалов из первого модуля Подписки – обновляемых в режиме реального времени материалов по применению классических методов машинного обучения в различных промышленных задачах, которые автор делает вместе с коллегами и учениками.

Автор благодарит Дмитрия Ларько за помощь в подготовке раздела по конструированию признаков в третьей части книги, Уилла Керсена за предоставленные материалы к первому разделу пятой части книги.

Во втором томе мы разберем собственно процесс предварительной подготовки данных, обсудим некоторые метрики качества, рассмотрим ряд полезных библиотек и фреймворков.

Часть 3

План предварительной подготовки данных

1. Введение

До этого момента мы знакомились с инструментами – основными питоновскими библиотеками, классами и функциями, необходимыми для предварительной подготовки данных и построения моделей машинного обучения. Мы брали относительно простые примеры, выполняли предварительную подготовку данных и строили модели машинного обучения без глубокого понимания, зачем нужна та или иная операция предварительной подготовки и что происходит «под капотом» этой операции. В реальной практике мы так действовать не можем, нам нужен четкий план действий и глубокое понимание каждого этапа.

План предварительной подготовки данных, как правило, будет состоять из двух этапов. Первый этап – операции, которые можно выполнить до разбиения на обучающую и тестовую выборки / до цикла перекрестной проверки. Второй этап – операции, которые можно выполнить только после разбиения на обучающую и тестовую выборки / внутри цикла перекрестной проверки.

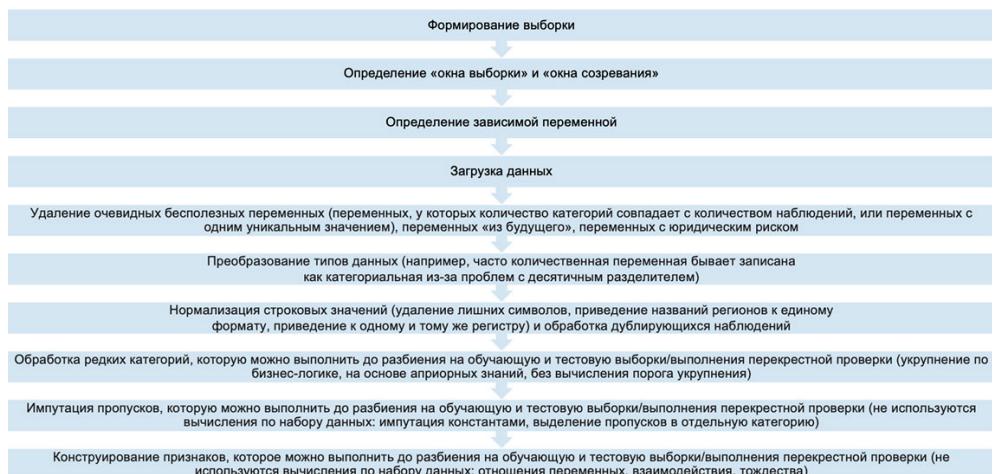
Если используются операции, использующие статистики, например укрупнение редких категорий по порогу, импутация пропусков статистиками, стандартизация, биннинг и конструирование признаков на основе статистик (frequency encoding, likelihood encoding), они должны быть осуществлены после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки.

Если мы используем случайное разбиение на обучающую и тестовую выборки и выполняем перечисленные операции до разбиения, получается, что для вычисления среднего и стандартного отклонения по каждому признаку для стандартизации, правил биннинга, частот и вероятностей положительного класса зависимой переменной в категориях признака использовались все наблюдения набора, часть из которых потом у нас войдет в тестовую выборку (по сути, выборку новых данных).

Если мы используем перекрестную проверку и выполняем перечисленные операции до перекрестной проверки, получается, что в каждом проходе перекрестной проверки для вычисления среднего и стандартного отклонения по каждому признаку для стандартизации, правил биннинга, частот и вероятностей положительного класса зависимой переменной в категориях признака использовались

все наблюдения набора, часть из которых у нас теперь находится в тестовом блоке (по сути, выборке новых данных). В таких случаях в Python используем классы `ColumnTransformer` и `Pipeline`. Случайное разбиение на обучающую и тестовую выборки и перекрестная проверка используются для сравнения конвейеров базовых моделей со значениями гиперпараметров по умолчанию. При подборе гиперпараметров лучшей практикой является комбинированная проверка, сочетающая случайное разбиение на обучающую и тестовую выборки и перекрестную проверку.

До разбиения на обучающую и тестовую выборки / до цикла перекрестной проверки



После разбиения на обучающую и тестовую выборки / внутри цикла перекрестной проверки



Рис. 1 План предварительной подготовки данных

2. Формирование выборки

2.1. ГЕНЕРАЛЬНАЯ И ВЫБОРОЧНАЯ СОВОКУПНОСТИ

Генеральная совокупность, или популяция (population), – совокупность всех объектов (единиц), относительно которых предполагается делать выводы при изучении конкретной задачи.

Генеральная совокупность состоит из всех объектов, которые имеют качества, свойства, интересующие исследователя. Например, в исследованиях телезрительской аудитории, проводимых компанией Mediascope, генеральной совокупностью будет население России в возрасте 4 лет и старше, проживающее в городах от 100 000 и более. А в исследованиях читательской аудитории, проводимых этой же компанией, генеральной совокупностью будет уже население России в возрасте 16 лет и старше, проживающее в городах от 100 000 и более. В исследованиях политических предпочтений в преддверии президентских выборов генеральной совокупностью будет население России в возрасте 18 лет и старше (поскольку право голосовать гражданин получает с 18 лет). В банковском скоринге генеральной совокупностью считаются все потенциально возможные заемщики банка. В таких случаях принято считать, что объем генеральной совокупности у нас неизвестен.

Выборка, или выборочная совокупность (sample), – набор объектов, выбранных с помощью определенной процедуры из генеральной совокупности для участия в исследовании.

Цель проведения выборочных обследований – на основе выборки сформировать суждение обо всей генеральной совокупности.

Допустим, нам необходимо провести исследование, цель которого – убедиться, что груши на дереве созрели. Решение заключается в том, чтобы сорвать несколько груш с дерева и попробовать их. Генеральная совокупность – все груши на дереве. Выборочная совокупность – сорванные груши с дерева. Если все сорванные груши созрели, то с большой вероятностью можно сделать вывод, что и все остальные груши на дереве тоже созрели. Если необходимо узнать, все ли груши созрели во всем саду, то это будет уже другая генеральная совокупность – груши во всем саду. Исследование будет состоять в том, чтобы срывать и пробовать груши с разных деревьев.

2.2. ХАРАКТЕРИСТИКИ ВЫБОРКИ

Перечень всех единиц наблюдения генеральной совокупности с базовой информацией представляет **основу выборки. Базовая информация** – набор характеристик, известных до проведения обследования для каждого элемента основы выборки (например, фамилия, имя и отчество респондента, адрес предприятия, регион проведения интервью и другие характеристики).

Элементы отбора при формировании выборочной совокупности называются **единицами отбора**. Объект, признаки которого подлежат регистрации, называется **единицей наблюдения**. Обычно единицей наблюдения в социологических опросах является конкретный человек, который будет отвечать на вопросы анкеты. Единица наблюдения может совпадать или не совпадать с единицей

отбора. При простой случайной выборке единицы отбора и единицы наблюдения совпадают. В случае использования многоступенчатой выборки сначала отбираются регионы, потом населенные пункты, затем предприятия или адреса проживания семей (все они и будут единицами отбора), и лишь на последнем этапе будут отобраны конкретные единицы наблюдения – респонденты.

Количество элементов выборки называется **объемом (размером)** выборки.

Соответствие характеристик выборки характеристикам популяции или генеральной совокупности в целом называется **репрезентативностью**. Репрезентативность определяет, насколько возможно обобщать результаты исследования с привлечением определенной выборки на всю генеральную совокупность, из которой она была отобрана. Корректный вывод обо всей генеральной совокупности можно сделать только на основании репрезентативной выборки. Поэтому при формировании выборки должен быть такой отбор элементов, чтобы выборка была репрезентативной.

В США одним из наиболее известных исторических примеров нерепрезентативной выборки считается случай, произошедший во время президентских выборов в 1936 году.

Журнал «Литрери Дайджест», успешно прогнозировавший события нескольких предшествующих выборов, ошибся в своих предсказаниях, разослав десять миллионов пробных бюллетеней своим подписчикам, а также людям, выбранным по телефонным книгам всей страны и людям из регистрационных списков автомобилей. В 25 % вернувшихся бюллетеней (почти 2,5 миллиона) голоса были распределены следующим образом:

57 % отдавали предпочтение кандидату-республиканцу Альфу Лэндону;

40 % выбрали действующего в то время президента-демократа Франклина Рузельята.

На действительных же выборах, как известно, победил Рузельйт, набрав более 60 % голосов. Ошибка «Литрери Дайджест» заключалась в следующем: желая увеличить репрезентативность выборки, – так как им было известно, что большинство их подписчиков считают себя республиканцами, – они расширили выборку за счёт людей, выбранных из телефонных книг и регистрационных списков. Однако они не учли современных им реалий и в действительности набрали ещё больше республиканцев: во время Великой депрессии обладать телефонами и автомобилями могли себе позволить в основном представители среднего и высшего класса (то есть большинство республиканцев, а не демократов).

В нашем игрушечном примере, когда нам нужно было убедиться, что груши на дереве созрели, примером нерепрезентативной выборки были бы груши, сорванные только с одной, южной стороны дерева. А если бы нам необходимо было узнать, все ли груши созрели во всем саду, то примером нерепрезентативной выборки были бы груши, сорванные с деревьев, которые росли поблизости (допустим, мы поленились пройти в глубь сада).

Отклонение результатов оценки значений, полученных с помощью выборки, от истинных неизвестных значений в генеральной совокупности называется **ошибкой выборки**.

В выборочных обследованиях мы будем оперировать статистиками. **Статистика** – это некоторая функция от выборочных наблюдений, например минимальное значение, среднее арифметическое, стандартное отклонение и др. Допустим, минимальный вес груши, средний вес груши.

Исследование всех объектов генеральной совокупности называется **сплошным обследованием**. Наиболее точные оценки могут быть получены при сплошном наблюдении, однако могут быть сложности. Основные проблемы, возникающие при сплошном наблюдении: ограничение по времени, ограничение финансовых ресурсов, ограничение человеческих ресурсов (здесь речь идет о физических и интеллектуальных ресурсах как опрашивающих, так и опрошенных).

Понятно, что мы не можем для получения рейтингов кандидатов на пост Президента РФ физически опросить все население России в возрасте от 18 лет и старше. Однако даже если сплошное обследование можно организовать, оно не гарантирует получения надежных результатов. Примером, когда сплошное обследование потерпело неудачу, была сплошная перепись населения России 1897 г. Когда анализировалась численность населения по возрастам, то получалось, что максимальные численности (пики) имели возрасты, кратные 5 и в особенности кратные 10. Большая часть населения в те времена была неграмотна и свой возраст помнила только приблизительно, с точностью до пяти или до десяти лет. Чтобы все-таки узнать, каково было распределение по возрастам на самом деле, нужно было не увеличивать объем данных, а, наоборот, создать выборку из нескольких процентов населения и провести комплексное исследование, основанное на перекрестном анализе нескольких источников: документов, свидетельств и личных показаний. Это дало бы гораздо более точную картину, нежели сплошная перепись. Для решения проблем, возникающих при сплошном обследовании, как раз и используют выборочные обследования.

2.3. ДЕТЕРМИНИРОВАННЫЕ И ВЕРОЯТНОСТНЫЕ ВЫБОРКИ

По способу отбора выборки делятся на:

- **дeterminированные;**
- **вероятностные.**

Детерминированный отбор – выборочный метод, в котором не применяется процедура случайного отбора единиц генеральной совокупности. Этот метод основан на индивидуальных суждениях исследователя. Примерами являются экспертный отбор, квотный отбор, отбор методом «снежного кома».

Выборка по методу «снежного кома» строится следующим образом. У каждого респондента, начиная с первого, просятся контакты его друзей, коллег, знакомых, которые подходили бы под условия отбора и могли бы принять участие в исследовании. Таким образом, за исключением первого шага, выборка формируется с участием самих объектов исследования. Метод часто применяется, когда необходимо найти и опросить труднодоступные группы респондентов (например, респондентов, имеющих высокий доход, респондентов, принадлежащих к одной профессиональной группе, респондентов, имеющих какие-либо схожие хобби/увлечения и т. д.).

При квотной выборке генеральная совокупность сначала разделяется на не-пересекающиеся группы. Затем пропорционально из каждой группы выбираются единицы наблюдения на основании предпочтений отбирающего. Например, интервьюер может получить задание отобрать 200 женщин и 300 мужчин возрастом от 45 до 60 лет. Это значит, что внутри каждой квоты интервьюер отбирает респондентов по своим предпочтениям.

Описанный второй шаг формирования квотной выборки относит её к детерминированному типу. Отбор элементов в квотную выборку не является случайным и может быть ненадёжным. Например, интервьюеры могут в первую очередь пытаться опрашивать тех людей, которые выглядят наиболее отзывчивыми или живут поблизости. Соответственно, менее отзывчивые люди или репонденты, живущие в труднодоступных местах, криминогенных районах, в которых интервьюер побоится опрашивать, имеют меньше шансов попасть в выборку.

Квотная выборка полезна, когда время ограничено, отсутствует основа для формирования вероятностной выборки, бюджет исследования небольшой или когда точность результатов не слишком важна.

Вероятностный отбор – выборочный метод, в котором состав выборки формируется случайным образом. В вероятностном отборе каждая единица генеральной совокупности имеет определенную вероятность включения в выборку. Нас будут интересовать вероятностные выборочные методы.

2.4. Виды, методы и способы вероятностного отбора

По виду отбора различают следующие вероятностные выборки:

- выборки с индивидуальным отбором;
- выборки с групповым отбором;
- выборки с комбинированным отбором.

Выборки с индивидуальным отбором осуществляют отбор из генеральной совокупности каждой единицы наблюдения в отдельности. Например, при обследовании удовлетворенности сотрудников предприятия размером заработной платы осуществляется отбор сотрудников.

Выборки с групповым отбором осуществляют отбор групп единиц. Например, при обследовании удовлетворенности сотрудников предприятия размером заработной платы осуществляется отбор отделов предприятия.

По методу отбора различают:

- выборки без возвращения (бесповторный отбор);
- выборки с возвращением (повторный отбор).

В выборках без возвращения (бесповторный отбор) отобранный элемент не возвращается в генеральную совокупность, из которой осуществлялся отбор. В выборках с возвращением (повторный отбор) отобранный объект возвращается в генеральную совокупность и имеет шанс быть отобранным повторно. Использование повторного метода дает большую ошибку выборки, чем использование бесповторного.

По способам отбора различают:

- простой случайный отбор;
- систематический отбор;
- вероятностно-пропорциональный отбор;
- расслоенный случайный отбор;
- кластерный (серийный) отбор.

Более подробное обсуждение этих способов выходит за рамки книги, разберем здесь лишь процедуру простого случайного отбора.

При проведении простого случайного отбора каждая единица генеральной совокупности имеет известную и равную вероятность отбора. В простом случайном отборе каждая единица отбирается независимо от другой. Для отбора используется таблица случайных чисел или компьютерная программа.

Здесь отметим, что к повторному отбору приравнивается простой случайный отбор из генеральной совокупности, объем которой неизвестен. При вычислении необходимого объема выборки для построения моделей банковского скоринга как раз предполагают, что имеет место повторный отбор.

2.5. Подходы к определению необходимого объема выборки

Необходимый объем выборки может быть известен по результатам предыдущих аналогичных исследований. Если же объем выборки неизвестен, его необходимо рассчитать.

2.5.1. Определение объема выборки согласно теории выборочных обследований

Согласно теории выборочных обследований объем необходимой выборки зависит от задаваемой точности оценки параметров, дисперсии оцениваемых параметров и способа отбора. Общее правило следующее: чем больше дисперсия оцениваемых параметров, тем больший объем выборки необходим для того, чтобы обеспечить требуемую точность. Поэтому предварительно по отобранным данным необходимо рассчитать дисперсию оцениваемых переменных. В зависимости от величины надежности выбирают значение стандартного нормального распределения.

В банковском скоринге для построения качественной модели данные о «хороших» и «плохих» клиентах максимально должны отражать поток клиентов с улицы.

Предположим, мы хотим быть уверенными на 95 %, что соотношение «хороших» и «плохих» заемщиков в обучающей выборке отражает генеральную совокупность заемщиков. В таких случаях обычно используют следующую формулу определения объема выборки для оценки генеральной доли при повторном случайном отборе (при этом предполагается, что выборка значительно меньше генеральной совокупности):

$$n = \frac{z_{\gamma}^2 w (1-w)}{\Delta_w^2},$$

где:

n – минимальный объем выборки;

z_{γ} – значение стандартного нормального распределения, определяемое в зависимости от выбранного доверительного уровня (доверительной вероятности);
 w – доля «плохих» на предварительной выборке (может быть получена, исходя из опыта имеющихся априорных знаний);

Δ_w – максимально допустимая предельная ошибка оценки доли «плохих» заемщиков (предельная ошибка выборки).

Доверительный уровень (доверительная вероятность) – это вероятность того, что генеральная доля лежит в границах полученного доверительного интервала: выборочная доля (w) ± ошибка выборки (Δ_w). Доверительный уровень устанавливает сам исследователь в соответствии со своими требованиями к надежности полученных результатов. Чаще всего применяются доверительные уровни, равные 0,95 или 0,99.

Допустим, среди 700 клиентов предварительной выборки 50 оказались «плохими». Оценка доли «плохих» клиентов, по имеющимся данным, для построения модели составила около 0,07, или 7 %. При таком значении оценки доли предположим, мы хотим ошибиться не более чем на 10 %, что будет соответствовать допустимой предельной ошибке оценки доли 0,007, т. е. $(50 / 700) * 0,1$. При этом задаем 95%-ную доверительную вероятность. В этом случае z -значение стандартного нормального закона распределения составит около 1,96. Вычис-

$$\text{ляем минимальный объем выборки: } n = \frac{z^2 w (1-w)}{\Delta_w^2} = \frac{3,84 \cdot 0,07 \cdot 0,93}{0,000049} = 5102.$$

На практике чаще всего нет возможности сформировать предварительную выборку и значение w неизвестно. В таком случае w принимается за 0,5 (самый консервативный сценарий). При этом значении размер ошибки выборки будет максимальен.

Допустим, оценка доли «плохих» клиентов неизвестна, принимаем ее за 0,5. При таком значении оценки доли предположим, мы хотим ошибиться не более чем на 10 %. При этом задаем 95%-ную доверительную вероятность. Получаем

$$n = \frac{z^2 w (1-w)}{\Delta_w^2} = \frac{3,84 \cdot 0,5 \cdot 0,5}{0,000049} = 19592.$$

На собеседованиях часто просят рассчитать объем выборки с определенной предельной ошибкой. Например, рассчитайте объем выборки, предельная ошибка которой составит 4 %. При этом мы принимаем, что доверительный уровень равен 95 %, а генеральная совокупность значительно больше выборки.

$$\text{Применяем знакомую формулу } n = \frac{z^2 w (1-w)}{\Delta_w^2} = \frac{3,84 \cdot 0,5 \cdot 0,5}{0,0016} = 600.$$

При этом не забывайте, вам помимо обучающей выборки еще нужно отложить наблюдения для проверки, здесь, соответственно, нужно выделить ситуацию, когда вы строите и проверяете базовую модель (вам нужен тестовый набор), и ситуацию, когда вы строите модели, настраивая гиперпараметры (вам нужен валидационный набор для настройки гиперпараметров, роль валидационного набора могут выполнять проверочные блоки перекрестной проверки и тестовый набор для итоговой оценки качества).

Когда предполагается, что выборка сопоставима с генеральной совокупностью (обычное явление при опросах организаций в B2B-исследованиях), формула определения объема выборки для оценки генеральной доли будет выглядеть несколько иначе, в ней будет фигурировать показатель объема генеральной совокупности.

$$n = \frac{\frac{z_\gamma^2 w (1-w)}{\Delta_w^2}}{1 + \frac{\frac{z_\gamma^2 w (1-w)}{\Delta_w^2} - 1}{N}}.$$

2.5.2. Определение объема выборки согласно правилу NEPV

В практике банковского скоринга для ответа на вопрос об объеме выборки часто используют правило «Number of Events Per Variable» (количество событий на одну переменную, NEPV), сформулированное Фрэнком Харреллом.

Для задачи бинарной классификации оно связывает минимальный объем выборки с количеством «событий» – наблюдений в миноритарной (наименьшей по размеру) категории зависимой переменной и количеством признаков, поданным на вход модели. Согласно этому правилу, необходимо взять количество наблюдений в обучающей выборке, относящихся к миноритарной категории зависимой переменной (в кредитном скоринге это «плохие» заемщики). Это число наблюдений нужно разделить на количество заданных признаков. Для логистической регрессии на один параметр должно приходиться не менее 20 событий, при построении дерева решений CHAID на один признак должно приходиться не менее 50 событий, а для модели случайного леса, градиентного бустинга, SVM и нейронной сети на одну независимую переменную должно приходиться не менее 200 событий.

Для задачи регрессии мы просто берем количество наблюдений и делим на количество признаков, и для линейной регрессии на один параметр должно приходиться не менее 20 наблюдений, для дерева решений CHAID (в тех случаях, когда реализация алгоритма позволяет решать задачу регрессии) на один признак должно приходиться не менее 50 наблюдений, для случайного леса и других сложных моделей на один признак должно приходиться не менее 200 наблюдений. По мнению Фрэнка Харрелла, для дерева решений CART правило NEPV невозможно сформулировать из-за высокой нестабильности метода и склонности к переобучению.

Если правило выполняется для обучающей выборки, то объем выборки для обучения является достаточным. В противном случае необходимо либо увеличить объем выборки, либо сократить количество признаков, подаваемых на вход модели. Затем вся та же самая процедура применяется к тестовой выборке, если правило выполняется, объем выборки для проверки достаточен.

Мы решаем задачу классификации, и у нас есть общая выборка объемом 4424 клиента, классифицированных на два класса: класс *Остается* (2492 клиента) и класс *Уходит* (1932 клиента). Мы разбили выборку на обучающую и тестовую и получили следующее распределение классов в выборках: *Остается* (1746 клиентов) и *Уходит* (1334 клиента).

Выясняем, достаточен ли объем выборки для обучения. У нас 1746 оставшихся клиентов, 1334 ушедших клиента и 9 независимых переменных. Ми-

норитарный класс – класс *Уходит*. Проверяя выполнение правила NEPV, мы получаем $1334 / 9 = 148,2$. Наша выборка обеспечивает достаточное количество событий на одну переменную, и мы можем использовать эту выборку для обучения.

Выясняем, достаточен ли объем выборки для проверки. У нас 746 оставшихся клиентов, 598 ушедших клиентов. Проверяя выполнение правила NEPV, мы получаем $598 / 9 = 66,4$. Наша выборка обеспечивает достаточное количество событий на одну переменную, и мы можем использовать эту выборку для проверки.

Если выполняется перекрестная проверка, роль обучающей выборки выполняет набор обучающих блоков, а роль тестовой выборки выполняет тестовый блок.

Опять же напомним: данная схема работает для построения базовой модели без подбора гиперпараметров.

2.5.3. Определение объема выборки с помощью кривых обучения и валидации

Кроме того, необходимый объем выборки можно определить с помощью кривых обучения и валидации. Их можно построить с помощью функции `learning_curve()`. Она запускает перекрестную проверку на наборах данных разного объема. Генератор перекрестной проверки разбивает весь набор данных k раз на обучающую выборку и тестовую выборку. В итоге для набора соответствующего размера мы получаем метрику для обучающей выборки, усредненную по k проходам, и метрику для тестовой выборки, усредненную по k проходам.

```
sklearn.model_selection.learning_curve(estimator, X, y, groups=None,
                                         train_sizes=array([0.1, 0.33, 0.55, 0.78, 1.]),
                                         cv=None, scoring=None, n_jobs=None, shuffle=False,
                                         random_state=None, return_times=False)
```

Модель машинного обучения, которая подвергается проверке
 Массив признаков
 Массив меток
 Стратегия перекрестной проверки
 Метрика качества
 Инкрементное обучение для ускорения его
 Количество используемых ядер процессора
 Перемешивание данных
 Стартовое значение генератора псевдослучайных чисел
 Возвращает время обучения и оценки

Идентификатор групп (только для стратегии проверки GroupKFold)
 Относительное (если переданы числа с плавающей точкой) или абсолютное (если переданы целые числа) количество наблюдений, которое будет использоваться для формирования кривой обучения. Обратите внимание, что для классификации количество наблюдений должно быть достаточно большим для адекватного представления классов. По умолчанию используется `np.linspace(0.1, 1.0, 5)`

Рис. 2 Параметры функции `learning_curve()`

В результате функция `learning_curve()` возвращает:

- `train_sizes_abs` – количество наблюдений, использованное для построения кривой обучения;
- `train_scores` – значения метрики на обучающих выборках перекрестной проверки;
- `test_scores` – значения метрики на тестовых выборках перекрестной проверки;
- `fit_times` – время, затраченное на обучение, в секундах;
- `score_times` – время, затраченное на оценку качества, в секундах.

Давайте загрузим необходимые библиотеки, классы и функции.

```
# импортируем библиотеки, классы и функции
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
```

Теперь загрузим данные. Данные записаны в файле *Response.csv*. Исходная выборка содержит записи о 30 259 клиентах, классифицированных на два класса: 0 – отклика нет (17 170 клиентов) и 1 – отклик есть (13 089 клиентов).

По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- категориальный признак *Ипотечный кредит [mortgage]*;
- категориальный признак *Страхование жизни [life_ins]*;
- категориальный признак *Кредитная карта [cre_card]*;
- категориальный признак *Дебетовая карта [deb_card]*;
- категориальный признак *Мобильный банк [mob_bank]*;
- категориальный признак *Текущий счет [curr_acc]*;
- категориальный признак *Интернет-доступ к счету [internet]*;
- категориальный признак *Индивидуальный займ [perloan]*;
- категориальный признак *Наличие сбережений [savings]*;
- категориальный признак *Пользование банкоматом за последнюю неделю [atm_user]*;
- категориальный признак *Пользование услугами онлайн-маркетплейса за последний месяц [markpl]*;
- количественный признак *Возраст [age]*;
- количественный признак *Давность клиентской истории [cus_leng]*;
- категориальная зависимая переменная *Отклик на предложение новой карты [response]*.

```
# загружаем данные
data = pd.read_csv('Data/Response.csv', sep=';')
data.head(3)
```

	mortgage	life_ins	cre_card	deb_card	mob_bank	curr_acc	internet	perloan	savings	atm_user	markpl	age	cus_leng	response
0	No	No	No	No	No	18.0	less than 3 years	No						
1	Yes	Yes	NaN	NaN	Yes	No	NaN	NaN	NaN	Yes	No	18.0	NaN	Yes
2	Yes	Yes	NaN	Yes	No	No	No	No	No	No	Yes	NaN	from 3 to 7 years	Yes

Формируем массив меток и массив признаков, создаем списки переменных и трансформеры, с помощью класса *ColumnTransformer* сопоставляем транс-

формеры со списками переменных, создаем конвейер для логистической регрессии и конвейер для градиентного бустинга, каждой модели машинного обучения будет соответствовать своя последовательность моделей предварительной подготовки данных.

```
# создаем массив меток и массив признаков
y = data.pop('response')

# создаем списки категориальных
# и количественных столбцов
categorical_features = data.select_dtypes(
    include='object').columns.tolist()
numeric_features = data.select_dtypes(
    exclude='object').columns.tolist()

# создаем трансформеры
numeric_transformer_logreg = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

numeric_transformer_boost = Pipeline([
    ('imputer', SimpleImputer(strategy='median'))
])

categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='constant',
                               fill_value='missing')),
    ('onehot', OneHotEncoder(sparse=False,
                            handle_unknown='ignore'))
])

# сопоставляем трансформеры спискам переменных
# для логистической регрессии
preprocessor_logreg = ColumnTransformer([
    ('num', numeric_transformer_logreg, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])

# сопоставляем трансформеры спискам переменных
# для градиентного бустинга
preprocessor_boost = ColumnTransformer([
    ('num', numeric_transformer_boost, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])

# формируем итоговый конвейер
pipe_logreg = Pipeline([
    ('preprocessor', preprocessor_logreg),
    ('Logreg', LogisticRegression(solver='lbfgs', max_iter=400))
])

pipe_boost = Pipeline([
    ('preprocessor', preprocessor_boost),
    ('boost', GradientBoostingClassifier(
        random_state=42))])

```

Теперь пишем функцию, которая будет строить графики на основе результатов, возвращенных функцией `learning_curve()`.

```
# пишем функцию, которая строит графики
# по результатам функции learning_curve()
def plot_learning_curve(estimator,
                       title,
                       X,
                       y,
                       axes=None,
                       ylim=None,
                       cv=None,
                       n_jobs=None,
                       train_sizes=np.linspace(.1, 1.0, 5)):
    """
    Строит 3 графика: кривые обучения и валидации, кривую зависимости между
    объемом обучающих данных и временем обучения, кривую зависимости между
    временем обучения и оценкой качества.
    """
```

Параметры

```
-----
estimator : модель машинного обучения для проверки.
title : заголовок диаграммы.
X : массив признаков.
y : массив меток.
axes : задаем область рисования (Axes) для построения кривых.
ylim : задает минимальное и максимальное значения по оси у.
cv : стратегия перекрестной проверки.
n_jobs : количество используемых ядер процессора.
train_sizes : абсолютное или относительное количество наблюдений.
"""
```

```
if axes is None:
    _, axes = plt.subplots(1, 3, figsize=(20, 5))
axes[0].set_title(title)
if ylim is not None:
    axes[0].set_ylim(*ylim)
axes[0].set_xlabel("Обучающие наблюдения")
axes[0].set_ylabel("Оценка")
train_sizes, train_scores, test_scores, fit_times, _ = \
    learning_curve(estimator, X, y, cv=cv,
                   scoring='roc_auc', n_jobs=n_jobs,
                   train_sizes=train_sizes,
                   return_times=True)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
fit_times_mean = np.mean(fit_times, axis=1)
fit_times_std = np.std(fit_times, axis=1)

# строим кривые обучения и валидации
axes[0].grid()
axes[0].fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="г")
```

```

axes[0].fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
                     color="g")
axes[0].plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Средняя оценка на обуч. блоках")
axes[0].plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Средняя оценка на тест. блоках")
axes[0].legend(loc="best")

# строим график зависимости между объемом
# обучающих данных и временем обучения
axes[1].grid()
axes[1].plot(train_sizes, fit_times_mean, 'o-')
axes[1].fill_between(train_sizes, fit_times_mean - fit_times_std,
                     fit_times_mean + fit_times_std, alpha=0.1)
axes[1].set_xlabel("Обучающие наблюдения")
axes[1].set_ylabel("Время обучения")
axes[1].set_title("Масштабируемость модели")
# строим график зависимости между временем
# обучения и оценкой качества
axes[2].grid()
axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
axes[2].fill_between(fit_times_mean, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1)
axes[2].set_xlabel("Время обучения")
axes[2].set_ylabel("Оценка")
axes[2].set_title("Качество модели")
return plt

```

А сейчас будем строить графики кривых обучения и валидации.

```

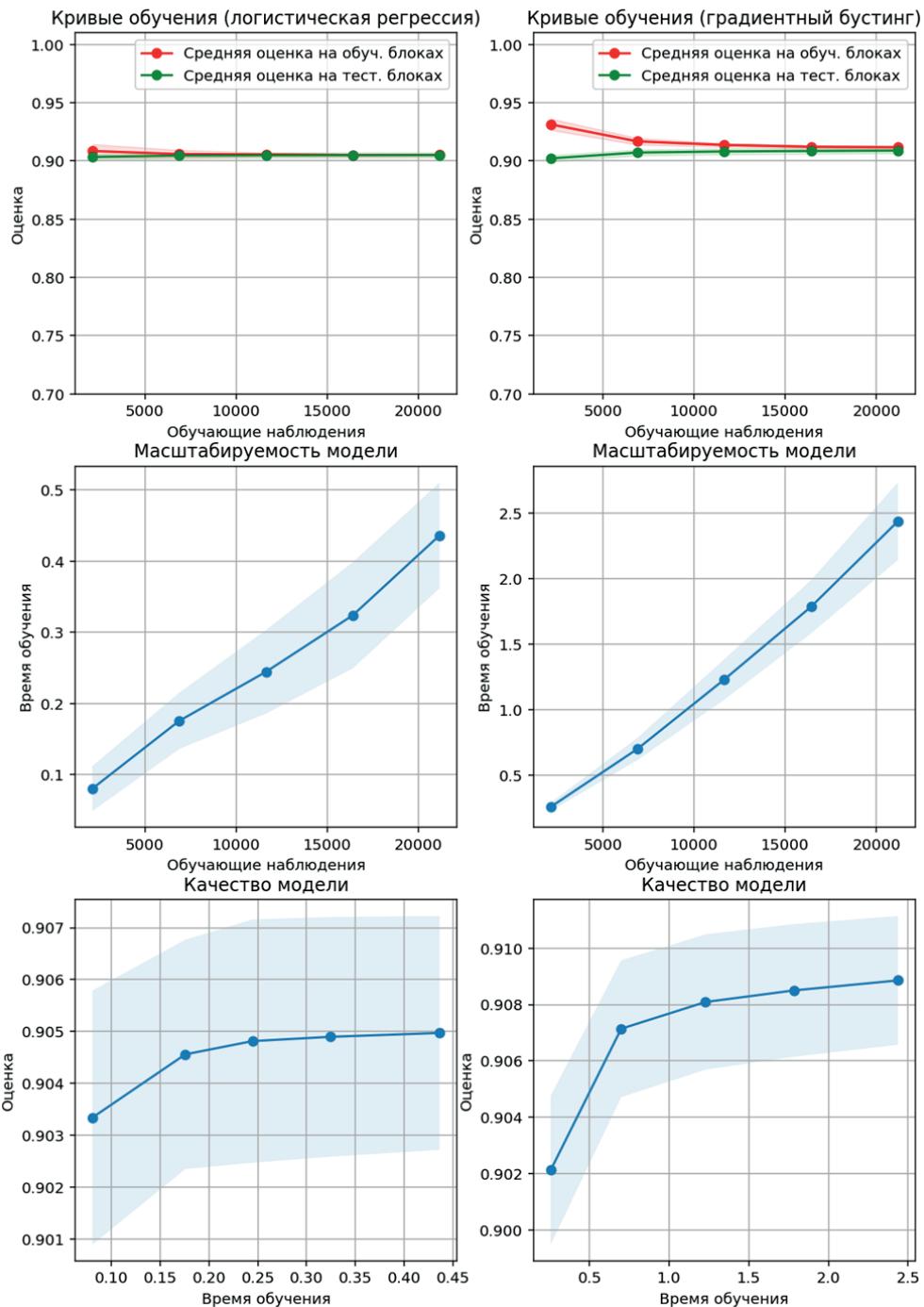
# задаем сетку и размеры графиков
fig, axes = plt.subplots(3, 2, figsize=(10, 15))
# задаем стратегию перекрестной проверки
cv = ShuffleSplit(n_splits=20, test_size=0.3, random_state=42)

# задаем заголовок
title = "Кривые обучения (логистическая регрессия)"
# строим графики для логистической регрессии
plot_learning_curve(pipe_logreg, title, data, y, axes=axes[:, 0],
                     ylim=(0.7, 1.01),
                     cv=cv, n_jobs=4)

# задаем заголовок
title = "Кривые обучения (градиентный бустинг)"
# строим графики для градиентного бустинга
plot_learning_curve(pipe_boost, title, data, y, axes=axes[:, 1],
                     ylim=(0.7, 1.01),
                     cv=cv, n_jobs=4)

# выводим графики
plt.show()

```



Для каждой модели машинного обучения мы выводим по три графика: кривые обучения и валидации, кривую зависимости между объемом обучающих

данных и временем обучения, кривую зависимости между временем обучения и оценкой качества.

График кривых обучения и валидации для логистической регрессии показывает, что независимо от размера обучающего набора оценка на обучении практически совпадает с оценкой на тесте.

График кривых обучения и валидации для градиентного бустинга показывает, что по мере увеличения размера обучающего набора оценка на обучении приближается к оценке на тесте (уменьшается гэп между ними), т. е. происходит уменьшение переобучения. При этом мы видим, что обе модели практически эквивалентны по качеству (как метрика качества у нас используется AUC) и увеличения оценки на тесте по мере роста объема данных практически не происходит, что может говорить о достаточноном объеме данных, и, скорее всего, улучшения качества можно добиться не за счет увеличения данных, а за счет тщательно продуманного конструирования признаков.

Графики кривой зависимости между объемом обучающих данных и временем обучения показывают, что обучение градиентного бустинга занимает больше времени. В том случае, когда мы используем набор размером 5000 наблюдений, обучение логистической регрессии занимает в среднем 0,12 секунды, а обучение градиентного бустинга – 0,5 секунды.

Графики кривой зависимости между временем обучения и оценкой качества показывают, сколько времени обучения требуется для получения соответствующего качества на teste. Например, мы можем принять, что разница между AUC 0,905 и AUC 0,908 не столь критична, чтобы обучаться 2,5 секунды вместо 0,4 секунды.

Теперь возьмем другой набор данных. Данные записаны в файле *StateFarm_missing.csv*. Исходная выборка содержит записи о 8293 клиентах, классифицированных на два класса: 0 – отклика нет на предложение автостраховки (7462 клиента) и 1 – отклик есть на предложение автостраховки (831 клиент). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- количественный признак *Пожизненная ценность клиента* [*Customer Lifetime Value*];
- категориальный признак *Вид страхового покрытия* [*Coverage*];
- категориальный признак *Образование* [*Education*];
- категориальный признак *Тип занятости* [*EmploymentStatus*];
- категориальный признак *Пол* [*Gender*];
- количественный признак *Доход клиента* [*Income*];
- количественный признак *Размер ежемесячной автостраховки* [*Monthly Premium Auto*];
- количественный признак *Количество месяцев со дня подачи последнего страхового требования* [*Months Since Last Claim*];
- количественный признак *Количество месяцев с момента заключения страхового договора* [*Months Since Policy Inception*];
- количественный признак *Количество открытых страховых обращений* [*Number of Open Complaints*];
- количественный признак *Количество полисов* [*Number of Policies*];
- категориальная зависимая переменная *Отклик на предложение автостраховки* [*Response*].

Давайте загрузим данные, опять создадим конвейеры и построим графики кривых обучения и валидации.

```
# загружаем данные
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')
data.head(3)
```

	Customer Lifetime Value	Coverage	Education	EmploymentStatus	Gender	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN	32.0	5.0	NaN	1.0	No
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN	13.0	42.0	NaN	NaN	No
2	NaN	NaN	NaN	Employed	F	48767.0	108.0	NaN	38.0	0.0	NaN	No

```
# формируем массив меток и массив признаков
y = data.pop('Response')

# создаем списки количественных
# и категориальных столбцов
cat_features = data.select_dtypes(
    include='object').columns.tolist()
num_features = data.select_dtypes(
    exclude='object').columns.tolist()

# сопоставляем трансформеры спискам переменных
# для логистической регрессии
preprocessor_lr = ColumnTransformer([
    ('num', numeric_transformer_logreg, num_features),
    ('cat', categorical_transformer, cat_features)
])

# сопоставляем трансформеры спискам переменных
# для градиентного бустинга
preprocessor_bst = ColumnTransformer([
    ('num', numeric_transformer_boost, num_features),
    ('cat', categorical_transformer, cat_features)
])

# формируем итоговый конвейер
pipe_lr = Pipeline([
    ('preprocessor', preprocessor_lr),
    ('logreg', LogisticRegression(solver='lbfgs',
                                  max_iter=400))
])

pipe_bst = Pipeline([
    ('preprocessor', preprocessor_bst),
    ('boost', GradientBoostingClassifier(
        random_state=42))
])

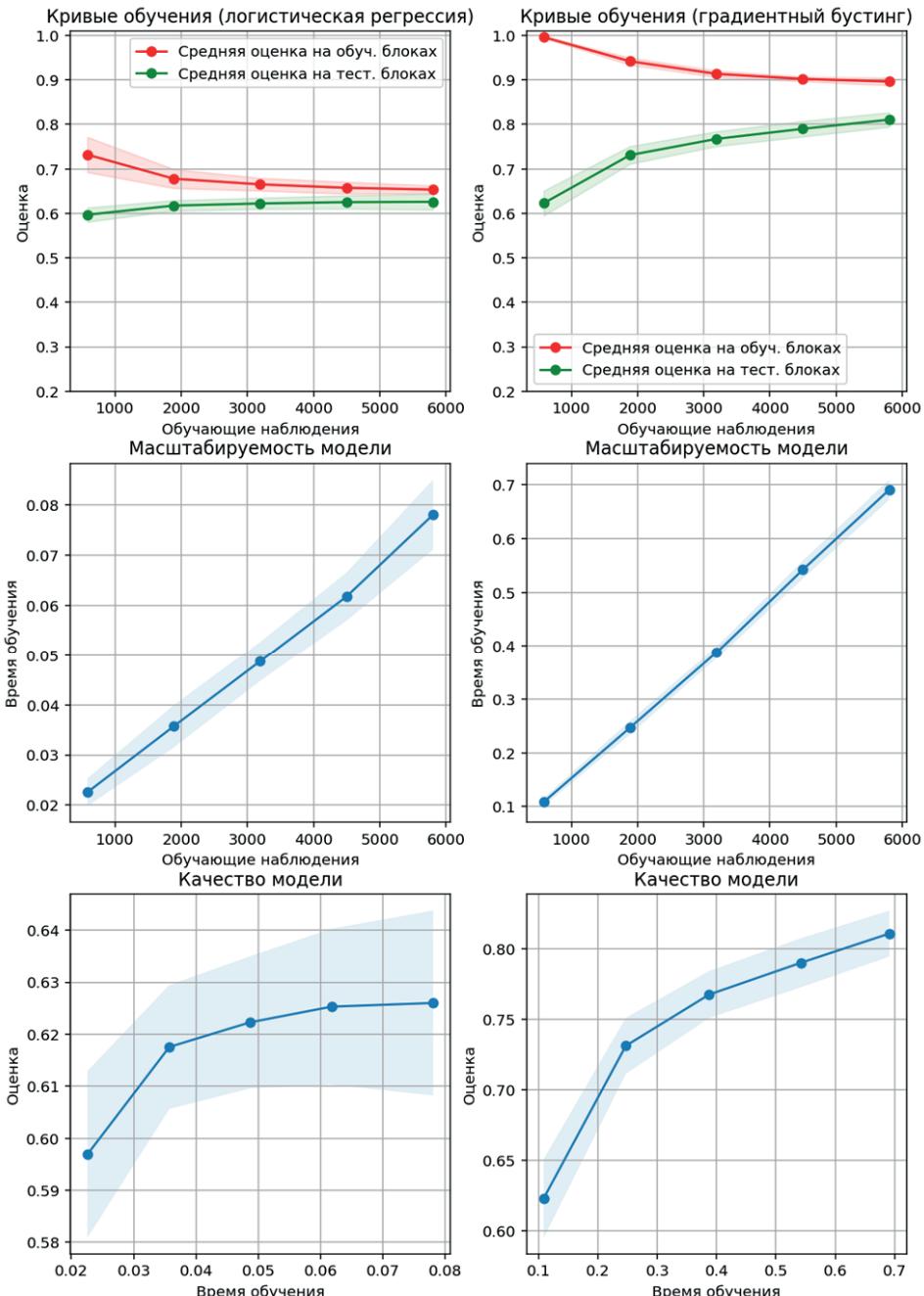
# задаем сетку и размеры графиков
fig, axes = plt.subplots(3, 2, figsize=(10, 15))

# задаем заголовок
title = "Кривые обучения (логистическая регрессия)"
# строим графики для логистической регрессии
plot_learning_curve(pipe_lr, title, data, y, axes=axes[:, 0],
                     ylim=(0.2, 1.01),
                     cv=cv, n_jobs=4)

# задаем заголовок
title = "Кривые обучения (градиентный бустинг)"
# строим графики для градиентного бустинга
```

```
plot_learning_curve(pipe_bst, title, data, y, axes=axes[:, 1],
                    ylim=(0.2, 1.01),
                    cv=cv, n_jobs=4)
```

```
# выводим графики
plt.show()
```



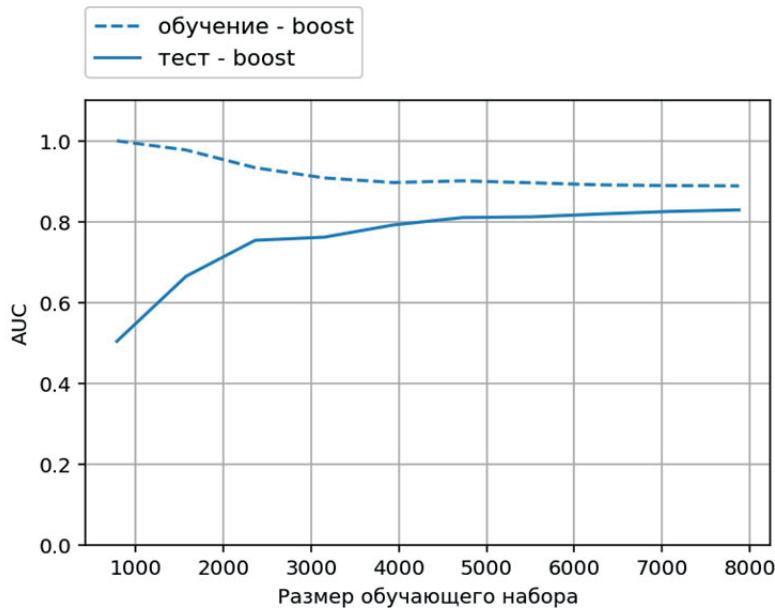
Вновь для каждой модели машинного обучения мы выводим по три графика: кривые обучения и валидации, кривую зависимости между объемом обучающих данных и временем обучения, кривую зависимости между временем обучения и оценкой качества.

Здесь уже графики кривых обучения и валидации для логистической регрессии и бустинга показывают, что по мере увеличения размера обучающего набора оценка на обучении сближается с оценкой на тесте. При этом если для логистической регрессии оценка на обучении практически сблизивается с оценкой на тесте, то для градиентного бустинга гэп хоть и уменьшается, но все еще остается значительным, что может говорить о недостаточном объеме данных, и, скорее всего, улучшения качества можно добиться за счет увеличения данных. При этом мы видим, что с точки зрения оценки на тесте модель градиентного бустинга существенно лучше модели логистической регрессии.

На практике часто пользуются упрощенной версией функции `plot_learning_curve_simple()`.

```
# импортируем класс KFold
from sklearn.model_selection import KFold

# пишем упрощенную версию plot_learning_curve()
def plot_learning_curve_simple(est, X, y):
    # получаем наборы для обучения, значения метрик
    training_set_size, train_scores, test_scores = learning_curve(
        est, X, y, train_sizes=np.linspace(.1, 1, 10), scoring='roc_auc',
        cv=KFold(20, shuffle=True, random_state=1))
    # извлекаем имя последнего этапа итогового конвейера -
    # название модели машинного обучения
    estimator_name = est.steps[-1][0]
    # строим кривые обучения и валидации
    line = plt.plot(training_set_size, train_scores.mean(axis=1), '--',
                    label="обучение - " + estimator_name)
    plt.plot(training_set_size, test_scores.mean(axis=1), '-',
            label="тест - " + estimator_name, c=line[0].get_color())
    # задаем координатную сетку
    plt.grid()
    # подписываем ось x
    plt.xlabel("Размер обучающего набора")
    # подписываем ось y
    plt.ylabel("AUC")
    # задаем пределы значений оси y
    plt.ylim(0, 1.1)
    # задаем расположение легенды
    plt.legend(loc=(0, 1.05), fontsize=11)
    # применяем упрощенную версию plot_learning_curve()
    plot_learning_curve_simple(pipe_bst, data, y)
```



3. Определение «окна выборки» и «окна созревания»

Прогнозные модели разрабатываются, исходя из предположения «прошлое отражает будущее». На основе этого предположения мы анализируем поведение прошлых клиентов, чтобы спрогнозировать поведение будущих клиентов. Для того чтобы корректно выполнить этот анализ, нужно собрать необходимые данные о клиентах за определенный период времени, а затем осуществить мониторинг клиентов в течение другого определенного периода времени, оценив, были они «хорошими» или «плохими». Собранные данные (независимые переменные) наряду с соответствующей классификацией (зависимой переменной, которая принимает значение *Хороший* или *Плохой*) составят основу для разработки прогнозной модели. Ключевыми терминами здесь будут «окно выборки» и «окно созревания».

«Окно выборки» – это период времени, в течение которого те или иные клиенты отбираются для анализа (попадают в выборку).

«Окно созревания» – это период времени, в течение которого клиент, собственно говоря, имел возможность себя проявить, и мы присваиваем клиенту соответствующий класс зависимой переменной.

Допустим, сделано предположение, что новый клиент получил кредит в определенный период времени (например, 1 января 2014 г.). В некоторый момент времени в будущем (например, через 90 дней) нам нужно определить, был этот клиент «хорошим» или «плохим» (чтобы классифицировать поведение).

Если мы возьмем все кредиты, выданные в январе 2014 года, и посмотрим на их качество с момента открытия до декабря 2015 года, окном выборки будет январь 2014 года, а окном созревания – 24 месяца, период с января 2014 года по декабрь 2015 года.

В некоторых случаях, таких как мошенничество и банкротство, временной период уже известен или предопределен. Но тем не менее вышеописанный анализ полезно выполнить для того, чтобы определить идеальное окно созревания.

Мы можем попробовать несколько подходов к определению окна выборки и окна созревания.

В ряде случаев окно созревания определяется требованиями регулирующих органов, т. е. горизонтом прогнозирования модели. Например, Базель II требует 12-месячное окно созревания, поэтому вероятность дефолта в моделях, построенных в соответствии с требованиями Базель II, определяется по 12-месячному окну созревания. Более поздние инициативы, такие как МСФО (IFRS) 9.3, предлагают использовать более длительный горизонт прогнозирования убытков, вплоть до срока действия кредита.

Второй подход сопоставляет окно созревания со сроком кредита. Например, если срок автокредита составляет четыре года, оценка заявок по этому кредиту должна основываться на четырехлетнем окне созревания. Логично, что отношения, в которые вступает кредитор, продолжаются четыре года, поэтому риск должен оцениваться в течение четырехлетнего периода. Этот подход хорошо работает для срочных кредитов. Если срок займа очень большой (скажем, более 8–10 лет), то можно применить третий подход, основанный на опреде-

лении окна созревания по данным винтажного анализа, чтобы получить более короткое окно созревания и использовать более свежие данные.

Для револьверных кредитов (например, возобновляемых кредитных карт) и кредитов с длительным сроком (например, ипотека) лучше рассмотреть третий подход. Мы берем ежемесячные или ежеквартальные отчеты по когортному или винтажному анализу, имеющиеся в любом отделе кредитных рисков, анализируем динамику по платежам или просрочкам и строим график появления «плохих» случаев (случаев просрочки 90+, отказа от услуг) с течением времени.

Классический пример винтажного анализа для просроченной задолженности свыше 90 дней и 12-квартального (3-летнего) окна созревания приведен на рисунке ниже. Данные, выделенные жирным шрифтом, показывают текущий статус просрочки платежа на определенный отчетный период времени.

	Срок кредита (в кварталах)											
	1 Qtr	2 Qtr	3 Qtr	4 Qtr	5 Qtr	6 Qtr	7 Qtr	8 Qtr	9 Qtr	10 Qtr	11 Qtr	12 Qtr
«Плохой» = 90 дней просрочки												
Дата открытия кредита												
Q1 13	0.00%	0.05%	1.10%	2.40%	2.80%	3.20%	3.50%	3.70%	3.80%	3.85%	3.85%	3.86%
Q2 13	0.00%	0.06%	1.20%	2.30%	2.70%	3.00%	3.30%	3.50%	3.60%	3.60%	3.60%	3.60%
Q3 13	0.00%	0.03%	0.90%	2.80%	3.20%	3.60%	4.10%	4.30%	4.40%	4.40%	4.45%	
Q4 13	0.00%	0.03%	1.00%	2.85%	3.20%	3.50%	3.80%	4.00%	4.00%	4.10%		
Q1 14	0.00%	0.04%	1.00%	2.20%	2.40%	2.70%	2.90%	2.90%	2.90%	4.10%		
Q2 14	0.00%	0.05%	1.20%	2.50%	2.90%	3.30%	3.50%					
Q3 14	0.00%	0.04%	1.30%	2.60%	3.00%	3.35%						
Q4 14	0.00%	0.08%	1.40%	2.60%	3.00%							
Q1 15	0.00%	0.02%	0.09%	2.20%								
Q2 15	0.00%	0.08%	1.50%									
Q3 15	0.00%	0.05%										
Q4 15	0.00%											

Рис. 3 Пример когортного/винтажного анализа

Таблица имеет достаточно простую интерпретацию. Так, на первой строчке 2,8 % заемщиков, получивших кредит в первом квартале 2013 г., выпали в просрочку более 90 дней через 5 кварталов.

Несмотря на то что показатели просрочек схожи, мы видим, что в некоторых когортах показатели просрочек выше при одинаковой зрелости. Это нормальное явление, поскольку маркетинговые кампании, экономические циклы, изменения кредитной политики и другие факторы могут влиять на качество кредитов.

У нас есть несколько сценариев для построения кривой созревания просрочек по представленным данным.

Первый сценарий заключается в использовании значений по диагонали, выделенных жирным шрифтом. Здесь показаны самые последние показатели просроченной задолженности. В портфелях, винтажность которых может отличаться по качеству, это может привести к появлению кривых, которые не очень полезны (винтажная кривая не будет плавно расти, поскольку могут быть «провалы»), и, следовательно, это лучший вариант для продуктов, в которых качество заявителя и кредитного счета довольно стабильно, например ипотека.

В случае колебания показателей есть два дополнительных сценария, которые могут помочь сгладить числа и дать нам более реалистичную диаграмму роста

просрочек 90+ с течением времени. Например, мы можем использовать средние значения по последним четырем когортам, как показано овалами в рисунке, или мы можем выбрать одну отдельную когорту, например кредитные счета, открытые в первом квартале 2013 года, как показано прямоугольной рамкой.

Длительность винтажного анализа	Срок кредита (в кварталах)											
	1 Qtr		2 Qtr		3 Qtr		4 Qtr		5 Qtr		6 Qtr	
Q1 13	0.00%	0.05%	1.10%	2.40%	2.80%	3.20%	3.50%	3.70%	3.80%	3.85%	3.85%	3.86%
Q2 13	0.00%	0.06%	1.20%	2.30%	2.70%	3.00%	3.30%	3.50%	3.60%	3.60%	3.60%	3.60%
Q3 13	0.00%	0.03%	0.90%	2.80%	3.20%	3.60%	4.10%	4.30%	4.40%	4.45%		
Q4 13	0.00%	0.03%	1.00%	2.85%	3.20%	3.50%	3.80%	4.00%	4.10%			
Q1 14	0.00%	0.04%	1.00%	2.20%	2.40%	2.70%	2.90%	2.90%	4.10%			
Q2 14	0.00%	0.05%	1.20%	2.50%	2.90%	3.30%	3.50%					
Q3 14	0.00%	0.04%	1.30%	2.60%	3.00%	3.35%						
Q4 14	0.00%	0.08%	1.40%	2.60%	3.00%							
Q1 15	0.00%	0.02%	0.09%	2.20%								
Q2 15	0.00%	0.08%	1.50%									
Q3 15	0.00%	0.05%										
Q4 15	0.00%											

Рис. 4 Пример когортного/винтажного анализа: подходы к построению кривой созревания

Ниже показан график накопленного уровня просрочек 90+ для двух когорт: кредитных счетов, открытых в 1-м квартале 2013 года, и кредитных счетов, открытых в 1-м квартале 2014 года.

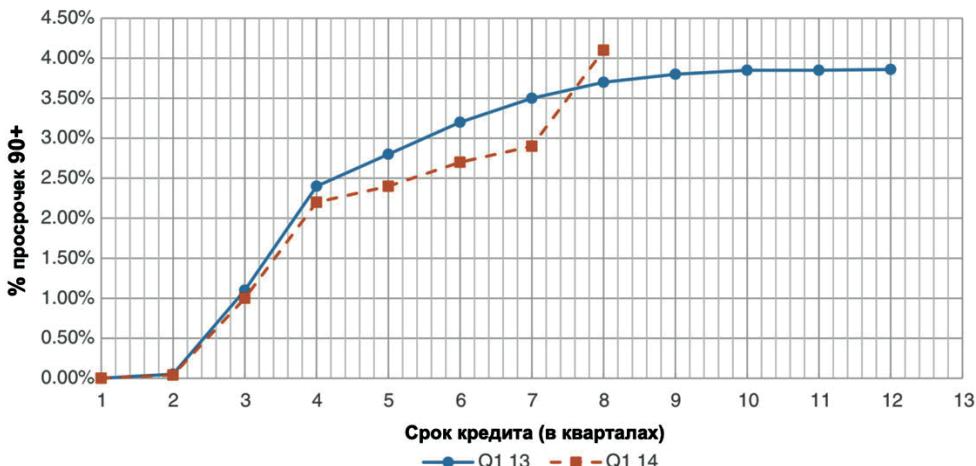


Рис. 5 Кривая винтажей

Перед нами пример типичного портфеля кредитных карт, где просрочки 90+ начинаются уже в самом начале и их количество растет с течением времени. Для счетов, открытых в первом квартале 2013 года, мы видим, что уровень просрочек 90+ быстро рос в первые несколько кварталов, а затем стабилизировался, когда срок кредитного счета стал приближаться к 10 кварталам. Для счетов, открытых во втором квартале 2014 года, мы видим, что уровень просрочек 90+ активно растет и о стабилизации говорить еще рано.

Мы сформируем выборку по такому периоду, когда можно считать, что уровень «плохих» случаев стабилизируется или когорта стала зрелой (т. е. когда накопленный уровень «плохих» случаев начинает выравниваться). В примере на рисунке выше хорошим окном выборки станут кредитные счета, которые были открыты 10–12 кварталов назад и дают 11-квартальное окно созревания.

Проектирование выборки по созревшей когорте осуществляется для того, чтобы минимизировать вероятность неправильной классификации клиентов (мы предоставляем всем клиентам достаточно времени, чтобы они могли стать «плохими») и убедиться в том, что определение «плохого» клиента, полученное по нашей выборке, не будет недооценивать итоговый ожидаемый процент «плохих» случаев. Например, если применительно к нашему примеру выборка будет спроектирована по кредитным счетам, открытым 4 квартала назад, мы увидим, что 2,4 % клиентов классифицированы как «плохие», однако уровень просрочек 90+ по-прежнему растет.

Поэтому некоторые кредитные счета, которые на самом деле являются «плохими», будут ошибочно помечены как «хорошие», если выборка будет спроектирована по 4-квартальному периоду.

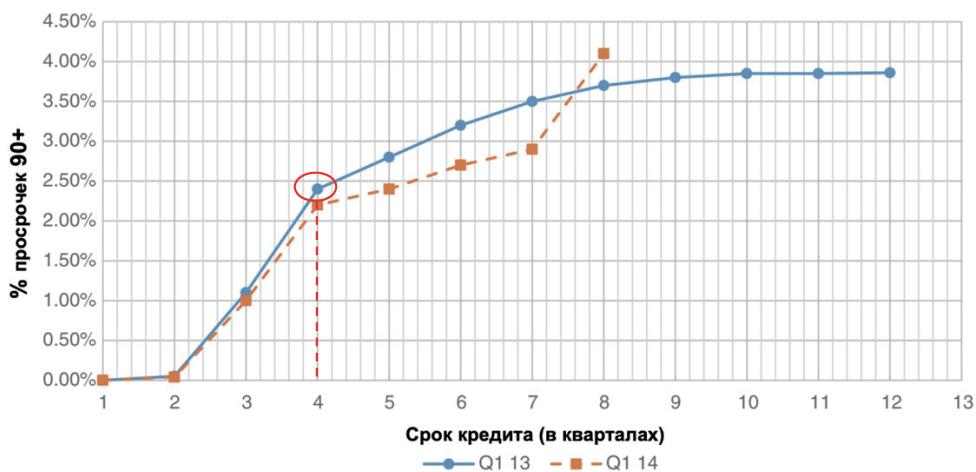


Рис. 6 Кривая винтажей: слишком короткое окно созревания

Временной горизонт «созревания» зависит от продукта, определения «плохого» клиента и конкурентной среды. Счета по кредитным картам считаются «зрелыми» после 12–18 месяцев, счета по автокредитам – обычно через 2–4 года, в то время как счета сроком от 4 до 5 лет считаются минимально допустимыми для разработки скоринговой карты по ипотечным кредитам. Поведенческие скоринговые карты предусматривают окно созревания из расчета 6–12 месяцев. Коллекторские модели строятся, как правило, на данных одного месяца, но все чаще компании строят такие карты для более коротких временных интервалов – до двух недель, чтобы облегчить разработку более подходящих способов взыскания долгов. По очевидным причинам счета по просрочкам 30+ будут становиться зрелыми быстрее, чем счета по просрочкам 90+. В условиях нестабильной макроэкономической ситуации, внутренних социально-экономических кризисов, как правило, счета по просрочкам также становятся зрелыми быстрее.

4. Определение зависимой переменной

Выбор зависимой переменной определяется целью построения прогнозной модели. Цели могут быть общими, например сокращение потерь по новым кредитным счетам, и конкретными, например сокращение числа дефолтов по одобренным заявкам в течение 6 месяцев после принятия положительного решения.

Зависимая переменная может быть количественной и категориальной.

В скоринге заявок зависимая переменная будет категориальной: погасит заемщик кредит («хороший») или не погасит («плохой»). Обычно к категории «плохой» относят клиентов, имеющих просроченную задолженность 90 дней и более. Этот период определяется требованиями банковского надзора. В соответствии с соглашением Базель II дефолт должника считается произошедшим, когда имело место одно или оба из следующих событий: банк считает, что должник не в состоянии полностью погасить свои кредитные обязательства перед банком без принятия банком таких мер, как реализация обеспечения (если такое имеется); должник более чем на 90 дней просрочил погашение любых существенных кредитных обязательств перед банком.

Разумеется, банк может строить различные скоринговые карты с разными значениями зависимой переменной, вводя дополнительные критерии определения «плохого» и «хорошего» заемщика, а также меняя срок просрочки платежей. Примерами зависимой переменной могут быть наличие просроченной задолженности более 30 дней, 60 дней, 90 дней и более по одному кредиту на текущий момент или худший статус за все время кредитной истории, размер просроченной задолженности, глубина просрочки.

5. Загрузка данных из CSV-файлов и баз данных SQL

Загрузка CSV-файлов выполняется с помощью функции `pd.read_csv()`, а загрузку XLS-файлов можно выполнить с помощью функции `pd.read_excel()`.

Часто при загрузке CSV-файлов у нас возникают проблемы с кодировкой. Питоновский пакет `chardet` помогает определить тип кодировки прочитываемого файла.

```
# импортируем класс UniversalDetector библиотеки chardet
from chardet.universaldetector import UniversalDetector

# создаем экземпляр класса UniversalDetector
detector = UniversalDetector()

# определяем кодировку файла Credit OTP.csv
with open('Data/Credit OTP.csv', 'rb') as fh:
    for line in fh:
        detector.feed(line)
        if detector.done:
            break
    detector.close()
print(detector.result)

{'encoding': 'windows-1251', 'confidence': 0.8897840802899516, 'language': 'Russian'}
```

Нередко нам приходится работать с большими наборами данных. Мы можем ускорить функцию `pd.read_csv()` за счет распараллеливания. Используем эвристики, используемые во фреймворке AutoML от Лаборатории искусственного интеллекта Сбербанка LAMA. Обратите внимание, что параметры `skiprows`, `nrows`, `index_col`, `header`, `names`, `chunksize` игнорируются и функция будет более требовательна к оперативной памяти. Давайте импортируем необходимые библиотеки, классы, модули и напишем функцию `read_csv()`, которая будет использовать ряд вспомогательных предварительно написанных функций.

```
# импортируем необходимые библиотеки, классы и модули
import numpy as np
import pandas as pd
from joblib import Parallel, delayed
from copy import copy
import warnings
import os

def get_filelen(fname):
    """
Получает длину csv-файла.

Параметры:
    fname: имя файла.

Возвращает:
    длина файла.
    """

    cnt_lines = -1
    with open(fname, "rb") as fin:
        for line in fin:
```

```
if len(line.strip()) > 0:  
    cnt_lines += 1  
  
return cnt_lines  
  
def get_batch_ids(arr, batch_size):  
    """  
        Генератор последовательностей, разбитых на батчи.  
  
    Параметры:  
        arr: последовательность.  
        batch_size: размерность.  
    Возвращает:  
        Батчи.  
    """  
  
    n = 0  
    while n < len(arr):  
        yield arr[n : n + batch_size]  
        n += batch_size  
  
def read_csv_batch(file, offset, cnt, **read_csv_params):  
    """  
        Читает батч данных из csv-файла.  
  
    Параметры:  
        file: путь к файлу.  
        offset: отступ.  
        cnt: количество строк для чтения.  
        **read_csv_params: вспомогательные параметры.  
    Возвращает:  
        Прочитанные данные.  
    """  
  
    read_csv_params = copy(read_csv_params)  
    if read_csv_params is None:  
        read_csv_params = {}  
  
    try:  
        usecols = read_csv_params.pop("usecols")  
    except KeyError:  
        usecols = None  
  
    header = pd.read_csv(file, nrows=0, **read_csv_params).columns  
    with open(file, "rb") as f:  
        f.seek(offset)  
        data = pd.read_csv(f, header=None, names=header,  
                           chunksize=None, nrows=cnt,  
                           usecols=usecols, **read_csv_params)  
  
    return data  
  
def get_file_offsets(file, n_jobs=None, batch_size=None):  
    """  
        Получает отступы.  
  
    Параметры:  
        file: путь к файлу.  
        n_jobs: количество ядер процессора для распараллеливания.  
        batch_size: размер батча.  
    Возвращает:  
        Кортеж отступов.  
    """
```

```

assert (
    n_jobs is not None or batch_size is not None
), "One of n_jobs or batch size should be defined"

lens = []
with open(file, "rb") as f:
    header_len = len(f.readline())
    length = 0
    for row in f:
        if len(row.strip()) > 0:
            lens.append(length)
            length += len(row)

lens = np.array(lens, dtype=np.int64) + header_len

if batch_size:
    indexes = list(get_batch_ids(lens, batch_size))
else:
    indexes = np.array_split(lens, n_jobs)

offsets = [x[0] for x in indexes]
cnts = [x.shape[0] for x in indexes]

return offsets, cnts

def _check_csv_params(**read_csv_params):
    """
    Проверяет параметры для функции `read_csv`.

    Параметры:
        **read_csv_params: прочитывает параметры.

    Возвращает:
        Новые параметры.
    """
    for par in ["skiprows", "nrows", "index_col",
               "header", "names", "chunksize"]:
        if par in read_csv_params:
            read_csv_params.pop(par)
            warnings.warn(
                "Parameter {0} will be ignored in parallel mode".format(par),
                UserWarning,
            )
    return read_csv_params

def read_csv(file, n_jobs=4, **read_csv_params):
    """
    Прочитывает данные из csv-файла.

    Параметры:
        file: путь к файлу.
        n_jobs: количество ядер процессора для распараллеливания.
        **read_csv_params: вспомогательные параметры.

    Возвращает:
        Прочитанные данные.
    """
    if n_jobs == 1:
        return pd.read_csv(file, **read_csv_params)

    if n_jobs == -1:
        n_jobs = os.cpu_count()

    _check_csv_params(**read_csv_params)
    offsets, cnts = get_file_offsets(file, n_jobs)

```

```
with Parallel(n_jobs) as p:
    res = p(
        delayed(read_csv_batch)(file, offset=offset, cnt=cnt,
                               **read_csv_params)
        for (offset, cnt) in zip(offsets, cnts)
    )
res = pd.concat(res, ignore_index=True)

return res
```

Теперь применим нашу функцию.

```
%time
data = read_csv('Data/paribas_train.csv', n_jobs=1)
CPU times: user 1.07 s, sys: 120 ms, total: 1.19 s
Wall time: 1.19 s
```

Библиотека pandas может считать данные из любой базы данных SQL, которая поддерживает адаптеры данных Python в рамках интерфейса Python DB-API. Чтение выполняется с помощью функций `pandas.read_sql()` и `pandas.read_sql_query()`, а запись в базу данных SQL выполняется с помощью метода `.to_sql()` объекта DataFrame.

Для иллюстрации программный код, приведенный ниже, считывает данные о котировках акций из файлов `msft.csv` и `aapl.csv`. Затем он подключается к файлу базы данных SQLite3. Если файл не существует, он создается «на лету». Затем программный код записывает данные MSFT в таблицу под названием STOCK_DATA. Если таблица не существует, она также будет создана. Если она уже существует, все данные заменяются данными о котировках акций MSFT. Наконец, программный код добавляет в эту таблицу данные о котировках акций AAPL.

```
# импортируем библиотеку SQLite
import sqlite3

# считываем данные о котировках акций из CSV-файла
msft = pd.read_csv('Data/msft.csv')
msft['Symbol']='MSFT'
aapl = pd.read_csv('Data/aapl.csv')
aapl['Symbol']='AAPL'

# создаем подключение
connection = sqlite3.connect('Data/stocks.sqlite')
# .to_sql() создаст базу SQL для хранения датафрейма в указанной таблице.
# параметр if_exists задает действие, которое нужно выполнить в том случае,
# если таблица уже существует ('fail' - выдать ошибку ValueError, 'replace' - # удалить та-
# блицу перед вставкой новых значений, 'append' - вставить
# новые значения в существующую таблицу)
msft.to_sql('STOCK_DATA', connection, if_exists='replace') aapl.to_sql('STOCK_DATA', con-
nection, if_exists='append')

# подтверждаем отправку данных в базу и закрываем подключение
connection.commit()
connection.close()
```

Чтобы убедиться в создании данных, можно открыть файл базы данных с помощью такого инструмента, как SQLite Database Browser (доступен по адресу <https://github.com/sqlitebrowser/sqlitebrowser>). Рисунок ниже показывает несколько записей в файле базы данных.

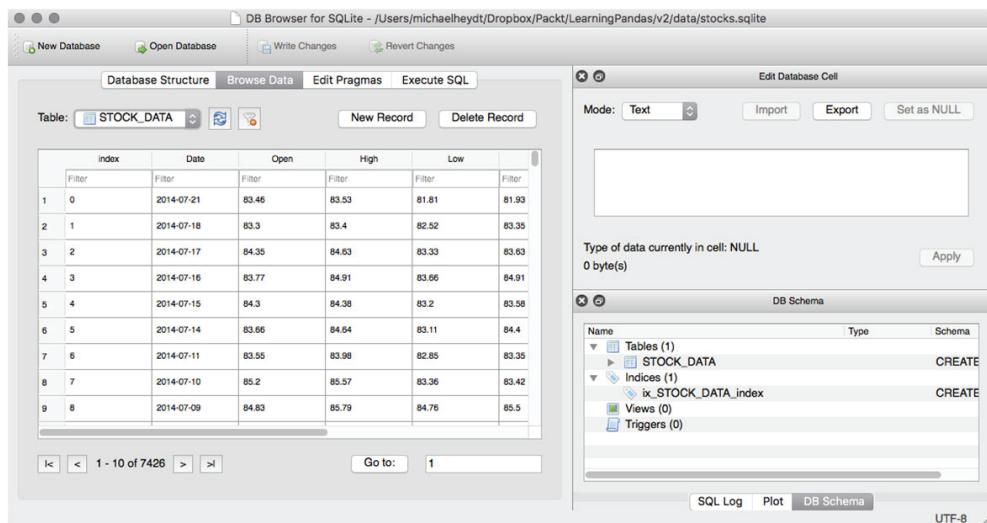


Рис. 7 SQLite Database Browser

Данные из базы данных SQL можно прочитать с помощью функции `pandas.read_sql()`. Следующий программный код демонстрирует выполнение запроса к файлу `stocks.sqlite` с помощью SQL и сообщает об этом пользователю.

```
# подключаемся к файлу базы данных
connection = sqlite3.connect('Data/stocks.sqlite')

# запрос всех записей в STOCK_DATA # возвращает датафрейм
# index_col задает столбец, который нужно сделать # индексом датафрейма
stocks = pd.read_sql("SELECT * FROM STOCK_DATA;",
                     connection, index_col='index')

# закрываем подключение
connection.close()

# выводим первые 5 наблюдений в извлеченных данных
stocks[:5]
```

	Date	Open	High	Low	Close	Volume	Symbol
index							
0	7/21/2014	83.46	83.53	81.81	81.93	2359300	MSFT
1	7/18/2014	83.30	83.40	82.52	83.35	4020800	MSFT
2	7/17/2014	84.35	84.63	83.33	83.63	1974000	MSFT
3	7/16/2014	83.77	84.91	83.66	84.91	1755600	MSFT
4	7/15/2014	84.30	84.38	83.20	83.58	1874700	MSFT

Кроме того, для отбора столбцов еще можно использовать условие WHERE в SQL. Чтобы продемонстрировать это, следующий программный код отбирает записи, в которых количество проторгованных акций MSFT превышает 29200100.

```
# открываем подключение
connection = sqlite3.connect('Data/stocks.sqlite')
# создаем строку-запрос
query = "SELECT * FROM STOCK_DATA WHERE Volume>29200100 AND Symbol='MSFT';"
# выполняем запрос
items = pd.read_sql(query, connection, index_col='index')

# выводим результат запроса
items
```

	Date	Open	High	Low	Close	Volume	Symbol
--	------	------	------	-----	-------	--------	--------

index

1081	5/21/2010	42.22	42.35	40.99	42.00	33610800	MSFT
1097	4/29/2010	46.80	46.95	44.65	45.92	47076200	MSFT
1826	6/15/2007	89.80	92.10	89.55	92.04	30656400	MSFT
3455	3/16/2001	47.00	47.80	46.10	45.33	40806400	MSFT
3712	3/17/2000	49.50	50.00	48.29	50.00	50860500	MSFT

Для этой же операции можно воспользоваться функцией `pandas.read_sql_query()`.

```
# можно воспользоваться функцией pandas.read_sql_query()
items2 = pd.read_sql_query(
    "SELECT * FROM STOCK_DATA WHERE Volume>29200100 AND Symbol='MSFT';",
    connection,
    index_col='index')
# закрываем подключение
connection.close()

# выводим результат запроса
items2
```

	Date	Open	High	Low	Close	Volume	Symbol
--	------	------	------	-----	-------	--------	--------

index

1081	5/21/2010	42.22	42.35	40.99	42.00	33610800	MSFT
1097	4/29/2010	46.80	46.95	44.65	45.92	47076200	MSFT
1826	6/15/2007	89.80	92.10	89.55	92.04	30656400	MSFT
3455	3/16/2001	47.00	47.80	46.10	45.33	40806400	MSFT
3712	3/17/2000	49.50	50.00	48.29	50.00	50860500	MSFT

Итоговым моментом является то, что большая часть программного кода в этих примерах была программным кодом SQLite3. Библиотека `pandas` в этих примерах используется лишь тогда, когда нужно применить метод

`.to_sql()`, функции `pandas.read_sql()` и `pandas.read_sql_query()`. Они принимают объект подключения, который может быть любым адаптером данных, совместимым с интерфейсом Python DB-API, поэтому вы можете работать с любой информацией базы данных, просто создав соответствующий объект подключения. Программный код на уровне `pandas` остается неизменным для любой поддерживаемой базы данных.

6. Удаление бесполезных переменных, переменных «из будущего», переменных с юридическим риском

Переменные, у которых количество категорий совпадает с количеством наблюдений, или переменные с одним уникальным значением (переменные-константы) бесполезны для анализа, и поэтому их удаляют самыми первыми.

Также необходимо удалить самыми первыми переменные «из будущего». Простой пример – мы предсказываем стоимость квадратного метра жилья. В нашем распоряжении есть исторические данные о реализованных сделках, среди признаков – время экспозиции квартиры. Однако когда нам нужно будет применить нашу модель для оценки стоимости новой квартиры, выставленной на продажу, у нас по этой квартире не будет данных о ее экспозиции.

Рисунки также несут переменные, в отношении которых мы знаем, что они не фиксировались в течение всего периода сбора исторических данных, или мы предполагаем, что в будущем не сможем получить информацию по этим переменным легальным способом.

Например, МФО фиксировала признак *Служба в армии* при выдаче микрокредита, а затем после корректировки правил кредитной политики отказалась от этого признака. У нас есть данные, где в течение первых 8 месяцев исторических данных признак *Служба в армии* фиксировался, а в последующие 4 месяца исторических данных он перестал фиксироваться (значения записаны как пропуски) и в новых данных его не будет.

Здесь можно привести пример, когда компания DoubleData фиксировала данные о характеристиках пользователей соцсетей, чтобы по ним оценить кредитоспособность или склонность к мошенничеству. Речь шла о социальном капитале (количество друзей, количество друзей друзей, количество неактивных друзей, количество друзей с дефолтом, уже доказано: чем больше у клиента друзей, находящихся в дефолте, тем больше его вероятность дефолта), активности (в рабочее время / в нерабочее время, днём/ночью, в выходные/будние дни), уникальности контента.

Ниже приведен профиль типичного фродстера в сети «ВКонтакте». Человек был осужден по ст. 159 УК РФ. Мошенничество с кредитами, обман клиентов и пр.

Всем отличного настроения

День рождения: 5 октября
Родной город: Омск
Семейное положение: не женат
Языки: Русский

Скрыть подробную информацию

Контактная информация

Город: Омск

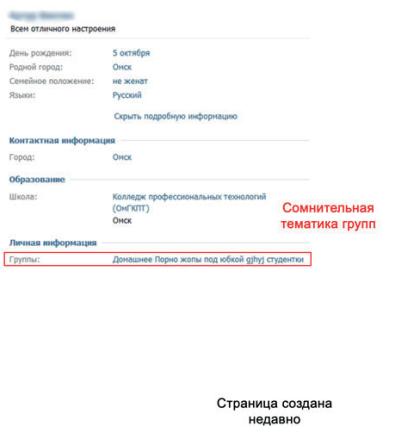
Образование

Школа: Колледж профессиональных технологий (ОмГХПТ)
Омск

Личная информация

Группы: Домашнее Перво жены под юбкой g7nuz студентки

Сомнительная тематика групп



Отсутствие личных, уникальных фотографий, карточек (если размещать на странице фотографии с монетами или банкнотами, это повлечет финансовый успех)



Большое количество «собачек» в друзьях



Страница создана недавно

Рис. 8 Профиль фродстера в сети «ВКонтакте»

Однако затем законность получения данных оспорила компания Mail.ru, которая владеет сетью «ВКонтакте», и данными DoubleData нельзя было уже воспользоваться.

Также рекомендуется внимательно включать в скоринговые модели переменные сегментации (регионы продаж, канал продаж, тип продукта), поскольку они часто содержат не всегда наблюдаемые факторы (маркетинг, кредитная политика, данные продаж и т. д.) и зависят от них. Мы часто можем отказаться от выдачи кредита в том или ином регионе, отказаться от того или иного продукта.

7. Преобразование типов переменных и знакомство со шкалами переменных

Преобразование типов переменных – один из первых этапов предварительной подготовки данных. Операции по преобразованию типов переменных не будут эффективными, если нет четкого понимания, какой тип шкалы у переменной, поскольку именно шкала определяет тип переменной.

Шкала – правило, определяющее, каким образом в процессе измерения каждому изучаемому объекту ставится в соответствие некоторое число или символы.

Это правило включает в себя три следующих вопроса.

- Можем ли мы вычислить точные расстояния между значениями? Можем ли мы сказать, на сколько и во сколько раз одно значение больше/меньше другого?
- Можем ли мы упорядочить значения по тому или иному критерию?
- Можем ли мы сказать, сколько наблюдений в каждом значении?

Выделяют количественные (непрерывные) и качественные (дискретные) шкалы.

7.1. Количественные (непрерывные) шкалы

Переменная с количественной (непрерывной) шкалой – это переменная, которая может принимать бесконечное (неисчислимое) количество значений. Все переменные с количественными шкалами являются характеристиками, которые количественно описывают продукт. Примерами непрерывных переменных являются возраст, температура, доход. Мы можем вычислить средний возраст, среднюю температуру и средний доход. Количественная шкала позволяет утвердительно ответить на все три вышеупомянутых вопроса. Возьмем переменную *Доход* со значениями 90 000, 90 000, 10 000, 10 000, 40 000, 10 000. Мы можем вычислить точные расстояния между значениями. Мы можем узнать, на сколько и во сколько раз одно значение больше/меньше другого, обратите внимание: второй тип сравнения не всегда возможен. Человек с доходом 40 000 рублей богаче человека с доходом 10 000 рублей на 30 000 рублей, или в 4 раза. Мы можем упорядочить значения по возрастанию дохода: 10 000, 40 000, 90 000. Мы можем сказать, что у нас 2 человека с доходом 90 000 рублей, 3 человека с доходом 10 000 рублей, один человек с доходом 40 000 рублей.

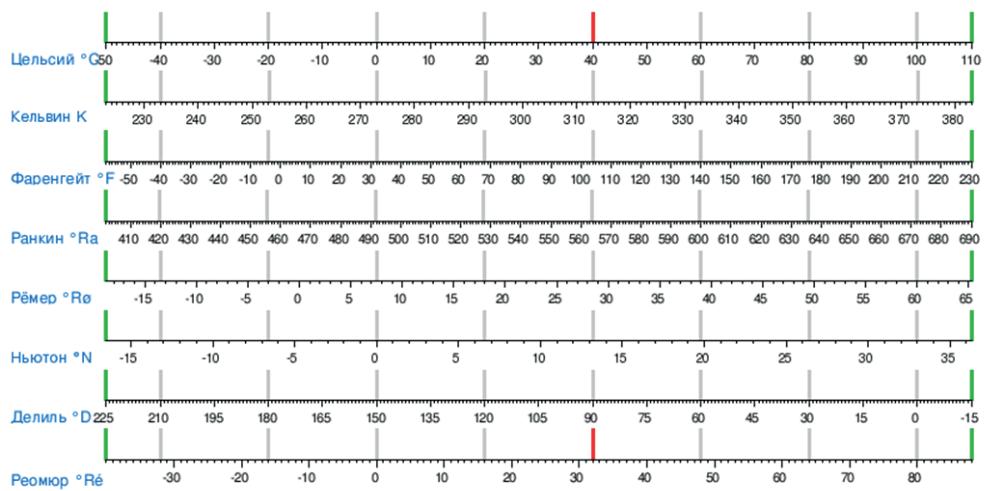
Среди количественных шкал выделяют:

- шкалу интервалов;
- шкалу отношений;
- абсолютную шкалу.

Шкала интервалов состоит из одинаковых интервалов и имеет условную нулевую точку (точку отсчета).

Примером шкалы интервалов могут служить шкалы для измерения температуры по Цельсию и Фаренгейту. Ноль по шкале Фаренгейта определяется

по самоподдерживающейся температуре смеси воды, льда и хлорида аммония (соответствует примерно $-17,8^{\circ}\text{C}$). Ноль шкалы Цельсия установлен таким образом, что температура тройной точки воды равна $0,01^{\circ}\text{C}$.



$$40^{\circ}\text{C} = 313.15\text{ K} = 104^{\circ}\text{F} = 563.67^{\circ}\text{Ra} = 28.5^{\circ}\text{Rø} = 13.2^{\circ}\text{N} = 90^{\circ}\text{D} = 32^{\circ}\text{R}$$

Рис. 9 Диаграмма перевода температур

Шкала интервалов позволяет сказать, насколько одно значение больше другого, но не позволяет сказать, во сколько раз оно больше. Например, повысив температуру с 1°C до 20°C , мы можем сказать, что температура 20°C на 19°C больше 1°C , но не можем сказать, что температура 20°C в 20 раз больше, чем 1°C . Из школьного курса физики вспомним, что температура среды (например, воздуха) определяется энергией молекул, составляющих эту среду. Для идеального газа внутренняя энергия равна сумме кинетических энергий его молекул, которая, в свою очередь, пропорциональна абсолютной температуре в кельвинах. Очевидно, что, например, при «двадцатикратном» нагреве с 1°C до 20°C абсолютная температура изменится всего в $(273 + 20) / (273 + 1) = 1,069$ раза. Ноль по шкале Цельсия соответствует 273 К. Другим примером шкал этого типа являются шкалы календарного времени.

Шкала отношений отличается от шкалы интервалов тем, что имеет естественную нулевую точку. Она позволяет сказать, насколько одно значение больше другого и во сколько раз оно больше. Примером шкалы отношений может служить переменная *Возраст*: мы знаем, что расстояние между 25 и 30 в два раза меньше, чем расстояние между 30 и 40, 30-летний на 5 лет старше 25-летнего.

Шкалы большинства физических величин (длина, масса, сила, давление, скорость и др.) являются шкалами отношений. При этом единица измерения в этих шкалах может быть произвольной. Например, возраст можно измерять в годах, месяцах, неделях. Длину мы можем измерять в километрах, милях, лье.

Абсолютная шкала, помимо естественной нулевой точки, имеет еще и естественную общепринятую единицу измерения.

Пример абсолютной шкалы – абсолютная шкала температуры, или шкала Кельвина. Ноль этой шкалы отвечает полному прекращению движения моле-

кул, т. е. самой низкой температуре, а единицей измерения является кельвин, который равен $1/273,16$ части термодинамической температуры тройной точки воды. Как и шкала отношений, абсолютная шкала также позволяет сказать, насколько одно значение больше другого и во сколько раз оно больше.

Резюмируя, можно сказать, что переменная с количественной шкалой – самая информативная переменная, у нас есть информация о расстояниях между значениями, можем упорядочить значения, можем сказать, сколько наблюдений принадлежат конкретному значению.

7.2. Качественные (дискретные) шкалы

Переменная с качественной (дискретной) шкалой – это переменная, которая может принимать одно значение из ограниченного и обычно фиксированного набора возможных значений. Каждое из возможных значений часто называется еще уровнем (level). Все переменные с качественными шкалами являются характеристиками, которые качественно описывают продукт. Примерами переменных с качественной (дискретной) шкалой являются пол, штат, социальный класс, уровень благосостояния, группа крови, гражданство, образование. Из-за дискретного характера шкалы вычисление среднего значения переменной не имеет смысла, мы не можем вычислить средний пол или среднюю сферу занятости.

Качественные (дискретные) шкалы бывают порядковыми и номинальными.

7.2.1. Порядковая шкала

Порядковая шкала позволяет утвердительно ответить только на два последних вопроса.

Пример переменной с порядковой шкалой – переменная Уровень дохода. Она имеет уровни *Низкий*, *Средний*, *Высокий*. Мы можем сказать, сколько у нас наблюдений в каждом уровне. Мы можем упорядочить уровни: *Низкий*, *Средний*, *Высокий*. Человек с уровнем *Средний* богаче человека с уровнем *Низкий*, а человек с уровнем *Высокий* богаче человека с уровнем *Средний*, но на сколько точно богаче, мы сказать не можем. Таким образом, порядковая шкала будет менее информативной, чем количественная: у нас исчезает информация о расстояниях между значениями, но мы по-прежнему можем упорядочить значения, можем сказать, сколько наблюдений принадлежат конкретному значению.

7.2.2. Номинальная шкала

Номинальная шкала позволяет утвердительно ответить только на последний вопрос.

Пример переменной с номинальной шкалой – переменная Сфера деятельности. Она имеет уровни *Строительство*, *Транспорт* и *Металлургия*. Мы можем сказать, сколько у нас наблюдений в каждом уровне. Однако мы не можем сказать, что уровень *Строительство* хуже/лучше/меньше/больше уровня *Металлургия*. У нас нет информации о расстояниях между уровнями. Номинальная переменная будет менее информативной, чем порядковая: мы можем лишь сказать, сколько наблюдений принадлежат конкретному значению. К переменным с номинальной шкалой относятся бинарные переменные – переменные с двумя значениями.

Ниже приведена таблица с примером, когда переменную *Образование* можно записать в трех типах шкалы.

ТИП ШКАЛЫ	Количественная	Порядковая	Номинальная
Пример переменной	Количество лет, потраченных на образование (от 0 до 20 лет)	Уровень образования (начальное, среднее, высшее)	Название университета (МГУ, МИФИ, МФТИ)
Можем сказать, насколько одно значение больше или меньше другого?	Да	Нет	Нет
Можем упорядочить значения?	Да	Да	Нет
Можем сказать, сколько наблюдений для каждого значения?	Да	Да	Да

Рис. 10 Три типа шкалы

Из-за неправильного десятичного разделителя количественная переменная может быть ошибочно записана как категориальная.

Бинарная зависимая переменная часто записывается с помощью значений 0 или 1 и прочитывается как целочисленная. В Python мы можем такую переменную не преобразовывать в категориальную (типы `object`), стратегия обработки определяется выбранным классом – классификатором или регрессором.

В Python и H2O порядковые переменные не поддерживаются, обрабатываем их как категориальные номинальные переменные (тип `object` в Python и тип `enum` в H2O) или количественные целочисленные (тип `int` в Python и H2O).

Таблица 1 Представление типов шкал в Python и H2O

	Python	H2O
Количественная	<code>int</code> <code>float</code>	<code>int</code> <code>real</code>
Порядковая	не поддерживается, представляют как <code>object</code> или как <code>int</code> <code>Categorical</code> (только для разведочного анализа)	не поддерживается, представляют как <code>enum</code> или как <code>int</code>
Номинальная	<code>object</code> <code>Categorical</code> (только для разведочного анализа)	<code>enum</code>

8. Нормализация строковых значений

Часто строковые значения переменных могут содержать лишние символы типа `&*_`.

Импортируем библиотеку pandas, прочитываем данные в датафрейм и смотрим первые пять наблюдений.

```
# импортируем библиотеки pandas и numpy
import pandas as pd
import numpy as np

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Extra_characters.csv', sep=';')
data.head()
```

	gender	marital
0	Женский	Женат
1	Мужской	Одинокий
2	Женский	Одинокий
3	Мужской	Одинокий
4	Женский	Одинокий

На первый взгляд вроде бы все в порядке, но давайте взглянем на уникальные значения переменных с помощью метода `.unique()`.

```
# создаем список переменных
cols = data.columns
# с помощью метода .unique() выводим уникальные
# значения переменных gender и marital
for col in cols:
    print(data[col].unique())
```

['Женский' 'Мужской' 'Женский&*' 'Мужской&*']
['Женат' 'Одинокий' '_Одинокий' '_Женат' 'Же&нат']

Видим лишние символы типа `&*_`.

Теперь удалим лишние символы типа `&*_` в переменных `gender` и `marital_status` с помощью метода `.str.replace()` объекта Series библиотеки pandas. Метод имеет общий вид:

```
Series.str.replace(pat, repl)
```

где

`pat` задает символ, который нужно найти

`repl` задает символ, на который нужно заменить найденный символ

```
# с помощью метода .str.replace() удаляем лишние символы
for col in cols:
```

```
data[col] = data[col].str.replace('[*&_]', '')
# выводим уникальные значения переменных
# gender и marital
for col in cols:
    print(data[col].unique())

['Женский' 'Мужской']
['Женат' 'Одинокий']
```

Видим, что лишние символы удалены.

Часто бывает необходимо выполнить транслитерацию строковых значений, например передать русские названия латиницей, потому что тот или иной пакет не поддерживает кириллицу. В этом нам поможет функция `translit()` библиотеки `transliterate`.

Давайте выполним транслитерацию строковых значений переменных `gender` и `marital_status` латиницей.

```
# импортируем библиотеку для транслитерации
from transliterate import translit
# выполняем транслитерацию
for col in cols:
    data[col] = data[col].apply(
        lambda x: translit(x, 'ru', reversed=True))
# выводим уникальные значения переменных
# gender и marital
for col in cols:
    print(data[col].unique())

['Zhenskij' 'Muzhskoj']
['Zhenat' 'Odinokij']
```

С помощью методов `str.lower()` и `str.upper()` можем задать нижний и верхний регистры соответственно.

```
# переводим строки (значения переменной gender)
# в нижний регистр
data['gender'] = data['gender'].str.lower()
# переводим строки (значения переменной marital)
# в ВЕРХНИЙ регистр
data['marital'] = data['marital'].str.upper()
data.head()
```

	gender	marital
0	zhenskij	ZHENAT
1	muzhskoj	ODINOKIJ
2	zhenskij	ODINOKIJ
3	muzhskoj	ODINOKIJ
4	zhenskij	ODINOKIJ

Часто при работе с базами данных бывает ситуация, когда у нас есть информация об именах, отчествах и фамилиях клиентов, но нет информации о поле. Давайте рассмотрим случай, когда по отчеству каждого клиента можно определить его пол.

```
# загружаем CSV-файл с ФИО клиентов,
# по которым нужно определить пол
data = pd.read_csv('Data/Gender_based_on_middle_name.csv',
                   encoding='cp1251', sep=';')
data.head(20)
```

	Клиент	Возраст	Регион	Статус
0	Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)
1	Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)

Теперь мы создадим переменную *Пол*, которая будет иметь значение *True*, если строковое значение переменной *Клиент* содержит паттерн *вна* (Викторовна, Дмитриевна), и *False* в противном случае. Для этого воспользуемся методом *.str.contains()*.

```
# создаем переменную Пол, которая будет иметь значение True,
# если строковое значение переменной Клиент содержит паттерн
# "вна" (Викторовна, Дмитриевна), и False в противном случае
data['Пол'] = data['Клиент'].str.contains('вна')
data.head(20)
```

	Клиент	Возраст	Регион	Статус	Пол
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	False
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	False
2	Абаймов Максим Дмитриевич	43	Москва- 4	Вернул(а)	False
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	True
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	True
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	True
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	True
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	False
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	False
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	True
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	True
11	Абдурасурова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	True
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	True
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	True
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	True
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	True
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	False
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	True
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	False
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)	False

Теперь переименуем категории переменной *Пол* с помощью словаря.

	Клиент	Возраст	Регион	Статус	Пол
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской
2	Абайимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)	Мужской

Еще можно было извлечь последние три символа в каждом строковом значении переменной *Клиент* и затем на основе полученных значений создать новую переменную.

```
# извлекаем последние три символа в каждом строковом
# значении переменной Клиент и затем на основе
# полученных значений создаем новую переменную
data['Пол2'] = data['Клиент'].str[-3:]
# переименуем категории переменной Пол2
d = {'вич': 'Мужской', 'вна': 'Женский'}
data['Пол2'] = data['Пол2'].map(d)
data.head(20)
```

50 ❖ План предварительной подготовки данных

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской
2	Абайимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
11	Абдурасурова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	Женский	Женский
12	Абдурахимова Алены Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский
16	Абраев Нурулан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45 лет	Екатеринбург- 8	Вернул(а)	Мужской	Мужской

Теперь с помощью метода `.str.lstrip()` удалим ненужный символ подчеркивания, с которого начинаются несколько значений переменной `Клиент`. Метод `.str.lstrip()` возвращает копию указанной строки, с начала которой (т. е. слева `l` – left) устраниены указанные символы.

```
# с помощью метода .str.lstrip() удалим ненужный символ
# подчеркивания, с которого начинаются несколько значений
# переменной Клиент, метод .str.lstrip() возвращает
# копию указанной строки, с начала (слева l - left)
# которой устраниены указанные символы
data['Клиент'] = data['Клиент'].str.lstrip('_')
data.head(20)
```

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской
1	Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской
2	Абайимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	Женский	Женский
12	Абдурахимова Алены Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45 лет	Екатеринбург- 8	Вернул(а)	Мужской	Мужской

Теперь с помощью метода `.str.rstrip()` удалим ненужные символы, которыми заканчиваются некоторые значения переменной `Возраст`. Метод `.str.rstrip()` возвращает копию указанной строки, с конца которой (справа `r - right`) устраниены указанные символы.

```
# с помощью метода .str.rstrip() удалим ненужные символы, которыми
# заканчиваются некоторые значения переменной Возраст, метод
# .str.rstrip() возвращает копию указанной строки, с конца
# (справа r - right) которой устраниены указанные символы
data['Возраст'] = data['Возраст'].str.rstrip(' лет')
data.head(20)
```

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской
1	Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской
2	Абайимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской
3	Абаумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург- 8	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45	Екатеринбург- 8	Вернул(а)	Мужской	Мужской

Теперь создадим переменную *Регион2*, мы просто извлечем цифры из переменной *Регион* с помощью метода `.str.extract()`.

```
# создадим переменную Регион2, извлекая
# цифры из переменной Регион
data['Регион2'] = data['Регион'].str.extract('(\d)', expand=True) data.head(20)
```

	Клиент	Возраст	Регион	Статус	Пол	Пол2	Регион2
0	Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской	2
1	Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской	2
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской	4
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский	4
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский	6
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский	6
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский	4
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской	8
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской	8
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский	8
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской	8
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский	4
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской	4
19	Абрамов Никита Валерьевич	45	Екатеринбург- 8	Вернул(а)	Мужской	Мужской	8

Теперь удалим последние три символа в каждом строковом значении переменной *Регион*.

```
# удаляем последние 3 символа в каждом строковом
# значении переменной Регион
data['Регион'] = data['Регион'].map(lambda x: str(x)[:-3])
data.head(20)
```

54 ❖ План предварительной подготовки данных

	Клиент	Возраст	Регион	Статус	Пол	Пол2	Регион2
0	Колесников Вячеслав Анатольевич	33	Красноярск	Вернул(а)	Мужской	Мужской	2
1	Саймурзанов Михаил Борисович	22	Красноярск	Вернул(а)	Мужской	Мужской	2
2	Абaimов Максим Дмитриевич	43	Москва	Вернул(а)	Мужской	Мужской	4
3	Абакумова Юлия Ивановна	22	Москва	Вернул(а)	Женский	Женский	4
4	Абанова Елена Владимировна	54	Санкт-Петербург	Вернул(а)	Женский	Женский	6
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург	Вернул(а)	Женский	Женский	6
6	Абдугалиева Айгуль Максутовна	27	Москва	Не вернул(а)	Женский	Женский	4
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург	Не вернул(а)	Мужской	Мужской	8
8	Абдуллин Евгений Эдуардович	22	Екатеринбург	Не вернул(а)	Мужской	Мужской	8
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский	8
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский	8
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург	Вернул(а)	Женский	Женский	8
12	Абдурахимова Алены Алимовна	57	Екатеринбург	Вернул(а)	Женский	Женский	8
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург	Вернул(а)	Женский	Женский	8
14	Аблец Юлия Сергеевна	33	Екатеринбург	Вернул(а)	Женский	Женский	8
15	Аболмасова Ирина Олеговна	38	Екатеринбург	Вернул(а)	Женский	Женский	8
16	Абраев Нурлан Мусайбекович	49	Екатеринбург	Вернул(а)	Мужской	Мужской	8
17	Абраменко Екатерина Владимировна	56	Москва	Вернул(а)	Женский	Женский	4
18	Абрамов Дмитрий Владимирович	51	Москва	Вернул(а)	Мужской	Мужской	4
19	Абрамов Никита Валерьевич	45	Екатеринбург	Вернул(а)	Мужской	Мужской	8

Удалим круглые скобки в переменной *Статус*.

```
# удаляем круглые скобки в переменной Статус
data['Статус'] = data['Статус'].str.replace('[(())]', '')
data.head(20)
```

	Клиент	Возраст	Регион	Статус	Пол	Пол2	Регион2
0	Колесников Вячеслав Анатольевич	33	Красноярск	Вернула	Мужской	Мужской	2
1	Саймурзанов Михаил Борисович	22	Красноярск	Вернула	Мужской	Мужской	2
2	Абаймов Максим Дмитриевич	43	Москва	Вернула	Мужской	Мужской	4
3	Абакумова Юлия Ивановна	22	Москва	Вернула	Женский	Женский	4
4	Абанова Елена Владимировна	54	Санкт-Петербург	Вернула	Женский	Женский	6
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург	Вернула	Женский	Женский	6
6	Абдугалиева Айгуль Максутовна	27	Москва	Не вернула	Женский	Женский	4
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург	Не вернула	Мужской	Мужской	8
8	Абдуллин Евгений Эдуардович	22	Екатеринбург	Не вернула	Мужской	Мужской	8
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернула	Женский	Женский	8
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернула	Женский	Женский	8
11	Абдурасурова Наталья Таджиловна	55	Екатеринбург	Вернула	Женский	Женский	8
12	Абдурахимова Алена Алимовна	57	Екатеринбург	Вернула	Женский	Женский	8
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург	Вернула	Женский	Женский	8
14	Аблец Юлия Сергеевна	33	Екатеринбург	Вернула	Женский	Женский	8
15	Аболмасова Ирина Олеговна	38	Екатеринбург	Вернула	Женский	Женский	8
16	Абраев Нурлан Мусайбекович	49	Екатеринбург	Вернула	Мужской	Мужской	8
17	Абраменко Екатерина Владимировна	56	Москва	Вернула	Женский	Женский	4
18	Абрамов Дмитрий Владимирович	51	Москва	Вернула	Мужской	Мужской	4
19	Абрамов Никита Валерьевич	45	Екатеринбург	Вернула	Мужской	Мужской	8

Выполняем итоговую нормализацию строковых значений переменной *Статус*.

```
# выполняем итоговую нормализацию строковых
# значений переменной Статус
data['Статус'] = np.where(data['Клиент'].str.contains('вна'),
                           data['Статус'],
                           data['Статус'].map(lambda x: str(x)[-1]))
data.head(20)
```

56 ❖ План предварительной подготовки данных

	Клиент	Возраст	Регион	Статус	Пол	Пол2	Регион2
0	Колесников Вячеслав Анатольевич	33	Красноярск	Вернул	Мужской	Мужской	2
1	Саймурзанов Михаил Борисович	22	Красноярск	Вернул	Мужской	Мужской	2
2	Абайимов Максим Дмитриевич	43	Москва	Вернул	Мужской	Мужской	4
3	Абакумова Юлия Ивановна	22	Москва	Вернула	Женский	Женский	4
4	Абанова Елена Владимировна	54	Санкт-Петербург	Вернула	Женский	Женский	6
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург	Вернула	Женский	Женский	6
6	Абдугалиева Айгуль Максутовна	27	Москва	Не вернула	Женский	Женский	4
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург	Не вернул	Мужской	Мужской	8
8	Абдуллин Евгений Эдуардович	22	Екатеринбург	Не вернул	Мужской	Мужской	8
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернула	Женский	Женский	8
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернула	Женский	Женский	8
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург	Вернула	Женский	Женский	8
12	Абдурахимова Алена Алимовна	57	Екатеринбург	Вернула	Женский	Женский	8
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург	Вернула	Женский	Женский	8
14	Аблец Юлия Сергеевна	33	Екатеринбург	Вернула	Женский	Женский	8
15	Аболмасова Ирина Олеговна	38	Екатеринбург	Вернула	Женский	Женский	8
16	Абраев Нурлан Мусайбекович	49	Екатеринбург	Вернул	Мужской	Мужской	8
17	Абраменко Екатерина Владимировна	56	Москва	Вернула	Женский	Женский	4
18	Абрамов Дмитрий Владимирович	51	Москва	Вернул	Мужской	Мужской	4
19	Абрамов Никита Валерьевич	45	Екатеринбург	Вернул	Мужской	Мужской	8

Часто данные могут быть некорректно записаны, например несколько полей могут быть записаны в одно, и необходимо извлечь их. Давайте загрузим данные.

```
# загружаем данные
data = pd.read_csv('Data/Raw_text.csv', encoding='cp1251')
data
```

	raw
0	KDR 1 2014-12-23 3242.0
1	MSK 1 2010-02-23 3453.7
2	KRSK 0 2014-06-20 2123.0
3	SPB 0 2014-03-14 1123.6
4	EKB 1 2013-01-15 2134.0

Видим, что несколько переменных записаны в один столбец *raw*. Давайте с помощью метода *.str.extract()* извлечем даты, создав переменную *date*.

```
# с помощью метода .str.extract() извлекаем
# даты из столбца raw, создав переменную date
data['date'] = data['raw'].str.extract(
    '(\d{4}-\d{2}-\d{2})', expand=True)
data
```

	raw	date
0	KDR 1 2014-12-23 3242.0	2014-12-23
1	MSK 1 2010-02-23 3453.7	2010-02-23
2	KRSK 0 2014-06-20 2123.0	2014-06-20
3	SPB 0 2014-03-14 1123.6	2014-03-14
4	EKB 1 2013-01-15 2134.0	2013-01-15

Извлекаем одиночные цифры из столбца *raw*, создав переменную *gender*.

```
# извлекаем одиночные цифры из столбца raw,
# создав переменную gender
data['gender'] = data['raw'].str.extract(
    '(\d)', expand=True)
data
```

	raw	date	gender
0	KDR 1 2014-12-23 3242.0	2014-12-23	1
1	MSK 1 2010-02-23 3453.7	2010-02-23	1
2	KRSK 0 2014-06-20 2123.0	2014-06-20	0
3	SPB 0 2014-03-14 1123.6	2014-03-14	0
4	EKB 1 2013-01-15 2134.0	2013-01-15	1

Извлекаем числа с плавающей точкой из столбца *raw*, создав переменную *score*.

```
# извлекаем числа с плавающей точкой из столбца raw,
# создав переменную score
data['score'] = data['raw'].str.extract(
    '(\d\.\d\.\d\.\.\d)', expand=True)
data
```

	raw	date	gender	score
0	KDR 1 2014-12-23 3242.0	2014-12-23	1	3242.0
1	MSK 1 2010-02-23 3453.7	2010-02-23	1	3453.7
2	KRSK 0 2014-06-20 2123.0	2014-06-20	0	2123.0
3	SPB 0 2014-03-14 1123.6	2014-03-14	0	1123.6
4	EKB 1 2013-01-15 2134.0	2013-01-15	1	2134.0

Наконец, извлекаем текст из столбца *raw*, создав переменную *city*.

```
# извлекаем текст из столбца raw, создав переменную city
data['city'] = data['raw'].str.extract(
    '(\w+)', expand=True)
data
```

	raw	date	gender	score	city
0	KDR 1 2014-12-23 3242.0	2014-12-23	1	3242.0	KDR
1	MSK 1 2010-02-23 3453.7	2010-02-23	1	3453.7	MSK
2	KRSK 0 2014-06-20 2123.0	2014-06-20	0	2123.0	KRSK
3	SPB 0 2014-03-14 1123.6	2014-03-14	0	1123.6	SPB
4	EKB 1 2013-01-15 2134.0	2013-01-15	1	2134.0	EKB

Полезной процедурой для нормализации строк является вычисление расстояния Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) – минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Таким образом, оно измеряет сходство двух строк.

Например, для слов Smith и Smythe расстояние Левенштейна будет равно 2 и вычисляется следующим образом:

Слово 1	S	m	i	t	h	
Слово 2	S	m	y	t	h	e
Оценка	0	0	1	0	0	1

Рис. 11 Пример вычисления расстояния Левенштейна

Расстояние Левенштейна активно применяется для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи), в маркетинге для поиска дублей в клиентских базах и унификации товарной номенклатуры, в биоинформатике для сравнения генов, хромосом и белков. В Python расстояние Левенштейна можно вычислить с помощью библиотеки python-Levenshtein (можно установить с помощью команды `pip install python-Levenshtein`).

Давайте вычислим расстояние Левенштейна для нашего игрушечного примера с помощью функции `distance()` библиотеки `python-Levenshtein`.

```
# импортируем функцию distance()
# библиотеки python-Levenshtein
from Levenshtein import distance
# вычисляем расстояние для двух строк
string1 = 'Smith'
string2 = 'Smythe'
distance(string1, string2)
```

2

С точки зрения применения определение расстояния между словами или текстовыми полями по Левенштейну обладает следующими недостатками:

- при перестановке местами слов или частей слов получаются сравнительно большие расстояния;

- расстояния между совершенно разными короткими словами оказываются небольшими, в то время как расстояния между очень похожими длинными словами оказываются значительными;
- расстояния между очень похожими словами, но в разных регистрах, оказываются большими, поэтому требуется предварительная «нормализация» строк, например приведение к одному и тому же регистру.

В некоторых недостатках мы сейчас убедимся.
Загружаем данные с двумя столбцами адресов.

```
# загружаем данные
df = pd.read_csv('Data/Levenshtein.csv', sep=';')
df
```

	line1	line2
0	9128 LEEWARD CIR, INDIANAPOLIS, IN	9128 Leeward Circle, Indianapolis, IN
1	101 OCEAN LANE DRIVE, KEY BISCAYNE, FL	101 Ocean Lane Drive Unit 1010, Key Biscayne, FL
2	9301 EVERGREEN DRIVE, PARMA, OH	9301 Evergreen Dr, Parma, OH
3	1817 BERTRAND DR, LAFAYETTE, LA	2924 Polo Ridge Ct, Charlotte, NC
4	201 E 87TH ST, NEW YORK, NY	201 E 87th Street 3E, New York, NY
5	799 CARRIGAN AVE, OVIEDO, FL	799 Carrigan Avenue, Oviedo, FL
6	4014 CADDIE DRIVE, ACWORTH, GA	10617 WEYBRIDGE DR, TAMPA, FL

Нетрудно увидеть, что есть очень похожие строки (выделены красными рамками). Вычисляем расстояние Левенштейна.

```
# вычисляем расстояние Левенштейна
df['distance'] = df.apply(
    lambda x: distance(x['line1'], x['line2']), axis=1)
df
```

	line1	line2	distance
0	9128 LEEWARD CIR, INDIANAPOLIS, IN	9128 Leeward Circle, Indianapolis, IN	22
1	101 OCEAN LANE DRIVE, KEY BISCAYNE, FL	101 Ocean Lane Drive Unit 1010, Key Biscayne, FL	30
2	9301 EVERGREEN DRIVE, PARMA, OH	9301 Evergreen Dr, Parma, OH	16
3	1817 BERTRAND DR, LAFAYETTE, LA	2924 Polo Ridge Ct, Charlotte, NC	26
4	201 E 87TH ST, NEW YORK, NY	201 E 87th Street 3E, New York, NY	15
5	799 CARRIGAN AVE, OVIEDO, FL	799 Carrigan Avenue, Oviedo, FL	17
6	4014 CADDIE DRIVE, ACWORTH, GA	10617 WEYBRIDGE DR, TAMPA, FL	22

Видим, что очень похожие строки имеют достаточно большие расстояния из-за того, что имеют разные регистры.

Выполним «нормализацию» строк, т. е. переведем буквы в верхний регистр, и снова вычислим расстояние Левенштейна.

```
# приводим строки к ВЕРХНЕМУ регистру
df['line1'] = df['line1'].str.upper()
```

60 ❖ План предварительной подготовки данных

```
df['line2'] = df['line2'].str.upper()  
# снова вычисляем расстояние Левенштейна  
df['distance_corr'] = df.apply(  
    lambda x: distance(x['line1'], x['line2']), axis=1)  
df
```

	line1	line2	distance	distance_corr
0	9128 LEEWARD CIR, INDIANAPOLIS, IN	9128 LEEWARD CIRCLE, INDIANAPOLIS, IN	22	3
1	101 OCEAN LANE DRIVE, KEY BISCAYNE, FL	101 OCEAN LANE DRIVE UNIT 1010, KEY BISCAYNE, FL	30	10
2	9301 EVERGREEN DRIVE, PARMA, OH	9301 EVERGREEN DR, PARMA, OH	16	3
3	1817 BERTRAND DR, LAFAYETTE, LA	2924 POLO RIDGE CT, CHARLOTTE, NC	26	23
4	201 E 87TH ST, NEW YORK, NY	201 E 87TH STREET 3E, NEW YORK, NY	15	7
5	799 CARRIGAN AVE, OVIEDO, FL	799 CARRIGAN AVENUE, OVIEDO, FL	17	3
6	4014 CADDIE DRIVE, ACWORTH, GA	10617 WEYBRIDGE DR, TAMPA, FL	22	22

Теперь мы видим уже более адекватные значения.

9. Обработка дублирующихся наблюдений

Нередко при подготовке выборки допускаются ошибки, в частности в набор данных несколько раз может попасть одно и то же наблюдение. Нужно убедиться, что наш набор не содержит дублирующихся наблюдений (строк). Для идентификации дублей используем метод `.duplicated()`, а для удаления дублей применяем метод `drop_duplicates()`.

```
# импортируем библиотеку pandas
import pandas as pd

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Verizon.csv', sep=';')
# смотрим первые пять наблюдений
data.head()
```

	longdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income	churn
0	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0
1	21.27	0.0	218,12	Нет	Бюджетный	СН	60.0	NaN	_Одинокий	2.0	95930,6	1
2	6,13	0.0	NaN	Да	NaN	NaN	25.0	Женский	NaN	2.0	295,34	1
3	16.46	0.0	57,66	Да	Бесплатный	NaN	93.0	Женский	Одинокий	0.0	NaN	1
4	NaN	0.0	16,01	Да	Бесплатный	СС	68.0	Женский&*	NaN	0.0	99832,9	1

```
# посмотрим наличие дублей
data[data.duplicated(keep=False)]
```

	longdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income	churn
0	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0
13	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0
14	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0

```
# удаляем дубли на месте, оставляя первое
# встретившееся наблюдение в памяти дубля
data.drop_duplicates(subset=None, keep='first',
                     inplace=True)

# посмотрим наличие дублей
data[data.duplicated(keep=False)]
```

10. Обработка редких категорий

Часто бывает, что наши переменные содержат редкие категории. Редкие категории являются источником шума в данных, который негативно повлияет на качество модели. Кроме того, при разбиении набора данных на обучающую и тестовую выборки может оказаться, что данная категория отсутствует в обучающей выборке, но присутствует в тестовой выборке. Это вызовет проблемы при моделировании. Например, логистическая регрессия, встретив в тестовых данных наблюдение с неизвестной категорией признака, не сможет вычислить прогноз, потому что категория не будет соответствовать схеме дамми-кодирования, полученной для переменной в обучающей выборке, и таким образом не будет вычислен соответствующий регрессионный коэффициент.

Давайте импортируем необходимые библиотеки и класс и загрузим данные с редкими категориями.

```
# импортируем необходимые библиотеки и класс
import pandas as pd
import numpy as np
from collections import defaultdict
# загружаем данные
data = pd.read_csv('Data/Rare_categories.csv', sep=';')
# выводим наблюдения
data.head()
```

TARGET	GEN_INDUSTRY	GEN_TITLE	ORG_TP_STATE	ORG_TP_FCAPITAL	JOB_DIR
0	0	Торговля	Рабочий	Частная компания	Без участия Вспомогательный техперсонал
1	0	Торговля	Рабочий	Индивидуальный предприниматель	Без участия Участие в основ. деятельности
2	0	Информационные технологии	Специалист	Государственная комп./учреж.	Без участия Участие в основ. деятельности
3	0	Образование	Руководитель среднего звена	Государственная комп./учреж.	Без участия Участие в основ. деятельности
4	0	Государственная служба	Специалист	Государственная комп./учреж.	Без участия Участие в основ. деятельности

Набор данных состоит из одних категориальных переменных. Давайте выведем частоты категорий по каждой переменной.

```
# создаем список категориальных переменных
cat_cols = data.dtypes[data.dtypes == 'object'].index.tolist()
# смотрим частоты по категориальным переменным
for col in cat_cols:
    print(data[col].value_counts(dropna=False))
    print('')
```

Торговля	2385
Другие сферы	1709
NaN	1367
Металлургия/Промышленность/Машиностроение	1356
Государственная служба	1286
Здравоохранение	1177
Образование	998
Транспорт	787
Сельское хозяйство	702
Строительство	574
Коммунальное хоз-во / Дорожные службы	533
Ресторанный бизнес / Общественное питание	408

Наука	403
Нефтегазовая промышленность	225
Сборочные производства	172
Банк/Финансы	169
Энергетика	145
Развлечения/Искусство	141
ЧОП / Детективная д-ть	136
Информационные услуги	108
Салоны красоты и здоровья	99
Информационные технологии	85
Химия/Парфюмерия/Фармацевтика	63
СМИ/Реклама/PR-агентства	49
Юридические услуги / нотариальные услуги	47
Страхование	28
Туризм	20
Недвижимость	16
Управляющая компания	12
Логистика	11
Подбор персонала	8
Маркетинг	4

Name: GEN_INDUSTRY, dtype: int64

Специалист	7010
Рабочий	3075
NaN	1367
Служащий	904
Руководитель среднего звена	697
Работник сферы услуг	563
Высококвалифиц. специалист	549
Руководитель высшего звена	427
Индивидуальный предприниматель	217
Другое	177
Руководитель низшего звена	136
Военнослужащий по контракту	88
Партнер	13

Name: GEN_TITLE, dtype: int64

Частная компания	6523
Государственная комп./учреж.	6112
NaN	1367
Индивидуальный предприниматель	957
Некоммерческая организация	243
Частная ком. с инос. капиталом	21

Name: ORG_TP_STATE, dtype: int64

Без участия	13688
NaN	1365
С участием	170

Name: ORG_TP_FCAPITAL, dtype: int64

Участие в основ. деятельности	11452
NaN	1367
Вспомогательный техперсонал	1025
Бухгалтерия, финансы, планир.	481
Адм-хоз. и трансп. службы	279
Снабжение и сбыт	217
Служба безопасности	164

Кадровая служба и секретариат	101
Пр-техн. обесп. и телеком.	75
Юридическая служба	53
Реклама и маркетинг	9
Name: JOB_DIR, dtype: int64	

Обработка редких категорий выполняется либо до разбиения на обучающую и тестовую выборки, либо после него в зависимости от причин, обусловивших появление таких категорий.

Если переменная содержит 2–3 редкие категории небольшой частоты, скорее всего, такие категории случайны и часто могли быть обусловлены очевидными ошибками ввода. В таком случае эти категории, как правило, объединяют с самой часто встречающейся категорией или по смыслу (например, у нас есть категории John, Mike, Jack и редкая категория Jon, последняя категория является, скорее всего, результатом ошибки ввода, и ее можно заменить на John). Это можно сделать до разбиения на обучающую и тестовую выборки.

Кроме того, у нас могут быть априорные знания, подтвержденные опытом и бизнес-логикой. Например, у нас есть редкие категории-регионы, поскольку мы в них редко выдаем кредиты из-за плохого процента погашений, эти регионы характеризуются нестабильной социально-экономической ситуацией (Республика Дагестан, Республика Адыгея, Чеченская республика) или на их территории находятся зоны, регистрируется высокий уровень преступности (Забайкальский край, Республика Мордовия). Мы можем отнести эти редко встречающиеся регионы к категории *Неблагополучные регионы*. Это тоже можно сделать до разбиения на обучающую и тестовую выборки.

Если таких априорных знаний нет, нам необходимо задать порог укрупнения – минимальное количество наблюдений в категории, ниже которого категория объявляется редкой. Поэтому для объективности решение о выборе такого порога должно приниматься уже после разбиения на обучающую и тестовую выборки. В противном случае получится, что решение о выборе порога мы принимали с учетом информации «из будущего». В данной ситуации чаще всего пишут собственный класс, в котором можно задавать порог укрупнения, и соответственно этот порог можно использовать в качестве гиперпараметра в сетке гиперпараметров при работе с классами *Pipeline* и *GridSearchCV*.

Множественные редкие категории часто объединяют в одну отдельную категорию, если подтверждается гипотеза о том, что редкие категории описывают определенный паттерн. Например, в кредитном scoringе укрупнение редких категорий в отдельную категорию нередко улучшает результат. Редкие типы кредитов могут соответствовать кредитам, выданным на эксклюзивных условиях, подобные кредиты выдаются людям с хорошей кредитной историей, и, таким образом, объединив редкие категории в отдельную группу, мы выделяем группу заемщиков с лучшим кредитным статусом.

Кроме того, применяется случайное присвоение редких категорий уже существующим категориям.

Ниже приведен пример выделения редких категорий признака *JOB_DIR* в отдельную категорию.

```
# записываем указанные категории переменной
# JOB_DIR в отдельную категорию OTHER
lst = ['Реклама и маркетинг', 'Юридическая служба']
data.loc[data['JOB_DIR'].isin(lst), 'JOB_DIR'] = 'OTHER'
# смотрим частоты
data['JOB_DIR'].value_counts(dropna=False)

Участие в основ. деятельности      11452
NaN                                1367
Вспомогательный техперсонал       1025
Бухгалтерия, финансы, планир.    481
Адм-хоз. и трансп. службы          279
Снабжение и сбыт                  217
Служба безопасности                164
Кадровая служба и секретариат    101
Пр-техн. обесп. и телеком.        75
OTHER                               62
Name: JOB_DIR, dtype: int64
```

Теперь давайте напишем собственный класс `RareGrouper`, с помощью которого можно задать порог укрупнения. Обратите внимание: здесь мы используем словарь со значением по умолчанию (`defaultdict`). Он похож на обычный словарь, за исключением одной особенности – при попытке обращаться к ключу, которого в нем нет, он сперва добавляет для него значение, используя функцию без аргументов, которая предоставляется при его создании (в нашем случае речь пойдет о функции `list()`). В методе `.fit()` для каждой переменной вычисляем относительные частоты категорий и получаем словарь вида

```
defaultdict(list,
            {'var1': ['категория выше порога', 'категория выше порога'],
             'var2': ['категория выше порога', 'категория выше порога']})
```

Обратите внимание, что нас всегда интересуют частоты категорий, вычисленные на обучающей выборке, таким образом, с порогом мы сравниваем частоты категорий, вычисленные на обучающей выборке.

```
# пишем класс, укрупняющий категории по порогу, категории
# с относительными частотами меньше порога запишем в
# отдельную категорию OTHER
class RareGrouper():
    """
    Параметры:
    threshold: int, значение по умолчанию 0.01
        Минимально допустимая относительная частота,
        при которой замены не происходит.
    """

    def __init__(self, threshold=0.01):
        self.d = defaultdict(list)
        self.threshold = threshold

    def fit(self, X, y=None):
```

```

# для каждой переменной вычисляем относительные
# частоты категорий и получаем словарь вида
# defaultdict(
#     list, {'var1': ['категория выше порога',
#                    'категория выше порога'],
#            'var2': ['категория выше порога',
#                    'категория выше порога']})
n_obs = len(X)
for col in X.columns:
    rel_freq = X[col].value_counts(dropna=False) / n_obs
    self.d[col] = rel_freq[rel_freq >= self.threshold].index
return self

def transform(self, X):
    # создаем копию датафрейма
    X = X.copy()
    # для каждой переменной категории (строковые значения), которых нет
    # в соответствующем списке словаря (их частоты ниже порога),
    # относим к категории Other
    for col in X.columns:
        X[col] = np.where(X[col].isin(self.d[col]), X[col], 'Other')
    return X

```

Применяем класс RareGrouper, для переменной *JOB_DIR* категории менее 200 наблюдений запишем в категорию OTHER, порог равен $200 / 15\ 223 = 0,013$.

```

# применяем класс, для переменной JOB_DIR категории
# менее 200 наблюдений запишем в категорию OTHER,
# порог равен  $200 / 15\ 223 = 0,013$ 
raregrouper = RareGrouper(threshold=0.013)
raregrouper.fit(data[['JOB_DIR']])
data['JOB_DIR'] = raregrouper.transform(data[['JOB_DIR']])
# смотрим частоты категорий
# переменной GEN_TITLE
data['JOB_DIR'].value_counts(dropna=False)

```

Участие в основн. деятельности	11452	Участие в основн. деятельности	11452
NaN	1367	NaN	1367
Вспомогательный техперсонал	1025	Вспомогательный техперсонал	1025
Бухгалтерия, финансы, планир.	481	Бухгалтерия, финансы, планир.	481
Адм-хоз. и трансп. службы	279	Адм-хоз. и трансп. службы	279
Снабжение и сбыт	217	Снабжение и сбыт	217
Other	402	Служба безопасности	164
Пр-техн. обсл. и телеком.	75	Кадровая служба и секретариат	101
Юридическая служба	53	Реклама и маркетинг	9
Name: JOB_DIR, dtype: int64		Name: JOB_DIR, dtype: int64	402

С помощью метода CHAID можно выполнить укрупнение категорий и посмотреть, с какими из существующих категорий была объединена та или иная редкая категория.

CHAID (*Chi-square Automatic Interaction Detector – Автоматический обнаружитель взаимодействий*) был разработан Гордоном Каасом в 1980 году и представляет собой метод на основе дерева решений, который исследует взаимосвязь между признаками и зависимой переменной с помощью статистических тестов. Зависимая переменная может быть измерена в категориальной или количественной шкале. Признаки могут быть только категориальными пере-

менными (количественные переменные должны быть предварительно преобразованы в категориальные порядковые с помощью биннинга). На первом этапе категории каждого признака объединяются, если они не имеют между собой статистически значимых отличий по отношению к зависимой переменной (для категориальной зависимой переменной используется критерий хи-квадрат, для количественной зависимой переменной используется F-критерий). Категории, которые дают значимые отличия по зависимой переменной, рассматриваются как отдельные. На втором этапе для разбиения узла выбирается признак, сильнее всего взаимодействующий с зависимой переменной. Получив новые узлы, мы повторяем для каждого из них вышеописанные шаги. Этот процесс продолжается до тех пор, пока есть возможность создания новых узлов, т. е. пока не останется переменных, позволяющих получать узлы, максимально отличающиеся по зависимой переменной, или пока не сработают правила остановки. Если CHAID применять для укрупнения категорий конкретной переменной, то нам по сути требуется только первый этап.

В среде Python укрупнение категорий на основе метода CHAID можно выполнить с помощью пакета CHAID. Этот пакет можно установить с помощью команды `pip install CHAID`.

Выполним для категориальной переменной `GEN_TITLE` укрупнение категорий на основе CHAID.

```
# укрупняем категории с помощью CHAID
from CHAID import Tree
# задаем название признака
independent_variable = 'GEN_TITLE'
# задаем название зависимой переменной
dep_variable = 'TARGET'
# создаем словарь, где ключом будет название
# признака, а значением - тип переменной
dct = {independent_variable: 'nominal'}
# строим дерево CHAID и выводим его
tree = Tree.from_pandas_df(data, dct, dep_variable, max_depth=1) tree.print_tree()

[], {0: 13411.0, 1: 1812.0}, (GEN_TITLE, p=9.43237916410679e-29, score=133.51911498532897,
groups=[['<missing>'], ['Специалист', 'Руководитель среднего звена', 'Другое'], ['Высококвалифиц. специалист', 'Служащий', 'Работник сферы услуг', 'Рабочий'], ['Военнослужащий по контракту', 'Индивидуальный предприниматель', 'Руководитель высшего звена', 'Руководитель низшего звена', 'Партнер']]), dof=3)
|-- ([<missing>], {0: 1317.0, 1: 50.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- ([Специалист', 'Руководитель среднего звена', 'Другое'], {0: 6984.0, 1: 900.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- ([Высококвалифиц. специалист', 'Служащий', 'Работник сферы услуг', 'Рабочий'], {0: 4379.0, 1: 712.0}, <Invalid Chaid Split> - the max depth has been reached)
--- (['Военнослужащий по контракту', 'Индивидуальный предприниматель', 'Руководитель высшего звена', 'Руководитель низшего звена', 'Партнер'], {0: 731.0, 1: 150.0}, <Invalid Chaid Split> - the max depth has been reached)
```

Первая строка вывода начинается с информации о частотах классов зависимости переменной `{0: 13411.0, 1: 1812.0}`, значение p показывает статистическую значимость, `score` показывает значение хи-квадрат (меньшее p -значение

говорит о более сильной взаимосвязи между признаком и зависимой переменной), `groups` показывает полученные категории. Далее приводятся узлы – укрупненные категории и распределение классов зависимой переменной в каждом узле. Также выводятся предупреждения, сообщающие, какое из правил остановки сработало: для всех узлов приводится предупреждение `the max depth has been reached` – достигнута максимальная глубина, это неудивительно, ведь мы выбрали глубину 1 (`max_depth=1`, т. е. дерево будет иметь один уровень, лежащий ниже корневого узла). Мы видим, что редкая категория Партнер объединена с категориями Военнослужащий по контракту, Индивидуальный предприниматель, Руководитель высшего звена, Руководитель низшего звена. Укрупнение редких категорий с помощью CHAID необходимо выполнять после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки, поскольку CHAID использует для работы биннинг на основе децилей и применяет статистические критерии для принятия решения об объединении категорий.

11. Появление новых категорий в новых данных

Существует еще проблема появления новых категорий в новых данных. Например, мы разработали и внедрили скоринговую модель. К моменту внедрения модели маркетинговая или кредитная политика банка поменялась, и у нас в переменной *Сфера занятости* появилась новая категория Няни, воспитательницы.

В банках часто применяется консервативный подход: новая категория приравнивается к категории, демонстрирующей наибольший уровень риска, потому что мы ничего не знаем об этой категории клиентов и их возможном кредитном статусе. В других случаях новую категорию приравнивают к наиболее редкой категории или наиболее частой категории, способ зависит от априорных знаний, накопленного опыта.

Допустим, у нас в исторических данных есть переменная *pay*. У нее есть категории CC, CH и AUTO.

```
CC      2561  
CH      977  
Auto    889  
Name: pay, dtype: int64
```

В новых данных появилась категория CP.

В функцию предобработки, которую мы будем применять к новым данным, добавим программный код, заменяющий все новые категории модой, т. е. категорией CC.

```
...  
# все новые категории переменной pay заменяем модой  
lst = ['CC', 'Auto', 'CH']  
replace_new_values = lambda s: 'CC' if s not in lst else s  
df['pay'] = df['pay'].map(replace_new_values)  
...
```

12. Импутация пропусков

Существует три типа возникновения пропусков: MCAR, MAR, MNAR.

MCAR («совершенно случайно пропущенные» – Missing Completely At Random) – тип возникновения пропусков, при котором вероятность пропуска для каждого наблюдения набора одинакова. Вероятность пропуска значения для переменной X не связана ни со значением самой переменной X , ни со значениями других переменных в наборе данных. Например, переменная *Доход* подчиняется условию MCAR, если клиенты, которые не сообщают о своем доходе, имеют в среднем такой же размер дохода, что и клиенты, которые указывают свой доход.

MAR («случайно пропущенные» – Missing At Random) – тип возникновения пропусков, когда данные пропущены не случайно, а ввиду некоторых закономерностей. Вероятность пропуска значения для переменной X может быть объяснена другими имеющимися переменными, не содержащими пропуски. Например, переменная *Доход* подчиняется условию MAR, если вероятность пропуска данных в переменной *Доход* зависит от наблюданной переменной, например от переменной *Образование*. Например, клиенты с низким уровнем образования могут иметь большее количество пропущенных значений дохода (т. е. чаще, чем другие респонденты, не отвечают на вопрос о доходе). Необходимо проанализировать взаимосвязь между переменной *Доход* и переменной *Образование*.

MNAR («не случайно пропущенные» – Missing Not At Random) – тип пропущенных данных, когда пропуск значения не является совершенно случайным и не может быть полностью объяснен другими переменными в наборе. Пропущенные значения остаются зависимыми от неизвестных нам факторов, необходимо провести дополнительные исследования. Здесь можно привести вышеописанный случай с пропусками в переменной *Доход*, но только теперь переменная *Образование* у нас отсутствует.

12.1. Способы импутации количественных и бинарных переменных

Пропуски в количественных переменных заменяются вычислennыми статистиками, обычно используется среднее или медиана. В случае данных, имеющих асимметричное распределение, предпочитают использовать медиану, а не среднее, так как на нее не влияет небольшое число наблюдений с очень большими или очень маленькими значениями.

Вновь обратите внимание, что импутацию средним, медианой и прочими статистиками необходимо выполнять после разбиения набора данных на обучающую и тестовую выборки (внутри цикла перекрестной проверки).

Помимо импутации средним или медианой, пропуски можно заменить значениями-константами. Для древовидных алгоритмов эффективной может быть импутация значением вне диапазона имеющихся значений. Если импутировать значением, которое будет больше любого имеющегося значения, то в дереве можно будет выбрать такое разбиение по этому признаку, что все наблюдения с известными значениями пойдут в левый узел, а все наблюде-

ния с пропусками – в правый. Если же импутировать значением, которое будет меньше любого имеющегося значения, то в дереве можно будет выбрать такое разбиение по этому признаку, что все наблюдения с пропусками пойдут в левый узел, а все наблюдения с известными значениями – в правый.

Часто применяют индикатор пропусков для соответствующей переменной. Он принимает значение 1, если переменная имеет пропуск, или 0, если переменная не содержит пропуск. В рамках бизнес-подхода индикатор пропусков носит временный характер. Мы строим уравнение регрессии и смотрим, является ли коэффициент для данного индикатора значимым. Если коэффициент значим, то выбор способа импутации признака, для которого создавался индикатор, может существенно повлиять на качество модели. Если коэффициент не является значимым, то выбор способа импутации признака, для которого создавался индикатор, не повлияет существенно на качество модели. Это необходимо для приоритизации операций импутации для десятков-сотен признаков. Допустим, у нас 200 признаков с пропусками, нужно выяснить, для каких признаков имеет смысл пробовать разные способы импутации, а каким признакам будет достаточно импутации медианой или средним.

Выполнить импутацию константами и создать индикаторы пропусков можно (и нужно, чтобы не повторять одни и те же операции для двух наборов данных) до разбиения на обучение и тест (до перекрестной проверки), потому что в рамках этой операции мы не делаем вычислений, охватывающих все наблюдения исходного набора.

Бинарные переменные, у которых есть пропуски, можно превратить в тринарные, где первую категорию можно закодировать как -1, вторую категорию – как 1, а пропуски – как 0.

Количественная переменная	Бинарная переменная
NA	1
6	0
5	1
8	NA
0	0
NA	0
9	1
3	NA
5	0

Количественная переменная			Бинарная переменная
медиана	среднее	-999	индикатор пропусков
5	5.14	-999	1
6	6	6	0
5	5	5	0
8	8	8	0
0	0	0	0
5	5.14	-999	1
9	9	9	0
3	3	3	0
5	5	5	0

данные способы импутации можно применять только после разбиения на обучение и тест (внутри цикла перекрестной проверки)

данные способы импутации можно применять до разбиения на обучение и тест (до цикла перекрестной проверки)

Рис. 12 Способ импутации количественных признаков

12.2. Способы импутации категориальных переменных

Пропуски в категориальных переменных можно заменить самой часто встречающейся категорией – модой. Обратите внимание: мод может быть несколько, и в программном коде вы должны указать, какую моду хотите использовать. Также заметьте, что поскольку мы используем вычисления, импутацию

модой можно осуществлять только после разбиения на обучение и тест (внутри цикла перекрестной проверки).

Пропуски в категориальных переменных часто кодируют отдельной категорией для пропусков, а также, как и в случае с количественными переменными, создают индикаторы пропусков (кроме случаев, когда категориальная переменная преобразуется в набор бинарных признаков с помощью дамми-кодирования).

Закодировать пропуски отдельной категорией и создать индикаторы можно (и нужно, чтобы не повторять одни и те же операции для двух наборов данных) до разбиения на обучение и тест (до перекрестной проверки), потому что в рамках этой операции мы не делаем вычислений, охватывающих все наблюдения исходного набора.

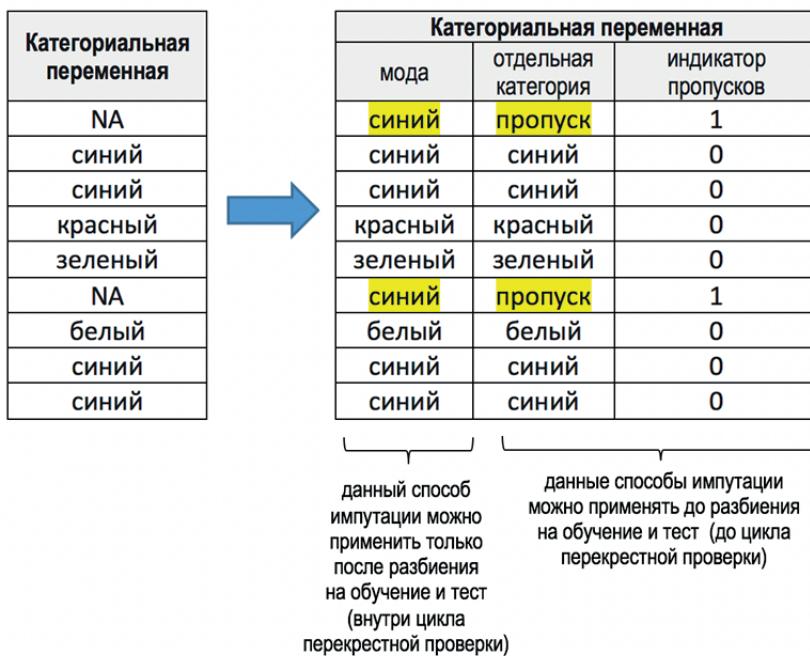


Рис. 13 Способ импутации категориальных признаков

В первой части мы уже познакомились с классом `SimpleImputer`, который часто применяется, когда нужно импутировать разные списки признаков. Для импутации пропусков можно также воспользоваться методом `.fillna()`. Для создания индикаторов пропусков используют функцию `pr.where()`. Функция имеет три аргумента: первый аргумент – проверяемое условие, второй аргумент – что возвращать, если проверяемое условие верно, третий аргумент – что возвращать, если проверяемое условие неверно. Кроме того, в библиотеке scikit-learn есть класс `MissingIndicator`.

Таблица 2 Методы pandas для импутации пропусков

импутация средним или медианой	<code>tr['income'].fillna(tr['income'].mean(), inplace=True) tst['income'].fillna(tr['income'].mean(), inplace=True)</code> <code>tr['income'].fillna(tr['income'].median(), inplace=True) tst['income'].fillna(tr['income'].median(), inplace=True)</code>
импутация модой	<code>tr['pay'].fillna(tr['pay'].value_counts().index[0], inplace=True) tst['pay'].fillna(tr['pay'].value_counts().index[0], inplace=True)</code>
индикатор пропусков	<code>data['pay_ind'] = np.where(data['pay'].isnull(), 1, 0)</code>
импутация константой	<code>data['income'].fillna(-999, inplace=True)</code>

12.3. ПРАКТИКА

Давайте импортируем необходимые библиотеки, классы и функции и загрузим данные Verizon.

```
# импортируем библиотеки pandas, numpy, missingno,  
# SimpleImputer, функции train_test_split()  
# и display()  
import pandas as pd  
import numpy as np  
import missingno as msno  
from sklearn.impute import SimpleImputer  
from sklearn.model_selection import train_test_split  
from IPython.display import display  
  
# включаем режим 'retina', если у вас экран Retina  
%config InlineBackend.figure_format = 'retina'  
  
# записываем CSV-файл в объект DataFrame  
data = pd.read_csv('Data/Verizon_missing.csv', sep=';')  
# выведем первые 3 наблюдения  
data.head()
```

	longdist	internat	local	age	income	billtype	pay	churn
0	16.0	0.0	5.0	46.0	34805.5	Бюджетный	CC	0
1	0.0	0.0	5.0	59.0	60111.8		NaN	NaN
2	13.0	0.0	NaN	NaN	13126.9	Бюджетный	Auto	0
3	NaN	0.0	10.0	36.0	NaN		NaN	CH
4	NaN	NaN	10.0	38.0	17499.2	Бесплатный	Auto	0

Для вывода информации о пропусках используем цепочку методов `.isnull()` и `.sum()`.

```
# смотрим количество пропусков по каждой переменной  
data.isnull().sum()
```

```
longdist      8  
internat     34  
local        14  
age          8  
income       29  
billtype     24  
pay          15  
churn         0  
dtype: int64
```

С помощью цепочки методов `.isnull()`, `.sum()` и `.sum()` можно вывести общее количество пропусков в наборе.

```
# общее количество пропусков  
data.isnull().sum().sum()
```

132

Разбиваем набор на обучающую и тестовую выборки.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий  
# массив признаков, тестовый массив признаков, обучающий массив  
# меток, тестовый массив меток  
train, test, y_train, y_test = train_test_split(  
    data.drop('churn', axis=1),  
    data['churn'],  
    test_size=.3,  
    stratify=data['churn'],  
    random_state=100)
```

Давайте выведем частоты категорий в признаке `pay` в обучающей и тестовой выборках.

```
# смотрим частоты категорий признака pay  
print("TRAIN:")  
print(train['pay'].value_counts(dropna=False))  
print("")  
print("TEST:")  
print(test['pay'].value_counts(dropna=False))
```

```
TRAIN:  
CC      42  
CH      25  
Auto    19  
NaN     13  
CD      1  
Name: pay, dtype: int64
```

```
TEST:  
CC      22
```

```
CH      14
Auto    5
NaN     2
CD      1
Name: pay, dtype: int64
```

Импутируем пропуски в признаке *pay* модой, вычисленной на обучающей выборке, и снова выведем частоты категорий в признаке *pay* в обучающей и тестовой выборках.

```
# выполняем импутацию модой
train['pay'].fillna(train['pay'].value_counts().index[0],
                     inplace=True)
test['pay'].fillna(train['pay'].value_counts().index[0],
                   inplace=True)
```

```
# смотрим частоты категорий признака pay
print("TRAIN:")
print(train['pay'].value_counts(dropna=False))
print("")
print("TEST:")
print(test['pay'].value_counts(dropna=False))
```

```
TRAIN:
CC      55
CH      25
Auto    19
CD      1
Name: pay, dtype: int64
```

```
TEST:
CC      24
CH      14
Auto    5
CD      1
Name: pay, dtype: int64
```

Видим, что самая частая категория – категория 'CC' – увеличилась в размере за счет того, что пропускам была присвоена эта категория.

Теперь импутируем пропуски в признаке *Income* средним значением, вычисленным в обучающей выборке.

```
# печатаем среднее признака income
print("среднее значение income: " + str(train['income'].mean()))
# печатаем значение признака income
# в строке с индексом 4
print(train['income'].iloc[4])
print(test['income'].iloc[4])
# выполняем импутацию средним
train['income'].fillna(train['income'].mean(),
                      inplace=True)
test['income'].fillna(train['income'].mean(),
                      inplace=True)
# печатаем значение признака income
# в строке с индексом 4
```

```
print(train['income'].iloc[4])
print(test['income'].iloc[4])
```

```
среднее значение income: 46635.54195121954
nan
nan
46635.54195121954
46635.54195121953
```

С помощью функции `display()` выведем первые пять наблюдений обучающей и тестовой выборок.

```
# взглянем на первые пять наблюдений каждой выборки
display(train.head())
display(test.head())
```

	longdist	internat	local	age	income	billtype	pay
18	19.0	NaN	11.0	88.0	66906.600000	Бесплатный	CH
19	19.0	NaN	96.0	79.0	37571.100000	Бесплатный	CC
66	0.0	0.0	1.0	94.0	13946.800000	Бюджетный	CC
65	9.0	6.0	12.0	93.0	46771.200000	Бюджетный	CC
74	26.0	0.0	NaN	32.0	46635.541951	Бюджетный	Auto

	longdist	internat	local	age	income	billtype	pay
44	3.0	0.0	9.0	45.0	70239.800000	Бесплатный	CC
59	21.0	NaN	NaN	33.0	60170.100000	NaN	CH
51	29.0	NaN	110.0	29.0	18861.600000	Бюджетный	CC
35	12.0	0.0	55.0	62.0	13965.400000	Бесплатный	CH
55	25.0	0.0	77.0	95.0	46635.541951	NaN	Auto

Пропуски в признаке `local` имптуируем групповыми средними. Мы воспользуемся средними значениями признака `local`, вычисленными по категориям признака `pay` в обучающей выборке. Для этого напишем класс `GroupImputer`, выполняющий имптуацию групповыми статистиками.

```
# пишем класс, выполняющий имптуацию
# групповыми статистиками
class GroupImputer():
    """
```

Автор: Eryk Lewinson
<https://github.com/erykm1>

Класс, выполняющий имптуацию групповыми статистиками.

Параметры

`group_cols: list`

Список группирующих столбцов.

`agg_col: str`

Агрегируемый столбец.

```

agg_func: str
    Агрегирующая функция.
"""
def __init__(self, group_cols, agg_col, agg_func='mean', return_df=True):

    if agg_func not in ['mean', 'median', 'min', 'max']:
        raise ValueError(f"Неизвестная агрегирующая функция {agg_func}")

    if type(group_cols) != list:
        raise ValueError("Задайте список группирующих столбцов")

    if type(agg_func) != str:
        raise ValueError("Агрегирующая функция должна +
                           "иметь строковое значение")

    self.group_cols = group_cols
    self.agg_col = agg_col
    self.agg_func = agg_func
    self.return_df = return_df

def fit(self, X, y=None):

    # проверка наличия пропусков в группирующих столбцах
    if pd.isnull(X[self.group_cols]).any(axis=None) == True:
        raise ValueError("Есть пропуски в группирующих столбцах")

    # получение датафрейма с групповыми статистиками
    self.impute_map_ = X.groupby(self.group_cols)[self.agg_col].agg(
        self.agg_func).reset_index(drop=False)

    return self

def transform(self, X, y=None):

    X = X.copy()

    # заполнение пропусков с помощью датафрейма
    # с групповыми статистиками
    for index, row in self.impute_map_.iterrows():
        ind = (X[self.group_cols] == row[self.group_cols]).all(axis=1)
        X.loc[ind, self.agg_col] = X.loc[ind, self.agg_col].fillna(
            row[self.agg_col])

    if not self.return_df:
        X = X.values

    return X

```

Применяем наш класс.

```

# выполняем импутацию пропусков признака local групповыми
# средними - средними значениями признака local,
# вычисленными по категориям признака pay
imp = GroupImputer(['pay'], agg_col='local', agg_func='mean')
imp.fit(train)
train = imp.transform(train)
test = imp.transform(test)

```

Вновь выведем первые пять наблюдений обучающей и тестовой выборок.

взглянем на первые пять наблюдений каждой выборки

```
display(train.head())
display(test.head())
```

	longdist	internat	local	age	income	billtype	pay
18	19.0	NaN	11.000000	88.0	66906.600000	Бесплатный	CH
19	19.0	NaN	96.000000	79.0	37571.100000	Бесплатный	CC
66	0.0	0.0	1.000000	94.0	13946.800000	Бюджетный	CC
65	9.0	6.0	12.000000	93.0	46771.200000	Бюджетный	CC
74	26.0	0.0	31.666667	32.0	46635.541951	Бюджетный	Auto

	pay	local
0	Auto	31.666667
1	CC	59.551020
2	CD	9.000000
3	CH	65.130435

	longdist	internat	local	age	income	billtype	pay
44	3.0	0.0	9.000000	45.0	70239.800000	Бесплатный	CC
59	21.0	NaN	65.130435	33.0	60170.100000		NaN CH
51	29.0	NaN	110.000000	29.0	18861.600000	Бюджетный	CC
35	12.0	0.0	55.000000	62.0	13965.400000	Бесплатный	CH
55	25.0	0.0	77.000000	95.0	46635.541951		NaN Auto

Для импутации пропусков в категориальных признаках можно применить вышеупомянутый метод CHAID. Он рассматривает пропуски как отдельную категорию и сравнивает ее с существующими категориями. Если категория пропусков и существующая категория не имеют между собой статистически значимых отличий по отношению к зависимой переменной, они объединяются. Таким образом, можно попробовать заменить пропуски той категорией, с которой они были объединены.

Давайте взглянем на частоты категорий переменной *billtype* в обучающей выборке.

смотрим частоты категорий признака billtype
train['billtype'].value_counts(dropna=False)

```
Бюджетный      50
Бесплатный     37
NaN              13
Name: billtype, dtype: int64
```

выполняем импутацию с помощью CHAID
from CHAID import Tree
сконкатенируем обучающий массив признаков и массив меток
train_data = pd.concat([train, y_train], axis=1)
задаем название признака
independent_variable = 'billtype'
задаем название зависимой переменной
dep_variable = 'churn'
создаем словарь, где ключом будет название
признака, а значением - тип переменной
dct = {independent_variable: 'nominal'}

```
# строим дерево CHAID и выводим его
tree = Tree.from_pandas_df(data, dct, dep_variable, max_depth=1)
tree.print_tree()

[], {0: 78.0, 1: 66.0}, (billtype, p=0.01635323632289041, score=5.764489380588449,
groups=[['<missing>', 'Бесплатный'], ['Бюджетный']]), dof=1)
|-- [<missing>, 'Бесплатный'], {0: 34.0, 1: 42.0}, <Invalid Chaid Split> - the max depth
has been reached)
+-- ['Бюджетный'], {0: 44.0, 1: 24.0}, <Invalid Chaid Split> - the max depth has been
reached)
```

Видим, что пропуски были объединены с категорией 'Бесплатный', поэтому пропуски можно импутировать данной категорией.

Импутацию пропусков с помощью CHAID необходимо выполнять после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки, поскольку CHAID использует для работы биннинг на основе децилей и статистические критерии.

Основная рекомендация по работе с пропусками сводится к тому, чтобы всегда руководствоваться здравым смыслом и бизнес-логикой, не надеясь на сложные методы импутации. Приведем примеры применения логического контроля при работе с пропусками.

Необходимые нам данные записаны в файле *Credit OTP_short.csv*. Исходная выборка содержит записи о 15 223 клиентах, классифицированных на два класса: 0 – отклика не было (13 411 клиентов) и 1 – отклик был (1812 клиентов). По каждому наблюдению (клиенту) фиксируются следующие переменные.

Список исходных переменных включает в себя:

- категориальный признак Уникальный идентификатор объекта в выборке [*AGREEMENT_RK*];
- бинарная зависимая переменная Отклик на маркетинговую кампанию [*TARGET*];
- количественный признак Возраст клиента [*AGE*];
- категориальный признак Социальный статус клиента относительно работы [*SOCSTATUS_WORK_FL*];
- категориальный признак Социальный статус клиента относительно пенсии [*SOCSTATUS_PENS_FL*];
- категориальный признак Пол клиента [*GENDER*];
- количественный признак Количество детей клиента [*CHILD_TOTAL*];
- количественный признак Количество иждивенцев клиента [*DEPENDANTS*];
- категориальный признак Образование [*EDUCATION*];
- категориальный признак Семейное положение [*MARITAL_STATUS*];
- категориальный признак Отрасль работы клиента [*GEN_INDUSTRY*];
- категориальный признак Должность [*GEN_TITLE*];
- категориальный признак Форма собственности компании [*ORG_TP_STATE*];
- категориальный признак Отношение к иностранному капиталу [*ORG_TP_FCAPITAL*];
- категориальный признак Направление деятельности внутри компании [*JOB_DIR*];
- категориальный признак Семейный доход [*FAMILY_INCOME*];

- количественный признак *Личный доход клиента в рублях [PERSONAL_INCOME]*;
- категориальный признак *Область регистрации клиента [REG_ADDRESS_PROVINCE]*;
- категориальный признак *Область фактического пребывания клиента [FACT_ADDRESS_PROVINCE]*;
- категориальный признак *Почтовый адрес область [POSTAL_ADDRESS_PROVINCE]*;
- категориальный признак *Область торговой точки, где клиент брал последний кредит [TP_PROVINCE]*;
- категориальный признак *Регион РФ [REGION_NM]*;
- количественный признак *Сумма последнего кредита клиента в рублях [CREDIT]*;
- количественный признак *Первоначальный взнос в рублях [FST_PAYMENT]*;
- количественный признак *Количество месяцев проживания по месту фактического пребывания [FACT_LIVING_TERM]*;
- количественный признак *Время работы на текущем месте в месяцах [WORK_TIME]*.

```
# увеличиваем количество выводимых столбцов
pd.set_option('display.max_columns', 60)
# загружаем набор данных
data = pd.read_csv('Data/Credit OTP_short.csv', sep=';')
# выводим первые пять наблюдений
data.head()
```

	AGREEMENT_RK	TARGET	AGE	SOCSTATUS_WORK_FL	SOCSTATUS_PENS_FL	GENDER	CHILD_TOTAL	DEPENDANTS	EDUCATION	MARITAL_STATUS
0	59910150	0	49		1	0	1	2	1	Среднее специальное Состою в браке
1	59910230	0	32		1	0	1	3	3	Среднее Состою в браке
2	59910525	0	52		1	0	1	4	0	Неполное среднее Состою в браке
3	59910803	0	39		1	0	1	1	1	Высшее Состою в браке
4	59911781	0	30		1	0	0	0	0	Среднее Состою в браке

Выясним, есть ли у нас пропуски.

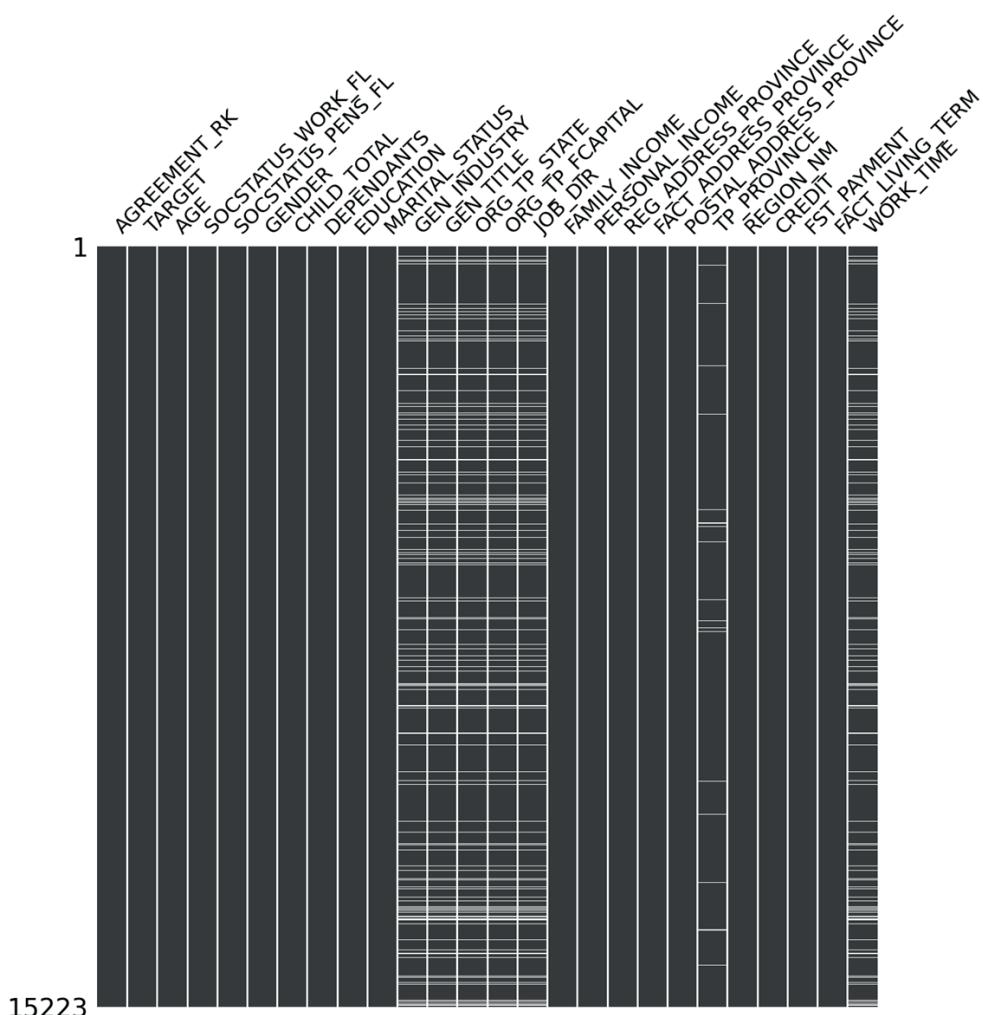
```
# выводим информацию о пропусках  
data.isnull().sum()
```

```
AGREEMENT_RK          0  
TARGET                0  
AGE                   0  
SOCSTATUS_WORK_FL    0  
SOCSTATUS_PENS_FL    0  
GENDER                0  
CHILD_TOTAL           0  
DEPENDANTS            0  
EDUCATION              0  
MARITAL_STATUS         0  
GEN_INDUSTRY          1367  
GEN_TITLE              1367  
ORG_TP_STATE           1367  
ORG_TP_FCAPITAL        1365  
JOB_DIR                1367  
FAMILY_INCOME          0  
PERSONAL_INCOME         0  
REG_ADDRESS_PROVINCE   0  
FACT_ADDRESS_PROVINCE  0  
POSTAL_ADDRESS_PROVINCE 0  
TP_PROVINCE             295  
REGION_NM               1  
CREDIT                  0  
FST_PAYMENT              0  
FACT_LIVING_TERM         0  
WORK_TIME                1368  
dtype: int64
```

Мы видим, что у нас наблюдается пропуск практически одинакового количества значений по категориальным переменным *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR* и количественной переменной *WORK_TIME*.

Теперь воспользуемся удобной библиотекой *missingno* для визуализации пропусков (в виде белых горизонтальных линий).

```
# визуализируем пропуски с помощью missingno  
msno.matrix(data, sparkline=False, figsize=(11, 11));
```



Видим, что пропуски в переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR* и *WORK_TIME* являются синхронными.

Мы могли бы заменить пропуски в категориальных переменных модой, но давайте еще подумаем, чем могли быть вызваны пропуски в этих переменных. Эти переменные так или иначе характеризуют занятость клиента. Возможно, пропуски обусловлены тем, что для определенной части клиентов (неработающих пенсионеров, молодых мам в декретном отпуске, временно неработающих и других категорий) в момент сбора данных невозможно было зафиксировать эти переменные, т. е. вопрос о занятости по отношению к этой части клиентов был неприменим. Применительно к пенсионерам это легко проверить, сопоставив значения данных переменных со значениями переменной *SOCSTATUS_PENS_FL*.

Давайте случайным образом отберем наблюдения, в которых переменная принимает значение 1 (т. е. клиент является пенсионером).

```
# случайно отберем наблюдения, в которых переменная
# SOCSTATUS_PENS_FL принимает значение 1
data[data['SOCSTATUS_PENS_FL'] == 1].sample(frac=0.003, random_state=42)
```

STATUS_PENS_FL	GENDER	CHILD_TOTAL	DEPENDANTS	EDUCATION	MARITAL_STATUS	GEN_INDUSTRY	GEN_TITLE	ORG_TP_STATE	ORG_TP_FCAPITAL
1	1	1	0	Среднее	Вдовец/Вдова	Металлургия/ Промышленность/ Машиностроение	Специалист	Частная компания	Без участия
1	1	1	0	Среднее специальное	Состою в браке		NaN	NaN	NaN
1	0	0	0	Среднее специальное	Не состоял в брачке	Банк/Финансы	Рабочий	Частная компания	Без участия
1	1	0	0	Среднее специальное	Состою в браке		NaN	NaN	NaN
1	1	1	0	Среднее специальное	Состою в браке		NaN	NaN	NaN
1	1	0	0	Среднее специальное	Состою в браке	Государственная служба	Специалист	Государственная комп./учреж.	Без участия

Мы видим, что не всегда значениям 1 переменной *SOCSTATUS_PENS_FL* соответствуют пропуски в переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR* и *WORK_TIME*. Это объясняется тем, что среди клиентов могут быть и работающие пенсионеры. Теперь выясним, сколько у нас наблюдений, в которых переменная *SOCSTATUS_PENS_FL* принимает значение 1 и при этом переменная из рассматриваемого списка имеет пропуск.

```
# создаем список интересующих нас переменных
ptrn = 'GEN_|ORG_|WORK_TIME|JOB'
work_cols = data.columns[data.columns.str.contains(ptrn)].tolist()
# выводим количество наблюдений, в которых переменная SOCSTATUS_PENS_FL
# принимает значение 1 и при этом переменная из рассматриваемого
# списка имеет пропуск
for col in work_cols:
    # записываем условие
    cond = (data[col].isnull()) & (data['SOCSTATUS_PENS_FL'] == 1)
    # записываем частоты
    freq = cond.value_counts()
    # печатаем имя переменной и частоты
    print(col)
    print(freq)
    print("")

GEN_INDUSTRY
False    13857
True     1366
dtype: int64

GEN_TITLE
False    13857
True     1366
dtype: int64

ORG_TP_STATE
False    13857
True     1366
dtype: int64

ORG_TP_FCAPITAL
False    13859
```

```
True      1364  
dtype: int64
```

```
JOB_DIR  
False     13857  
True      1366  
dtype: int64
```

```
WORK_TIME  
False    13856  
True     1367  
dtype: int64
```

Видим, что для наших признаков количество наблюдений, когда у нас интересующая переменная содержала пропуск и переменная *SOCSTATUS_PENS_FL* принимала значение 1 (количество значений True), почти точно совпадает с количеством пропусков. Значит, наша гипотеза верна, и более логично пропуски в категориальных переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR* заменить не модой, а выделить в отдельную категорию 'Не указано' или 'Пенсионер'. Количественную переменную *Время работы на текущем месте в месяцах [WORK_TIME]* из этого списка удалим и пропуски в ней будем импютировать отдельно. Поскольку выполняем импютизацию константным значением, эту процедуру делаем до разбиения на обучающую и тестовую выборки.

```
# удаляем из списка переменную WORK_TIME  
work_cols.remove('WORK_TIME')  
# заменяем пропуски в переменных GEN_INDUSTRY, GEN_TITLE,  
# ORG_TP_STATE, ORG_TP_FCAPITAL, JOB_DIR на "Не указано",  
# если в интересующей нас переменной есть пропуск  
# и при этом переменная SOCSTATUS_PENS_FL имеет значение 1  
for col in work_cols:  
    data[col] = np.where(cond, 'Не указано', data[col])
```

Взглянем на результаты. Видим, что почти все пропуски в переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR* заменены на категорию 'Не указано'.

```
# смотрим результаты  
for col in work_cols:  
    print(data[col].value_counts(dropna=False))  
    print("")
```

Торговля	2385
Другие сферы	1709
Не указано	1367
Металлургия/Промышленность/Машиностроение	1356
Государственная служба	1286
Здравоохранение	1177
Образование	998
Транспорт	787
Сельское хозяйство	702
Строительство	573
Коммунальное хоз-во / Дорожные службы	533

Ресторанный бизнес / Общественное питание	408
Наука	403
Нефтегазовая промышленность	225
Сборочные производства	172
Банк/Финансы	169
Энергетика	145
Развлечения/Искусство	141
ЧОП/Детективная д-ть	136
Информационные услуги	108
Салоны красоты и здоровья	99
Информационные технологии	85
Химия/Парфюмерия/Фармацевтика	63
СМИ/Реклама/PR-агентства	49
Юридические услуги / нотариальные услуги	47
Страхование	28
Туризм	20
Недвижимость	16
Управляющая компания	12
Логистика	11
Подбор персонала	8
Маркетинг	4
NaN	1

Name: GEN_INDUSTRY, dtype: int64

Специалист	7009
Рабочий	3075
Не указано	1367
Служащий	904
Руководитель среднего звена	697
Работник сферы услуг	563
Высококвалифиц. специалист	549
Руководитель высшего звена	427
Индивидуальный предприниматель	217
Другое	177
Руководитель низшего звена	136
Военнослужащий по контракту	88
Партнер	13
NaN	1

Name: GEN_TITLE, dtype: int64

Частная компания	6523
Государственная комп./учреж.	6111
Не указано	1367
Индивидуальный предприниматель	957
Некоммерческая организация	243
Частная ком. с инос. капиталом	21
NaN	1

Name: ORG_TP_STATE, dtype: int64

Без участия	13685
Не указано	1367
С участием	170
NaN	1

Name: ORG_TP_FCAPITAL, dtype: int64

Участие в основ. деятельности 11451

Не указано	1367
Вспомогательный техперсонал	1025
Бухгалтерия, финансы, планир.	481
Адм-хоз. и трансп. службы	279
Снабжение и сбыт	217
Служба безопасности	164
Кадровая служба и секретариат	101
Пр-техн. обесп. и телеком.	75
Юридическая служба	53
Реклама и маркетинг	9
NaN	1
Name: JOB_DIR, dtype: int64	

Оставшиеся пропуски мы можем как раз заменить модой после разбиения на обучающую и тестовую выборки.

Теперь приступим к импутации пропусков переменной *Время работы на текущем месте в месяцах [WORK_TIME]*. Напомним, речь идет о времени работы на текущем месте в месяцах. Если в переменной *WORK_TIME* есть пропуск и при этом переменная *SOCSTATUS_PENS_FL* принимает значение 1, заменяем пропуск нулем. Поскольку выполняем импутацию константным значением, эту процедуру делаем до разбиения на обучающую и тестовую выборки.

```
# выполняем импутацию переменной WORK_TIME нулями
cond = (data['WORK_TIME'].isnull()) & (data['SOCSTATUS_PENS_FL'] == 1)
data['WORK_TIME'] = np.where(cond, 0, data['WORK_TIME'])
```

Посмотрим количество пропусков в переменной *WORK_TIME*.

```
# посмотрим количество пропусков в WORK_TIME
data['WORK_TIME'].isnull().sum()
```

1

Теперь в переменной *WORK_TIME* один пропуск.

При фиксации переменной *WORK_TIME* может быть ситуация, что разница между возрастом и временем работы может быть меньше 16 лет. Например, у 30-летнего время работы в годах составляет 20 лет, получается, он работает с 10 лет. Или человеку 30 лет, а работает он 100 лет, получается, он начал работать за 70 лет до своего рождения.

Итак, давайте проверим переменную *WORK_TIME* на наличие этой ошибки. Не забываем месяцы работы перевести в года, поскольку возраст, с которым мы будем сравнивать, измеряется в годах.

```
# проверяем, есть ли наблюдения, в которых разница между
# возрастом и временем работы в годах меньше 16 (например,
# у 30-летнего время работы в годах составляет 20 лет,
# получается, он работает с 10 лет)
data[(data['AGE'] - data['WORK_TIME'] / 12) < 16][['AGE', 'WORK_TIME']]
```

AGE	WORK_TIME
148	43
676	53
1092	45
2179	23
2373	31
2532	58
2983	44
3039	40
3323	39
3797	39
4155	40
4191	33
	360.0
	780.0
	1312.0
	156.0
	187.0
	4320.0
	600.0
	480.0
	612.0
	300.0
	328.0
	228.0

Итак, мы видим, что у нас есть наблюдения, в которых время работы пре-вышает возраст. Клиент с индексом 8984, например, работает аж 238 996 лет!

В наблюдениях, в которых разница между возрастом и временем работы в годах меньше 16 лет, значения переменной *WORK_TIME* заменим пропусками, а после разбиения на обучающую и тестовую выборки заменим пропуски минимальными значениями переменной *WORK_TIME*, вычисленными по категориям переменной *GEN_INDUSTRY*. У нас нет достоверной информации о времени работы в годах по этим клиентам. Замена пропусков нуле-выми значениями может быть слишком пессимистичной стратегией, а им-путация средним или медианой, наоборот, может быть слишком оптими-стичной стратегией. Замена пропусков глобальными минимальными значе-ниями может быть грубой, поэтому возьмем их с учетом сферы занятости. Отметим, что в каждом банке будет свой регламент обработки пропусков, основанный на априорных знаниях и бизнес-логике. На примере этой ситу-ации вы видите, что наша задача заключается не только в поиске пропусков и их замене, но и в поиске недостоверных данных, приравнивании их к про-пускам и последующей замене. Как вариант можно было вообще удалить эти наблюдения.

```
# заменяем значения на пропуски по условию
cond = (data['AGE'] - data['WORK_TIME'] / 12) < 16
data['WORK_TIME'] = np.where(cond, np.NaN, data['WORK_TIME'])
```

Теперь обратим внимание на переменную *Количество месяцев проживания по месту фактического пребывания [FACT_LIVING_TERM]*. В ней нет пропусков, но здесь необходимо учесть момент, что часто бывает ситуация, когда из-за ошибок ввода данных стаж проживания превышает возраст клиента.

Давайте проверим переменную *FACT_LIVING_TERM* на наличие этой ошиб-ки. Не забываем месяцы проживания перевести в года, поскольку возраст, с которым мы будем сравнивать, измеряется в годах.

```
# проверяем, есть ли наблюдения, в которых количество
# лет проживания по месту фактического пребывания
# превышает возраст
data[data['FACT_LIVING_TERM'] / 12 > data['AGE']][['AGE', 'FACT_LIVING_TERM']]
```

AGE	FACT_LIVING_TERM
988	51
1420	60
1640	23
2071	41
2373	31
3711	23
5369	51
5681	31
5772	42
5888	45
6186	49
	28101997

Видим, что у нас есть наблюдения, в которых количество лет проживания по месту фактического пребывания превышает возраст. Клиент с индексом 6186, например, живет $28\ 101\ 997 / 12 = 2\ 341\ 833$ года и, возможно, является прапорителем человека!

Наблюдения, в которых количество лет проживания по месту фактического пребывания превышает возраст, записываем как пропуски, а затем после разбиения на обучающую и тестовую выборки импутируем их минимальными значениями, вычисленными в категориях переменной *Область фактического пребывания клиента [FACT_ADDRESS_PROVINCE]*.

```
# заменяем значения на пропуски по условию
cond = data['FACT_LIVING_TERM'] / 12 > data['AGE']
data['FACT_LIVING_TERM'] = np.where(cond,
                                      np.Nan,
                                      data['FACT_LIVING_TERM'])
```

Сейчас обратим внимание на пропуски в переменной *Область торговой точки*, где клиент брал последний кредит [*TP_PROVINCE*]. Мы могли бы заменить пропуски модой после разбиения на обучающую и тестовую выборки, но логичнее использовать уже имеющуюся у нас информацию – переменную *Область фактического пребывания клиента [FACT_ADDRESS_PROVINCE]*. Скорее всего, человек будет брать кредит там, где фактически проживает (если исключить ситуацию переезда). В противном случае получится так: у нас пропуск в переменной *TP_PROVINCE*, заменяя модой – категорией *Краснодарский край*, а соответствующее значение в переменной *FACT_ADDRESS_PROVINCE* – *Московская область*. У нас – нелогичная ситуация: человек живет в Московской области, а кредит взял в Краснодарском крае.

```
# пропуски в переменной TP_PROVINCE заменим значением
# переменной FACT_ADDRESS_PROVINCE
data['TP_PROVINCE'] = np.where(data['TP_PROVINCE'].isnull(),
                                data['FACT_ADDRESS_PROVINCE'],
                                data['TP_PROVINCE'])
```

Теперь разбиваем набор на обучающую и тестовую выборки.

```
# создаем обучающий массив признаков, тестовый массив признаков,
# обучающий массив меток, тестовый массив меток
train, test, y_train, y_test = train_test_split(
    data.drop('TARGET', axis=1),
    data['TARGET'],
    test_size=.3,
    stratify=data['TARGET'],
    random_state=100)
```

Вспоминаем про оставшиеся пропуски в переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR*. Заменим их на моду с помощью класса *SimpleImputer*.

```
# заменяем пропуски в переменных GEN_INDUSTRY, GEN_TITLE,
# ORG_TP_STATE, ORG_TP_FCAPITAL, JOB_DIR модами
simp = SimpleImputer(strategy='most_frequent')
simp.fit(train[work_cols])
train[work_cols] = simp.transform(train[work_cols])
test[work_cols] = simp.transform(test[work_cols])
```

Вспоминаем про пропуски в переменных *WORK_TIME* и *FACT_LIVING_TERM*. Здесь воспользуемся нашим классом *GroupImputer*.

```
# заменяем пропуски в переменной WORK_TIME минимальными
# значениями, вычисленными по категориям GEN_INDUSTRY
imp = GroupImputer(['GEN_INDUSTRY'],
                    agg_col='WORK_TIME',
                    agg_func='min')
imp.fit(train)
train = imp.transform(train)
test = imp.transform(test)

# заменяем пропуски в переменной FACT_LIVING_TERM минимальными
# значениями, вычисленными по категориям FACT_ADDRESS_PROVINCE
imp = GroupImputer(['FACT_ADDRESS_PROVINCE'],
                    agg_col='FACT_LIVING_TERM',
                    agg_func='min')
imp.fit(train)
train = imp.transform(train)
test = imp.transform(test)
```

13. Обработка выбросов

Выброс – это значение, которое значительно отличается от остальных. Например, если наугад измерять температуру предметов в комнате, получим цифры от 18 до 22 °C, но радиатор отопления будет иметь температуру в 70 °C. Значение 70 °C будет выбросом.

Можно выделить следующие причины выбросов:

- ошибки измерения (менеджер, фиксируя доход работающего пенсионера, ввел вместо 20 000 сумму в 200 000 рублей; произошла поломка датчика, в силу которой получаем очень большие показания; при импорте данных из базы данных произошло смещение десятичного разделителя – вместо значений утилизации 0,6 получаем 0,006, или произошло умножение на 100 и получили 60);
- особенности конструирования признаков (например, используем переменную – отношение двух переменных, у которой знаменатель близок к нулю, например делим 100 на 0,001 и получаем 100 000);
- объективная природа данных (мы выдаем кредиты, размер кредита обычно варьирует в диапазоне от 10 тыс. до 100 тыс. долларов, и вдруг мы видим кредит 20 млн долларов – за кредитом пришел Билл Гейтс; мы фиксируем количество просрочек 30+, у нас могут быть значения 0, 1, 2, 3, 4, 5, 6, 7, 96 и 98, где 96 и 98 – это на самом деле не количество просрочек, а служебные коды, указывающие на то, что с клиентом невозможно связаться).

Всегда нужно понять причину появления выбросов. От этого зависит выбор способа борьбы с выбросами. Если выбросы – это ошибка измерения, можно сделать винзоризацию или импутацию. Если выбросы – это не выбросы, а переменная действительно так распределена, то выполняем преобразования (логарифм, корни четвертой, третьей, второй степени). Эти преобразования полезны тем, что сдвигают слишком большие значения к среднему значению. Если выбросы из-за того, что переменная – отношение двух других (utiлизация, коэффициент долговой нагрузки) и знаменатель близок к нулю, то можно поменять определение переменной, использовать псевдосчетчики (лапласовское сглаживание).

Разберем на конкретных примерах.

Итак, мы выдаем кредиты и фиксируем сумму кредита в базе данных. Аналитик, работая с базой, видит, что размер кредита варьирует в диапазоне от 10 тыс. до 100 тыс. долларов, и вдруг встречает наблюдение, в котором размер кредита равен 20 млн долларов. Начинающий аналитик часто спешит избавиться от выбросов, например делает винзоризацию – процедуру, в ходе которой выбросам присваиваются значения, равные соответственно нижней или верхней границе (например, 5-му и 95-му процентилю¹), относительно которых идентифицируются выбросы (двухсторонняя винзоризация). Нередко

¹ Процентиль – показатель того, какой процент значений находится ниже определенного уровня. Например, 90 % значений данных находятся ниже 90-го процентиля, а 10 % значений данных находятся ниже 10-го процентиля.

винзоризацию выполняют при задании только одной границы – верхней или нижней (односторонняя винзоризация).

Допустим, мы выполнили одностороннюю винзоризацию по верхней границе, приравняли значение 20 млн долларов к значению, соответствующему 99-му процентилю. Но позже выясняется, что это пришел Билл Гейтс и заполнил анкету, ему выдали кредит на сумму больше, чем 99-й процентиль, потому что он намного богаче, чем 99-й процентиль. Выдав один кредит Биллу Гейтсу, банк может заработать больше денег, чем на всех остальных клиентах вместе взятых. И без риска. Это пример того, когда выброс обусловлен объективными причинами и важно понимать, что выброс – это не всегда плохо. Если мы выполним винзоризацию, потеряем ценную информацию. В свое время кризис 2008 года был обусловлен тем, что многие аналитики недооценили риски, исключив из данных объективные выбросы.

Еще актуальнее тема выбросов для медицинских исследований. Например, вы предсказываете смертность от диабета по уровню определенного гормона в крови. Ограничиваете уровень гормона по 99-му процентилю. А в результате модель работает плохо, потому что 99 % пациентов выживают, а вся критически важная информация сосредоточилась выше 99-го процентиля. Для модели процентиль уровня 99,1 и процентиль уровня 99,9 выглядят одинаково, а смертность разная. Поэтому вместо винзоризации надо делать преобразование.

Возьмем другой пример. Менеджер, фиксируя доход работающего пенсионера, ввел вместо 20 000 сумму в 200 000 рублей. Данный выброс обусловлен ошибкой и нуждается в замене средним или медианой.

Выбросы ухудшают качество линейных моделей. Приведем график, у нас есть признак x и зависимая переменная y .

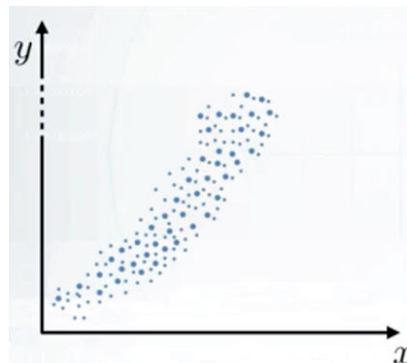


Рис. 14 Отсутствие выбросов

Если мы обучаем простую линейную модель, ее прогнозы могут выглядеть так же, как красная линия (рис. 15). Но если у вас есть один выброс – огромное значение признака x , прогнозы линейной модели будут больше похожи на фиолетовую линию.

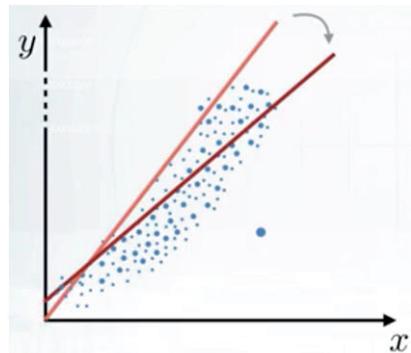


Рис. 15 Появление выброса в признаке

То же самое касается не только значений признаков, но и значений зависимой переменной. Например, представим, что у нас есть модель, обученная на данных со значениями зависимой переменной от нуля до единицы. Давайте подумаем, что произойдет, если мы добавим в обучающие данные новое наблюдение со значением зависимой переменной 1000. Когда мы заново обучим модель, она будет предсказывать аномально высокие значения.

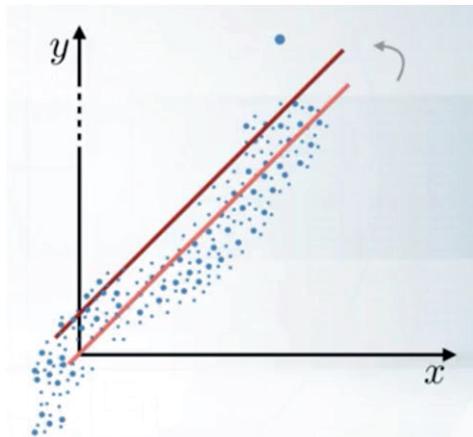


Рис. 16 Появление выброса в зависимой переменной

Выбросы не дадут выполнить полноценную стандартизацию, и в решении будут доминировать переменные большего масштаба, разный масштаб переменных вновь негативно отразится на сходимости градиентного спуска.

Деревья решений устойчивы к выбросам, поскольку разбиения основаны на количестве наблюдений внутри диапазонов значений, выбранных для расщепления, а не на абсолютных значениях. Если у нас есть наблюдение со значением 99 999, дерево может создать два узла, например с правилами

« ≤ 5 » и « > 5 », и отнести наблюдение со значением 99 999 в правый узел. Ниже приведены примеры, которые показывают, как дерево обрабатывает выбросы. Здесь у нас есть признак *credit* и зависимая переменная *response*.

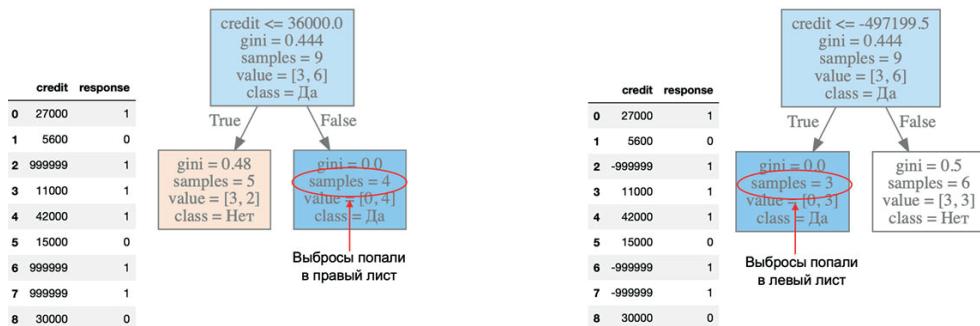


Рис. 17 Обработка выбросов деревьями решений

Кроме того, приведенный рисунок наглядно поясняет, почему для древовидных алгоритмов эффективной может быть импутация значением вне диапазона имеющихся значений. Если имптуировать пропуск значением, которое будет больше любого имеющегося значения, то в дереве можно будет выбрать такое разбиение по этому признаку, что все наблюдения с известными значениями пойдут в левый узел, а все наблюдения с пропусками – в правый (левая часть рисунка).

Если же имптуировать пропуск значением, которое будет меньше любого имеющегося значения, то в дереве можно будет выбрать такое разбиение по этому признаку, что все наблюдения с пропусками пойдут в левый узел, а все наблюдения с известными значениями – в правый (правая часть рисунка).

14. Описательные статистики

14.1. Пифагорейские средние, медиана и мода

К пифагорейским средним относится арифметическое среднее, гармоническое среднее и геометрическое среднее.

Арифметическое среднее вычисляется как сумма всех значений, поделенная на их количество:

$$AM = \frac{a_1 + a_2 + \dots + a_n}{n}.$$

Геометрическое среднее получается от перемножения значений и извлечения из этого произведения корня, показатель которого равен количеству этих значений:

$$GM = \sqrt[n]{a_1 \times a_2 \times \dots \times a_n}.$$

Гармоническое среднее вычисляется как количество значений, поделенное на сумму обратных этим значениям:

$$HM = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \dots + \frac{1}{a_n}}.$$

Для всех наборов данных с положительными значениями, содержащих хотя бы одну пару неодинаковых значений, среднее гармоническое всегда является наименьшим из трех средних, а среднее арифметическое всегда является наибольшим из трех средних, а среднее геометрическое всегда находится между ними. Если все значения в непустом наборе данных равны, три средних всегда равны друг другу: например, гармоническое, геометрическое и арифметическое средние в наборе {2, 2, 2} равны 2. Поскольку гармоническое среднее списка чисел склоняется в пользу наименьших элементов списка, оно стремится (по сравнению с арифметическим средним) смягчить воздействие больших выбросов и усилить влияние небольших выбросов. Например, это свойство гармонического среднего использовано в *F*-мере, которая является гармоническим средним точности и полноты.

Медиана занимает центральное положение в упорядоченном ряду чисел. Медиана – это такое число, что половина из значений выборки больше него, а другая половина не больше него. Например, у нас есть набор с различными элементами {4, 2, 1, 3, 7, 6, 5}, получили упорядоченный ряд {1, 2, 3, 4, 5, 6, 7}. Медианой здесь будет 4, половина значений (5, 6, 7) больше 4, половина значений (1, 2, 3) не больше 4. Например, у нас есть набор с повторяющимися значениями {3, 3, 3, 1, 2, 2, 2}, получили упорядоченный ряд {1, 2, 2, 2, 3, 3, 3}. Медианой здесь будет 2, половина значений (3, 3, 3) больше 2, половина значений (1, 2, 2) не больше 2.

В более общем случае медиану можно найти, упорядочив элементы выборки по возрастанию или убыванию и взяв средний элемент. Например, набор $\{11, 9, 3, 5, 5\}$ после упорядочивания превращается в упорядоченный ряд $\{3, 5, 5, 9, 11\}$, и медианой является число 5.

Если имеется чётное количество случаев и два средних значения различаются, то медианой, по определению, может служить любое число между ними. Например, в наборе $\{1, 3, 5, 7\}$ медианой может служить любое число из интервала $(3, 5)$. На практике в этом случае чаще всего используют среднее арифметическое двух средних значений (в примере выше это число $(3 + 5) / 2 = 4$).

Для выборок с чётным числом элементов можно также ввести понятие «нижней медианы» (элемент с номером $n/2$ в упорядоченном ряду из n элементов, в примере выше это число 3) и «верхней медианы» (элемент с номером $(n + 2) / 2$, в примере выше это число 5). Эти понятия определены не только для числовых данных, но и для любой порядковой шкалы.

Можно также сказать, что медиана является 50-м процентилем.

Основная особенность медианы при описании данных по сравнению со средним значением заключается в том, что медиана более robustна, ее не искажает небольшая доля чрезвычайно больших или малых значений и, следовательно, она обеспечивает лучшее представление о «типичном» значении. Например, медианный доход может быть лучшим способом дать представление о «типичном» доходе, потому что распределение доходов может быть очень асимметричным.

Допустим, 10 человек зарабатывают: 101, 102, 103, 104, 105, 106, 107, 108, 109 и один зарабатывает 10 тысяч монет. Тогда средняя зарплата будет больше тысячи монет, но медианная зарплата (зарплата человека, занимающего центральное положение в ряду) будет всего 105 монет.

Мода – это самое часто встречающееся значение. Мод может быть несколько.

	age	income	region
0	NaN	NaN	MSK
1	23.0	4560.55	MSK
2	24.0	NaN	NaN
3	30.0	NaN	EKAT
4	NaN	7888.10	NaN
5	55.0	9000.50	SPB
6	37.0	NaN	SPB

Рис. 18 Несколько мод в переменной region

14.2. Квантиль

Квантиль (мужской род, ударение падает на последний слог) – значение, которое заданная случайная величина не превышает с фиксированной вероятностью. 0,25-квантиль (говорят *квантиль уровня 0,25*) – это значение, ниже которого будет лежать 25 % значений числового ряда, а выше – 75 % значений числового ряда.

Самым известным квантилем является 0,5-квантиль или медиана – значение, которое делит упорядоченный числовой ряд пополам, то есть ровно половина остальных значений больше него, а другая половина меньше его.

Среди всех возможных квантилей обычно выделяют определенные семейства.

Квантили одного семейства делят упорядоченный числовой ряд (диапазон значений признака) на заданное число равнодоступных частей. Семейство определяется тем, сколько частей получается. Наиболее популярными квантилями являются квартили, разбивающие упорядоченный числовой ряд на 4 равнодоступные части таким образом, что 25 % единиц совокупности будут меньше по величине Q_1 (первого квартиля); 25 % будут заключены между Q_1 и Q_2 (между первым и вторым квартилями); 25 % – между Q_2 и Q_3 (между вторым и третьим квартилями); остальные 25 % превосходят Q_3 (третий квартиль). Первый квартиль – это 0,25-квантиль, второй квартиль – это 0,50-квартиль, третий квартиль – это 0,75-квартиль. А еще бывают квинтили – когда разбивают на 5 равнодоступных частей, секстили – когда разбивают на 6 равнодоступных частей, септили – на 7 равнодоступных частей, октили – на 8 равнодоступных частей, децили – на 10 равнодоступных частей, виджинтили – когда разбивают на 20 равнодоступных частей, процентили – на 100 равнодоступных частей. Разница между нижним и верхним квартилями будет межквартильным размахом.

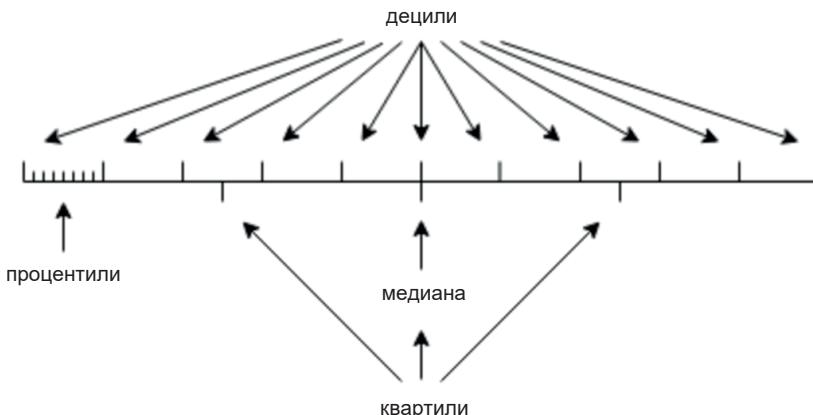


Рис. 19 Различные семейства квантилей

14.3. ДИСПЕРСИЯ И СТАНДАРТНОЕ ОТКЛОНЕНИЕ

Дисперсия позволяет нам понять, насколько сильно значения отклоняются от среднего. Она определяется по следующей формуле:

$$s_n^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2.$$

Для каждого наблюдения мы вычисляем разницу между имеющимся значением и средним значением. Разница может быть положительным или отрицательным значением, поэтому мы возводим ее в квадрат, чтобы убедиться в том,

что отрицательные значения оказывают кумулятивный эффект на результат. Затем эти значения суммируются и делятся на количество наблюдений минус 1, в итоге получаем аппроксимированное среднее значение различий.

Поскольку используется возведение в квадрат, единица измерения дисперсии отличается от единицы измерения фактических значений.

Вышеприведенную формулу называют смещенной оценкой дисперсии. Поскольку наблюдаемые значения будут ближе к выборочному среднему значению, а не среднему значению генеральной совокупности, то стандартное отклонение, которое вычисляется с использованием отклонений от выборочного среднего, недооценивает желаемое стандартное отклонение генеральной совокупности. Использование $n - 1$ вместо n немножко увеличивает получаемый результат.

Поэтому модифицированную формулу $s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$ называют несмешенной оценкой дисперсии.

Стандартное отклонение определяется путем вычисления квадратного корня из дисперсии и имеет следующую формулу:

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}.$$

Как и дисперсия, стандартное отклонение отражает разницу между всеми значениями и средним значением. Используя квадратный корень из дисперсии, стандартное отклонение измеряется в тех же единицах, что и значения исходного набора данных.

14.4. КОРРЕЛЯЦИЯ И КОВАРИАЦИЯ

Корреляция – это согласованные изменения двух (парная корреляция) или большего количества признаков (множественная корреляция). Суть корреляции заключается в том, что при изменении значения одной переменной происходит изменение (уменьшение или увеличение) другой(их) переменной(ых).

При положительной корреляции более высоким значениям одного признака соответствуют более высокие значения другого, а более низким значениям одного признака – низкие значения другого.

При отрицательной корреляции более высоким значениям одного признака соответствуют более низкие значения другого, а более низким значениям одного признака – высокие значения другого.

Коэффициент корреляции – двумерная описательная статистика, количественная мера связи (совместной изменчивости) двух переменных. Коэффициент корреляции всегда будет принимать значение от -1 до 1 . Он вычисляется путем деления ковариации переменных на произведение стандартных отклонений обеих переменных:

$$r_{XY} = \frac{\text{cov}_{XY}}{\sigma_X \sigma_Y} = \frac{\sum (X - \bar{X})(Y - \bar{Y})}{\sqrt{\sum (X - \bar{X})^2 (Y - \bar{Y})^2}}.$$

Ковариация показывает, как связаны две переменные. Положительная ковариация означает, что переменные связаны прямой зависимостью, а отрицательная ковариация указывает на обратную зависимость. Ковариация позволяет определить, связаны ли значения, однако она не дает понимания, насколько сильно значения переменных изменяются вместе, поскольку является ненормированной величиной. Чтобы измерить силу, с которой обе переменные меняются вместе, нам и нужно вычислить корреляцию. Корреляция стандартизирует меру совместной изменчивости двух переменных и, следовательно, показывает, насколько сильно обе переменные могут изменяться вместе (насколько сильно они связаны друг с другом).

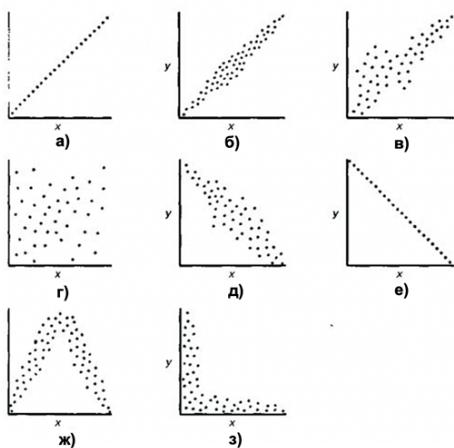
Если коэффициент корреляции равен -1 , переменные имеют идеальную отрицательную корреляцию (обратно коррелированы) и меняются в противоположных направлениях. Если одна переменная увеличивается, другая переменная пропорционально уменьшается. Отрицательный коэффициент корреляции, превышающий -1 , но меньше 0 , указывает на отрицательную корреляцию, отличную от идеальной. При этом сила отрицательной корреляции растет по мере приближения к -1 .

Если коэффициент корреляции равен 0 , между переменными нет связи.

Если коэффициент корреляции равен 1 , переменные имеют идеальную положительную корреляцию. Это означает, что если одна переменная изменяется на заданную величину, вторая переменная изменяется пропорционально в том же самом направлении. Положительный коэффициент корреляции меньше 1 , но больше 0 указывает на положительную корреляцию, отличную от идеальной. Сила положительной корреляции растет по мере приближения к 1 .

Если между двумя исследуемыми признаками установлена тесная зависимость, то из этого еще не следует их причинная взаимообусловленность (каузальность). За счет эффектов одновременного влияния неучтенных факторов смысл истинной связи может искажаться. Поэтому такую корреляцию часто называют «ложной».

Пример ложной корреляции – положительная корреляция между количеством гнездовий аистов и рождаемостью. Если взять любую деревню в ареале обитания аистов, посчитать там количество аистов, а потом сравнить с рождаемостью, то выяснится, что существует прямая связь: чем больше в деревне аистов, тем больше там рождается детей. На самом деле больше детей рождается в более крупных деревнях, где одновременно есть больше строений, подходящих для сооружения гнезд аистов. Таким образом, здесь «третьим фактором» будет размер деревни.

Примеры корреляций

- а) строгая положительная корреляция
 б) сильная положительная корреляция
 в) слабая положительная корреляция
 г) нулевая корреляция
 д) отрицательная корреляция
 е) строгая отрицательная корреляция
 ж) нелинейная корреляция
 з) нелинейная корреляция

Рис. 20 Примеры корреляций

Для порядковых переменных используются следующие коэффициенты корреляции:

- коэффициент ранговой корреляции Спирмена;
- коэффициент ранговой корреляции Кендалла;
- коэффициент ранговой корреляции Гудмена–Краскела.

Если, по меньшей мере, одна из двух переменных имеет порядковую шкалу либо не является нормально распределённой, используется ранговая корреляция Спирмана или Кендалла. Применение коэффициента Кендалла предпочтительно, если в исходных данных имеются выбросы.

Таблица 3 Коэффициенты корреляции

Типы шкал		Меры связи
Переменная X	Переменная Y	
Интервальная или отношений	Интервальная или отношений	Коэффициент Пирсона
Порядковая, интервальная или отношений	Порядковая, интервальная или отношений	Коэффициент Спирмена
Порядковая	Порядковая	Коэффициент Кендалла
Дихотомическая	Дихотомическая	Коэффициент фи
Дихотомическая	Порядковая	Рангово-бисериальный коэффициент
Дихотомическая	Интервальная или отношений	Точечно-бисериальный коэффициент
Интервальная	Порядковая	–

Давайте вычислим коэффициент корреляции Пирсона между двумя интервальными переменными *Вес* и *Рост*.

```
# импортируем необходимые библиотеки и функции
import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from sklearn.model_selection import train_test_split

# загружаем данные
data = pd.read_csv('Data/Corr.csv', sep=';', index_col=0)
# выводим наблюдения датафрейма
data
```

	Вес	Рост	Наличие_медалей
Дима	72	160	1
Гриша	66	144	0
Миша	68	154	0
Коля	74	210	1
Федя	68	182	1
Рома	64	159	0

Сначала вычислим коэффициент корреляции с помощью метода `.corr()` библиотеки pandas.

```
# вычисляем коэффициент корреляции Пирсона
# с помощью метода .corr() библиотеки pandas
data['Вес'].corr(data['Рост'])
```

0.6791907247260153

Теперь вычислим коэффициент корреляции по формуле $r_{XY} = \frac{\text{cov}_{XY}}{\sigma_X \sigma_Y}$.

```
# вычисляем коэффициент корреляции Пирсона вручную
data['Вес'].cov(data['Рост']) / (data['Вес'].std() * data['Рост'].std())
0.6791907247260153
```

Теперь вычислим коэффициент корреляции по формуле

$$r_{XY} = \frac{\sum z_{X_i} z_{Y_i}}{n-1},$$

где $z_{X_i} = \frac{X_i - \bar{X}}{S_X}$ и $z_{Y_i} = \frac{Y_i - \bar{Y}}{S_Y}$.

```
# еще можно вычислить корреляции Пирсона так
X = (data['Вес'] - data['Вес'].mean()) / data['Вес'].std()
Y = (data['Рост'] - data['Рост'].mean()) / data['Рост'].std()
r_XY = sum(X * Y) / (len(data) - 1)
r_XY

0.6791907247260153
```

Теперь вычислим коэффициент корреляции Пирсона и p -значение с помощью функции `pearsonr()` библиотеки `scipy`.

```
# теперь вычислим коэффициент корреляции Пирсона и p-значение
# с помощью функции pearsonr() библиотеки scipy
weight = np.array(data['Вес'])
height = np.array(data['Рост'])
stats.pearsonr(weight, height)

(0.6791907247260152, 0.13786926734093197)
```

А сейчас вычислим точечный бисериальный коэффициент корреляции между интервальной переменной *Рост* и дихотомической переменной *Наличие_медалей* (0 – Нет медалей, 1 – Есть медали).

```
# вычисляем точечный бисериальный коэффициент
medals = np.array(data['Наличие_медалей'])
stats.pointbiserialr(height, medals)

PointbiserialrResult(correlation=0.7230871254698281, pvalue=0.10440416804243928)
```

Вычислим его вручную по формуле $r_{bis} = \frac{\bar{y}_1 - \bar{y}_0}{\sigma} \times \sqrt{\frac{n_1 \times n_0}{n(n-1)}}$.

```
# теперь вычислим его вручную
y_mean_0 = data[data['Наличие_медалей'] == 0].mean()['Рост']
y_mean_1 = data[data['Наличие_медалей'] == 1].mean()['Рост']
std = data['Рост'].std()
n0 = data['Наличие_медалей'].value_counts()[0]
n1 = data['Наличие_медалей'].value_counts()[1]
n = len(data)
r_bis = ((y_mean_1 - y_mean_0) / std) * np.sqrt((n1 * n0) / (n * (n - 1)))
r_bis
```

0.7230871254698279

Коррелированность двух или нескольких признаков называется мультиколлинеарностью. Одной из важнейших предпосылок линейных моделей является отсутствие мультиколлинеарности. При сильной мультиколлинеарности матрица нормальных уравнений для линейной регрессии становится плохо обусловленной, ее определитель близок нулю, и это приводит к следующим явлениям:

- сильный разброс оценок коэффициентов регрессии (большие положительные и большие отрицательные оценки коэффициентов регрессии, значительно выше 1,0 по модулю);

- резкое изменение оценок коэффициентов регрессии при добавлении или удалении признака;
- неправильный знак перед коэффициентом регрессии (противоречит здравому смыслу и бизнес-логике);
- присутствие в модели большого количества статистически незначимых оценок коэффициентов регрессии.

Чем плохи большие значения коэффициентов? Большие коэффициенты делают возможным сильное изменение прогнозируемой величины при небольшом изменении признаков. Вариабельность оценок коэффициентов регрессии не позволит судить о степени влияния признака на зависимую переменную и построить статистически устойчивую и точную модель.

Для борьбы с мультиколлинеарностью в линейных моделях применяется регуляризация.

Деревья решений более устойчивы к корреляциям признаков, однако оценки важностей высококоррелированных признаков будут смещеными. Важность одинаково распределится по двум высококоррелированным признакам (будто бы признаки поделились друг с другом оценкой важности).

14.5. ПОЛУЧЕНИЕ СВОДКИ ОПИСАТЕЛЬНЫХ СТАТИСТИК

В БИБЛИОТЕКЕ PANDAS

С помощью метода `.describe()` можно вывести сводку описательных статистик по количественным переменным.

Приведем пример из задачи прогнозирования отклика ОТП Банка.

# смотрим статистики для количественных переменных											Пример взят из задачи прогнозирования отклика ОТП Банка	
	AGE	CHILD_TOTAL	DEPENDANTS	PERSONAL_INCOME	OWN_AUTO	CREDIT	TERM	FST_PAYMENT	FACT_LIVING_TERM	WORK_TIME		
количество непропущенных значений	count	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	13855.000	
среднее значение	mean	40.406	1.099	0.645	13853.836	0.116	14667.959	8.101	3398.563	3039.340	292.212	
стандартное отклонение	std	11.601	0.995	0.812	9015.468	0.321	12147.873	4.094	5158.109	262455.437	24364.832	
квадтили	min	21.000	0.000	0.000	24.000	0.000	20000.000	3.000	0.000	-26.000	1.000	
	25%	30.000	0.000	0.000	8000.000	0.000	6500.000	6.000	1000.000	41.000	24.000	
	50%	39.000	1.000	0.000	12000.000	0.000	11550.000	6.000	2000.000	108.000	48.000	
	75%	50.000	2.000	1.000	17000.000	0.000	19170.000	10.000	4000.000	204.000	110.000	
	max	67.000	10.000	7.000	250000.000	2.000	119700.000	36.000	140000.000	28101997.000	2867859.000	

Обращаем внимание на:

- пропуски в переменной `WORK_TIME`;
- отрицательное минимальное значение `FACT_LIVING_TERM` (противоречит здравому смыслу);
- нулевые минимальные значения переменных, при конструировании новых признаков на базе таких переменных нужно быть особо внимательным, т. к. при делении на ноль могут появиться бесконечные значения (*infinite values*);
- отмечаем аномально большие максимальные значения переменных `FACT_LIVING_TERM` и `WORK_TIME`, например такие значения могут снизить качество линейной модели.

Возьмем еще один пример из промышленной задачи.

смотрим статистики для количественных переменных
data.describe()

	1	2	3	4
количество непропущенных значений	count	820529.000000	808921.000000	831550.000000
среднее значение	mean	11.012513	12.329729	11.344055
стандартное отклонение	std	99.986889	0.796650	1.662548
	min	-480.088690	7.000000	3.000000
квартили	25%	-56.357813	12.000000	10.000000
	50%	11.029669	13.000000	11.000000
	75%	78.379658	13.000000	13.000000
	max	545.896248	13.000000	15.000000
				128.900592

Обращаем внимание на:

- примерно одинаковые средние и медианы (указывает на распределение, близкое к нормальному).

Сводку статистик можно выводить и по категориальным переменным.

смотрим статистики для категориальных переменных
data.select_dtypes(include='object').describe()

Пример взят из задачи
преднозирования отклика ОТП Банка

	AGREEMENT_RK	SOCSTATUS_WORK_FL	SOCSTATUS_PENS_FL	GENDER	DL_DOCUMENT_FL
количество непропущенных значений	count	15223	15223	15223	15223
количество уникальных значений	unique	15223	2	2	1
мода	top	59910150	1	0	1
частота моды	freq	1	13847	13176	9964
					15223

Обращаем внимание на:

- переменные, у которых количество уникальных значений (категорий) совпадает с количеством наблюдений;
- переменные с одним уникальным значением (константные переменные).

Такие переменные будут бесполезны.

Таблица 4 Методы pandas для вычисления статистик

Python, библиотека pandas	
среднее	data['income'].mean()
мода	data['income'].mode()
медиана	data['income'].median()
квантиль	data['income'].quantile(0.25)
	data['income'].quantile(0.50)
дисперсия	data['income'].var()
стандартное отклонение	data['income'].std()
коэффициент корреляции	data['child_total'].corr(data['age'])

15. Нормальное распределение

15.1. Знакомство с нормальным распределением

Нормальное распределение (или рапределение Гаусса) является наиболее важным и наиболее широко используемым распределением в статистике. Его иногда называют «колоколообразной кривой». Например, ряд статистических тестов предполагает, что данные подчиняются нормальному распределению. Нормальное распределение пригодится нам и в машинном обучении. Для построения линейной регрессии должны быть выполнены предпосылки о том, что остатки – разности между фактическими и спрогнозированными значениями зависимой переменной – подчиняются нормальному распределению (наряду с предпосылками о гомоскедастичности², случайном характере остатков и отсутствии автокорреляции остатков). Для выполнения этих предпосылок мы можем применить к зависимой переменной и/или признакам преобразования, делающие распределение более похожим на нормальное (логарифм, корень и др.)³.

Строго говоря, некорректно говорить о «нормальном распределении», поскольку существует много нормальных распределений. Нормальные распределения могут отличаться своими средними и стандартными отклонениями. На рисунке ниже мы видим четыре нормальных распределения. У зеленого распределения среднее равно 0, а стандартное отклонение 1, у красного распределения среднее равно 0, а стандартное отклонение 0,2, у фиолетового распределения среднее равно –2, а стандартное отклонение 0,5.

Эти нормальные распределения являются симметричными с относительно большими значениями в центре распределения и меньшими значениями в хвостах.

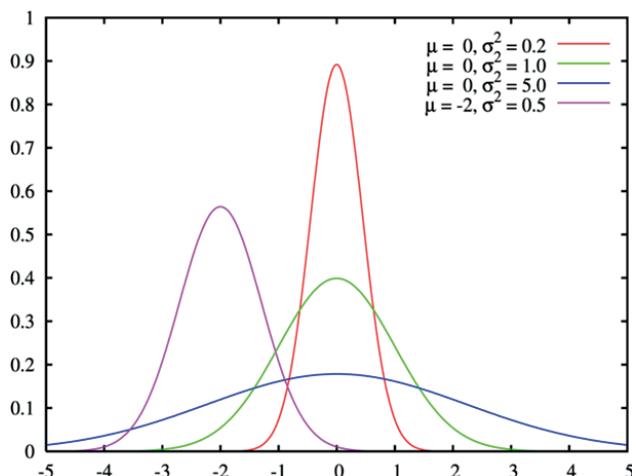


Рис. 21 Четыре нормальных распределения

² Дисперсия остатков одинакова для всех значений признака.

³ Заметим, что мы можем применять эти преобразования и для других задач, например для борьбы с выбросами, ухудшающими качество линейных моделей, и для работы с нелинейностью при использовании линейных моделей.

Плотность нормального распределения (высота для данного значения на оси x) определяется по формуле:

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Нормальное распределение определяется параметрами σ и μ , являющимися стандартным отклонением и средним распределения соответственно. Символ e – это основание натурального логарифма, а π – это число пи.

Изменение μ смещает центр распределения вправо или влево, не влияя на саму форму кривой плотности. Посмотрите на фиолетовую кривую.

Изменение σ определяет разброс значений случайной величины около среднего, остроконечность кривой. Когда данные имеют малый разброс, то вся их масса сконцентрирована у центра, и мы получим остроконечную кривую (посмотрите на красную кривую). Если же у данных большой разброс, то они «размажутся» по широкому диапазону (посмотрите на синюю кривую).

Приведем основные свойства нормальных распределений:

- нормальные распределения симметричны относительно своих средних;
- среднее значение, мода и медиана нормального распределения совпадают;
- площадь под нормальным распределением равна 1;
- нормальные распределения плотнее в центре и менее плотны в хвостах (проще говоря, средние и близкие к средним значения встречаются чаще, чем крайние – экстремально большие и экстремально малые);
- нормальные распределения определяются двумя параметрами: средним (μ) и стандартным отклонением (σ);
- примерно 95 % площади нормального распределения лежат в пределах 2 стандартных отклонений от среднего.

Представьте, мы играем в азартную игру. Допустим, мы формулируем вопрос «если подбросить честную монету 100 раз, какова вероятность выпадения 60 и более решек?». Вероятность выпадения ровно x решек за N подбрасываний рассчитывается по формуле:

$$P(x) = \frac{N!}{x!(N-x)!} \pi^x (1-\pi)^{N-x},$$

где x – количество решек (60), N – количество подбрасываний монеты (100), π – вероятность выпадения решки (0,5). Таким образом, чтобы решить эту проблему, вам нужно вычислить вероятность выпадения 60 решек, затем вероятность выпадения 61 решки, 62 и т. д. и сложить эти вероятности. Представьте, сколько времени потребовалось бы для вычисления биномиальных вероятностей до появления калькуляторов и компьютеров.

Абрахам де Муавр, статистик XVIII века и консультант азартных игроков, часто привлекался к проведению этих длительных вычислений. Де Муавр заметил, что когда число событий (подбрасываний монет) увеличивается, форма биномиального распределения приближается к очень плавной кривой. Биномиальные распределения для 2, 4 и 12 подбрасываний показаны на рисунке ниже.

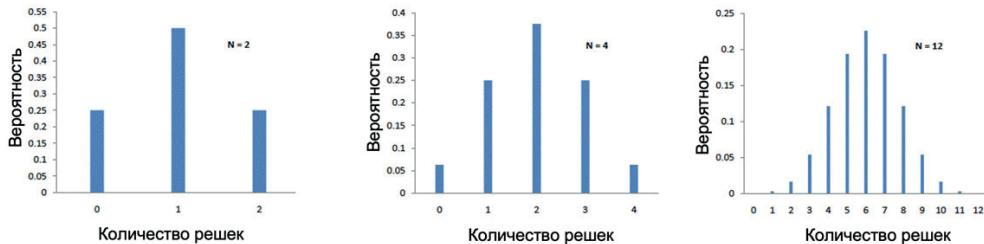


Рис. 22 Примеры биномиальных распределений (высота синего столбика – вероятность)

Де Муавр рассуждал так: если бы я мог найти математическое выражение для этой кривой, я мог бы гораздо легче решать такие задачи, как нахождение вероятности выпадения 60 и более решек из 100 подбрасываний монеты. В точности это он и сделал, и кривая, которую он открыл, теперь называется «нормальной кривой».

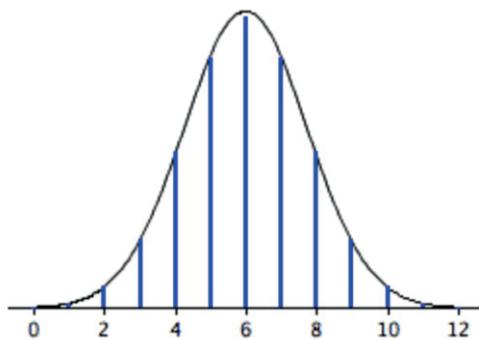


Рис. 23 Апроксимация биномиального распределения для 12 бросков монет нормальным распределением (гладкая кривая – это нормальное распределение, обратите внимание, насколько хорошо она аппроксимирует биномиальные вероятности, представленные высотой синих столбиков)

Важность нормальной кривой обусловлена тем, что распределения многих природных явлений, по крайней мере, являются приблизительно нормальными. Одно из первых применений нормального распределения – анализ ошибок измерений в ходе астрономических наблюдений, ошибок, произошедших из-за несовершенства инструментов и наблюдателей. Галилей в XVII веке отметил, что эти ошибки были симметричными, при этом небольшие ошибки возникали чаще, чем большие. Это привело к нескольким гипотезам о распределении ошибок, но только в начале XIX века было установлено, что эти ошибки соответствуют нормальному распределению.

Независимо друг от друга математики Адрейн в 1808 г. и Гаусс в 1809 г. разработали формулу для нормального распределения и показали, что ошибки хорошо соответствуют этому распределению.

Это же распределение было обнаружено Лапласом в 1778 г., когда он вывел чрезвычайно важную центральную предельную теорему. Лаплас показал, что даже если распределение не является нормальным, средние значения повторно извлеченных выборок из распределения будут распределены почти нормально.

мально, и чем больше будет размер извлекаемой выборки, тем ближе к нормальному будет распределение средних.

А центральная предельная теорема Ляпунова 1900 г. объясняет широкое распространение нормального закона распределения и поясняет механизм его образования. Она звучит так: если случайная величина X является суммой очень большого числа взаимно независимых случайных величин $X_1, X_2, X_3, \dots, X_n$, влияние каждой из которых на всю сумму ничтожно мало, то X имеет распределение, близкое к нормальному.

Возьмем физику. Количество молекул воздуха очень велико, и каждая из них своим движением оказывает ничтожно малое влияние на всю совокупность. Поэтому скорость молекул воздуха распределена нормально. Теперь возьмем пример из биологии. Допустим, у нас есть большая популяция некоторых особей. Каждая из них (или подавляющее большинство) оказывает несущественное влияние на жизнь всей популяции, следовательно, продолжительность жизни этих особей тоже распределена по нормальному закону. Рост взрослых мужчин или взрослых женщин по этой же причине распределен поциальному закону.

Как обсуждалось ранее, у нормальных распределений вовсе не обязательно должны быть одинаковые средние и стандартные отклонения. Нормальное распределение со средним значением 0 и стандартным отклонением 1 называется *стандартным нормальнym распределением*.

Значение из любого нормального распределения может быть преобразовано в соответствующее значение в стандартном нормальном распределении при помощи следующей формулы:

$$Z = (X - \mu) / \sigma,$$

где Z – это значение стандартного нормального распределения, X – значение исходного распределения, μ – среднее значение исходного распределения, а σ – стандартное отклонение исходного распределения. По сути, мы выполняем уже знакомую нам процедуру стандартизации, отсюда и название распределения.

15.2. Коэффициент остроты вершинности, коэффициент эксцесса и коэффициент асимметрии

На практике для анализа распределения переменной и подбора преобразований, максимизирующих нормальность, пользуются коэффициентом остроты вершинности, или куртосисом (kurtosis), коэффициентом эксцесса (excess kurtosis), коэффициентом асимметрии (skewness), строят гистограмму распределения, графики квантиль–квантиль и ящиковую диаграмму.

Коэффициент остроты вершинности, или куртосис (kurtosis), – это мера остроты пика в распределении случайной величины. Он характеризует распределение, в котором значения величины либо сосредоточены близко к среднему значению, либо, наоборот, распределены далеко от него. Коэффициент остроты вершинности вычисляется по формуле:

$$\text{Kurtosis} = \frac{\sum_{i=1}^N (Y_i - \bar{Y})^4 / N}{s^4}.$$

В приведенной формуле Y_i – фактическое значение зависимой переменной для наблюдения, \bar{Y} – среднее значение зависимой переменной, N – размер выборки, s – стандартное отклонение. Логика показателя следующая: любые стандартизованные значения, которые меньше 1 (т. е. данные, лежащие в пределах одного стандартного отклонения от среднего значения, где будет находиться «пик»), практически не вносят вклад в куртосис, поскольку возведение числа меньше 1 в четвертую степень приблизит его к нулю (в большей мере, чем возведение во вторую и третью степени). Наоборот, значения, равные 2 или 3 стандартным отклонениям, станут очень большими. Значение 2 превратится в 16, значение 3 превратится в 81. Поэтому единственными значениями данных, которые вносят вклад в куртосис, являются значения вне области пика, т. е. выбросы.

Для стандартного нормального распределения коэффициент острокурткости равен 3. На практике используют эту же формулу, но с вычитанием 3, чтобы коэффициент был равен 0.

$$\text{Excess kurtosis} = \frac{\sum_{i=1}^N (Y_i - \bar{Y})^4 / N}{s^4} - 3.$$

И такой коэффициент называют уже коэффициентом эксцесса (excess kurtosis). Прилагательное excess здесь буквально означает «выходящий за рамки нормы».

Это дает удобство интерпретации. Если коэффициент эксцесса положителен (коэффициент острокурткости > 3), то распределение будет иметь острую вершину, мы имеем дело с острокуртким (лептокуртическим) распределением. Если коэффициент эксцесса отрицателен (коэффициент острокурткости < 3), то распределение будет иметь пологую вершину, мы имеем дело с плосковершинным (платокуртическим) распределением. Если коэффициент эксцесса равен нулю (коэффициент острокурткости равен 3), наше распределение является нормальным (мезокуртическим).

Острокурткные (лептокуртические) распределения имеют более тяжелые хвосты. Примерами таких распределений являются t -распределение Стьюдента, распределение Рэлея, распределение Лапласа, экспоненциальное распределение, распределение Пуассона и логистическое распределение. Иногда такие распределения называют супергауссовыми (пик этих распределений находится над пиком нормального распределения).



Рис. 24 Виды распределений с точки зрения эксцесса

Плосковершинные (платокуртические) распределения имеют более тонкие хвосты. Примерами таких распределений являются непрерывное и дискретное

равномерное распределение, распределение приподнятого косинуса. Самым плосковершинным распределением из всех является распределение Бернули с $p = 1/2$ (например, количество выпадений «решки» при подбрасывании монеты), для которого эксцесс равен -2 . Иногда такие распределения называют субгауссовыми (пик этих распределений находится под пиком нормального распределения).

Супергауссово, гауссово и субгауссово распределения

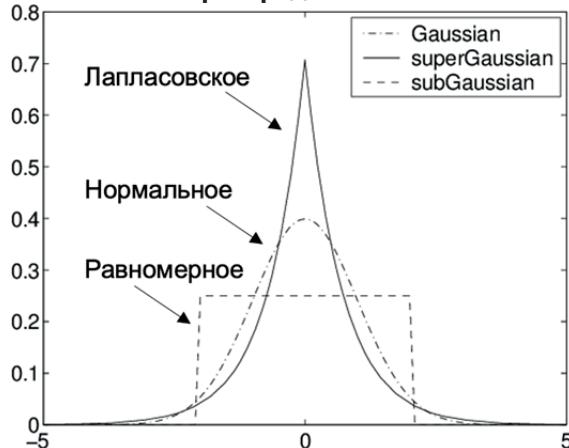


Рис. 25 Виды распределений по отношению к нормальному распределению

Ниже приведен график распределений с разным значением эксцесса.

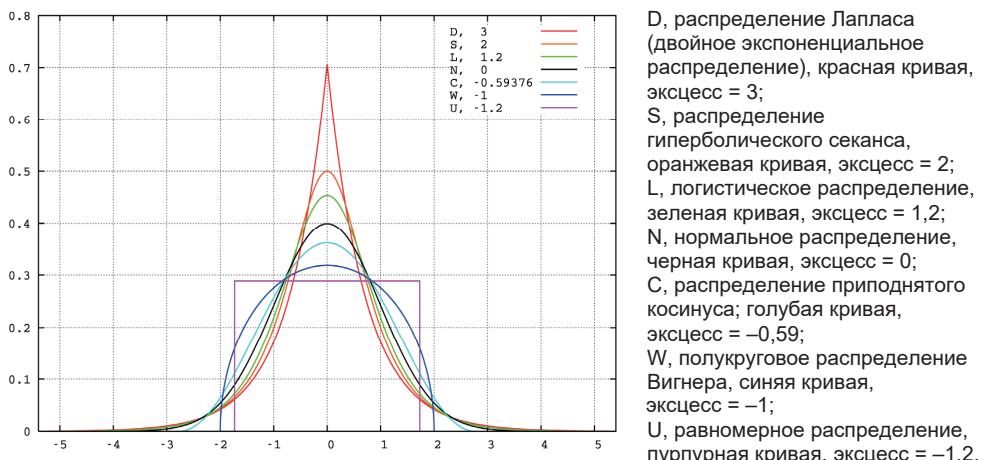


Рис. 26 Виды распределений с разными значениями эксцесса

Коэффициент асимметрии, или скоса (skewness), – мера асимметрии распределения случайной величины. Он вычисляется по формуле:

$$Skewness = \frac{\sum_{i=1}^N (Y_i - \bar{Y})^3 / N}{S^3}.$$

Использование нечетной, третьей степени позволяет сохранить знак, а значит, информацию о том, где расположен выброс – слева или справа. Значения меньше среднего будут иметь отрицательное отклонение. Большие отрицательные отклонения станут очень большими отрицательными значениями при возведении в куб. Если у вас будет преобладание чисел меньше среднего, вы получите отрицательный коэффициент асимметрии. Если у вас будет преобладание чисел больше среднего, вы получите положительный коэффициент асимметрии. Если у вас симметрия, числа взаимно скомпенсируют друг друга.

Для стандартного нормального распределения коэффициент асимметрии равен нулю. Если коэффициент асимметрии положителен, то распределение будет скосено вправо, если отрицателен, распределение будет скосено влево. При левосторонней асимметрии мода больше медианы, а медиана больше, чем среднее значение. При правосторонней асимметрии среднее значение больше медианы, а медиана больше моды. Для нормального распределения мода, медиана и среднее значение одинаковы.

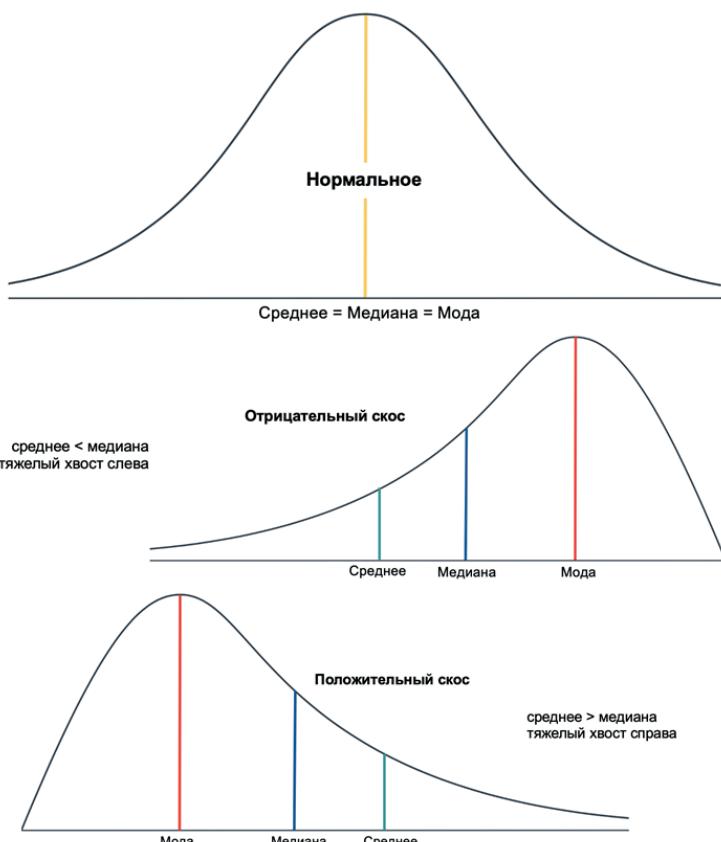


Рис. 27 Среднее, медиана и мода для нормального распределения, распределения, скосенного влево, распределения, скосенного вправо

В бизнесе вы часто встретите правостороннюю асимметрию в наборах данных, которые отражают величины, выраженные положительными числами (например, объемы продаж или размеры активов). Причина возникновения такой асимметрии заключается в том, что такие данные не могут быть меньше нуля (наличие границы с одной стороны), но не ограничены конкретной верхней границей. В результате много значений сконцентрировано вблизи нуля, и количество значений становится все меньше и меньше при движении по горизонтальной оси гистограммы вправо.

15.3. ГИСТОГРАММА РАСПРЕДЕЛЕНИЯ И ГРАФИК КВАНТИЛЬ–КВАНТИЛЬ

Чтобы построить гистограмму распределения, просто подсчитывают, сколько раз значение случайной величины попало в каждый интервал. Для перехода к вероятностям достаточно разделить количество значений в каждом интервале на общее число наблюдений. В этом случае сумма всех столбцов гистограммы будет равна 1 (как и площадь под кривой закона распределения).

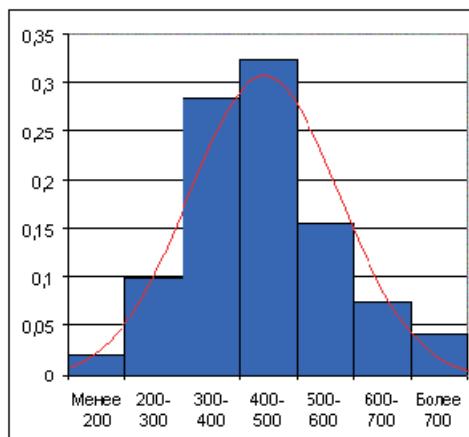


Рис. 28 Гистограмма распределения

График квантиль–квантиль (график Q–Q) представляет собой графический метод для сравнения двух распределений путем построения их квантилей друг относительно друга.

Мы визуализируем связь между квантилями исходного, наблюдаемого распределения и квантилями теоретического распределения (по умолчанию используется нормальное распределение). Если наблюдаемые значения попадают на диагональную линию, то теоретическое распределение хорошо подходит к наблюдаемым данным.

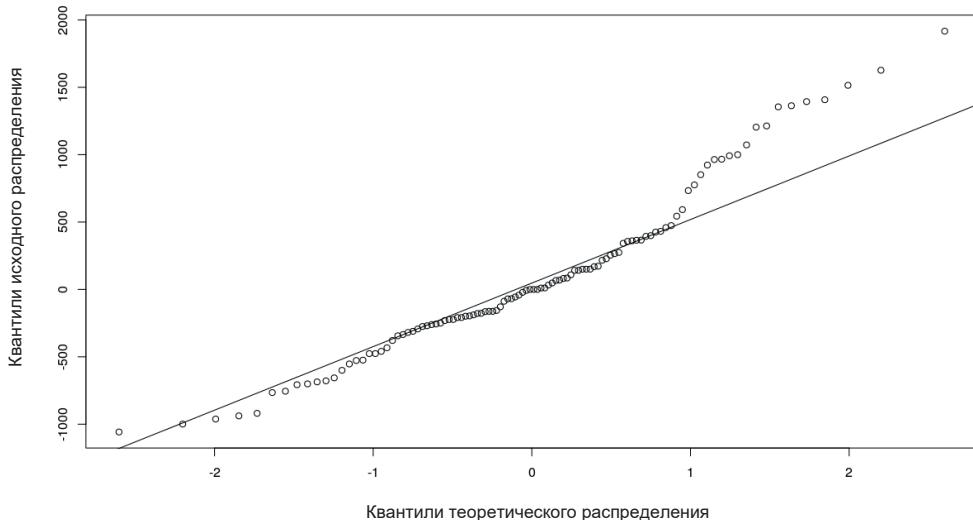


Рис. 29 График квантиль–квантиль

Если не все наблюдаемые значения попадают на диагональную линию, просят выполнить такое преобразование переменной, которое позволит приблизить распределение к нормальному.

15.4. ВЫЧИСЛЕНИЕ КОЭФФИЦИЕНТА АСИММЕТРИИ И КОЭФФИЦИЕНТА ЭКСЦЕССА, ПОСТРОЕНИЕ ГИСТОГРАММЫ И ГРАФИКА КВАНТИЛЬ–КВАНТИЛЬ ДЛЯ ПОДБОРА ПРЕОБРАЗОВАНИЙ, МАКСИМИЗИРУЮЩИХ НОРМАЛЬНОСТЬ

Для построения гистограммы и графика квантиль–квантиль в Python нам понадобятся библиотеки SciPy и seaborn. Для иллюстрации воспользуемся данными, записанными в файле *Normality.csv*.

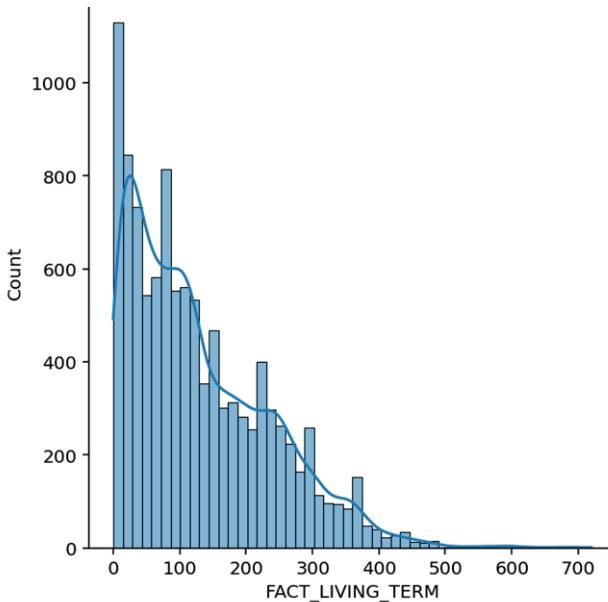
```
# импортируем необходимые библиотеки
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
import seaborn as sns
from scipy.stats import norm
from scipy import stats
from sklearn.preprocessing import PowerTransformer

# загружаем набор данных
train = pd.read_csv('Data/Normality.csv', sep=';')
# выводим первые пять наблюдений
train.head()
```

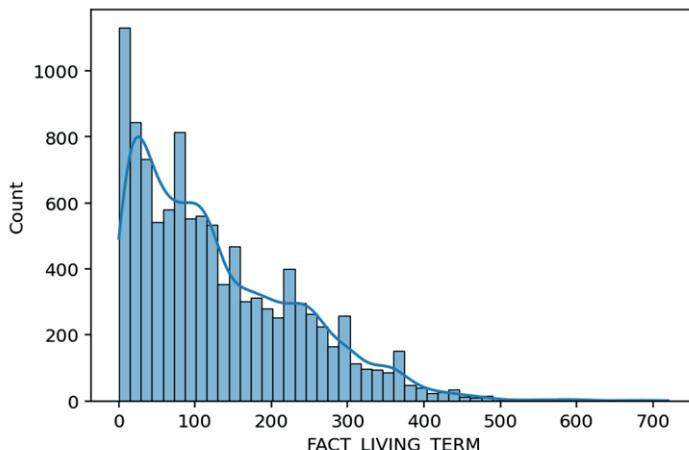
	AGE	PERSONAL_INCOME	CREDIT	FST_PAYMENT	FACT_LIVING_TERM	LOAN_AVG_DLQ_AMT
0	41	20000.0	65000.0	10000.0	204.0	0.0
1	53	13000.0	16460.0	10000.0	252.0	0.0
2	23	9500.0	25000.0	12000.0	96.0	0.0
3	31	7000.0	2500.0	790.0	7.0	0.0
4	49	15000.0	9300.0	1112.0	22.0	0.0

Давайте построим гистограмму распределения и график квантиль–квантиль для переменной *FACT_LIVING_TERM*. Для построения гистограммы можно воспользоваться функциями библиотеки seaborn `displot()` и `histplot()`.

```
# строим гистограмму распределения
# для переменной FACT_LIVING_TERM
sns.displot(data=train, x='FACT_LIVING_TERM', kde=True);
```

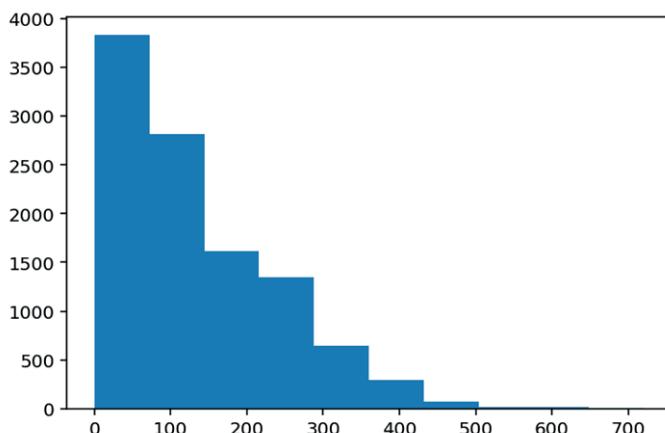


```
# строим гистограмму распределения
# для переменной FACT_LIVING_TERM
sns.histplot(data=train, x='FACT_LIVING_TERM', kde=True);
```



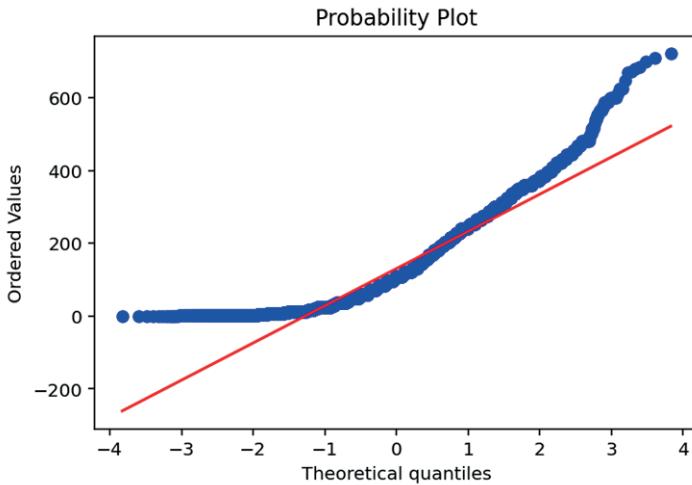
Разумеется, всегда можно воспользоваться функцией `plt.hist()`.

```
# строим гистограмму распределения
# для переменной FACT_LIVING_TERM
plt.hist(train['FACT_LIVING_TERM']);
```



Для построения графика квантиль–квантиль обратимся к функции `probplot()` модуля `stats` библиотеки SciPy.

```
# строим график квантиль–квантиль
# для переменной FACT_LIVING_TERM
fig = plt.figure()
res = stats.probplot(train['FACT_LIVING_TERM'], plot=plt)
```



Для вычисления скоса и эксцесса нам потребуются методы `.skew()` и `.kurtosis()` объекта Series библиотеки pandas.

```
# вычисляем скос и эксцесс
print("Скос", train['FACT_LIVING_TERM'].skew())
print("Эксцесс", train['FACT_LIVING_TERM'].kurtosis())
```

```
Скос 1.0181444392860792
Эксцесс 0.9035560079843634
```

Можно написать собственные функции, вычисляющие скос и эксцесс.

```
# пишем функцию, вычисляющую скос
def skewness(var, df):
    num = np.mean(np.power(df[var] - df[var].mean(), 3))
    dem = np.power(df[var].std(), 3)
    res = num / dem
    return res

# применяем функцию для вычисления скоса
skewness('FACT_LIVING_TERM', train)

1.0178578174556934

# пишем функцию, вычисляющую эксцесс
def excess_kurtosis(var, df):
    num = np.mean(np.power(df[var] - df[var].mean(), 4))
    dem = np.power(df[var].std(), 4)
    res = (num / dem) - 3
    return res

# применяем функцию для вычисления эксцесса
excess_kurtosis('FACT_LIVING_TERM', train)
```

```
0.9018366578097732
```

Здесь мы видим умеренную правостороннюю асимметрию и слабовыраженный эксцесс.

15.5. ПОДБОР ПРЕОБРАЗОВАНИЙ, МАКСИМИЗИРУЮЩИХ НОРМАЛЬНОСТЬ ДЛЯ ПРАВОСТОРОННЕЙ АСИММЕТРИИ

На рисунке внизу представлены преобразования различной строгости для устранения правосторонней асимметрии. Под строгостью подразумевается та сила, с которой мы преобразовываем наши значения, т. е. смещаем их влево. Например, возьмем исходное значение 16 и применим преобразование квадратного корня, $\sqrt{16} = 4$. Теперь применим обратное преобразование, $1/16 = 0,0625$. Видим, что обратное преобразование является более строгим, сильнее смещает значение влево и применяется в случаях очень сильной правосторонней асимметрии.

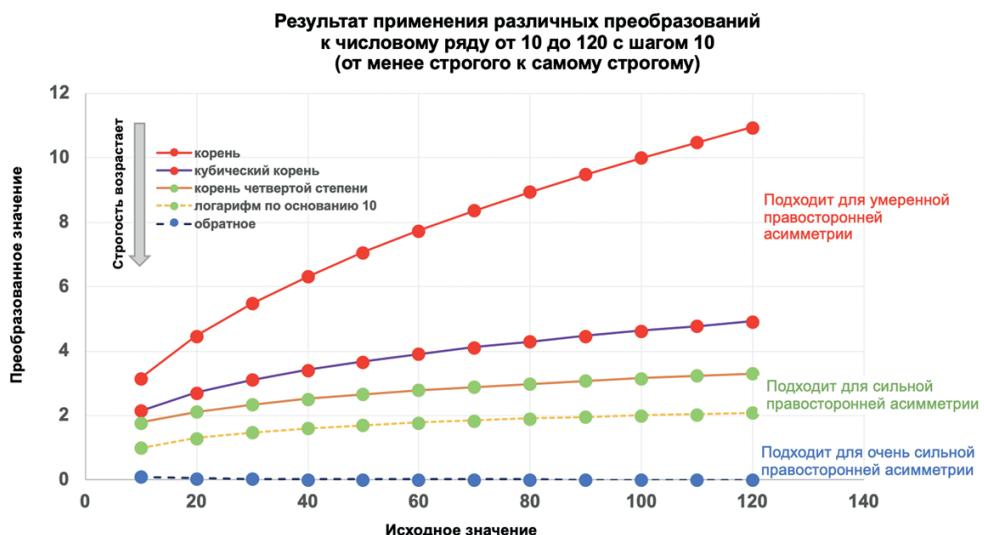


Рис. 30 Результаты применения разных распределений

Часто пишут функцию, которая автоматически вычисляет исходный скос и скос после выполненных преобразований, и по результатам выбирают наиболее эффективное преобразование.

```
# пишем функцию, которая вычисляет исходный скос и
# скос после выполненных преобразований
def diagnostics_skewness(df):
    # создаем списки
    col_list = df.select_dtypes(include=['number']).columns
    skew_initial_list = []
    skew_pos_reciprocal_list = []
    skew_neg_reciprocal_list = []
    skew_log_list = []
    skew_corr_log_001_list = []
    skew_corr_log_01_list = []
    skew_corr_log_1_list = []
    skew_corr_log_5_list = []
    skew_cbrt_list = []
    skew_sqrt_list = []
```

```

# создаем копию датафрейма
df_ = df.copy()
# запускаем цикл, который вычисляет скос для
# исходной переменной и после преобразования
for i in col_list:
    df_[i] = df_[i].fillna(df_[i].median())
    skew_initial = df_[i].skew()
    skew_pos_reciprocal = (1 / (df_[i].clip(0.01))).skew()
    skew_neg_reciprocal = (-1 / (df_[i].clip(0.01))).skew()
    skew_log = np.log(df[i].clip(0.01)).skew()
    skew_corr_log_001 = np.log((df_[i].clip(0.01) /
                                df_[i].mean()) + 0.001).skew()
    skew_corr_log_01 = np.log((df_[i].clip(0.01) /
                               df_[i].mean()) + 0.01).skew()
    skew_corr_log_1 = np.log((df_[i].clip(0.01) /
                              df_[i].mean()) + 0.1).skew()
    skew_corr_log_5 = np.log((df_[i].clip(0.01) /
                              df_[i].mean()) + 0.5).skew()
    skew_cbrt = (np.sign(df_[i]) * np.cbrt(df_[i].abs())).skew()
    skew_sqrt = (np.sign(df_[i]) * np.sqrt(df_[i].abs())).skew()
    skew_initial_list.append(skew_initial)
    skew_pos_reciprocal_list.append(skew_pos_reciprocal)
    skew_neg_reciprocal_list.append(skew_neg_reciprocal)
    skew_log_list.append(skew_log)
    skew_corr_log_001_list.append(skew_corr_log_001)
    skew_corr_log_01_list.append(skew_corr_log_01)
    skew_corr_log_1_list.append(skew_corr_log_1)
    skew_corr_log_5_list.append(skew_corr_log_5)
    skew_cbrt_list.append(skew_cbrt)
    skew_sqrt_list.append(skew_sqrt)

# формируем таблицу с результатами
result = pd.DataFrame({'Переменная': col_list,
                       'Skew_init': skew_initial_list,
                       'Skew_pos_recip': skew_pos_reciprocal_list,
                       'Skew_neg_recip': skew_neg_reciprocal_list,
                       'Skew_log': skew_log_list,
                       'Skew_adj_log (k=0.001)': skew_corr_log_001_list,
                       'Skew_adj_log (k=0.01)': skew_corr_log_01_list,
                       'Skew_adj_log (k=0.1)': skew_corr_log_1_list,
                       'Skew_adj_log (k=0.5)': skew_corr_log_5_list,
                       'Skew_cbrt': skew_cbrt_list,
                       'Skew_sqrt': skew_sqrt_list})
result = result.sort_values(by='Skew_init', ascending=False)
result = np.round(result, 3)
cm = sns.light_palette('magenta', as_cmap=True)
return result.style.background_gradient(cmap=cm)

```

Давайте применим нашу функцию к загруженным данным.

```

# применяем нашу функцию
diagnostics_skewness(train)

```

Переменная	Skew_init	Skew_pos_recip	Skew_neg_recip	Skew_log	Skew_adj_log (k=0.001)	Skew_adj_log (k=0.01)	Skew_adj_log (k=0.1)	Skew_adj_log (k=0.5)	Skew_cbrt	Skew_sqrt
FST_PAYMENT	6.744000	3.195000	-3.195000	-2.788000	-2.021000	-1.195000	0.046000	1.014000	0.186000	1.600000
LOAN_AVG_DLQ_AMT	5.856000	-2.270000	2.270000	2.288000	2.302000	2.325000	2.388000	2.511000	2.519000	2.857000
PERSONAL_INCOME	5.024000	101.043000	-101.043000	0.130000	0.164000	0.253000	0.437000	0.835000	0.941000	1.428000
CREDIT	2.836000	1.518000	-1.518000	0.067000	0.070000	0.091000	0.265000	0.701000	0.737000	1.152000
FACT_LIVING_TERM	1.018000	29.702000	-29.702000	-1.446000	-1.207000	-0.954000	-0.433000	0.063000	-0.223000	0.135000
AGE	0.247000	0.543000	-0.543000	-0.142000	-0.141000	-0.137000	-0.103000	-0.005000	-0.010000	0.055000

15.5.1. Обратное преобразование, отрицательное обратное преобразование

$$\frac{1}{x}, -\frac{1}{x}$$

Очень строгое преобразование, которое применяется для распределения, очень сильно скошенного вправо. В ситуации распределения, очень сильно скошенного вправо, это преобразование может сработать лучше, чем логарифм. Используют обычно при работе с переменными – результатами деления одной переменной на другую (т. е. разные коэффициенты, отношения). Его нельзя применить к нулевым значениям, поэтому если есть нулевые значения, нужно добавить константу. Применение данного преобразования к отрицательным значениям не дает содержательной интерпретации, поэтому оно обычно применяется к положительным значениям. Обратное преобразование отношений легко интерпретируется: плотность населения, выраженная как количество человек на единицу площади, станет размером площади на человека; количество пациентов на одного врача станет количеством врачей на одного пациента.

На практике результаты обратного преобразования умножают или делят на определенную константу, например на 1000 или 10 000, для удобства интерпретации, но сама по себе эта операция не влияет на асимметрию или линейность.

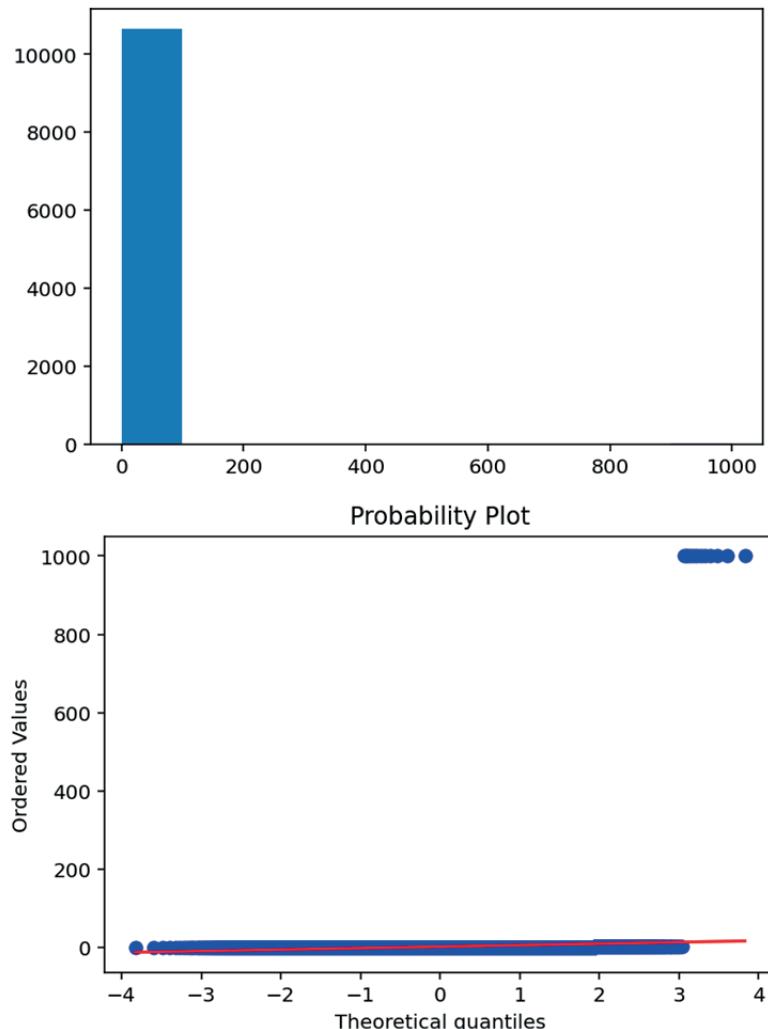
Положительное обратное преобразование меняет порядок значений, имеющих один и тот же знак, на обратный: наибольшее значение становится наименьшим и т. д. Также применяют отрицательное обратное преобразование. Отрицательное обратное преобразование сохраняет порядок значений, имеющих один и тот же знак.

Давайте вычислим скос и эксцесс, гистограмму распределения, график квантиль–квантиль, применив обратное преобразование для переменной *FACT_LIVING_TERM*. При этом используем метод *.clip()*, чтобы не брать отрицательные числа и ноль.

```
# вычисляем скос и эксцесс, гистограмму распределения
# и график квантиль–квантиль, применив обратное
# преобразование для переменной FACT_LIVING_TERM,
# используем .clip(), чтобы не брать
# отрицательные числа и ноль
var = np.reciprocal(train['FACT_LIVING_TERM'].clip(0.001))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

```
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос 29.75263889372499
Эксцесс 883.3956108502981



Мы получили очень маленькие значения, применив очень строгое преобразование в случае умеренной правосторонней асимметрии. Нужно попробовать менее строгие преобразования.

15.5.2. Логарифм

$$\log_{10}(x), \log_e(x), \log_2(x)$$

Строгое преобразование, которое применяется для распределения, сильно скосленного вправо. Применяют логарифм по основанию 10, e и 2. Поскольку логарифм нуля, а равно и любого отрицательного числа, не определен, перед использованием логарифмического преобразования ко всем значениям нужно добавить константу, чтобы сделать их положительными. Например, можно добавить 0,1 и получить $\log(x + 0,1)$.

Еще одна тонкость связана с тем, что в новых данных у нас могут появиться более высокие отрицательные значения, чем в обучающей выборке, и константа сможет сделать положительными только часть значений.

С помощью метода `.clip()` можно приравнять все значения к нижнему или верхнему пороговому значению (параметры `lower` и `upper` соответствен-но). Например, у нас есть переменная `score` со значениями 9, -7, 0, -1, 5. Задав `df['score'].clip(-4, 6)`, мы получаем переменную `score` со значениями 6, -4, 0, -1, 5. Перед логарифмированием мы просто приравниваем нижнее пороговое значение к очень маленькому положительному значению.

Обратите внимание, что выбор основания логарифма также важен. Более высокие основания сжимают значения сильнее.

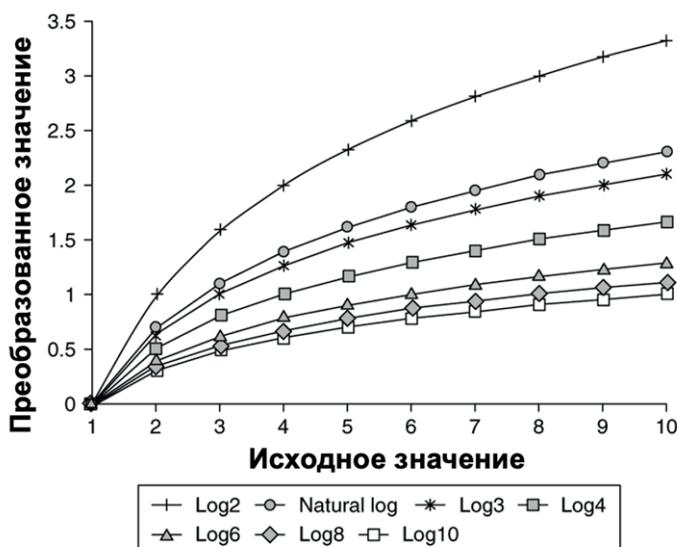


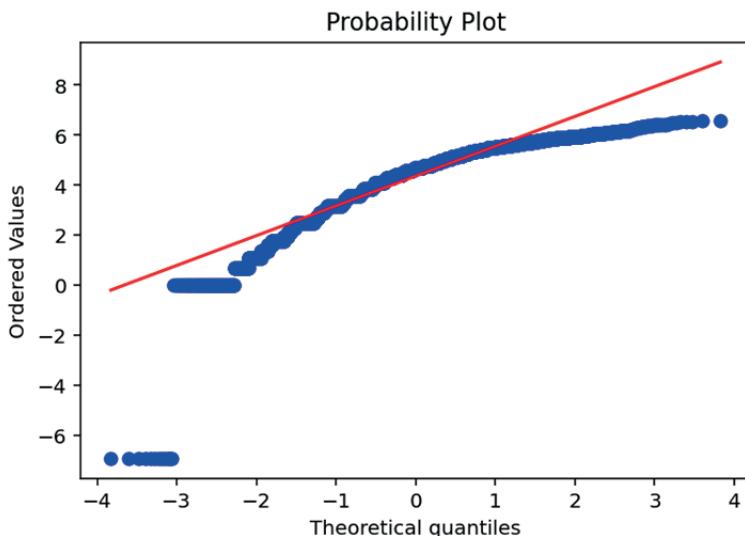
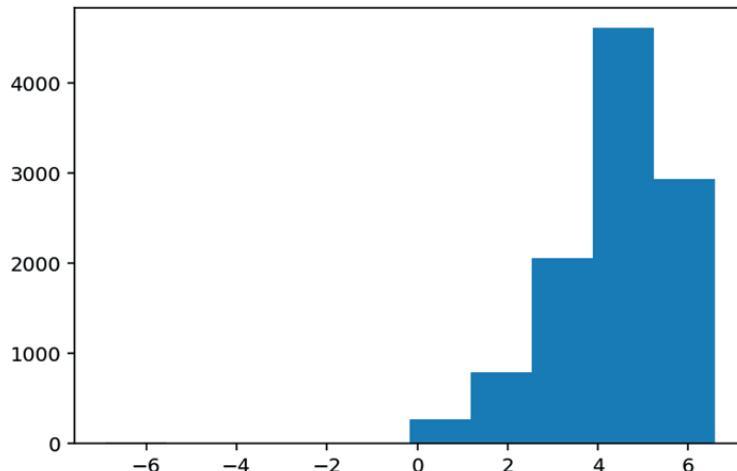
Рис. 31 Преобразования с помощью логарифмов разных оснований

Давайте вычислим скос и эксцесс, гистограмму распределения, график квантиль–квантиль, применив уже логарифмическое преобразование для переменной `FACT_LIVING_TERM`. При этом вновь используем метод `.clip()`, чтобы не брать отрицательные числа и ноль.

```
# вычисляем скос и эксцесс, гистограмму распределения
# и график квантиль-квантиль, применив логарифмическое
# преобразование для переменной FACT_LIVING_TERM,
# используем .clip(), чтобы не брать
# отрицательные числа и ноль
var = np.log(train['FACT_LIVING_TERM'].clip(0.001))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос -1.7641384728316158

Эксцесс 7.61034596892353



Теперь мы получили левостороннюю асимметрию, что тоже не является желаемым результатом.

15.5.3. Логарифм с нормированием на среднее

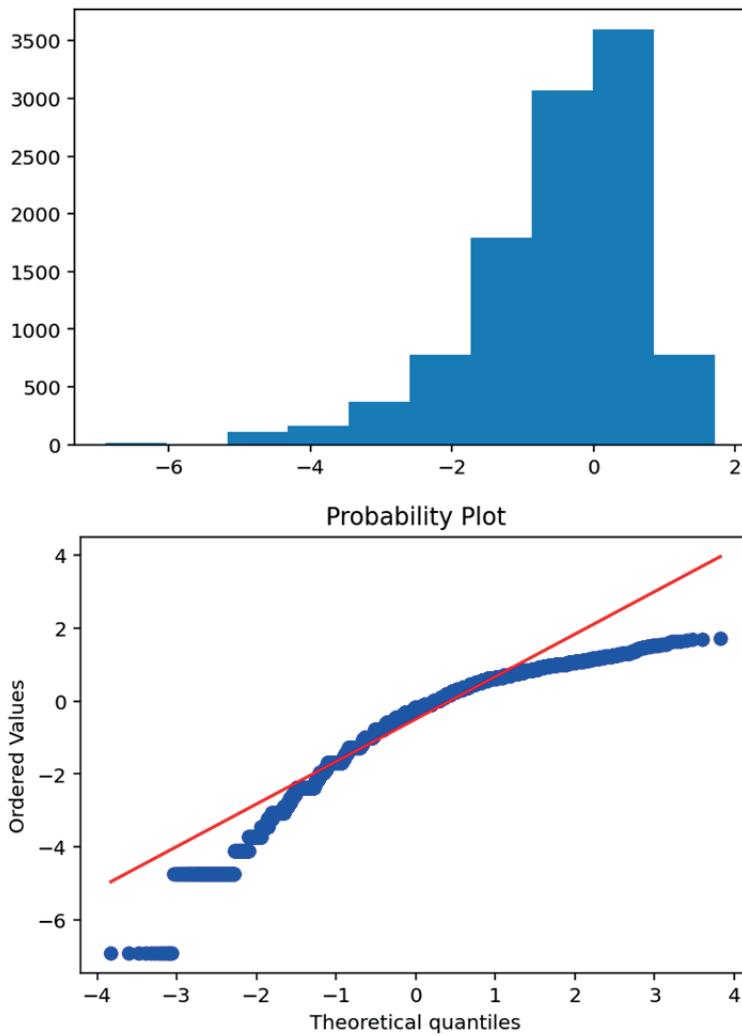
$$\log\left(\frac{x}{\text{mean}(x)} + k\right)$$

В данном преобразовании используется формула $\log\left(\frac{x}{\text{mean}(x)} + k\right)$, где k – маленькое значение (от 0 до 1), его называют корректором асимметрии. В этом преобразовании k будет работать как фактор, определяющий форму распределения. Значения k , близкие к нулю, делают данные более скошенными влево, а значения k , близкие к единице, делают данные менее скошенными влево.

Давайте применим логарифмирование с нормированием на среднее со значением k , близким к 0.

```
# вычисляем скос и эксцесс, гистограмму распределения
# и график квантиль-квантиль, применив логарифмическое
# преобразование по формуле log(x / mean(x) + k),
# где k - небольшое значение, близкое к 0,
# чтобы сильнее смещать распределение влево,
# используем .clip(), чтобы не брать
# отрицательные числа и ноль
k = 0.001
var = np.log((train['FACT_LIVING_TERM'].clip(0.001) /
              train['FACT_LIVING_TERM'].mean()) + k)
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос -1.210585215501142
 Эксцесс 1.7521218621895827

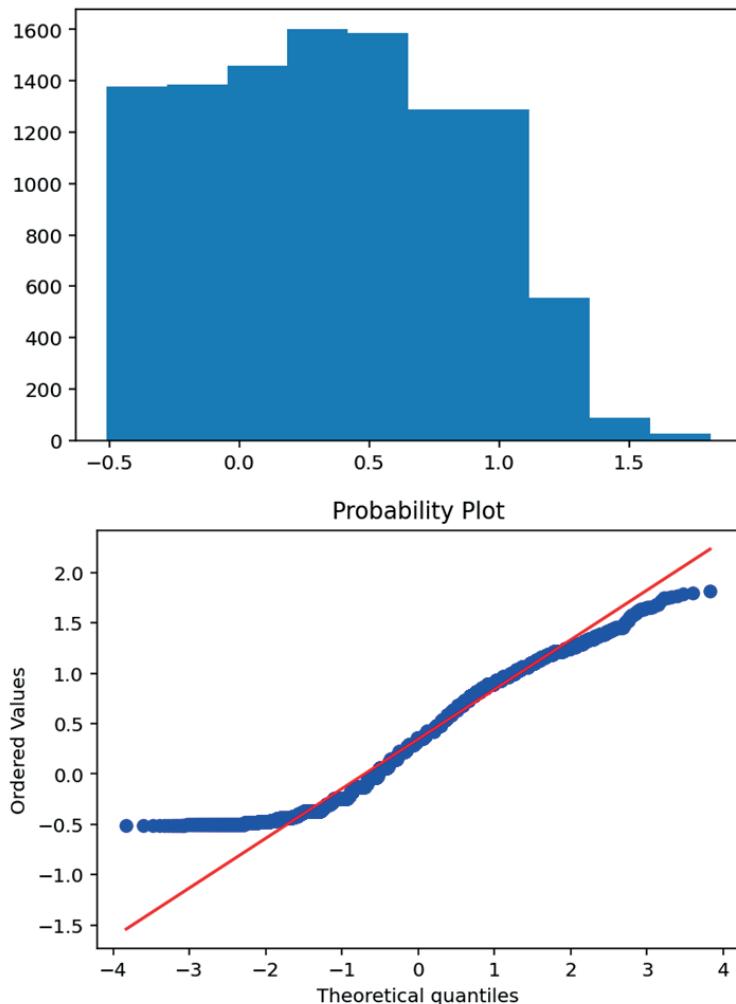


Опять мы получили левостороннюю асимметрию. Давайте применим логарифмирование с нормированием на среднее со значением k , близким к 1.

```
# вычисляем скос и эксцесс, гистограмму распределения
# и график квантиль-квантиль, применив логарифмическое
# преобразование по формуле  $\log(x / \text{mean}(x) + k)$ ,
# где  $k$  - небольшое значение, близкое к 1,
# чтобы сильнее смещать распределение влево,
# используем .clip(), чтобы не брать
# отрицательные числа и ноль
k = 0.6
var = np.log((train['FACT_LIVING_TERM'].clip(0.001) /
              train['FACT_LIVING_TERM'].mean()) + k)
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
```

```
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос 0.1212535167386184
Эксцесс -0.962190130053846



Стало намного лучше. Теперь попробуем преобразования корней.

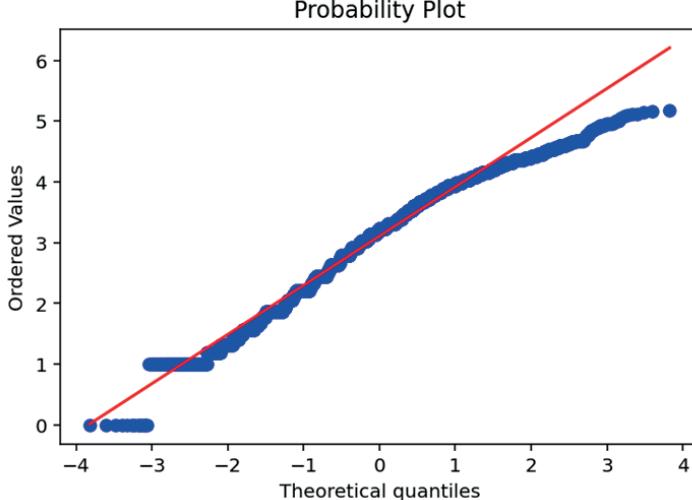
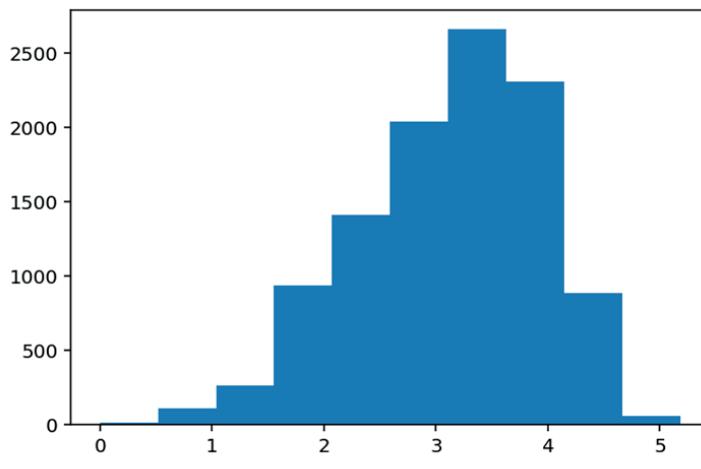
15.5.4. Корень четвертой степени

$$\sqrt[4]{x}$$

Корень четвертой степени является менее строгим преобразованием, чем логарифм. На практике при использовании корней обычно корень берут от модуля числа (чтобы не вычислять корни отрицательных чисел) и затем учитывают знак числа: $\text{sign}(x)^*(\text{abs}(x)^{1/3})$, $\text{sign}(x)^*(\text{abs}(x)^{1/2})$. В целом корни применяются для распределения, умеренно скошенного вправо.

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# корнем четвертой степени, используем модуль,
# чтобы не вычислять корни отрицательных чисел,
# и затем учтываем знак числа
var = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/4))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос -0.4419381477873673
 Эксцесс -0.30500979706137255



Появилась небольшая отрицательная асимметрия. Давайте попробуем более мягкие преобразования корней.

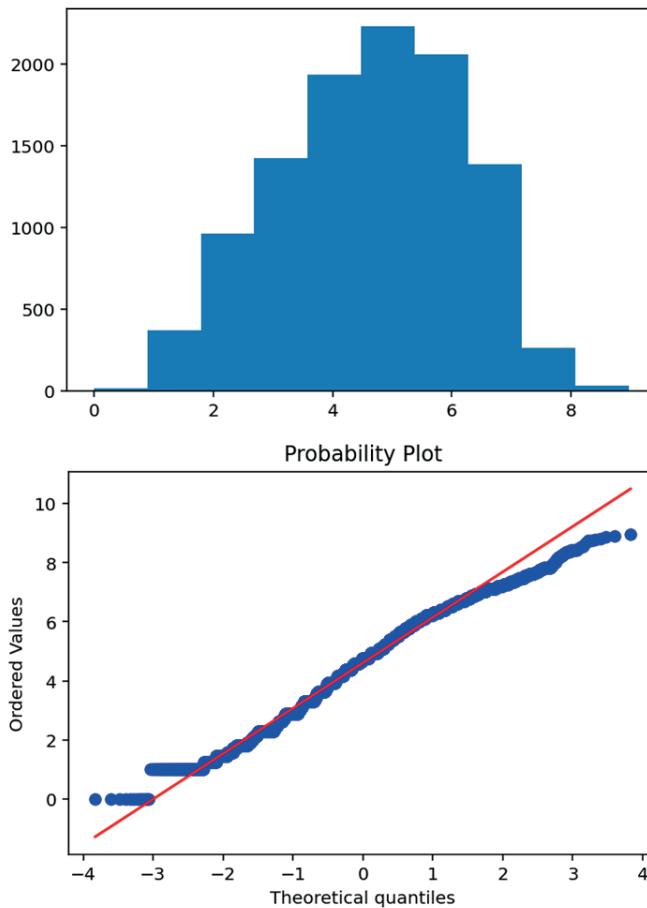
15.5.5. Корень третьей степени

$$\sqrt[3]{x}$$

Кубический корень является менее строгим преобразованием, чем корень четвертой степени.

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# корнем третьей степени, используем модуль,
# чтобы не вычислять корни отрицательных чисел,
# и затем учтываем знак числа
var = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/3))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос -0.222969034559257
 Эксцесс -0.6070016622741847



Это преобразование уже можно брать в работу, однако попробуем еще варианты.

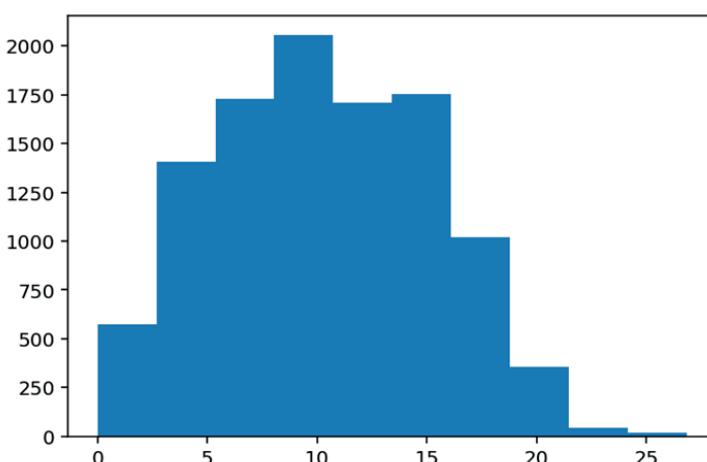
15.5.6. Квадратный корень

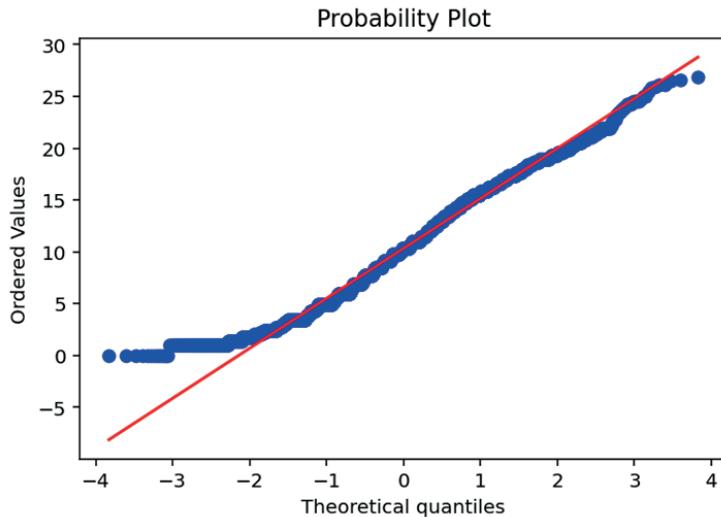
$$\sqrt{x}$$

Квадратный корень – менее строгое преобразование, чем корень четвертой степени и кубический корень. Квадратный корень часто дает хорошее качество для переменных с умеренной правосторонней асимметрией, значения которых представляют собой частоты. Например, речь может идти о распределении частоты редких случайных событий, произошедших за определенный период, которое подчиняется распределению Пуассона. Также квадратный корень дает хорошее качество для переменных с умеренной правосторонней асимметрией, содержащих большое количество очень небольших или нулевых значений.

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# квадратным корнем, используем модуль,
# чтобы не вычислять корни отрицательных чисел,
# и затем учтываем знак числа
var = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/2))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)
```

Скос 0.13476855245733127
 Эксцесс -0.7037178210399051





Тоже хороший результат.

На рисунке ниже приведен еще один пример подбора оптимального преобразования для переменной с правосторонней асимметрией.

Сработали плохо (еще больше увеличили правостороннюю асимметрию), потому что преобразования предназначены для левосторонней асимметрии (сильно смешают вправо). При этом возведение в куб дало наибольшую правостороннюю асимметрию, потому что оно предназначено для более сильно выраженной левосторонней асимметрии, нежели возведение в квадрат

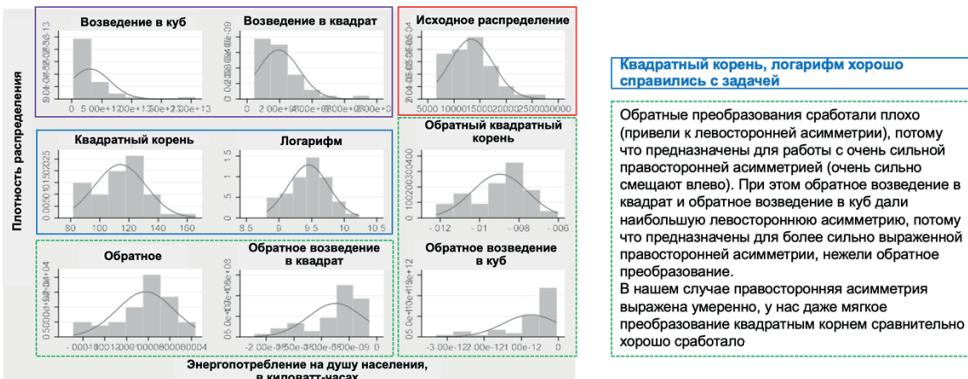


Рис. 32 Пример подбора преобразований для переменной с правосторонней асимметрией

15.6. ПОДБОР ПРЕОБРАЗОВАНИЙ, МАКСИМИЗИРУЮЩИХ НОРМАЛЬНОСТЬ ДЛЯ ЛЕВОСТОРОННЕЙ АСИММЕТРИИ

Теперь рассмотрим преобразования для работы с левосторонней асимметрией.

15.6.1. Экспоненциальное преобразование

$$\exp(x), 2^x$$

Это преобразование может хорошо сработать, если данные содержат логарифмический тренд (отток, выживаемость). Преобразование зависит от диапазона значений и является одним из самых мощных при работе с данными, скошенными влево.

15.6.2. Квадратный корень разности между константой и исходным значением переменной

$$\sqrt{k-x}$$

Преобразование применяется для распределения, умеренно скошенного влево. Константа k подбирается так, чтобы при вычитании из нее исходного значения переменной итоговое наименьшее значение было равно 1. Обычно в качестве константы берут максимальное значение переменной +1.

15.6.3. Логарифм разности между константой и исходным значением переменной

$$\log(k-x)$$

Преобразование применяется для распределения, сильно скошенного влево. Константа k подбирается так, чтобы при вычитании из нее исходного значения переменной итоговое наименьшее значение было равно 1. Обычно в качестве константы берут максимальное значение переменной +1.

15.6.4. Возвведение в степень

$$x^2, x^3$$

Преобразование применяется для распределения, скошенного влево. Подбор степени – нетривиальная задача, часто зависит от характеристики переменной. На практике обычно используют вторую или третью степень. Возведение в куб применяется для работы с распределением, сильно скошенным влево, возведение в квадрат – для работы с распределением, умеренно скошенным влево.

15.7. ПРЕОБРАЗОВАНИЕ БОКСА–КОКСА

В качестве альтернативы перечисленным преобразованиям можно применить преобразование Бокса–Кокса с параметром λ , которое выражается следующим образом:

$$y(\lambda) = \begin{cases} \frac{y^\lambda - 1}{\lambda}, & \text{если } \lambda \neq 0; \\ \log(y), & \text{если } \lambda = 0. \end{cases}$$

Преобразование Бокса–Кокса представляет собой целое семейство преобразований, которое позволяет автоматически находить оптимальную трансфор-

мацию для той или иной переменной. При $\lambda = -1$ выполняется обратное преобразование. При $\lambda = -0,5$ выполняется преобразование обратного квадратного корня. При $\lambda = 0$ выполняется логарифмическое преобразование. При $\lambda = 0,5$ выполняется преобразование квадратного корня. Если $\lambda = 1$, то закон распределения исходной последовательности не изменяется, хотя при этом последовательность получит сдвиг за счет вычитания единицы из каждого ее значения.

Таблица 5 Преобразования в зависимости от значения параметра λ

Значение параметра λ	Преобразование
$\lambda = -1,0$	$y(\lambda) = 1/y$
$\lambda = -0,5$	$y(\lambda) = \frac{1}{\sqrt{y}}$
$\lambda = 0,0$	$y(\lambda) = \ln(y)$
$\lambda = 0,5$	$y(\lambda) = \sqrt{y}$
$\lambda = 2,0$	$y(\lambda) = y^2$

Данные, которые подвергаем преобразованию Бокса–Кокса, должны быть положительными.

На рисунке ниже показано, как выглядят кривые Бокса–Кокса преобразования при различных значениях параметра λ . Горизонтальная шкала представлена в логарифмическом масштабе.

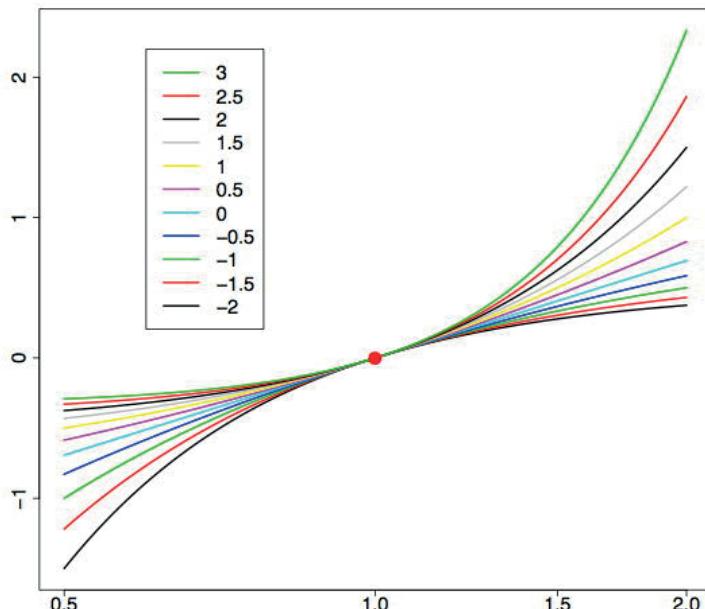


Рис. 33 Преобразование Бокса–Кокса для разных значений параметра λ

Верхняя кривая на рисунке соответствует значению $\lambda = 3$, а нижняя – значению $\lambda = -2$.

Для того чтобы в результате преобразования Бокса–Кокса закон распределения результирующей последовательности был максимально приближен к нормальному закону, необходимо выбрать оптимальное значение параметра λ .

Одним из способов определения оптимальной величины этого параметра является максимизация логарифма функции правдоподобия. Кроме того, оптимальное значение параметра подбирают, найдя максимальное значение коэффициента корреляции между квантилями функции нормального распределения и отсортированной преобразованной последовательностью.

Обратите внимание, мы всегда применяем одинаковое значение параметра λ к обучающему и тестовому наборам. По сути, вычисление значение параметра λ – это мини-модель, которую мы строим на обучающем наборе, и применяем ее к переменным обучающего и тестового наборов. Нельзя отдельно вычислить значение параметра λ на обучающем наборе, отдельно вычислить значение параметра λ на тестовом наборе и затем использовать эти значения для преобразования переменной в соответствующем наборе.

В 2000 году Йео и Джонсон предложили новое семейство преобразований, которое позволяет обойти ограничения преобразования Бокса–Кокса. Оно учитывает отрицательные и нулевые значения.

$$y(\lambda) = \begin{cases} \frac{(y+1)^\lambda - 1}{\lambda}, & \text{если } \lambda \neq 0, y \geq 0; \\ \log(y+1), & \text{если } \lambda = 0, y \geq 0; \\ \frac{(1-y)^{2-\lambda} - 1}{\lambda - 2}, & \text{если } \lambda \neq 2, y < 0; \\ -\log(1-y), & \text{если } \lambda = 2, y < 0. \end{cases}$$

При оценивании параметра преобразования мы находим такое значение λ , которое минимизирует расстояние (дивергенцию) Кульбака–Лейблера между нормальным распределением и преобразованным распределением.

15.7.1. Преобразование Бокса–Кокса с помощью функции `boxcox()` питоновской библиотеки SciPy

Давайте выполним преобразование Бокса–Кокса с помощью функции `boxcox()` питоновской библиотеки SciPy.

```
# импортируем функцию boxcox()
from scipy.stats import boxcox
# выполняем преобразование Бокса–Кокса
var, lam = boxcox(train['FACT_LIVING_TERM'].clip(0.001))
print("Lambda: %f" % lam)

Lambda: 0.373383

# строим гистограмму распределения и график
# квантиль–квантиль, применив преобразование
# Бокса–Кокса
var = pd.Series(var)
```

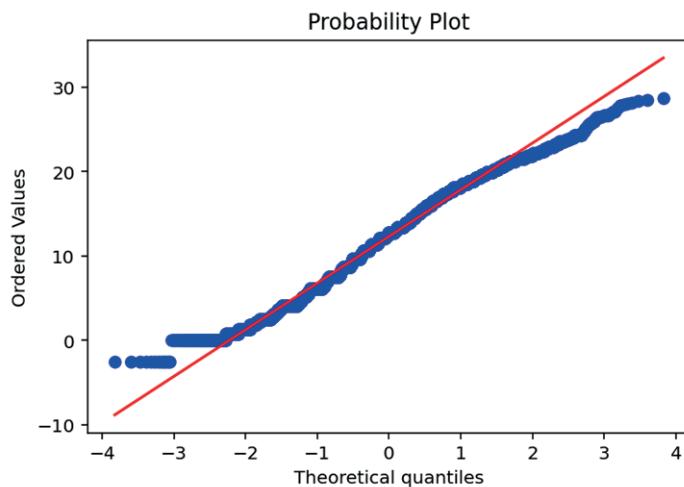
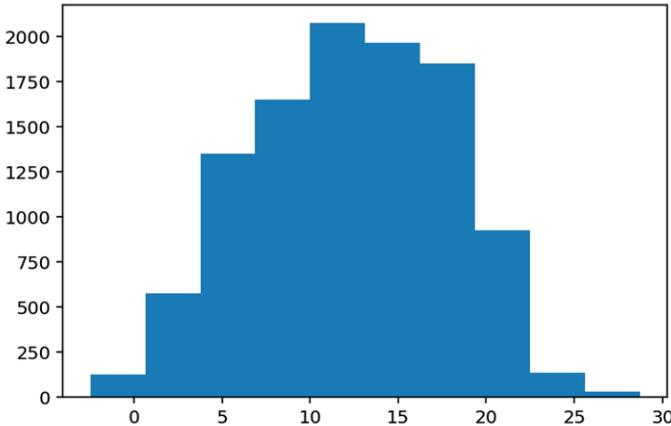
```

print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
plt.hist(var)
fig = plt.figure()
res = stats.probplot(var, plot=plt)

```

Скос -0.12681079843856088

Эксцесс -0.677976445569866



Вполне достойный результат.

Помним, что к переменной в тестовой выборке мы применяем значение `lambda`, вычисленное для этой переменной в обучающей выборке. Программный код выглядел бы так:

```

train['FACT_LIVING_TERM'], fitted_lambda = boxcox(
    train['FACT_LIVING_TERM'])
test['FACT_LIVING_TERM'] = boxcox(
    test['FACT_LIVING_TERM'], fitted_lambda)

```

15.7.2. Преобразование Бокса–Кокса с помощью класса PowerTransformer питоновской библиотеки scikit-learn

Как мы уже знаем, класс `PowerTransformer` библиотеки `scikit-learn` позволяет применить преобразование Бокса–Кокса или преобразование Йео–Джонсона к выбранным переменным. Кроме того, он позволяет выполнить еще и стандартизацию.

Применим преобразование Бокса–Кокса к переменным, принимающим только положительные значения. Обратите внимание: с помощью атрибута `lambda_` можно взглянуть на значения `lambda`, вычисленные для каждой переменной.

```
# создаем список количественных переменных
num_columns = ['PERSONAL_INCOME', 'AGE', 'CREDIT']
# смотрим минимальные значения
for c in num_columns:
    print(c, train[c].min())

PERSONAL_INCOME 1950.0
AGE 21
CREDIT 2000.0

# импортируем класс PowerTransformer
from sklearn.preprocessing import PowerTransformer

# создаем экземпляр класса PowerTransformer
# и обучаем его - вычисляем значения lambda
power = PowerTransformer(
    method='box-cox', standardize=False).fit(train[num_columns])

# смотрим значения lambda
print(power.lambdas_)

[-0.04318489  0.28126877 -0.03649645]
```

Теперь применим преобразование Йео–Джонсона к переменным, содержащим нулевые значения. Если бы мы применяли преобразование Бокса–Кокса, нам пришлось бы воспользоваться константой, чтобы превратить их в небольшие положительные значения, иначе получим ошибку `ValueError`: `The Box-Cox transformation can only be applied to strictly positive data.`

```
# создаем список количественных переменных
num_columns2 = ['FST_PAYMENT', 'FACT_LIVING_TERM', 'LOAN_AVG_DLQ_AMT']
# смотрим минимальные значения
for c in num_columns2:
    print(c, train[c].min())

FST_PAYMENT 0.0
FACT_LIVING_TERM 0.0
LOAN_AVG_DLQ_AMT 0.0

# создаем экземпляр класса PowerTransformer
# и обучаем его - вычисляем значения lambda
power = PowerTransformer(
    method='box-cox', standardize=False).fit(train[num_columns2])
```

ValueError: The Box-Cox transformation can only be applied to strictly positive data

```
# создаем экземпляр класса PowerTransformer  
# и обучаем его - вычисляем значения lambda  
power = PowerTransformer(  
    method='yeo-johnson', standardize=False).fit(train[num_columns2])  
# применяем преобразование Йео-Джонсона к переменным  
# в обучающем массиве признаков  
train[num_columns2] = power.transform(train[num_columns2])
```

16. Конструирование признаков



Рис. 34 Типы конструирования признаков

Выделяют два типа конструирования признаков – статическое конструирование признаков (static feature engineering) и динамическое конструирование признаков (dynamic feature engineering).

Статическое конструирование признаков подразумевает, что мы создаем признаки вручную. Мы создаем фактические признаки (они присутствуют в нашем наборе данных) и подаем на вход модели. Динамическое конструирование признаков подразумевает, что мы создаем признаки в ходе построения модели «на лету», эти признаки создаются только в процессе обучения модели и в наш набор данных не записываются.

Каждый тип можно разделить еще на два подтипа: конструирование признаков исходя из предметной области, опыта, бизнес-логики (domain-based feature engineering) и конструирование признаков исходя из особенностей алгоритма метода машинного обучения (algorithm-based feature engineering). Мы начнем со статического конструирования признаков исходя из предметной области, бизнес-логики.

16.1. СТАТИЧЕСКОЕ КОНСТРУИРОВАНИЕ ПРИЗНАКОВ ИСХОДЯ ИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

16.1.1. Поиск сильных переменных

Практически в любой задаче будут переменные, обладающие высокой прогнозной силой (признаки, сильнее всего связанные с зависимой переменной). Их нужно найти в первую очередь, если они отсутствуют, попробовать их создать или приобрести в ходе внешнего обогащения данными.

При построении скринговых моделей для розничных кредитов такими переменными будут отношение суммы кредитных обязательств к личному доходу, отношение размера ежемесячной выплаты по кредиту к размеру заработной платы, скринговый балл БКИ, количество просрочек, сумма просрочки (если несколько просрочек, смотрим максимальную сумму просрочки, общую сумму по просрочкам, общее количество дней по просрочкам), время с последней просрочки, возраст (обычно скачок около 40 лет), пол (мужчины платят хуже), утилизация (если есть кредитная карта), наличие ипотеки (большая кредитная нагрузка, ухудшающая кредитоспособность).

При построении скринговых моделей для МФО нужно учитывать краткосрочность продуктов, если у банка задача заключается в том, чтобы опреде-

лить размер дохода и наличие стабильного источника, то у МФО задача состоит в том, чтобы определить только наличие стабильного источника (мы и так знаем, что выдаем кредит человеку с низким доходом, которому банк отказал или с большой вероятностью откажет). У пенсионеров риск ниже, им пенсию гарантированно заплатят, а у заемщика на сдельной работе могут быть деньги через неделю, а могут и не быть. Для этого в анкеты МФО включают переменные, выявляющие регулярность выплаты зарплаты (промежуток между датой займа и датой последней зарплаты, промежуток между датой займа и датой ближайшей зарплаты, количество раз получения зарплаты). Однако эти переменные нужно тестировать, не факт, что люди будут правильно заполнять и что эти переменные вообще будут работать.

Например, можно дополнительно смотреть цикличность платежей по дням месяца и дням недели. Должны быть видны всплески, например 15-го и 30-го или по пятницам, в зависимости от того, как местные предприятия платят аванс/зарплату. Помимо переменных, фиксирующих регулярность выплаты зарплаты, нужно смотреть время дня, в которое обратился клиент за займом, а также сочетание времени обращения и сферы занятости – эта информация может быть объективнее. Здесь мы отделяем ситуацию, когда человек вечером или в обеденный перерыв пришёл, от ситуации, когда человек обратился в рабочее время (особенно это важно, если человек не является офисным работником, а работает на производстве).

При построении скоринговых моделей для ипотеки сильными переменными будет отношение суммы кредита к стоимости приобретаемого жилья (известно как показатель LTV – loan to value) или отношение суммы первоначального взноса к стоимости приобретаемого жилья.

При прогнозировании стоимости жилья сильной переменной будет средняя стоимость квартир в радиусе ближайших 300 метров от рассматриваемой квартиры (средняя стоимость квартир, находящихся в том же геохеше, что и рассматриваемая квартира) с поправкой на количество комнат, этажность и материал стен дома. Это вполне логично, ведь человек назначает стоимость квартиры, учитывая цены на похожие квартиры, продающиеся в этом же доме или ближайших домах с тем же метражом, этажностью и материалом стен. Однако это может не подойти для оценки элитной недвижимости, у нас просто могут отсутствовать аналогичные предложения в радиусе 300 метров от этой квартиры.

Также сильными переменными будут год постройки дома, «возраст» квартиры на момент сделки, площадь квартиры, расстояние до метро (в метрах и минутах), расстояние до глобального/локального «центра», расстояние до железнодорожной станции, расстояния до объектов торговой и социальной инфраструктуры (расстояние до ближайшей школы, детского сада, магазина и т. д.). В США сильной переменной может быть средняя стоимость обучения в ближайшей школе за год, люди готовы платить более высокую плату за недвижимость, чтобы их ребенок учился в более престижной школе.

16.1.2. Арифметические операции

Речь идет о переменных, полученных в результате деления одной переменной на другую (отношения), суммирования переменных (суммы), вычитания из одной переменной другой (разности), умножения одной переменной на другую. Например, у нас есть сумма кредитных обязательств за месяц и ежемесячный

доход, можем поделить сумму кредитных обязательств на ежемесячный доход и получить долговую нагрузку на доход. У нас есть суммы просрочек по трем кредитам и суммы трех кредитов, можем просуммировать суммы просрочек, просуммировать суммы кредитов, итоговую сумму просрочек поделить на итоговую сумму кредитов. У нас есть семейный доход и личный доход, можем из семейного дохода вычесть личный доход. У нас есть суммы транзакций, выделяем суммы транзакций пополнения и суммы транзакций снятия, из сумм транзакций пополнения можно вычесть суммы транзакций снятия. В идеале такие переменные нужно создавать на основе бизнес-логики и здравого смысла, однако если знаний в рассматриваемой предметной области недостаточно или признаки анонимизированы, то можно отобрать топ-10 признаков на основе взаимной информации или SHAP (при этом помним, что отбор признаков – это тоже модель, которая строится на обучающей выборке, а на тестовой проверяется, если используется настройка количества отбираемых признаков, то потребуется проверочная выборка или можно использовать проверочные блоки перекрестной проверки, запущенной на обучающей выборке) и выполнить перечисленные арифметические действия с этими отобранными признаками.

Давайте создадим переменную *payt*, которая является отношением выданной суммы кредита к сроку кредита, то есть ежемесячной суммой кредита. При вычислении признаков, которые являются отношениями количественных признаков, в случае когда происходит деление на ноль, могут быть получены бесконечные значения. Нужно предупредить возникновение такой ситуации.

```
# создаем переменную payt, которая является отношением выданной
# суммы кредита (credit_sum) к сроку кредита (credit_month),
# то есть ежемесячной суммой кредита
cond = (data['credit_sum'] == 0) | (data['credit_month'] == 0)
data['payt'] = np.where(cond, 0,
                      data['credit_sum'] / data['credit_month'])
data.head()
```

job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income	credit_count	overdue_credit_count	payt
SPC	17679.00	12	1_1	0.417126	GRD	КЕМЕРОВСКАЯ	25000.0	1.0	0.0	1473.250000
SPC	17353.33	10	1_1	0.488022	SCH	ВОЛГОГРАДСКАЯ	30000.0	0.0	0.0	1735.333000
UMN	15369.00	10	1_6	0.707397	GRD	ТАТАРСТАН	35000.0	4.0	0.0	1538.900000
SPC	18301.00	10	1_32	0.343173	GRD	ТВЕРСКАЯ	30000.0	0.0	0.0	1830.100000
PNA	11459.00	18	1_1	0.554380	SCH	ЧЕЛЯБИНСКАЯ	11000.0	0.0	0.0	636.611111

16.1.3. Создание переменной, у которой значения основаны на значениях исходной переменной

Часто при конструировании признаков, исходя из предметной области, нужно проявить фантазию, и вы должны понимать, что сам признак – «шкатулка с секретом», часто просто указатель, в какую сторону нужно двигаться, чтобы извлечь ценную информацию. Например, нередко приходится работать с таким признаком, как сфера занятости, который обычно состоит из множества категорий. Сама по себе сфера занятости может быть слабо связана с кредитоспособностью, однако если мы заменим сферу занятости средней заработной платой в этой сфере, то можем получить более полезный признак. Регионы мы можем

заменить средней заработной платой в регионе, размером и глубиной просрочки в регионе, криминогенностью региона. Чаще всего мы подобные операции выполняем через создание переменной, у которой значения основаны на значениях исходной переменной. Давайте приведем пример такой переменной.

Сначала выведем уникальные значения исходной переменной, в данном случае – значения переменной *job_position*.

```
# выводим уникальные значения исходной переменной,
# в данном случае – значения переменной job_position
print(data['job_position'].unique())
['SPC' 'UMN' 'PNA' 'WOI' 'WRK' 'BIS' 'ATP' 'DIR' 'NOR' 'OTHER' 'INP' 'WRP' 'BIU' 'PNI']
```

Допустим, мы получили из какого-то внешнего источника информацию о средней заработной плате клиентов с различными профессиями. Тогда создаем словарь, в котором ключом будет значение исходной переменной *job_position*, а значением – значение нашей будущей переменной *avrzarplata*.

```
# затем создаем словарь, в котором ключом будет значение
# исходной переменной job_position, а значением – значение
# будущей переменной avrzarplata
dct = {'UMN': 51000, 'SPC': 63000, 'INP': 55000, 'DIR': 60000,
       'ATP': 46000, 'PNA': 71000, 'BIS': 86000, 'WOI': 76000,
       'NOR': 54000, 'WRK': 77000, 'WRP': 75000, 'PNV': 67000,
       'BIU': 43000, 'PNI': 69000, 'HSK': 74000, 'PNS': 44000,
       'INV': 88000, 'ONB': 62000, 'OTHER': 20000}
# создаем новую переменную avrzarplata, у которой значения
# сопоставлены значениям переменной job_position
data['avrzarplata'] = data['job_position'].map(dct)
data.head()
```

living_region	monthly_income	credit_count	overdue_credit_count	paym	avrzarplata
КЕМЕРОВСКАЯ	25000.0	1.0	0.0	1473.250000	63000
ВОЛГОГРАДСКАЯ	30000.0	0.0	0.0	1735.333000	63000
ТАТАРСТАН	35000.0	4.0	0.0	1538.900000	51000
ТВЕРСКАЯ	30000.0	0.0	0.0	1830.100000	63000
ЧЕЛЯБИНСКАЯ	11000.0	0.0	0.0	636.611111	71000

16.1.4. Создание бинарной переменной на основе значений количественной переменной

Большую важность имеют бинарные переменные (их еще называют переменными-флагами). Возьмем такую сферу, как рынок недвижимости. При прогнозировании стоимости квадратного метра жилья нужно учитывать, что квартиры на первом, на последнем этаже, без балкона стоят дешевле, и поэтому можно создать переменные-флаги *Квартира находится на первом этаже*, *Квартира находится на последнем этаже*, *Квартира без балкона*. При этом при создании переменных *Квартира находится на первом этаже*, *Квартира наход-*

дится на последнем этаже нужно учитывать, что, скорее всего, для таунхаусов, 2–3-этажных шлакоблочных домов вряд ли этот индикатор будет работать, поэтому данный факт нужно оговорить специальным условием. Квартиры на втором и третьем этажах в целом привлекательны, но если на первом этаже находится бар, ресторан или магазин, это может стать фактором, понижающим стоимость. Можно сделать индикатор, который в случае наличия магазина на первом этаже для квартир, расположенных на первых трех этажах, принимает значение 1 или значение 0 в противном случае. Квартиры, которые продаются по ипотеке, как правило, стоят дороже, можно создать индикатор *Квартира была реализована по ипотеке*.

Давайте создадим бинарную переменную, основываясь на значениях одной или нескольких количественных переменных. Сначала создадим переменную *retired*, которая принимает значение 'Yes', если значение переменной *age* больше 60, и значение 'No' в противном случае.

```
# создаем новую переменную retired, которая принимает
# значение 'Yes', если значение переменной age больше 60,
# и значение 'No' в противном случае
data['retired'] = np.where(data['age'] >= 60, 'Yes', 'No')
data.head()
```

living_region	monthly_income	credit_count	overdue_credit_count	paym	avrzarplata	retired
КЕМЕРОВСКАЯ	25000.0	1.0	0.0	1473.250000	63000	No
ВОЛГОГРАДСКАЯ	30000.0	0.0	0.0	1735.333000	63000	No
ТАТАРСТАН	35000.0	4.0	0.0	1538.900000	51000	No
ТВЕРСКАЯ	30000.0	0.0	0.0	1830.100000	63000	No
ЧЕЛЯБИНСКАЯ	11000.0	0.0	0.0	636.611111	71000	Yes

А теперь создадим переменную *age_inc*, которая принимает значение 'Yes', если речь идет о клиентах старше 35 лет и с суммой кредита свыше 10 000, и значение 'No' в противном случае.

```
# создаем новую переменную age_inc, которая принимает
# значение 'Yes', если речь идет о клиентах старше
# 35 лет и с суммой кредита свыше 10 000,
# и значение 'No' в противном случае
cond = (data['age'] > 35) & (data['credit_sum'] > 10000)
data['age_inc'] = np.where(cond, 'Yes', 'No')
data.head()
```

monthly_income	credit_count	overdue_credit_count	paym	avrzarplata	retired	age_inc
25000.0	1.0	0.0	1473.250000	63000	No	No
30000.0	0.0	0.0	1735.333000	63000	No	Yes
35000.0	4.0	0.0	1538.900000	51000	No	No
30000.0	0.0	0.0	1830.100000	63000	No	Yes
11000.0	0.0	0.0	636.611111	71000	Yes	Yes

16.1.5. Создание переменной, у которой каждое значение – среднее значение количественной переменной, взятое по уровню категориальной переменной

При решении задач часто повысить качество модели позволяет переменная, у которой каждое значение – это среднее значение количественной переменной, взятое по уровню категориальной переменной (групповое среднее).

```
# создаем переменную, у которой каждое значение –
# среднее значение monthly_income в категории
# переменной living_region
means = data.groupby('living_region')['monthly_income'].mean()
data['region_mean_income'] = data['living_region'].map(means)
data.head()
```

living_region	monthly_income	credit_count	overdue_credit_count	paym	avrzarplata	retired	age_inc	region_mean_income
КЕМЕРОВСКАЯ	25000.0	1.0	0.0	1473.250000	63000	No	No	31722.560294
ВОЛГогРАДСКАЯ	30000.0	0.0	0.0	1735.333000	63000	No	Yes	31708.270771
ТАТАРСТАН	35000.0	4.0	0.0	1538.900000	51000	No	No	36600.517612
ТВЕРСКАЯ	30000.0	0.0	0.0	1830.100000	63000	No	Yes	34083.284007
ЧЕЛЯБИНСКАЯ	11000.0	0.0	0.0	636.611111	71000	Yes	Yes	33618.413074

Поскольку здесь мы используем вычисления по набору данных (вычисляем среднее), то такие переменные нужно создавать после разбиения на обучающую и тестовую выборки (внутри цикла перекрестной проверки). Для категорий или комбинаций категорий в тестовой выборке используем групповые статистики, вычисленные для категорий или комбинаций категорий в обучающей выборке. При этом в тестовой выборке может быть комбинация категорий признаков, которая отсутствует в обучающей выборке. В таком случае у нас будет пропуск, который чаще всего кодируют отдельным значением вне диапазона или средним значением, вычисленным в обучающем датафрейме (менее предпочтительный вариант). Групповые средние, как правило, используют для улучшения качества моделей на основе ансамблей деревьев.

Давайте напишем класс MeanGrouper, вычисляющий групповые статистики.

```
# пишем класс, вычисляющий групповые статистики
class MeanGrouper():
    """
```

Класс, вычисляющий групповые статистики.

Параметры

group_cols: list

Список группирующих столбцов.

agg_col: str

Агрегируемый столбец.

agg_func: str

Агрегирующая функция.

Возвращает

```

X : pandas.DataFrame
    Датафрейм с групповыми статистиками.
"""

def __init__(self, group_cols, agg_col, agg_func='mean'):

    if agg_func not in ['mean', 'median', 'min', 'max']:
        raise ValueError(
            f"Неизвестная агрегирующая функция {agg_func}")

    if type(agg_func) != str:
        raise ValueError("Агрегирующая функция должна" +
                         "иметь строковое значение")

    # превращаем столбец в список
    if type(group_cols) != list:
        group_cols = [group_cols]

    self.group_cols = group_cols
    self.agg_col = agg_col
    self.agg_func = agg_func

def fit(self, X, y=None):

    # проверка наличия пропусков в группирующих столбцах
    if pd.isnull(X[self.group_cols]).any(axis=None) == True:
        raise ValueError(f"Есть пропуски в группирующих столбцах")

    # задаем имя
    self.name = '{0}_by_{1}_{2}'.format(
        self.agg_col, self.group_cols, self.agg_func)

    # получаем средние по каждой комбинации категорий
    # признаков в списке
    self.grp = X.groupby(self.group_cols)[[self.agg_col]].agg(
        self.agg_func)

    return self

def transform(self, X, y=None):

    # присваиваем имя
    self.grp.columns = [self.name]
    # записываем результаты
    X = pd.merge(X, self.grp,
                 left_on=self.group_cols,
                 right_index=True, how='left')

    # пропуски кодируем отдельным значением -1
    X = X.fillna(-1)

    return X

```

Создаем игрушечные обучающий и тестовый датафреймы. При этом в тестовом датафрейме у нас будет комбинация категорий признаков, которая отсутствует в обучающем датафрейме (выделена синей рамкой), что является часто встречающейся ситуацией на практике.

```
# создаем игрушечный обучающий датафрейм
tr_example = pd.DataFrame(
    {'income': [10, 20, 40, 25, 15],
     'region': ['MSK', 'SPB', 'SPB', 'MSK', 'SPB'],
     'agecat': ['yng', 'old', 'yng', 'yng', 'yng']})

# создаем игрушечный тестовый датафрейм
tst_example = pd.DataFrame(
    {'income': [15, 45, 15, 75, 30],
     'region': ['MSK', 'SPB', 'SPB', 'MSK', 'SPB'],
     'agecat': ['yng', 'old', 'yng', 'old', 'yng']})
```

Давайте взглянем на эти датафреймы.

```
# смотрим игрушечный обучающий датафрейм
tr_example
```

	income	region	agecat
0	10	MSK	yng
1	20	SPB	old
2	40	SPB	yng
3	25	MSK	yng
4	15	SPB	yng

```
# смотрим игрушечный тестовый датафрейм
tst_example
```

	income	region	agecat
0	15	MSK	yng
1	45	SPB	old
2	15	SPB	yng
3	75	MSK	old
4	30	SPB	yng

Теперь вычислим доход, усредненный по комбинациям региона и возрастной группы.

```
# применяем наш класс
grp = MeanGrouper(['region', 'agecat'],
                    agg_col='income',
                    agg_func='mean')
imp.fit(tr_example)
tr_example = grp.transform(tr_example)
tst_example = grp.transform(tst_example)
```

Смотрим результаты.

```
# смотрим результаты в игрушечном
# обучающем датафрейме
tr_example
```

	income	region	agecat	income_by_['region', 'agecat']_mean
0	10	MSK	yng	17.5
1	20	SPB	old	20.0
2	40	SPB	yng	27.5
3	25	MSK	yng	17.5
4	15	SPB	yng	27.5

смотрим результаты в игрушечном

тестовом датафрейме

tst_example

	income	region	agecat	income_by_['region', 'agecat']_mean
0	15	MSK	yng	17.5
1	45	SPB	old	20.0
2	15	SPB	yng	27.5
3	75	MSK	old	-1.0
4	30	SPB	yng	27.5

Здесь нас особо интересует тестовый набор. Мы видим, что для комбинаций категорий, которые есть в обучающем наборе, мы используем средние, вычисленные для этих комбинаций по обучающему набору. Комбинация категорий MSK – old отсутствует в обучающем наборе (выделена синей рамкой), соответственно, в тестовом наборе получает пропуск, который мы заменяем –1.

16.1.6. Объединение нескольких бинарных переменных в одну количественную переменную

Часто бинарные переменные можно объединить в количественную переменную. Например, у нас есть несколько бинарных переменных, измеряющих благосостояние клиента (наличие дома, наличие коттеджа, наличие гаража, наличие земельного участка). Можно создать переменную – оценку благосостояния клиента как сумму единичных значений этих бинарных переменных.

```
# загружаем данные
data = pd.read_csv('Data/FE2.csv', sep=';')
# выводим наблюдения
data[100:105]
```

	HS_PRESENCE_FL	COT_PRESENCE_FL	GAR_PRESENCE_FL	LAND_PRESENCE_FL
100	1	0	0	1
101	0	0	0	0
102	0	0	0	0
103	1	0	0	1
104	0	0	0	0

```
# вычисляем балл как сумму единичных
# значений бинарных переменных
data['score'] = data.sum(axis=1)
data[100:105]
```

	HS_PRESENCE_FL	COT_PRESENCE_FL	GAR_PRESENCE_FL	LAND_PRESENCE_FL	score
100	1	0	0	1	2
101	0	0	0	0	0
102	0	0	0	0	0
103	1	0	0	1	2
104	0	0	0	0	0

16.1.7. Вычисление расстояния между двумя точками по географическим координатам (через формулу гаверсинусов)

Анализуя данные о продажах квартир, мы можем встретить данные о координатах квартиры (широте и долготе).

```
# загружаем и смотрим данные
data = pd.read_csv('Data/Flats.csv', encoding='cp1251', sep=';')
data.head()
```

Rooms_Number	Object_Type	Settlement	District	Street	House_Number	Metro	Metro_nm	Space_Total	Lat	Long	Date_Postr
0	1	Хрущевка	Новосибирск	Первомайский	Маяковского	24	Речной вокзал	11260	30.5	54.9738	83.0860
1	1	Типовая	Новосибирск	Первомайский	Звездная	14	Речной вокзал	12130	28.9	54.9551	83.0681
2	1	Типовая	Новосибирск	Центральный	Лермонтова	36	Сибирская	438	31.9	55.0446	82.9267
3	2	Хрущевка	Новосибирск	Центральный	Достоевского	20	Сибирская	312	44.7	55.0456	82.9192
4	2	Улучшенный планировки	Новосибирск	Советский	Демакова	12	Речной вокзал	23340	49.0	54.8634	83.1041

В таких случаях мы можем определить глобальный центр как точку, характеризующуюся максимальной концентрацией наиболее дорогих квартир, или точку, где расположено наибольшее количество деловых и торговых центров. Можно взять исторический центр. Можно найти такой «центр» с точки зрения экологической обстановки, криминогенности, транспортной доступности. Можно взять за центр адрес почтамта. Далее мы вычисляем удаленность квартиры от «центра» по формуле гаверсинусов. Как вариант можно в разрезе районов/микрорайонов найти локальные центры и в разрезе каждого района/микрорайона также по формуле гаверсинусов вычислить удаленность квартиры от локального «центра». Часто в качестве локального центра берут адрес органа власти в данной административно-территориальной единице (например, адрес администрации района).

Чтобы разобраться в работе формулы, необходимо начать с такого определения, как большой круг. Большой круг – круг, получаемый при сечении шара плоскостью, проходящей через его центр. Диаметр любого большого круга совпадает с диаметром сферы, поэтому все большие круги имеют одинаковый периметр и один центр, совпадающий с центром шара. Иногда под термином

«большой круг» подразумевают большую окружность, то есть окружность, получаемую при сечении сферы плоскостью, проходящей через её центр.

Сферическая геометрия отличается от обычной евклидовой, и уравнения расстояния также принимают другую форму. В евклидовой геометрии кратчайшее расстояние между двумя точками – прямая линия. На сфере прямых линий не бывает. Эти линии на сфере являются частью больших кругов – окружностей, центры которых совпадают с центром сферы.

Большой круг делит сферу на две полусфера

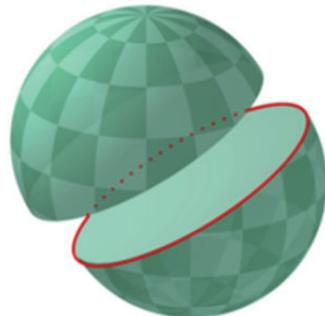


Рис. 35 Большой круг (источник: Википедия)

Через любые две точки на поверхности сферы, если они не прямо противоположны друг другу (то есть не являются антиподами), можно провести уникальный большой круг. Через две противоположные точки можно провести бесконечно много больших кругов, но расстояние между ними будет одинаково на любом круге и равно половине окружности круга, или πR , где R – радиус сферы. Меньшая дуга большого круга между двумя точками является кратчайшим расстоянием между ними по поверхности сферы (ортодромией). В этом смысле большие круги выполняют роль прямых линий в сферической геометрии.

Длина короткой дуги – кратчайшее расстояние между двумя точками

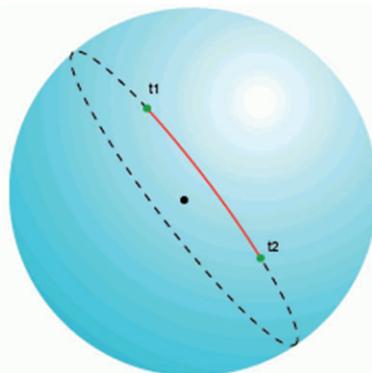


Рис. 36 Меньшая дуга большого круга (источник: Википедия)

Для вычисления сферических расстояний между двумя точками как раз и применяется формула гаверсинусов:

$$d = 2r \arcsin \left(\sqrt{\text{haversin}(\phi_2 - \phi_1) + \cos(\phi_1) \cos(\phi_2) \text{haversin}(\lambda_2 - \lambda_1)} \right) = \\ = 2r \arcsin \left(\sqrt{\sin^2 \left(\frac{\phi_2 - \phi_1}{2} \right) + \cos(\phi_1) \cos(\phi_2) \sin^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right)} \right),$$

где:

r – радиус Земли (6 372 795 метров), необходим для перевода углового расстояния в метрическое (переводим радианы в метры);

ϕ_1, ϕ_2 – широта точки 1 и широта точки 2 в радианах;

λ_1, λ_2 – долгота точки 1 и долгота точки 2 в радианах.

Меньшая дуга большого круга (расстояние между точками P и Q) выделена красным, также показаны точки-антиподы u и v

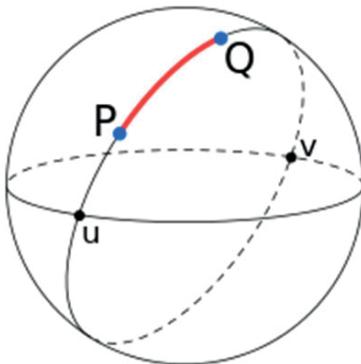


Рис. 37 Меньшая дуга большого круга и точки-антиподы (источник: Википедия)

Для вычисления расстояния между двумя точками по формуле гаверсинусов мы воспользуемся функцией `haversine()` из одноименного пакета <https://github.com/mapado/haversine>. Мы не будем ее импортировать, а подробно разберем скрипт `haversine.py`, в котором она записана.

Из библиотеки `math` мы импортируем необходимые математические функции. Кроме того, нам потребуется класс `Eunit`. Он нужен для создания перечислений – набора ограниченных неизменяемых значений, которые можно присвоить переменной.

```
# импортируем необходимые функции
from math import radians, cos, sin, asin, sqrt, pi
# класс Eunit нужен для создания перечислений – набора
# ограниченных неизменяемых значений, которые можно
# присвоить переменной
from enum import Eunit
```

Записываем в переменную радиус Земли в километрах и для работы с различными единицами измерения создаем класс `Unit` с классом-родителем `Enum`.

```
# записем радиус Земли в километрах
_AVG_EARTH_RADIUS_KM = 6371.0088

# создаем класс для работы с различными
# единицами измерения
class Unit(Enum):
    KILOMETERS = 'km'
    METERS = 'm'
    MILES = 'mi'
    NAUTICAL_MILES = 'nmi'
    FEET = 'ft'
    INCHES = 'in'
    RADIANS = 'rad'
    DEGREES = 'deg'
```

Затем создаем словарь для выполнения преобразований из одной единицы в другую.

```
# создаем словарь для выполнения преобразований
# одной единицы измерения в другую
_CONVERSIONS = {
    Unit.KILOMETERS:      1.0,
    Unit.METERS:          1000.0,
    Unit.MILES:            0.621371192,
    Unit.NAUTICAL_MILES:  0.539956803,
    Unit.FEET:              3280.839895013,
    Unit.INCHES:            39370.078740158,
    Unit.RADIANS:           1/_AVG_EARTH_RADIUS_KM,
    Unit.DEGREES:            (1/_AVG_EARTH_RADIUS_KM)*(180.0/pi)
}
```

Пишем функцию `get_avg_earth_radius()`, которая вычисляет радиус Земли в зависимости от заданной единицы измерения. Она будет использоваться внутри функции `haversine()`.

```
# пишем функцию, вычисляющую радиус Земли в зависимости
# от заданной единицы измерения
def get_avg_earth_radius(unit):
    unit = Unit(unit)
    return _AVG_EARTH_RADIUS_KM * _CONVERSIONS[unit]
```

Наконец, пишем функцию `haversine()`, которая вычисляет расстояние между двумя точками по формуле гаверсинусов.

```
# пишем функцию вычисления расстояния между двумя точками
# по формуле гаверсинусов
def haversine(point1, point2, unit=Unit.KILOMETERS):
    """
    Параметры
    -----
    point1: 2-элементный кортеж
        Координаты первой точки (широта, долгота в градусах).
    
```

```

point2: 2-элементный кортеж
    Координаты второй точки (широта, долгота в градусах).
unit: str, по умолчанию 'км'
    Единица измерения.

Возвращает
-----
distance: расстояние между двумя точками
"""
# задаем координаты
lat1, lng1 = point1
lat2, lng2 = point2

# переводим градусы в радианы
lat1 = radians(lat1)
lng1 = radians(lng1)
lat2 = radians(lat2)
lng2 = radians(lng2)

# вычисляем разницу координат по широте
lat = lat2 - lat1

# вычисляем разницу координат по долготе
lng = lng2 - lng1

# вычисляем угловое расстояние в радианах
d = sin(lat * 0.5) ** 2 + cos(lat1) * cos(lat2) * sin(lng * 0.5) ** 2
# возвращаем расстояние в заданных единицах измерения
return 2 * get_avg_earth_radius(unit) * asin(sqrt(d))

```

Сейчас мы зададим координаты центра и вычислим расстояния от каждой квартиры до «центра» в метрах, воспользовавшись функцией `haversine()`.

```

# задаем координаты «центра», сначала широту, затем долготу
city_center_coordinates = (55.0415000, 82.9346000)
# создаем переменную – расстояние квартиры от «центра»
# в метрах, воспользовавшись функцией haversine()
data['dist_center'] = data.apply(
    lambda row: haversine(
        row['Lat'], row['Long'],
        city_center_coordinates,
        unit='м'), axis=1)
# смотрим результаты
data.head()

```

District	Street	House_Number	Metro	Metro_m	Space_Total	Lat	Long	Date_Postr	dist_center
Первомайский	Маяковского	24	Речной вокзал	11260	30.5	54.9738	83.0860	1971	12242.320295
Первомайский	Звездная	14	Речной вокзал	12130	28.9	54.9551	83.0681	1974	12837.513174
Центральный	Лермонтова	36	Сибирская	438	31.9	55.0446	82.9267	1973	610.036608
Центральный	Достоевского	20	Сибирская	312	44.7	55.0456	82.9192	1964	1081.876736
Советский	Демакова	12	Речной вокзал	23340	49.0	54.8634	83.1041	1989	22568.472530

16.1.8. Геохеширование

Геохеш – это открытая система кодирования геоданных, изобретенная Густаво Нимейером (Gustavo Niemeyer) и преобразующая данные географического местонахождения (широту и долготу) в короткую строку, состоящую из букв и цифр. Это иерархическая пространственная структура данных, разделяющая пространство на сегменты в виде сетки. Она является одной из множества реализаций кривой Z-порядка (Z-order curve), относящейся к семейству «кривых, заполняющих пространство» (space-filling curves).

Геохеш обладает такими полезными свойствами, как настраиваемая точность и возможность последовательно удалять символы с конца кода с постепенной потерей точности для уменьшения размера. Вследствие постепенной утери точности близлежащие места будут часто (хотя и не всегда) обладать схожими префиксами. Чем длиннее совпадающая часть префикса, тем ближе две точки будут расположены друг к другу.

Например, пара координат 57.64911, 10.40744 (вблизи оконечности полуострова Ютландия в Дании) преобразуется в несколько более короткий геохеш `u4rguydqyj`.

В таблице ниже показана взаимосвязь между длиной конкретного геохеша и точностью фиксации координат.

Таблица 6 Взаимосвязь между длиной конкретного геохеша и точностью фиксации координат

Геохеш	Широта, долгота	Мин. широта, мин. долгота, макс. широта, макс. долгота
9	22.5, -112.5	0, -135, 45, -90
9q	36.5625, -118.125	33.75, -123.75, 39.375, -112.5
9q4	34.45312, -120.23437	33.75, -120.9375, 35.15625, -119.53125
9q4g	34.36523, -119.70703	34.27734, -119.88281, 34.45312, -119.53125
9q4gu	34.43115, -119.68505	34.40917, -119.70703, 34.45312, -119.66308
9q4gu1	34.41741, -119.70153	34.41467, -119.70703, 34.42016, -119.69604
9q4gu1y	34.41947, -119.69810	34.41879, -119.69879, 34.42016, -119.69741
9q4gu1y4	34.41922, -119.69861	34.41913, -119.69879, 34.41930, -119.69844
9q4gu1y4z	34.41928, -119.69846	34.41926, -119.69849, 34.41930, -119.69844

Главным образом геохеш может применяться как:

- уникальный идентификатор;
- представление точечных данных, например в базах данных;
- геохеши также было предложено использовать для геотеггинга.

Использование геохешированных данных имеет два преимущества. Во-первых, данные, проиндексированные по геохешу, будут представлять собой срезы заданной прямоугольной зоны, количество которых зависит от требуемой точности и наличия «сбойных строк» геохешей. Это особенно полезно в тех базах данных, где запросы по одному индексу выполняются намного проще или же быстрее, чем запросы по нескольким индексам. Во-вторых, такая структура индекса может применяться для быстрого приближенного поиска расстояний, поскольку ближайшие точки обычно имеют схожие геохеши.

В основе геохеша лежит кодировка base32: используем все цифры (0–9) и почти все строчные буквы алфавита, кроме а, i, l, o. Приведем пример декодиро-

вания хеша ezs42 в широту и долготу в десятичном виде. Сначала необходимо декодировать их из формата base 32, используя следующую таблицу символов:

Десятичный	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Десятичный	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base 32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

Результатом операции станет следующая последовательность битов: 011011111100000100. Начав считать слева с цифры 0 в первой позиции, мы условимся использовать цифры в нечетных позициях для кодирования долготы (01111100000000), а цифры в четных позициях – для кодирования широты (101111001001).

Затем каждый бинарный код будет использоваться в серии побитовых операций деления, снова слева направо. Для значения широты интервал от -90 до +90 градусов будет разделен пополам, образуя два интервала: от -90 до 0 градусов и от 0 до +90 градусов соответственно. Поскольку первый бит равен 1, выбирается большая по значению половина интервала, которая и становится текущей. Процедура повторяется для всех битов кода. Искомое значение широты находится в центре последнего результирующего интервала. Долгота вычисляется аналогичным образом, с учетом того, что начальный интервал составляет от -180 до +180 градусов.

Допустим, код широты 101111001001. Первый бит слева равен 1, поэтому искомое значение широты находится где-то между 0 и 90 градусами. При отсутствии других битов мы примем значение широты, равное 45, что даст ошибку ± 45 градусов. Но поскольку имеются дополнительные биты, операцию деления можно повторить, при этом каждый последующий бит уменьшит ошибку вдвое. Влияние на точность операции деления для каждого бита показано в таблице. На каждом шаге соответствующая половина диапазона обозначена зеленым цветом. Нулевое значение бита обозначает нижнюю часть диапазона, единица – верхнюю его часть.

В колонке «mean value» приводится значение широты, просто расположено в центре текущего диапазона. Каждый последующий бит все более уточняет его, уменьшая ошибку.

Таблица 7 От битов к координатам

Код широты 101111001001

bit position	bit value	min	mid	max	mean value	maximum error
0	1	-90.000	0.000	90.000	45.000	45.000
1	0	0.000	45.000	90.000	22.500	22.500
2	1	0.000	22.500	45.000	33.750	11.250
3	1	22.500	33.750	45.000	39.375	5.625
4	1	33.750	39.375	45.000	42.188	2.813
5	1	39.375	42.188	45.000	43.594	1.406
6	0	42.188	43.594	45.000	42.891	0.703
7	0	42.188	42.891	43.594	42.539	0.352
8	1	42.188	42.539	42.891	42.715	0.176
9	0	42.539	42.715	42.891	42.627	0.088
10	0	42.539	42.627	42.715	42.583	0.044
11	1	42.539	42.583	42.627	42.605	0.022

Код долготы 011111000000

bit position	bit value	min	mid	max	mean value	maximum error
0	0	-180.000	0.000	180.000	-90.000	90.000
1	1	-180.000	-90.000	0.000	-45.000	45.000
2	1	-90.000	-45.000	0.000	-22.500	22.500
3	1	-45.000	-22.500	0.000	-11.250	11.250
4	1	-22.500	-11.250	0.000	-5.625	5.625
5	1	-11.250	-5.625	0.000	-2.813	2.813
6	0	-5.625	-2.813	0.00	-4.219	1.406
7	0	-5.625	-4.219	-2.813	-4.922	0.703
8	0	-5.625	-4.922	-4.219	-5.273	0.352
9	0	-5.625	-5.273	-4.922	-5.449	0.176
10	0	-5.625	-5.449	-5.273	-5.537	0.088
11	0	-5.625	-5.537	-5.449	-5.581	0.044
12	0	-5.625	-5.581	-5.537	-5.603	0.022

Для наглядности числа в таблице были округлены до трех десятичных знаков после запятой.

Итоговое округление следует выполнять очень осторожно, следя за соблюдением условия

$$\min < \text{round}(\text{значение}) < \max.$$

Поэтому округление 42,605 до 42,61 или 42,6 будет верным, а до 43 – уже нет.

В таблице ниже показана взаимосвязь между длиной конкретного геохеша и ошибкой.

Таблица 8 Взаимосвязь между длиной конкретного геохеша и ошибкой

Длина геохеша	Бит широты	Бит долготы	Ошибка, градусов широты	Ошибка, градусов долготы	Ошибка, км
1	2	3	±23	±23	±2500
2	5	5	±2,8	±5,6	±630
3	7	8	±0,70	±0,70	±78
4	10	10	±0,087	±0,18	±20
5	12	13	±0,022	±0,022	±2,4
6	15	15	±0,0027	±0,0055	±0,61
7	17	18	±0,00068	±0,00068	±0,076
8	20	20	±0,000085	±0,00017	±0,019

Геохеши можно использовать для поиска точек в непосредственной близости друг от друга на основании общего префикса кода. Однако существуют пограничные случаи.

Например, два местоположения могут находиться близко друг к другу, но по разным сторонам меридиана 180° . В этом случае геохеши не будут иметь общего префикса (разные значения долготы для близко расположенных точек). Точки, расположенные к Северному и Южному полюсам, будут иметь сильно отличающиеся геохеши (разные значения долготы для близко расположенных точек).

Два близких местоположения по разным сторонам экватора (или гринвичского меридиана) также не будут иметь общего длинного префикса, поскольку лежат в разных «половинах» мира. Проще говоря, двоичная широта (или долгота) одного местоположения будет 011111..., а другого – 100000..., поэтому общий префикс будет отсутствовать, а большинство битов – зеркально перевернуто.

Кроме того, необходимо помнить про нелинейность. Поскольку большинство существующих реализаций геохеша основаны на координатах долготы и широты, расстояние между двумя геохешами отражает расстояние в координатах широта/долгота между двумя точками, которое не преобразуется в фактическое расстояние (здесь смотрите формулу гаверсинусов).

Приведем примеры нелинейности для системы широта–долгота.

На экваторе (0 градусов) длина градуса долготы составляет 111,320 км, а градуса широты – 110,574 км, ошибка равна 0,67 %.

При 30 градусах (средние широты) ошибка составляет $110,852 / 96,486 = 14,89\%$.

Однако эти ограничения связаны не с геохешированием и не с самими координатами, а со сложностью отображения координат на сфере в двумерные координаты.

Давайте загрузим данные по продажам недвижимости, у нас есть координаты квартир, выполним их геохеширование с помощью библиотеки pygeohash.

```
# импортируем библиотеку pygeohash для геохеширования
import pygeohash as gh
# создаем переменную - геохеш
data['geohash'] = data.apply(lambda x: gh.encode(
    x['Lat'], x['Long'], precision=5), axis=1)
# выводим первые пять наблюдений датафрейма
data.head()
```

Settlement	District	Street	House_Number	Metro	Metro_m	Space_Total	Lat	Long	Date_Postr	dist_center	geohash
Новосибирск	Первомайский	Маяковского	24	Речной вокзал	11260	30.5	54.9738	83.0860	1971	12242.320295	vcg0d
Новосибирск	Первомайский	Звездная	14	Речной вокзал	12130	28.9	54.9551	83.0681	1974	12837.513174	vcg0d
Новосибирск	Центральный	Лермонтова	36	Сибирская	438	31.9	55.0446	82.9267	1973	610.036608	vcfcp
Новосибирск	Центральный	Достоевского	20	Сибирская	312	44.7	55.0456	82.9192	1964	1081.876736	vcfcn
Новосибирск	Советский	Демакова	12	Речной вокзал	23340	49.0	54.8634	83.1041	1989	22568.472530	vcg05

В задачах машинного обучения геохеш редко выступает в качестве самостоятельного признака (часто начинающие специалисты пытаются использовать геохеш как еще один категориальный признак). Геохеш служит в качестве вспомогательной группирующей переменной. Допустим, мы прогнозируем стоимость квартиры, нас будет интересовать нагруженность месторасположения социальной инфраструктурой, для этого мы вычислим количество детских садов, школ, торговых центров, магазинов, остановок общественного транспорта (можно считать количество маршрутов), находящихся в том же геохеше, что и рассматриваемая квартира. Кроме того, можно пойти дальше и вычислить среднюю стоимость обучения в школах или средний размер чека в магазинах, находящихся в том же геохеше, что и интересующая нас квартира.

Кроме того, можно вычислить геохеш условного глобального или локально-го «центра» и получить новую переменную – количество совпадающих символов в геохеше координат квартиры и геохеше координат «центра». Чем больше совпадающих символов, тем ближе квартира расположена к «центру».

```
# вычисляем геохеш «центра»
geohash_center = gh.encode(55.0415000, 82.9346000, precision=5)
geohash_center

'vefcfp'

# преобразовываем геохеш «центра» во множество
geohash_center_set = set(geohash_center)
# создаем переменную - количество совпадающих символов
# (учитываем повторяющиеся символы)
data['match'] = data['geohash'].apply(
    lambda x: sum([p1 == p2 for p1, p2 in zip(geohash_center, x)]))
```

```
# создаем переменную - количество совпадающих символов
# (не учитываем повторяющиеся символы)
data['match2'] = data['geohash'].apply(
    lambda x: len(set(x)) & geohash_center_set))
data.head()
```

District	Street	House_Number	Metro	Metro_m	Space_Total	Lat	Long	Date_Postr	dist_center	geohash	match	match2
Первомайский	Маяковского	24	Речной вокзал	11260	30.5	54.9738	83.0860	1971	12242.320295	vcg0d	2	2
Первомайский	Звездная	14	Речной вокзал	12130	28.9	54.9551	83.0681	1974	12837.513174	vcg0d	2	2
Центральный	Лермонтова	36	Сибирская	438	31.9	55.0446	82.9267	1973	610.036608	vfcfp	5	4
Центральный	Достоевского	20	Сибирская	312	44.7	55.0456	82.9192	1964	1081.876736	vfcfn	4	3
Советский	Демакова	12	Речной вокзал	23340	49.0	54.8634	83.1041	1989	22568.472530	vcg05	2	2

16.1.9. Манхэттенское расстояние

Манхэттенское расстояние – расстояние между двумя точками, которое равно сумме модулей разностей их координат. У него очень много имен: расстояние городских кварталов, метрика такси, прямоугольная метрика.

```
# пишем функцию, вычисляющую манхэттенское расстояние
def manhattan_distance(lat1, lng1, lat2, lng2):
    a = np.abs(lat2 - lat1)
    b = np.abs(lng1 - lng2)
    return a + b

# создаем переменную - манхэттенское расстояние
data['manhattan_distance'] = manhattan_distance(
    data['Lat'],
    data['Long'],
    city_center_coordinates[0],
    city_center_coordinates[1])
data.head()
```

Street	House_Number	Metro	Metro_m	Space_Total	Lat	Long	Date_Postr	dist_center	geohash	match	match2	manhattan_distance
Маяковского	24	Речной вокзал	11260	30.5	54.9738	83.0860	1971	12242.320295	vcg0d	2	2	0.2191
Звездная	14	Речной вокзал	12130	28.9	54.9551	83.0681	1974	12837.513174	vcg0d	2	2	0.2199
Лермонтова	36	Сибирская	438	31.9	55.0446	82.9267	1973	610.036608	vfcfp	5	4	0.0110
Достоевского	20	Сибирская	312	44.7	55.0456	82.9192	1964	1081.876736	vfcfn	4	3	0.0195
Демакова	12	Речной вокзал	23340	49.0	54.8634	83.1041	1989	22568.472530	vcg05	2	2	0.3476

16.1.10. Восстановление координат

по адресу/восстановление адреса по координатам

Часто данные о недвижимости содержат адреса, а вот координаты отсутствуют. Загрузим такие данные.

```
# загружаем данные с отсутствующими координатами
data = pd.read_csv('Data/Flats_miss_coords.csv', sep=';')
data
```

Rooms_Number	Object_Type	Settlement	District	Street	House_Number	Metro	Metro_m	Space_Total	Date_Postr	
0	1	Хрущевка	Новосибирск	Первомайский	Маяковского	24	Речной вокзал	11260	30.5	1971
1	1	Типовая	Новосибирск	Первомайский	Звездная	14	Речной вокзал	12130	28.9	1974
2	1	Типовая	Новосибирск	Центральный	Лермонтова	36	Сибирская	438	31.9	1973
3	2	Хрущевка	Новосибирск	Центральный	Достоевского	20	Сибирская	312	44.7	1964
4	2	Улучшенной планировки	Новосибирск	Советский	Демакова	12	Речной вокзал	23340	49.0	1989
5	3	Улучшенной планировки	Новосибирск	Советский	Демакова	6	Речной вокзал	23490	65.0	1997
6	3	Улучшенной планировки	Новосибирск	Первомайский	Твардовского	22	Речной вокзал	19300	48.3	2014
7	1	Хрущевка	Новосибирск	Первомайский	Твардовского	12	Речной вокзал	17910	29.3	1977

Видим, что у нас нет информации о широте и долготе, однако с помощью API Google и Яндекс мы можем легко ее получить.

Сначала создадим переменную *Address*, которую будем использовать для получения координат.

```
# создадим переменную, которую будем использовать
# для получения координат
data['Address'] = (data['House_Number'].astype(str) + ', ' +
                    data['Street'] + ', ' +
                    data['Settlement'])
data['Address'].head()
```

0	24, Маяковского, Новосибирск
1	14, Звездная, Новосибирск
2	36, Лермонтова, Новосибирск
3	20, Достоевского, Новосибирск
4	12, Демакова, Новосибирск

Name: Address, dtype: object

Воспользуемся библиотекой geopy, она предлагает классы для различных сервисов геокодирования, в частности класс GoogleV3 для Google Geocoding API (V3), класс Nominatim для OpenStreetMap Nominatim, класс Yandex для API Яндекс.Карт. Обратите внимание, для того чтобы воспользоваться классами GoogleV3 и Yandex, вам потребуется получить API-ключ. Подробную информацию о том, как получить API-ключ для Google Geocoding API (V3), можно найти здесь: <https://developers.google.com/maps/documentation/geolocation/get-api-key>. Давайте импортируем необходимые нам классы библиотеки geopy.

```
# импортируем необходимые классы
from geopy.geocoders import GoogleV3, Nominatim, Yandex
```

Напишем функцию *set_coords()*, которая будет возвращать координаты с помощью различных сервисов геокодирования.

```
# пишем функцию, вычисляющую координаты
def set_coords(df, provider, col_addr, col_lat, col_lon):
    loc_lat = []
    loc_lon = []

    if provider == 'Google_API':
        g = GoogleV3(api_key=' ваш-арі-ключ ')
    if provider == 'Nominatim':
        g = Nominatim(user_agent='user')
    if provider == 'Yandex':
        g = Yandex(api_key=' ваш-арі-ключ ')
```

```

for address in df[col_addr]:
    try:
        inputAddress = address
        location = g.geocode(inputAddress, timeout=15)
        loc_lat.append(location.latitude)
        loc_lon.append(location.longitude)
    except:
        loc_lat.append(np.NaN)
        loc_lon.append(np.NaN)

df[col_lat] = loc_lat
df[col_lon] = loc_lon

```

Применяем нашу функцию.

```

# получаем координаты с помощью разных сервисов геокодирования
set_coords(data, 'Google_API', 'Address',
           'Lat_GoogleV3', 'Long_GoogleV3')
set_coords(data, 'Nominatim', 'Address',
           'Lat_Nominatim', 'Long_Nominatim')
set_coords(data, 'Yandex', 'Address',
           'Lat_Yandex', 'Long_Yandex')

```

Смотрим результаты.

Address	Lat_GoogleV3	Long_GoogleV3	Lat_Nominatim	Long_Nominatim	Lat_Yandex	Long_Yandex
24, Маяковского, Новосибирск	54.973891	83.085882	54.973822	83.086091	54.973879	83.085962
14, Звездная, Новосибирск	54.955138	83.068255	NaN	NaN	54.955096	83.068122
36, Лермонтова, Новосибирск	55.044545	82.926788	55.044530	82.926635	55.044552	82.926727
20, Достоевского, Новосибирск	55.045899	82.919126	55.045586	82.919197	55.045573	82.919226
12, Демакова, Новосибирск	54.863429	83.103899	54.863477	83.104130	54.863451	83.104234
6, Демакова, Новосибирск	54.862378	83.101770	54.862718	83.100508	54.862440	83.101728
22, Твардовского, Новосибирск	54.936667	83.126886	54.933799	83.125606	54.933796	83.125515
12, Твардовского, Новосибирск	54.938324	83.126128	54.938346	83.126072	54.938348	83.126054

Применяем нашу функцию.

```

# получаем координаты с помощью разных сервисов геокодирования
set_coords(data, 'Google_API', 'Address',
           'Lat_GoogleV3', 'Long_GoogleV3')
set_coords(data, 'Nominatim', 'Address',
           'Lat_Nominatim', 'Long_Nominatim')
set_coords(data, 'Yandex', 'Address',
           'Lat_Yandex', 'Long_Yandex')

```

Смотрим наши данные.

Address	Lat_GoogleV3	Long_GoogleV3	Lat_Nominatim	Long_Nominatim	Lat_Yandex	Long_Yandex
24, Маяковского, Новосибирск	54.973891	83.085882	54.973822	83.086091	54.973879	83.085962
14, Звездная, Новосибирск	54.955138	83.068255	Nan	Nan	54.955096	83.068122
36, Лермонтова, Новосибирск	55.044545	82.926788	55.044530	82.926635	55.044552	82.926727
20, Достоевского, Новосибирск	55.045899	82.919126	55.045586	82.919197	55.045573	82.919226
12, Демакова, Новосибирск	54.863429	83.103899	54.863477	83.104130	54.863451	83.104234
6, Демакова, Новосибирск	54.862378	83.101770	54.862718	83.100508	54.862440	83.101728
22, Твардовского, Новосибирск	54.936667	83.126886	54.933799	83.125606	54.933796	83.125515
12, Твардовского, Новосибирск	54.938324	83.126128	54.938346	83.126072	54.938348	83.126054

Видим, что при использовании сервиса OpenStreetMap Nominatim по одному наблюдению координаты получить не удалось. Давайте заменим пропуски с помощью сервиса Google Geocoding API (V3). Когда у нас уже есть столбцы со значениями широты и долготы, но в некоторых наблюдениях есть пропуски, применять нашу функцию уже нецелесообразно, потому что она будет вычислять координаты по всем наблюдениям. Давайте модифицируем нашу функцию так, чтобы она вычисляла координаты только для тех наблюдений, в которых значения широты и долготы пропущены.

```
# пишем функцию, вычисляющую координаты в случае пропусков
def find_miss_coords(df, provider, col_addr, col_lat, col_lon):
    missing = df[col_lat].isnull() | df[col_lon].isnull()
    loc_lat = []
    loc_lon = []

    if provider == 'Google_API':
        g = GoogleV3(api_key=' ваш-арі-ключ ')
    if provider == 'Nominatim':
        g = Nominatim(user_agent='artemgruzdev')
    if provider == 'Yandex':
        g = Yandex(api_key=' ваш-арі-ключ ')

    for address in df[col_addr][missing]:
        try:
            inputAddress = address
            location = g.geocode(inputAddress, timeout=15)
            loc_lat.append(location.latitude)
            loc_lon.append(location.longitude)
        except:
            loc_lat.append(np.NaN)
            loc_lon.append(np.NaN)
    df.loc[missing, col_lat] = loc_lat
    df.loc[missing, col_lon] = loc_lon
```

Применяем нашу функцию к переменным *Lat_Nominatim* и *Long_Nominatim*, в которых есть пропуски.

```
# получаем координаты для пропусков
find_miss_coords(data, 'Google_API', 'Address',
                  'Lat_Nominatim', 'Long_Nominatim')
# смотрим данные
data
```

Address	Lat_GoogleV3	Long_GoogleV3	Lat_Nominatim	Long_Nominatim	Lat_Yandex	Long_Yandex
24, Маяковского, Новосибирск	54.973891	83.085882	54.973822	83.086091	54.973879	83.085962
14, Звездная, Новосибирск	54.955138	83.068255	54.955138	83.068255	54.955096	83.068122
36, Лермонтова, Новосибирск	55.044545	82.926788	55.044530	82.926635	55.044552	82.926727
20, Достоевского, Новосибирск	55.045899	82.919126	55.045586	82.919197	55.045573	82.919226
12, Демакова, Новосибирск	54.863429	83.103899	54.863477	83.104130	54.863451	83.104234
6, Демакова, Новосибирск	54.862378	83.101770	54.862718	83.100508	54.862440	83.101728
22, Твардовского, Новосибирск	54.936667	83.126886	54.933799	83.125606	54.933796	83.125515
12, Твардовского, Новосибирск	54.938324	83.126128	54.938346	83.126072	54.938348	83.126054

А теперь выполним обратную операцию – восстановим адрес по координатам.

```
# создаем экземпляр класса Nominatim
geolocator = Nominatim(user_agent='artemgruzdev')

# пишем функцию восстановления адреса по координатам
def extract_address_from_coords(row):
    coords = f"[{row['Lat_Yandex']}, {row['Long_Yandex']}]"
    location = geolocator.reverse(coords, exactly_one=True)
    address = location.raw['address']

    country = address.get('country', np.nan)
    state = address.get('state', np.nan)
    county = address.get('county', np.nan)
    region = address.get('region', np.nan)
    municipality = address.get('municipality', np.nan)
    city_district = address.get('city_district', np.nan)
    postcode = address.get('postcode', np.nan)

    row['country'] = country
    row['state'] = state
    row['county'] = county
    row['region'] = region
    row['municipality'] = municipality
    row['city_district'] = city_district
    row['postcode'] = postcode
    return row
```

применяем нашу функцию

```
data = data.apply(extract_address_from_coords, axis=1)
data
```

Lat_Yandex	Long_Yandex	country	state	county	region	municipality	city_district	postcode
54.973879	83.085962	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Первомайский район	630046
54.955102	83.068131	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Первомайский район	630000
55.044552	82.926736	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Центральный район	630091
55.045573	82.919235	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Центральный район	630091
54.863451	83.104234	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Советский район	630128
54.862446	83.101737	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Советский район	630128
54.933796	83.125515	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Первомайский район	630068
54.938353	83.126063	Россия	Новосибирская область	городской округ Новосибирск	Сибирский федеральный округ	NaN	Первомайский район	630068

16.1.11. Макроэкономические признаки

В задачах прогнозирования стоимости квадратного метра жилья важными переменными будут курс доллара (в России после кризисов 1998 и 2008 годов стоимость квартиры в рублях оставалась прежней, а стоимость квартиры в долларах снижалась), курс евро, стоимость цен на нефть, уровень инфляции, которые влияют на покупательскую способность. Также можно использовать индексы потребительских настроений, вычисляемые администрациями некоторых областей, или индекс потребительских настроений GfK Rus (берется по России в целом и по регионам: Северо-Западный, Центральный, Южный, Поволжье, Урал, Сибирь, Дальний Восток, Москва, Северный Кавказ). Индекс потребительских настроений GfK Rus вычисляется как разница между положительными и отрицательными ответами в процентах с прибавлением 100. Диапазон изменения индексов 0–200, величины выше 100 – преобладание положительных ответов, ниже – отрицательных. Кроме того, нужно учитывать коэффициент доступности жилья (*housing price to income ratio*), он показывает, сколько лет нужно семье копить свои доходы, чтобы приобрести жилье. Коэффициент определяется как отношение медианной стоимости жилья к медианному размеру дохода семьи за год.

В задачах кредитного scoringа нужно учитывать прокси-метрики уровня безработицы в той сфере занятости, в которой работает заемщик (например, количество дней, в течение которых уволенный сотрудник находит работу в данной сфере занятости, количество банкротств предприятий в данной сфере занятости). При построении scoringовых моделей для ипотеки целесообразно учитывать курс доллара, курс евро, стоимость цен на нефть, коэффициент доступности жилья.

16.1.12. Агрегаты (на основе банковских транзакций)

Часто данные записаны в формате «один клиент – несколько наблюдений». Как правило, речь идет о транзакционных данных, когда у вас, например, есть суммы снятия и суммы пополнения в разные даты по одному и тому же клиенту. Такие данные часто приводят к агрегированному формату «один клиент – одно наблюдение». Для этого с помощью агрегирующих функций вычисляют различные статистики по клиенту – общую сумму снятия наличных, среднюю сумму снятия наличных, наиболее часто встречающийся MCC-код транзакции, общее количество транзакций, уникальное количество транзакций.

Давайте загрузим игрушечный набор данных (сформирован на основе задачи Rosbank ML Competition).

```
# загружаем и смотрим данные
data = pd.read_csv('Data/rosbank.csv', sep=';')
data.head(12)
```

	PERIOD	cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	01.10.2017	0	5200		NaN	810 21OCT17:00:00:00	5023.0	POS	0
1	01.10.2017	0	6011		NaN	810 12OCT17:12:24:07	20000.0	DEPOSIT	0
2	01.12.2017	0	5921		NaN	810 05DEC17:00:00:00	767.0	POS	0
3	01.10.2017	0	5411		NaN	810 21OCT17:00:00:00	2031.0	POS	0
4	01.10.2017	0	6012		NaN	810 24OCT17:13:14:24	36562.0	C2C_OUT	0
5	01.10.2017	1	5814		NaN	810 16OCT17:00:00:00	380.0	POS	0
6	01.10.2017	1	5814		NaN	810 10OCT17:00:00:00	378.0	POS	0
7	01.10.2017	1	5814		NaN	810 16OCT17:00:00:00	199.0	POS	0
8	01.10.2017	1	5814		NaN	810 11OCT17:00:00:00	400.0	POS	0
9	01.07.2017	1	5411		NaN	810 26JUL17:00:00:00	598.0	POS	0
10	01.06.2017	5	5944		NaN	810 18JUN17:00:00:00	3719.0	POS	1
11	01.06.2017	5	6012		NaN	810 14JUN17:00:00:00	10000.0	C2C_OUT	1

Набор включает следующие переменные:

- *PERIOD* – месяц транзакции;
- *cl_id* – id клиента;
- *MCC* – код категории продавца;
- *channel_type* – канал привлечения клиента;
- *currency* – валюта;
- *TRDATETIME* – дата/время транзакции;
- *amount* – сумма транзакции;
- *trx_category* – вид транзакции:
 - ◆ POS – оплата через POS-терминал;
 - ◆ C2C_OUT – перевод на карту (исходящий платёж);
 - ◆ C2C_IN – перевод на карту (входящий платёж);
 - ◆ DEPOSIT – пополнение карты в банкомате;
 - ◆ WD_ATM_PARTNER – снятие наличных в банкоматах-партнерах;
 - ◆ WD_ATM_ROS – снятие наличных в банкоматах Росбанка;
 - ◆ BACK_TRX – возврат средств при возврате покупки;
 - ◆ WD_ATM_OTHER – снятие в других банкоматах;
 - ◆ CAT – операции в банкоматах;
 - ◆ CASH_ADV – снятие карты в кассе банка;

- *target_flag* – зависимая переменная (0 – пользуется услугами, 1 – перестал пользоваться).

Каждому клиенту соответствует несколько наблюдений. Переменным *TRDATETIME* и *PERIOD* присвоим тип *datetime*, пропуски в переменной заполним *channel_type* отдельной категорией. Затем отсортируем наблюдения по идентификатору клиента и дате/времени транзакции и обновим индекс.

```
# переводим переменные TRDATETIME и PERIOD в тип datetime
data['TRDATETIME'] = pd.to_datetime(data['TRDATETIME'], format='%d.%b%Y:%X')
data['PERIOD'] = pd.to_datetime(data['PERIOD'], format='%d.%m.%Y')
# заполняем пропуски в channel_type отдельной категорией
data['channel_type'] = data['channel_type'].fillna('other')

# сортируем наблюдения по клиентам и дате транзакции
data = data.sort_values(['cl_id', 'TRDATETIME'],
                       ascending=(True, True))

# обновляем индекс
data.reset_index(drop=True, inplace=True)
data
```

	PERIOD	cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	2017-10-01	0	6011	other	810	2017-10-12 12:24:07	20000.00	DEPOSIT	0
1	2017-10-01	0	5200	other	810	2017-10-21 00:00:00	5023.00	POS	0
2	2017-10-01	0	5411	other	810	2017-10-21 00:00:00	2031.00	POS	0
3	2017-10-01	0	6012	other	810	2017-10-24 13:14:24	36562.00	C2C_OUT	0
4	2017-12-01	0	5921	other	810	2017-12-05 00:00:00	767.00	POS	0
5	2017-07-01	1	5411	other	810	2017-07-26 00:00:00	598.00	POS	0
6	2017-10-01	1	5814	other	810	2017-10-10 00:00:00	378.00	POS	0
7	2017-10-01	1	5814	other	810	2017-10-11 00:00:00	400.00	POS	0
8	2017-10-01	1	5814	other	810	2017-10-16 00:00:00	380.00	POS	0
9	2017-10-01	1	5814	other	810	2017-10-16 00:00:00	199.00	POS	0
10	2017-04-01	5	5621	other	810	2017-04-06 00:00:00	1399.00	POS	1
11	2017-04-01	5	5499	other	810	2017-04-25 00:00:00	1387.61	POS	1
12	2017-06-01	5	6012	other	810	2017-06-14 00:00:00	10000.00	C2C_OUT	1
13	2017-06-01	5	5691	other	810	2017-06-17 00:00:00	3190.00	POS	1
14	2017-06-01	5	5944	other	810	2017-06-18 00:00:00	3719.00	POS	1

Сначала мы вычислим агрегаты, не приводя исходный набор к формату «один клиент – одно наблюдение». Мы просто вычислим агрегированные значения и запишем в исходный набор, чтобы убедиться, что под капотом все происходит так, как нам нужно. Сделать это можно с помощью следующего синтаксиса:

```
data['агрегат'] = data.groupby(
    'группирующий столбец')[['агрегируемый столбец']].transform(
    lambda x: x.агрегирующая функция())

# вычисляем сумму транзакций для каждого клиента
data['amount_sum_cl_id'] = data.groupby(
    'cl_id')[['amount']].transform(lambda x: x.sum())
# вычисляем сумму транзакций по каждому периоду для каждого клиента
```

```

data['amount_sum_cl_id_period'] = data.groupby(
    ['cl_id', 'PERIOD'])['amount'].transform(lambda x: x.sum())
# вычисляем общее количество типов
# транзакций для каждого клиента
data['total_number_trx_cat_cl_id'] = data.groupby(
    'cl_id')['trx_category'].transform(lambda x: x.count())
# вычисляем уникальное количество типов
# транзакций для каждого клиента
data['uniq_number_trx_cat_cl_id'] = data.groupby(
    'cl_id')['trx_category'].transform(lambda x: x.unique())
# вычисляем самый часто встречающийся тип транзакции
# для каждого клиента
data['trx_category_mode_cl_id'] = data.groupby(
    'cl_id')['trx_category'].transform(lambda x: x.value_counts().index[0])
# вычисляем разницу в днях между самой ранней и самой поздней
# датами транзакции для каждого клиента
data['diff_days_cl_id'] = data.groupby(
    'cl_id')['TRDATETIME'].transform(lambda x: x.max() - x.min()).dt.days
# вычисляем разницу в днях между конкретной и самой ранней
# датами транзакции для каждого клиента
data['days_from_first_transaction'] = data.groupby(
    'cl_id')['TRDATETIME'].transform(lambda x: x - x.min()).dt.days
data

```

amount_sum_cl_id	amount_sum_cl_id_period	total_num_trx_cat_cl_id	uniq_num_trx_cat_cl_id	trx_category_mode_cl_id	diff_days_cl_id	days_from_first_transaction
64383.00	63616.00	5	3	POS	53	0
64383.00	63616.00	5	3	POS	53	8
64383.00	63616.00	5	3	POS	53	8
64383.00	63616.00	5	3	POS	53	12
64383.00	767.00	5	3	POS	53	53

1955.00	598.00	5	1	POS	82	0
1955.00	1357.00	5	1	POS	82	76
1955.00	1357.00	5	1	POS	82	77
1955.00	1357.00	5	1	POS	82	82
1955.00	1357.00	5	1	POS	82	82
19695.61	2786.61	5	2	POS	73	0
19695.61	2786.61	5	2	POS	73	19
19695.61	16909.00	5	2	POS	73	69
19695.61	16909.00	5	2	POS	73	72
19695.61	16909.00	5	2	POS	73	73

	PERIOD	cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	2017-10-01	0	6011	other	810	2017-10-12 12:24:07	20000.00	DEPOSIT	0
1	2017-10-01	0	5200	other	810	2017-10-21 00:00:00	5023.00	POS	0
2	2017-10-01	0	5411	other	810	2017-10-21 00:00:00	2031.00	POS	0
3	2017-10-01	0	6012	other	810	2017-10-24 13:14:24	36562.00	C2C_OUT	0
4	2017-12-01	0	5921	other	810	2017-12-05 00:00:00	767.00	POS	0

Теперь выполним агрегацию так, чтобы привести набор переменных к формату «один клиент – одно наблюдение». Сделать это можно с помощью следующего синтаксиса:

```
aggdata = df.groupby('группирующий столбец').agg(
    {'агрегируемый столбец': 'агрегирующая функция'})

# можно агрегировать с помощью
# методов .groupby() и .agg()

aggregations = [
    'amount': 'sum',
    'trx_category': lambda x: x.value_counts().index[0],
    'target_flag': 'first'
}
agg_data = data.groupby('cl_id').agg(aggregations)

# смотрим агрегированный набор
agg_data
```

	amount	trx_category	target_flag
cl_id			
0	64383.00	POS	0
1	1955.00	POS	0
5	19695.61	POS	1

Кроме того, мы можем написать собственную функцию, выполняющую агрегацию.

```
# пишем функцию агрегации
def build_features(data):
    aggregated = data.groupby('cl_id')[['channel_type']].first()
    ids = aggregated.index
    aggregated['cl_id'] = ids
    # вычислим средний временной промежуток между транзакциями
    aggregated['mean_diff'] = data.groupby('cl_id')['TRDATETIME'].apply(
        lambda x: np.mean(np.abs(x.diff()))).dt.days
    # вычисляем самую позднюю дату транзакции во всем наборе
    global_max_date = data['TRDATETIME'].max()
    # вычисляем самую позднюю дату транзакции по каждому периоду
    # для каждого клиента
    data['max_date'] = data.groupby(
        ['cl_id', 'PERIOD'])['TRDATETIME'].transform(lambda x: x.max())
    # вычисляем разницу между самой поздней датой транзакции во всем наборе
    # и самой поздней датой транзакции по каждому периоду для каждого клиента
    data['global_diff'] = (global_max_date - data['max_date']).dt.days
    # берем среднюю разницу между самой поздней датой транзакции во всем
    # наборе и самой поздней датой транзакции для каждого клиента
    aggregated['global_diff_mean'] = data.groupby(
        'cl_id')['global_diff'].mean()
    # вычисляем сумму транзакций по каждому периоду для каждого клиента
    data['amount_sum_cl_id_period'] = data.groupby(
        ['cl_id', 'PERIOD'])['amount'].transform(lambda x: x.sum())
    # вычисляем для каждого клиента среднюю абсолютную разность
    # на основе сумм транзакций по каждому периоду
    aggregated['mad_of_amount_sum_cl_id_period'] = data.groupby(
        'cl_id')['amount_sum_cl_id_period'].apply(lambda x: x.mad())
    # вычисляем общее количество транзакций для каждого клиента
    aggregated['total_number_transact_cl_id'] = data.groupby(
        'cl_id')['TRDATETIME'].apply(lambda x: x.count())
```

```

# вычисляем уникальное количество MCC-кодов для каждого клиента
aggregated['uniq_number_MCC_cl_id'] = data.groupby(
    'cl_id')['MCC'].apply(lambda x: x.unique())
# вычисляем уникальное количество MCC-кодов по каждому периоду
# для каждого клиента
data['uniq_number_MCC_cl_id_period'] = data.groupby(
    ['cl_id', 'PERIOD'])['MCC'].transform(lambda x: x.unique())
# вычисляем для каждого клиента уникальное количество
# MCC-кодов, усредненное по периодам
aggregated['uniq_number_MCC_cl_id_period'] = data.groupby(
    'cl_id')['uniq_number_MCC_cl_id_period'].apply(lambda x: x.mean())
# вычисляем самый часто встречающийся тип транзакции
# для каждого клиента
aggregated['trx_category_cl_id_mode'] = data.groupby(
    'cl_id')['trx_category'].apply(lambda x: x.value_counts().index[0])
# вычисляем разницу в днях между самой ранней и самой поздней
# датами транзакции для каждого клиента
aggregated['diff_days_cl_id'] = data.groupby(
    'cl_id')['TRDATETIME'].apply(lambda x: x.max() - x.min()).dt.days
# вычисляем разницу в днях между самой ранней и самой поздней
# датами POS-транзакции для каждого клиента
aggregated['diff_days_POS'] = data[data['trx_category'] == 'POS'].groupby(
    'cl_id')['TRDATETIME'].apply(lambda x: x.max() - x.min()).dt.days
# вычисляем разницу в днях между самой ранней и самой поздней
# датами транзакции по каждому периоду для каждого клиента
data['diff_days_cl_id_period'] = data.groupby(
    ['cl_id', 'PERIOD'])['TRDATETIME'].transform(
        lambda x: x.max() - x.min()).dt.days
# вычисляем для каждого клиента разницу в днях между самой ранней
# и самой поздней датами транзакции, усредненную по периодам
aggregated['mean_of_diff_days_cl_id_period'] = data.groupby(
    'cl_id')['diff_days_cl_id_period'].apply(lambda x: x.mean())
# вычисляем разницу в днях между конкретной и самой ранней
# датами транзакции для каждого клиента
data['days_from_first_transaction'] = data.groupby(
    'cl_id')['TRDATETIME'].transform(lambda x: x - x.min()).dt.days
# вычисляем среднюю разницу в днях между конкретной и самой ранней
# датами транзакции для каждого клиента
aggregated['mean_of_days_from_first_transaction'] = data.groupby(
    'cl_id')['days_from_first_transaction'].apply(lambda x: x.mean())
# вычисляем среднюю абсолютную разность для
# вышесозданной переменной по каждому клиенту
aggregated['mad_of_days_from_first_transaction'] = data.groupby(
    'cl_id')['days_from_first_transaction'].apply(lambda x: x.mad())
# создаем тип транзакции, пополняет она или, наоборот, сокращает счет
trx_lst = ['DEPOSIT', 'C2C_IN', 'BACK_TRX']
data['trx_type'] = data.apply(
    lambda x: 1 if x['trx_category'] in trx_lst else -1, axis=1)
# вычисляем сумму каждой транзакции с учетом знака
# (пополнения или снятия) по каждому клиенту
data['amount_signed'] = data['amount'] * data['trx_type']
# вычисляем среднюю сумму каждой транзакции с учетом знака
# (пополнения или снятия) по каждому клиенту
aggregated['amount_signed'] = data.groupby(
    'cl_id')['amount_signed'].apply(lambda x: x.mean())
# вычисляем сумму всех транзакций по каждому клиенту
aggregated['amount_sum'] = data.groupby(
    'cl_id')['amount'].apply(lambda x: x.sum())

```

```

# вычисляем минимальную сумму транзакции по каждому клиенту
aggregated['amount_min'] = data.groupby(
    'cl_id')['amount'].apply(lambda x: x.min())
# вычисляем сумму всех транзакций с учетом знака
# ( пополнения или снятия ) по каждому клиенту
aggregated['amount_signed_sum'] = data.groupby(
    'cl_id')['amount_signed'].apply(lambda x: x.sum())
# вычисляем сумму транзакций снятия по каждому клиенту
aggregated['amount_snyatie'] = (aggregated['amount_sum'] -
                                aggregated['amount_signed_sum'])
# вычисляем сумму транзакций пополнения по каждому клиенту
aggregated['amount_popolnenie'] = (aggregated['amount_sum'] -
                                    aggregated['amount_snyatie'])
# вычисляем уникальное количество периодов по каждому клиенту
aggregated['PERIOD_uniq_count'] = data.groupby('cl_id')['PERIOD'].nunique()
# вычисляем общее количество транзакций DEPOSIT по каждому клиенту
aggregated['count_DEPOSIT'] = data[data['trx_category'] == 'DEPOSIT'].groupby(
    'cl_id')['trx_category'].count()
# вычисляем общее количество транзакций POS по каждому клиенту
aggregated['count_POS'] = data[data['trx_category'] == 'POS'].groupby(
    'cl_id')['trx_category'].count()
# вычисляем сумму транзакций за первые 60 дней по каждому клиенту
sum_amount_first_60_days = data.sort_values('TRDATETIME').groupby('cl_id').apply(
    lambda x: x['amount'][((x['TRDATETIME'] - x['TRDATETIME'].values[0]) /
                           np.timedelta64(1, 'D')) <= 60]).sum()
# вычисляем сумму транзакций за первые 180 дней по каждому клиенту
sum_amount_first_180_days = data.sort_values('TRDATETIME').groupby('cl_id').apply(
    lambda x: x['amount'][((x['TRDATETIME'] - x['TRDATETIME'].values[0]) /
                           np.timedelta64(1, 'D')) <= 180]).sum()
# вычисляем отношение суммы транзакций за первые 60 дней к
# сумме транзакций за первые 180 дней по каждому клиенту
aggregated['ratio_first_60_180_days'] = (sum_amount_first_60_days /
                                         sum_amount_first_180_days)
# вычисляем сумму транзакций за последние 30 дней по каждому клиенту
sum_amount_last_30_days = data.sort_values('TRDATETIME').groupby('cl_id').apply(
    lambda x: x['amount'][((x['TRDATETIME'].values[-1] - x['TRDATETIME']) /
                           np.timedelta64(1, 'D')) <= 30]).sum()
# вычисляем сумму транзакций за последние 60 дней по каждому клиенту
sum_amount_last_60_days = data.sort_values('TRDATETIME').groupby('cl_id').apply(
    lambda x: x['amount'][((x['TRDATETIME'].values[-1] - x['TRDATETIME']) /
                           np.timedelta64(1, 'D')) <= 60]).sum()
# вычисляем отношение суммы транзакций за последние 30 дней к
# сумме транзакций за последние 60 дней по каждому клиенту
aggregated['ratio_last_30_60_days'] = (sum_amount_last_30_days /
                                         sum_amount_last_60_days)
# вычисляем сумму транзакций POS по каждому клиенту
aggregated['amount_sum_POS'] = data[data['trx_category'] == 'POS'].groupby(
    'cl_id')['amount'].sum()
# вычисляем последний тип транзакции по каждому клиенту
aggregated['last_trx_cat_cl_id'] = data.sort_values(
    'TRDATETIME').groupby('cl_id')['trx_category'].last()
# вычисляем общее количество транзакций DEPOSIT
# по каждому периоду для каждого клиента
data['count_DEPOSIT_cl_id_period'] = data[data['trx_category'] == 'DEPOSIT'].groupby(
    ['cl_id', 'PERIOD'])['trx_category'].transform(lambda x: x.count())
# вычисляем среднюю абсолютную разность для вышесозданной
# переменной по каждому клиенту
aggregated['mad_of_count_POS_cl_id_period'] = data.groupby(
    'cl_id')['count_DEPOSIT_cl_id_period'].apply(lambda x: x.mad())

```

```

# вычисляем разность между суммами транзакций в самую раннюю
# и самую позднюю даты для каждого клиента
aggregated['amount_diff_min_max_date'] = data.sort_values(
    'TRDATETIME').groupby('cl_id').apply(
        lambda x: x['amount'][x['TRDATETIME'] == x['TRDATETIME'].values[0]].sum() -
        x['amount'][x['TRDATETIME'] == x['TRDATETIME'].values[-1]].sum())
# добавляем зависимую переменную
aggregated['target_flag'] = data.groupby('cl_id')['target_flag'].first()
# удаляем индекс cl_id
aggregated.reset_index('cl_id', drop=True, inplace=True)
# имплементируем пропуски нулями
return aggregated.fillna(0)

```

Давайте применим нашу функцию и посмотрим результаты.

```

# увеличиваем количество столбцов
pd.set_option('display.max_columns', 60)
# выполняем агрегирование
agg_data2 = build_features(data)

```

	channel_type	cl_id	mean_diff	global_diff_mean	mad_of_amount_sum_cl_id_period	total_number_transact_cl_id	uniq_number_MCC_cl_id
0	other	0	13	32.8	20111.6800	5	5
1	other	1	20	66.4	242.8800	5	2
2	other	5	18	191.6	6778.7472	5	5

Разберем, правильно ли были вычислены некоторые признаки. Начнем с признака *mean_diff*. Речь идет о среднем временном промежутке между транзакциями. Разберем на примере клиента с индексом 0.

	channel_type		cl_id	mean_diff	global_diff_mean	mad_of_amount_sum_cl_id_period	total_number_transact_cl_id	uniq_number_MCC_cl_id		
	PERIOD		cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	2017-10-01	0	6011	other	810	2017-10-12 12:24:07	20000.00	DEPOSIT	0	
1	2017-10-01	0	5200	other	810	2017-10-21 00:00:00	5023.00	POS	0	
2	2017-10-01	0	5411	other	810	2017-10-21 00:00:00	0	POS	0	
3	2017-10-01	0	6012	other	810	2017-10-24 13:14:24	36562.00	C2C_OUT	0	
4	2017-12-01	0	5921	other	810	2017-12-05 00:00:00	41	POS	0	

Рассмотрим признак *global_diff*. Речь идет о средней разнице между самой поздней датой транзакции во всем наборе и самой поздней датой транзакции по каждому периоду для каждого клиента. Разберем на примере клиента с индексом 0.

	channel_type		cl_id	mean_diff	global_diff_mean	mad_of_amount_sum_cl_id_period	total_number_transact_cl_id	uniq_number_MCC_cl_id		
	PERIOD		cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	2017-10-01	0	6011	other	810	2017-10-12 12:24:07	2017-10-24 00:00:00	DEPOSIT	0	
1	2017-10-01	0	5200	other	810	2017-10-21 00:00:00	2017-10-24 00:00:00	POS	0	
2	2017-10-01	0	5411	other	810	2017-10-21 00:00:00	2017-10-24 00:00:00	POS	0	
3	2017-10-01	0	6012	other	810	2017-10-24 13:14:24	2017-10-24 13:14:24	C2C_OUT	0	
4	2017-12-01	0	5921	other	810	2017-12-05 00:00:00	2017-12-05 00:00:00	POS	0	

166 ❖ План предварительной подготовки данных

Рассмотрим признаки `total_number_transact_cl_id` и `uniq_number_MCC_cl_id`. Здесь мы вычисляем общее количество транзакций для каждого клиента и уникальное количество MCC-кодов для каждого клиента соответственно. Разберем на примере клиента с индексом 1.

	channel_type	cl_id	mean_diff	global_diff_mean	mad_of_amount_sum_cl_id_period	total_number_transact_cl_id	uniq_number_MCC_cl_id	
0	other	0	13	32.8	20111.6800	5	5	
1	other	1	20	66.4	242.8800	5	2	
2	other	5	18	191.6	6778.7472	5	5	
PERIOD	cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
5	2017-07-01	1	5411	other	810 2017-07-26 00:00:00	598.00	POS	0
6	2017-10-01	1	5814	other	810 2017-10-10 00:00:00	378.00	POS	0
7	2017-10-01	1	5814	other	810 2017-10-11 00:00:00	400.00	POS	0
8	2017-10-01	1	5814	other	810 2017-10-16 00:00:00	380.00	POS	0
9	2017-10-01	1	5814	other	810 2017-10-16 00:00:00	199.00	POS	0

Рассмотрим признак `trx_category_cl_id_mode`. Мы вычислили самый часто встречающийся тип транзакции для каждого клиента. Разберем на примере клиента с индексом 0.

	uniq_number_MCC_cl_id_period	trx_category_cl_id_mode	diff_days_cl_id	diff_days_POS	mean_of_diff_days_cl_id_period	mean_of_days_from_first_transaction		
	3.4	POS	53	45	9.6	16.2		
	1.0	POS	82	82	4.8	63.4		
	2.6	POS	73	73	10.0	46.6		
PERIOD	cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	2017-10-01	0	6011	other	810 2017-10-12 12:24:07	20000.00	DEPOSIT	0
1	2017-10-01	0	5200	other	810 2017-10-21 00:00:00	5023.00	POS	0
2	2017-10-01	0	5411	other	810 2017-10-21 00:00:00	2031.00	POS	0
3	2017-10-01	0	6012	other	810 2017-10-24 13:14:24	36562.00	C2C_OUT	0
4	2017-12-01	0	5921	other	810 2017-12-05 00:00:00	767.00	POS	0

Рассмотрим признак `diff_days_cl_id`. Мы вычислили разницу в днях между самой ранней и самой поздней датами транзакции для каждого клиента.

	uniq_number_MCC_cl_id_period	trx_category_cl_id_mode	diff_days_cl_id	diff_days_POS	mean_of_diff_days_cl_id_period	mean_of_days_from_first_transaction		
	3.4	POS	53	45	9.6	16.2		
	1.0	POS	82	82	4.8	63.4		
	2.6	POS	73	73	10.0	46.6		
PERIOD	cl_id	MCC	channel_type	currency	TRDATETIME	amount	trx_category	target_flag
0	2017-10-01	0	6011	other	810 2017-10-12 12:24:07	20000.00	DEPOSIT	0
1	2017-10-01	0	5200	other	810 2017-10-21 00:00:00	5023.00	POS	0
2	2017-10-01	0	5411	other	810 2017-10-21 00:00:00	2031.00	POS	0
3	2017-10-01	0	6012	other	810 2017-10-24 13:14:24	36562.00	C2C_OUT	0
4	2017-12-01	0	5921	other	810 2017-12-05 00:00:00	767.00	POS	0

Обратите внимание, что агрегированные признаки мы можем создавать только после разбиения на обучающую и тестовую выборки и нужно всегда задавать себе вопрос, как я буду представлять признак в тестовой выборке, не обращаясь при этом к информации тестовой выборки. Далеко не все признаки мы можем использовать в модели. Некоторые признаки были созданы в исследовательских целях. Например, мы вычисляли разницу между самой поздней датой транзакции во всем наборе (глобальной самой поздней датой транзакции) и самой поздней датой транзакции по каждому периоду для каждого клиента, однако в реальности может отсутствовать информация о глобальной самой поздней дате транзакции. Мы вычисляли отношение суммы транзакций за первые 60 дней к сумме транзакций за первые 180 дней по каждому клиенту, но новые данные могут быть собраны за менее длительный период.

16.1.13. MCC-коды

При работе с транзакциями мы имеем дело с MCC-кодами, по ним мы можем определить назначение транзакции. Набор, с которым мы сейчас работали, как раз содержит MCC-коды.

```
# загружаем и смотрим данные, скачанные
# с сайта https://mcc-codes.ru/code
data = pd.read_csv('Data/mcc_codes.csv')
data.head()
```

MCC	Название	Описание
0 742	Ветеринарные услуги	Лицензированные специалисты в основном занимаю...
1 763	Сельскохозяйственные кооперативы	Ассоциации и кооперативы, которые предоставляю...
2 780	Услуги садоводства и ландшафтного дизайна	Ландшафтные архитекторы и другие поставщики ус...
3 1520	Генеральные подрядчики – жилое и коммерческое ...	Генеральные подрядчики, в основном занимающиеся...
4 1711	Генеральные подрядчики по вентиляции, теплоснабж...	Специальные торговые подрядчики, которые работают...

На основе столбцов *MCC* и *Группа* мы сейчас сформируем словарь и с помощью него создадим новую переменную *mcc_group* с названиями групп, соответствующими каждому MCC-коду.

```
# формируем датафрейм из столбцов MCC и Название
mcc_codes_table = data.loc[:, ['MCC', 'Название']]
# создаем словарь, ключами будут MCC-коды, а значениями - группы
mcc_map = dict(sorted(mcc_codes_table.values.tolist()))
data['mcc_group'] = data['MCC'].map(mcc_map)
# создаем новую переменную mcc_group, сопоставляя MCC-кодам группы
data['mcc_group'].head()
```

0	Ветеринарные услуги
1	Сельскохозяйственные кооперативы
2	Услуги садоводства и ландшафтного дизайна
3	Генеральные подрядчики – жилое и коммерческое ...
4	Генеральные подрядчики по вентиляции, теплоснабж...
Name:	mcc_group, dtype: object

16.1.14. Индикатор платежной дисциплины

В кредитном скоринге часто используют индикатор платежной дисциплины. Эта переменная фиксирует, насколько аккуратно заемщик гасит кредит.

Давайте загрузим данные с такой переменной.

загружаем данные

```
data = pd.read_csv('Data/Text_payment_discipline.csv', sep=';')
data.head(10)
```

	TEXT_PAYMENT_DISCIPLINE
0	CC0...
1	CCCCCCCCCCCCCCCC0X00000000000000000000XXXXXX
2	CCCCCCCCCCCCCCCC0000000000000000000000XXXXXX
3	CCCCCCCCCCCCCCCCCCCCX0X0XX00000X00X0XX0X0XX
4	CCCCCCCCCCCCCCCCCCCC00000000000000000000000000000000
5	XXXXX000X0XX00X00X0XX
6	0
7	CCCCX
8	XX00
9	XX00

Здесь используются следующие обозначения:

- 0 – своевременный платеж;
- 1 – просрочка 1..30 дней;
- 2 – просрочка 31..60 дней;
- 3 – просрочка 61..90 дней;
- 4 – просрочка 91..120 дней;
- 5 – просрочка 121+ дней, передан коллекторам, продан, списан;
- X – статус неизвестен;
- C – договор закрыт.

Даже из такой странной на первый взгляд переменной можно извлечь массу признаков.

```
# вычисляем количество своевременных платежей
data['count_0'] = data['TEXT_PAYMENT_DISCIPLINE'].str.count('0')
# вычисляем количество просрочек 1..30 дней
data['count_1'] = data['TEXT_PAYMENT_DISCIPLINE'].str.count('1')
# вычисляем количество просрочек 31..60 дней
data['count_2'] = data['TEXT_PAYMENT_DISCIPLINE'].str.count('2')
# вычисляем количество просрочек 61..90 дней
data['count_3'] = data['TEXT_PAYMENT_DISCIPLINE'].str.count('3')
# вычисляем количество просрочек 91..120 дней
data['count_4'] = data['TEXT_PAYMENT_DISCIPLINE'].str.count('4')
# вычисляем количество просрочек 121+ дней
```

```

data['count_5'] = data['TEXT_PAYMENT_DISCIPLINE'].str.count('5')
# вычисляем количество просрочек не больше 90 дней
data['count_delinq_up_to_90'] = np.sum(data.loc[:, 'count_1':'count_3'], axis=1)
# вычисляем общее количество просрочек
data['count_delinq'] = np.sum(data.loc[:, 'count_1':'count_5'], axis=1)
# вычислим длину строки
data['Length'] = data['TEXT_PAYMENT_DISCIPLINE'].map(len)
# вычисляем отношение количества своевременных платежей к длине строки
data['ratio'] = data['count_0'] / data['length']
# вычисляем отношение количества просрочек к длине строки
data['ratio2'] = data['count_delinq'] / data['length']
# получаем самый первый символ
data['first_letter'] = data['TEXT_PAYMENT_DISCIPLINE'].str[0]
# возвращаем позицию символа 0
data['position_0'] = data['TEXT_PAYMENT_DISCIPLINE'].str.find('0', 0)

# смотрим результаты
data.head()

```

AYMENT_DISCIPLINE	count_0	count_1	count_2	count_3	count_4	count_5	count_delinq_up_to_90	count_delinq	length	ratio	ratio2	first_letter	position_0
CCCCCCC000000000000	10	0	0	0	0	0	0	0	55	0.181818	0.0	C	45
10000000XXXXXX0XXXXX	22	0	0	0	0	0	0	0	47	0.468085	0.0	C	14
10000000XXXXXX0XXXXX	24	0	0	0	0	0	0	0	47	0.510638	0.0	C	14
00000X00X0XX000XX	13	0	0	0	0	0	0	0	44	0.295455	0.0	C	21
00000000000000000000	24	0	0	0	0	0	0	0	44	0.545455	0.0	C	20

16.1.15. Получение информации о курсах валют

Часто нам нужно получить информацию о курсе валюты на соответствующую дату. Например, информация о курсе доллара может быть полезной при прогнозировании стоимости жилья или оценке вероятности дефолта по ипотеке.

Загружаем наши данные.

```

# прочитываем данные
data = pd.read_csv('Data/Flats2.csv', sep=';')
data

```

District	Street	House_Number	Metro	Metro_m	Stor	Storeys	Wall	Space_Total	Space_Living	Cost_KV	Date_Postr	Date_Create
Заельцовский	Красный проспект	181	Заельцовская	2890	13	17	Кирпич	54.1	18	50831.792976	2010	21.04.2009
Заельцовский	Красный проспект	181	Заельцовская	2890	10	17	Кирпич	54.5	18	52000.000000	2010	17.08.2009
Центральный	Красный проспект	181	Заельцовская	2830	8	17	Кирпич	37.0	20	87837.837838	2008	27.09.2016
Центральный	Красный проспект	181	Заельцовская	2830	2	17	Кирпич	42.0	17	90238.095238	2008	28.07.2018
Центральный	Красный проспект	181	Заельцовская	2830	13	17	Кирпич	28.0	14	110714.285714	2008	23.06.2017
Заельцовский	Красный проспект	181	Заельцовская	2890	1	17	Кирпич	59.0	36	83050.847458	2010	10.03.2015

Переменную *Date_Create* переводим в формат даты.

```

# переводим переменную Date_Create в формат даты
data['Date_Create'] = pd.to_datetime(data['Date_Create'],
format='%d.%m.%Y')

```

Теперь воспользуемся классом `CurrencyConverter` библиотеки `CurrencyConverter` (<https://pypi.org/project/CurrencyConverter/>) и классом `date` модуля `datetime`.

```
# импортируем класс CurrencyConverter библиотеки currency_converter
from currency_converter import CurrencyConverter
# импортируем класс date модуля datetime
from datetime import date
```

Создаем экземпляр класса `CurrencyConverter`. У класса есть параметр `fallback_on_missing_rate`, который позволяет вычислить курс для даты, приходящейся на выходной день или праздник, для этого либо берется предшествующий рабочий день, ближайший к этой дате (если для параметра `fallback_on_missing_rate_method` задано значение `'last_known'`), либо выполняется линейная интерполяция (если для параметра `fallback_on_missing_rate_method` задано значение `'linear_interpolation'`).

```
# создаем экземпляр класса CurrencyConverter
c = CurrencyConverter(fallback_on_missing_rate=True)
```

Теперь пишем функцию, которая с помощью класса `CurrencyConverter` вычисляет курс валюты (в данном случае – курс доллара к рублю).

```
# пишем функцию, вычисляющую курс валюты
def get_rates(x):
    y = x.year
    m = x.month
    d = x.day
    rate = c.convert(1, 'USD', 'RUB', date=date(y, m, d))
    return rate
```

Применяем функцию, вычисляем переменную с курсом доллара на соответствующую дату.

```
# создаем переменную - курс доллара на текущую дату
data['USD_rate'] = data['Date_Create'].apply(get_rates)
data
```

Street	House_Number	Metro	Metro_m	Stor	Storeys	Wall	Space_Total	Space_Living	Cost_KV	Date_Postr	Date_Create	USD_rate
Красный проспект	181	Заельцовская	2890	13	17	Кирпич	54.1	18	50831.792976	2010	2009-04-21	34.163857
Красный проспект	181	Заельцовская	2890	10	17	Кирпич	54.5	18	52000.000000	2010	2009-08-17	32.319500
Красный проспект	181	Заельцовская	2830	8	17	Кирпич	37.0	20	87837.837838	2008	2016-09-27	63.927184
Красный проспект	181	Заельцовская	2830	2	17	Кирпич	42.0	17	90238.095238	2008	2018-07-28	62.826988
Красный проспект	181	Заельцовская	2830	13	17	Кирпич	28.0	14	110714.285714	2008	2017-06-23	59.668576
Красный проспект	181	Заельцовская	2890	1	17	Кирпич	59.0	36	83050.847458	2010	2015-10-03	65.957385

16.2. СТАТИЧЕСКОЕ КОНСТРУИРОВАНИЕ ПРИЗНАКОВ ИСХОДЯ ИЗ АЛГОРИТМА

Линейные модели (линейная регрессия, логистическая регрессия и другие), метод опорных векторов и нейронные сети не умеют работать с категориаль-

ными признаками напрямую, поэтому при работе с этими методами мы должны представить каждую категорию признака в виде бинарного признака, для этого выполняем дамми-кодирование.

В отличие от деревьев решений и их более продвинутых потомков – ансамблей на основе деревьев решений, линейные модели не способны уловить сложные нелинейные взаимосвязи между признаками и зависимой переменной, для этого мы выполняем биннинг. Кроме того, в отличие от древовидных алгоритмов линейные модели не могут описывать сложные взаимодействия, и мы должны их создать самостоятельно для улучшения качества.

Для ансамблей на основе деревьев решений (случайный лес, градиентный бустинг) часто бывает полезно преобразовать категориальный признак с большим количеством уровней⁴ в количественный, представив каждую категорию в виде числа. Это обусловлено проблемой множественных сравнений, актуальной для древовидных алгоритмов: в качестве признака расщепления чаще всего выбирается тот, по которому может быть рассмотрено наибольшее количество вариантов расщепления (при этом нет гарантии, что этот признак является действительно полезным с точки зрения взаимосвязи с зависимой переменной или информативности). Поясним подробнее.

В бинарном дереве CART, лежащем в основе случайного леса и некоторых реализациях градиентного бустинга (XGBoost), для категориального признака с k категориями будет рассмотрено $2^{k-1} - 1$ вариантов разбиений (при условии что категориальный признак обрабатывается по принципу «как есть»). Как была получена эта формула? Здесь все возможные разбиения для категориального признака удобно представить по аналогии с двоичным представлением числа. Если атрибут имеет k уникальных значений, то у нас будет 2^k разбиений. Первое (где все нули) и последнее (все единицы) нас не интересуют, получаем $2^k - 2$. И так как порядок множеств здесь тоже неважен, получаем $(2^k - 2)/2$, или $2^{k-1} - 1$ первых (с единицами) двоичных представлений. Если $\{A, B, C, D, E\}$ – все возможные значения некоторого атрибута X, то для текущего разбиения, которое имеет представление, скажем $\{0, 0, 1, 0, 1\}$, получаем правило $X \text{ in } \{C, E\}$ для правой ветви и $[\text{not } \{0, 0, 1, 0, 1\} = \{1, 1, 0, 1, 0\}] = X \text{ in } \{A, B, D\}$ для левой ветви. Например, для признака, принимающего значения A, B, C, D, могут быть рассмотрены точки расщепления A и BCD, B и ACD, C и ABD, D и ABC, AB и CD, AC и BD, AD и BC.

Итак, количество возможных разбиений возрастает лавинообразно с увеличением количества категорий. Например, при $k = 33$ будет рассмотрено четыре миллиона возможных вариантов расщепления. Чем больше уровней у признака, тем больше вариантов разбиения и выше вероятность выбрать признак в качестве признака расщепления совершенно случайно без гарантии того, что признак сильно связан с зависимой переменной. Таким образом, алгоритм при выборе разбиений будет склоняться в пользу категориальных признаков с большим количеством уровней. Как вариант можно перекодировать категориальный признак с большим количеством уровней в количественный признак, например представив категории в виде частот, и, таким образом, перейти к $k - 1$ вариантам разбиения.

⁴ Такие признаки называют высококардинальными (кардинальность – это количество категорий).

В питоновской библиотеке scikit-learn каждый уровень категориальной переменной мы чаще всего кодируем дамми-переменной. Для этого нужно выполнить дамми-кодирование. Дамми-кодирование применительно к ансамблям на основе деревьев решений, способным обрабатывать категориальные переменные по принципу «как есть», стирает важную информацию о структуре категориального признака, по сути разбив один центральный признак на множество отдельных бинарных признаков. Бинарный признак может быть разбит только одним способом, а категориальный признак с k уровнями может быть разбит $2^{k-1} - 1$ способами. Таким образом, в полученном пространстве признаков количественные переменные могут получить большую важность, чем категориальные переменные, представленные бинарными признаками. Однако этот эффект характерен для дамми-кодирования категориальных признаков невысокой кардинальности.

Когда речь о дамми-кодировании категориальных признаков с высокой кардинальностью, это может привести к огромному увеличению размерности пространства признаков. Это значительно замедляет обучение, и если модель случайным образом выбирает часть признаков для каждого дерева или разбиения, то шансы присутствия только дамми-переменных в выборке отобранных признаков (речь идет о случайном отборе признаков для каждого дерева / разбиения каждого узла) искусственно увеличиваются, а шансы других переменных, которые должны учитываться в дереве/разбиении, сокращаются. Это заставляет модель рассматривать дамми-переменные как более полезные, что не обязательно так.

Кроме того, при использовании дамми-кодирования требуется большее количество разбиений, и мы будем наблюдать очень глубокие деревья, ориентированные влево. На каждом разбиении дерева могут отделять только одну категорию от других. Деревья должны помещать каждую категорию в отдельную ячейку, нет другого способа разделить один бинарный столбец, кроме как разделить между 0 и 1. Это приводит к большему количеству разбиений, необходимых для достижения такой же точности, как в случае применения других, более компактных кодировок. Помимо замедления обучения, такая схема не позволяет деревьям объединять похожие категории вместе, что может снизить качество модели.

Если рассматривать применение дамми-кодирования для всех трех методов на основе деревьев – одиночного дерева, случайного леса и градиентного бустинга, то именно для градиентного бустинга, использующего неглубокие деревья, дамми-кодирование высококардинальных признаков будет наименее удачным способом кодировки, небольшая глубина просто не даст возможности найти оптимальное расщепление.

Проиллюстрируем применение дамми-кодирования для дерева.

Импортируем необходимые библиотеки, классы и функции и загрузим данные.

```
# импортируем необходимые библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from dtreeviz.trees import dtreeviz
from utils import DFOneHotEncoder
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

```
# загружаем данные для визуализации работы дерева
example_enc = pd.read_csv('Data/example_enc.csv', sep=';')
example_enc.head()
```

	job_position	open_account_flg
0	UMN	0
1	UMN	0
2	SPC	0
3	SPC	0
4	SPC	0

```
# количество уникальных значений
print(example_enc['job_position'].nunique())
```

18

Переменная *job_position* является высококардинальной (18 категорий). Давайте избавимся от редких категорий, выполним дамми-кодирование, построим дерево на массиве полученных дамми-переменных и визуализируем дерево с помощью функции `dtreeviz()` одноименной библиотеки.

```
# избавляемся от редких категорий
example_enc.loc[example_enc['job_position'].value_counts() \
    [example_enc['job_position']].values < 20,
    'job_position'] = 'OTHER'

# увеличиваем количество отображаемых столбцов
pd.set_option('display.max_columns', 50)

# выполняем дамми-кодирование
enc = DFOneHotEncoder()
X_ohc = enc.fit_transform(example_enc[['job_position']])
X_ohc.head()
```

	job_position_ATP	job_position_BIS	job_position_BIU	job_position_DIR	job_position_INP	job_position_NOR	job_position_OTHER	job_position_PNA
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
# строим дерево решений
ohc_classifier = DecisionTreeClassifier(random_state=42)
ohc_classifier.fit(X_ohc.values,
                    example_enc['open_account_flg'].values)

# визуализируем построенное дерево
viz = dtreeviz(
    # модель дерева
    ohc_classifier,
    # массив признаков
    X_ohc,
    # массив меток
    example_enc['open_account_flg'],
```

```
# название зависимой переменной
target_name='open_account_flg',
# названия признаков
feature_names=X_ohc.columns,
# метки классов зависимой переменной
class_names=['label-0', 'label-1'])
```

viz

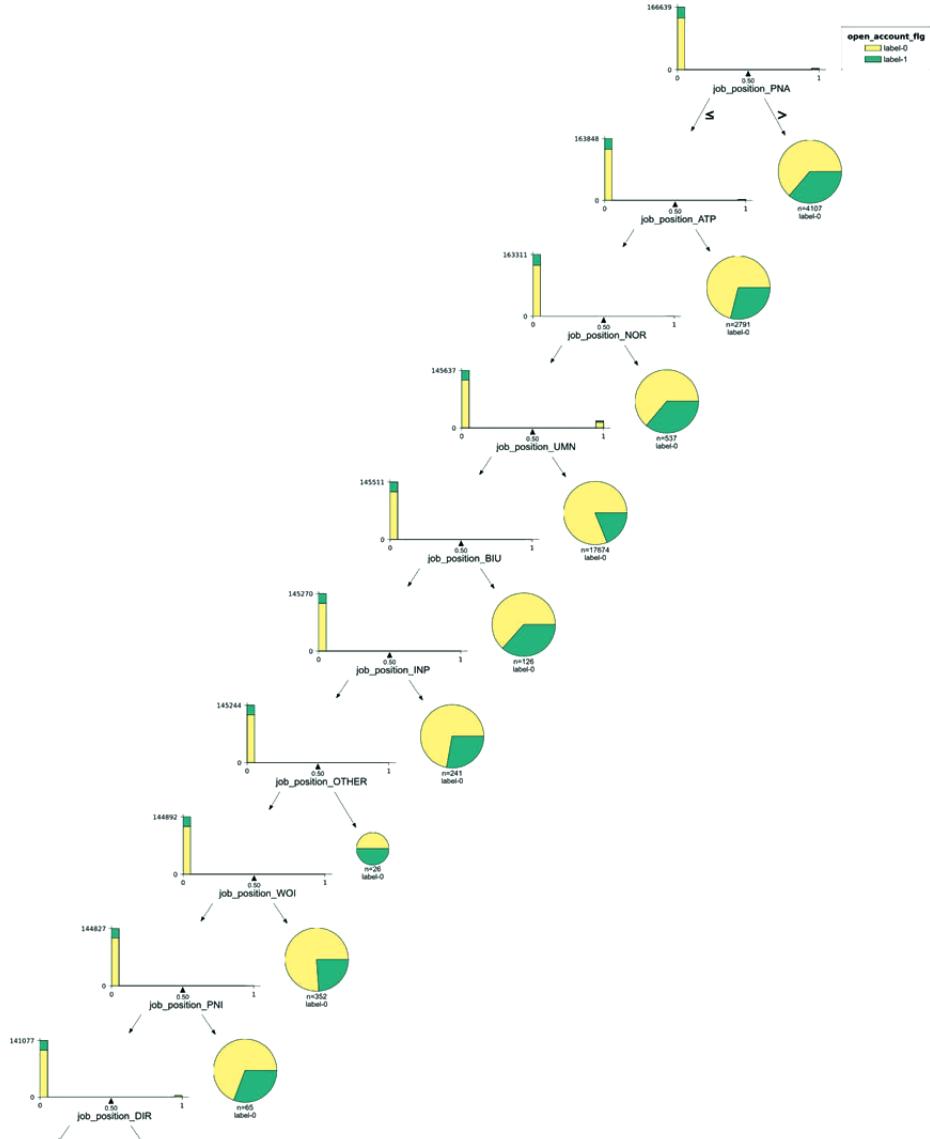


Рис. 38 Фрагмент дерева CART, использовалось дамми-кодирование

Действительно, мы наблюдаем очень глубокое дерево, ориентированное влево. Убеждаемся, что для градиентного бустинга, использующего неглубо-

кие деревья, дамми-кодирование высококардинальных признаков будет наименее удачным способом кодировки.

Чаще всего для превращения категориального признака в количественный используют:

- Label Encoding, когда категориям признака в лексикографическом порядке присваивают целые числа (имеет смысл для порядковых переменных);
- Frequency Encoding, когда каждую категорию признака представляем как относительную частоту наблюдений в данной категории (по сути, получаем вероятность появления категории в наборе данных);
- Ordinal Encoding, способ, похожий на Label Encoding с той только разницей, что мы присваиваем целые числа категориям в зависимости от порядка их появления в наборе данных;
- Likelihood Encoding, когда каждую категорию признака заменяем средним значением зависимой переменной в данной категории (для бинарной классификации это будет соответствовать вероятности положительного класса зависимой переменной в данной категории признака).

Наиболее эффективной кодировкой является Likelihood Encoding. Эффективность обусловлена тем, что при применении данной кодировки деревьям требуется меньшее количество разбиений, меньшая глубина, и ансамбли будут работать лучше (особенно градиентный бустинг, поскольку он является ансамблем неглубоких деревьев, для случайного леса эффективным может быть сопутствующий контроль глубины деревьев). Ранжирование становится не искусственным (как в LabelEncoding), а имеет линейную взаимосвязь с зависимой переменной.

Также обратите внимание: многие методы кластерного анализа не умеют работать с категориальными признаками, и когда нужно представить категориальный признак как количественный, некоторые способы кодировки категориальных признаков могут оказаться полезными.

16.2.1. Дамми-кодирование (One-Hot Encoding)

Выше мы упомянули, что линейные модели (линейная регрессия, логистическая регрессия и другие), метод опорных векторов и нейронные сети не умеют работать с категориальными признаками напрямую, поэтому если мы используем один из этих методов и у нас есть категориальные признаки, мы должны выполнить дамми-кодирование.

Кроме того, библиотека scikit-learn под капотом преобразовывает датаФреймы pandas в массивы NumPy, в которых каждый столбец должен быть количественным признаком. Поэтому мы должны представить категориальную переменную в виде набора количественных переменных. Самым распространенным способом такого представления является дамми-кодирование, еще называемое *one-hot-кодированием* (*one-hot-encoding* или *one-out-of-N encoding*). Если перевести дословно, *one-hot-encoding* – это *кодирование с одним горячим состоянием*.

Дамми-кодирование заключается в том, чтобы представить уровни категориальной переменной в виде новых признаков, которые могут принимать значения 0 и 1. Выделяют дамми-кодирование по методу неполного ранга и по методу полного ранга.

16.2.1.1. Дамми-кодирование по методу неполного ранга

При выполнении дамми-кодирования по методу неполного ранга мы заменяем категориальную переменную с k уровнями k дамми-переменными.

Например, категориальная переменная *State* имеет возможные уровни Washington, Arizona, Nevada, California и Oregon. Для того чтобы закодировать эти пять возможных уровней, мы создаем пять новых признаков Washington, Arizona, Nevada, California и Oregon.

Мы словно задаем вопросы: «данный штат является Вашингтоном?», «данный штат является Аризоной?», «данный штат является Невадой?», «данный штат является Калифорнией?», «данный штат является Орегоном?». Признак равен 1, если *State* имеет соответствующую категорию, или равен 0 в противном случае. В каждом наблюдении только один из пяти новых признаков будет равен 1. Вот почему данная операция называется кодированием с одним горячим (активным) состоянием.

State	Washington	Arizona	Nevada	California	Oregon
Washington	1	0	0	0	0
Arizona	0	1	0	0	0
Nevada	0	0	1	0	0
California	0	0	0	1	0
Oregon	0	0	0	0	1

Рис. 39 Дамми-кодирование по методу неполного ранга

Проблема такого представления – в избыточности. Например, если мы знаем, что [1, 0, 0, 0, 0] представляет «Washington», [0, 1, 0, 0, 0] представляет «Arizona», [0, 0, 1, 0, 0] представляет «Nevada», [0, 0, 0, 1, 0] представляет «California», нам не нужна еще одна переменная для представления «Oregon», вместо этого мы могли бы использовать только нулевые значения для Washington, Arizona, Nevada и California. Для линейных моделей без регуляризации дамми-кодирование по методу неполного ранга может привести к тому, что матрица признаков станет сингулярной. Проблема заключается в том, что когда инвертируется сингулярная или плохо обусловленная матрица, она может вернуть результат, не являющийся истинно обратным. Однако для линейных моделей с регуляризацией такой тип кодирования не представляет сложностей.

16.2.1.2. Дамми-кодирование по методу полного ранга

При выполнении дамми-кодирования по методу полного ранга мы заменяем категориальную переменную с k уровнями $k - 1$ дамми-переменными. Теперь, чтобы закодировать эти пять возможных уровней, мы создаем четыре новых признака Washington, Arizona, Nevada, California, а последнюю категорию Oregon объявляем опорной (обычно в качестве опорной категории берут либо первую, либо последнюю категорию) и с ней будем сравнивать все остальные категории.

State	Washington	Arizona	Nevada	California
Washington	1	0	0	0
Arizona	0	1	0	0
Nevada	0	0	1	0
California	0	0	0	1
Oregon	0	0	0	0

Рис. 40 Дамми-кодирование по методу полного ранга

Поясним подробнее. Допустим, мы предсказываем вероятность покупки и используем логистическую регрессию. В нашем исследовании используется переменная *Зарплата* с категориями *Низкая* (1), *Средняя* (2) и *Высокая* (3), опорной категорией станет последняя категория *Высокая*. На рисунке показано, что она будет автоматически переделана в две дамми-переменные вида 1/0, эти новые две переменные будут включены в логистическую регрессию, и коэффициент при каждой из них будет указывать отличие клиентов с данным значением *Зарплата* от клиентов со значением *Зарплата* = 3. Проще говоря, клиенты с низкой зарплатой (1), клиенты со средней зарплатой (1) сравниваются с клиентами, получающими высокую зарплату (0).

Зарплата	Низкая	Средняя
Низкая	1	0
Средняя	0	1
Высокая	0	0

Допустим, мы построили логистическую регрессию и получили следующие регрессионные коэффициенты:

	Регрессионный коэффициент	Экспоненциальный коэффициент
Зарплата		
Зарплата(1)	-2,456	0,086
Зарплата(2)	-1,705	0,182

Мы можем сделать следующие выводы.

У клиентов с низкой зарплатой (1), по сравнению с клиентами, получающими высокую зарплату (0), натуральный логарифм шансов совершить покупку на 2,456 меньше, или шансы совершить покупку в 0,086 раза меньше.

У клиентов со средней зарплатой (2) по сравнению с клиентами, получающими высокую зарплату (0), натуральный логарифм шансов совершить покупку на 1,705 меньше, или шансы совершить покупку в 0,182 раза меньше.

Для линейных моделей с регуляризацией дамми-кодирование по методу полного ранга может быть избыточным и является нежелательным.

Дамми-кодирование можно выполнить с помощью класса `OneHotEncoder` библиотеки `scikit-learn` и функции `get_dummies()` библиотеки `pandas`. У класса есть параметр `drop`, а у функции – параметр `drop_first`, задающий тип дамми-коди-

рования. По умолчанию для параметра `drop_first` класса `OneHotEncoder` задано значение `None`, для параметра `drop_first` функции `get_dummies()` задано значение `False`, при которых выполняется дамми-кодирование по методу неполного ранга.

Ниже приводятся примеры применения класса `OneHotEncoder` библиотеки `scikit-learn` и функции `get_dummies()` библиотеки `pandas`.

```
# импортируем класс OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
```

```
# загружаем данные
tr = pd.read_csv('Data/Stat_FE_train.csv', sep=';')
tr
```

	Class	Response
0	A	1
1	A	0
2	A	1
3	A	1
4	B	1
5	B	1
6	B	0
7	C	1
8	C	1

```
# выполняем дамми-кодирование
# по методу неполного ранга
dummies_unfull_rank_class = pd.get_dummies(tr['Class'],
                                             drop_first=False)
dummies_unfull_rank_class
```

	A	B	C	Class
0	1	0	0	A
1	1	0	0	A
2	1	0	0	A
3	1	0	0	A
4	0	1	0	B
5	0	1	0	B
6	0	1	0	B
7	0	0	1	C
8	0	0	1	C

```
# выполняем дамми-кодирование
# по методу полного ранга
dummies_full_rank_class = pd.get_dummies(tr['Class'],
                                         drop_first=True)
dummies_full_rank_class
```

B	C	Class
0	0	0
1	0	0
2	0	0
3	0	0
4	1	0
5	1	0
6	1	0
7	0	1
8	0	1

```
# создаем экземпляр класса OneHotEncoder
ohe = OneHotEncoder(sparse=False, drop=None,
                     handle_unknown='ignore')
# обучаем модель дамми-кодирования - определяем
# дамми для переменной Class
ohe.fit(tr['Class'].values.reshape(-1, 1))
# выполняем дамми-кодирование переменной Class по методу
# неполного ранга в обучавшем массиве признаков
ohe_train_unfull_rank = ohe.transform(tr['Class'].values.reshape(-1, 1))

ohe_train_unfull_rank

array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

```
# создаем экземпляр класса OneHotEncoder
ohe = OneHotEncoder(sparse=False, drop='first',
                     handle_unknown='ignore')
# обучаем модель дамми-кодирования - определяем
# дамми для переменной Class
ohe.fit(tr['Class'].values.reshape(-1, 1))

# выполняем дамми-кодирование переменной Class по методу
# полного ранга в обучавшем массиве признаков
ohe_train_full_rank = ohe.transform(tr['Class'].values.reshape(-1, 1))
ohe_train_full_rank
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 1.]])
```

16.2.2. Кодирование контрастами сумм (Sum Encoding, Effect Encoding)

При выполнении дамми-кодирования по методу полного ранга коэффициенты при дамми-переменных выражают влияние каждой категории по сравнению с опорной. Кодирование контрастами применяется, когда нужно сравнить вклад каждой категории со средним вкладом по всем категориям. Кодирование контрастами похоже на дамми-кодирование по методу полного ранга с той только разницей, что в опорном уровне, представленном нулями, нули заменяются на -1 .

State	Washington	Arizona	Nevada	California
Washington	1	0	0	0
Arizona	0	1	0	0
Nevada	0	0	1	0
California	0	0	0	1
Oregon	-1	-1	-1	-1

Рис. 41 Кодирование контрастами сумм

```
# выполняем кодирование контрастами сумм
effects_class = pd.get_dummies(tr['Class']).astype(int)
effects_class = effects_class.iloc[:, :-1]
effects_class.loc[np.all(effects_class == 0, axis=1)] = -1
effects_class
```

A	B	Class
0	1	0
1	1	0
2	1	0
3	1	0
4	0	1
5	0	1
6	0	1
7	-1	-1
8	-1	-1

16.2.3. Присвоение категориям в лексикографическом порядке целочисленных значений, начиная с 0 (Label Encoding)

16.2.3.1. Общее знакомство

Категориям переменной в лексикографическом порядке можно присвоить целочисленные значения (начиная с 0), и в итоге категориальная переменная теряет свою категориальную природу и превращается в количественную.

Допустим, у нас есть переменная с 4 категориями A, B, C и D. Тогда мы присвоим A – 0, B – 1, C – 2, D – 3.

Переменная Class	Переменная Class	Кодировка Label Encoding для переменной Class	
A	0	B	1
B	1	A	0
C	2	C	2
D	3	A	0
		A	0
		D	3
		D	3
		B	1

Рис. 42 Label Encoding

Данный вид кодирования можно выполнить с помощью класса `LabelEncoder` библиотеки `scikit-learn`.

```
# импортируем класс LabelEncoder
from sklearn.preprocessing import LabelEncoder
# создаем экземпляр класса LabelEncoder
labelenc = LabelEncoder()

# выполняем Label Encoding
tr['Class_labelenc'] = labelenc.fit_transform(tr['Class'])
tr
```

Class	Response	Class_labelenc
0	A	1
1	A	0
2	A	1
3	A	1
4	B	1
5	B	1
6	B	0
7	C	1
8	C	1

Мы можем выполнить Label Encoding до разбиения на обучающую и тестовую выборки (до цикла перекрестной проверки).

Часто пишут, что схема Label Encoding предназначена для порядковых признаков, а для номинальных признаков не имеет смысла. Например, у нас есть признак *Цвет автомобиля* с категориями *Синий*, *Желтый*, *Красный*, мы можем закодировать их как 2, 0 и 1, но кажется, что это не будет иметь большого смысла, ведь, выполнив такую кодировку, мы предполагаем определенный порядок $0 < 1 < 2$, который здесь не выполняется. На практике схему часто используют и для номинальных признаков, особенно когда признаки являются высококардинальными и мы применяем градиентный бустинг. Обучение на подвыборках будет приводить к тому, что какие-то категории, представленные теперь количественными значениями, от подвыборки к подвыборке будут выпадать (если они редкие), и в силу этого у нас от подвыборки к подвыборке будет разный набор расщепляющих значений, разные расщепления и, соответственно, разные деревья. От разнообразия композиция лишь выиграет.

Схема кодировки должна предусмотреть возможность обработки новых категорий. Новые категории часто заменяют средним значением или нулем. Пропуски нередко кодируют значением вне диапазона (9999 или -9999), в классе LabelEncoder пропуски обрабатываются в последнюю очередь, поэтому им присваивается наибольшее целочисленное значение.

```
# выполняем Label Encoding для серии с пропуском
s = pd.Series(['Paris', 'Moscow', 'Moscow', np.nan, 'Tokyo'])
labelenc.fit_transform(s)

array([1, 0, 0, 3, 2])
```

16.2.3.2. Написание собственного класса CustomLabelEncoder

Давайте напишем собственный класс CustomLabelEncoder. Класс мы будем писать с использованием аннотирования и для экономии ресурсов воспользуемся хеш-функцией. Импортируем функцию `murmurhash3_32()`, генерирующую простую и быструю хеш-функцию MurmurHash2, а из модуля `typing` импортируем некоторые аннотации типов. В нашем случае аннотирование будет удобно тем, что пользователь класса по каждому методу сразу увидит, что приходит на вход и что будет на выходе.

```
# импортируем функцию murmurhash3_32, генерирующую
# простую и быструю хеш-функцию MurmurHash2
from sklearn.utils.murmurhash import murmurhash3_32
# импортируем аннотации типов из модуля typing
from typing import Optional, Union
```

Кроме того, мы создадим свою аннотацию под названием `AggarrayLike` на основе аннотации `Union`. Она полезна, когда переменная может обладать свойствами нескольких объявленных типов, в нашем случае свойствами массива NumPy, объекта Series библиотеки pandas, объекта DataFrame библиотеки pandas.

```
# задаем свою аннотацию под названием AggarrayLike
AggarrayLike = Union[np.ndarray, pd.Series, pd.DataFrame]
```

Класс может кодировать категории отдельных переменных или взаимодействий (интеракций) переменных. Редкие и новые категории кодируются как 0. Для больших наборов в целях сокращения времени выполнения кодировки можно использовать случайную подвыборку. В случае если надо создать пересечение категорий, мы делаем единую категорию из нескольких категорий. Пусть на вход приходят два столбца, [1,2,3] и ['a', 'b', 'c']. Тогда мы создаем единый столбец ['1_a', '2_b', '3_c']. Дополнительно для экономии места берем хеш от всех значений и получаем закодированный массив типа [245624562, -24562345634, 34562456]. Обратите внимание: в этом классе мы реализуем собственный метод `.fit_transform()`, не прибегая к классу `TransformerMixin`.

```
# создаем собственный класс CustomLabelEncoder
class CustomLabelEncoder:
    """
    Автор: Антон Вахрушев
    https://www.kaggle.com/btbpanda

    Классический LabelEncoder. Может кодировать категории
    отдельных переменных или интеракций переменных.
    Редкие и новые категории кодируются как 0.

    Параметры
    -----
    unknown: int, значение по умолчанию 1
        Порог для отнесения редких и новых категорий
        в отдельную группу.
    sample: int, значение по умолчанию None
        Размер случайной подвыборки для выполнения кодировки.
    random_state: int, значение по умолчанию 42
        Стартовое значение генератора псевдослучайных чисел
        для создания случайной подвыборки.
    """

    def __init__(self, unknown: int = 1,
                 sample: Optional[int] = None,
                 random_state: int = 42):
        # порог для отнесения редких и новых категорий
        # в отдельную группу
        self.unknown = unknown
        # размер случайной подвыборки для выполнения кодирования
        self.sample = sample
        # стартовое значение генератора псевдослучайных чисел
        # для создания случайной подвыборки
        self.random_state = random_state

    @staticmethod
    def _make_series(df: pd.DataFrame):
        """
        В случае если надо создать взаимодействие категорий,
        сделаем единую категорию из нескольких категорий.
        Пусть на вход приходят два столбца, [1,2,3], ['a', 'b', 'c']
        Тогда мы создаем единый столбец ['1_a', '2_b', '3_c']
        Дополнительно для экономии места берем хеш от всех
        значений и получаем закодированный массив типа
        [245624562, -24562345634, 34562456]
        """

```

```

res = np.empty((df.shape[0],), dtype=np.int32)

for n, inter in enumerate(zip(*[df[x] for x in df.columns])):
    h = murmurhash3_32('_'.join(map(str, inter)),
                         seed=42)
    res[n] = h

return pd.Series(res)

def _check_types(self, X: ArrayLike,
                 sample: Optional[int] = None) -> pd.Series:
    """
    Наш класс работает с Series. Этот метод проверяет,
    что пришло на вход, и создает Series.
    """
    if len(X.shape) > 1:
        # если пришел двумерный массив, надо закодировать
        # взаимодействие категорий, воспользуемся _make_series()
        # для создания единой серии
        X = self._make_series(pd.DataFrame(X))
    elif type(X) is np.ndarray:
        # если одномерный, значит, просто проверим тип
        X = pd.Series(X)

    # если задан sample и длина исходной серии больше подвыборки
    if sample and X.shape[0] > sample:
        # формируем подвыборку
        X = X.sample(sample, random_state=self.random_state)

    return X

def fit(self, X: ArrayLike):
    """
    Создаем кодировку
    """
    # в случае если надо закодировать интеракции, на вход
    # подаем датафрейм и делаем из него серию
    X = self._check_types(X, self.sample)

    # получаем абсолютные частоты категорий
    vc = X.value_counts(dropna=False)
    # создаем массив NumPy с категориями,
    # у которых частота выше порога unknown
    vals = vc.index[vc > self.unknown].values
    # создаем словарь, у которого ключи - категории,
    # значения - целочисленные значения, начинающиеся
    # с 1, и превращаем в серию
    self.encoding = pd.Series({x: n for (n, x) in enumerate(vals, 1)})

    return self

def transform(self, X: ArrayLike) -> np.ndarray:
    """
    Применяем кодировку
    """
    X = self._check_types(X)

```

```

# заменяем на нули те категории, которые не нашлись
res = X.map(self.encoding).fillna(0).astype(np.int32).values

return res

def fit_transform(self, X: ArrayLike) -> np.ndarray:
    """
    Метод .fit_transform() для совместимости
    """
    self.fit(X)

    return self.transform(X)

```

Давайте с помощью нашего класса выполним кодировку для одного признака.

```

# создаем экземпляр класса CustomLabelEncoder
custlabelenc = CustomLabelEncoder()
# выполняем Label Encoding
tr['Class_custlabelenc'] = custlabelenc.fit_transform(tr['Class'])
tr

```

Class	Response	Class_labelenc	Class_custlabelenc
0	A	1	0
1	A	0	0
2	A	1	0
3	A	1	0
4	B	1	1
5	B	1	1
6	B	0	1
7	C	1	2
8	C	1	2

Видим, что категории заменяются целочисленными значениями в лексикографическом порядке с той только разницей, что значения начинаются с 1, а не с 0.

Теперь создадим еще один признак и выполним кодировку для взаимодействия признаков.

```

# создаем еще один признак
tr['Agecat'] = pd.DataFrame(
    ['old', 'old', 'yng', 'yng', 'old',
     'old', 'yng', 'old', 'old'])
# выполняем Label Encoding для взаимодействия признаков
tr['Class_custlabelenc2'] = custlabelenc.fit_transform(
    tr[['Class', 'Agecat']])
tr

```

	Class	Response	Class_labelenc	Class_custlabelenc	Agecat	Class_custlabelenc2
0	A	1	0		1 old	1
1	A	0	0		1 old	1
2	A	1	0		1 yng	2
3	A	1	0		1 yng	2
4	B	1	1		2 old	3
5	B	1	1		2 old	3
6	B	0	1		2 yng	0
7	C	1	2		3 old	4
8	C	1	2		3 old	4

Видим, что каждая комбинация категорий признаков *Class* и *Agecat* закодирована целочисленным значением, а самая редкая комбинация с частотой 1 закодирована как 0.

16.2.4. Кодирование частотами (Frequency/Count Encoding)

16.2.4.1. Общее знакомство

Каждую категорию признака можно представить как количество наблюдений в данной категории (абсолютную частоту) и как количество наблюдений в данной категории, поделенное на общее количество наблюдений (относительную частоту).

Переменная Class		Кодировка Frequency Encoding для переменной Class	
Переменная Class	относительная частота	абсолютная частота	
A	0,44 (4 из 9)	4	
B	0,33 (3 из 9)	3	
C	0,22 (2 из 9)	2	

Рис. 43 Frequency Encoding

```
# создаем признак Class_abs_freq, у которого
# каждое значение - абсолютная частота
abs_freq = tr['Class'].value_counts()
tr['Class_abs_freq'] = tr['Class'].map(abs_freq)
tr
```

Class	Response	Class_labelenc	Class_custlabelenc	Agecat	Class_custlabelenc2	Class_abs_freq
0	A	1	0	1	old	1
1	A	0	0	1	old	1
2	A	1	0	1	yng	2
3	A	1	0	1	yng	2
4	B	1	1	2	old	3
5	B	1	1	2	old	3
6	B	0	1	2	yng	0
7	C	1	2	3	old	4
8	C	1	2	3	old	4

```
# создаем признак Class_rel_freq, у которого
# каждое значение - относительная частота
rel_freq = tr['Class'].value_counts() / len(tr['Class'])
tr['Class_rel_freq'] = tr['Class'].map(rel_freq)
tr
```

Class	Response	Class_labelenc	Class_custlabelenc	Agecat	Class_custlabelenc2	Class_abs_freq	Class_rel_freq
0	A	1	0	1	old	1	0.444444
1	A	0	0	1	old	1	0.444444
2	A	1	0	1	yng	2	0.444444
3	A	1	0	1	yng	2	0.444444
4	B	1	1	2	old	3	0.333333
5	B	1	1	2	old	3	0.333333
6	B	0	1	2	yng	0	0.333333
7	C	1	2	3	old	4	0.222222
8	C	1	2	3	old	4	0.222222

Обратите внимание, что поскольку мы используем вычисления, то кодировку частотами нужно выполнять строго после разбиения на обучающую и тестовую выборки (внутри цикла перекрестной проверки). Следует помнить, что в обучающем наборе могут встречаться не все возможные категории признака, некоторые категории могут присутствовать только в тестовом наборе или появятся только тогда, когда модель будет применяться к новым данным. Поэтому схема кодировки должна предусмотреть возможность обработки новых категорий. Новые категории часто заменяют 1.

Кроме того, когда мы найдем комбинацию оптимальных значений гиперпараметров модели на проверочной выборке / проверочных блоках перекрестной проверки и нам нужно будет построить модель с этой комбинацией на всей исторической выборке, мы должны заново вычислить частоты для категорий на всей исторической выборке.

На практике чаще используют кодирование относительными частотами, потому что абсолютные частоты сильно зависят от размера обучающего набора данных.

Кодирование частотами может приводить к коллизиям, когда разные категории встречаются одинаковое количество раз и мы должны присвоить им одинаковые частоты. В таких случаях можно попробовать добавить случайный шум, добавить константу или объединить, если исходные категории не отличаются по зависимой переменной (можно использовать тест хи-квадрат), и посмотреть, какой вариант дает наилучший результат. Для линейной регрессии и логистической регрессии часто бывает полезным выполнить для полученной переменной преобразование, максимизирующее нормальность.

16.2.4.2. Написание собственных классов CountEncoder и CountEncoder2

Давайте импортируем класс `Counter` из модуля `collections` и самостоятельно реализуем класс под названием `CountEncoder`, выполняющий кодирование частотами. При этом реализуем собственный метод `.fit_transform()`.

```
# импортируем класс Counter
from collections import Counter

# пишем собственный класс CountEncoder, заменяющий категории
# относительными или абсолютными частотами
class CountEncoder():
```

Классический CountEncoder. Может кодировать категории переменных абсолютными или относительными частотами. Умеет обрабатывать категории с одинаковыми частотами, новые категории, пропуски.

Параметры

- - - - -

`min_count: float`, значение по умолчанию 0.04
Пороговое значение для частоты категории, меньше которого возвращаем `nan_value` - специальное значение для пропусков и редких категорий, для способа кодировки 'frequency' задаем число с плавающей точкой, для способа кодировки 'count' задаем целое число.

`encoding_method: str`, значение по умолчанию 'frequency'
Способ кодирования частотами – абсолютными ('count')
или относительными ('frequency').

`correct_equal_freq: bool`, значение по умолчанию `False`
Включает корректировку одинаковых частот для категорий

включает корректировку одинаковых параметров `value: int` значение по умолчанию `-1`

Специальное значение для пропусков и редких категорий. `None`, значение по умолчанию - 1

copy: bool, значение по умолчанию true

Возвращает копию.

三

```

# если для параметра min_count задано значение, которое
# меньше/равно 0, или значение больше 1 и при этом для
# параметра encoding_method задано значение 'frequency',
# выдать ошибку
if ((min_count <= 0) | (min_count > 1)) &
    encoding_method == 'frequency'):
    raise ValueError(f"min_count for 'frequency' should be "
                      f"between 0 and 1, was {min_count}")

# если для параметра encoding_method задано значение 'count'
# и значение для параметра min_count не является
# целочисленным, выдать ошибку
if encoding_method == 'count' and not isinstance(min_count, int):
    raise ValueError("encoding_method 'count' requires "
                      "integer values for min_count")

# пороговое значение для частоты категории, меньше которого
# возвращаем nan_value - специальное значение для пропусков
# и редких категорий, для способа кодировки 'frequency'
# задаем число с плавающей точкой, для способа кодировки
# 'count' задаем целое число
self.min_count = min_count
# способ кодирования частотами - абсолютными ('count')
# или относительными ('frequency')
self.encoding_method = encoding_method
# корректировка одинаковых частот для категорий
self.correct_equal_freq = correct_equal_freq
# специальное значение для пропусков и редких категорий
self.nan_value = nan_value
# выполнение копирования
self.copy = copy

def __is_numpy(self, X):
    """
    Метод проверяет, является ли наш объект массивом NumPy.
    """
    return isinstance(X, np.ndarray)

def fit(self, X, y=None):
    # создаем пустой словарь counts
    self.counts = {}

    # записываем результат метода __is_numpy
    is_np = self.__is_numpy(X)

    # записываем общее количество наблюдений
    n_obs = len(X)

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        if is_np:
            X = X.reshape(-1, 1)
        else:
            X = X.to_frame()

    # записываем количество столбцов
    ncols = X.shape[1]

```

```
for i in range(ncols):
    # если выбрано значение 'frequency' для encoding_method
    if self.encoding_method == 'frequency':
        # если объект - массив NumPy:
        if is_np:
            # создаем временный словарь cnt, ключи - категории
            # переменной, значения - абсолютные частоты категорий
            cnt = dict(Counter(X[:, i]))
            # абсолютные частоты заменяем на относительные
            cnt = {key: value / n_obs for key, value in cnt.items()}
        # если объект - Dataframe pandas:
    else:
        # создаем временный словарь cnt,
        # ключи - категории переменной,
        # значения - относительные частоты категорий
        cnt = (X.iloc[:, i].value_counts() / n_obs).to_dict()

    # если значение min_count больше 0
    if self.min_count > 0:
        # если относительная частота категории меньше min_count,
        # возвращаем nan_value
        cnt = dict((k, self.nan_value if v < self.min_count else v)
                   for k, v in cnt.items())

# если выбрано значение 'count' для encoding_method
if self.encoding_method == 'count':
    # если объект - массив NumPy:
    if is_np:
        # создаем временный словарь cnt,
        # ключи - категории переменной,
        # значения - абсолютные частоты категорий
        cnt = dict(Counter(X[:, i]))
    # если объект - Dataframe pandas:
    else:
        # создаем временный словарь cnt,
        # ключи - категории переменной,
        # значения - абсолютные частоты категорий
        cnt = (X.iloc[:, i].value_counts()).to_dict()

    # если значение min_count больше 0:
    if self.min_count > 0:
        # если абсолютная частота категории меньше min_count,
        # возвращаем nan_value
        cnt = dict((k, self.nan_value if v < self.min_count else v)
                   for k, v in cnt.items())

# обновляем словарь counts, ключом словаря counts будет
# индекс переменной, значением словаря counts будет
# словарь cnt, ключами будут категории переменной,
# значениями - частоты категорий переменной
self.counts.update({i: cnt})

# если для параметра correct_equal_freq задано значение True,
# добавляем случайный шум к частотам категорий
if self.correct_equal_freq:
    noise_param = 0.01
    np.random.seed(0)
```

```

for v in self.counts.values():
    for key, value in v.items():
        noise_value = value * noise_param
        noise = np.random.uniform(-noise_value, noise_value)
        v[key] = value + noise

return self

def transform(self, X):
    # выполняем копирование массива
    if self.copy:
        X = X.copy()

    # записываем результатом метода __is_numpy
    is_np = self.__is_numpy(X)

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        if is_np:
            X = X.reshape(-1, 1)
        else:
            X = X.to_frame()

    # записываем количество столбцов
    ncols = X.shape[1]

    for i in range(ncols):
        cnt = self.counts[i]

        # дополняем словарь неизвестными категориями,
        # которых не было в методе fit

        # если объект - массив NumPy
        if is_np:
            unknown_categories = set(X[:, i]) - set(cnt.keys())

        # если объект - датафрейм pandas
        else:
            unknown_categories = set(X.iloc[:, i].values) - set(cnt.keys())

        for category in unknown_categories:
            cnt[category] = 1.0

    # если объект - массив NumPy:
    if is_np:

        # получаем из словаря по каждой переменной кортеж
        # с категориями и список с частотами
        k, v = list(zip(*sorted(cnt.items())))

        # кортеж преобразовываем в массив
        v = np.array(v)

        # возвращаем индексы
        ix = np.searchsorted(k, X[:, i], side='left')
        # заменяем категории частотами с помощью индексов

```

```

X[:, i] = v[ix]
# если объект - Dataframe pandas:
else:
    # заменяем категории частотами
    X.iloc[:, i].replace(cnt, inplace=True)

return X

def fit_transform(self, X):
    """
    Метод .fit_transform() для совместимости
    """
    self.fit(X)

    return self.transform(X)

```

Наш класс умеет кодировать категории переменных абсолютными или относительными частотами. Он умеет обрабатывать категории с одинаковыми частотами, новые категории, пропуски.

Давайте загрузим обучающий и тестовый наборы, взглянем на первые пять и последние пять наблюдений.

```

# загружаем данные и смотрим их
freq_train = pd.read_csv('Data/freq_train.csv', sep=';')
# выведем первые пять наблюдений
freq_train.head()

```

	EDUCATION	MARITAL_STATUS	FAMILY_INCOME
0	Среднее специальное	Разведен(a)	от 20000 до 50000 руб.
1	Среднее специальное	Состою в браке	от 10000 до 20000 руб.
2	Высшее	Не состоял в браке	от 10000 до 20000 руб.
3	Среднее специальное	Состою в браке	от 20000 до 50000 руб.
4	Среднее специальное	Не состоял в браке	от 10000 до 20000 руб.

```

# выведем последние пять наблюдений
freq_train.tail()

```

	EDUCATION	MARITAL_STATUS	FAMILY_INCOME
10651	Среднее специальное	Не состоял в браке	от 10000 до 20000 руб.
10652	Среднее специальное	Состою в браке	от 20000 до 50000 руб.
10653	Среднее специальное	Состою в браке	от 10000 до 20000 руб.
10654	Высшее	Состою в браке	от 10000 до 20000 руб.
10655	Среднее специальное	Вдовец/Вдова	от 10000 до 20000 руб.

Давайте выведем относительные частоты категорий по каждому признаку.

```
# посмотрим относительные частоты категорий по признакам
for i in freq_train.columns:
    print(freq_train[i].value_counts(normalize=True))
    print("")
```

Среднее специальное 0.422860
 Среднее 0.312875
 Высшее 0.205518
 Неоконченное высшее 0.035661
 Неполное среднее 0.023086
 Name: EDUCATION, dtype: float64

Состою в браке 0.617399
 Не состоял в браке 0.236017
 Разведен(а) 0.084084
 Вдовец/Вдова 0.039696
 Гражданский брак 0.022804
 Name: MARITAL_STATUS, dtype: float64

от 10000 до 20000 руб. 0.465372
 от 20000 до 50000 руб. 0.396490
 от 5000 до 10000 руб. 0.102571
 свыше 50000 руб. 0.031625
 до 5000 руб. 0.003941
 Name: FAMILY_INCOME, dtype: float64

Давайте выведем абсолютные частоты категорий по каждому признаку.

```
# посмотрим абсолютные частоты категорий по признакам
for i in freq_train.columns:
    print(freq_train[i].value_counts())
    print("")
```

Среднее специальное 4506
 Среднее 3334
 Высшее 2190
 Неоконченное высшее 380
 Неполное среднее 246
 Name: EDUCATION, dtype: int64

Состою в браке 6579
 Не состоял в браке 2515
 Разведен(а) 896
 Вдовец/Вдова 423
 Гражданский брак 243
 Name: MARITAL_STATUS, dtype: int64

от 10000 до 20000 руб. 4959
 от 20000 до 50000 руб. 4225
 от 5000 до 10000 руб. 1093
 свыше 50000 руб. 337
 до 5000 руб. 42
 Name: FAMILY_INCOME, dtype: int64

Проверяем работу класса `CountEncoder` на датафрейме `pandas`. По умолчанию выполняется кодирование относительными частотами, категории с относительной частотой меньше 0,04 будут заменены на -1.

```
# создаем копию датафрейма
freq_train_pandas = freq_train.copy()

# создаем экземпляр класса CountEncoder
count = CountEncoder()

# выполняем кодирование относительными частотами
# в датафрейме pandas
freq_train_pandas = count.fit_transform(freq_train_pandas)

# смотрим результаты
freq_train_pandas
```

	EDUCATION	MARITAL_STATUS	FAMILY_INCOME		EDUCATION	MARITAL_STATUS	FAMILY_INCOME
0	0.422860	0.084084	0.396490		0	Среднее специальное	Разведен(а) от 20000 до 50000 руб.
1	0.422860	0.617399	0.465372		1	Среднее специальное	Состою в браке от 10000 до 20000 руб.
2	0.205518	0.236017	0.465372		2	Высшее	Не состоял в браке от 10000 до 20000 руб.
3	0.422860	0.617399	0.396490		3	Среднее специальное	Состою в браке от 20000 до 50000 руб.
4	0.422860	0.236017	0.465372		4	Среднее специальное	Не состоял в браке от 10000 до 20000 руб.
...				
10651	0.422860	0.236017	0.465372	10651	Среднее специальное	Не состоял в браке от 10000 до 20000 руб.	
10652	0.422860	0.617399	0.396490	10652	Среднее специальное	Состою в браке от 20000 до 50000 руб.	
10653	0.422860	0.617399	0.465372	10653	Среднее специальное	Состою в браке от 10000 до 20000 руб.	
10654	0.205518	0.617399	0.465372	10654	Высшее	Состою в браке от 10000 до 20000 руб.	
10655	0.422860	-1.000000	0.465372	10655	Среднее специальное	Вдовец/Вдова от 10000 до 20000 руб.	

Исходные категории и частоты

Состою в браке	6579	0.617399	-0.04
Не состоял в браке	2515	0.236017	
Разведен(а)	896	0.084084	
Вдовец/Вдова	423	0.039696	
Гражданский брак	243	0.022804	

Выведем относительные частоты категорий по каждому признаку.

```
# посмотрим относительные частоты категорий по признакам
for i in freq_train_pandas.columns:
    print(freq_train_pandas[i].value_counts(normalize=True))
    print("")
```

0.422860	0.422860
0.312875	0.312875
0.205518	0.205518
-1.000000	0.058746
Name: EDUCATION, dtype: float64	
0.617399	0.617399

```
0.236017    0.236017
0.084084    0.084084
-1.000000    0.062500
Name: MARITAL_STATUS, dtype: float64
```

```
0.465372    0.465372
0.396490    0.396490
0.102571    0.102571
-1.000000    0.035567
Name: FAMILY_INCOME, dtype: float64
```

Теперь выполним кодирование абсолютными частотами.

```
# создаем копию датафрейма
freq_train_pandas = freq_train.copy()

# создаем экземпляр класса CountEncoder
count = CountEncoder(encoding_method='count', min_count=425)

# выполняем кодирование абсолютными частотами
# в датафрейме pandas
freq_train_pandas = count.fit_transform(freq_train_pandas)

# смотрим результаты
freq_train_pandas
```

	EDUCATION	MARITAL_STATUS	FAMILY_INCOME
0	4506	896	4225
1	4506	6579	4959
2	2190	2515	4959
3	4506	6579	4225
4	4506	2515	4959
...
10651	4506	2515	4959
10652	4506	6579	4225
10653	4506	6579	4959
10654	2190	6579	4959
10655	4506	-1	4959

Выведем абсолютные частоты категорий по каждому признаку.

```
# посмотрим абсолютные частоты категорий по признакам
for i in freq_train_pandas.columns:
    print(freq_train_pandas[i].value_counts())
    print("")
```

```
4506    4506
3334    3334
2190    2190
-1      626
Name: EDUCATION, dtype: int64
```

```
6579    6579
2515    2515
896     896
-1      666
Name: MARITAL_STATUS, dtype: int64
```

```
4959    4959
4225    4225
1093   1093
-1      379
Name: FAMILY_INCOME, dtype: int64
```

Проверяем наш класс `CountEncoder` на массиве NumPy. Вновь напомним, что по умолчанию выполняется кодирование относительными частотами, категории с относительной частотой меньше 0,04 будут заменены на -1.

```
# создаем из датафрейма массив NumPy
freq_train_numpy = freq_train.values

# создаем экземпляр класса CountEncoder
count = CountEncoder()

# выполняем кодирование относительными частотами
# в массиве NumPy
freq_train_numpy = count.fit_transform(freq_train_numpy)

# смотрим результаты
freq_train_numpy

array([[0.42286036036036034, 0.08408408408408409, 0.39649024024024027],
       [0.42286036036036034, 0.6173986486486487, 0.4653716216216216],
       [0.20551801801801803, 0.23601726726726727, 0.4653716216216216],
       ...,
       [0.42286036036036034, 0.6173986486486487, 0.4653716216216216],
       [0.20551801801801803, 0.6173986486486487, 0.4653716216216216],
       [0.42286036036036034, -1.0, 0.4653716216216216]], dtype=object)

array([['Среднее специальное', 'Разведен(а)', 'от 20000 до 50000 руб.'],
       ['Среднее специальное', 'Состою в браке', 'от 10000 до 20000 руб.'],
       ['Высшее', 'Не состоял в браке', 'от 10000 до 20000 руб.'],
       ...,
       ['Среднее специальное', 'Состою в браке', 'от 10000 до 20000 руб.'],
       ['Высшее', 'Состою в браке', 'от 10000 до 20000 руб.'],
       ['Среднее специальное', 'Вдовец/Вдова', 'от 10000 до 20000 руб.']],
      dtype=object)
```

Выведем относительные частоты категорий по каждому признаку.

```
# посмотрим относительные частоты категорий по признакам
freq_train_numpy = pd.DataFrame(freq_train_numpy)
for i in freq_train_numpy.columns:
    print(freq_train_numpy[i].value_counts(normalize=True))
    print("")
```

```
0.422860    0.422860
0.312875   0.312875
```

```

0.205518    0.205518
-1.000000   0.058746
Name: 0, dtype: float64

0.617399    0.617399
0.236017    0.236017
0.084084    0.084084
-1.000000   0.062500
Name: 1, dtype: float64

0.465372    0.465372
0.396490    0.396490
0.102571    0.102571
-1.000000   0.035567
Name: 2, dtype: float64

```

Теперь выполним кодирование абсолютными частотами.

```

# создаем из датафрейма массив NumPy
freq_train_numpy = freq_train.values

# создаем экземпляр класса CountEncoder
count = CountEncoder(encoding_method='count', min_count=425)

# выполняем кодирование абсолютными частотами
# в массиве NumPy
freq_train_numpy = count.fit_transform(freq_train_numpy)

# смотрим результаты
freq_train_numpy

array([[4506, 896, 4225],
       [4506, 6579, 4959],
       [2190, 2515, 4959],
       ...,
       [4506, 6579, 4959],
       [2190, 6579, 4959],
       [4506, -1, 4959]], dtype=object)

```

Выведем абсолютные частоты категорий по каждому признаку.

```

# посмотрим абсолютные частоты категорий по признакам
freq_train_numpy = pd.DataFrame(freq_train_numpy)
for i in freq_train_numpy.columns:
    print(freq_train_numpy[i].value_counts())
    print("")

```

```

4506    4506
3334    3334
2190    2190
-1      626
Name: 0, dtype: int64

6579    6579
2515    2515
896     896

```

```
-1      666
Name: 1, dtype: int64

4959    4959
4225    4225
1093    1093
-1      379
Name: 2, dtype: int64
```

Теперь проверим класс CountEncoder на работу с одинаковыми частотами категорий. Мы можем добавить к частотам случайный шум (по умолчанию эта возможность отключена). Давайте создадим датафрейм с тремя категориями, две из которых будут одинаковыми по частоте, и применим наш класс с настройками по умолчанию.

```
# создаем датафрейм, в котором две категории
# совпадают по частоте
```

```
dct = {'class': ['A', 'A', 'A', 'A', 'B',
                 'B', 'B', 'B', 'C']}
expl = pd.DataFrame(data=dct)
expl
```

	class
0	A
1	A
2	A
3	A
4	B
5	B
6	B
7	B
8	C

```
# проверяем работу класса с настройками по умолчанию
```

```
count = CountEncoder()
expl['class_freq'] = count.fit_transform(expl['class'])
expl
```

class	class_freq
0	A 0.444444
1	A 0.444444
2	A 0.444444
3	A 0.444444
4	B 0.444444
5	B 0.444444
6	B 0.444444
7	B 0.444444
8	C 0.111111

Теперь применим наш класс, включив добавление шума к частотам категорий.

```
# проверяем работу класса, включив добавление
# шума к частотам категорий
count = CountEncoder(correct_equal_freq=True)
expl['class_freq_noise'] = count.fit_transform(expl[['class']])
expl
```

class	class_freq	class_freq_noise
0	A 0.444444	0.444878
1	A 0.444444	0.444878
2	A 0.444444	0.444878
3	A 0.444444	0.444878
4	B 0.444444	0.446357
5	B 0.444444	0.446357
6	B 0.444444	0.446357
7	B 0.444444	0.446357
8	C 0.111111	0.111339

Видим, что к частотам категорий добавлен шум.

А сейчас проверим способность нашего класса обработать новые категории, не встретившиеся в обучающих данных. Новые категории (выделим красной рамкой) будем заменять единицей.

Начнем с датафреймов pandas.

```
# создаем обучающий датафрейм pandas
train = pd.DataFrame(
    {'class': ['A', 'A', 'A', 'A', 'B',
              'B', 'B', 'C', 'C'],
     'city': ['MSK', 'MSK', 'MSK', 'SPB', 'EKB',
              'EKB', 'MSK', 'EKB', 'SPB']})
train
```

```
class    city
0      A   MSK
1      A   MSK
2      A   MSK
3      A   SPB
4      B   EKB
5      B   EKB
6      B   MSK
7      C   EKB
8      C   SPB
```

```
# проверяем работу класса на
# обучающем датафрейме pandas
count = CountEncoder()
res = count.fit_transform(train)
res
```

```
class    city
0  0.444444  0.444444
1  0.444444  0.444444
2  0.444444  0.444444
3  0.444444  0.222222
4  0.333333  0.333333
5  0.333333  0.333333
6  0.333333  0.444444
7  0.222222  0.333333
8  0.222222  0.222222
```

```
# создаем тестовый датафрейм pandas
test = pd.DataFrame(
    {'class': ['A', 'A', 'A', 'A',
              'B', 'B', 'D', 'C'],
     'city': ['NSK', 'MSK', 'MSK', 'SPB',
              'SPB', 'EKB', 'MSK', 'NSK']})
test
```

	class	city
0	A	NSK
1	A	MSK
2	A	MSK
3	A	SPB
4	B	SPB
5	B	EKB
6	D	MSK
7	C	NSK

```
# проверяем работу класса на
# тестовом датафрейме pandas
res = count.transform(test)
res
```

	class	city
0	0.444444	1.000000
1	0.444444	0.444444
2	0.444444	0.444444
3	0.444444	0.222222
4	0.333333	0.222222
5	0.333333	0.333333
6	1.000000	0.444444
7	0.222222	1.000000

Теперь проверим работу класса на массивах NumPy.

```
# создаем обучающий массив NumPy
train = train.values
train

array([['A', 'MSK'],
       ['A', 'MSK'],
       ['A', 'MSK'],
       ['A', 'SPB'],
       ['B', 'EKB'],
       ['B', 'EKB'],
       ['B', 'MSK'],
       ['C', 'EKB'],
       ['C', 'SPB']], dtype=object)

# проверяем работу класса на обучающем массиве NumPy
res = count.fit_transform(train)
res
```

```
array([[0.4444444444444444, 0.4444444444444444],  
       [0.4444444444444444, 0.4444444444444444],  
       [0.4444444444444444, 0.4444444444444444],  
       [0.4444444444444444, 0.2222222222222222],  
       [0.3333333333333333, 0.3333333333333333],  
       [0.3333333333333333, 0.3333333333333333],  
       [0.3333333333333333, 0.4444444444444444],  
       [0.2222222222222222, 0.3333333333333333],  
       [0.2222222222222222, 0.2222222222222222]], dtype=object)  
  
# создаем тестовый массив NumPy  
test = test.values  
test  
  
array([('A', 'NSK'),  
       ('A', 'MSK'),  
       ('A', 'MSK'),  
       ('A', 'SPB'),  
       ('B', 'SPB'),  
       ('B', 'EKB'),  
       ('D', 'MSK'),  
       ('C', 'NSK')], dtype=object)  
  
# проверяем работу класса на тестовом массиве NumPy  
res = count.transform(test)  
res  
  
array([[0.4444444444444444, 1.0],  
       [0.4444444444444444, 0.4444444444444444],  
       [0.4444444444444444, 0.4444444444444444],  
       [0.4444444444444444, 0.2222222222222222],  
       [0.3333333333333333, 0.2222222222222222],  
       [0.3333333333333333, 0.3333333333333333],  
       [1.0, 0.4444444444444444],  
       [0.2222222222222222, 1.0]], dtype=object)
```

Наконец, убедимся, что класс умеет обрабатывать пропуски. Значения в первых пяти наблюдениях заменим на пропуски и применим наш класс. Напомним, пропуски мы заменяем единицей.

```
# создаем копию датасета  
freq_train_pandas = freq_train.copy()  
# задаем пропуски в первых пяти наблюдениях  
freq_train_pandas.iloc[:5] = np.Nan  
# выполняем кодирование относительными частотами  
# в датасете pandas  
freq_train_pandas = count.fit_transform(freq_train_pandas)  
freq_train_pandas.head(10)
```

	EDUCATION	MARITAL_STATUS	FAMILY_INCOME
0	1.000000	1.000000	1.000000
1	1.000000	1.000000	1.000000
2	1.000000	1.000000	1.000000
3	1.000000	1.000000	1.000000
4	1.000000	1.000000	1.000000
5	0.422485	0.617211	0.396303
6	0.312875	0.617211	0.396303
7	0.205424	0.617211	0.396303
8	0.422485	0.617211	0.396303
9	0.312875	0.617211	0.102571

Если тяжело разобраться, что именно делает класс, можно разложить его на более простые компоненты. Для краткости разберем только на примере датфрейма pandas.

```
# вновь создаем копию датфрейма
freq_train_pandas = freq_train.copy()
# записываем количество столбцов
ncols = freq_train_pandas.shape[1]
ncols

3

# задаем значение nan_value, на которое будем заменять,
# если относительная частота будет меньше min_count
nan_value = -1
min_count = 0.1
# создаем пустой словарь counts
counts = {}
# записываем общее количество наблюдений
n_obs = len(freq_train_pandas)

for i in range(ncols):
    # печатаем название переменной
    print(i)
    # создаем временный словарь cnt, ключи - категории переменной,
    # значения - относительные частоты категорий
    cnt = (freq_train_pandas.iloc[:, i].value_counts() / n_obs).to_dict()
    # печатаем временный словарь cnt
    print(cnt)
    print("")
    # если относительная частота категории меньше min_count,
    # возвращаем nan_value
    if min_count > 0:
        cnt = dict((k, nan_value if v < min_count else v)
                   for k, v in cnt.items())
    # печатаем временный словарь cnt после применения условия
    print(cnt)
    print("")
```

Теперь напишем собственный класс `CountEncoder2`, который может кодировать абсолютными частотами не только категории отдельной переменной, но и совместное появление категорий во взаимодействиях переменных. Обратите внимание на использование наследования от уже ранее созданного класса `CustomLabelEncoder`.

```
class CountEncoder2(CCustomLabelEncoder):
```

Автор: Антон Вахрушев
<https://www.kaggle.com/btbpanda>

Классический CountEncoder. Умеет кодировать категории переменной и интеракций переменных абсолютными частотами. Многие методы схожи с LabelEncoder, поэтому отнаследуемся от него.

```
def __init__(self):
    super().__init__()

def fit(self, X: ArrayLike):
    # в случае если надо закодировать частоту интеракции,
    # на вход подаем датафрейм и делаем из него серию
    X = self._check_types(X, self.sample)
    # вычисляем частоты
    vc = X.value_counts(dropna=False)
    # вхождения с частотой 1 хранить не надо, не найденные
    # в словаре мы и так будем заменять на 1
    self.encoding = vc[vc > 1]

    return self
```

```
def transform(self, X: ArrayLike) -> np.ndarray:
    X = self._check_types(X)
    # заменяем на 1 те категории, которые не нашлись
    res = X.map(self.encoding).fillna(1).astype(np.int32).values

    return res
```

С помощью класса CountEncoder2 закодируем абсолютными частотами отдельный признак и взаимодействие признаков.

```
# создаем копию датафрейма
freq_train_pandas = freq_train.copy()

# создаем экземпляр класса CountEncoder2
count2 = CountEncoder2()

# выполняем кодирование признака абсолютными частотами
# в датафрейме pandas
freq_train_pandas['ED_count'] = count2.fit_transform(
    freq_train_pandas[['EDUCATION']])

# выполняем кодирование взаимодействия признаков
# абсолютными частотами в датафрейме pandas
freq_train_pandas['ED_MAR_count'] = count2.fit_transform(
    freq_train_pandas[['EDUCATION', 'MARITAL_STATUS']])

# смотрим результаты
freq_train_pandas
```

	EDUCATION	MARITAL_STATUS	FAMILY_INCOME	ED_count	ED_MAR_count
0	Среднее специальное	Разведен(a)	от 20000 до 50000 руб.	4506	411
1	Среднее специальное	Состою в браке	от 10000 до 20000 руб.	4506	2788
2	Высшее	Не состоял в браке	от 10000 до 20000 руб.	2190	560
3	Среднее специальное	Состою в браке	от 20000 до 50000 руб.	4506	2788
4	Среднее специальное	Не состоял в браке	от 10000 до 20000 руб.	4506	1006
...
10651	Среднее специальное	Не состоял в браке	от 10000 до 20000 руб.	4506	1006
10652	Среднее специальное	Состою в браке	от 20000 до 50000 руб.	4506	2788
10653	Среднее специальное	Состою в браке	от 10000 до 20000 руб.	4506	2788
10654	Высшее	Состою в браке	от 10000 до 20000 руб.	2190	1367
10655	Среднее специальное	Вдовец/Вдова	от 10000 до 20000 руб.	4506	191

16.2.5. Кодирование вероятностями (Likelihood Encoding)

16.2.5.1. Общее знакомство

Теперь рассмотрим кодирование вероятностями зависимой переменной (Likelihood Encoding), которое обычно считается «чисто конкурсным» приемом, хотя это не так, при соблюдении аккуратности этот тип кодировки можно использовать и в рамках промышленного подхода. Существует множество способов такой кодировки.

Когда зависимая переменная Y является бинарной, мы можем сопоставить каждое индивидуальное значение X_i категориальной переменной X скаляру S_i , представляющему собой оценку вероятности $Y = 1$ для $X = X_i$:

$$X_i \rightarrow S_i \cong P(Y|X=X_i).$$

Проще говоря, мы можем представить каждую категорию признака как вероятность появления значения 1 зависимой переменной в этой категории (отсюда и название «кодирование вероятностями», или Likelihood Encoding).

Обратите внимание, что поскольку мы используем вычисления, то кодирование вероятностями нужно выполнять строго после разбиения на обучающую и тестовую выборки (внутри цикла перекрестной проверки). Как и в случае кодировки частотами, при кодировке вероятностями следует позаботиться о том, чтобы схема кодировки предусматривала возможность обработки новых категорий (обычно новые категории кодируют глобальным средним – средним значением зависимой переменной, вычисленным по обучающему набору).

Чтобы вычислить вероятность для каждого уровня, мы по каждой категории смотрим количество наблюдений, в которых зависимая переменная приняла значение 1, делим это количество на общее количество наблюдений в данной категории:

$$S_i = \frac{n_{iY}}{n_i},$$

где

n_{iY} – количество наблюдений i -й категории, в которых зависимая переменная приняла значение 1;

n_i – общее количество наблюдений i -й категории.

16.2.5.2. Кодирование простым средним значением зависимой переменной

Например, у нас есть признак *Class*. Он имеет категории: A, B и C. Категория A встречается в 4 наблюдениях. В трех из 4 наблюдений зависимая переменная принимает значение 1. 3 делим на 4, получаем 0,75. Вероятность для категории A будет равна 0,75. По сути, мы заменяем категорию признака средним значением зависимой переменной в этой категории, отсюда второе название данной кодировки – кодирование обычным средним значением зависимой переменной (Simple Mean Target Encoding).

Переменная Class	Переменная Class	Зависимая переменная	Кодировка Simple Mean Target Encoding для переменной Class
A	0,75 (3 из 4)	A	0,75
B	0,66 (2 из 3)	A	0,75
C	1,00 (2 из 2)	A	0,75
		B	0,66
		B	0,66
		B	0,66
		C	1,00
		C	1,00

Рис. 44 Simple Mean Target Encoding

Давайте напишем класс `TargetEncoder`, выполняющий кодирование простым средним значением зависимой переменной. Собственный метод `.fit_transform()` реализовывать не будем, а воспользуемся классом `TransformerMixin`. На этапе обучения класс создает таблицу, в соответствии с которой категориям признака в обучающей выборке будут сопоставлены средние значения зависимой переменной в этих категориях. На этапе применения происходит сопоставление. Для пропусков и новых категорий используем глобальные средние, вычисленные на этапе обучения модели.

```
# импортируем класс TransformerMixin
from sklearn.base import TransformerMixin

# пишем класс, выполняющий кодирование простым
# средним значением зависимой переменной
class TargetEncoder(TransformerMixin):
    """
    Автор: Dmitry Larko
    https://www.kaggle.com/dmitrylarko

    Параметры
    -----
    columns_names :
        Список признаков.
    """

    def __init__(self, columns_names):

        # инициализируем публичные атрибуты
        # список признаков, которые будем кодировать
        self.columns_names = columns_names

        # создаем пустой словарь, куда будем сохранять средние
        # значения зависимой переменной в каждой категории признака:
        # ключами в словаре будут названия признаков, а значениями
        # - таблицы, в которых напротив каждой категории признака
        # будет указано среднее значение зависимой переменной
        # в данной категории признака
        self.learned_values = {}

        # создадим переменную, в которой будем хранить глобальное
        # среднее (среднее значение зависимой переменной
```

```
# по обучающему набору)
self.dataset_mean = np.nan

# fit должен принимать в качестве аргументов только X и у
def fit(self, X, y=None):

    # выполняем копирование массива во избежание предупреждения
    # SettingWithCopyWarning "A value is trying to be set on a
    # copy of a slice from a DataFrame (Происходит попытка изменить
    # значение в копии среза данных датафрейма)"
    X_ = X.copy()

    # добавляем в новый массив признаков зависимую переменную
    # и называем ее __target__, именно эту переменную __target__
    # будем использовать в дальнейшем для вычисления среднего
    # значения зависимой переменной для каждой категории признака
    X_[ '__target__' ] = y

    # в цикле для каждого признака, который участвует в кодировании
    # (присутствует в списке self.columns_names):
    for c in [x for x in X_.columns if x in self.columns_names]:
        # формируем набор, состоящий из значений данного признака
        # и значений зависимой переменной
        # группируем данные по категориям признака, считаем среднее
        # значение зависимой переменной для каждой категории признака
        # и обновляем индекс
        self.learned_values[c] = (X_[[c, '__target__']])
            .groupby(c)[ '__target__' ].mean()
            .reset_index()

    # вычисляем среднее значение зависимой переменной по обучающему набору
    # и записываем в переменную dataset_mean, будем использовать
    # dataset_mean, если значение признака пропущено или в тестовом
    # наборе встретилась новая категория признака
    self.dataset_mean = np.mean(y)

    return self

# transform принимает в качестве аргумента только X
def transform(self, X):

    # скопирую массив признаков, значения которых будем кодировать,
    # этот массив будем изменять при вызове метода .transform(),
    # поэтому важно его скопировать, чтобы не изменить
    # исходный массив признаков
    transformed_X = X[self.columns_names].copy()

    # в цикле для каждого признака, который участвует в кодировании
    for c in transformed_X.columns:
        # формируем датафрейм, состоящий из значений данного признака,
        # и выполняем слияние с датафреймом, содержащим средние
        # значения зависимой переменной для каждой категории признака
        # нам нужны только средние значения зависимой переменной
        # для каждой категории признака, поэтому в датафрейме
        # оставляем лишь столбец '__target__'
        transformed_X[c] = (
            transformed_X[[c]].merge(self.learned_values[c],
                on=c, how='left'))['__target__']
```

```
# пропуски или новые категории признаков заменяем средним  
# значением зависимой переменной по обучающему набору  
transformed_X = transformed_X.fillna(self.dataset_mean)  
  
# возвращаем закодированный массив признаков  
return transformed_X
```

Давайте создадим обучающий и тестовый датафреймы pandas и применим наш класс `TargetEncoder`. При этом в тестовой выборке у нас будет новая категория (выделена красной рамкой).

```
# создаем обучающий датамфрейм pandas
train = pd.DataFrame({
    'Class': ['A', 'A', 'A', 'A', 'B', 'B', 'B', 'C', 'C'],
    'Response': [1, 0, 1, 1, 1, 1, 1, 0, 1]})
```

Class	Response	
0	A	1
1	A	0
2	A	1
3	A	1
4	B	1
5	B	1
6	B	0
7	C	1
8	C	1

```
# создаем тестовый датафрейм pandas
test = pd.DataFrame(
    {'Class': ['D', 'A', 'B', 'C', 'B', 'C']})
test
```

Class	
0	D
1	A
2	B
3	C
4	B
5	C

Class
0 0.750000
1 0.750000
2 0.750000
3 0.750000
4 0.666667
5 0.666667
6 0.666667
7 1.000000
8 1.000000

```
# применяем модель к тестовой выборке, категориям признака
# сопоставляем средние значения зависимой переменной
# в категориях признака, вычисленные на обучающей выборке
enc_test = te.transform(test)
enc_test
```

Class
0 0.777778
1 0.750000
2 0.666667
3 1.000000
4 0.666667
5 1.000000

К сожалению, часто категориальные переменные имеют редкие уровни, и, следовательно, оценка $P(Y|X = X_i)$ будет довольно ненадежной. Допустим, некоторый уровень встретился всего несколько раз, и в соответствующих наблюдениях зависимая переменная принимает значение 1. Тогда среднее значение зависимой переменной тоже будет единицей. При этом на тестовом наборе может возникнуть совсем другая ситуация. В нашем примере такой проблемной категорией является категория C, встречается она меньше остальных категорий, но при этом вероятность ее появления равна 1.

Переменная Class		Переменная Class	Зависимая переменная	Кодировка Simple Mean Target Encoding для переменной Class	
				Прогноз	0,75
A	0,75 (3 из 4)	A	1	0,75	
B	0,66 (2 из 3)	A	0	0,75	
C	1,00 (2 из 2)	A	1	0,75	Модель
		A	1	0,75	
		B	1	0,66	
		B	1	0,66	Модель
		B	0	0,66	
		C	1	1,00	
		C	1	1,00	Модель

Рис. 45 Проблемы Simple Mean Target Encoding – неправдоподобные оценки вероятности в редких категориях и «утечка»

Вычисление среднего значения зависимой переменной в категории признака – это модель. По сути, мы строим здесь три модели для трех категорий. При таком подходе «в лоб» получается, что для интересующего наблюдения мы делаем прогноз с помощью модели, которая строилась по наблюдениям, включая это интересующее наблюдение. Возникает «утечка».

16.2.5.3. Кодирование простым средним значением зависимой переменной по схеме leave-one-out

Можно попытаться избежать утечки, воспользовавшись схемой «leave-one-out». По каждому наблюдению вычисляем среднее значение зависимой переменной, используя все оставшиеся наблюдения данной категории, кроме него самого. Например, нам нужно вычислить значение для первого наблюдения, относящегося к категории A. Рассматриваем все наблюдения категории A, кроме первого. У нас три таких наблюдения. В двух наблюдениях из трех зависимая переменная принимает значение $1, 2 / 3 = 0,66$.

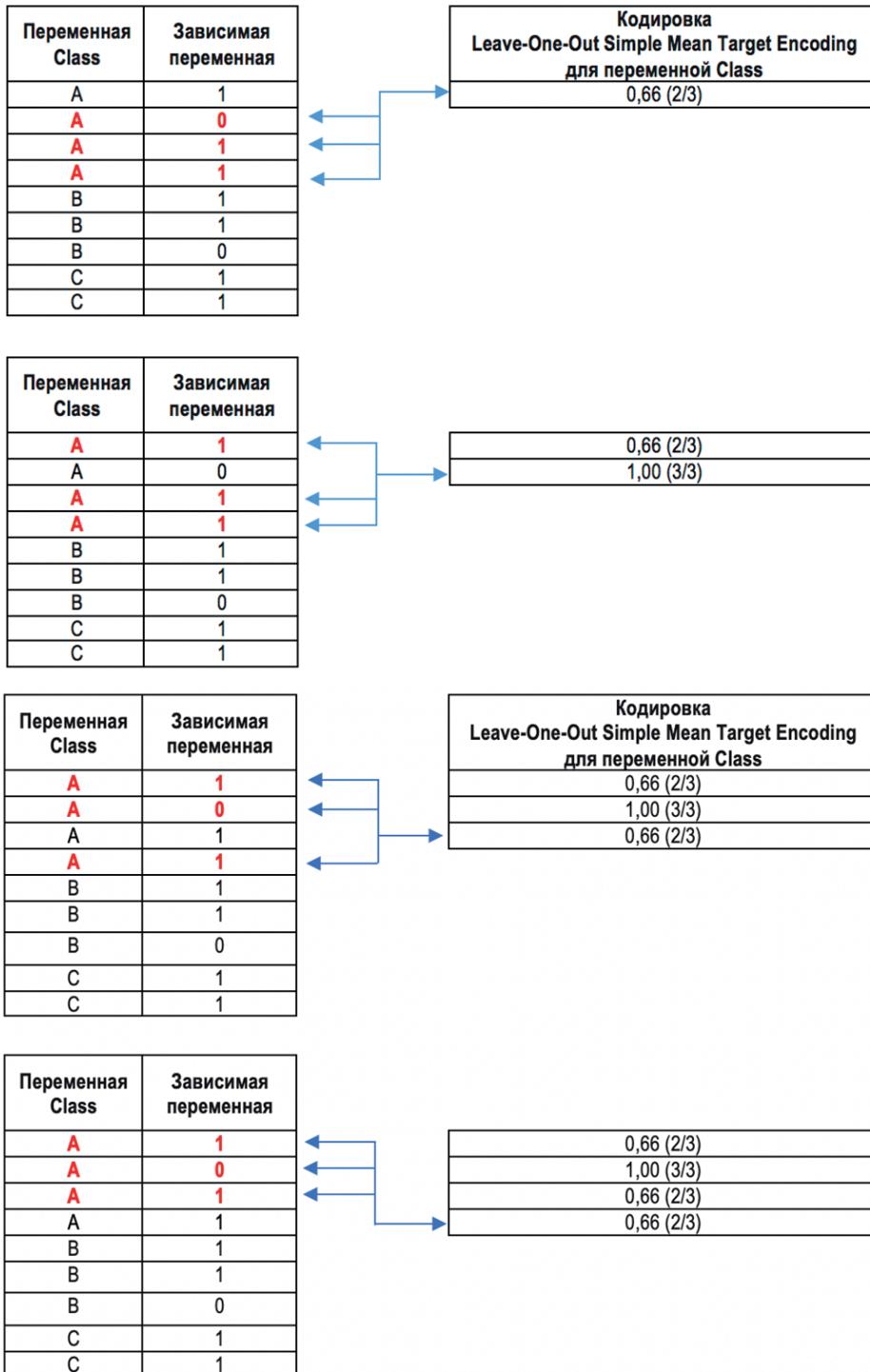


Рис. 46 Leave One Out Simple Mean Target Encoding

Класс `LeaveOneOutEncoder` пакета `category_encoders` позволяет выполнить кодирование простым средним значением зависимой переменной по схеме «leave-one-out». Пакет `category_encoders` можно установить с помощью команды `pip install category_encoders`.

```
# импортируем класс LeaveOneOutEncoder
from category_encoders import LeaveOneOutEncoder

# создаем экземпляр класса LeaveOneOutEncoder
loo_enc = LeaveOneOutEncoder()

# обучаем и применяем модель к обучающей выборке,
# т. е. для каждого признака создаем таблицу,
# в соответствии с которой категориям признака
# в обучающей выборке будут сопоставлены средние
# значения зависимой переменной в этих категориях,
# вычисленные по схеме loo, и сопоставляем
loo_encoded_train = loo_enc.fit_transform(
    train['Class'], train['Response'])
loo_encoded_train
```

Class
0 0.666667
1 1.000000
2 0.666667
3 0.666667
4 0.500000
5 0.500000
6 1.000000
7 1.000000
8 1.000000

Для категорий признака в тестовой выборке используем средние значения зависимой переменной в категориях признака, вычисленные на обучающей выборке.

```
# применяем модель к тестовой выборке, категориям признака
# сопоставляем средние значения зависимой переменной
# в категориях признака, вычисленные на обучающей выборке
loo_encoded_test = loo_enc.transform(test['Class'])
loo_encoded_test
```

Class
0 0.777778
1 0.750000
2 0.666667
3 1.000000
4 0.666667
5 1.000000

На практике оказалось, что схема «leave-one-out» совсем не препятствует утечке. Деревья решений с легкостью могут идеально классифицировать наблюдения обучающего набора, выполнив всего лишь одно разбиение.

16.2.5.4. Кодирование простым средним значением зависимой переменной по схеме «K-fold»

Сначала мы попробуем выполнить кодирование простыми средними значениями зависимой переменной и затем построить модель машинного обучения прямо внутри цикла перекрестной проверки.

Давайте загрузим обучающий и тестовый наборы и взглянем на них.

```
# загружаем данные
train = pd.read_csv('Data/Stat_FE2_train.csv', sep=';')
test = pd.read_csv('Data/Stat_FE2_test.csv', sep=';')
# выводим наблюдения обучающего набора
train
```

	living_region	job_position	open_account_flg
0	Московская область	Служащий	0
1	Московская область	Заместитель руководителя	1
2	Пермский край	Служащий	1
3	Московская область	Руководитель	0
4	Пермский край	Руководитель	1
5	Пермский край	Руководитель	1
6	Пермский край	Руководитель	1
7	Пермский край	Руководитель	0
8	Московская область	Заместитель руководителя	1
9	Московская область	Заместитель руководителя	0
10	Свердловская область	Заместитель руководителя	0
11	Свердловская область	Служащий	0
12	Свердловская область	Заместитель руководителя	0
13	Свердловская область	Заместитель руководителя	1
14	Свердловская область	Служащий	0

```
# выводим наблюдения тестового набора
test
```

	living_region	job_position	open_account_flg
0	Пермский край	Служащий	0
1	Московская область	Заместитель руководителя	0
2	Пермский край	Служащий	1
3	Московская область	Руководитель	0
4	Пермский край	Служащий	1
5	Пермский край	Руководитель	0
6	Пермский край	Руководитель	1
7	Московская область	Заместитель руководителя	1
8	Московская область	Заместитель руководителя	1
9	Свердловская область	Заместитель руководителя	0
10	Московская область	Заместитель руководителя	0
11	Свердловская область	Служащий	1

Для выполнения перекрестной проверки нам потребуется класс `StratifiedKFold`.

```
# импортируем класс StratifiedKFold
from sklearn.model_selection import StratifiedKFold
# создаем экземпляр класса StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# импортируем функции roc_auc_score() и clone()
from sklearn.metrics import roc_auc_score
from sklearn.base import clone
# импортируем класс GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingClassifier
# создаем экземпляр класса GradientBoostingClassifier
boost = GradientBoostingClassifier(random_state=42)
```

Теперь создадим список из признаков и массив меток.

```
# создаем список из двух признаков
cat_cols = ['living_region', 'job_position']
# создаем массив меток
y = train['open_account_flg'].values
```

Выполняем кодирование простыми средними значениями зависимой переменной и затем строим модель градиентного бустинга на каждой итерации цикла перекрестной проверки.

```
# создаем пустые списки, в которые
# будем записывать значения метрик
scores = []
tr_scores = []

# выполняем кодирование простым средним значением по схеме K-fold
for train_index, test_index in skf.split(train, y):
    train_df = train.loc[train_index, cat_cols].reset_index(drop=True)
    valid_df = train.loc[test_index, cat_cols].reset_index(drop=True)
```

```
train_y, valid_y = y[train_index], y[test_index]
te = TargetEncoder(columns_names=cat_cols)
X_tr = te.fit_transform(train_df, train_y).values
X_val = te.transform(valid_df).values
print(f"TRAIN:\n{np.column_stack((train_df, train_y))}\n")
print(f"VALID:\n{np.column_stack((valid_df, valid_y))}\n")
print(f"TRAIN encoded:\n{X_tr}\n")
print(f"VALID encoded:\n{X_val}\n")
print("")  
model = clone(boost)
model.fit(X_tr, train_y)

predictions = model.predict_proba(X_val)[:, 1]
scores.append(roc_auc_score(valid_y, predictions))

train_preds = model.predict_proba(X_tr)[:, 1]
tr_scores.append(roc_auc_score(train_y, train_preds))

print("Train AUC score: {:.4f} Valid AUC score: {:.4f}, STD: {:.4f}".format(
    np.mean(tr_scores), np.mean(scores), np.std(scores)
))
)
```

Для экономии выведем происходящее в первой итерации перекрестной проверки и значения метрик.

```
TRAIN:  
[['Московская область' 'Заместитель руководителя' 1]  
 ['Московская область' 'Руководитель' 0]  
 ['Пермский край' 'Руководитель' 1]  
 ['Пермский край' 'Руководитель' 1]  
 ['Пермский край' 'Руководитель' 1]  
 ['Московская область' 'Заместитель руководителя' 1]  
 ['Московская область' 'Заместитель руководителя' 0]  
 ['Свердловская область' 'Заместитель руководителя' 0]  
 ['Свердловская область' 'Служащий' 0]  
 ['Свердловская область' 'Заместитель руководителя' 0]  
 ['Свердловская область' 'Заместитель руководителя' 1]  
 ['Свердловская область' 'Служащий' 0]]
```

```
TRAIN index:  
[ 1  3  4  5  6  8  9 10 11 12 13 14]
```

```
VALID:  
[['Московская область' 'Служащий' 0]  
 ['Пермский край' 'Служащий' 1]  
 ['Пермский край' 'Руководитель' 0]]
```

```
VALID index:  
[0 2 7]
```

```
TRAIN encoded:  
[[0.5 0.5 ]  
 [0.5 0.75]  
 [1. 0.75]  
 [1. 0.75]  
 [1. 0.75]  
 [0.5 0.5 ]
```

```
[0.5  0.5 ]
[0.2  0.5 ]
[0.2  0. ]
[0.2  0.5 ]
[0.2  0.5 ]
[0.2  0. ]]
```

```
VALID encoded:
[[0.5  0. ]
 [1.   0. ]
 [1.   0.75]]
```

Видим, что значения признаков для обучающей выборки перекрестной проверки – это простые средние значения зависимой переменной в категориях признаков. Например, возьмем категорию 'Московская область'. Она встречается 4 раза, в двух наблюдениях из четырех зависимая переменная принимает значение 1, $2 / 4 = 0.5$. Значения признаков в проверочной выборке перекрестной проверки – это обычные средние значения зависимой переменной в категориях признаков, вычисленные на обучающей выборке. Проблема «утечки» никуда не исчезла.

Теперь разберем схему «K-fold». Мы используем перекрестную проверку, например разбиваем набор на 5 блоков: блоки 1, 2, 3, 4, 5. Вычисляем среднее значение зависимой переменной в каждой категории признака по каждому блоку перекрестной проверки. Для этого используем наблюдения, относящиеся к данной категории и не попавшие в данный блок.

Например, вычисляем среднее значение зависимой переменной в категории В в блоке 1 (выделены черной рамкой). Для вычислений используем только наблюдения категории В в блоках с 2 по 5 (отмечены красным жирным шрифтом).

Переменная Class	Зависимая переменная	Блок	Кодировка K-fold Simple Mean Target Encoding для переменной Class
A	1	5	
A	1	3	
B	0	2	
B	0	1	
A	1	1	
B	0	2	
B	0	1	
A	1	4	0,4 (2 из 5)
A	0	4	
C	1	4	
C	0	4	
B	1	4	
A	1	2	
A	0	5	
B	1	4	0,4 (2 из 5)
A	1	5	
C	0	3	
A	1	4	
B	0	2	
B	1	1	0,4 (2 из 5)

Рис. 47 K-fold Simple Mean Target Encoding

Аналогично поступаем для каждой категории в каждом блоке и строим модель машинного обучения.

Попробуем реализовать эту схему. Давайте вычислим глобальное среднее – среднее значение зависимой переменной в обучающей выборке.

```
# вычисляем глобальное среднее – среднее значение  
# зависимой переменной в обучающей выборке  
glob_mean = y.mean()  
glob_mean
```

```
0.4666666666666667
```

Создаем копию обучающей выборки и для удобства дальнейшей визуализации сокращаем имена столбцов, категорию сокращаем до первой буквы в названии категории, создаем список категориальных столбцов.

```
# создаем копию обучающей выборки  
new_train = train.copy()  
# сокращаем имена столбцов  
new_train.rename(  
    columns={'living_region': 'reg',  
             'job_position': 'job'},  
    inplace=True)  
# категорией будет первая буква в названии категории  
new_train['reg'] = new_train['reg'].str[0]  
new_train['job'] = new_train['job'].str[0]  
  
# задаем список категориальных признаков  
cat_cols = ['reg', 'job']  
new_train
```

	reg	job	open_account_flg
0	M	C	0
1	M	3	1
2	П	C	1
3	M	P	0
4	П	P	1
5	П	P	1
6	П	P	1
7	П	P	0
8	M	3	1
9	M	3	0
10	C	3	0
11	C	C	0
12	C	3	0
13	C	3	1
14	C	C	0

Теперь для каждого категориального признака создаем столбец, каждое значение которого – глобальное среднее.

```
# для каждого категориального признака создаем столбец,
# каждое значение которого – глобальное среднее
for col in cat_cols:
    new_train[col + '_mean_target'] = [glob_mean for _ in range(
        new_train.shape[0])]

new_train
```

	reg	job	open_account_flg	reg_mean_target	job_mean_target
0	M	C	0	0.466667	0.466667
1	M	Z	1	0.466667	0.466667
2	P	C	1	0.466667	0.466667
3	M	P	0	0.466667	0.466667
4	P	P	1	0.466667	0.466667
5	P	P	1	0.466667	0.466667
6	P	P	1	0.466667	0.466667
7	P	P	0	0.466667	0.466667
8	M	Z	1	0.466667	0.466667
9	M	Z	0	0.466667	0.466667
10	C	Z	0	0.466667	0.466667
11	C	C	0	0.466667	0.466667
12	C	Z	0	0.466667	0.466667
13	C	Z	1	0.466667	0.466667
14	C	C	0	0.466667	0.466667

Отключаем предупреждения и запускаем перекрестную проверку. Мы вычисляем среднее значение зависимой переменной для категории признака, попавшей в проверочную выборку перекрестной проверки, используя данные вне этой выборки (т. е. используя данные обучающей выборки). Например, мы используем 5-блочную перекрестную проверку, и нам нужно вычислить среднее значение зависимой переменной для категории A, попавшей в проверочную выборку. Для вычисления этого среднего значения используем лишь наблюдения категории A в обучающей выборке перекрестной проверки. Если вместо категорий у нас значения NaN, заменяем глобальным средним. В итоге получаем новую обучающую выборку.

```
# отключаем предупреждения Anaconda
import warnings
warnings.filterwarnings('ignore')
```

```

# вычисляем среднее значение зависимой переменной для категории
# признака в проверочной выборке перекрестной проверки,
# используя данные вне этой выборки (т. е. используя данные
# обучающей выборки)
# например, мы используем 5-блочную перекрестную проверку
# и нам нужно вычислить среднее значение зависимой переменной
# для категории A, попавшей в проверочную выборку, для вычисления
# этого среднего значения используются лишь наблюдения категории A
# в обучающей выборке, если вместо категорий у нас
# значения NaN, заменяем глобальным средним, в итоге
# получаем новую обучающую выборку
for cnt, (train_idx, valid_idx) in enumerate(
    skf.split(new_train, new_train['open_account_flg']), 1):
    X_train_cv, X_valid_cv = (new_train.iloc[train_idx, :],
                               new_train.iloc[valid_idx, :])
    print(f'{cnt}-я итерация\n')
    print(f"TRAIN:\n{X_train_cv}\n")

    for col in cat_cols:
        means = X_valid_cv[col].map(
            X_train_cv.groupby(col)['open_account_flg'].mean())
        X_valid_cv[col + '_mean_target'] = means.fillna(glob_mean)
    print(f"VALID encoded:\n{X_valid_cv}\n")
    new_train.iloc[valid_idx] = X_valid_cv
    print(f"new_train:\n{new_train}\n")

```

Давайте подробно разберем происходящее под капотом. Начинаем с первой итерации.

1-я итерация

TRAIN:

	reg	job	open_account_flg	reg_mean_target	job_mean_target
1	M	3		1	0.466667
3	M	P		0	0.466667
4	Π	P		1	0.466667
5	Π	P		1	0.466667
6	Π	P		1	0.466667
8	M	3		1	0.466667
9	M	3		0	0.466667
10	C	3		0	0.466667
11	C	C		0	0.466667
12	C	3		0	0.466667
13	C	3		1	0.466667
14	C	C		0	0.466667

VALID encoded:

	reg	job	open_account_flg	reg_mean_target	job_mean_target
0	M	C		0	0.5
2	Π	C		1	1.0
7	Π	P		0	1.0

new_train:

	reg	job	open_account_flg	reg_mean_target	job_mean_target
0	M	C		0	0.500000
1	M	3		1	0.466667
2	Π	C		1	1.000000

3	M	P	0	0.466667	0.466667
4	P	P	1	0.466667	0.466667
5	P	P	1	0.466667	0.466667
6	P	P	1	0.466667	0.466667
7	P	P	0	1.000000	0.750000
8	M	3	1	0.466667	0.466667
9	M	3	0	0.466667	0.466667
10	C	3	0	0.466667	0.466667
11	C	C	0	0.466667	0.466667
12	C	3	0	0.466667	0.466667
13	C	3	1	0.466667	0.466667
14	C	C	0	0.466667	0.466667

В разделе TRAIN записана исходная обучающая выборка. Здесь мы должны обратить внимание на значения в двух крайних столбцах справа. Это просто глобальные средние.

В разделе VALID encoded находятся наблюдения, попавшие в проверочную выборку. Категории признаков, попавшие в проверочную выборку, закодированы средними значениями зависимой переменной в этих категориях, вычисленными на обучающей выборке. Например, для категории 'M' переменной *reg* вычислено значение 0,5. Обращаемся к обучающей выборке. Категория 'M' переменной *reg* встретилась в 4 наблюдениях, в двух из 4 наблюдений зависимая переменная принимает значение 1, $2 / 4 = 0,5$. Для категории 'P' переменной *job* вычислено значение 0,5. Обращаемся к обучающей выборке. Категория 'P' переменной *job* встретилась в 4 наблюдениях, в трех из 4 наблюдений зависимая переменная принимает значение 1, $3 / 4 = 0,75$. Теперь вносим изменения в нашу обучающую выборку. В двух крайних столбцах справа в наблюдениях, соответствующих индексам наблюдений проверочной выборки, глобальные средние заменяем на значения, полученные для этих наблюдений (берем их из раздела VALID encoded).

Переходим ко второй итерации.

2-я итерация

TRAIN:

	reg	job	open_account_flg	reg_mean_target	job_mean_target
0	M	C	0	0.500000	0.000000
1	M	3	1	0.466667	0.466667
2	P	C	1	1.000000	0.000000
3	M	P	0	0.466667	0.466667
4	P	P	1	0.466667	0.466667
5	P	P	1	0.466667	0.466667
6	P	P	1	0.466667	0.466667
7	P	P	0	1.000000	0.750000
9	M	3	0	0.466667	0.466667
11	C	C	0	0.466667	0.466667
13	C	3	1	0.466667	0.466667
14	C	C	0	0.466667	0.466667

VALID encoded:

	reg	job	open_account_flg	reg_mean_target	job_mean_target
8	M	3	1	0.250000	0.666667
10	C	3	0	0.333333	0.666667
12	C	3	0	0.333333	0.666667

```
new_train:  
    reg job open_account_flg reg_mean_target job_mean_target  
0     M   C             0      0.500000      0.000000  
1     M   3             1      0.466667      0.466667  
2     П   C             1      1.000000      0.000000  
3     M   P             0      0.466667      0.466667  
4     П   P             1      0.466667      0.466667  
5     П   P             1      0.466667      0.466667  
6     П   P             1      0.466667      0.466667  
7     П   P             0      1.000000      0.750000  
8     M   3             1      0.250000      0.666667  
9     M   3             0      0.466667      0.466667  
10    C   3             0      0.333333      0.666667  
11    C   C             0      0.466667      0.466667  
12    C   3             0      0.333333      0.666667  
13    C   3             1      0.466667      0.466667  
14    C   C             0      0.466667      0.466667
```

В разделе TRAIN записана модифицированная обучающая выборка. Мы обращаем внимание на значения в двух крайних столбцах справа. В них наряду с глобальными средними записаны закодированные значения для наблюдений, попавших в проверочную выборку на прошлой итерации.

В разделе VALID encoded находятся наблюдения, попавшие в проверочную выборку. Вновь категории признаков, попавшие в проверочную выборку, закодированы средними значениями зависимой переменной в этих категориях, вычисленными на обучающей выборке. На основе этих наблюдений вносим изменения в нашу обучающую выборку. В двух крайних столбцах справа в наблюдениях, соответствующих индексам наблюдений проверочной выборки, глобальные средние заменяем на значения, полученные для этих наблюдений (берем их из раздела VALID encoded).

В итоге после пяти итераций получаем новую обучающую выборку.

5-я итерация

```
TRAIN:  
    reg job open_account_flg reg_mean_target job_mean_target  
0     M   C             0      0.500000      0.000000  
2     П   C             1      1.000000      0.000000  
3     M   P             0      0.500000      0.666667  
4     П   P             1      0.666667      0.333333  
5     П   P             1      0.666667      0.333333  
6     П   P             1      0.750000      0.666667  
7     П   P             0      1.000000      0.750000  
8     M   3             1      0.250000      0.666667  
10    C   3             0      0.333333      0.666667  
11    C   C             0      0.250000      0.333333  
12    C   3             0      0.333333      0.666667  
14    C   C             0      0.250000      0.333333
```

```
VALID encoded:  
    reg job open_account_flg reg_mean_target job_mean_target  
1     M   3             1      0.333333      0.333333  
9     M   3             0      0.333333      0.333333  
13    C   3             1      0.000000      0.333333
```

```
new_train:
    reg job open_account_flg reg_mean_target job_mean_target
0     M   C             0      0.500000      0.000000
1     M   3             1      0.333333      0.333333
2     П   C             1      1.000000      0.000000
3     M   P             0      0.500000      0.666667
4     П   P             1      0.666667      0.333333
5     П   P             1      0.666667      0.333333
6     П   P             1      0.750000      0.666667
7     П   P             0      1.000000      0.750000
8     M   3             1      0.250000      0.666667
9     M   3             0      0.333333      0.333333
10    C   3             0      0.333333      0.666667
11    C   C             0      0.250000      0.333333
12    C   3             0      0.333333      0.666667
13    C   3             1      0.000000      0.333333
14    C   C             0      0.250000      0.333333
```

Удаляем из новой обучающей выборки зависимую переменную и по желанию исходные категориальные признаки.

```
# удаляем из новой обучающей выборки зависимую переменную
# и по желанию исходные категориальные признаки
new_train.drop(cat_cols + ['open_account_flg'],
               axis=1, inplace=True)
```

new_train

	reg_mean_target	job_mean_target
0	0.500000	0.000000
1	0.333333	0.333333
2	1.000000	0.000000
3	0.500000	0.666667
4	0.666667	0.333333
5	0.666667	0.333333
6	0.750000	0.666667
7	1.000000	0.750000
8	0.250000	0.666667
9	0.333333	0.333333
10	0.333333	0.666667
11	0.250000	0.333333
12	0.333333	0.666667
13	0.000000	0.333333
14	0.250000	0.333333

На основе полученной обучающей выборки можно строить модель машинного обучения.

Обратите внимание, что перекрестную проверку мы применяем только для обучающей выборки, для тестовой выборки мы используем обычные средние

значения зависимой переменной в соответствующих категориях признака, вычисленные для обучающей выборки.

Если качество на тестовой выборке нас устраивает, нужно построить модель на всем историческом наборе. Мы должны заново вычислить средние значения зависимой переменной по схеме «k-fold» для категорий признака на всем историческом наборе.

Для категорий признака в новых данных мы используем обычные средние значения зависимой переменной в соответствующих категориях признака, вычисленные на всей исторической выборке.

16.2.5.5. Кодирование средним значением зависимой переменной, сглаженным через сигмоиду

Чтобы сгладить эффект редких уровней и избежать утечек, мы можем вычислить оценку вероятности S_i для каждого уровня признака как смесь двух вероятностей: апостериорной вероятности Y для $X = X_i$, вычисляемой с помощью уравнения $S_i = n_{iY}/n_i$, и априорной вероятности Y (базовой вероятности, получаемой по всему обучающему набору). Эти две вероятности «смешиваются» с помощью весового коэффициента, который является функцией от размера категории признака. Вычисляемое среднее значение называют средним значением, сглаженным через сигмоидальную функцию.

Данный подход изложил в 2001 году Дэниэл Миччи-Баррека в своей статье «A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems» («Схема предварительной обработки категориальных признаков с большим количеством уровней для задач классификации и прогнозирования»).

Ниже приведена формула, предложенная им.

$$S_i = \lambda(n_i) \frac{n_{iY}}{n_i} + (1 - \lambda(n_i)) \frac{n_Y}{n_{TR}}$$

где

n_{iY} – количество наблюдений i -й категории признака, в которых зависимая переменная приняла значение 1;

n_i – общее количество наблюдений i -й категории признака;

n_Y – количество наблюдений обучающего набора данных, в которых зависимая переменная приняла значение 1;

n_{TR} – общее количество наблюдений обучающего набора данных;

$\lambda(n_i)$ – весовой коэффициент, представляет собой монотонно возрастающую функцию от n_i , ограниченную диапазоном значений от 0 до 1.

Эту формулу часто сводят к более простой формуле:

$$S_{level} = \lambda(n_{level}) \times mean(level) + (1 - \lambda(n_{level})) \times mean(dataset).$$

Обоснование данной формулы сводится к тому, что если размер категории является большим, мы должны больше доверять оценке апостериорной вероятности. Закодированное значение уровня будет ближе к среднему значению зависимой переменной в данном уровне. Однако если размер категории мал, мы должны ориентироваться на априорную вероятность. Закодированное значение уровня будет ближе к среднему значению зависимой переменной в обучающем наборе (глобальному среднему).

Обычно весовой коэффициент $\lambda(n_i)$ задается как функция с одной или несколькими настройками и может быть оптимизирован на основе характеристик данных. Например, коэффициент $\lambda(n_i)$ может быть определен как

$$\lambda(n_i) = \frac{1}{1 + e^{-\frac{(n_i - k)}{f}}},$$

где $\lambda(n_i)$ является сигмоидной функцией, которая предполагает значение 0,5 для $n_i = k$.

Здесь n_i – частота i -й категории (например, частота для категории А будет равна 4, для категории В – 3, для категории С – 2). Параметр k – точка перегиба сигмоиды, используемая для смешивания вероятностей. Он будет соответствовать половине минимально допустимого размера категории, при которой мы полностью «доверяем» апостериорной вероятности (среднему значению зависимой переменной в категории признака). Если k становится больше размера категории, влияние апостериорной вероятности уменьшается, а влияние априорной вероятности возрастает. Позже мы проиллюстрируем это подробнее.

Параметр f – обратная величина наклона сигмоиды в точке перегиба. Она определяет соотношение между апостериорной и априорной вероятностями. По мере стремления f к 0 сигмоида превращается в ступенчатую функцию Хевисайда.

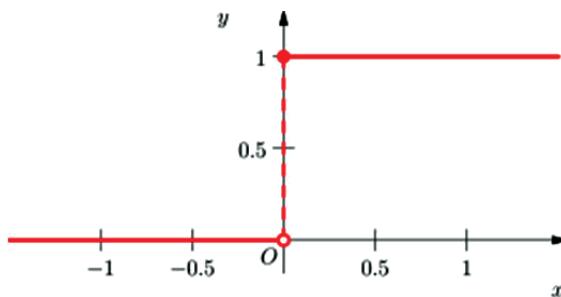


Рис. 48 Ступенчатая функция Хевисайда. Источник: Википедия

Давайте напишем собственный класс `SmoothingTargetEncoder`, выполняющий кодирование средним значением зависимой переменной, слаженным через сигмоиду. Собственный метод `.fit_transform()` реализовывать не буд-

дем, а воспользуемся классом `TransformerMixin`. Класс позволяет настраивать значения параметров k и f . У метода `.transform()` есть параметр `smoothing`. Если для него задано значение `smoothing=True`, вычисляются сглаженные по сигмоиде средние значения зависимой переменной в категориях признаков. Если для него задано значение `smoothing=False`, вычисляются обычные средние значения зависимой переменной в категориях признаков. Для пропусков и новых категорий используем глобальные средние, вычисленные на этапе обучения модели. Более конкретно: пропуски в обучающей выборке / историческом наборе мы заменяем глобальным средним, вычисленным в обучающей выборке / историческом наборе. Пропуски в проверочной выборке / тестовой выборке / наборе новых данных мы заменяем глобальным средним, вычисленным в обучающей выборке / обучающей-проверочной выборке / историческом наборе. Новые категории в тестовой выборке / наборе новых данных заменяем глобальным средним, вычисленным в обучающей выборке / историческом наборе.

```
# пишем класс SmoothingTargetEncoder, выполняющий кодирование
# средним значением зависимой переменной, сглаженным
# через сигмоиду
class SmoothingTargetEncoder(TransformerMixin):
    """
```

Автор: Dmitry Larko
<https://www.kaggle.com/dmitrylarko>

Параметры

`columns_names: list`

Список признаков.

`k: int`, значение по умолчанию 2

Половина минимально допустимого размера категории,
при которой мы полностью «доверяем» апостериорной
вероятности.

`f: float`, значение по умолчанию 0.25

Угол наклона сигмоиды (определяет соотношение между
апостериорной и априорной вероятностями).

"""

```
def __init__(self, columns_names, k=2, f=0.25):
```

`# инициализируем публичные атрибуты`

`# список признаков, которые будем кодировать`
`self.columns_names = columns_names`

`# половина минимально допустимого размера категории,`
`# при которой мы полностью «доверяем»`
`# апостериорной вероятности`
`self.k = k`

`# угол наклона сигмоиды (определяет соотношение между`
`# апостериорной и априорной вероятностями)`
`self.f = f`

```

# создаем пустой словарь, куда будем сохранять обычные
# средние значения зависимой переменной в каждой
# категории признака:
# ключами в словаре будут названия признаков, а значениями
# - таблицы, в которых напротив каждой категории признака
# будет указано среднее значение зависимой переменной
# в данной категории признака
self.simple_values = {}

# создаем пустой словарь, куда будем сохранять сглаженные
# средние значения зависимой переменной в каждой
# категории признака:
# ключами в словаре будут названия признаков, а значениями
# - таблицы, в которых напротив каждой категории признака
# будет указано сглаженное среднее значение зависимой
# переменной в данной категории признака
self.smoothing_values = {}

# создадим переменную, в которой будем хранить глобальное среднее
# (среднее значение зависимой переменной по обучающему набору)
self.dataset_mean = np.nan

def smoothing_func(self, N):
    # метод задает весовой коэффициент - сигмоиду для сглаживания
    return 1 / (1 + np.exp(-(N-self.k) / self.f))

# fit должен принимать в качестве аргументов только X и y
def fit(self, X, y=None):

    # выполняем копирование массива во избежание предупреждения
    # SettingWithCopyWarning "A value is trying to be set on a
    # copy of a slice from a DataFrame (Происходит попытка изменить
    # значение в копии среза данных датафрейма)"
    X_ = X.copy()

    # создадим переменную, в которой будем хранить глобальное среднее
    # (среднее значение зависимой переменной по обучающему набору)
    self.dataset_mean = np.mean(y)

    # добавляем в новый массив признаков зависимую переменную
    # и называем ее __target__, именно эту переменную __target__
    # будем использовать в дальнейшем для вычисления среднего значения
    # зависимой переменной для каждой категории признака
    X_[ '__target__' ] = y

    # в цикле для каждого признака, который участвует в кодировании
    # (присутствует в списке self.columns_names)
    for c in [x for x in X_.columns if x in self.columns_names]:
        # формируем набор, состоящий из значений данного признака
        # и значений зависимой переменной
        # группируем данные по категориям признака, считаем среднее
        # значение зависимой переменной для каждой категории
        # признака и обновляем индекс
        stats = (X_[[c, '__target__']]
                 .groupby(c)[ '__target__' ]
                 .agg(['mean', 'size'])
                 .reset_index())

```

```

# вычисляем весовой коэффициент alpha, который "перемешает"
# среднее значение зависимой переменной в категории признака
# и глобальное среднее, вычисленное по обучающему набору
stats['alpha'] = self.smoothing_func(stats['size'])

# вычисляем сглаженное среднее значение зависимой переменной
# для категории признака согласно формуле
stats['__smooth_target__'] = (stats['alpha'] * stats['mean']
                               + (1 - stats['alpha']) *
                               self.dataset_mean)

# вычисляем обычное среднее значение зависимой переменной
# для категории признака
stats['__simple_target__'] = stats['mean']

# формируем набор, состоящий из признака, сглаженных средних
# значений зависимой переменной для категорий признака и обычных
# средних значений зависимой переменной для категорий признака
stats = (stats.drop([x for x in stats.columns
                     if x not in ['__smooth_target__',
                                   '__simple_target__', c]],
                     axis=1).reset_index())

# сохраним сглаженные средние значения зависимой переменной
# для каждой категории признака в словарь
self.smoothing_values[c] = stats[[c, '__smooth_target__']]
# сохраним обычные средние значения зависимой переменной
# для каждой категории признака в словарь
self.simple_values[c] = stats[[c, '__simple_target__']]

return self

# transform принимает в качестве аргумента только X
def transform(self, X, smoothing=True):
    # скопируем массив данных с признаками, значения которых
    # будем кодировать, этот массив будем изменять при вызове
    # метода .transform(), поэтому важно его скопировать,
    # чтобы не изменить исходный массив признаков
    transformed_X = X[self.columns_names].copy()

    if smoothing:
        # в цикле для каждого признака, который
        # участвует в кодировании
        for c in transformed_X.columns:
            # формируем датафрейм, состоящий из значений данного признака,
            # и выполняем слияние с датафреймом, содержащим сглаженные
            # средние для каждой категории признака, нам нужны только
            # сглаженные средние для каждой категории признака, поэтому
            # в датафрейме оставляем лишь столбец '__smooth_target__'
            transformed_X[c] = (transformed_X[[c]]
                               .merge(self.smoothing_values[c],
                                      on=c, how='left')
                               )['__smooth_target__']

    else:
        # в цикле для каждого признака, который
        # участвует в кодировании

```

```

for c in transformed_X.columns:
    # формируем датафрейм, состоящий из значений данного
    # признака, и выполняем слияние с датафреймом, содержащим
    # обычные средние значения зависимой переменной
    # для каждой категории признака
    # нам нужны только обычные средние значения зависимой
    # переменной для каждой категории признака, поэтому в
    # датафрейме оставляем лишь столбец '__simple_target__'
    transformed_X[c] = (transformed_X[[c]]
        .merge(self.simple_values[c],
            on=c, how='left')
    )['__simple_target__']

# пропуски или новые категории признаков заменяем средним
# значением зависимой переменной по обучающему набору
transformed_X = transformed_X.fillna(self.dataset_mean)

# возвращаем закодированный массив признаков
return transformed_X

```

Проиллюстрируем, как меняется сигмоида в зависимости от значений параметров k и f . Сначала посмотрим, как сигмоида изменяется в зависимости от разных значений параметра f .

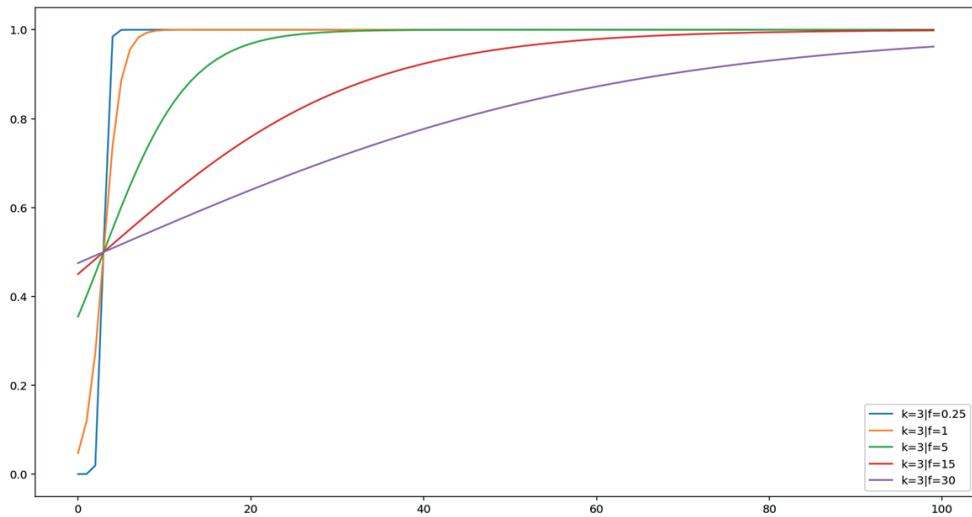
```

# включаем режим 'retina', если у вас экран Retina
%config InlineBackend.figure_format = 'retina'

# создаем массив значений
x = np.linspace(0, 100, 100)

# меняем значения f
plot = pd.DataFrame()
te = SmoothingTargetEncoder([], 3, 0.25)
plot["k=3|f=0.25"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 3, 1)
plot["k=3|f=1"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 3, 5)
plot["k=3|f=5"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 3, 15)
plot["k=3|f=15"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 3, 30)
plot["k=3|f=30"] = te.smoothing_func(x)
plot.plot(figsize=(15,8));

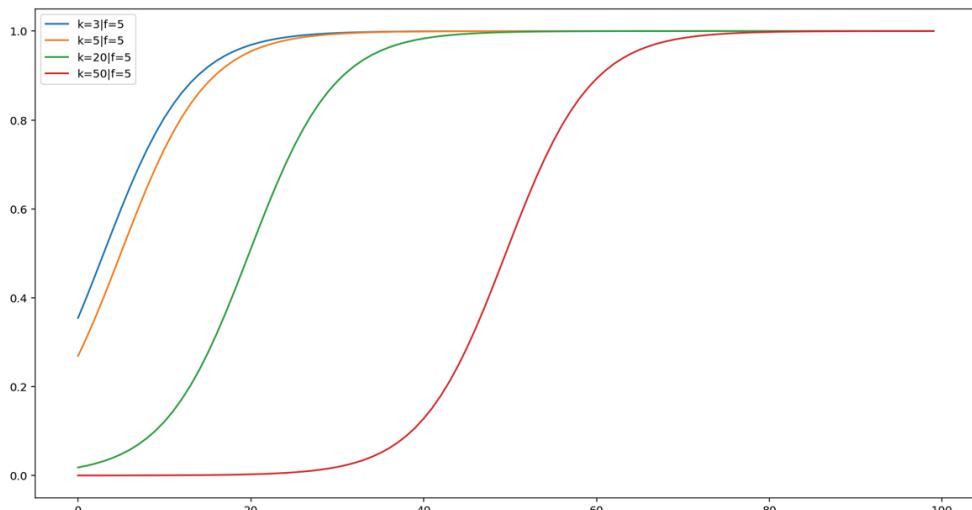
```



Убеждаемся, что параметр f контролирует крутизну наклона сигмоиды. Маленькие значения f дают более крутой наклон.

Теперь посмотрим, как сигмоида изменяется в зависимости от разных значений параметра k .

```
# меняем значения k
plot = pd.DataFrame()
te = SmoothingTargetEncoder([], 3, 5)
plot["k=3|f=5"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 5, 5)
plot["k=5|f=5"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 20, 5)
plot["k=20|f=5"] = te.smoothing_func(x)
te = SmoothingTargetEncoder([], 50, 5)
plot["k=50|f=5"] = te.smoothing_func(x)
plot.plot(figsize=(15,8));
```



Параметр k контролирует перемещение точки перегиба относительно оси абсцисс. Большие значения k приводят к тому, что точка перегиба появляется правее.

Теперь выясним, как k и n работают все вместе при неизменном значении f . Итак, k – это точка, где вес категории равен 0,5. Например, мы задали $k = 2$, а размер редкой категории, для которой нужно скорректировать оценку (n_{rare}), равен 2. Соответственно, когда $k = 2$ и $n_{rare} = 2$, коэффициент $\lambda(n_{rare})$ будет равен 0,5, т. е. 0,5 * среднее значение зависимой переменной по категории + 0,5 * среднее значение зависимой переменной по обучающему набору. Мы в равной степени доверяем апостериорной и априорной вероятностям. Когда $k = 1$ и $n_{rare} = 2$, то коэффициент $\lambda(n_{rare})$ перед средним значением зависимой переменной по категории увеличится (степень увеличения зависит от f), а перед средним значением зависимой переменной по обучающему набору уменьшится. Теперь мы в большей степени будем доверять апостериорной вероятности. Когда $k = 3$ и $n_{rare} = 2$, то коэффициент $\lambda(n_{rare})$ перед средним значением зависимой переменной по категории уменьшится, а перед средним значением зависимой переменной по набору увеличится. Теперь мы в большей степени будем доверять априорной вероятности, что, собственно, нам и нужно.

Поясним на примере, с которого мы начинали объяснение кодирования вероятностями.

Переменная Class	
A	0,75 (3 из 4)
B	0,66 (2 из 3)
C	1,00 (2 из 2)

Переменная Class	Зависимая переменная	Кодировка Simple Mean Target Encoding для переменной Class
A	1	0,75
A	0	0,75
A	1	0,75
A	1	0,75
B	1	0,66
B	1	0,66
B	0	0,66
C	1	1,00
C	1	1,00

Мы выполним кодирование для категорий A, B и C, при этом сохраним значение параметра f неизменным (пусть оно будет равно 0,25), значение параметра k будем увеличивать с 1 до 3.

Установим k равным 1.

$$\lambda = \frac{1}{1 + \exp^{-\frac{(n_{rare}-1)}{0,25}}}$$

	n	$mean(level)$	$mean(dataset)$	λ	$1 - \lambda$	Результат кодировки
A	4	0,75	0,778	0,99	0,01	$0,99 \cdot 0,75 + 0,01 \cdot 0,778 = 0,75$
B	3	0,66	0,778	0,99	0,01	$0,99 \cdot 0,66 + 0,01 \cdot 0,778 = 0,6667$
C	2	1,00	0,778	0,98	0,02	$0,98 \cdot 1,0 + 0,02 \cdot 0,778 = 0,996$

При первом варианте кодировки для категории C вес априорной вероятности резко падает, и мы больше ориентируемся на апостериорную вероятность, т. е. на среднее значение зависимой переменной в данной категории признака. Как результат закодированное значение признака будет находиться ближе к среднему значению зависимой переменной в данной категории признака.

Установим k равным 2.

$$\lambda = \frac{1}{1 + \exp^{-\frac{(n_{rare}-2)}{0,25}}}$$

	n	$mean(level)$	$mean(dataset)$	λ	$1 - \lambda$	Результат кодировки
A	4	0,75	0,778	0,99	0,01	$0,99 \cdot 0,75 + 0,01 \cdot 0,778 = 0,75$
B	3	0,66	0,778	0,98	0,02	$0,98 \cdot 0,66 + 0,02 \cdot 0,778 = 0,6687$
C	2	1,00	0,778	0,5	0,5	$0,5 \cdot 1,0 + 0,5 \cdot 0,778 = 0,8889$

При втором варианте вес апостериорной вероятности и вес априорной вероятности будут одинаковы. Здесь закодированное значение признака представляет собой среднее значение двух средних – среднего значения зависимой переменной в данной категории признака и среднего значения зависимой переменной в обучающем наборе.

Установим k равным 3.

$\lambda = \frac{1}{1 + exp^{-\frac{(n_{rare}-3)}{0,25}}}$	n	$mean(level)$	$mean(dataset)$	λ	$1 - \lambda$	Результат кодировки
	A 4	0,75	0,778	0,98	0,02	$0,98*0,75 + 0,02*0,778 = 0,7505$
	B 3	0,66	0,778	0,5	0,5	$0,5*0,66 + 0,5*0,778 = 0,7222$
	C 2	1,00	0,778	0,018	0,982	$0,018*1,0 + 0,982*0,778 = 0,7818$

При третьем варианте кодировки вес апостериорной вероятности резко падает, и мы больше ориентируемся на априорную вероятность, т. е. на среднее значение зависимой переменной в обучающем наборе. Закодированное значение признака будет находиться ближе к среднему значению зависимой переменной в обучающем наборе. Это и есть нужный нам результат.

В рамках кодирования средним значением, слаженным через сигмоиду, выделяют два подхода. В рамках первого подхода слаживание не применяется к тестовой выборке и набору новых данных. В рамках второго подхода слаживание применяется к тестовой выборке и набору новых данных. Рассмотрим первый подход.

Мы начинаем с того, что применяем слаживание для обучающей выборки. Если у нас есть надежные априорные знания, позволяющие избежать настройки гиперпараметров (здесь гиперпараметрами будут k и f), то для проверки используете тестовую выборку. Для тестовой выборки мы используем обычные средние значения зависимой переменной в соответствующих категориях признака, вычисленные для обучающей выборки. Если требуется настройка гиперпараметров k и f , схема немного изменится. Откладываем проверочную выборку. На обучающей выборке выполняем слаживание с помощью разных комбинаций k и f , на проверочной выборке проверяем качество моделей машинного обучения при разных комбинациях k и f . Для проверочной выборки мы используем обычные средние значения зависимой переменной в соответствующих категориях признака, вычисленные на обучающей выборке. На проверочной выборке нашли наилучшую комбинацию гиперпараметров k и f , объединяя обучающую и проверочную выборки. Выполняем слаживание на обучающе-проверочной выборке, мы должны заново вычислить слаженные средние значения зависимой переменной для категорий признака в обучающе-проверочной выборке с помощью найденной наилучшей комбинации гиперпараметров k и f . На тестовой выборке получаем итоговую оценку качества модели машинного обучения, соответствующей наилучшей комбинации гиперпараметров k и f . Для тестовой выборки мы используем обычные средние значения зависимой переменной в соответствующих категориях признака, вычисленные на обучающе-проверочной выборке. Если итоговая оценка качества нас устраивает, объединяя обучающую, проверочную и тестовую выборки в исторический набор. Выполняем слаживание на историческом наборе, мы должны заново вычислить слаженные средние значения зависимой переменной для категорий признака на всем историческом наборе с помощью найденной комбинации наилучших гиперпараметров. Для набора новых дан-

ных мы используем обычные средние значения зависимой переменной в соответствующих категориях признака, вычисленные для исторического набора.

Таким образом, для проверочной выборки / тестовой выборки / набора новых данных мы всегда используем обычные средние значения зависимой переменной в соответствующих категориях признака, полученные в обучающей выборке / обучающе-проверочной выборке / историческом наборе данных. Сторонники данного подхода рассуждают так. Проверочная/тестовая выборка – это прообраз новых данных. В новых данных зависимой переменной нет, нам неоткуда взять информацию о ней, и нам не нужна защита от «утечек» в виде применения сглаженных значений.

Теперь рассмотрим второй подход, когда сглаживание применяется к тестовой выборке и набору новых данных.

Мы опять начинаем с того, что применяем сглаживание для обучающей выборки. Если есть надежные априорные знания, позволяющие избежать настройки гиперпараметров (здесь гиперпараметрами будут k и f), то для проверки используем тестовую выборку. Для тестовой выборки мы используем сглаженные средние значения зависимой переменной в соответствующих категориях признака, вычисленные для обучающей выборки. Если требуется настройка гиперпараметров k и f , откладываем проверочную выборку. На обучающей выборке выполняем сглаживание с помощью разных комбинаций k и f , на проверочной выборке проверяем качество моделей машинного обучения при разных комбинациях k и f . Для проверочной выборки мы используем сглаженные средние значения зависимой переменной в соответствующих категориях признака, вычисленные на обучающей выборке. На проверочной выборке нашли наилучшую комбинацию гиперпараметров k и f , объединяя обучающую и проверочную выборки. Выполняем сглаживание на обучающе-проверочной выборке, мы должны заново вычислить сглаженные средние значения зависимой переменной для категорий признака в обучающе-проверочной выборке с помощью найденной наилучшей комбинации гиперпараметров k и f . На тестовой выборке получаем итоговую оценку качества модели машинного обучения, соответствующей наилучшей комбинации гиперпараметров k и f . Для тестовой выборки мы используем сглаженные средние значения зависимой переменной в соответствующих категориях признака, вычисленные на обучающе-проверочной выборке. Если итоговая оценка качества нас устраивает, объединяя обучающую, проверочную и тестовую выборку, в исторический набор. Выполняем сглаживание на историческом наборе, мы должны заново вычислить сглаженные средние значения зависимой переменной для категорий признака на всем историческом наборе с помощью найденной наилучшей комбинации гиперпараметров. Для набора новых данных мы используем сглаженные средние значения зависимой переменной в соответствующих категориях признака, вычисленные для исторического набора.

Таким образом, для проверочной выборки / тестовой выборки / набора новых данных мы всегда используем сглаженные средние значения зависимой переменной в соответствующих категориях признака, полученные в обучающей выборке / обучающе-проверочной выборке / историческом наборе данных. Сторонники данного подхода рассуждают так. Мы выполнили сглаживание в обучающей выборке, распределение признаков изменилось. Проверка

качества модели, построенной на обучающей выборке с одним распределением признаков, на тестовой выборке с другим распределением признаков не корректна. Апельсины нужно сравнивать с апельсинами, а не с яблоками. К тому же если в обучающей выборке были редкие категории, неправдоподобные оценки в случае использования обычных групповых средних мы переносим и в тестовую выборку.

На практике используются оба подхода. При этом отметим, что второй подход становится более распространенным по сравнению с первым.

Давайте проиллюстрируем кодирование слаженными средними значениями зависимой переменной. Вычислим значения для обучающей выборки.

```
# задаем список признаков
lst = ['living_region', 'job_position']
# создаем экземпляр класса SmoothingTargetEncoder
ste = SmoothingTargetEncoder(columns_names=lst, f=2, k=4)

# обучаем и применяем модель к обучающей выборке, т. е.
# для каждого признака создаем таблицу, в соответствии
# с которой категориям признака в обучающей выборке будут
# сопоставлены слаженные средние значения
# зависимой переменной в этих категориях, и сопоставляем
enc_train = ste.fit_transform(train, train['open_account_flg'])
enc_train
```

	living_region	job_position
0	0.425169	0.358333
1	0.425169	0.491035
2	0.674153	0.358333
3	0.425169	0.549661
4	0.674153	0.549661
5	0.674153	0.549661
6	0.674153	0.549661
7	0.674153	0.549661
8	0.425169	0.491035
9	0.425169	0.491035
10	0.300678	0.491035
11	0.300678	0.358333
12	0.300678	0.491035
13	0.300678	0.491035
14	0.300678	0.358333

Теперь вычислим значения для тестовой выборки. Для категорий признака в тестовой выборке используем обычные средние значения зависимой переменной в этих категориях, вычисленные на обучающей выборке.

```
# применяем модель к тестовой выборке,
# категории признака в тестовой выборке заменяются на обычные
```

```
# средние значения зависимой переменной в этих категориях,
# вычисленные на обучающей выборке
enc_test = ste.transform(test, smoothing=False)
enc_test
```

	living_region	job_position
0	0.8	0.25
1	0.4	0.50
2	0.8	0.25
3	0.4	0.60
4	0.8	0.25
5	0.8	0.60
6	0.8	0.60
7	0.4	0.50
8	0.4	0.50
9	0.2	0.50
10	0.4	0.50
11	0.2	0.25

А теперь для категорий признака в тестовой выборке используем слаженные средние значения зависимой переменной в этих категориях, вычисленные на обучающей выборке.

```
# применяем модель к тестовой выборке,
# категории признака в тестовой выборке заменяются
# на слаженные средние значения зависимой переменной
# в этих категориях, вычисленные на обучающей выборке
enc_test = ste.transform(test, smoothing=True)
enc_test
```

	living_region	job_position
0	0.674153	0.358333
1	0.425169	0.491035
2	0.674153	0.358333
3	0.425169	0.549661
4	0.674153	0.358333
5	0.674153	0.549661
6	0.674153	0.549661
7	0.425169	0.491035
8	0.425169	0.491035
9	0.300678	0.491035
10	0.425169	0.491035
11	0.300678	0.358333

Допустим, мы построили модель машинного обучения и нас устроило то качество, которое мы получаем при выбранных значениях гиперпараметров k и f . Тогда обучаем модель сглаживания на всем историческом наборе.

создаем исторический набор

```
full_data = pd.concat([train, test], axis=0, ignore_index=True)  
full_data
```

	living_region	job_position	open_account_flg
0	Московская область	Служащий	0
1	Московская область	Заместитель руководителя	1
2	Пермский край	Служащий	1
3	Московская область	Руководитель	0
4	Пермский край	Руководитель	1
5	Пермский край	Руководитель	1
6	Пермский край	Руководитель	1
7	Пермский край	Руководитель	0
8	Московская область	Заместитель руководителя	1
9	Московская область	Заместитель руководителя	0
10	Свердловская область	Заместитель руководителя	0
11	Свердловская область	Служащий	0
12	Свердловская область	Заместитель руководителя	0
13	Свердловская область	Заместитель руководителя	1
14	Свердловская область	Служащий	0
15	Пермский край	Служащий	0
16	Московская область	Заместитель руководителя	0
17	Пермский край	Служащий	1
18	Московская область	Руководитель	0
19	Пермский край	Служащий	1
20	Пермский край	Руководитель	0
21	Пермский край	Руководитель	1
22	Московская область	Заместитель руководителя	1
23	Московская область	Заместитель руководителя	1
24	Свердловская область	Заместитель руководителя	0
25	Московская область	Заместитель руководителя	0
26	Свердловская область	Служащий	1

```
# создаем экземпляр класса SmoothingTargetEncoder
full_stc = SmoothingTargetEncoder(columns_names=lst, f=2, k=4)
# обучаем и применяем модель к историческому набору, т. е.
# для каждого признака создаем таблицу, в соответствии
# с которой категориям признака в историческом наборе
# будут сопоставлены слаженные средние значения
# зависимой переменной в этих категориях, и сопоставляем
enc_full_data = full_stc.fit_transform(
    full_data, full_data['open_account_flg'])
enc_full_data
```

	living_region	job_position
0	0.403864	0.497793
1	0.403864	0.455335
2	0.689637	0.497793
3	0.403864	0.497793
4	0.689637	0.497793
5	0.689637	0.497793
6	0.689637	0.497793
7	0.689637	0.497793
8	0.403864	0.455335
9	0.403864	0.455335
10	0.321427	0.455335
11	0.321427	0.497793
12	0.321427	0.455335
13	0.321427	0.455335
14	0.321427	0.497793
15	0.689637	0.497793
16	0.403864	0.455335
17	0.689637	0.497793
18	0.403864	0.497793
19	0.689637	0.497793
20	0.689637	0.497793
21	0.689637	0.497793
22	0.403864	0.455335
23	0.403864	0.455335
24	0.321427	0.455335
25	0.403864	0.455335
26	0.321427	0.497793

Допустим, к нам поступили новые данные.

```
# создаем набор новых данных
new_data = pd.DataFrame(
    {'living_region': ['Московская область',
                      'Краснодарский край',
                      'Пермский край',
                      'Свердловская область'],
     'job_position': ['Заместитель руководителя',
                      'Служащий',
                      np.NaN,
                      'Руководитель']})
new_data
```

	living_region	job_position
0	Московская область	Заместитель руководителя
1	Краснодарский край	Служащий
2	Пермский край	NaN
3	Свердловская область	Руководитель

Обратите внимание: в новых данных признак *living_region* содержит новую категорию 'Краснодарский край', а признак *job_position* содержит пропуск.

Давайте вычислим глобальное среднее в историческом наборе.

```
# вычисляем глобальное среднее в историческом наборе
full_data['open_account_flg'].mean()
```

0.4814814814814815

Запоминаем, что именно этим значением должны быть заменены новая категория и пропуск.

Теперь вычислим по историческому набору средние значения зависимой переменной в категориях соответствующего признака.

```
# вычислим средние значения зависимой переменной
# в категориях признака living_region
# по исторической выборке
full_data.groupby('living_region')['open_account_flg'].mean()
```

```
living_region
Московская область      0.400000
Пермский край          0.700000
Свердловская область   0.285714
Name: open_account_flg, dtype: float64
```

```
# вычислим средние значения зависимой переменной
# в категориях признака job_position
# по исторической выборке
full_data.groupby('job_position')['open_account_flg'].mean()
```

```
job_position
Заместитель руководителя 0.454545
```

```
Руководитель          0.500000
Служащий            0.500000
Name: open_account_flg, dtype: float64
```

Именно этими значениями согласно первому подходу должны быть заменены категории соответствующих признаков в наборе новых данных.

Итак, применяем модель к новым данным.

```
# применяем модель к набору новых данных,
# категории признака в наборе новых данных заменяются
# на обычные средние значения зависимой переменной
# в этих категориях, вычисленные на историческом наборе
enc_new_data = full_stc.transform(new_data, smoothing=False)
enc_new_data
```

	living_region	job_position	living_region	job_position
0	0.400000	0.454545	0	Московская область Заместитель руководителя
1	0.481481	0.500000	1	Краснодарский край Служащий
2	0.700000	0.481481	2	Пермский край NaN
3	0.285714	0.500000	3	Свердловская область Руководитель

Теперь категории признаков в новых данных заменим на сглаженные средние значения зависимой переменной в этих категориях, вычисленные на историческом наборе.

```
# применяем модель к набору новых данных,
# категории признака в наборе новых данных заменяются
# на сглаженные средние значения зависимой переменной
# в этих категориях, вычисленные на историческом наборе
enc_new_data = full_stc.transform(new_data, smoothing=True)
enc_new_data
```

	living_region	job_position
0	0.403864	0.455335
1	0.481481	0.497793
2	0.689637	0.481481
3	0.321427	0.497793

Часто бывает, что категориальная переменная содержит пропущенные значения. В таком случае можно ввести дополнительное значение X – нулевое значение и оценивать вероятность Y для $X = X_0$ с помощью стандартной формулы:

$$S_0 = \lambda(n_0) \frac{n_{0Y}}{n_0} + (1 - \lambda(n_0)) \frac{n_Y}{n_{TR}}.$$

Преимущество данного подхода заключается в том, что если пропуск действительно влияет на зависимую переменную, то оценка учитывает это. Если же наличие пропусков не влияет на значение зависимой переменной, то оценка

будет стремиться к априорной вероятности зависимой переменной, и это будет соответствовать «нейтральности» пропущенного значения.

Класс `TargetEncoder` пакета `category_encoders` также позволяет выполнить кодирование средними значениями зависимой переменной с использованием сглаживания через сигмоидальную функцию.

Ниже приводится список параметров и гиперпараметров для класса `TargetEncoder`.

Параметр/ гиперпараметр	Предназначение
<code>cols</code>	Задает список признаков для кодировки
<code>min_samples_leaf</code>	Задает частоту категории, которую надо учитывать при вычислении среднего значения зависимой переменной в категории признака
<code>smoothing</code>	Задает сглаживание (баланс между апостериорной и априорной вероятностями)

Давайте выполним кодирование для обучающей и тестовой выборок.

```
# импортируем класс TargetEncoder
# из пакета category_encoders
from category_encoders import TargetEncoder

# создаем экземпляр класса TargetEncoder
# для обучающей выборки
target_enc = TargetEncoder(cols=lst,
                           smoothing=2,
                           min_samples_leaf=4)

# обучаем и применяем модель к обучающей выборке,
# т. е. для каждого признака создаем таблицу,
# в соответствии с которой категориям признака
# в обучающей выборке будут сопоставлены сглаженные
# средние значения зависимой переменной в этих
# категориях, и сопоставляем
target_encoded_train = target_enc.fit_transform(
    train, train['open_account_flg'])

# обучаем и применяем модель к тестовой выборке,
# т. е. для каждого признака создаем таблицу,
# в соответствии с которой категориям признака
# в тестовой выборке будут сопоставлены сглаженные
# средние значения зависимой переменной в этих
# категориях, вычисленные на обучающей выборке
target_encoded_test = target_enc.transform(
    test)
```

Вгляднем на результаты в обучающей выборке.

```
# взглянем на результаты кодировки
# в обучающей выборке
target_encoded_train
```

	living_region	job_position
0	0.425169	0.358333
1	0.425169	0.491035
2	0.674153	0.358333
3	0.425169	0.549661
4	0.674153	0.549661
5	0.674153	0.549661
6	0.674153	0.549661
7	0.674153	0.549661
8	0.425169	0.491035
9	0.425169	0.491035
10	0.300678	0.491035
11	0.300678	0.358333
12	0.300678	0.491035
13	0.300678	0.491035
14	0.300678	0.358333

Выясним, как было получено сглаженное среднее значение зависимой переменной для категории 'Московская область' признака *living_region* (выделены красной рамкой).

	living_region	job_position	living_region	job_position	open_account_flg
0	0.425169	0.358333	Московская область	Служащий	0
1	0.425169	0.491035	Московская область	Заместитель руководителя	1
2	0.674153	0.358333	Пермский край	Служащий	1
3	0.425169	0.549661	Московская область	Руководитель	0
4	0.674153	0.549661	Пермский край	Руководитель	1
5	0.674153	0.549661	Пермский край	Руководитель	1
6	0.674153	0.549661	Пермский край	Руководитель	1
7	0.674153	0.549661	Пермский край	Руководитель	0
8	0.425169	0.491035	Московская область	Заместитель руководителя	1
9	0.425169	0.491035	Московская область	Заместитель руководителя	0
10	0.300678	0.491035	Свердловская область	Заместитель руководителя	0
11	0.300678	0.358333	Свердловская область	Служащий	0
12	0.300678	0.491035	Свердловская область	Заместитель руководителя	0
13	0.300678	0.491035	Свердловская область	Заместитель руководителя	1
14	0.300678	0.358333	Свердловская область	Служащий	0

Сначала вычисляем среднее значение зависимой переменной для категории 'Московская область'. Из пяти наблюдений, относящихся к категории 'Московская область', в двух зависимая переменная принимает значение 1, $2 / 5 = 0,4$. Затем вычисляем среднее значение зависимой переменной в нашем наборе данных, $7 / 15 = 0,47$. Теперь вычисляем весовой коэффициент λ . Поскольку у нас $\text{min_samples_leaf}=4$ и $\text{smoothing}=2$, то k равно 4, а f равно 2. Весовой коэффициент равен $1 / (1 + \exp(-(5 - 4)/2)) = 0,62$. Наконец, вычисляем сглаженное среднее, $0,62 * 0,4 + (1 - 0,62) * 0,47 = 0,25 + 0,18 = 0,43$.

Мы получили сглаженные средние значения зависимой переменной, идентичные значениям, которые вычислили ранее с помощью класса `SmoothingTargetEncoder`.

А если мы посмотрим закодированные значения для тестовой выборки, это будут сглаженные средние значения зависимой переменной в категориях признаков, вычисленные на обучающей выборке. Таким образом, здесь реализован второй подход.

```
# взглянем на результаты кодировки
```

```
# в тестовой выборке
```

```
target_encoded_test
```

	<code>living_region</code>	<code>job_position</code>	<code>open_account_flg</code>
0	0.674153	0.358333	0
1	0.425169	0.491035	0
2	0.674153	0.358333	1
3	0.425169	0.549661	0
4	0.674153	0.358333	1
5	0.674153	0.549661	0
6	0.674153	0.549661	1
7	0.425169	0.491035	1
8	0.425169	0.491035	1
9	0.300678	0.491035	0
10	0.425169	0.491035	0
11	0.300678	0.358333	1

16.2.5.6. Кодирование средним значением зависимой переменной, сглаженным через сигмоиду по схеме «K-Fold»

Кодирование средним значением зависимой переменной, сглаженным через сигмоиду, можно дополнить схемой «K-Fold».

	Переменная Class	Номер блока	Зависимая переменная	Кодировка Kfold Blended Mean Target Encoding для переменной Class
1	A	0	1	0,568
2	A	0	1	0,568
3	A	0	1	0,568
4	A	0	1	0,568
5	A	0	0	0,568
6	A	0	1	0,568
7	A	1	0	0,616
8	A	1	1	0,616
9	A	2	1	0,596
10	B	0	0	0,507
11	B	0	1	0,507
12	B	1	0	0,517
13	B	1	0	0,517
14	B	1	1	0,517
15	B	1	1	0,517
16	B	1	0	0,517
17	B	2	0	0,524
18	C	0	0	0,543
19	C	1	0	0,543
20	C	2	1	0,472

Рис. 49 K-fold Smoothed Mean Target Encoding

Выясним, как было получено сглаженное среднее значение зависимой переменной для категории A признака Class в блоке 0 (выделены синим жирным шрифтом). Сначала вычисляем среднее значение зависимой переменной для категории A в блоке 0. Для вычисления этого среднего значения используются лишь наблюдения в категории A в обучающих блоках 1 и 2 (выделены красным жирным шрифтом), $2 / 3 = 0,67$. Затем вычисляем среднее значение зависимой переменной в нашем наборе данных, $11 / 20 = 0,55$. Теперь вычисляем весовой коэффициент λ . Задаем k равным 20, а f равным 10. Весовой коэффициент равен $1 / (1 + \exp(-(3 - 20)/10)) = 0,154$. Наконец, вычисляем сглаженное среднее, $0,154 * 0,67 + (1 - 0,154) * 0,55 = 0,10 + 0,47 = 0,57$.

Кодирование средним значением зависимой переменной, сглаженным через сигмоиду по схеме «K-Fold», можно выполнить с помощью класса H2OTargetEncoderEstimator библиотеки h2o. Ниже приводятся параметры класса H2OTargetEncoderEstimator.

```
from h2o.estimators import H2OTargetEncoderEstimator(
    model_id=None, ← Идентификатор модели (генерируется автоматически)
    fold_column=None, ← Название столбца с блоками фолдов
    keep_original_categorical_columns=True, ← Сохранение оригинальных
    blending=False, ← Смешивание вероятностей
    inflection_point=10.0, ← Параметр k
    smoothing=20.0, ← Параметр f
    data_leakage_handling='none', ← Схема защиты от «утечки». Можно выбрать
    values['none', 'leave_one_out', 'k_fold']
    noise=0.01, ← Добавление случайного шума к закодированным значениям
    seed=-1) ← Стартовое значение генератора псевдослучайных чисел
```

Давайте импортируем библиотеку h2o, запустим кластер и импортируем класс H2OTargetEncoderEstimator.

```
# импортируем библиотеку H2O
import h2o
# запускаем кластер H2O
h2o.init()

# импортирум класс H2OTargetEncoderEstimator
from h2o.estimators import H2OTargetEncoderEstimator
```

Создаем игрушечный обучающий датафрейм, в котором столбцы *Class* и *Response* идентичны столбцам *Class* и *Response* в примере, приведенном в начале этого раздела. Кроме того, создадим игрушечный тестовый датафрейм.

```
# создадим игрушечные обучающий и тестовый датафреймы
tr_h2o = pd.DataFrame(
    {'Class': ['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A',
              'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
              'C', 'C', 'C'],
     'Response': [1, 1, 1, 1, 0, 0, 1, 1, 0,
                  1, 0, 0, 1, 1, 0, 0, 0, 1]})

tst_h2o = pd.DataFrame(
    {'Class': ['A', 'B', 'A', 'C', 'D'],
     'Response': [1, 0, 1, 0, 0]})
```

Преобразовываем наши датафреймы во фреймы H2O.

```
# преобразовываем датафреймы pandas во фреймы h2o
tr_h2o = h2o.H2OFrame(tr_h2o)
tst_h2o = h2o.H2OFrame(tst_h2o)
```

Преобразовываем зависимую переменную в категориальную.

```
# преобразовываем зависимую переменную в категориальную
tr_h2o['Response'] = tr_h2o['Response'].asfactor()
tst_h2o['Response'] = tst_h2o['Response'].asfactor()
```

Задаем название зависимой переменной, название столбца с номерами фолдов (блоков) перекрестной проверки и собственно столбец с номерами фолдов.

```
# задаем название зависимой переменной
target = 'Response'

# задаем название столбца с номерами фолдов
fold_column = 'kfold_column'

# создаем столбец с номерами фолдов
# и преобразовываем во фрейм h2o
fold_col = h2o.H2OFrame(
    pd.DataFrame({'C1': [0, 0, 0, 0, 0, 0, 1, 1, 2, 0,
                        0, 1, 1, 1, 1, 2, 0, 1, 2]}))
```

Добавляем столбец с номерами фолдов в обучающую выборку.

```
# добавляем столбец с номерами фолдов
# в обучающую выборку
tr_h2o[fold_column] = fold_col
```

Теперь с помощью класса `H2OTargetEncoderEstimator` выполним кодирование в обучающей выборке.

```
# выполняем кодирование в обучающей выборке

# создаем экземпляр класса H2OTargetEncoderEstimator
te = H2OTargetEncoderEstimator(
    fold_column=fold_column,
    data_leakage_handling='k_fold',
    blending=True,
    inflection_point=20,
    smoothing=10,
    noise=0,
    seed=None)

# обучаем модель на обучающей выборке,
# т. е. для каждого признака создаем таблицу,
# в соответствии с которой категориям признака
# в обучающей выборке будут сопоставлены слаженные
# по схеме kfold средние значения зависимой
# переменной в этих категориях
te.train(x=['Class'],
         y=target,
         training_frame=tr_h2o)

# применяем модель к обучающей выборке,
# для каждого признака категориям сопоставляем
# слаженные по схеме kfold средние значения
# зависимой переменной в этих категориях
tr_te = te.transform(frame=tr_h2o, as_training=True)
tr_te
```

Class_te	Class	kfold_column	Response
0.568021	A	0	1
0.568021	A	0	1
0.568021	A	0	1
0.568021	A	0	1
0.568021	A	0	0
0.568021	A	0	1
0.615779	A	1	0
0.615779	A	1	1
0.596295	A	2	1
0.50714	B	0	0

Видим закодированные значения для признака *Class*. В самом начале раздела мы подробно объяснили способ их вычисления.

Теперь с помощью класса `H20TargetEncoderEstimator` выполним кодирование в тестовой выборке. По умолчанию реализован второй подход. Для категорий признака в тестовой выборке используем сглаженные средние значения зависимой переменной в этих категориях, вычисленные на обучающей выборке. Новые категории в тестовой выборке кодируются глобальным средним, вычисленным на обучающей выборке.

```
# применяем модель к тестовой выборке,
# для каждого признака категориям сопоставляем сглаженные
# средние значения зависимой переменной в этих категориях,
# вычисленные на обучающей выборке
tst_te = te.transform(frame=tst_h2o, as_training=False, noise=0)
tst_te
```

Class_te	Class	Response
0.652538	A	1
0.47977	B	0
0.652538	A	1
0.50714	C	0
0.55	D	0

При желании можно реализовать первый подход, когда для категорий признака в тестовой выборке используем обычные средние значения зависимой переменной в этих категориях, вычисленные на обучающей выборке.

```
# создаем экземпляр класса H20TargetEncoderEstimator
te = H20TargetEncoderEstimator(
    fold_column=fold_column,
    data_leakage_handling=None,
    blending=None,
    inflection_point=None,
    smoothing=None,
    noise=0,
    seed=None)

# обучаем модель на обучающей выборке,
# т. е. для каждого признака создаем таблицу,
# в соответствии с которой категориям признака
# в тестовой выборке будут сопоставлены обычные
# средние значения зависимой переменной в этих
# категориях, вычисленные на обучающей выборке
te.train(x=['Class'],
         y=target,
         training_frame=tr_h2o)

# применяем модель к тестовой выборке,
# для каждого признака категориям сопоставляем обычные
# средние значения зависимой переменной в этих
# категориях, вычисленные на обучающей выборке
tst_te = te.transform(frame=tst_h2o, as_training=False, noise=0)
tst_te
```

Class_te	Class	Response
0.7777778	A	1
0.375	B	0
0.7777778	A	1
0.3333333	C	0
0.55	D	0

16.2.5.7. Кодирование средним значением зависимой переменной, сглаженным через параметр регуляризации

Часто применяют схему кодировки средним значением зависимой переменной, сглаженным через параметр регуляризации. Она выполняется по формуле:

$$S_c = \frac{(p_c \times n_c + p_{global} \times a)}{(n_c + a)},$$

где

p_c – среднее значение зависимой переменной в категории признака;

n_c – количество наблюдений в категории;

p_{global} – среднее значение зависимой переменной в обучающем наборе (глобальное среднее);

a – параметр регуляризации, можно рассматривать как размер категории, которому мы доверяем.

Переменная Class	Зависимая переменная
A	1
A	0
A	1
A	1
B	1
B	1
B	0
C	1
C	1

$a = 2$

	n	$mean(level)$	$mean(dataset)$	Результат кодировки
A	4	0,75	0,77	$(0,75*4 + 0,77*2) / (4+2) = 0,7566$
B	3	0,66	0,77	$(0,66*3 + 0,77*2) / (3+2) = 0,704$
C	2	1,00	0,77	$(1*2 + 0,77*2) / (2+2) = 0,885$

$a = 3$

	n	$mean(level)$	$mean(dataset)$	Результат кодировки
A	4	0,75	0,77	$(0,75*4 + 0,77*3) / (4+3) = 0,7585$
B	3	0,66	0,77	$(0,66*3 + 0,77*3) / (3+3) = 0,715$
C	2	1,00	0,77	$(1*2 + 0,77*3) / (2+3) = 0,862$

Давайте реализуем такое кодирование в Python.

```
# пишем функцию, которая выполняет кодирование средним значением
# зависимой переменной, слаженным через параметр регуляризации
def simple_smooth_mean(df, feature, target, alpha):
    # вычисляем глобальное среднее
    mean = df[target].mean()
    # вычисляем частоты и средние по каждой категории
    agg = df.groupby(feature)[target].agg(['count', 'mean'])
    counts = agg['count']
    means = agg['mean']
    # вычисляем слаженные средние
    smooth = (counts * means + alpha * mean) / (counts + alpha)
    # заменяем каждое значение соответствующим
    # слаженным средним
    return df[feature].map(smooth)

# выполняем кодирование и смотрим результаты
tr['Class_smpl_smooth_mean_enc'] = simple_smooth_mean(
    tr,
    feature='Class',
    target='Response',
    alpha=2)
```

tr

	Class	Response	Class_labelenc	Class_custlabelenc	Agecat	Class_custlabelenc2	Class_abs_freq	Class_rel_freq	Class_smpl_smooth_mean_enc
0	A	1	0	1	old	1	4	0.444444	0.759259
1	A	0	0	1	old	1	4	0.444444	0.759259
2	A	1	0	1	yng	2	4	0.444444	0.759259
3	A	1	0	1	yng	2	4	0.444444	0.759259
4	B	1	1	2	old	3	3	0.333333	0.711111
5	B	1	1	2	old	3	3	0.333333	0.711111
6	B	0	1	2	yng	0	3	0.333333	0.711111
7	C	1	2	3	old	4	2	0.222222	0.888889
8	C	1	2	3	old	4	2	0.222222	0.888889

16.2.5.8. Кодирование средним значением зависимой переменной, вычисленным по «прошлому» (упрощенный вариант кодировки, применяющийся в библиотеке CatBoost)

Теперь рассмотрим кодирование средним значением зависимой переменной, вычисленным по «прошлому» (упрощенный вариант кодировки, применяющийся в библиотеке CatBoost). Эту схему еще называют кодированием расширяющимся средним значением зависимой переменной. Давайте проиллюстрируем ее на игрушечном примере.

Переменная Class	Зависимая переменная
A	1
A	0
A	1
A	1
B	1
B	1
B	0
C	1
C	1

Сначала выполняем перемешивание данных.



Переменная Class	Зависимая переменная
A	1
A	0
A	1
A	1
B	1
B	1
B	0
C	1
C	1

Переменная Class	Зависимая переменная
B	1
A	1
B	1
A	0
A	1
B	0
C	1
C	1
A	1

Когда категория признака встречается в первый раз, мы заменяем ее средним значением зависимой переменной в обучающем наборе (глобальным средним).

Переменная Class	Зависимая переменная	History Target Encoding
B	1	0,77
A	1	0,77
B	1	
A	0	
A	1	
B	0	
C	1	0,77
C	1	
A	1	

Особенность заключается в том, что для вычисления среднего значения зависимости переменной в категории признака мы используем лишь те наблюдения, которые предшествовали рассматриваемому наблюдению (т. е. являлись для рассматриваемого наблюдения «историей»). Например, вычислим среднее значение зависимой переменной для наблюдения 5 (выделено красным). Здесь признак *Class* имеет категорию A. Нас будут интересовать наблюдения с категорией A, которые предшествовали наблюдению 5 (выделено зеленым). У нас два таких наблюдения. В одном наблюдении из этих двух зависимая переменная принимает значение 1, $1 / 2 = 0,5$.

Переменная Class	Зависимая переменная	History Target Encoding
B	1	0,77
A	1	0,77
B	1	1,00 (1/1)
A	0	1,00 (1/1)
A	1	0,50 (1/2)
B	0	1,00 (2/2)
C	1	0,77
C	1	1,00 (1/1)
A	1	0,66 (2/3)

Давайте реализуем это кодирование в Python.

Пишем функцию `history_mean_encoding()`, которая выполняет кодирование средними значениями зависимой переменной, вычисленными по «прошлому».

```
# пишем функцию, которая выполняет кодирование средними значениями
# зависимой переменной, вычисленными по «прошлому»
def history_mean_encoding(df, feature, target, random_state):
    # записываем индекс
    df['index'] = df.index
    # задаем стартовое значение генератора
    # псевдослучайных чисел
    np.random.seed(random_state)
    # перемешиваем
    df = df.sample(n=len(df), replace=False)
    # вычисляем суммарное значение зависимой переменной
    # нарастающим итогом, интересующее наблюдение
    # в расчетах не участвует
    cumsum = df.groupby(feature)[target].cumsum() - df[target]
    # вычисляем количество наблюдений нарастающим итогом,
    # интересующее наблюдение в расчетах не участвует
    cumcnt = df.groupby(feature).cumcount()
    # вычисляем глобальное среднее
    global_mean = df[target].mean()
    # получаем закодированные значения
    df[feature+'_history_mean_encoded'] = (cumsum / cumcnt).fillna(
        global_mean)
    # восстанавливаем исходный порядок наблюдений
    # по ранее сохраненному индексу
    df = df.sort_values('index')
    # удаляем индекс
    df.drop('index', axis=1, inplace=True)
    return df
```

Применяем нашу функцию и смотрим результаты.

```
# выполняем кодирование средними значениями зависимой
# переменной, вычисленными по «прошлому»
history_mean_encoding(tr, 'Class', 'Response', random_state=4)
```

Class_labelenc	Class_custlabelenc	Agecat	Class_custlabelenc2	Class_abs_freq	Class_rel_freq	Class_smpl_smooth_mean_enc	Class_history_mean_encoded
0	1	old		1	4	0.444444	0.759259
0	1	old		1	4	0.444444	0.759259
0	1	yng		2	4	0.444444	0.759259
0	1	yng		2	4	0.444444	0.777778
1	2	old		3	3	0.333333	0.711111
1	2	old		3	3	0.333333	0.711111
1	2	yng		0	3	0.333333	0.711111
2	3	old		4	2	0.222222	0.888889
2	3	old		4	2	0.222222	0.888889

Ниже приведена схема, поясняющая, что происходит под капотом функции `history_mean_encoding()`.

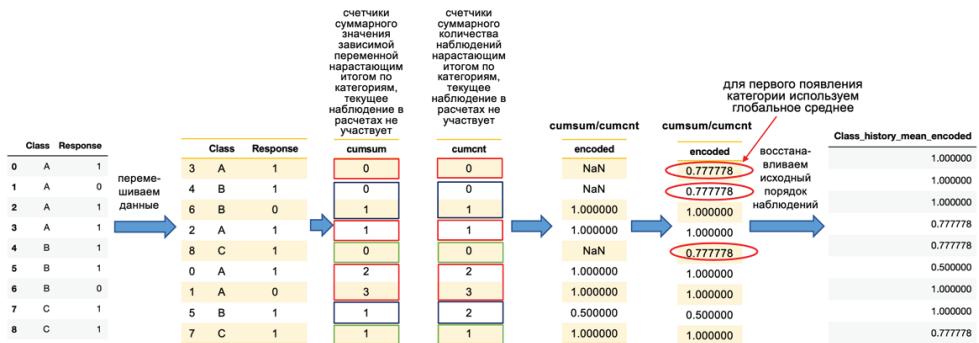


Рис. 50 Кодировка средним значением зависимой переменной, вычисленным по «прошлому»

Возьмем столбец `cumsum`. У нас – три категории, три счетчика. По умолчанию все счетчики начинаются с 0. Посмотрим, как работает счетчик для категории A. У нас 4 наблюдения категории A. В первом появлении категории A (индекс 3) зависимая переменная принимает значение 1, наблюдение в расчетах не участвует, счетчик равен 0. Во втором появлении категории A (индекс 2) зависимая переменная принимает значение 1, наблюдение в расчетах не участвует, мы смотрим значение зависимой переменной 1 для первого появления категории A, счетчик увеличивается на 1 и становится равным 1. В третьем появлении категории A (индекс 0) зависимая переменная принимает значение 1, наблюдение в расчетах не участвует, мы смотрим значение зависимости переменной 1 для второго появления категории A, счетчик увеличивается на 1 и становится равным 2. В четвертом появлении категории A (индекс 1) зависимая переменная принимает значение 0, наблюдение в расчетах не участвует, мы смотрим значение зависимости переменной 1 для третьего появления категории A, счетчик увеличивается на 1 и становится равным 3. И так по каждой категории.

Возьмем столбец `cumcnt`. У нас – три категории, три счетчика. По умолчанию все счетчики начинаются с 0. Посмотрим, как работает счетчик для категории A. У нас 4 наблюдения категории A. Первое появление категории A (индекс 3) не учитываем, наблюдение в расчетах не участвует, счетчик равен 0. Второе появление категории A (индекс 2) не учитываем, зато учитываем пер-

вое появление категории А, счетчик увеличивается на 1 и становится равным 1. Третье появление категории А (индекс 0) не учитываем, зато учитываем второе появление категории А, счетчик увеличивается на 1 и становится равным 2. Четвертое появление категории А (индекс 1) не учитываем, зато учитываем третье появление категории А, счетчик увеличивается на 1 и становится равным 3. И так по каждой категории.

Давайте напишем собственный класс `TargetEncodingExpandingMean`, выполняющий кодирование средним значением зависимой переменной по «прошлому». Здесь реализован собственный метод `.fit_transform()`. Для пропусков и новых категорий используем глобальные средние, вычисленные на этапе обучения модели.

```
# пишем класс TargetEncodingExpandingMean, выполняющий
# кодирование средним значением зависимой переменной
# по «прошлому»
class TargetEncodingExpandingMean():
    """
    Автор: Dmitry Larko
    https://www.kaggle.com/dmitrylarko

    Параметры
    -----
    columns_names: list
        Список признаков.
    """

    def __init__(self, columns_names, random_state):
        # инициализируем публичные атрибуты

        # список признаков, которые будем кодировать
        self.columns_names = columns_names

        # создаем пустой словарь, куда будем сохранять обычные
        # средние значения зависимой переменной в каждой
        # категории признака:
        # ключами в словаре будут названия признаков, а значениями
        # - таблицы, в которых напротив каждой категории признака
        # будет указано среднее значение зависимой переменной
        # в данной категории признака
        self.learned_values = {}

        # создадим переменную, в которой будем хранить глобальное
        # среднее (среднее значение зависимой переменной
        # по обучающему набору)
        self.dataset_mean = np.nan

        self.random_state = random_state

    # fit должен принимать в качестве аргументов только X и y
    def fit(self, X, y=None):

        # выполняем копирование массива во избежание предупреждения
```

```

# SettingWithCopyWarning "A value is trying to be set on
# a copy of a slice from a DataFrame (Происходит попытка
# изменить значение в копии среза данных датафрейма)"
X_ = X.copy()

# добавляем в новый массив признаков зависимую переменную
# и называем ее __target__, именно эту переменную __target__
# будем использовать в дальнейшем для вычисления среднего
# значения зависимой переменной для каждой категории признака
X_[ '__target__' ] = y

# создадим переменную, в которой будем хранить глобальное среднее
# (среднее значение зависимой переменной по обучающему набору)
self.dataset_mean = np.mean(y)

# в цикле для каждого признака, который участвует в кодировании
# (присутствует в списке self.columns_names)
for c in [x for x in X_.columns if x in self.columns_names]:
    # формируем набор, состоящий из значений данного признака
    # и значений зависимой переменной, группируем данные по
    # категориям признака, считаем среднее значение зависимой
    # переменной для каждой категории признака
    stats = (X_[[c, '__target__']]
              .groupby(c)[ '__target__' ]
              .agg(['mean', 'size']))

    # вычисляем обычное среднее значение зависимой переменной
    # для категории признака
    stats['__target__'] = stats['mean']

    # формируем набор, состоящий из признака и обычных средних
    # значений зависимой переменной для категорий признака
    stats = (stats.drop([x for x in stats.columns
                           if x not in ['__target__', c]], axis=1)
              .reset_index())

    # сохраним обычные средние значения зависимой переменной
    # для каждой категории признака в словарь
    self.learned_values[c] = stats

return self

# transform выполняет преобразование для новых данных,
# transform принимает в качестве аргумента только X
def transform(self, X):

    # скопируем массив данных с признаками, значения которых
    # будем кодировать, этот массив будем изменять при вызове
    # метода .transform(), поэтому важно его скопировать,
    # чтобы не изменить исходный массив признаков
    transformed_X = X[self.columns_names].copy()

```

```
# в цикле для каждого признака, который участвует в кодировании
for c in transformed_X.columns:
    # формируем датафрейм, состоящий из значений данного признака,
    # и выполняем слияние с датафреймом, содержащим обычные средние
    # значения зависимой переменной для каждой категории признака,
    # нам нужны только обычные средние значения зависимой переменной
    # для каждой категории признака, поэтому в датафрейме оставляем
    # лишь столбец '__target__'
    transformed_X[c] = (transformed_X[[c]].merge(
        self.learned_values[c], on=c, how='left'))['__target__']

# пропуски или новые категории признаков заменяем средним
# значением зависимой переменной по обучающему набору
transformed_X = transformed_X.fillna(self.dataset_mean)

# возвращаем закодированный массив признаков
return transformed_X

# метод fit_transform выполняет преобразование для обучающих данных,
# fit_transform принимает в качестве аргументов только X, y
def fit_transform(self, X, y):

    # применяем метод fit, чтобы вычислить средние значения
    # зависимой переменной для категорий признаков
    self.fit(X, y)

    # скопируем массив данных с признаками, значения которых
    # будем кодировать, этот массив будем изменять при вызове
    # метода .transform(), поэтому важно его скопировать,
    # чтобы не изменить исходный массив признаков
    X_ = X[self.columns_names].copy().reset_index(drop=True)

    # добавляем в новый массив признаков зависимую переменную
    # и называем ее __target__, именно эту переменную __target__
    # будем использовать в дальнейшем для вычисления среднего
    # значения зависимой переменной для каждой категории признака
    X_['__target__'] = y

    # добавляем индекс наблюдения, чтобы после перемешивания
    # восстановить исходный порядок наблюдений
    X_['index'] = X_.index

    # создаем пустой датафрейм, в который будем сохранять
    # закодированные значения признаков
    X_transformed = pd.DataFrame()

    # задаем стартовое значение генератора псевдослучайных чисел
    np.random.seed(self.random_state)

    # в цикле для каждого признака, который участвует в кодировании
    # (присутствует в списке self.columns_names)
    for c in self.columns_names:
        # формируем набор, состоящий из значений данного признака,
        # значений зависимой переменной и столбца index
        X_shuffled = X_[[c, '__target__', 'index']].copy()
```

```

# перемешиваем наблюдения
X_shuffled = X_shuffled.sample(n=len(X_shuffled), replace=False)

# добавим счетчик количества наблюдений cnt, чтобы
# посчитать размер категории нарастающим итогом
X_shuffled['cnt'] = 1

# вычисляем суммарное значение зависимой переменной
# нарастающим итогом, интересующее наблюдение
# в расчетах не участвует
X_shuffled['cumsum'] = (X_shuffled
                         .groupby(c, sort=False)[ '__target__ ']
                         .apply(lambda x: x.shift().cumsum()))

# вычисляем количество наблюдений нарастающим итогом,
# интересующее наблюдение в расчетах не участвует
X_shuffled['cumcnt'] = (X_shuffled
                         .groupby(c, sort=False)[ 'cnt' ]
                         .apply(lambda x: x.shift().cumsum()))

# получаем средние значения зависимой переменной
# для каждого признака на предшествующих данных
X_shuffled['encoded'] = (X_shuffled[ 'cumsum' ] /
                          X_shuffled[ 'cumcnt' ])

# пропуски или новые категории признаков заменяем средним
# значением зависимой переменной по обучающему набору
X_shuffled[ 'encoded' ] = X_shuffled[ 'encoded' ].fillna(
    self.dataset_mean)

# отсортируем полученный датафрейм с закодированными
# значениями по столбцу index, чтобы восстановить
# порядок наблюдений
X_transformed[c] = X_shuffled.sort_values(
    'index')[ 'encoded' ].values

return X_transformed

```

Давайте применим кодирование средними значениями зависимой переменной по «истории» к тому же обучающему датафрейму, к которому мы ранее применили функцию `history_mean_encoding()`.

```

# создаем экземпляр класса TargetEncodingExpandingMean
teem = TargetEncodingExpandingMean(
    columns_names=[ 'Class' ], random_state=4)
# выполняем кодирование для обучающего датафрейма
enc_tr = teem.fit_transform(tr, tr[ 'Response' ])
enc_tr

```

Class	Class
0 1.000000	A
1 1.000000	A
2 1.000000	A
3 0.777778	A
4 0.777778	B
5 0.500000	B
6 1.000000	B
7 1.000000	C
8 0.777778	C

Результаты совпадают с теми, что мы получили ранее с помощью функции `history_mean_encoding()`.

Для категорий признака класс `TargetEncodingExpandingMean` в тестовой выборке возвращает обычные средние значения зависимой переменной в этих категориях, вычисленные на обучающей выборке. Давайте создадим тестовые данные и убедимся в этом.

```
# создаем тестовые данные
tst = pd.DataFrame({'Class': ['A', 'A', 'A', 'A', 'D',
                               'D', 'B', 'B', 'C', 'C']})

# выполняем кодирование для тестового датафрейма
enc_tst = teem.transform(tst)
enc_tst
```

Class	Class
0 0.750000	A
1 0.750000	A
2 0.750000	A
3 0.750000	A
4 0.777778	D
5 0.777778	D
6 0.666667	B
7 0.666667	B
8 1.000000	C
9 1.000000	C

Данный вид кодировки можно автоматически выполнить с помощью класса `CatBoostEncoder` пакета `category_encoders`.

```
# импортируем класс CatBoostEncoder
# из пакета category_encoders
from category_encoders import CatBoostEncoder

# создаем экземпляр класса CatBoostEncoder
# для обучающей выборки
catboost_enc = CatBoostEncoder()

# обучаем и применяем модель к обучающей выборке
ctbst_enc_tr = catboost_enc.fit_transform(
    tr[['Class']], tr['Response'])
```

Давайте взглянем на результаты кодировки в обучающей выборке.

```
# взглянем на результаты кодировки в обучающей выборке
ctbst_enc_tr
```

Class	Class
0 0.777778	A
1 0.888889	A
2 0.592593	A
3 0.694444	A
4 0.777778	B
5 0.888889	B
6 0.925926	B
7 0.777778	C
8 0.888889	C

Под капотом происходит следующее.

Перемешивание данных не выполняется (что, кстати, является недостатком, ведь мы можем так передать отсортированные категории, поэтому перед применением класса делайте предварительное перемешивание самостоятельно). Вычисляем глобальное среднее и задаем аддитивную константу, позволяющую избежать нуля в числителе и знаменателе формулы, по которой вычисляются расширяющиеся средние.

```
# вычисляем глобальное среднее
global_mean = tr['Response'].mean()
# аддитивная константа, позволяющая избежать нуля
# в числителе и знаменателе
a = 1
```

У нас есть признак и зависимая переменная в обучающей выборке.

	Class	Response
0	A	1
1	A	0
2	A	1
3	A	1
4	B	1
5	B	1
6	B	0
7	C	1
8	C	1

Мы создаем таблицу с суммарным значением зависимой переменной и общим количеством наблюдений для каждой категории.

```
# получаем таблицу с суммарным значением зависимой
# переменной и общим количеством наблюдений
# для каждой категории
tbl1 = tr['Response'].groupby(
    tr['Class']).agg(['sum', 'count'])
tbl1
```

	sum	count
Class		
A	3	4
B	2	3
C	2	2

Затем мы создаем таблицу с накопленными суммарными значениями зависимости переменной и накопленными суммами наблюдений для каждой категории.

```
# получаем таблицу с накопленными суммарными значениями
# зависимой переменной и накопленными суммами
# наблюдений для каждой категории
tbl2 = tr['Response'].groupby(
    tr['Class']).agg(['cumsum', 'cumcount'])
tbl2
```

	cumsum	cumcount
0	1	0
1	1	1
2	2	2
3	3	3
4	1	0
5	2	1
6	2	2
7	1	0
8	2	1

Вычисляем закодированные значения для обучающей выборки по формуле:

$$\frac{(\text{cumsum} - y) + (\text{mean}_{\text{global}} * a)}{(\text{cumcount} + a)}.$$

```
# получаем закодированные значения для обучающей выборки
tr_enc_values = ((tbl2['cumsum'] - tr['Response'] + global_mean * a) /
                  (tbl2['cumcount'] + a))
tr_enc_values
```

```
0    0.777778
1    0.888889
2    0.592593
3    0.694444
4    0.777778
5    0.888889
6    0.925926
7    0.777778
8    0.888889
dtype: float64
```

Видим, что они совпадают со значениями, автоматически вычисленными для обучающей выборки.

Теперь взглянем на результаты кодировки в тестовой выборке.

```
# применяем модель к тестовой выборке
ctbst_enc_tst = catboost_enc.transform(
    tst['Class'])

# взглянем на результаты кодировки в тестовой выборке
ctbst_enc_tst
```

Class	Class
0 0.755556	A
1 0.755556	A
2 0.755556	A
3 0.755556	A
4 0.777778	D
5 0.777778	D
6 0.694444	B
7 0.694444	B
8 0.925926	C
9 0.925926	C

Закодированными значениями для тестовой выборки будут расширяющиеся средние по категориям, вычисленные по формуле:

$$\frac{\text{sum} + (\text{mean}_{\text{global}} * a)}{(\text{count} + a)}.$$

Для вычислений используем таблицу с суммарным значением зависимой переменной и общим количеством наблюдений для каждой категории, полученную на обучающей выборке. А затем эти расширяющиеся средние по категориям просто сопоставляем наблюдениям, пропуски кодируем глобальным средним.

```
# получаем закодированные значения для тестовой выборки
```

```
# вычисляем расширяющиеся средние по категориям
level_means = ((tbl1['sum'] + global_mean * a) /
                (tbl1['count'] + a))
print(level_means)
tst_enc_values = tst['Class'].map(level_means)
# пропуски из-за новых категорий кодируем глобальным средним
tst_enc_values.fillna(global_mean, inplace=True)
# сопоставляем наблюдениям расширяющиеся
# средние по категориям
tst_enc_values
```

Class	
A	0.755556
B	0.694444
C	0.925926
dtype: float64	
0	0.755556
1	0.755556
2	0.755556
3	0.755556
4	0.777778
5	0.777778
6	0.694444
7	0.694444
8	0.925926
9	0.925926
Name: Class, dtype: float64	

16.2.6. Присвоение категориям в зависимости от порядка их появления целочисленных значений, начиная с 1 (Ordinal Encoding)

Категориям переменной в зависимости от порядка их появления в наборе можно присвоить целочисленные значения (начиная с 1), и в итоге категориальная переменная теряет свою категориальную природу и превращается в количественную.

Допустим, у нас есть переменная с 4 категориями: A, B, C и D. Первой категорией в нашем наборе будет категория B, второй категорией – категория A, третьей категорией – категория C, четвертой категорией – категория D. Тогда мы присвоим A – 2, B – 1, C – 3, D – 4.

Переменная Class	Переменная Class	Кодировка Ordinal Encoding для переменной Class
A	B	1
B	A	2
C	C	3
D	A	2
	A	2
	D	4
	D	4
	B	1

Рис. 51 Ordinal Encoding

Категориям переменной в зависимости от порядка их появления в наборе можно присвоить целочисленные значения (начиная с 1), и в итоге категориальная переменная теряет свою категориальную природу и превращается в количественную.

Допустим, у нас есть переменная с 4 категориями: A, B, C и D. Первой категорией в нашем наборе будет категория B, второй категорией – категория A, третьей категорией – категория C, четвертой категорией – категория D. Тогда мы присвоим A – 2, B – 1, C – 3, D – 4.

Такие признаки нужно создавать после разбиения на обучающую и тестовую выборки.

```
# импортируем класс OrdinalEncoder из пакета category_encoders
from category_encoders import OrdinalEncoder

# создаем экземпляр класса OrdinalEncoder
ord_enc = OrdinalEncoder(cols=['living_region', 'job_position'])

# выполняем кодирование переменных living_region
# и job_position в обучающей выборке
ord_enc_train = ord_enc.fit_transform(train)
ord_enc_train
```

living_region	job_position	open_account_flg		living_region	job_position
0	1	1	0	0	Московская область Служащий
1	1	2	1	1	Московская область Заместитель руководителя
2	2	1	1	2	Пермский край Служащий
3	1	3	0	3	Московская область Руководитель
4	2	3	1	4	Пермский край Руководитель
5	2	3	1	5	Пермский край Руководитель
6	2	3	1	6	Пермский край Руководитель
7	2	3	0	7	Пермский край Руководитель
8	1	2	1	8	Московская область Заместитель руководителя
9	1	2	0	9	Московская область Заместитель руководителя
10	3	2	0	10	Свердловская область Заместитель руководителя
11	3	1	0	11	Свердловская область Служащий
12	3	2	0	12	Свердловская область Заместитель руководителя
13	3	2	1	13	Свердловская область Заместитель руководителя
14	3	1	0	14	Свердловская область Служащий

В зависимости от обстоятельств данный тип кодировки может выполняться как до разбиения на обучающую и тестовую выборки (до перекрестной проверки), так и после разбиения на обучающую и тестовую выборки (внутри цикла перекрестной проверки). Если мы предполагаем, что с высокой вероятностью в наших данных могут появиться новые категории, делаем эту кодировку после разбиения на обучающую и тестовую выборки, чтобы сымитировать появление новых категорий. При этом срок, который мы определяем для отслеживания новых категорий, зависит от срока жизни модели. Если вероятность появления новых категорий мала, кодировку обычно делают на всем наборе.

16.2.7. Присвоение категориям, отсортированным по процентной доле наблюдений положительного класса зависимой переменной, целочисленных значений, начиная с 0 (Ordinal Encoding 2)

Для задачи бинарной классификации каждой категории признака, отсортированной по процентной доле наблюдений положительного класса зависимой переменной, сопоставляется целое число, начиная с 0. Для задачи регрессии аналогично каждой категории признака, отсортированной по среднему значению зависимой переменной в этой категории, сопоставляется целое число, начиная с 0. Допустим, у нас есть бинарная зависимая переменная и признак с 4 категориями: A, B, C и D. Сначала посчитаем для каждой категории процентную долю наблюдений, попавших во второй (положительный) класс зависимой переменной: A – 0,25, B – 0,5, C – 0,33, D – 0,66. Потом отсортируем категории по возрастанию: A – 0,25, C – 0,33, B – 0,5, D – 0,66. Затем каждой категории присваиваем порядковый номер, начиная с 0: A – 0, C – 1, B – 2, D – 3.

Данная кодировка реализована в библиотеке h2o под названием SortByResponse.

Переменная Class		
A	0,25	1 из 4
B	0,5	1 из 2
C	0,33	1 из 3
D	0,66	2 из 3

A	0,25	
C	0,33	
B	0,5	
D	0,66	

Переменная Class	Зависимая переменная	Кодировка SortByResponse для переменной Class
A	0	0
A	1	0
C	1	1
D	1	3
A	0	0
A	0	0
B	0	2
B	1	2
C	0	1
D	1	3
C	0	1
D	0	3

Рис. 52 Ordinal Encoding 2

16.2.8. Бинарное кодирование (Binary Encoding)

В основе бинарного кодирования лежит двоичное представление числа, поэтому немного расскажем о нем.

В десятичной системе счисления мы оперируем десятью знаками-цифрами: от 0 до 9. Когда счет достигает числа 9, вводится новый более старший разряд – десятки. При этом разряд единиц обнуляется и счет в этом разряде опять начинается с нуля. После числа 19 разряд десятков увеличивается на 1, а разряд единиц снова обнуляется. Получается число 20. Когда десятки дойдут до 9, впереди них появится третий разряд – сотни.

Формирование каждого последующего числа в двоичной системе счисления аналогично тому, как это происходит в десятичной, за исключением того, что используются всего лишь две цифры: 0 и 1. Как только разряд достигает своего предела, то есть единицы, появляется новый разряд, а старый обнуляется:

```

0
1
10
11
100
101
110
111

```

В ходе бинарного кодирования мы выполняем Ordinal Encoding, т. е. каждую категорию представляем в виде целочисленного значения, используем двоичное представление этого числа (например, значение 5 будет представлено как 101, а значение 10 – как 1010) и создаем столбцы в количестве, соответствующем количеству битов, необходимому для этого двоичного представления.

Давайте создадим игрушечный датафрейм и выполним Ordinal Encoding.

```
# создаем игрушечный датафрейм
toy_df = pd.DataFrame({'Class': ['A', 'B', 'C', 'D', 'E', 'F']})
# выполняем Ordinal Encoding
ord_enc_toy_df = OrdinalEncoder().fit_transform(toy_df)
ord_enc_toy_df
```

Class	
0	1
1	2
2	3
3	4
4	5
5	6

Мы можем составить таблицу с двоичным представлением этих чисел и понять, как будут выглядеть столбцы, сгенерированные для нашего признака.

Число	Двоичное представление
1	1
2	10
3	11
4	100
5	101
6	110

Столбцы для признака		
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0

Рис. 53 Схема бинарного кодирования

Давайте убедимся, выполнив бинарное кодирование с помощью класса `BinaryEncoder` пакета `category_encoders`.

```
# импортируем класс BinaryEncoder из пакета category_encoders
from category_encoders import BinaryEncoder
# выполняем Binary Encoding
bi_enc_toy_df = BinaryEncoder().fit_transform(toy_df)
# взглянем на результаты кодировки
bi_enc_toy_df
```

	Class_0	Class_1	Class_2
0	0	0	1
1	0	1	0
2	0	1	1
3	1	0	0
4	1	0	1
5	1	1	0

Мы можем выполнить `Binary Encoding` до разбиения на обучающую и тестовую выборки (до цикла перекрестной проверки), поскольку не делаем никаких вычислений.

16.2.9. Бинарное кодирование с хешированием (Hashing)

Бинарное кодирование с хешированием предполагает создание хеш-функции $h:U \rightarrow \{1, 2, \dots, B\}$, которая преобразует уровни категориального признака в числа от 1 до B . После этого бинарные признаки можно проиндексировать значениями хеш-функции:

$$g_j(x) = [h(f(x)) = j], j = 1, \dots, B.$$

Нам не нужно хранить соответствия между уровнями категориального признака и индексами бинарных признаков, нам нужно лишь вычислить хеш-функцию, которая автоматически дает правильную индексацию.

Хеширование можно выполнить с помощью класса `HashingEncoder` пакета `category_encoders`.

Хеш-функции имеют не очень приятный побочный эффект: они могут давать одинаковый хеш для двух разных входных элементов (данное явление называется «коллизия»), это обязательно произойдет, если вы, например, попытаетесь представить 1000 различных категорий в виде 10 столбцов.

На этот случай в классе `HashingEncoder` есть параметр `n_components`, который позволяет задавать количество битов для представления признака (по умолчанию используется 8, т. е. 8 бит, для высококардинальных признаков можно задать до 32 бит).

```
# импортируем класс HashingEncoder из пакета category_encoders
from category_encoders import HashingEncoder
# выполняем Hashing Encoding
hash_enc_toy_df = HashingEncoder().fit_transform(toy_df)
hash_enc_toy_df
```

	col_0	col_1	col_2	col_3	col_4	col_5	col_6	col_7
0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1
3	0	0	0	0	0	1	0	0
4	0	0	1	0	0	0	0	0
5	0	0	1	0	0	0	0	0

16.2.10. Взаимодействия

Линейные модели, в отличие от деревьев, не умеют моделировать взаимодействия, поэтому нужно обязательно попробовать создать их, руководствуясь здравым смыслом и бизнес-логикой.

```
# пишем функцию, которая создает взаимодействие
# в результате конъюнкции переменных
# feature1 и feature2
def make_interact(df, feature1, feature2):
    df[feature1 + ' + ' + feature2] = (df[feature1].astype(str) + ' + '
                                         + df[feature2].astype(str))
make_interact(train, 'living_region', 'job_position')
```

```
# еще можно так
train['liv_reg + job_pos'] = train.apply(
    lambda x: f'{x['living_region']} + {x['job_position']}]',
    axis=1)

train.head()
```

	living_region	job_position	open_account_flg	living_region + job_positon	liv_reg + job_pos
0	Московская область	Служащий	0	Московская область + Служащий	Московская область + Служащий
1	Московская область	Заместитель руководителя	1	Московская область + Заместитель руководителя	Московская область + Заместитель руководителя
2	Пермский край	Служащий	1	Пермский край + Служащий	Пермский край + Служащий
3	Московская область	Руководитель	0	Московская область + Руководитель	Московская область + Руководитель
4	Пермский край	Руководитель	1	Пермский край + Руководитель	Пермский край + Руководитель

Здесь мы создали двумя способами 2-факторное взаимодействие. На практике обычно ограничиваются 2-факторными и 3-факторными взаимодействиями.

Взаимодействия обычно создают до разбиения на обучающую и тестовую выборки (до перекрестной проверки), поэтому информацию о пропусках в признаках, участвующих во взаимодействии, переносим и во взаимодействие. А вот способы представления этой информации о пропусках могут быть разные.

Чаще всего, если хотя бы в одной из переменных есть пропуск, во взаимодействие мы записываем пропуск. Такая стратегия используется, когда пропуски редки, как в нашем случае. Здесь значение NaN является временным, после разбиения на обучающую и тестовую выборки мы заменим его какой-нибудь статистикой. Если пропуски встречаются часто, важно сохранить информацию о пропуске как категории, наличие пропуска может быть связано с зависимой переменной (например, холостые мужчины с небольшим стажем хуже выплачивают кредит и предпочитают уклоняться от ответа на вопрос о семейном положении и стаже, оставляют соответствующие поля в анкете пустыми).

Стаж	Семейное положение	Взаимодействие (пропуски редки)	Взаимодействие (пропуски встречаются часто)
менее года	NaN	NaN	менее года + NaN
NaN	NaN	NaN	NaN + NaN
от года до 3 лет	женат	от года до 3 лет + женат	от года до 3 лет + женат
менее года	не женат	менее года + не женат	менее года + не женат
NaN	женат	NaN	NaN + женат

Рис. 54 Разные способы представления пропусков во взаимодействиях

В последнее время для поиска взаимодействий используются ансамбли деревьев решений (строим ансамбль деревьев и ищем самые часто встречающиеся последовательные расщепления, при этом они должны давать наименьшую ошибку прогнозирования и встречаться в верхних уровнях деревьев, т. е. у них должна быть минимальная глубина использования). Для этих задач можно использовать пакеты R `inTrees` и `randomForestExplainer`.

На рисунке ниже приведен фрагмент таблицы взаимодействий признаков и график средней минимальной глубины для 30 самых часто встречающихся взаимодействий, которые можно получить с помощью функций `min_depth_interactions()` и `plot_min_depth_interactions()` соответственно в пакете R `randomForestExplainer`.

Таблица 9 Таблица взаимодействий признаков, которую можно получить с помощью функции `min_depth_interactions()` пакета R `randomForestExplainer` (фрагмент)

	variable	root_variable	mean_min_depth	occurrences	interaction	uncond_mean_min_depth
2	age	atm_user	1.178357	499	atm_user:age	1.70
4	age	cus_leng	1.070140	499	cus_leng:age	1.70
3	age	cre_card	1.214345	496	cre_card:age	1.70
5	age	markpl	1.247848	496	markpl:age	1.70
29	deb_card	cus_leng	2.508329	495	cus_leng:deb_card	3.99
1	age	age	1.567174	494	age:age	1.70

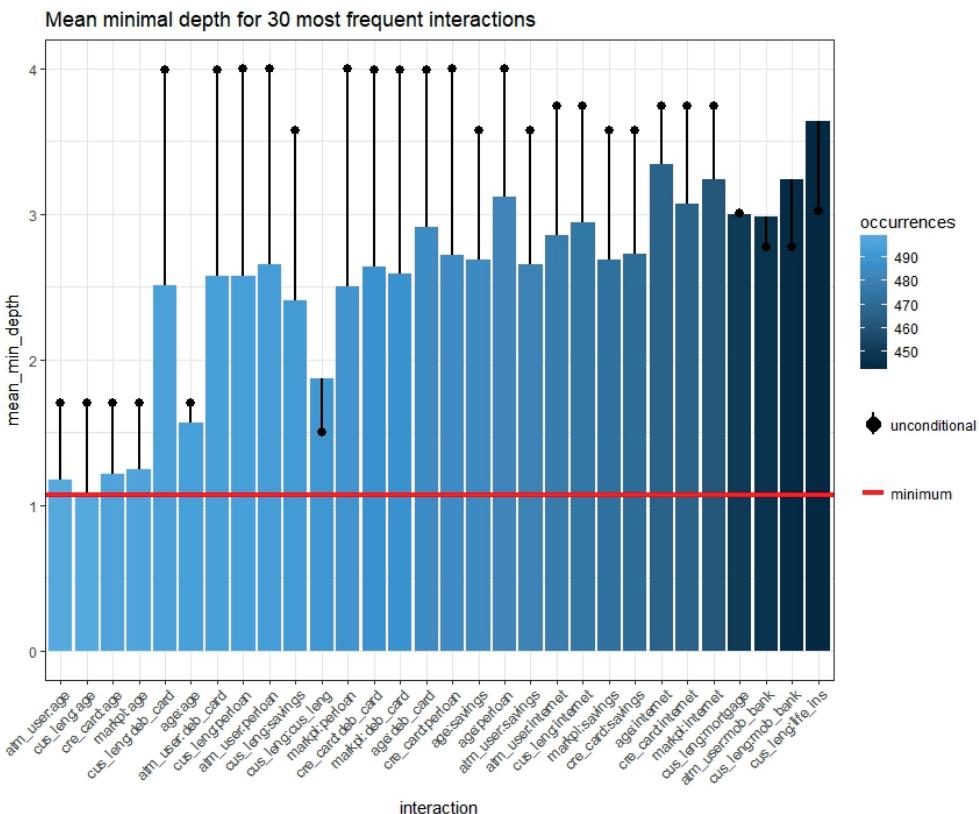


Рис. 55 График средней минимальной глубины для 30 самых часто встречающихся взаимодействий, который можно получить с помощью функции `plot_min_depth_interactions()` пакета R `randomForestExplainer`

Наша задача заключается в том, чтобы искать самые часто встречающиеся взаимодействия, имеющие наименьшую среднюю глубину использования

(здесь используется случайный лес, самые полезные разбиения происходят в верхних уровнях дерева). Наиболее часто встречающимися взаимодействиями являются *atm_user: age*, *cus_leng:age*, *cre_card:age* и *markpl:age*. При этом они имеют наименьшие средние значения минимальной глубины.

16.2.11. Биннинг

Для количественных независимых переменных биннинг – это разбивка диапазона значений переменной на интервалы (бины).

Например, есть переменная *Возраст* с диапазоном значений от 20 до 70 лет, можно разбить на интервалы: от 18 до 30 лет, от 31 года до 50 лет, от 51 года до 70 лет. В итоге получим категориальную переменную, в которой заданные нами интервалы являются категориями.

Для категориальных независимых переменных биннинг – это переназначение (группировка) исходных категорий переменной.

Например, есть переменная *Возраст* с категориями от 18 до 25 лет, от 26 до 35 лет, от 36 до 45 лет, от 46 до 55 лет, от 56 до 65 лет. Категории можно укрупнить, из пяти категорий сделать три: от 18 до 35 лет, от 36 до 55 лет, 56 лет и старше.

Основная причина проведения биннинга – это борьба с нелинейностью при построении скоринговых моделей на основе логистической регрессии. Часто взаимосвязь между непрерывной переменной и событием является нелинейной. Уравнение логистической регрессии, несмотря на то что ее выходное значение подвергается нелинейному преобразованию путем логита, все равно моделирует линейные зависимости между признаками и зависимой переменной.

Для иллюстрации можно взять пример с нелинейной зависимостью между возрастом и событием (например, оттоком). Допустим, рассчитанный регрессионный коэффициент в уравнении логистической регрессии получился отрицательным. Это значит, что вероятность оттока с возрастом уменьшается. После проведенного биннинга, когда были выделены категории до 26 лет, от 26 до 36 лет, от 36 до 45 лет и от 45 лет, оказалось, что зависимость между возрастом и событием нелинейная. Первая (молодые) и последняя (старший возраст) категории склонны к оттоку, а промежуточные сегменты, наоборот, не склонны к оттоку.

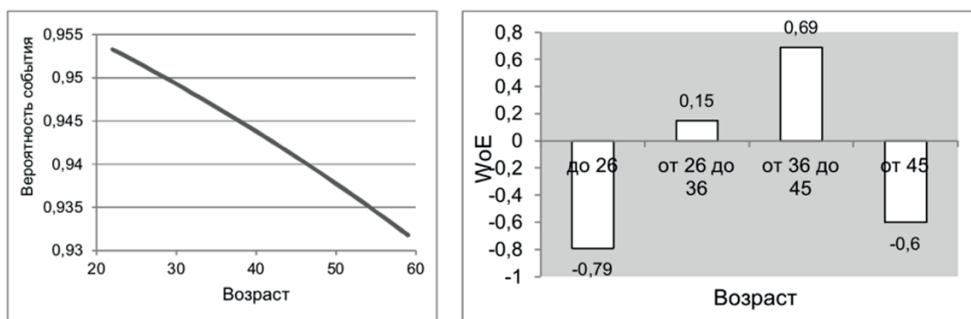


Рис. 56 Моделирование нелинейности с помощью биннинга

Однако у биннинга имеются и серьезные недостатки. Биостатистик Фрэнк Харрелл приводит ряд причин, по которым не следует проводить биннинг количественных независимых переменных:

- потеря прогнозной силы переменной в силу снижения ее информативности (вспомним, что наиболее полную информацию несет количественная шкала);
- в основе биннинга лежит некорректное предположение о том, что зависимость между признаком и откликом внутри интервалов является монотонной (это предположение еще менее разумно, чем предположение о линейности);
- при разбиении всего диапазона значений переменной на интервалы с равным количеством наблюдений первый и последний интервалы могут оказаться очень широкими, потому что плотность распределения в них может быть низкой (если взять, например, нормально распределенную величину);
- очевидный субъективизм категоризации, выражющийся в том, что если нескольким исследователям предложить категоризировать переменную, они выберут разные границы интервалов.

В силу недостатков, изложенных ниже, биннинг как инструмент борьбы с нелинейностью используется все реже и уступает место преобразованиям на основе ограниченных кубических сплайнов, логарифма, квадратного корня.

Чаще всего применяют биннинг:

- на основе интервалов одинаковой ширины;
- интервалов, заданных вручную (согласно бизнес-логике);
- на основе интервалов с одинаковым количеством наблюдений – квантилей (децилей, виджинтилей, квинтилей, квартилей);
- на основе зависимой переменной (здесь выделяют биннинг на основе WoE и IV и биннинг на основе CHAID с использованием критерия хи-квадрат или F-критерия).

16.2.11.1. Биннинг на основе интервалов, созданных вручную или одинаковой ширины

В питоновской библиотеке pandas биннинг на основе интервалов одинаковой ширины или интервалов, заданных вручную, выполняется с помощью функции `cut()`. Функция `cut()` имеет общий вид:

```
pandas.cut(x, bins, right=True, labels=None,
            precision=3, include_lowest=False)
```

где

<code>x</code>	Задает 1-мерный входной массив для биннинга
<code>bins</code>	Задает правило биннинга: целочисленное значение – определяет количество бинов одинаковой ширины; последовательность скаляров – определяет точки разбиения, допускается разная ширина бинов; <code>IntervalIndex</code> – определяет точные границы бинов

<code>right</code>	Закрывает интервалы справа, если задано значение True (по умолчанию), либо закрывает интервалы слева, если задано значение False
<code>labels</code>	Задает метки бинов. Должен иметь длину, совпадающую с количеством бинов. Если задано значение False (по умолчанию), возвращает числовые метки бинов
<code>precision</code>	Задает точность, используемую для хранения и отображения числовых меток бинов
<code>include_lowest</code>	Если задано значение True, включает самое нижнее значение точек разбиения

Давайте загрузим данные.

```
# загружаем и смотрим данные
dev = pd.read_csv('Data/Stat_FE3.csv', sep=';')
dev.head()
```

	<code>tariff_id</code>	<code>credit_sum</code>	<code>monthly_income</code>	<code>open_account_flg</code>
0	1_4	33579.0	36000.0	0
1	1_32	23511.0	45000.0	0
2	1_5	39990.0	50000.0	0
3	1_3	3490.0	35000.0	0
4	1_6	36358.0	50000.0	0

На основе признака `monthly_income` создаем новый признак, используем биннинг на основе интервалов, заданных вручную.

```
# задаем точки, в которых будут находиться границы интервалов
# (до 50 000, от 50 000 до 200 000, от 200 000 и выше)
bins = [-np.inf, 50000, 200000, np.inf]
# задаем метки для категорий будущей переменной
group_names = ['Low', 'Average', 'High']
# осуществляем биннинг переменной monthly_income
# и записываем результаты в новую переменную incomecat
dev['incomecat'] = pd.cut(dev['monthly_income'], bins,
                           labels=group_names)
```

посмотрим частоты категорий нового признака

```
dev['incomecat'].value_counts()
```

Low	96522
Average	22803
High	197
Name:	incomecat, dtype: int64

Теперь на основе признака `monthly_income` создаем еще один новый признак, на этот раз используем биннинг на основе интервалов одинаковой ширины.

```
# а теперь выполним биннинг на основе
# интервалов одинаковой ширины
dev['incomecat2'] = pd.cut(dev['monthly_income'], 10)
# посмотрим частоты категорий нового признака
dev['incomecat2'].value_counts()

(4105.0, 94500.0]      115748
(94500.0, 184000.0]    3438
(184000.0, 273500.0]   224
(273500.0, 363000.0]   77
(363000.0, 452500.0]   17
(452500.0, 542000.0]   7
(542000.0, 631500.0]   4
(631500.0, 721000.0]   4
(721000.0, 810500.0]   2
(810500.0, 900000.0]   1
Name: incomecat2, dtype: int64
```

Обратите внимание на квадратные и круглые скобки интервалов. Интервал закрывается либо слева, либо справа, то есть соответствующий конец включается в данный интервал.

Согласно принятой в математике нотации интервалов круглая скобка означает, что соответствующий конец не включается (открыт), а квадратная – что включается (закрыт). По умолчанию интервалы открыты слева и закрыты справа. Интервал (4105.0, 94500.0] содержит наблюдения, в которых значение переменной больше 4105, но при этом меньше или равно 94 500.

16.2.11.2. Биннинг на основе квантилей

В питоновской библиотеке pandas биннинг на основе квантилей выполняется с помощью функции `qcut()`. Функция `qcut()` имеет общий вид:

```
pandas.qcut(x, q, labels=None, retbins=False,
            precision=3, duplicates='raise')
```

где

<code>x</code>	Задает 1-мерный входной массив или серию для биннинга
<code>q</code>	Задает количество квантилей (целое число или массив квантилей)
<code>labels</code>	Задает метки бинов. Должен иметь длину, совпадающую с количеством бинов. Если задано значение <code>False</code> (по умолчанию), возвращает числовые метки бинов
<code>retbins</code>	Возвращает бины или метки
<code>precision</code>	Задает точность, используемую для хранения и отображения числовых меток бинов
<code>duplicates</code>	Если границы бинов не уникальны, выдает <code>ValueError</code> или удаляет их

На основе признака `monthly_income` создаем новый признак, используем биннинг на основе квартилей. Напомним: квартили делят упорядоченный ряд на четыре равнодолженные части.

```
# осуществляем биннинг переменной monthly_income
# на основе квартилей и записываем результаты
# в новую переменную income_quartile
dev['income_quartile'] = pd.qcut(dev['monthly_income'], 4)
# посмотрим частоты категорий нового признака
dev['income_quartile'].value_counts(normalize=True)

(4999.999, 25000.0]    0.289880
(35000.0, 50000.0]    0.260714
(25000.0, 35000.0]    0.256974
(50000.0, 90000.0]    0.192433
Name: income_quartile, dtype: float64
```

16.2.11.3. Биннинг на основе зависимой переменной

В ходе биннинга нам нужно создать не просто признаки, а признаки, обладающие высокой прогнозной силой. Для задачи бинарной классификации эта прогнозная сила выражается в высокой способности отличать отрицательный класс зависимой переменной от положительного класса. Для этого при выполнении биннинга ориентируются на два показателя – WoE и IV. При этом биннинг на основе WoE и IV нужен только для логистической регрессии.

Представьте себе, у нас есть признак *credit_sum*, и на его основе мы хотим создать новый категориальный признак *credsumcat*. Давайте зададим границы категорий и выполним биннинг.

```
# взглянем на минимальное и максимальное значения
print(dev['credit_sum'].min())
print(dev['credit_sum'].max())

2736.0
200000.0

# задаем точки, в которых будут находиться границы
# категорий будущего признака credsumcat
bins = [-np.inf, 10000, 30000, 50000, np.inf]
# осуществляем биннинг признака credit_sum и записываем
# результаты в новый признак credsumcat
dev['credsumcat'] = pd.cut(dev['credit_sum'], bins)
```

Теперь построим самую простую таблицу сопряженности между новым признаком *credsumcat* и зависимой переменной *open_account_flg*.

```
# строим таблицу сопряженности credsumcat * open_account_flg
biv = pd.crosstab(dev['credsumcat'], dev['open_account_flg'])

open_account_flg      0      1
credsumcat
(-inf, 10000.0]    7378  2825
(10000.0, 30000.0] 62408 13428
(30000.0, 50000.0] 17921  3299
(50000.0, inf]   10796  1467
```

Исходя из этой таблицы, мы можем вычислить «вес» каждой категории. В кредитном скоринге *WoE* (от *weight of evidence*), или вес категории, вычисляется по формуле:

$$WoE_i = \ln\left(\frac{F_i^1}{F_i^0}\right) = \ln\left(\frac{\% \text{ of } events_i}{\% \text{ of } non-events_i}\right),$$

где

i – категория переменной;

\ln – натуральный логарифм;

F_i^0 – относительная частота класса 0 (отрицательного класса) в категории;

F_i^1 – относительная частота класса 1 (положительного класса) в категории.

Наблюдения отрицательного класса часто называют не-событиями, а наблюдения положительного класса – событиями.

Вычислим вручную WoE для категории (-inf, 10000.0]. Относительная частота класса 0 в этой категории равна $7378 / (7378 + 62\,408 + 17\,921 + 10\,796)$, или $7378 / 98\,503 = 0,075$. Относительная частота класса 1 равна $2825 / (2825 + 13\,428 + 3299 + 1467)$, или $2825 / 21\,019 = 0,134$. Отношение частот равно $0,134 / 0,075 = 1,79$. Натуральный логарифм этого отношения $\ln(1,79) = 0,582$.

Итак, WoE измеряет предсказательную силу каждой категории или сгруппированной категории с точки зрения способности отличать класс 0 от класса 1. Здесь *отрицательные* числа будут обозначать, что отдельно взятая категория выделяет *большую пропорцию представителей класса 0 (отрицательного класса)*, чем представителей класса 1 (положительного класса). *Положительные* числа будут обозначать, что отдельно взятая категория выделяет *большую пропорцию представителей класса 1 (положительного класса)*, чем представителей класса 0 (отрицательного класса). В случае с категорией (-inf, 10000.0] мы видим положительное значение WoE, т. к. относительная частота положительного класса (0,134) больше относительной частоты отрицательного класса (0,075).

Существует еще одна интерпретация WoE. Если WoE для категории равно 0, то среднее значение зависимой переменной в категории равно среднему значению зависимой переменной в выборке. Если значение WoE для категории положительно, то среднее значение зависимой переменной в категории больше среднего значения зависимой переменной в выборке. Если значение WoE для категории отрицательно, то среднее значение зависимой переменной в категории меньше среднего значения зависимости переменной в выборке.

При работе с WoE нужно придерживаться четырех правил:

- необходимо создавать не более 10 категорий;
- пропущенные значения группируются в отдельную категорию;
- каждая категория должна содержать не менее 5 % наблюдений;
- категории не должны содержать нулевого количества событий или не-событий.

Значения WoE, как и процент «плохих», должны в достаточной мере отличаться по группам. Группировка выполняется так, чтобы максимизировать разницу между представителями класса 0 или представителями класса 1. Мы должны выявить и отделить категории, которые хорошо дифференцируют

клиентов. Категории со схожими значениями WoE объединяют, потому что такие категории содержат практически одинаковое количество представителей класса 0 и представителей класса 1 и демонстрируют одинаковое «поведение». Если WoE сохраняет монотонность как для небольших, так и для крупных категорий, выбирайте более крупные категории. Несмотря на то что абсолютное значение WoE важно, разница между WoE групп играет ключевую роль. Чем больше разница между смежными категориями, тем выше прогнозная сила данной переменной. Аналогичная идея используется в методе CHAID: мы укрупняем категории переменной так, чтобы они максимально отличались друг от друга по зависимой переменной, и тем самым добиваемся максимизации взаимосвязи между признаком и зависимой переменной, но поскольку выходом логистической регрессии является логит, мы берем натуральный логарифм отношения процента «событий» к проценту «не-событий».

В идеале WoE непропущенных значений должно быть монотонным, восходя от отрицательных значений к положительным или, наоборот, без смены зависимости на обратную. Это обусловлено тем, что хотя логистическая регрессия не предполагает линейной связи между зависимой переменной и признаком, она требует линейную связь между признаком и логарифмом шансов.

Давайте автоматически вычислим WoE для каждой категории признака *credsumcat*.

```
# пишем функцию, которая вычисляет WoE для
# каждой категории выбранного признака,
# добавляем a = 0.0001, чтобы избежать деления на 0
def WoE(df, feature, target):
    biv = pd.crosstab(df[feature], df[target])
    a = 0.0001
    WoE = (np.log((biv[1] / sum(biv[1]) + a) /
                  (biv[0] / sum(biv[0]) + a)))
    return WoE

# вычисляем WoE для каждой категории признака credsumcat
WoE(dev, 'credsumcat', 'open_account_flg')

credsumcat
(-inf, 10000.0]    0.584076
(10000.0, 30000.0]      0.008308
(30000.0, 50000.0]     -0.147606
(50000.0, inf]       -0.450776
dtype: float64
```

Поскольку мы используем логарифм, то размещаем полученные категории в логарифмической шкале, которая является естественной для логистической регрессии.

Теперь вычислим среднее значение зависимой переменной в выборке и средние значения зависимой переменной в категориях переменной *credsumcat*.

```
# вычислим значение зависимой переменной в выборке
dev['open_account_flg'].mean()

0.175858837703519
```

```
# вычислим средние значения зависимой переменной
# в категориях переменной credsumcat
dev.groupby('credsumcat')['open_account_flg'].apply(lambda x: x.mean())

credsumcat
(-inf, 10000.0]      0.276879
(10000.0, 30000.0]   0.177066
(30000.0, 50000.0]   0.155467
(50000.0, inf]       0.119628
Name: open_account_flg, dtype: float64
```

Вспомним интерпретацию WoE. Если значение WoE для категории положительно, то среднее значение зависимой переменной в категории больше среднего значения зависимой переменной в выборке. Если значение WoE для категории отрицательно, то среднее значение зависимой переменной в категории меньше среднего значения зависимой переменной в выборке.

```
credsumcat
(-inf, 10000.0]      0.584076      0.277 > 0.176
(10000.0, 30000.0]   0.008308      0.177 > 0.176
(30000.0, 50000.0]   -0.147606     0.155 < 0.176
(50000.0, inf]       -0.450776     0.119 < 0.176
dtype: float64
```

Для проверки качества биннинга с помощью WoE можно построить логистическую регрессию с одним WOE-трансформированным признаком. Если регрессионный коэффициент не равен 1 и свободный член не равен ln (процент «событий» / процент «не-событий»), то биннинг выполнен некорректно. Это обусловлено тем, что когда мы строим логистическую регрессию с одним WOE-трансформированным признаком, оценка максимального правдоподобия имеет явное решение⁵:

$$\beta_0 = \ln\left(\frac{\% \text{ of events}}{\% \text{ of non-events}}\right) \text{ и } \beta_1 = 1.$$

Давайте проверим качество биннинга переменной *credsumcat*.

```
# выполняем WoE-трансформацию переменной credsumcat
woe_values = WoE(dev, 'credsumcat', 'open_account_flg')
dev['experiment'] = dev['credsumcat'].map(woe_values)

# строим логистическую регрессию с WoE-трансформированной переменной
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(solver='liblinear').fit(
    dev[['experiment']], dev['open_account_flg'])

# удаляем WoE-трансформированную переменную
dev.drop('experiment', axis=1, inplace=True)

# печатаем значение константы и коэффициента
intercept = np.round(np.asscalar(logreg.intercept_), 3)
coef = np.round(np.asscalar(logreg.coef_), 3)

print('константа:', intercept)
print('коэффициент:', coef)
```

⁵ Guoping Zeng. A Necessary Condition for a Good Binning Algorithm in Credit Scoring.

константа: -1.545

коэффициент: 1.0

```
# вычислим процент не-событий и событий
percents = (dev['open_account_flg'].value_counts() /
             len(dev['open_account_flg']))
percents

0 0.824141
1 0.175859
Name: open_account_flg, dtype: float64

# убедимся, что константа - натуральный логарифм
# отношения процента событий к не-событиям

import math
number = percents[1] / percents[0]
base = 2.72
math.log(number, base)

-1.5436847899687964
```

Убедились в том, что биннинг выполнен корректно.

Итак, с помощью WoE мы категоризировали переменную так, чтобы она максимально эффективно отличала один класс от другого, и могли бы ее по-дать на вход модели, но надо убедиться в том, насколько она будет полезна по сравнению с остальными переменными. Для этого используется IV (от information value), или информационное значение. Его можно вычислить для отдельной категории и для всей переменной. Информационное значение для категории вычисляется как разность между относительной частотой класса 1 и относительной частотой класса 0 в данной категории, умноженная на натуральный логарифм отношения этих частот.

$$IV_i = \left(F_i^1 - F_i^0 \right) \times \ln \left(\frac{F_i^1}{F_i^0} \right).$$

Вычислим информационное значение для категории (-inf, 10000.0]. Разность между относительными частотами равна 0,134 – 0,075 = 0,059. Информационное значение равно $0,059 \times \ln(0,56) = 0,059 \times 0,584 = 0,034$.

Давайте напишем функцию, которая будет вычислять IV для каждой категории выбранного признака.

```
# пишем функцию, которая вычисляет IV для каждой категории
# выбранного признака, добавляем a = 0.0001, чтобы
# избежать деления на 0
def IV_cat(df, feature, target):
    biv = pd.crosstab(df[feature], df[target])
    a = 0.0001
    IV_cat = ((biv[1] / sum(biv[1]) + a) -
               (biv[0] / sum(biv[0]) + a)) * np.log(
                   (biv[1] / sum(biv[1]) + a) / (biv[0] / sum(biv[0]) + a))
    return IV_cat
```

```
# вычисляем IV для каждой категории признака credsumcat
IV_cat(dev, 'credsumcat', 'open_account_flg')
```

```
credsumcat
(-inf, 10000.0]    0.034753
(10000.0, 30000.0]    0.000044
(30000.0, 50000.0]    0.003687
(50000.0, inf]    0.017944
dtype: float64
```

Итоговое информационное значение используется для измерения прогнозной силы переменной в целом, для этого информационные значения, вычисленные по каждой категории, складываются.

$$\text{IV} = \sum_{i=1}^k \left(F_i^1 - F_i^0 \right) \times \ln \left(\frac{F_i^1}{F_i^0} \right).$$

Информационное значение всегда является положительной величиной. При интерпретации итоговых значений IV руководствуются правилом (по Наиму Сиддики):

- меньше 0,02 – характеристика не обладает предсказательной способностью;
- от 0,02 до 0,1 – слабая предсказательная способность;
- от 0,1 до 0,3 – средняя предсказательная способность;
- 0,3 и выше – высокая предсказательная способность.

На практике всецело полагаться на итоговое информационное значение для оценки прогнозной силы переменной не стоит.

Итоговое информационное значение зависит от количества категорий / уникальных значений (чем больше категорий / уникальных значений, тем больше будет IV) и поэтому может быть произвольно высоким. Распространенная ошибка заключается в создании переменной с большим количеством небольших по размеру категорий. Это позволяет получить высокое информационное значение, на основании чего делается ошибочный вывод, что переменная обладает высокой прогнозной силой. Если использовать много категорий, итоговое информационное значение возрастет, но с практической точки зрения будет бесполезным, потому что будет измерять шум.

Итоговое информационное значение может маскировать проблемы. У вас могут быть две категоризированные переменные, у первой переменной может быть нарушена монотонность, встречаться редкие категории, категории с практически нулевым количеством не-событий/событий, а у другой переменной такие проблемы могут отсутствовать, но при этом обе переменные могут иметь одинаковое итоговое IV.

Поэтому наша задача – получить максимальное информационное значение, при этом выполнив четыре вышеперечисленных правила и внимательно проанализировав WoE.

Напишем функцию, которая будет вычислять итоговое значение IV для признака.

```
# пишем функцию, которая вычисляет итоговое значение IV
# для выбранного признака, добавляем a = 0.0001,
# чтобы избежать деления на 0
def IV(df, feature, target):
    biv = pd.crosstab(df[feature], df[target])
    a = 0.0001
    IV = sum(((biv[1] / sum(biv[1]) + a) -
              (biv[0] / sum(biv[0]) + a)) * np.log(
        (biv[1] / sum(biv[1]) + a) / (biv[0] / sum(biv[0]) + a)))
    return IV

# вычисляем итоговое IV для переменной credsumcat
IV(dev, 'credsumcat', 'open_account_flg')

0.05642812895354752
```

Сейчас переменную *credsumcat* мы создавали на основе переменной *credit_sum*. Здесь мы взяли переменную *credit_sum* без использования какой-то априорной информации о ее прогнозной силе. Следует помнить, что целесообразно выполнять биннинг сильных переменных, если категоризировать слабую переменную, то и категоризированная переменная будет слабой. Поэтому IV часто используется для сравнительной оценки прогнозной силы переменных. Просто взять количественные переменные и по ним вычислить IV мы не можем, выше мы уже говорили, что IV зависит от количества категорий (уникальных значений). Поэтому на практике поступают так: каждую переменную делят на 10 квантилей – групп с примерно одинаковым количеством наблюдений – и уже по такой переменной измеряют IV.

Давайте напишем функцию, которая автоматически вычислит IV по всем количественным переменным, у которых больше 10 уникальных значений.

```
# пишем функцию, вычисляющую IV по всем
# количественным признакам
def numeric_IV(df):
    # создаем список, в который будем записывать IV
    iv_list = []
    # создаем копию датафрейма
    df = df.copy()
    # записываем константу, которую будем добавлять,
    # чтобы избежать деления на 0
    a = 0.0001
    # задаем зависимую переменную
    target = df['open_account_flg']
    # отбираем столбцы, у которых больше 10 уникальных значений
    df = df.loc[:, df.apply(pd.Series.nunique) > 10]
    # из этих столбцов отбираем только количественные
    numerical_columns = df.select_dtypes(include=['number']).columns
    # запускаем цикл, который вычисляет IV
    # по каждому выбранному признаку
    for var_name in numerical_columns:
        # разбиваем признак на 10 квантилей
        df[var_name] = pd.qcut(df[var_name].values, 10,
                               duplicates='drop').codes
    # строим таблицу сопряженности между категоризированным
    # признаком и зависимой переменной
```

```

biv = pd.crosstab(df[var_name], target)
# вычисляем IV на основе таблицы сопряженности
IV = sum(((biv[1] / sum(biv[1]) + a) -
          (biv[0] / sum(biv[0]) + a)) * np.log(
    (biv[1] / sum(biv[1]) + a) / (biv[0] / sum(biv[0]) + a)))
# добавляем вычисленное IV в список, где хранятся IV
iv_list.append(IV)
# создаем датафрейм с двумя столбцами, в одном - названия
# признаков, в другом - IV этих переменных
result = pd.DataFrame({'Название переменной': numerical_columns,
                       'IV': iv_list})
# добавляем дополнительный столбец «Полезность»
# градация по Наиму Сиддики
result['Полезность'] = ['Подозрительно высокая' if x > 0.5 else 'Сильная'
                        if x <= 0.5 and x > 0.3 else 'Средняя'
                        if x <= 0.3 and x > 0.1 else 'Слабая'
                        if x <= 0.1 and x > 0.02 else 'Бесполезная'
                        for x in result['IV']]
# возвращаем датафрейм, отсортированный по убыванию IV
return (result.sort_values(by='IV', ascending=False))

```

Применяем написанную нами функцию.

```
# вычисляем итоговые IV по признакам
numerical_IV(dev)
```

Название переменной	IV	Полезность
0 credit_sum	0.065526	Слабая
1 monthly_income	0.005984	Бесполезная

Можно выполнить биннинг по WoE с помощью пакета PyWoE, написанного Денисом Суржко (<https://github.com/Densur/PyWoE>).

Давайте импортируем необходимые классы.

```
# импортируем необходимые классы
from woe import *
```

Сейчас мы должны создать модель – экземпляр класса WoE, задав:

- максимально возможное количество бинов (гиперпараметр qnt_num);
- минимальное количество наблюдений в бине (гиперпараметр min_block_size);
- тип признака (параметр v_type, значение 'c' задается для количественного признака, значение 'd' задается для дискретного признака);
- тип зависимой переменной (параметр t_type, значение 'c' задается для количественной зависимой переменной, значение 'b' задается для бинарной зависимой переменной).

```
# создаем модель – экземпляр класса WoE, задаем максимально возможное
# количество бинов, минимальное количество наблюдений в
# бине, тип признака, тип зависимой переменной
woe = WoE(qnt_num=10, min_block_size=10, v_type='c', t_type='b')
```

Обучаем созданную модель, т. е. вычисляем WoE.

```
# обучаем модель – вычисляем WoE
woe.fit(dev['credit_sum'], dev['open_account_flg']);
```

Теперь применяем обученную модель – выполняем WoE-трансформацию переменной *credit_sum* и выводим информацию о бинах.

```
# выполняем WoE-трансформацию переменной credit_sum
woe.transform(dev['credit_sum'])
# уменьшаем количество знаков после десятичной точки pd.set_option('display.float_format',
lambda x: '%.3f' % x)
# выводим информацию о бинах
print(woe.bins)
```

	mean	bad	obs	good	woe	bins	labels
0	0.274	2976	10871	7895	-0.569	-inf	0
1	0.168	1826	10879	9053	0.056	9.239	1
2	0.151	1639	10849	9210	0.182	9.481	2
3	0.177	1928	10866	8938	-0.011	9.639	3
4	0.200	2168	10866	8698	-0.155	9.758	4
5	0.196	2125	10864	8739	-0.131	9.895	5
6	0.182	1974	10871	8897	-0.039	10.038	6
7	0.162	1759	10878	9119	0.101	10.180	7
8	0.181	1965	10846	8881	-0.036	10.324	8
9	0.127	1380	10866	9486	0.383	10.602	9
10	0.118	1279	10866	9587	0.470	10.869	10

	mean	bad	obs	good	woe	bins	labels
0	0.274	2976	10871	7895	-0.569	-inf	0
1	0.168	1826	10879	9053	0.056	9.239	1
2	0.151	1639	10849	9210	0.182	9.481	2
3	0.177	1928	10866	8938	-0.011	9.639	3
4	0.200	2168	10866	8698	-0.155	9.758	4
5	0.196	2125	10864	8739	-0.131	9.895	5
6	0.182	1974	10871	8897	-0.039	10.038	6
7	0.162	1759	10878	9119	0.101	10.180	7
8	0.181	1965	10846	8881	-0.036	10.324	8
9	0.127	1380	10866	9486	0.383	10.602	9
10	0.118	1279	10866	9587	0.470	10.869	10

нарушение
монотонности

В выводе по каждому бину приводятся следующие показатели:

- mean – отношение количества наблюдений положительного класса (bad) к общему количеству наблюдений (obs);
- bad – количество наблюдений положительного класса;
- obs – общее количество наблюдений;
- good – количество наблюдений отрицательного класса;
- WoE – WoE;
- bins – нижняя граница категории;
- labels – целочисленная метка.

С помощью метода `.force_monotonic()` можно улучшить монотонность. Параметр `hypothesis` метода `.force_monotonic()` задает гипотезу о взаимосвязи между признаком и зависимой переменной (0 – прямая, 1 – обратная).

```
# улучшаем монотонность
woe_monotonic = woe.force_monotonic(hypothesis=0)
print(woe_monotonic.bins)
```

	mean	bad	obs	good	woe	bins	labels
0	0.274	2976	10871	7895	-0.569	-inf	0
1	0.179	11660	65195	53535	-0.020	9.239	1
2	0.171	3724	21724	18000	0.031	10.180	2
3	0.127	1380	10866	9486	0.383	10.602	3
4	0.118	1279	10866	9587	0.470	10.869	4

С помощью метода `.optimize()` мы можем применить метод деревьев решений CART для биннинга переменной по WoE, максимизирующего определенную метрику качества (для валидации по умолчанию используется 3-блочная перекрестная проверка). Для метода `.optimize()` есть следующие гиперпараметры: гиперпараметр `criterion` задает критерий разбиения узлов в дереве, гиперпараметр `max_depth` задает дерево фиксированной глубины ($2^{\text{max_depth}}$ бинов), гиперпараметр `min_samples_leaf` задает минимальное количество наблюдений в каждом из оптимизируемых бинов, параметр `cv` задает количество блоков перекрестной проверки, параметр `scoring` задает оптимизируемую метрику.

```
# выполняем биннинг по WoE с оптимизацией по AUC  
# (используется дерево CART)  
woe2 = woe.optimize(max_depth=3, scoring='roc_auc', cv=5)  
print(woe2.bins)
```

	mean	bad	obs	good	woe	bins	labels
0	0.317	1460	4605	3145	-0.777	-inf	0
1	0.243	1509	6206	4697	-0.409	8.893	1
2	0.177	15652	88557	72905	-0.006	9.237	2
3	0.119	2398	20154	17756	0.457	10.652	3

Биннинг на основе метода CHAID можно выполнить с помощью пакета CHAID. Этот пакет можно установить с помощью команды `pip install CHAID`.

Выполним для категориальной переменной `tariff_id` биннинг на основе CHAID.

```
([], {0: 98503.0, 1: 21019.0}, {tariff_id, p=0.0, score=4849.043527542213, groups=[['1_0', '1_9'], ['1_1', '1_17'], ['1_16', '1_5', '1_2', '1_7', '1_94'], ['1_19', '1_4'], ['1_21', '1_22', '1_23'], ['1_24', '1_25', '1_6'], ['1_3', '1_41', '1_91'], ['1_32', '1_99'], ['1_43', '1_44']]), dof=8))
-- ([1_0', '1_9'], {0: 5069.0, 1: 359.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_1', '1_17'], {0: 39687.0, 1: 9244.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_16', '1_5', '1_2', '1_7', '1_94'], {0: 8504.0, 1: 940.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_19', '1_4'], {0: 8060.0, 1: 1053.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_21', '1_22', '1_23'], {0: 906.0, 1: 35.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_24', '1_25', '1_6'], {0: 23847.0, 1: 3934.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_3', '1_41', '1_91'], {0: 1879.0, 1: 774.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_32', '1_99'], {0: 6816.0, 1: 4132.0}, <Invalid Chaid Split> - the max depth has been reached)
-- ([1_43', '1_44'], {0: 3735.0, 1: 548.0}, <Invalid Chaid Split> - the max depth has been reached)
```

Первая строка вывода начинается с информации о частотах классов зависимой переменной `{0: 98503.0, 1: 21019.0}`, значение `p` показывает статистическую значимость, `score` показывает значение хи-квадрат, `groups` показывает полученные категории. Далее приводятся узлы – укрупненные категории и распределение классов зависимой переменной в каждом узле. Также выводятся предупреждения, сообщающие, какое из правил остановки сработало: для большинства узлов приводится предупреждение `the max depth has been reached` – достигнута максимальная глубина, это неудивительно, ведь мы выбрали глубину 1 (дерево будет иметь один уровень, лежащий ниже корневого узла). Для одного из узлов выведено предупреждение `the minimum parent node size threshold has been reached` – достигнут порог по минимальному размеру родительского узла, это обусловлено тем, что по умолчанию минимальное количество наблюдений в разбиваемом узле (регулируется параметром `min_parent_node_size`) должно быть не менее 30, а наш узел содержит всего 23 наблюдения.

Укрупненные категории необходимо создавать, придерживаясь тех же самых правил, что и при биннинге количественных переменных:

- количество категорий не должно превышать 10;
- каждая категория должна содержать не менее 5 % наблюдений;
- категории не должны содержать нулевого количества событий или несобытий.

Для оценки прогнозной силы укрупненных категорий и новой переменной можно ориентироваться на WoE и IV.

Биннинг на основе CHAID используют для осмысленного укрупнения редких категорий (если нужно обосновать перед регулятором).

Обратите внимание, что нельзя создать с помощью биннинга новую переменную на общем наборе данных, а потом разбить набор на обучающую и тестовую выборки и работать с такой переменной в соответствующей выборке, как с обычной исторической переменной. Это обусловлено тем, что для биннинга используется информация о распределении значений переменной по всему набору данных. В результате получится, что в тестовой выборке мы будем использовать переменную, категории которой были получены, исходя из информации всего набора данных. Важно понять, что биннинг – это минимодель, которую мы строим на обучающей выборке. В итоге получаем параметры – правила дискретизации (для количественных переменных) и правила перегруппировки (для категориальных переменных), которые применяются

к соответствующей переменной в обучающей и тестовой выборках. Поэтому биннинг делается после разбиения на обучающую и тестовую выборки (внутри цикла перекрестной проверки).

16.2.12. Добавление меток кластеров на основе кластеризации по методу k -средних

С помощью кластеризации по методу k -средних мы можем получить метки кластеров и добавить их в качестве новых признаков. Давайте загрузим данные, записанные в файле `winequality-red.csv`.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/winequality-red.csv', sep=';')
data.head(3)
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.400	0.700	0.000	1.900	0.076	11.000	34.000	0.998	3.510	0.560	9.400
1	7.800	0.880	0.000	2.600	0.098	25.000	67.000	0.997	3.200	0.680	9.800
2	7.800	0.760	0.040	2.300	0.092	15.000	54.000	0.997	3.260	0.650	9.800

Набор включает в себя следующие переменные:

- количественный признак *Фиксированная кислотность [fixed acidity]*;
- количественный признак *Летучая кислотность [volatile acidity]*;
- количественный признак *Лимонная кислота [citric acid]*;
- количественный признак *Остаточный сахар [residual sugar]*;
- количественный признак *Хлориды [chlorides]*;
- количественный признак *Свободный диоксид серы [free sulfur dioxide]*;
- количественный признак *Общий диоксид серы [total sulfur dioxide]*;
- количественный признак *Плотность [density]*;
- количественный признак *Водородный показатель [pH]*;
- количественный признак *Содержание сульфатов [sulphates]*;
- количественный признак *Содержание алкоголя [alcohol]*;
- количественная зависимая переменная *Оценка вина по десятибалльной шкале [quality]*.

Теперь создадим массив признаков и массив меток, а также копию массива признаков. Обратите внимание: кластерный анализ является методом машинного обучения без учителя, поэтому массив меток, выступающий как раз в качестве учителя, нам не понадобится.

```
# создаем массив меток и массив признаков
label = data.pop('quality')
# создаем копию массива признаков
data_copy = data.copy()
data.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.400	0.700	0.000	1.900	0.076	11.000	34.000	0.998	3.510	0.560	9.400
1	7.800	0.880	0.000	2.600	0.098	25.000	67.000	0.997	3.200	0.680	9.800
2	7.800	0.760	0.040	2.300	0.092	15.000	54.000	0.997	3.260	0.650	9.800

Необходимо помнить, что метод k -средних, как и многие методы кластерного анализа, требует, чтобы переменные имели один и тот же масштаб (поскольку чаще всего используется евклидова метрика и из-за разных масштабов можно получить непредсказуемые результаты). Кроме того, желательно, чтобы переменные имели нормальное распределение (это позволяет избежать получения кластеров с одним наблюдением-выбросом).

Давайте выполним стандартизацию, при этом будем работать через список признаков, чтобы вернуть датафрейм.

```
# импортируем класс MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
# создаем экземпляр класса MinMaxScaler
minmaxscaler = MinMaxScaler()
# выполняем стандартизацию
cols = data.columns.tolist()
data[cols] = minmaxscaler.fit_transform(data[cols])
data.head(3)
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154
1	0.283	0.521	0.000	0.116	0.144	0.338	0.216	0.494	0.362	0.210	0.215
2	0.283	0.438	0.040	0.096	0.134	0.197	0.170	0.509	0.409	0.192	0.215

Теперь выполняем кластеризацию k -средних с помощью класса KMeans, получаем метки кластеров и записываем в датафрейм отдельную переменную с метками кластеров.

```
# импортируем класс KMeans
from sklearn.cluster import KMeans
# создаем экземпляр класса KMeans и обучаем
kmeans = KMeans(n_clusters=3, random_state=42).fit(data)
# получаем метки кластеров
pred = kmeans.predict(data)
# создаем отдельную переменную с метками кластеров
data['cluster_id'] = pred
data.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	cluster_id
0	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154	2
1	0.283	0.521	0.000	0.116	0.144	0.338	0.216	0.494	0.362	0.210	0.215	2
2	0.283	0.438	0.040	0.096	0.134	0.197	0.170	0.509	0.409	0.192	0.215	2
3	0.584	0.110	0.560	0.068	0.105	0.225	0.191	0.582	0.331	0.150	0.215	1
4	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154	2

Теперь выполняем стандартизацию так, чтобы в итоге у нас был возвращен массив NumPy. Воспользуемся ранее созданной копией массива признаков.

```
# выполняем стандартизацию
data_copy = minmaxscaler.fit_transform(data_copy)
data_copy
```

```
array([[0.24778761, 0.39726027, 0.          , ..., 0.60629921, 0.13772455, 0.15384615],
       [0.28318584, 0.52054795, 0.          , ..., 0.36220472, 0.20958084, 0.21538462],
       [0.28318584, 0.43835616, 0.04        , ..., 0.40944882, 0.19161677, 0.21538462],
       ...,
       [0.15044248, 0.26712329, 0.13        , ..., 0.53543307, 0.25149701, 0.4        ],
       [0.11504425, 0.35958904, 0.12        , ..., 0.65354331, 0.22754491, 0.27692308],
       [0.12389381, 0.13013699, 0.47        , ..., 0.51181102, 0.19760479, 0.4        ]])
```

Вновь выполняем кластеризацию k -средних с помощью класса KMeans, получаем метки кластеров и вставляем в начало массива NumPy отдельный столбец с метками кластеров.

```
# создаем экземпляр класса KMeans и обучаем
kmeans = KMeans(n_clusters=3, random_state=42).fit(data_copy)
# получаем метки кластеров
pred = kmeans.predict(data_copy)
# вставляем столбец с метками кластеров в начало массива NumPy
data_copy = np.insert(data_copy, 0, pred, axis=1)
data_copy

array([[1.          , 0.24778761, 0.39726027, ..., 0.60629921, 0.13772455, 0.15384615],
       [1.          , 0.28318584, 0.52054795, ..., 0.36220472, 0.20958084, 0.21538462],
       [1.          , 0.28318584, 0.43835616, ..., 0.40944882, 0.19161677, 0.21538462],
       ...,
       [1.          , 0.15044248, 0.26712329, ..., 0.53543307, 0.25149701, 0.4        ],
       [1.          , 0.11504425, 0.35958904, ..., 0.65354331, 0.22754491, 0.27692308],
       [0.          , 0.12389381, 0.13013699, ..., 0.51181102, 0.19760479, 0.4        ]])
```

16.2.13. Добавление расстояний от точки до каждого центроида

В качестве признаков можно использовать расстояния от точки до каждого центроида.

```
# удаляем столбец с метками кластеров
data_copy = np.delete(data_copy, 0, axis=1)

# пишем функцию, вычисляющую расстояния
# от точки до каждого центроида
def distances_to_centroids(data):
    # записываем матрицу с координатами центроидов
    centroids = kmeans.cluster_centers_
    # вычисляем расстояния от точки до каждого центроида
    deltas = data[:, np.newaxis, :] - centroids
    distances = np.sqrt(np.sum((deltas) ** 2, 2))
    # добавляем расстояния в массив
    data = np.append(data, distances, axis=1)
    return data

# вычисляем расстояния от точки до каждого центроида
# и добавляем в массив
new_data = distances_to_centroids(data_copy)
new_data
```

```
array([[0.24778761, 0.39726027, 0.          , ... , 0.49472169, 0.26781997, 0.66914964],
       [0.28318584, 0.52054795, 0.          , ... , 0.40832851, 0.35784666, 0.68135402],
       [0.28318584, 0.43835616, 0.04       , ... , 0.37575016, 0.22829003, 0.57944815],
       ... ,
       [0.15044248, 0.26712329, 0.13      , ... , 0.38308111, 0.27110112, 0.59867249],
       [0.11504425, 0.35958904, 0.12      , ... , 0.43403244, 0.32443219, 0.71193208],
       [0.12389381, 0.13013699, 0.47      , ... , 0.42523439, 0.45606836, 0.45025858]])
```

16.2.14. Добавление признаков на основе метода главных компонент

Кроме того, можно создать признаки на основе метода главных компонент.

Анализ главных компонент представляет собой метод, который осуществляет вращение данных, с тем чтобы преобразованные признаки не коррелировали между собой. Алгоритм начинает работу с того, что сначала находит направление максимальной дисперсии, помеченное как «компоненты 1». Речь идет о направлении (или векторе) данных, который содержит большую часть информации, или, другими словами, направление, вдоль которого признаки коррелируют друг с другом сильнее всего. Затем алгоритм находит направление, которое содержит наибольшее количество информации и при этом ортогонально (расположено под прямым углом) первому направлению. В двумерном пространстве существует только одна возможная ориентация, расположенная под прямым углом, но в пространствах большей размерности может быть (бесконечно) много ортогональных направлений. Направления, найденные с помощью этого алгоритма, называются *главными компонентами* (*principal components*), поскольку они являются основными направлениями дисперсии данных. Обычно создают 2–3 главные компоненты.

Поскольку алгоритм работает с дисперсией переменных, здесь также потребуется стандартизация переменных, в нашем случае она уже проведена.

```
# импортируем класс PCA
from sklearn.decomposition import PCA
# создаем экземпляр класса PCA
pca = PCA(n_components=2)
# выделяем компоненты
data_projected = pca.fit_transform(
    data.loc[:, data.columns != 'cluster_id'])
# вставляем компоненты как признаки
data.insert(0, 'PCATwo', data_projected[:, 1])
data.insert(0, 'PCAOne', data_projected[:, 0])
data.head()
```

	PCAOne	PCATwo	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	cluster_id
0	-0.272	-0.195	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154	2
1	-0.226	-0.259	0.283	0.521	0.000	0.116	0.144	0.338	0.216	0.494	0.362	0.210	0.215	2
2	-0.187	-0.180	0.283	0.438	0.040	0.096	0.134	0.197	0.170	0.509	0.409	0.192	0.215	2
3	0.422	-0.032	0.584	0.110	0.560	0.068	0.105	0.225	0.191	0.582	0.331	0.150	0.215	1
4	-0.272	-0.195	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154	2

16.2.15. Добавление признаков на основе расстояния от точки до ее k -го ближайшего соседа

С помощью метода ближайших соседей можно создать признак – расстояние от точки до ее k -го ближайшего соседа (например, расстояние до 3-го ближайшего соседа, расстояние до 4-го ближайшего соседа и т. д.).

Давайте удалим переменные `PCAOne`, `PCATwo` и `cluster_id` – результаты предыдущих экспериментов – и вычислим расстояния.

```
# удаляем переменные PCAOne, PCATwo и cluster_id
data.drop(['PCAOne', 'PCATwo', 'cluster_id'],
          axis=1, inplace=True)
```

Затем импортируем класс `NearestNeighbors` и пишем функцию, вычисляющую расстояние от точки до ее k -го соседа.

```
# импортируем класс NearestNeighbors
from sklearn.neighbors import NearestNeighbors

# пишем функцию, которая вычисляет расстояние от точки до k-го соседа
def k_distances(X, n_neigh):
    neigh = NearestNeighbors(n_neighbors=n_neigh)
    nbrs = neigh.fit(X)
    distances, _ = nbrs.kneighbors(X)
    distances = distances[:, distances.shape[1] - 1]
    is_np = isinstance(X, np.ndarray)
    if is_np:
        X = np.insert(X, X.shape[1], distances, axis=1)
    else:
        X['distances'] = distances
    return X
```

На основе датафрейма pandas создадим массив NumPy.

```
# создадим массив NumPy на основе
# датафрейма pandas
data_array = data.values
```

Давайте вычислим расстояния для датафрейма и массива NumPy.

```
# вычисляем расстояния для датафрейма pandas
data = k_distances(data, 4)
data.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	distances
0	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154	0.075
1	0.283	0.521	0.000	0.116	0.144	0.338	0.216	0.494	0.362	0.210	0.215	0.225
2	0.283	0.438	0.040	0.096	0.134	0.197	0.170	0.509	0.409	0.192	0.215	0.156
3	0.584	0.110	0.560	0.068	0.105	0.225	0.191	0.582	0.331	0.150	0.215	0.162
4	0.248	0.397	0.000	0.068	0.107	0.141	0.099	0.568	0.606	0.138	0.154	0.075

```
# вычисляем расстояния для массива NumPy
data_array = k_distances(data_array, 4)
data_array
```

```
array([[0.24778761, 0.39726027, 0.          , ..., 0.13772455, 0.15384615, 0.07458349],
       [0.28318584, 0.52054795, 0.          , ..., 0.20958084, 0.21538462, 0.22524686],
       [0.28318584, 0.43835616, 0.04       , ..., 0.19161677, 0.21538462, 0.15576311],
       ...,
       [0.15044248, 0.26712329, 0.13       , ..., 0.25149701, 0.4          , 0.15897532],
       [0.11504425, 0.35958904, 0.12       , ..., 0.22754491, 0.27692308, 0.18368717],
       [0.12389381, 0.13013699, 0.47       , ..., 0.19760479, 0.4          , 0.20871273]])
```

16.2.16. Добавление признаков на основе комбинации разности и отношения для методов на основе деревьев

Для методов на основе деревьев эффективными могут быть комбинации разности и отношения признаков. Обычно на обучающей выборке обучают модель CatBoost с достаточным количеством деревьев (не меньше 300), получают топ-30 признаков с точки зрения важностей по SHAP. Формируют два списка с четными и нечетными индексами признаков из полученного списка топ-30 признаков. Затем из каждого признака, находящегося в первом списке, вычитаем признак, находящийся во втором списке, полученный результат делим на признак из первого списка или признак из второго списка (можно случайно выбирать).

Например, такие признаки позволяли улучшить качество модели XGBoost в соревновании Santander Customer Satisfaction на Kaggle <https://www.kaggle.com/c/santander-customer-satisfaction/overview>. Ниже приводится упрощенный вариант решения для этого соревнования с использованием комбинаций разности и отношения признаков.

```
# импортируем необходимые библиотеки
import xgboost as xgb
from catboost import CatBoostClassifier, Pool

# загружаем наборы
train = pd.read_csv('Data/santander_train.csv')
test = pd.read_csv('Data/santander_test.csv')

# формируем массив меток и массив признаков
labels = train.pop('TARGET')
# сохраняем ID теста
test_id = test['ID']

# удаляем ID
train.drop('ID', axis=1, inplace=True)
test.drop('ID', axis=1, inplace=True)

# удаляем константные признаки
constant_features = [
    feat for feat in train.columns if train[feat].nunique() == 1
]
train.drop(constant_features, axis=1, inplace=True) test.drop(constant_features, axis=1,
inplace=True)
```

```

# удаляем дублирующиеся признаки
duplicated_features = ['ind_var6_0', 'ind_var6', 'num_var6_0',
                       'num_var6', 'saldo_var6',
                       'delta_imp_reemb_var13_1y3',
                       'delta_imp_reemb_var17_1y3',
                       'delta_imp_reemb_var33_1y3',
                       'delta_imp_trasp_var17_in_1y3',
                       'delta_imp_trasp_var17_out_1y3',
                       'delta_imp_trasp_var33_in_1y3',
                       'delta_imp_trasp_var33_out_1y3',
                       'saldo_medio_var13_medio_ult1']

]

train.drop(duplicated_features, axis=1, inplace=True) test.drop(duplicated_features, axis=1,
inplace=True)

# обучаем модель CatBoost
train_pool = Pool(train, labels)
clf = CatBoostClassifier(n_estimators=1200,
                        learning_rate=0.08,
                        random_strength=0.15,
                        max_depth=2,
                        random_seed=0,
                        logging_level='Silent')
clf.fit(train_pool)

# вычисляем важности по SHAP
shap_values = clf.get_feature_importance(
    train_pool, type='ShapValues') shap_values = shap_values[:, :-1]

# формируем список топ-30 важных признаков, отобранных по SHAP
# (использовался CatBoost из 1200 деревьев)
shap_feat = list(train.columns[np.argsort(
    np.abs(shap_values).mean(0))[:-1]])
top_shap_feat = shap_feat[:30]

# отбираем четные и нечетные элементы списка
top_left = [y for x, y in enumerate(top_shap_feat) if x%2 == 0]
top_right = [y for x, y in enumerate(top_shap_feat) if x%2 != 0]

# создаем новые признаки – комбинации разности и деления
for i in range(len(top_left)):
    for j in range(len(top_right)):
        colName = top_left[i] + "_SUB_" + top_right[j] + "DIV" + top_left[i]
        train[colName] = np.where(
            ((train[top_left[i]] - train[top_right[j]]) == 0) |
            (train[top_left[i]] == 0),
            0,
            (train[top_left[i]] - train[top_right[j]]) / train[top_left[i]])
        test[colName] = np.where(
            ((test[top_left[i]] - test[top_right[j]]) == 0) |
            (test[top_left[i]] == 0),
            0,
            (test[top_left[i]] - test[top_right[j]]) / test[top_left[i]])

```

```
# создаем экземпляр XGBClассifier (гиперпараметры были
# подобраны в ходе перекрестной проверки, запущенной
# на обучающей выборке)
xgb_model_sklearn = xgb.XGBClassifier(
    eta=0.04,
    n_estimators=150,
    max_depth=4,
    subsample=0.9,
    colsample_bytree=0.6,
    objective='binary:logistic',
    random_state=42)
# строим модель
xgb_model_sklearn.fit(train, labels)
# получаем вероятности
preds_prob = xgb_model_sklearn.predict_proba(test)[:, 1]
# оформляем посылку
pd.DataFrame({'ID': test_id, 'TARGET': preds_prob}).to_csv(
    'submission_santander.csv', index=False)
```

16.3. ДИНАМИЧЕСКОЕ КОНСТРУИРОВАНИЕ ПРИЗНАКОВ ИСХОДЯ ИЗ ОСОБЕННОСТЕЙ АЛГОРИТМА

Теперь выясним, что представляет из себя динамическое конструирование признаков (dynamic feature engineering). Напомним: динамическое конструирование признаков подразумевает, что мы создаем признаки в ходе построения модели «на лету», эти признаки создаются только в процессе обучения модели и в наш набор данных не записываются.

16.3.1. Преобразование категориальных признаков в количественные внутри библиотеки CatBoost

Ранее мы говорили, что для ансамблей на основе деревьев решений часто бывает полезно преобразовать категориальный признак в количественный, представив каждую категорию в виде числа. Однако это преобразование мы выполняли вручную. В новой библиотеке градиентного бустинга CatBoost категориальные признаки превращаются в количественные автоматически. Рассмотрим, как происходит преобразование категориальных признаков в количественные при решении задачи классификации. Оно происходит в три этапа.

- 1) CatBoost принимает на вход набор значений признаков и значений зависимой переменной (т. е. моделируемых значений). Рисунок иллюстрирует, как выглядят результаты на данном этапе.

	Признак ₁	Признак ₂	Признак ₃	Зависимая переменная
1	2	40	Рок	1
2	3	55	Инди	0
3	5	34	Поп	1
4	2	45	Рок	0
5	4	53	Рок	0
6	2	48	Инди	1
7	5	42	Рок	1
...

Рис. 57 Исходный набор значений признаков и зависимой переменной

- 2) Наблюдения несколько раз случайным образом перемешиваются, и создается несколько случайных перестановок набора данных. Рисунок ниже показывает, как выглядят результаты на данном этапе.

	Признак ₁	Признак ₂	Признак ₃	Зависимая переменная
1	4	53	Рок	0
2	3	55	Инди	0
3	2	40	Рок	1
4	5	42	Рок	1
5	5	34	Поп	1
6	2	48	Инди	1
7	2	45	Рок	0
...

Рис. 58 Набор значений признаков и зависимой переменной после случайных перестановок

- 3) Все категориальные признаки преобразовываются в количественные по формуле:

$$\text{avg_target} = \frac{\text{countInClass} + \text{prior}}{\text{totalCount} + 1}.$$

countInClass – сколько раз значение зависимой переменной равно 1 для наблюдений с данной категорией признака. Учитываются только наблюдения с данной категорией признака до интересующего нас наблюдения (расчеты выполняются в порядке следования наблюдений после перемешивания), то есть интересующее нас наблюдение в расчетах не участвует.

prior – исходное значение числителя, определенное начальными параметрами (например, 0,05).

totalCount – общее количество наблюдений до интересующего нас наблюдения, для которых категория признака совпадает с категорией признака для текущего наблюдения. Еще раз заметьте: интересующее нас наблюдение в расчетах не участвует.

Допустим, мы хотим вычислить значение для Признак₃ в наблюдении 4. Мы видим, что у нас только одно наблюдение с категорией «Рок», в котором зависимая переменная принимает значение 1 (не считая рассматриваемое наблюдение). Поэтому *countInClass* равен 1. При этом у нас два наблюдения с категорией «Рок» до рассматриваемого. Поэтому *totalCount* равен 2.

На рисунке подробно приводятся вычисления. Обратите внимание, что эти значения рассчитываются для каждого наблюдения по отдельности с использованием данных предыдущих наблюдений.

Используем для вычисления статистики по наблюдению 4 только «историю» – предыдущие наблюдения		Признак ₁	Признак ₂	Признак ₃	Зависимая переменная	Вычисление значений
	1	4	53	Рок	0	$(0 + \text{prior}) / (0 + 1) = 0.05 / 1 = 0.05$
	2	3	55	Инди	0	$(0 + \text{prior}) / (0 + 1) = 0.05 / 1 = 0.05$
	3	2	40	Рок	1	$(0 + \text{prior}) / (1 + 1) = 0.05 / 2 = 0.025$
	4	5	42	Рок	1	$(1 + \text{prior}) / (2 + 1) = 1.05 / 3 = 0.35$
	5	5	34	Поп	1	$(0 + \text{prior}) / (0 + 1) = 0.05 / 1 = 0.05$
	6	2	48	Инди	1	$(0 + \text{prior}) / (1 + 1) = 0.05 / 2 = 0.025$
	7	2	45	Рок	0	$(2 + \text{prior}) / (3 + 1) = 2.05 / 4 = 0.5125$

Рис. 59 Вычисление значений будущей количественной переменной для категориального признака

Рисунок показывает, как выглядят итоговые результаты.

	Признак ₁	Признак ₂	Признак ₃	Зависимая переменная
1	4	53	0,05	0
2	3	55	0,05	0
3	2	40	0,025	1
4	5	42	0,35	1
5	5	34	0,05	1
6	2	48	0,025	1
7	2	45	0,5125	0
...

Рис. 60 Замена категорий признака количественными значениями

16.3.2. Биннинг категориальных признаков внутри библиотеки H2O

В библиотеке машинного обучения H2O вместо превращения категориального признака с большим числом категорий в количественный мы можем укрупнить категории прямо в ходе построения модели случайного леса или градиентного бустинга. Для этого используется гиперпараметр `nbins_cats`. Для гиперпараметра `nbins_cats` по умолчанию используется значение 1024, т. е. для категориального признака создается не менее 1024 бинов. Если количество категорий меньше значения гиперпараметра `nbins_cats`, каждая категория получает свой бин.

Допустим, у нас есть переменная `Class`. Если у нее есть уровни A, B, C, D, E, F, G и мы зададим `nbins_cats=8`, то будут сформировано 7 бинов: {A}, {B}, {C}, {D}, {E}, {F} и {G}. Каждая категория получает свой бин. Будет рассмотрено $2^6 - 1 = 63$ точки расщепления. Если мы зададим `nbins_cats=10`, то все равно будут получены те же самые бины, потому что у нас всего 7 категорий. Если количество категорий больше значения гиперпараметра `nbins_cats`, категории будут сгруппированы в бины в лексикографическом порядке. Например, если мы зададим `nbins_cats=2`, то будет сформировано 2 бина: {A, B, C, D} и {E, F, G}. У нас будет одна точка расщепления.

Для признаков с большим количеством категорий небольшое значение гиперпараметра `nbins_cats` может внести в процесс создания точек расщепле-

ния дополнительную случайность (поскольку категории группируются в определенном смысле произвольным образом), в то время как большие значения гиперпараметра `nbins_cats` (например, значение гиперпараметра `nbins_cats`, совпадающее с количеством категорий), наоборот, снижают эту случайность, каждая отдельная категория может быть рассмотрена при формировании точки разбиения, что приводит к переобучению на обучающем наборе.

Очевидно, что гиперпараметр `nbins_cats` может быть полезен при наличии в наборе данных одного или нескольких признаков с очень большим количеством категорий, поскольку в такой ситуации, например, методы на основе деревьев при поиске оптимальных расщепляющих значений будут склоняться в пользу именно этих признаков, даже если они не обладают высокой прогнозной силой. Снизив количество категорий для таких переменных за счет выбора меньшего значения гиперпараметра `nbins_cats`, можно скорректировать тенденцию выбирать преимущественно признаки с большим числом категорий и тем самым улучшить качество модели. В то же время следует помнить, если вы будете таким образом укрупнять категории действительно важного признака, то можете ухудшить качество модели. Один из советов: если признак с большим количеством категорий является важным с точки зрения важности на основе усредненного уменьшения неоднородности, но при этом не является таковым с точки зрения пермутированной важности, можно попробовать укрупнить его категории с помощью биннинга или превратить в количественный признак.

16.3.3. Связывание взаимно исключающих признаков внутри библиотеки LightGBM

Высокоразмерные данные, с которыми мы часто работаем, обычно являются очень разреженными. Разреженность пространства признаков позволяет разработать подход к сокращению числа признаков, практически не дающий ухудшения точности модели. В частности, в разреженном пространстве признаков многие признаки являются взаимоисключающими, т. е. они никогда не принимают ненулевые значения одновременно. Кроме того, можно ввести уровень конфликтности (conflict rate) – количество наблюдений, в которых у признаков могут быть одновременно ненулевые значения.

Взаимоисключающие признаки А и В, они не принимают ненулевые значения одновременно: нулевому значению одного признака всегда соответствует ненулевое значение другого

Признак А	Признак В
0	1
2	0
1	0
0	2
3	0
0	2
0	1

Взаимоисключающие признаки А и В, уровень конфликтности = 2
(допускаются 2 наблюдения, в которых у А и В могут быть одновременно ненулевые значения, выделены красным)

Признак А	Признак В
0	1
2	0
1	0
0	2
3	0
1	2
3	1

Рис. 61 Взаимоисключающие признаки и уровень конфликтности

Мы можем смело связывать взаимоисключающие признаки в один, который мы называем *связкой взаимоисключающих признаков* – *exclusive feature bundle*. С помощью тщательно проработанного алгоритма исследования признаков возможно выполнить гистограммирование признаков на основе связок признаков, как это обычно выполнялось для отдельных признаков. Таким образом, сложность построения гистограммы изменяется с $O(\text{количество наблюдений} \times \text{количество признаков})$ до $O(\text{количество наблюдений} \times \text{количество связок})$, где количество связок будет меньше количества признаков. Таким образом, можно значительно ускорить обучение градиентного бустинга над деревьями решений без ущерба для точности.

Здесь нужно решить две задачи. Первая задача заключается в определении того, какие признаки необходимо связывать. Вторая задача состоит в том, каким образом сконструировать связку.

Первую задачу авторы LightGBM свели к задаче раскраски графа, взяв признаки в качестве вершин и добавляя ребра для каждого двух признаков, если они не являются взаимоисключающими, а затем использовали жадный алгоритм, который мог дать достаточно хорошее решение задачи раскраски графа для получения связок признаков. Далее авторы LightGBM обратили внимание на наличие довольно большого количества признаков, которые хоть и не являются на 100 % взаимоисключающими, но также редко принимают ненулевые значения одновременно. Они предложили регулировать эту ситуацию с помощью настройки уровня конфликтности. Если наш алгоритм допускает небольшой уровень конфликтности (т. е. небольшое количество строк, где могут быть одновременно ненулевые значения), мы можем получить еще меньшее количество связок признаков и еще больше повысить вычислительную эффективность.

Итак, сначала строим граф со взвешенными ребрами, где веса соответствуют общему уровню конфликтности между признаками. Затем отсортируем признаки по их степеням в порядке убывания. Наконец, исследуем каждый признак в упорядоченном списке и либо присоединим его к уже существующей связке с небольшой степенью конфликтности (регулируется значением отдельного гиперпараметра), либо создадим новую связку. Сложность алгоритма составляет $O(\text{количество признаков}^2)$, и выполнятся он будет только один раз перед обучением.

Алгоритм формирования связок признаков

Ввод: F (признаки), K (порог, задающий максимально допустимый уровень конфликтности)

создаем граф G

сортируем признаки по степени и получаем упорядоченный список $\text{searchOrder} \leftarrow G.\text{sortByDegree}()$

задаем список, где будут храниться связки, и список со значениями конфликтности в связках $\text{bundles} \leftarrow \{\}, \text{bundlesConflict} \leftarrow \{\}$

проходим по упорядоченному списку признаков, и если уровень конфликтности меньше или равен порогу, то добавляем признак к существующей связке, а если уровень конфликтности больше порога K , то добавляем признак в качестве новой связки

для i в searchOrder **выполнять**

- $\text{needNew} \leftarrow \text{True}$
- для** j от 1 до $\text{len}(\text{bundles})$ **выполнять**
 - $\text{cnt} \leftarrow \text{ConflictCnt}(\text{bundles}[j], F[i])$
 - если** $\text{cnt} + \text{bundlesConflict}[i] \leq K$ **то**
 - $\text{bundles}[j].\text{add}(F[i]), \text{needNew} \leftarrow \text{False}$
 - break**
 - если** needNew **то**
 - добавляем** $F[i]$ в качестве новой связки в список связок bundles

Вывод: список связок bundles

Эта сложность приемлема в ситуации, когда количество признаков не очень велико, но может быть по-прежнему слишком высоко для работы с миллионами признаков. Для дальнейшего повышения скорости авторы предложили еще более эффективную стратегию упорядочивания без построения графа: упорядочивание по количеству ненулевых значений, что схоже с упорядочиванием по степеням, поскольку большее количество ненулевых значений обычно приводит к большей вероятности конфликтов.

Для решения второй задачи необходим хороший способ объединения признаков в одной связке, чтобы уменьшить соответствующую сложность обучения. Ключевая мысль заключается в том, чтобы гарантировать возможность идентификации значений исходных признаков по значениям связок признаков. Поскольку алгоритм на основе гистограммирования вместо непрерывных значений признака хранит дискретные бины, возможно сконструировать связку признаков, позволив значениям взаимоисключающих признаков располагаться в разных бинах. Это можно сделать, добавив постоянное смещение к исходным значениям признаков.

Предположим, имеются два признака, объединенных в связку. Первоначально признак А принимает значение $[0, 10)$, а признак В принимает значение $[0, 20)$. После добавления константы 10 к значениям признака В обновленный признак принимает значения $[10, 30)$. После этого можно безопасно объединить признаки А и В и использовать связку признаков с диапазоном $[0, 30]$ для замены оригинальных признаков А и В.

Алгоритм EFB способен связывать большое количество взаимно исключающих признаков в гораздо более плотные признаки, что позволяет эффективно снизить количество ненужных вычислений для нулевых значений признаков.

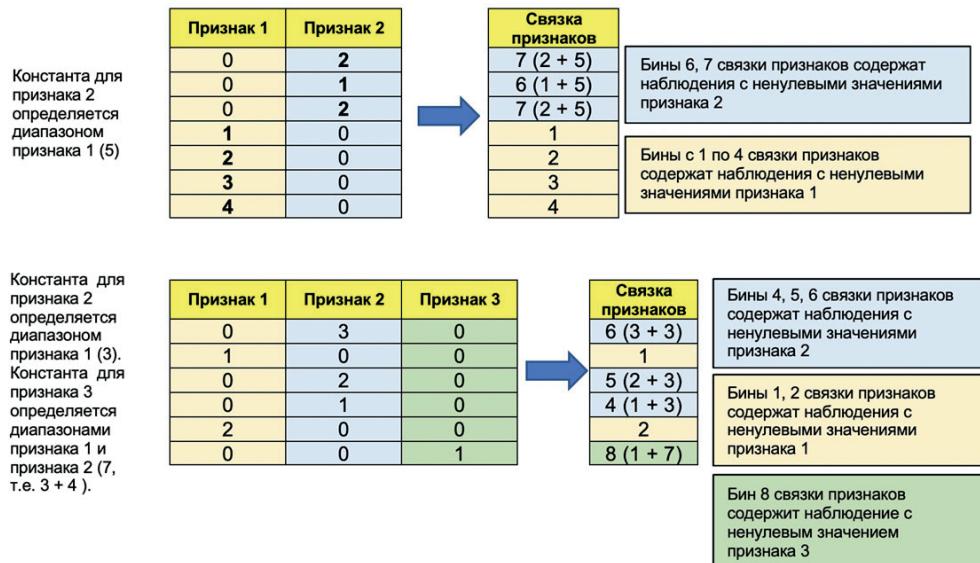


Рис. 62 Визуализация объединения взаимно исключающих признаков в связке

Алгоритм объединения взаимно исключающих признаков в связке**Ввод:** $numData$ (количество наблюдений)**Ввод:** F (одна связка взаимно исключающих признаков)инициализируем нулями список с диапазонами $binRanges$ и счетчик количества бинов $totalBin$ (будет представлять собой сумму бинов по некоторым признакам связки) $binRanges \leftarrow \{0\}$, $totalBin \leftarrow 0$ записываем в $binRanges$ диапазоны признаков, которые будут выполнять роль констант**для** f в F **выполнять**увеличиваем счетчик количества бинов на количество бинов признака
 $totalBin += f.numBin$

обновляем диапазон в соответствии со счетчиком количества бинов

 $binRanges.append(totalBin)$ $newBin \leftarrow newBin(numData)$

вычисляем бины связки

для i от 1 до $numData$ **выполнять**инициализируем бин связки нулем
 $newBin[i] \leftarrow 0$ вычисляем бин связки для каждого ненулевого значения исходного признака, добавив к бину исходного признака константу $binRanges$, так чтобы диапазон данного признака не пересекался с диапазонами других признаков**для** j от 1 до $len(F)$ **выполнять**если $F[j].bin[i] \neq 0$ то
 $newBin[i] \leftarrow F[j].bin[i] + binRanges[j]$ **Вывод:** $newBin, binRanges$

16.4. Конструирование признаков для временных рядов

Прежде чем приступить к конструированию признаков, выясним, как вообще лучше всего работать с датафреймом, содержащим даты. Обратите внимание, мы будем приводить примеры не только для pandas, но и для Polars – популярной библиотеки для быстрой предварительной подготовки данных, которая часто применяется для работы с временными рядами. Давайте импортируем необходимые библиотеки, модули, функции и загрузим данные, хранящиеся в файле *example.csv*. Разделителем в этом файле является табуляция. У нас две переменные – переменная даты и продажи. Продажи являются ежедневными.

```
# импортируем библиотеки numpy, pandas,
# polars, bottleneck
import numpy as np
import pandas as pd
import polars as pl
import bottleneck as bn

# импортируем необходимые классы и функции
from catboost import CatBoostRegressor, Pool
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error

# импортируем модуль warnings
import warnings

# импортируем модуль pyplot библиотеки matplotlib
import matplotlib.pyplot as plt

# настраиваем визуализацию
%config InlineBackend.figure_format = 'retina'

# загружаем данные
data = pd.read_csv('Data/example.csv', sep='\t')
data.head()
```

	date	sales
0	09.01.2018	2400
1	10.01.2018	2800
2	11.01.2018	2500
3	12.01.2018	2890
4	13.01.2018	2610
5	14.01.2018	2500
6	15.01.2018	2750
7	16.01.2018	2700
8	17.01.2018	2250
9	18.01.2018	2350
10	19.01.2018	2550
11	20.01.2018	3000

Теперь с помощью функции `pd.to_datetime()` преобразовываем столбец с датой в тип `datetime`, указав с помощью параметра `format` нужный формат дат.

```
# преобразовываем столбец с датой в тип datetime
data['date'] = pd.to_datetime(data['date'],
                             format='%d.%m.%Y')
data
```

	date	sales
0	2018-01-09	2400
1	2018-01-10	2800
2	2018-01-11	2500
3	2018-01-12	2890
4	2018-01-13	2610
5	2018-01-14	2500
6	2018-01-15	2750
7	2018-01-16	2700
8	2018-01-17	2250
9	2018-01-18	2350
10	2018-01-19	2550
11	2018-01-20	3000

С помощью параметров `parse_dates` и `date_parser` функции `pd.to_datetime()` можно указать столбец с датами и формат дат, чтобы выполнить парсинг дат сразу при создании датафрейма.

```
# загружаем данные, сразу выполнив парсинг дат
data = pd.read_csv('Data/example.csv', sep='\t',
                    parse_dates=['date'],
                    date_parser=lambda col: pd.to_datetime(
                        col, format='%d.%m.%Y'))
```

	date	sales
0	2018-01-09	2400
1	2018-01-10	2800
2	2018-01-11	2500
3	2018-01-12	2890
4	2018-01-13	2610
5	2018-01-14	2500
6	2018-01-15	2750
7	2018-01-16	2700
8	2018-01-17	2250
9	2018-01-18	2350
10	2018-01-19	2550
11	2018-01-20	3000

Кроме того, с помощью параметра `index_col` можно указать столбец с датами в качестве индекса. Как правило, хранение дат в виде индекса может быть более удобной практикой, потому что практически всегда столбец с датами перед построением модели нужно удалять, в противном случае это может привести к появлению ошибок, поскольку модели машинного обучения, как правило, не умеют напрямую работать с датами. Как вариант, если хочется работать непосредственно с датами, можно перевести их в UNIX-время (также называют POSIX-время) – количество секунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года (четверг).

```
# переводим столбец с датами в UNIX-время
start = pd.Timestamp('1970-01-01')
d = pd.Timedelta('1s')
data['unix'] = (data['date'] - start) // d
data
```

	date	sales	unix
0	2018-01-09	2400	1515456000
1	2018-01-10	2800	1515542400
2	2018-01-11	2500	1515628800
3	2018-01-12	2890	1515715200
4	2018-01-13	2610	1515801600
5	2018-01-14	2500	1515888000
6	2018-01-15	2750	1515974400
7	2018-01-16	2700	1516060800
8	2018-01-17	2250	1516147200
9	2018-01-18	2350	1516233600
10	2018-01-19	2550	1516320000
11	2018-01-20	3000	1516406400

Итак, давайте сразу при создании датафрейма прочитаем столбец с датами как индекс и выполним парсинг дат.

```
# загружаем данные, сразу прочитав столбец
# с датами как индекс и выполнив парсинг дат
data = pd.read_csv('Data/example.csv', sep='\t',
                    index_col=['date'],
                    parse_dates=['date'],
                    date_parser=lambda col: pd.to_datetime(
                        col, format='%d.%m.%Y'))
```

sales

date

2018-01-09	2400
2018-01-10	2800
2018-01-11	2500
2018-01-12	2890
2018-01-13	2610
2018-01-14	2500
2018-01-15	2750
2018-01-16	2700
2018-01-17	2250
2018-01-18	2350
2018-01-19	2550
2018-01-20	3000

При желании можно удалить имя индекса.

```
# удаляем имя индекса
data.index.name = None
data
```

sales

2018-01-09	2400
2018-01-10	2800
2018-01-11	2500
2018-01-12	2890
2018-01-13	2610
2018-01-14	2500
2018-01-15	2750
2018-01-16	2700
2018-01-17	2250
2018-01-18	2350
2018-01-19	2550
2018-01-20	3000

Теперь приступим к конструированию признаков. Начнем с лаговых переменных.

16.4.1. Лаговые переменные

Лаговая переменная – это переменная, значение которой мы берем не за текущий, а за отстоящий от него на определенное расстояние предыдущий момент времени. Проще говоря, это переменные, взятые с запаздыванием во времени. Величина интервала запаздывания как раз и называется *лагом*. Такие переменные мы часто будем записывать так: lag 10 или lag = 10, что будет эквивалентно записи $t - 10$.

Лаги можно создать с помощью метода `.shift()`, указав с помощью параметра `periods` порядок лага или количество периодов – моментов, на которое значение должно отстоять от текущего.

Параметр `periods` метода `.shift()` часто опускают.

```
data['Lag3'] = data['sales'].shift(periods=3)
```

или

```
data['Lag3'] = data['sales'].shift(3)
```

Допустим, у нас есть данные продаж за 12 дней и мы с помощью метода `.shift()` создали лаги с запаздыванием на 3, 5, 8, 12 и 13 дней.

создаем лаги

```
# лаг с запаздыванием на 3 дня
data['Lag3'] = data['sales'].shift(3)
# лаг с запаздыванием на 4 дня
data['Lag4'] = data['sales'].shift(4)
# лаг с запаздыванием на 5 дней
data['Lag5'] = data['sales'].shift(5)
# лаг с запаздыванием на 6 дней
data['Lag6'] = data['sales'].shift(6)
# лаг равен длине набора
data['Lag12'] = data['sales'].shift(12)
# лаг превышает длину набора
data['Lag13'] = data['sales'].shift(13)
# смотрим результаты
data
```

	sales	<i>t</i> Lag3	<i>t</i> Lag4	<i>t</i> Lag5	<i>t</i> Lag6	<i>t</i> Lag12	<i>t</i> Lag13
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	3 NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	4 NaN	5 NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN	NaN	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0	NaN	NaN
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0	NaN	NaN
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0	NaN	NaN
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0	NaN	NaN
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0	NaN	NaN
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0	NaN	NaN

Порядок лага, настраиваемый с помощью параметра `periods`, не должен быть равен или превышать длину набора данных, на основе которого он создается, в противном случае он будет состоять из одних пропусков, что мы и видим для Lag12 и Lag13. Чем больше порядок лагов, тем меньше наблюдений используется при его вычислении.

Отметим, что лаги будут полезными признаками при коротких горизонтах прогнозирования (до 30 дней включительно). Чем короче горизонт прогнозирования, тем эффективнее могут быть лаги в качестве признаков.

Лаги у нас были созданы до разбиения на обучающую и тестовую выборки. Давайте удалим Lag12 и Lag13 и разобьем наш набор данных так, чтобы в тестовую выборку попали 4 последних наблюдения. Допустим, нам надо будет прогнозировать продажи на 4 дня вперед, таким образом, горизонт прогнозирования – 4 дня.

```
# удалим переменные Lag12 и Lag13
data.drop(['Lag12', 'Lag13'], axis=1, inplace=True)
# задаем горизонт
HORIZON = 4
# разбиваем набор на обучающую и тестовую выборки
train, test = data[0:len(data)-HORIZON], data[len(data)-HORIZON:]
```

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0

смотрим обучающую выборку
train

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0

смотрим тестовую выборку
test

Обратите внимание: чем выше порядок лага, тем более ранние наблюдения обучающей выборки попадают в тестовую выборку.

Однако не все лаги в teste используют наблюдения обучающего набора, лаг 3 «залез» в тест, что является подсматриванием в будущее (обведено красным овалом), которое нам неизвестно.

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0

Рис. 63 При вычислении лага 3 произошла «протечка»

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0

Рис. 63 Окончание

Одна из самых частых ошибок, связанных с подготовкой лаговой переменной, связана с тем, что лаговая переменная для тестового набора берется из самого тестового набора. Таким образом, необходимо создавать лаговые переменные так, чтобы они не проникали в тестовый набор. Проиллюстрируем на нашем примере.

обучающая выборка

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0

тестовая выборка

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0

Lag 3
«проникает» в
тест, мы берем
информацию из
тестовой
выборки

Lag 4 и выше
не «проникают» в
тест, мы берем
информацию из
обучающей выборки

Рис. 64 Лаги, у которых порядок меньше горизонта прогнозирования, «залазят» в тест

Здесь легко увидеть, что Lag 3 «проникает» в тест, мы берем информацию из тестового набора. Поэтому лаги вида L_{t-k} лучше создавать так, чтобы k был равен или превышал горизонт прогнозирования. Впрочем, в рамках соревновательного (непромышленного) подхода допускается создание лагов, у которых порядок будет меньше горизонта прогнозирования, однако тогда значения зависят от переменной в тестовой выборке нужно заменить на значения NaN. Если лаг залезет в тест, в тесте появятся значения NaN. В таком случае чем больше горизонт прогнозирования будет превышать порядок лага, тем больше пропусков будет в тесте. Очевидный недостаток лагов меньше горизонта прогнозирования заключается в том, что появляется дополнительная задача импутации пропусков в тесте.

На практике для избежания протечек при вычислении лагов (а также скользящих и расширяющихся статистик) поступают двумя способами:

- значения зависимой переменной в наблюдениях исходного набора, которые будут соответствовать будущей тестовой выборке (набору новых данных), заменяют значениями NaN;
- берем обучающую выборку и удлиняем ее на длину горизонта прогнозирования, зависимая переменная в наблюдениях, соответствующих новым временным меткам (т. е. в тестовой выборке / наборе новых данных), получает значения NaN.

В обоих случаях мы формируем защиту от протечек при вычислении лагов в тестовой выборке / наборе новых данных.

Давайте продемонстрируем первый способ.

Мы снова загрузим данные, сразу при создании датафрейма прочитав столбец с датами как индекс и выполнив парсинг дат. Поскольку эти данные мы будем загружать еще не раз, напишем функцию загрузки `load_data()` и с ее помощью загрузим данные.

```
# пишем функцию загрузки
def load_data():
    # загружаем данные, сразу прочитав столбец
    # с датами как индекс и выполнив парсинг дат
    data = pd.read_csv('Data/example.csv', sep='\t',
                       index_col=['date'],
                       parse_dates=['date'],
                       date_parser=lambda col: pd.to_datetime(
                           col, format='%d.%m.%Y'))
    # удаляем имя индекса
    data.index.name = None
    return data
```

Значения в наблюдениях, которые будут приходиться на тестовую выборку (последние 4 наблюдения исходного набора), заменяем на значения NaN. Затем формируем лаги порядка 1, 2, 3, 4, 5.

```
# значения в наблюдениях, которые будут приходиться на тест
# (последние 4 наблюдения), заменяем на значения NaN
HORIZON = 4
data['sales'].iloc[-HORIZON:] = np.NaN
```

```
# создаем лаги, лаги меньше горизонта прогнозирования
# получат пропуски в наблюдениях, приходящихся на тест,
# при этом чем меньше порядок лага горизонта
# прогнозирования, тем больше пропусков в тесте
```

```
# лаг с запаздыванием на 1 день
data['Lag1'] = data['sales'].shift(1)
# лаг с запаздыванием на 2 дня
data['Lag2'] = data['sales'].shift(2)
# лаг с запаздыванием на 3 дня
data['Lag3'] = data['sales'].shift(3)
# лаг с запаздыванием на 4 дня
data['Lag4'] = data['sales'].shift(4)
# лаг с запаздыванием на 5 дней
data['Lag5'] = data['sales'].shift(5)
data
```

	sales	Lag1	Lag2	Lag3	Lag4	Lag5
2018-01-09	2400.0	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800.0	2400.0	NaN	NaN	NaN	NaN
2018-01-11	2500.0	2800.0	2400.0	NaN	NaN	NaN
2018-01-12	2890.0	2500.0	2800.0	2400.0	NaN	NaN
2018-01-13	2610.0	2890.0	2500.0	2800.0	2400.0	NaN
2018-01-14	2500.0	2610.0	2890.0	2500.0	2800.0	2400.0
2018-01-15	2750.0	2500.0	2610.0	2890.0	2500.0	2800.0
2018-01-16	2700.0	2750.0	2500.0	2610.0	2890.0	2500.0
2018-01-17	NaN	2700.0	2750.0	2500.0	2610.0	2890.0
2018-01-18	NaN	NaN	2700.0	2750.0	2500.0	2610.0
2018-01-19	NaN	NaN	NaN	2700.0	2750.0	2500.0
2018-01-20	NaN	NaN	NaN	NaN	2700.0	2750.0

Наша
тестовая выборка

Видим, что лаги, у которых порядок меньше горизонта прогнозирования, «залезают» в тестовую выборку – последние 4 наблюдения исходного набора и получают пропуски в наблюдениях, приходящихся на тестовую выборку (выделены красными рамками), при этом чем больше горизонт прогнозирования превышает порядок лага, тем больше пропусков будет в тестовой выборке. Так срабатывает защита от протечек.

Теперь проиллюстрируем второй способ. Удалим ранее созданные лаги и сформируем обучающую выборку – первые 8 наблюдений исходного набора.

```
# удалим лаги по параметру 'Lag'
data.drop(data.filter(regex='Lag').columns, axis=1,
           inplace=True)
# создаем обучающую выборку – первые 8
# наблюдений исходного набора
train = data.iloc[:HORIZON]
train
```

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

Пишем функцию `calculate_lags()`, которая берет нашу обучающую выборку и удлиняет ее на длину горизонта прогнозирования, зависимая переменная в наблюдениях, соответствующих новым временным меткам (т. е. в тестовой выборке / наборе новых данных), получает значения NaN.

```
# пишем функцию для создания лагов
# в обучающей и тестовой выборках
def calculate_lags(train, target, horizon, lags_range,
                   freq='D', aggregate=False):
    """
    Создает лаги в обучающей и тестовой выборках.

Параметры
-----
train:
    Обучающий набор.
target:
    Название зависимой переменной.
horizon:
    Горизонт прогнозирования.
lags_range:
    Диапазон значений порядка лагов.
freq: str, значение по умолчанию 'D'
    Частота временного ряда.
aggregate: bool, значение по умолчанию False
    Вычисляет агрегированный лаг.
"""

    if min(lags_range) < horizon:
        warnings.warn(f"\nКоличество периодов для лагов нужно задавать\n"
                      f"равным или больше горизонта прогнозирования")

    if pd.__version__ >= '1.4':
        # создаем метки времени для горизонта
        future_dates = pd.date_range(start=train.index[-1],
                                      periods=horizon + 1,
                                      freq=freq,
                                      inclusive='right')
    else:
        # создаем метки времени для горизонта
```

```

future_dates = pd.date_range(start=train.index[-1],
                             periods=horizon + 1,
                             freq=freq,
                             closed='right')

# формируем новый удлиненный индекс
new_index = train.index.append(future_dates)
# выполняем переиндексацию
new_df = train.reindex(new_index)
# создаем лаги
for i in lags_range:
    new_df[f'Lag_{i}'] = new_df[target].shift(i)

if aggregate and min(lags_range) >= horizon:
    # вычисляем агрегированный лаг
    new_df['Agg_Lag'] = new_df[new_df.filter(
        regex='Lag').columns].mean(axis=1)

train = new_df.iloc[:-horizon]
test = new_df.iloc[-horizon:]

return train, test

```

Применяем нашу функцию, создаем лаги порядка 3, 4 и 5.

```

# создаем лаги для обучающей и тестовой выборок
tr, tst = calculate_lags(
    train, target='sales', horizon=4,
    lags_range=range(3, 6), freq='D')

```

```
/var/folders/x9/w1kvj6ms0p52l8j6crt19vmh0000gn/T/ipykernel_29451/909112346.py:24: UserWarning:
Количество периодов для лагов нужно задавать
равным или больше горизонта прогнозирования
```

Функция предупредила нас, что у нас есть лаг с порядком меньше горизонта. Смотрим лаги в обучающей и тестовой выборках.

```
# смотрим лаги в обучающей выборке
```

```
tr
```

	sales	Lag_3	Lag_4	Lag_5
2018-01-09	2400.0	NaN	NaN	NaN
2018-01-10	2800.0	NaN	NaN	NaN
2018-01-11	2500.0	NaN	NaN	NaN
2018-01-12	2890.0	2400.0	NaN	NaN
2018-01-13	2610.0	2800.0	2400.0	NaN
2018-01-14	2500.0	2500.0	2800.0	2400.0
2018-01-15	2750.0	2890.0	2500.0	2800.0
2018-01-16	2700.0	2610.0	2890.0	2500.0

```
# смотрим лаги в тестовой выборке
tst
```

	sales	Lag_3	Lag_4	Lag_5
2018-01-17	NaN	2500.0	2610.0	2890.0
2018-01-18	NaN	2750.0	2500.0	2610.0
2018-01-19	NaN	2700.0	2750.0	2500.0
2018-01-20	NaN	2700.0	2750.0	

Видим, что *Lag_3* получает пропуск, когда пытается использовать информацию тестовой выборки.

Полезно добавлять агрегированные лаги. Чаще всего речь идет об усредненных лагах. Например, $(L_{t-7} + L_{t-14} + L_{t-21})/3$ или $(L_{t-1} + L_{t-2} + \dots + L_{t-7})/7$. Такие переменные используются в библиотеке Greykite. Однако можно брать не только простое среднее, но и усреднять лаги с использованием различных весов. Можно брать медиану, стандартное отклонение значений лагов. По-прежнему помните, что в агрегации могут участвовать лаги, не использующие информацию теста (т. е. лаги, у которых порядок равен горизонту или превышает его). Наша функция умеет создавать агрегированные лаги. Они будут созданы, когда параметр *aggregate* задан равным *True* и порядок лагов равен или превышает горизонт прогнозирования.

```
# создаем лаги и агрегированный лаг
# для обучающей и тестовой выборок
tr, tst = calculate_lags(
    train, target='sales', horizon=4,
    lags_range=range(4, 6), freq='D', aggregate=True)

# смотрим лаги и агрегированный лаг в обучающей выборке
tr
```

	sales	Lag_4	Lag_5	Agg_Lag
2018-01-09	2400.0	NaN	NaN	NaN
2018-01-10	2800.0	NaN	NaN	NaN
2018-01-11	2500.0	NaN	NaN	NaN
2018-01-12	2890.0	NaN	NaN	NaN
2018-01-13	2610.0	2400.0	NaN	2400.0
2018-01-14	2500.0	2800.0	2400.0	2600.0
2018-01-15	2750.0	2500.0	2800.0	2650.0
2018-01-16	2700.0	2890.0	2500.0	2695.0

```
# смотрим лаги и агрегированный лаг в тестовой выборке
tst
```

	sales	Lag_4	Lag_5	Agg_Lag
2018-01-17	NaN	2610.0	2890.0	2750.0
2018-01-18	NaN	2500.0	2610.0	2555.0
2018-01-19	NaN	2750.0	2500.0	2625.0
2018-01-20	NaN	2700.0	2750.0	2725.0

Теперь напишем функцию `weighted_average_lag()`, которая позволяет вычислить агрегированный лаг на основе взвешенного среднего лагов, и применим ее к лагам в полученной обучающей выборке.

```
# пишем функцию, которая вычисляет агрегированный
# лаг на основе взвешенного среднего лагов
def weighted_average_lag(data, lags, lags_weights,
                         intermediate_results):
    df = data.copy()
    for cnt, i in enumerate(df[lags].columns):
        df[i] = df[i] * lags_weights[cnt]
    if intermediate_results:
        print(df)
    data['Weighted_Average_Lag'] = df[lags].mean(axis=1)
    return data

# вычислим агрегированный лаг на основе
# взвешенного среднего лагов
weighted_average_lag(tr, lags=['Lag_4', 'Lag_5'],
                      lags_weights=[1, 2],
                      intermediate_results=True)
```

	sales	Lag_4	Lag_5	Agg_Lag
2018-01-09	2400.00000	NaN	NaN	NaN
2018-01-10	2800.00000	NaN	NaN	NaN
2018-01-11	2500.00000	NaN	NaN	NaN
2018-01-12	2890.00000	NaN	NaN	NaN
2018-01-13	2610.00000	2400.00000	NaN	2400.00000
2018-01-14	2500.00000	2800.00000	4800.00000	2600.00000
2018-01-15	2750.00000	2500.00000	5600.00000	2650.00000
2018-01-16	2700.00000	2890.00000	5000.00000	2695.00000

	sales	Lag_4	Lag_5	Agg_Lag	Weighted_Average_Lag
2018-01-09	2400.00000	NaN	NaN	NaN	NaN
2018-01-10	2800.00000	NaN	NaN	NaN	NaN
2018-01-11	2500.00000	NaN	NaN	NaN	NaN
2018-01-12	2890.00000	NaN	NaN	NaN	NaN
2018-01-13	2610.00000	2400.00000	NaN	2400.00000	2400.00000
2018-01-14	2500.00000	2800.00000	2400.00000	2600.00000	3800.00000
2018-01-15	2750.00000	2500.00000	2800.00000	2650.00000	4050.00000
2018-01-16	2700.00000	2890.00000	2500.00000	2695.00000	3945.00000

А сейчас вычислим лаги в Polars. Сначала реализуем первый способ защиты от протечек. Для этого датафрейм pandas `data` с защитой от протечек (т. е. со значениями `NaN` для наблюдений, приходящихся на тестовую часть) преобразовываем в датафрейм Polars `polars_data`.

```
# преобразовываем датафрейм pandas в датафрейм Polars
polars_data = pl.DataFrame(data)
polars_data

shape: (12, 1)

sales
_____
f64
2400.0
2800.0
2500.0
2890.0
2610.0
2500.0
2750.0
2700.0
null
null
null
null
```

Помним, что в Polars нет индекса, а пропускам соответствуют значения `null`. Опять формируем лаги порядка 1, 2, 3, 4, 5.

```
# создаем лаги в Polars
polars_data = polars_data.with_columns([
    pl.col('sales').shift(1).alias('Lag_1'),
    pl.col('sales').shift(2).alias('Lag_2'),
    pl.col('sales').shift(3).alias('Lag_3'),
    pl.col('sales').shift(4).alias('Lag_4'),
    pl.col('sales').shift(5).alias('Lag_5')
])
polars_data
```

shape: (12, 6)

sales	Lag_1	Lag_2	Lag_3	Lag_4	Lag_5
f64	f64	f64	f64	f64	f64
2400.0	null	null	null	null	null
2800.0	2400.0	null	null	null	null
2500.0	2800.0	2400.0	null	null	null
2890.0	2500.0	2800.0	2400.0	null	null
2610.0	2890.0	2500.0	2800.0	2400.0	null
2500.0	2610.0	2890.0	2500.0	2800.0	2400.0
2750.0	2500.0	2610.0	2890.0	2500.0	2800.0
2700.0	2750.0	2500.0	2610.0	2890.0	2500.0
null	2700.0	2750.0	2500.0	2610.0	2890.0
null	null	2700.0	2750.0	2500.0	2610.0
null	null	null	2700.0	2750.0	2500.0
null	null	null	null	2700.0	2750.0

Опять видим, что лаги, у которых порядок меньше горизонта прогнозирования, «залезают» в тестовую выборку – последние 4 наблюдения исходного набора и получают пропуски в наблюдениях, приходящихся на тестовую выборку (выделены красными рамками).

Для удобства можем добавить в Polars даты. Даты возьмем из исходного датафрейма pandas.

```
# создаем копию
data2 = data.copy()
# на основе индекса создаем переменную с датами
data2['date'] = data2.index
# столбец с датами ставим первым
first_column = data2.pop('date')
data2.insert(0, 'date', first_column)
# преобразовываем датафрейм pandas в датафрейм Polars
polars_data = pl.DataFrame(data2)
# присваиваем столбцу с датами тип Date
polars_data = polars_data.with_column(
    pl.col('date').cast(pl.Date))
polars_data
```

shape: (12, 2)

	date	sales
	date	f64
2018-01-09	2400.0	
2018-01-10	2800.0	
2018-01-11	2500.0	
2018-01-12	2890.0	
2018-01-13	2610.0	
2018-01-14	2500.0	
2018-01-15	2750.0	
2018-01-16	2700.0	
2018-01-17	null	
2018-01-18	null	
2018-01-19	null	
2018-01-20	null	

```
# создаем список лагов
lags_lst = list(range(1, 6))
# создаем лаги в Polars
for i in lags_lst:
    polars_data = polars_data.with_columns([
        pl.col('sales').shift(i).alias(f'Lag_{i}')
    ])
polars_data
```

shape: (12, 7)

	date	sales	Lag_1	Lag_2	Lag_3	Lag_4	Lag_5
	date	f64	f64	f64	f64	f64	f64
2018-01-09	2400.0	null	null	null	null	null	null
2018-01-10	2800.0	2400.0	null	null	null	null	null
2018-01-11	2500.0	2800.0	2400.0	null	null	null	null
2018-01-12	2890.0	2500.0	2800.0	2400.0	null	null	null
2018-01-13	2610.0	2890.0	2500.0	2800.0	2400.0	null	null
2018-01-14	2500.0	2610.0	2890.0	2500.0	2800.0	2400.0	
2018-01-15	2750.0	2500.0	2610.0	2890.0	2500.0	2800.0	
2018-01-16	2700.0	2750.0	2500.0	2610.0	2890.0	2500.0	
2018-01-17	null	2700.0	2750.0	2500.0	2610.0	2890.0	
2018-01-18	null	null	2700.0	2750.0	2500.0	2610.0	
2018-01-19	null	null	null	2700.0	2750.0	2500.0	
2018-01-20	null	null	null	null	2700.0	2750.0	

Теперь проиллюстрируем второй способ защиты от протечек в Polars.

Обучающий датафрейм pandas `train` с защитой от протечек (т. е. со значениями NaN для наблюдений, приходящихся на тестовую часть) преобразовываем в датафрейм Polars `polars_train_data`.

```
# создаем копию
train2 = train.copy()
# на основе индекса создаем переменную с датами
train2['date'] = train2.index
# столбец с датами ставим первым
first_column = train2.pop('date')
train2.insert(0, 'date', first_column)
# преобразовываем датафрейм pandas в датафрейм Polars
polars_train_data = pl.DataFrame(train2)
# присваиваем столбцу с датами тип Date
polars_train_data = polars_train_data.with_column(
    pl.col('date').cast(pl.Date))
polars_train_data
```

`shape: (8, 2)`

date	sales
date	f64
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

Пишем функцию `polars_calculate_lags()`, аналогичную ранее написанной функции `calculate_lags()`, которая берет нашу обучающую выборку и удлиняет ее на длину горизонта прогнозирования, зависимая переменная в наблюдениях, соответствующих новым временным меткам (т. е. в тестовой выборке / наборе новых данных), получает значения `null`.

```
# пишем функцию для создания лагов
# в обучающей и тестовой выборках
def polars_calculate_lags(polars_train, target_column, date_column,
                           horizon, lags_range, aggregate=False):
    """
```

Создает лаги в обучающей и тестовой выборках.

Параметры

`train`:

Обучающий набор.

`target_column`:

Название зависимой переменной.

`date_column`:

Название переменной с датами.

```

horizon:
    Горизонт прогнозирования.
lags_range:
    Диапазон значений порядка лагов.
aggregate: bool, значение по умолчанию False
    Вычисляет агрегированный лаг.
"""
# вычисляем длину горизонта
h = len(horizon)

if min(lags_range) < h:
    warnings.warn(f"\nКоличество периодов для лагов нужно задавать\n"
                  f"равным или больше горизонта прогнозирования")

# удлиняем набор на длину горизонта
dates = polars_train.select(date_column).to_series()
steps = pl.Series(date_column, horizon).str.strptime(
    pl.Date, fmt='%Y-%m-%d')
final_dates = dates.append(steps).to_frame()
polars_df = polars_train.join(final_dates, how='outer', on=date_column)

# создаем лаги в Polars
for i in lags_range:
    polars_df = polars_df.with_columns([
        pl.col(target_column).shift(i).alias(f'Lag_{i}')
    ])

if aggregate and min(lags_range) >= h:
    # вычисляем агрегированный лаг
    polars_df = polars_df.with_columns([polars_df.select(
        pl.col('^Lag_.*$')).mean(axis=1).alias('Agg_Lag')])

train = polars_df[0:len(final_dates) - h]
test = polars_df[len(final_dates) - h:]

return train, test

```

Применяем нашу функцию, создаем лаги порядка 3, 4 и 5.

```
# создаем лаги для обучающей и тестовой выборок
polars_train, polars_test = polars_calculate_lags(
    polars_train_data, 'sales', 'date',
    horizon=['2018-01-17', '2018-01-18',
              '2018-01-19', '2018-01-20'],
    lags_range=range(3, 6))
```

```
/var/folders/x9/w1kvj6ms0p52l8j6crt19vmh0000gn/T/ipykernel_32991/909112346.py:24: User-
Warning:
Количество периодов для лагов нужно задавать
равным или больше горизонта прогнозирования
warnings.warn(f"\nКоличество периодов для лагов нужно задавать\n"
```

Опять функция предупредила нас, что у нас есть лаг с порядком меньше горизонта.

316 ❖ План предварительной подготовки данных

Смотрим лаги в обучающей и тестовой выборках.

```
# смотрим лаги в обучающей выборке  
polars_train
```

shape: (8, 5)

date	sales	Lag_3	Lag_4	Lag_5
date	f64	f64	f64	f64
2018-01-09	2400.0	null	null	null
2018-01-10	2800.0	null	null	null
2018-01-11	2500.0	null	null	null
2018-01-12	2890.0	2400.0	null	null
2018-01-13	2610.0	2800.0	2400.0	null
2018-01-14	2500.0	2500.0	2800.0	2400.0
2018-01-15	2750.0	2890.0	2500.0	2800.0
2018-01-16	2700.0	2610.0	2890.0	2500.0

```
# смотрим лаги в тестовой выборке  
polars_test
```

shape: (4, 5)

date	sales	Lag_3	Lag_4	Lag_5
date	f64	f64	f64	f64
2018-01-17	null	2500.0	2610.0	2890.0
2018-01-18	null	2750.0	2500.0	2610.0
2018-01-19	null	2700.0	2750.0	2500.0
2018-01-20	null	null	2700.0	2750.0

```
# создаем лаги и агрегированный лаг  
# для обучающей и тестовой выборок  
polars_train, polars_test = polars_calculate_lags(  
    polars_train_data, 'sales', 'date',  
    horizon=['2018-01-17', '2018-01-18',  
            '2018-01-19', '2018-01-20'],  
    lags_range=range(4, 6),  
    aggregate=True)
```

```
# смотрим лаги и агрегированный лаг в обучающей выборке  
polars_train
```

shape: (8, 5)

date	sales	Lag_4	Lag_5	Agg_Lag
date	f64	f64	f64	f64
2018-01-09	2400.0	null	null	null
2018-01-10	2800.0	null	null	null
2018-01-11	2500.0	null	null	null
2018-01-12	2890.0	null	null	null
2018-01-13	2610.0	2400.0	null	2400.0
2018-01-14	2500.0	2800.0	2400.0	2600.0
2018-01-15	2750.0	2500.0	2800.0	2650.0
2018-01-16	2700.0	2890.0	2500.0	2695.0

смотрим лаги и агрегированный лаг в тестовой выборке
polars_test

shape: (4, 5)

date	sales	Lag_4	Lag_5	Agg_Lag
date	f64	f64	f64	f64
2018-01-17	null	2610.0	2890.0	2750.0
2018-01-18	null	2500.0	2610.0	2555.0
2018-01-19	null	2750.0	2500.0	2625.0
2018-01-20	null	2700.0	2750.0	2725.0

16.4.2. Разности

Вы можете создавать разности между значениями соответствующего лага. Например, речь может идти о разнице между продажами на прошлой неделе и продажами на позапрошлой неделе или о разнице между продажами на прошлой неделе и продажами четырьмя неделями ранее.

Давайте в датафрейме pandas data сгенерируем лаги 3, 4, 5, а на их основе – разности в 1 и 2 периода.

```
# создаем лаги
data['Lag3'] = data['sales'].shift(3)
data['Lag4'] = data['sales'].shift(4)
data['Lag5'] = data['sales'].shift(5)

# создаем разности на основе лагов
data['Diff_on_Lag3'] = data['Lag3'].diff()
data['Diff_on_Lag4'] = data['Lag4'].diff()
data['Diff_on_Lag5'] = data['Lag5'].diff()
data['Diff2_on_Lag4'] = data['Lag4'].diff(2)
data['Diff2_on_Lag5'] = data['Lag5'].diff(2)
```

318 ❖ План предварительной подготовки данных

	sales	Lag3	Lag4	Lag5	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	400.0	NaN	NaN	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	-300.0	400.0	NaN	NaN	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	390.0	-300.0	400.0	100.0	NaN
2018-01-16	2700	2610.0	2890.0	2500.0	-280.0	390.0	-300.0	90.0	100.0
2018-01-17	2250	2500.0	2610.0	2890.0	-110.0	-280.0	390.0	110.0	90.0
2018-01-18	2350	2750.0	2500.0	2610.0	250.0	-110.0	-280.0	-390.0	110.0
2018-01-19	2550	2700.0	2750.0	2500.0	-50.0	250.0	-110.0	140.0	-390.0
2018-01-20	3000	2250.0	2700.0	2750.0	-450.0	-50.0	250.0	200.0	140.0

Разобьем наш набор данных так, чтобы в тестовую выборку попало 4 последних наблюдения. Таким образом, горизонт прогнозирования – 4 дня.

```
# разбиваем набор на обучающую и тестовую выборки
train, test = data[0:len(example)-HORIZON], data[len(example)- HORIZON:]
```

```
# смотрим обучающую выборку
train
```

	sales	Lag3	Lag4	Lag5	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
2018-01-09	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890.0	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610.0	2800.0	2400.0	NaN	400.0	NaN	NaN	NaN	NaN
2018-01-14	2500.0	2500.0	2800.0	2400.0	-300.0	400.0	NaN	NaN	NaN
2018-01-15	2750.0	2890.0	2500.0	2800.0	390.0	-300.0	400.0	100.0	NaN
2018-01-16	2700.0	2610.0	2890.0	2500.0	-280.0	390.0	-300.0	90.0	100.0

```
# смотрим тестовую выборку
test
```

	sales	Lag3	Lag4	Lag5	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
2018-01-17	2250	2500.0	2610.0	2890.0	-110.0	-280.0	390.0	110.0	90.0
2018-01-18	2350	2750.0	2500.0	2610.0	250.0	-110.0	-280.0	-390.0	110.0
2018-01-19	2550	2700.0	2750.0	2500.0	-50.0	250.0	-110.0	140.0	-390.0
2018-01-20	3000	2250.0	2700.0	2750.0	-450.0	-50.0	250.0	200.0	140.0

Обратите внимание, что разности, созданные на основе *Lag3* (меньше горизонта прогнозирования), который использует информацию тестовой выборки, также будут использовать информацию тестовой выборки (выделено красной

рамкой). Давайте посмотрим, как образуется «протечка» в разности, вычисленной на основе *Lag3*.



Рис. 65 Разность, созданная на основе лага с порядком меньше горизонта прогнозирования, тоже дает «протечку»

А в остальных разностях «протечки» не происходит.

Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5																																																																								
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-09</td><td>2400</td></tr> <tr><td>2018-01-10</td><td>2800</td></tr> <tr><td>2018-01-11</td><td>2500</td></tr> <tr><td>2018-01-12</td><td>2890</td></tr> <tr><td>2018-01-13</td><td>2610</td></tr> <tr><td>2018-01-14</td><td>2500</td></tr> <tr><td>2018-01-15</td><td>2750</td></tr> <tr><td>2018-01-16</td><td>2700</td></tr> </tbody> </table> <p style="color: red;">-280 (2610-2890) -110 (2500-2610) 250 (2750-2500) -50 (2700-2750)</p>	sales		2018-01-09	2400	2018-01-10	2800	2018-01-11	2500	2018-01-12	2890	2018-01-13	2610	2018-01-14	2500	2018-01-15	2750	2018-01-16	2700	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-09</td><td>2400</td></tr> <tr><td>2018-01-10</td><td>2800</td></tr> <tr><td>2018-01-11</td><td>2500</td></tr> <tr><td>2018-01-12</td><td>2890</td></tr> <tr><td>2018-01-13</td><td>2610</td></tr> <tr><td>2018-01-14</td><td>2500</td></tr> <tr><td>2018-01-15</td><td>2750</td></tr> <tr><td>2018-01-16</td><td>2700</td></tr> </tbody> </table> <p style="color: blue;">390 (2890-2500) -200 (2610-2890) -110 (2500-2610) 250 (2750-2500)</p>	sales		2018-01-09	2400	2018-01-10	2800	2018-01-11	2500	2018-01-12	2890	2018-01-13	2610	2018-01-14	2500	2018-01-15	2750	2018-01-16	2700	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-09</td><td>2400</td></tr> <tr><td>2018-01-10</td><td>2800</td></tr> <tr><td>2018-01-11</td><td>2500</td></tr> <tr><td>2018-01-12</td><td>2890</td></tr> <tr><td>2018-01-13</td><td>2610</td></tr> <tr><td>2018-01-14</td><td>2500</td></tr> <tr><td>2018-01-15</td><td>2750</td></tr> <tr><td>2018-01-16</td><td>2700</td></tr> </tbody> </table> <p style="color: red;">110 (2610-2500) -390 (2500-2890) -140 (2750-2610) 200 (2700-2500)</p>	sales		2018-01-09	2400	2018-01-10	2800	2018-01-11	2500	2018-01-12	2890	2018-01-13	2610	2018-01-14	2500	2018-01-15	2750	2018-01-16	2700	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-09</td><td>2400</td></tr> <tr><td>2018-01-10</td><td>2800</td></tr> <tr><td>2018-01-11</td><td>2500</td></tr> <tr><td>2018-01-12</td><td>2890</td></tr> <tr><td>2018-01-13</td><td>2610</td></tr> <tr><td>2018-01-14</td><td>2500</td></tr> <tr><td>2018-01-15</td><td>2750</td></tr> <tr><td>2018-01-16</td><td>2700</td></tr> </tbody> </table> <p style="color: blue;">90 (2890-2800) 110 (2610-2500) -390 (2500-2890) 140 (2750-2610)</p>	sales		2018-01-09	2400	2018-01-10	2800	2018-01-11	2500	2018-01-12	2890	2018-01-13	2610	2018-01-14	2500	2018-01-15	2750	2018-01-16	2700
sales																																																																											
2018-01-09	2400																																																																										
2018-01-10	2800																																																																										
2018-01-11	2500																																																																										
2018-01-12	2890																																																																										
2018-01-13	2610																																																																										
2018-01-14	2500																																																																										
2018-01-15	2750																																																																										
2018-01-16	2700																																																																										
sales																																																																											
2018-01-09	2400																																																																										
2018-01-10	2800																																																																										
2018-01-11	2500																																																																										
2018-01-12	2890																																																																										
2018-01-13	2610																																																																										
2018-01-14	2500																																																																										
2018-01-15	2750																																																																										
2018-01-16	2700																																																																										
sales																																																																											
2018-01-09	2400																																																																										
2018-01-10	2800																																																																										
2018-01-11	2500																																																																										
2018-01-12	2890																																																																										
2018-01-13	2610																																																																										
2018-01-14	2500																																																																										
2018-01-15	2750																																																																										
2018-01-16	2700																																																																										
sales																																																																											
2018-01-09	2400																																																																										
2018-01-10	2800																																																																										
2018-01-11	2500																																																																										
2018-01-12	2890																																																																										
2018-01-13	2610																																																																										
2018-01-14	2500																																																																										
2018-01-15	2750																																																																										
2018-01-16	2700																																																																										
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-17</td><td>2250</td></tr> <tr><td>2018-01-18</td><td>2350</td></tr> <tr><td>2018-01-19</td><td>2550</td></tr> <tr><td>2018-01-20</td><td>3000</td></tr> </tbody> </table>	sales		2018-01-17	2250	2018-01-18	2350	2018-01-19	2550	2018-01-20	3000	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-17</td><td>2250</td></tr> <tr><td>2018-01-18</td><td>2350</td></tr> <tr><td>2018-01-19</td><td>2550</td></tr> <tr><td>2018-01-20</td><td>3000</td></tr> </tbody> </table>	sales		2018-01-17	2250	2018-01-18	2350	2018-01-19	2550	2018-01-20	3000	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-17</td><td>2250</td></tr> <tr><td>2018-01-18</td><td>2350</td></tr> <tr><td>2018-01-19</td><td>2550</td></tr> <tr><td>2018-01-20</td><td>3000</td></tr> </tbody> </table>	sales		2018-01-17	2250	2018-01-18	2350	2018-01-19	2550	2018-01-20	3000	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr><th colspan="2">sales</th></tr> </thead> <tbody> <tr><td>2018-01-17</td><td>2250</td></tr> <tr><td>2018-01-18</td><td>2350</td></tr> <tr><td>2018-01-19</td><td>2550</td></tr> <tr><td>2018-01-20</td><td>3000</td></tr> </tbody> </table>	sales		2018-01-17	2250	2018-01-18	2350	2018-01-19	2550	2018-01-20	3000																																
sales																																																																											
2018-01-17	2250																																																																										
2018-01-18	2350																																																																										
2018-01-19	2550																																																																										
2018-01-20	3000																																																																										
sales																																																																											
2018-01-17	2250																																																																										
2018-01-18	2350																																																																										
2018-01-19	2550																																																																										
2018-01-20	3000																																																																										
sales																																																																											
2018-01-17	2250																																																																										
2018-01-18	2350																																																																										
2018-01-19	2550																																																																										
2018-01-20	3000																																																																										
sales																																																																											
2018-01-17	2250																																																																										
2018-01-18	2350																																																																										
2018-01-19	2550																																																																										
2018-01-20	3000																																																																										
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding-bottom: 5px;">sales</th> <th style="text-align: left; padding-bottom: 5px;">Diff_on_Lag3</th> <th style="text-align: left; padding-bottom: 5px;">Diff_on_Lag4</th> <th style="text-align: left; padding-bottom: 5px;">Diff_on_Lag5</th> <th style="text-align: left; padding-bottom: 5px;">Diff2_on_Lag4</th> <th style="text-align: left; padding-bottom: 5px;">Diff2_on_Lag5</th> </tr> </thead> <tbody> <tr> <td>2018-01-17</td><td>2250</td><td>-110.0</td><td>-280.0</td><td>390.0</td><td>110.0</td><td>90.0</td></tr> <tr> <td>2018-01-18</td><td>2350</td><td>250.0</td><td>-110.0</td><td>-280.0</td><td>-390.0</td><td>110.0</td></tr> <tr> <td>2018-01-19</td><td>2550</td><td>-50.0</td><td>250.0</td><td>-110.0</td><td>140.0</td><td>-390.0</td></tr> <tr> <td>2018-01-20</td><td>3000</td><td>-450.0</td><td>-50.0</td><td>250.0</td><td>200.0</td><td>140.0</td></tr> </tbody> </table>				sales	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5	2018-01-17	2250	-110.0	-280.0	390.0	110.0	90.0	2018-01-18	2350	250.0	-110.0	-280.0	-390.0	110.0	2018-01-19	2550	-50.0	250.0	-110.0	140.0	-390.0	2018-01-20	3000	-450.0	-50.0	250.0	200.0	140.0																																						
sales	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5																																																																						
2018-01-17	2250	-110.0	-280.0	390.0	110.0	90.0																																																																					
2018-01-18	2350	250.0	-110.0	-280.0	-390.0	110.0																																																																					
2018-01-19	2550	-50.0	250.0	-110.0	140.0	-390.0																																																																					
2018-01-20	3000	-450.0	-50.0	250.0	200.0	140.0																																																																					

Рис. 66 Разности, созданные на основе лагов с порядком, равным горизонту прогнозирования и выше, не дают «протечек»

Теперь создадим разности в polars. Для этого воспользуемся датафреймом Polars polars_data. Давайте удалим лаги 1, 2 и 3 и создадим разности 1 и 2 с помощью метода .diff().

```
# удаляем лаги 1, 2, 3
polars_data = polars_data.drop(['Lag_1', 'Lag_2', 'Lag_3'])
# создаем разности в Polars
polars_data = polars_data.with_columns([
    pl.col('Lag_4').diff(1).alias('Diff_on_Lag4'),
    pl.col('Lag_5').diff(1).alias('Diff_on_Lag5'),
    pl.col('Lag_4').diff(2).alias('Diff2_on_Lag4'),
    pl.col('Lag_5').diff(2).alias('Diff2_on_Lag5')
])
polars_data
```

shape: (12, 8)

date	sales	Lag_4	Lag_5	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
date	f64	f64	f64	f64	f64	f64	f64
2018-01-09	2400.0	null	null	null	null	null	null
2018-01-10	2800.0	null	null	null	null	null	null
2018-01-11	2500.0	null	null	null	null	null	null
2018-01-12	2890.0	null	null	null	null	null	null
2018-01-13	2610.0	2400.0	null	null	null	null	null
2018-01-14	2500.0	2800.0	2400.0	400.0	null	null	null
2018-01-15	2750.0	2500.0	2800.0	-300.0	400.0	100.0	null
2018-01-16	2700.0	2890.0	2500.0	390.0	-300.0	90.0	100.0
2018-01-17	null	2610.0	2890.0	-280.0	390.0	110.0	90.0
2018-01-18	null	2500.0	2610.0	-110.0	-280.0	-390.0	110.0
2018-01-19	null	2750.0	2500.0	250.0	-110.0	140.0	-390.0
2018-01-20	null	2700.0	2750.0	-50.0	250.0	200.0	140.0

16.4.3. Скользящие статистики

Кроме лагов и разностей, в качестве признаков можно использовать статистики, меняющиеся со временем, – скользящие статистики. В рамках подхода «скользящее окно» библиотека pandas вычисляет статистику по «окну» данных, представляющему определенный период времени. Затем окно смещается на определенный интервал времени и статистика постоянно вычисляется для каждого нового окна до тех пор, пока окно охватывает даты временного ряда. Библиотека pandas непосредственно поддерживает скользящие оконные функции, предлагая метод .rolling() объекта Series и объекта DataFrame. В таблице ниже приведен ряд таких функций.

Таблица 10 Скользящие оконные функции в библиотеке pandas

Функция	Описание
<code>.rolling().mean()</code>	Среднее значение в окне
<code>.rolling().std()</code>	Стандартное отклонение в окне
<code>.rolling().var()</code>	Дисперсия в окне
<code>.rolling().min()</code>	Минимальное значение в окне
<code>.rolling().max()</code>	Максимальное значение в окне
<code>.rolling().cov()</code>	Коэффициент ковариации в окне
<code>.rolling().quantile()</code>	Оценка для заданного процентиля / выборочного квантиля в окне
<code>.rolling().corr()</code>	Коэффициент корреляции в окне
<code>.rolling().median()</code>	Медиана в окне
<code>.rolling().sum()</code>	Сумма в окне
<code>.rolling().apply()</code>	Применение пользовательской функции к значениям окна
<code>.rolling().count()</code>	Количество непропущенных значений в окне
<code>.rolling().skew()</code>	Коэффициент асимметрии в окне
<code>.rolling().kurt()</code>	Коэффициент эксцесса в окне

Самая часто используемая скользящая статистика – простое скользящее среднее (simple moving average):

$$SMA = \frac{A_1 + A_2 + \dots + A_n}{n}.$$

Заметим, что скользящее среднее используется не только для конструирования признаков, но и в качестве прогнозной модели (когда прогноз – скользящее среднее n последних наблюдений), а также для сглаживания выбросов, краткосрочных колебаний и более четкого выделения долгосрочных тенденций в ряде данных. На практике ширину окна скользящего среднего устанавливают равной горизонту прогнозирования или больше его. При этом ширина окна скользящей статистики не должна быть равна или превышать длину набора данных, на основе которого она создается, в противном случае она будет состоять из одного значения и пропусков.

Заново загрузим данные продаж и вычислим скользящие средние с шириной окна 3, шириной окна 12 (когда ширина окна равна длине набора), шириной окна 13 (когда ширина окна больше длины набора). Для этого воспользуемся параметром `window` метода `.rolling()`.

```
# создаем скользящие средние с размерами окна 3, 12 и 13
```

```
data['rolling_mean3'] = data['sales'].rolling(window=3).mean()
data['rolling_mean12'] = data['sales'].rolling(window=12).mean()
data['rolling_mean13'] = data['sales'].rolling(window=13).mean()
data
```

	sales	rolling_mean3	rolling_mean12	rolling_mean13
2018-01-09	2400			
2018-01-10	2800			
2018-01-11	2500			
2018-01-12	2890			
2018-01-13	2610			
2018-01-14	2500			
2018-01-15	2750			
2018-01-16	2700			
2018-01-17	2250			
2018-01-18	2350			
2018-01-19	2550			
2018-01-20	3000			
		NaN	NaN	NaN
		NaN	NaN	NaN
		2566.666667	NaN	NaN
		2730.000000	NaN	NaN
		2666.666667	NaN	NaN
		2666.666667	NaN	NaN
		2620.000000	NaN	NaN
		2650.000000	NaN	NaN
		2566.666667	NaN	NaN
		2433.333333	NaN	NaN
		2383.333333	NaN	NaN
		2633.333333	2608.333333	NaN

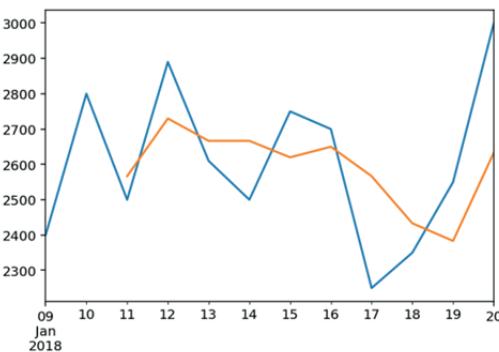
Видим, что скользящее среднее с шириной окна, равной длине набора, состоит только из одного непропущенного значения, а скользящее среднее с шириной окна, превышающей длину набора, состоит из одних пропусков.

Давайте визуализируем скользящее среднее с шириной окна 3.

```
# визуализируем продажи и скользящее
```

```
# среднее продаж с шириной окна 3
```

```
data['sales'].plot()
data['rolling_mean3'].plot();
```



Для уменьшения количества значений NaN можно регулировать значение параметра `min_periods` метода `.rolling()`. Этот параметр задает минимальное количество наблюдений в окне, требуемое для вычисления значения (в противном случае результатом будет значение NaN). По умолчанию значение параметра `min_periods` равно значению параметра `window`.

Давайте создадим скользящее среднее с шириной окна 3 и минимальным количеством наблюдений в окне, равным 2. Для этого воспользуемся параметром `window` метода `.rolling()`.

```
# создаем скользящие средние, окно шириной 3 и с минимальным
# количеством наблюдений в окне, равным 2
data['rolling_mean3_min_periods_2'] = data['sales'].rolling(
    window=3, min_periods=2).mean()
data
```

	<code>sales</code>	<code>rolling_mean3</code>	<code>rolling_mean12</code>	<code>rolling_mean13</code>	<code>rolling_mean3_min_periods_2</code>
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	2600.000000
2018-01-11	2500	2566.666667	NaN	NaN	2566.666667
2018-01-12	2890	2730.000000	NaN	NaN	2730.000000
2018-01-13	2610	2666.666667	NaN	NaN	2666.666667
2018-01-14	2500	2666.666667	NaN	NaN	2666.666667
2018-01-15	2750	2620.000000	NaN	NaN	2620.000000
2018-01-16	2700	2650.000000	NaN	NaN	2650.000000
2018-01-17	2250	2566.666667	NaN	NaN	2566.666667
2018-01-18	2350	2433.333333	NaN	NaN	2433.333333
2018-01-19	2550	2383.333333	NaN	NaN	2383.333333
2018-01-20	3000	2633.333333	2608.333333	NaN	2633.333333

Скользящие статистики можно вычислить в исходном наборе, рассматривая последние N наблюдений в качестве тестовой выборки, где N – длина горизонта. После вычисления статистик в исходном наборе происходит разбиение на обучающую и тестовую выборки и строится модель.

Самый простой способ получить скользящие статистики, не используя информацию тестовой выборки, – вычислить их с лагом, равным горизонту прогнозирования (или выше). Если горизонт прогнозирования равен 4 дням, скользящее среднее нужно вычислять с лагом 4. Параметр `min_periods` обычно задают равным 1, в противном случае у нас будет большое количество пропусков. Давайте вычислим скользящие средние с шириной окна 3 и шириной окна 4 с лагом 3, 4 и 5, предварительно удалив ранее созданные переменные. Последние 4 наблюдения набора рассматриваем как будущую тестовую выборку, защиту от протечек не формируем.

```
# удалили переменные по паттерну 'rolling_mean'
data.drop(data.filter(regex='rolling_mean').columns,
           axis=1, inplace=True)

# вычисляем скользящие средние с шириной 3 и 4
# и с лагом 3, 4, 5 и min_periods 1
data['rolling_mean3_lag3'] = data['sales'].shift(periods=3).rolling(
    min_periods=1, window=3).mean()
```

```
data['rolling_mean4_lag3'] = data['sales'].shift(periods=3).rolling(
    min_periods=1, window=4).mean()
data['rolling_mean3_lag4'] = data['sales'].shift(periods=4).rolling(
    min_periods=1, window=3).mean()
data['rolling_mean4_lag4'] = data['sales'].shift(periods=4).rolling(
    min_periods=1, window=4).mean()
data['rolling_mean3_lag5'] = data['sales'].shift(periods=5).rolling(
    min_periods=1, window=3).mean()
data['rolling_mean4_lag5'] = data['sales'].shift(periods=5).rolling(
    min_periods=1, window=4).mean()
data
```

	sales	rolling_mean3_lag3	rolling_mean4_lag3	rolling_mean3_lag4	rolling_mean4_lag4	rolling_mean3_lag5	rolling_mean4_lag5
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.000000	2400.000000	NaN	NaN	NaN	NaN
2018-01-13	2610	2600.000000	2600.000000	2400.000000	2400.000000	NaN	NaN
2018-01-14	2500	2566.666667	2566.666667	2600.000000	2600.000000	2400.000000	2400.000000
2018-01-15	2750	2730.000000	2647.500000	2566.666667	2566.666667	2600.000000	2600.000000
2018-01-16	2700	2666.666667	2700.000000	2730.000000	2647.500000	2566.666667	2566.666667
2018-01-17	2250	2666.666667	2625.000000	2666.666667	2700.000000	2730.000000	2647.500000
2018-01-18	2350	2620.000000	2687.500000	2666.666667	2625.000000	2666.666667	2700.000000
2018-01-19	2550	2650.000000	2640.000000	2620.000000	2687.500000	2666.666667	2625.000000
2018-01-20	3000	2566.666667	2550.000000	2650.000000	2640.000000	2620.000000	2687.500000

Наша
тестовая выборка

Теперь внимательно посмотрим, как была вычислена каждая скользящая статистика. Наблюдения обучающей выборки пометим синим цветом, наблюдения тестовой выборки пометим красным цветом.

Давайте посмотрим, как вычисляется скользящее среднее с шириной окна 3 и лагом 3 (*rolling_mean3_lag3*).

```
# вычисление rolling_mean3_lag3
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2800 + 2500 + 2890) / 3)
print((2500 + 2890 + 2610) / 3)
print((2890 + 2610 + 2500) / 3)
print((2610 + 2500 + 2750) / 3)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700 + 2250) / 3)
```

	<i>rolling_mean3_lag3</i>
nan	NaN
nan	NaN
nan	NaN
2400.0	2400.000000
2600.0	2600.000000
2566.666666666665	2566.666667
2730.0	2730.000000
2666.666666666665	2666.666667
2666.666666666665	2666.666667
2620.0	2620.000000
2650.0	2650.000000
2566.666666666665	2566.666667

Видим, что скользящее среднее с шириной окна 3 и лагом 3 в последнем вычислении использует наблюдение тестовой выборки.

Смотрим, как вычисляется скользящее среднее с шириной окна 4 и лагом 3 (*rolling_mean4_lag3*).

```
# вычисление rolling_mean4_lag3
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700 + 2250) / 4)
```

	<i>rolling_mean4_lag3</i>
nan	NaN
nan	NaN
nan	NaN
2400.0	2400.000000
2600.0	2600.000000
2566.666666666665	2566.666667
2647.5	2647.500000
2700.0	2700.000000
2625.0	2625.000000
2687.5	2687.500000
2640.0	2640.000000
2550.0	2550.000000

Видим, что скользящее среднее с шириной окна 4 и лагом 3 в последнем вычислении использует наблюдение тестовой выборки.

Теперь смотрим, как вычисляется скользящее среднее с шириной окна 3 и лагом 4 (*rolling_mean3_lag4*), т. е. порядок лага совпадает с горизонтом прогнозирования.

```
# вычисление rolling_mean3_lag4
print(np.nan)
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2800 + 2500 + 2890) / 3)
print((2500 + 2890 + 2610) / 3)
print((2890 + 2610 + 2500) / 3)
print((2610 + 2500 + 2750) / 3)
print((2500 + 2750 + 2700) / 3)
```

	rolling_mean3_lag4
nan	NaN
2400	2400.000000
2600.0	2600.000000
2566.666666666665	2566.666667
2730.0	2730.000000
2666.666666666665	2666.666667
2666.666666666665	2666.666667
2620.0	2620.000000
2650.0	2650.000000

Видим, что скользящее среднее с шириной окна 3 и лагом 4 ни в одном из вычислений не использует информацию тестовой выборки.

Теперь смотрим, как вычисляется скользящее среднее с шириной окна 4 и лагом 4 (*rolling_mean4_lag4*), т. е. вновь порядок лага совпадает с горизонтом прогнозирования.

```
# вычисление rolling_mean4_lag4
print(np.nan)
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
```

	rolling_mean4_lag4
nan	NaN
2400.0	2400.000000
2600.0	2600.000000
2566.666666666665	2566.666667
2647.5	2647.500000
2700.0	2700.000000
2625.0	2625.000000
2687.5	2687.500000
2640.0	2640.000000

Видим, что скользящее среднее с шириной окна 4 и лагом 4 ни в одном из вычислений тоже не использует информацию тестовой выборки.

Легко понять, что скользящее среднее с шириной окна 3 и лагом 5 и скользящее среднее с шириной окна 4 и лагом 5 тоже не используют информацию тестовой выборки.

Теперь посмотрим, какие результаты мы бы получили, если бы оставили значение `min_periods` по умолчанию.

```
# вычисляем скользящие средние с шириной 3 и 4
# и с лагом 3, 4, 5 с min_periods по умолчанию
data['rolling_mean3_lag3'] = data['sales'].shift(periods=3).rolling(
    window=3).mean()
data['rolling_mean4_lag3'] = data['sales'].shift(periods=3).rolling(
    window=4).mean()
data['rolling_mean3_lag4'] = data['sales'].shift(periods=4).rolling(
    window=3).mean()
data['rolling_mean4_lag4'] = data['sales'].shift(periods=4).rolling(
    window=4).mean()
data['rolling_mean3_lag5'] = data['sales'].shift(periods=5).rolling(
    window=3).mean()
data['rolling_mean4_lag5'] = data['sales'].shift(periods=5).rolling(
    window=4).mean()
data
```

	sales	rolling_mean3_lag3	rolling_mean4_lag3	rolling_mean3_lag4	rolling_mean4_lag4	rolling_mean3_lag5	rolling_mean4_lag5
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-14	2500	2566.666667	NaN	NaN	NaN	NaN	NaN
2018-01-15	2750	2730.000000	2647.5	2566.666667	NaN	NaN	NaN
2018-01-16	2700	2666.666667	2700.0	2730.000000	2647.5	2566.666667	NaN
2018-01-17	2250	2666.666667	2625.0	2666.666667	2700.0	2730.000000	2647.5
2018-01-18	2350	2620.000000	2687.5	2666.666667	2625.0	2666.666667	2700.0
2018-01-19	2550	2650.000000	2640.0	2620.000000	2687.5	2666.666667	2625.0
2018-01-20	3000	2566.666667	2550.0	2650.000000	2640.0	2620.000000	2687.5

Видим, что у нас возрастает количество пропусков. Например, для столбца `rolling_mean4_lag5` вся обучающая часть состоит из пропусков.

Попробуем второй способ получить скользящие статистики, не используя информацию тестовой выборки.

Мы предварительно удалим ранее созданные скользящие средние, а значения в наблюдениях, которые будут приходиться на тестовую выборку (последние 4 наблюдения исходного набора), заменим на значения `NaN`, тем самым сформировав защиту от протечек.

```
# удалим переменные по паттерну 'rolling_mean'
data.drop(data.filter(regex='rolling_mean').columns,
           axis=1, inplace=True)

# значения в наблюдениях, которые будут приходиться на тест
# (последние 4 наблюдения), заменим на значения NaN
data['sales'].iloc[-HORIZON:] = np.NaN
data
```

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0
2018-01-17	NaN
2018-01-18	NaN
2018-01-19	NaN
2018-01-20	NaN



Теперь создаем столбец `rolling_mean4` – скользящее среднее с шириной окна 4, при этом скользящее среднее считаем с лагом 1. Позже поясним, зачем нам нужно вычислять скользящее среднее с лагом 1. Из-за лага скользящие средние будут записываться в столбец `rolling_mean4`, начиная с 5-й строки (а не с 4-й).

```
# вычисляем скользящее среднее с шириной окна 4 и лагом 1
data['rolling_mean4_lag1'] = data['sales'].shift(
    periods=1).rolling(window=4, min_periods=1).mean()
data
```

	sales	rolling_mean4_lag1
2018-01-09	2400.0	NaN
2018-01-10	2800.0	2400.000000
2018-01-11	2500.0	2600.000000
2018-01-12	2890.0	2566.666667
2018-01-13	2610.0	2647.500000
2018-01-14	2500.0	2700.000000
2018-01-15	2750.0	2625.000000
2018-01-16	2700.0	2687.500000
2018-01-17	NaN	2640.000000
2018-01-18	NaN	2650.000000
2018-01-19	NaN	2725.000000
2018-01-20	NaN	2700.000000

Наша
тестовая выборка

Посмотрим, как было вычислено скользящее среднее с шириной окна 4 и лагом 1.

```
# вычисление rolling_mean4_lag1
print(пр.нан)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700) / 2)
print(2700 / 1)
```

	rolling_mean4_lag1
nan	NaN
2400.0	2400.000000
2600.0	2600.000000
2566.666666666665	2566.666667
2647.5	2647.500000
2700.0	2700.000000
2625.0	2625.000000
2687.5	2687.500000
2640.0	2640.000000
2650.0	2650.000000
2725.0	2725.000000
2700.0	2700.000000

Взглянем на первое скользящее среднее в тестовой выборке. Это будет среднее значение, взятое по последним 4 наблюдениям обучающей выборки.

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

2640

Второе, третье и четвертое скользящее среднее вычисляем как среднее, взятое по последним трем, двум, одному наблюдению в обучающей выборке соответственно.

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

2650

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

2725

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

2700

Таким образом, при вычислении скользящих средних для более поздних моментов времени в тестовой выборке мы используем средние, взятые по более поздним наблюдениям в обучающей выборке, с уменьшением ширины окна. Уменьшение ширины окна позволяет учитывать только более поздние наблюдения в обучающей выборке при вычислении скользящих средних для более поздних моментов времени в тестовой выборке. При вычислении скользящих средних для тестовой выборки мы ни разу не использовали значения продаж в тестовой выборке.

Легко понять, если бы мы вычисляли скользящее среднее с шириной окна 4 без лага 1, то для последнего наблюдения мы не смогли бы вычислить скользящее среднее.

```
# вычисляем скользящее среднее с шириной окна 4 без лага
data['rolling_mean4'] = data['sales'].rolling(
    window=4, min_periods=1).mean()
data
```

	sales	rolling_mean4_lag1	rolling_mean4
2018-01-09	2400.0	NaN	2400.000000
2018-01-10	2800.0	2400.000000	2600.000000
2018-01-11	2500.0	2600.000000	2566.666667
2018-01-12	2890.0	2566.666667	2647.500000
2018-01-13	2610.0	2647.500000	2700.000000
2018-01-14	2500.0	2700.000000	2625.000000
2018-01-15	2750.0	2625.000000	2687.500000
2018-01-16	2700.0	2687.500000	2640.000000
2018-01-17	NaN	2640.000000	2650.000000
2018-01-18	NaN	2650.000000	2725.000000
2018-01-19	NaN	2725.000000	2700.000000
2018-01-20	NaN	2700.000000	NaN

Для скользящего среднего с шириной окна больше горизонта прогнозирования лаг 1 не требуется. Давайте вычислим скользящие средние с шириной окна 5 без лага и с лагом.

```
# вычисляем скользящие средние с шириной окна
# 5 без лага и с лагом
data['rolling_mean5'] = data['sales'].rolling(
    window=5, min_periods=1).mean()
data['rolling_mean5_lag1'] = data['sales'].shift(1).rolling(
    window=5, min_periods=1).mean()
data
```

	sales	rolling_mean4_lag1	rolling_mean4	rolling_mean5	rolling_mean5_lag1
2018-01-09	2400.0	NaN	2400.000000	2400.000000	NaN
2018-01-10	2800.0	2400.000000	2600.000000	2600.000000	2400.000000
2018-01-11	2500.0	2600.000000	2566.666667	2566.666667	2600.000000
2018-01-12	2890.0	2566.666667	2647.500000	2647.500000	2566.666667
2018-01-13	2610.0	2647.500000	2700.000000	2640.000000	2647.500000
2018-01-14	2500.0	2700.000000	2625.000000	2660.000000	2640.000000
2018-01-15	2750.0	2625.000000	2687.500000	2650.000000	2660.000000
2018-01-16	2700.0	2687.500000	2640.000000	2690.000000	2650.000000
2018-01-17	NaN	2640.000000	2650.000000	2640.000000	2690.000000
2018-01-18	NaN	2650.000000	2725.000000	2650.000000	2640.000000
2018-01-19	NaN	2725.000000	2700.000000	2725.000000	2650.000000
2018-01-20	NaN	2700.000000	NaN	2700.000000	2725.000000

Посмотрим, как было вычислено скользящее среднее с шириной окна 5 без лага.

```
# вычисление rolling_mean5
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2400 + 2800 + 2500 + 2890 + 2610) / 5)
```

```
print((2800 + 2500 + 2890 + 2610 + 2500) / 5)
print((2500 + 2890 + 2610 + 2500 + 2750) / 5)
print((2890 + 2610 + 2500 + 2750 + 2700) / 5)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700) / 2)
print(2700 / 1)
```

```
2400.0
2600.0
2566.666666666665
2647.5
2640.0
2660.0
2650.0
2690.0
2640.0
2650.0
2725.0
2700.0
```

Рекомендации для второго способа – используйте лаг 1, если берете ширину окна скользящего среднего по горизонту прогнозирования, можно отказаться от лага, если ширина окна превышает горизонт прогнозирования, не берите ширину окна, равную или превышающую длину обучающей выборки, используйте `min_periods=1`.

Универсальной рекомендацией для обоих способов является защита от протечек в виде значений `Nan` для тестовой части набора.

Кроме того, можно вычислить агрегированные скользящие средние. Такие переменные используются в библиотеке Greykite. Вы можете брать не только простое среднее скользящих средних, но и усреднять скользящие средние с применением различных весов. Можно брать медиану, стандартное отклонение скользящих средних. И опять по-прежнему помните, что в агрегации могут участвовать скользящие статистики, не использующие информацию теста.

Давайте напишем функцию `average_moving_stats()`, которая вычисляет агрегированные скользящие статистики на основе усреднения – обычного и с весами.

```
# пишем функцию, которая вычисляет агрегированные скользящие
# статистики на основе усреднения – обычного и с весами
def average_moving_stats(data, moving_stats, moving_stats_weights,
                           intermediate_results, out_column):
    df = data.copy()
    for cnt, i in enumerate(df[moving_stats].columns):
        df[i] = df[i] * moving_stats_weights[cnt]
    if intermediate_results:
        print(df)
    data[f'{out_column}'] = df[moving_stats].mean(axis=1)
    return data

# вычисляем обычное среднее скользящих средних
data = average_moving_stats(
    data=data,
```

```

moving_stats=['rolling_mean4', 'rolling_mean5'],
moving_stats_weights=[1, 1],
intermediate_results=False,
out_column='averaged_moving_mean')
data

```

	sales	rolling_mean4_lag1	rolling_mean4	rolling_mean5	rolling_mean5_lag1	averaged_moving_mean
2018-01-09	2400.0	NaN	2400.000000	2400.000000	NaN	2400.000000
2018-01-10	2800.0	2400.000000	2600.000000	2600.000000	2400.000000	2600.000000
2018-01-11	2500.0	2600.000000	2566.666667	2566.666667	2600.000000	2566.666667
2018-01-12	2890.0	2566.666667	2647.500000	2647.500000	2566.666667	2647.500000
2018-01-13	2610.0	2647.500000	2700.000000	2640.000000	2647.500000	2670.000000
2018-01-14	2500.0	2700.000000	2625.000000	2660.000000	2640.000000	2642.500000
2018-01-15	2750.0	2625.000000	2687.500000	2650.000000	2660.000000	2668.750000
2018-01-16	2700.0	2687.500000	2640.000000	2690.000000	2650.000000	2665.000000
2018-01-17	NaN	2640.000000	2650.000000	2640.000000	2690.000000	2645.000000
2018-01-18	NaN	2650.000000	2725.000000	2650.000000	2640.000000	2687.500000
2018-01-19	NaN	2725.000000	2700.000000	2725.000000	2650.000000	2712.500000
2018-01-20	NaN	2700.000000		NaN	2700.000000	2700.000000

Кроме того, можно создавать скользящие средние с регуляризацией, когда мы случайно пропускаем наблюдения для подсчета статистики по окну.

```

# функция вычисления скользящего среднего с регуляризацией
def regularized_mean(x, frac=0.5, random_state=42, replace=False,
                      verbose=False, axis=None):
    np.random.seed(random_state)
    x_sampled = x.sample(frac=frac, replace=replace)
    if verbose:
        print(x_sampled)
        print('')
    return np.mean(x_sampled)

# вычисляем скользящее среднее с регуляризацией
data['sales'].shift(1).rolling(window=4, min_periods=1).apply(
    lambda x: regularized_mean(x, frac=0.8, verbose=True))

2018-01-10      2400.0
2018-01-09      NaN
dtype: float64

2018-01-09      NaN
2018-01-10      2400.0
dtype: float64

2018-01-10      2400.0
2018-01-12      2500.0
2018-01-09      NaN
dtype: float64

```

334 ❖ План предварительной подготовки данных

```
2018-01-11    2800.0
2018-01-13    2890.0
2018-01-10    2400.0
dtype: float64

2018-01-12    2500.0
2018-01-14    2610.0
2018-01-11    2800.0
dtype: float64

2018-01-13    2890.0
2018-01-15    2500.0
2018-01-12    2500.0
dtype: float64

2018-01-14    2610.0
2018-01-16    2750.0
2018-01-13    2890.0
dtype: float64

2018-01-15    2500.0
2018-01-17    2700.0
2018-01-14    2610.0
dtype: float64

2018-01-16    2750.0
2018-01-18      NaN
2018-01-15    2500.0
dtype: float64

2018-01-17    2700.0
2018-01-19      NaN
2018-01-16    2750.0
dtype: float64

2018-01-18      NaN
2018-01-20      NaN
2018-01-17    2700.0
dtype: float64

2018-01-09        NaN
2018-01-10  2400.000000
2018-01-11  2400.000000
2018-01-12  2450.000000
2018-01-13  2696.666667
2018-01-14  2636.666667
2018-01-15  2630.000000
2018-01-16  2750.000000
2018-01-17  2603.333333
2018-01-18  2625.000000
2018-01-19  2725.000000
2018-01-20  2700.000000
Name: sales, dtype: float64
```

Разумеется, не стоит ограничиваться вычислением скользящих средних, вы можете попробовать применить скользящие медианы, суммы, минимумы, максимумы, стандартные отклонения, скользящие средние абсолютные отклонения, скользящие медианные абсолютные отклонения, скользящие разности (разность между первым и последним элементами окна, разность между максимальным и минимальным значениями окна).

Мы подробно остановимся на скользящих средних абсолютных отклонениях, скользящих медианных абсолютных отклонениях, скользящих разностях, поскольку эти статистики, основанные на разностях, могут быть особенно полезны, когда для прогнозирования временных рядов используется случайный лес или градиентный бустинг.

Давайте напишем функцию для вычисления скользящего среднего абсолютного отклонения и применим ее. Для этого вспомним формулу среднего

$$MAD = \frac{1}{n} \sum_{i=1}^n |x_i - \bar{X}|.$$

```
# пишем функцию для вычисления среднего абсолютного отклонения
def mad(data, axis=None):
    return np.nanmean(np.abs(data - np.nanmean(data, axis)),
                      axis)
```

Теперь в датафрейме `data` мы удалим ранее созданные скользящие средние и создадим столбец `rolling_mad4` – скользящие средние абсолютные отклонения с шириной окна 4 и `min_periods=1`, при этом скользящие средние абсолютные отклонения опять считаем с лагом 1.

```
# удалим переменные по паттерну 'mean'
data.drop(data.filter(regex='mean').columns,
           axis=1, inplace=True)
# считаем скользящие средние абсолютные отклонения,
# окно шириной 4, с лагом 1
data['rolling_mad4'] = data['sales'].shift(1).rolling(
    window=4, min_periods=1).apply(mad)
data
```

	sales	rolling_mad4
2018-01-09	2400.0	NaN
2018-01-10	2800.0	0.000000
2018-01-11	2500.0	200.000000
2018-01-12	2890.0	155.555556
2018-01-13	2610.0	197.500000
2018-01-14	2500.0	145.000000
2018-01-15	2750.0	132.500000
2018-01-16	2700.0	132.500000
2018-01-17	NaN	85.000000
2018-01-18	NaN	100.000000
2018-01-19	NaN	25.000000
2018-01-20	NaN	0.000000

Теперь вычислим эти скользящие средние абсолютные отклонения вручную.

```
# вручную считаем скользящие средние абсолютные отклонения,  
# окно шириной 4, с лагом 1  
data['rolling_mad4_manually'] = np.NaN  
  
mean_value = np.nanmean(2400)  
data.iloc[1, 2] = np.nanmean(np.abs(2400 - mean_value))  
  
mean_value = np.nanmean([2400, 2800])  
data.iloc[2, 2] = np.nanmean([np.abs(2400 - mean_value),  
                           np.abs(2800 - mean_value)])  
  
mean_value = np.nanmean([2400, 2800, 2500])  
data.iloc[3, 2] = np.nanmean([np.abs(2400 - mean_value),  
                           np.abs(2800 - mean_value),  
                           np.abs(2500 - mean_value)])  
  
mean_value = np.nanmean([2400, 2800, 2500, 2890])  
data.iloc[4, 2] = np.nanmean([np.abs(2400 - mean_value),  
                           np.abs(2800 - mean_value),  
                           np.abs(2500 - mean_value),  
                           np.abs(2890 - mean_value)])  
  
mean_value = np.nanmean([2800, 2500, 2890, 2610])  
data.iloc[5, 2] = np.nanmean([np.abs(2800 - mean_value),  
                           np.abs(2500 - mean_value),  
                           np.abs(2890 - mean_value),  
                           np.abs(2610 - mean_value)])  
  
mean_value = np.nanmean([2500, 2890, 2610, 2500])  
data.iloc[6, 2] = np.nanmean([np.abs(2500 - mean_value),  
                           np.abs(2890 - mean_value),  
                           np.abs(2610 - mean_value),  
                           np.abs(2500 - mean_value)])  
  
mean_value = np.nanmean([2890, 2610, 2500, 2750])  
data.iloc[7, 2] = np.nanmean([np.abs(2890 - mean_value),  
                           np.abs(2610 - mean_value),  
                           np.abs(2500 - mean_value),  
                           np.abs(2750 - mean_value)])  
  
mean_value = np.nanmean([2610, 2500, 2750, 2700])  
data.iloc[8, 2] = np.nanmean([np.abs(2700 - mean_value),  
                           np.abs(2750 - mean_value),  
                           np.abs(2500 - mean_value),  
                           np.abs(2610 - mean_value)])  
  
mean_value = np.nanmean([2500, 2750, 2700, np.nan])  
data.iloc[9, 2] = np.nanmean([np.abs(2700 - mean_value),  
                           np.abs(2750 - mean_value),  
                           np.abs(2500 - mean_value)])  
  
mean_value = np.nanmean([2750, 2700, np.nan, np.nan])  
data.iloc[10, 2] = np.nanmean([np.abs(2700 - mean_value),  
                           np.abs(2750 - mean_value)])
```

```
mean_value = np.nanmean([2700, np.nan, np.nan, np.nan])
data.iloc[11, 2] = np.nanmean(np.abs(2700 - mean_value))
data
```

	sales	rolling_mad4	rolling_mad4_manually
2018-01-09	2400.0	NaN	NaN
2018-01-10	2800.0	0.000000	0.000000
2018-01-11	2500.0	200.000000	200.000000
2018-01-12	2890.0	155.555556	155.555556
2018-01-13	2610.0	197.500000	197.500000
2018-01-14	2500.0	145.000000	145.000000
2018-01-15	2750.0	132.500000	132.500000
2018-01-16	2700.0	132.500000	132.500000
2018-01-17	NaN	85.000000	85.000000
2018-01-18	NaN	100.000000	100.000000
2018-01-19	NaN	25.000000	25.000000
2018-01-20	NaN	0.000000	0.000000

Теперь напишем функцию для вычисления скользящего медианного абсолютного отклонения и применим ее. Для этого вспомним формулу медианного абсолютного отклонения $MdAD = \text{median}(|x_i - \text{median}(X)|)$.

```
# пишем функцию для вычисления медианного абсолютного отклонения
def mdad(data, axis=None):
    return np.nanmedian(np.abs(data - np.nanmedian(data, axis)),
                        axis)
```

Создадим столбец *rolling_mdad4* – скользящие медианные абсолютные отклонения с шириной окна 4 и *min_periods*=1, при этом скользящие медианные абсолютные отклонения считаем с лагом 1.

```
# считаем скользящие медианные абсолютные отклонения,
# окно шириной 4, с лагом 1
data['rolling_mdad4'] = data['sales'].shift(1).rolling(
    window=4, min_periods=1).apply(mdad)
data
```

	sales	rolling_mad4	rolling_mad4_manually	rolling_mdad4
2018-01-09	2400.0	NaN	NaN	NaN
2018-01-10	2800.0	0.000000	0.000000	0.0
2018-01-11	2500.0	200.000000	200.000000	200.0
2018-01-12	2890.0	155.555556	155.555556	100.0
2018-01-13	2610.0	197.500000	197.500000	195.0
2018-01-14	2500.0	145.000000	145.000000	140.0
2018-01-15	2750.0	132.500000	132.500000	55.0
2018-01-16	2700.0	132.500000	132.500000	125.0
2018-01-17	NaN	85.000000	85.000000	70.0
2018-01-18	NaN	100.000000	100.000000	50.0
2018-01-19	NaN	25.000000	25.000000	25.0
2018-01-20	NaN	0.000000	0.000000	0.0

Теперь вычислим эти скользящие медианные абсолютные отклонения вручную.

```

median_value = np.nanmedian([2890, 2610, 2500, 2750])
data.iloc[7, 4] = np.nanmedian([np.abs(2890 - median_value),
                               np.abs(2610 - median_value),
                               np.abs(2500 - median_value),
                               np.abs(2750 - median_value)]))

median_value = np.nanmedian([2610, 2500, 2750, 2700])
data.iloc[8, 4] = np.nanmedian([np.abs(2700 - median_value),
                               np.abs(2750 - median_value),
                               np.abs(2500 - median_value),
                               np.abs(2610 - median_value)]))

median_value = np.nanmedian([2500, 2750, 2700, np.nan])
data.iloc[9, 4] = np.nanmedian([np.abs(2700 - median_value),
                               np.abs(2750 - median_value),
                               np.abs(2500 - median_value)]))

median_value = np.nanmedian([2750, 2700, np.nan, np.nan])
data.iloc[10, 4] = np.nanmedian([np.abs(2700 - median_value),
                               np.abs(2750 - median_value)]))

median_value = np.nanmedian([2700, np.nan, np.nan, np.nan])
data.iloc[11, 4] = np.nanmedian(np.abs(2700 - median_value))
data

```

	sales	rolling_mad4	rolling_mad4_manually	rolling_mdad4	rolling_mdad4_manually
2018-01-09	2400.0	NaN	NaN	NaN	NaN
2018-01-10	2800.0	0.000000	0.000000	0.0	0.0
2018-01-11	2500.0	200.000000	200.000000	200.0	200.0
2018-01-12	2890.0	155.555556	155.555556	100.0	100.0
2018-01-13	2610.0	197.500000	197.500000	195.0	195.0
2018-01-14	2500.0	145.000000	145.000000	140.0	140.0
2018-01-15	2750.0	132.500000	132.500000	55.0	55.0
2018-01-16	2700.0	132.500000	132.500000	125.0	125.0
2018-01-17	NaN	85.000000	85.000000	70.0	70.0
2018-01-18	NaN	100.000000	100.000000	50.0	50.0
2018-01-19	NaN	25.000000	25.000000	25.0	25.0
2018-01-20	NaN	0.000000	0.000000	0.0	0.0

Вычислим скользящие разности между максимальным и минимальным значениями с шириной окна 4 и `min_periods=1`, считаем их с лагом 1.

```

# считаем скользящие разности между максимальным
# и минимальным значениями окна, окно шириной 4,
# с лагом 1
data['rolling_diff4_max_min'] = data['sales'].shift(periods=1).rolling(
    window=4, min_periods=1).apply(lambda x: max(x) - min(x))
data

```

	sales	rolling_mad4	rolling_mad4_manually	rolling_mdad4	rolling_mdad4_manually	rolling_diff4_max_min
2018-01-09	2400.0	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800.0	0.000000	0.000000	0.0	0.0	NaN
2018-01-11	2500.0	200.000000	200.000000	200.0	200.0	NaN
2018-01-12	2890.0	155.555556	155.555556	100.0	100.0	NaN
2018-01-13	2610.0	197.500000	197.500000	195.0	195.0	490.0
2018-01-14	2500.0	145.000000	145.000000	140.0	140.0	390.0
2018-01-15	2750.0	132.500000	132.500000	55.0	55.0	390.0
2018-01-16	2700.0	132.500000	132.500000	125.0	125.0	390.0
2018-01-17	NaN	85.000000	85.000000	70.0	70.0	250.0
2018-01-18	NaN	100.000000	100.000000	50.0	50.0	250.0
2018-01-19	NaN	25.000000	25.000000	25.0	25.0	50.0
2018-01-20	NaN	0.000000	0.000000	0.0	0.0	0.0

Теперь вычислим скользящие разности между максимальным и минимальным значениями вручную.

```
# вручную считаем скользящие разности между максимальным и
# минимальным значениями окна, окно шириной 4, с лагом 1
data['rolling_diff4_max_min_manually'] = np.NaN
data.iloc[4, 6] = (np.nanmax([2400, 2800, 2500, 2890]) -
                   np.nanmin([2400, 2800, 2500, 2890]))
data.iloc[5, 6] = (np.nanmax([2800, 2500, 2890, 2610]) -
                   np.nanmin([2800, 2500, 2890, 2610]))
data.iloc[6, 6] = (np.nanmax([2500, 2890, 2610, 2500]) -
                   np.nanmin([2500, 2890, 2610, 2500]))
data.iloc[7, 6] = (np.nanmax([2890, 2610, 2500, 2750]) -
                   np.nanmin([2890, 2610, 2500, 2750]))
data.iloc[8, 6] = (np.nanmax([2610, 2500, 2750, 2700]) -
                   np.nanmin([2610, 2500, 2750, 2700]))
data.iloc[9, 6] = (np.nanmax([2500, 2750, 2700, np.nan]) -
                   np.nanmin([2500, 2750, 2700, np.nan]))
data.iloc[10, 6] = (np.nanmax([2750, 2700, np.nan, np.nan]) -
                   np.nanmin([2750, 2700, np.nan, np.nan]))
data.iloc[11, 6] = (np.nanmax([2700, np.nan, np.nan, np.nan]) -
                   np.nanmin([2700, np.nan, np.nan, np.nan]))
data
```

	sales	rolling_mad4	rolling_mad4_manually	rolling_mdad4	rolling_mdad4_manually	rolling_diff4_max_min	rolling_diff4_max_min_manually
2018-01-09	2400.0	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800.0	0.000000	0.000000	0.0	0.0	NaN	NaN
2018-01-11	2500.0	200.000000	200.000000	200.0	200.0	NaN	NaN
2018-01-12	2890.0	155.555556	155.555556	100.0	100.0	NaN	NaN
2018-01-13	2610.0	197.500000	197.500000	195.0	195.0	490.0	490.0
2018-01-14	2500.0	145.000000	145.000000	140.0	140.0	390.0	390.0
2018-01-15	2750.0	132.500000	132.500000	55.0	55.0	390.0	390.0
2018-01-16	2700.0	132.500000	132.500000	125.0	125.0	390.0	390.0
2018-01-17	NaN	85.000000	85.000000	70.0	70.0	250.0	250.0
2018-01-18	NaN	100.000000	100.000000	50.0	50.0	250.0	250.0
2018-01-19	NaN	25.000000	25.000000	25.0	25.0	50.0	50.0
2018-01-20	NaN	0.000000	0.000000	0.0	0.0	0.0	0.0

Вычислим скользящие разности между последним и первым значениями с шириной окна 4 и `min_periods=1`, считаем их с лагом 1.

```
# считаем скользящие разности между последним и
# первым значениями окна, окно шириной 4, с лагом 1
data['rolling_diff4_last_frst'] = data['sales'].shift(periods=1).rolling(
    window=4, min_periods=1).apply(lambda x: x.iloc[-1] - x.iloc[0])
data
```

	sales	rolling_mad4	rolling_mad4_manually	rolling_mdad4	rolling_mdad4_manually	rolling_diff4_max_min	rolling_diff4_max_min_manually	rolling_diff4_last_frst
18-09	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
18-10	2800.0	0.000000	0.000000	0.0	0.0	NaN	NaN	NaN
18-11	2500.0	200.000000	200.000000	200.0	200.0	NaN	NaN	NaN
18-12	2890.0	155.555556	155.555556	100.0	100.0	NaN	NaN	NaN
18-13	2610.0	197.500000	197.500000	195.0	195.0	490.0	490.0	490.0
18-14	2500.0	145.000000	145.000000	140.0	140.0	390.0	390.0	-190.0
18-15	2750.0	132.500000	132.500000	55.0	55.0	390.0	390.0	0.0
18-16	2700.0	132.500000	132.500000	125.0	125.0	390.0	390.0	-140.0
18-17	NaN	85.000000	85.000000	70.0	70.0	250.0	250.0	90.0
18-18	NaN	100.000000	100.000000	50.0	50.0	250.0	250.0	NaN
18-19	NaN	25.000000	25.000000	25.0	25.0	50.0	50.0	NaN
18-20	NaN	0.000000	0.000000	0.0	0.0	0.0	0.0	NaN

Теперь вычислим скользящие разности между последним и первым значениями вручную.

```
# вручную считаем скользящие разности между последним и
# первым значениями окна, окно шириной 4, с лагом 1
data['rolling_diff4_last_frst_manually'] = np.nan
data.iloc[4, 8] = ([2400, 2800, 2500, 2890].pop(-1) -
                   [2400, 2800, 2500, 2890].pop(0))
data.iloc[5, 8] = ([2800, 2500, 2890, 2610].pop(-1) -
                   [2800, 2500, 2890, 2610].pop(0))
data.iloc[6, 8] = ([2500, 2890, 2610, 2500].pop(-1) -
                   [2500, 2890, 2610, 2500].pop(0))
data.iloc[7, 8] = ([2890, 2610, 2500, 2750].pop(-1) -
                   [2890, 2610, 2500, 2750].pop(0))
data.iloc[8, 8] = ([2610, 2500, 2750, 2700].pop(-1) -
                   [2610, 2500, 2750, 2700].pop(0))
data.iloc[9, 8] = ([2500, 2750, 2700, np.nan].pop(-1) -
                   [2500, 2750, 2700, np.nan].pop(0))
data.iloc[10, 8] = ([2750, 2700, np.nan, np.nan].pop(-1) -
                   [2750, 2700, np.nan, np.nan].pop(0))
data.iloc[11, 8] = ([2700, np.nan, np.nan, np.nan].pop(-1) -
                   [2700, np.nan, np.nan, np.nan].pop(0))
data
```

<code>rolling_mdad4</code>	<code>rolling_mdad4_manually</code>	<code>rolling_diff4_max_min</code>	<code>rolling_diff4_max_min_manually</code>	<code>rolling_diff4_last_frst</code>	<code>rolling_diff4_last_frst_manually</code>
NaN	NaN	NaN	NaN	NaN	NaN
0.0	0.0	NaN	NaN	NaN	NaN
200.0	200.0	NaN	NaN	NaN	NaN
100.0	100.0	NaN	NaN	NaN	NaN
195.0	195.0	490.0	490.0	490.0	490.0
140.0	140.0	390.0	390.0	-190.0	-190.0
55.0	55.0	390.0	390.0	0.0	0.0
125.0	125.0	390.0	390.0	-140.0	-140.0
70.0	70.0	250.0	250.0	90.0	90.0
50.0	50.0	250.0	250.0	NaN	NaN
25.0	25.0	50.0	50.0	NaN	NaN
0.0	0.0	0.0	0.0	NaN	NaN

Вновь при вычислении скользящих средних/медианных абсолютных отклонений, разностей для более поздних моментов времени в тестовой части мы используем средние/медианные абсолютные отклонения, разности, взятые по более поздним наблюдениям в обучающей части, с уменьшением ширины окна. Уменьшение ширины окна позволяет сильнее учитывать более поздние наблюдения в обучающей части при вычислении скользящих средних/медианных абсолютных отклонений, разностей для более поздних моментов времени в тестовой части. При вычислении скользящих средних/медианных абсолютных отклонений, разностей для тестовой части мы ни разу не использовали значения продаж в тестовой части.

Последнее скользящее среднее/медианное абсолютное отклонение, последняя разность между максимальным и минимальным значениями в тестовой части всегда будет равна 0, обычно нулевое значение заменяют последним скользящим средним/медианным абсолютным отклонением, последней разностью между максимальным и минимальным значениями в обучающей части. Разности между последним и первым значениями в тестовой части получат пропуски, обычно их заменяют нулями.

Все скользящие статистики мы вычислили в исходном наборе, рассматривая последние N наблюдений в качестве тестовой выборки (N – длина горизонта) и сформировав для них защиту в виде значений NaN. Однако можно было взять обучающую выборку и удлинить ее на длину горизонта прогнозирования, зависимая переменная в наблюдениях, соответствующих новым временными метками (т. е. в тестовой выборке / наборе новых данных), получает значения NaN. После вычисления статистик на удлиненном наборе происходит разбиение на обучающую и тестовую выборки и строится модель.

Для больших наборов данных скользящие статистики можно вычислить с помощью быстрых функций NumPy, написанных на языке С. Их можно найти в модуле move пакета bottleneck.

Функция	Описание
<code>move_mean()</code>	Среднее значение в окне
<code>move_std</code>	Стандартное отклонение в окне
<code>move_var()</code>	Дисперсия в окне
<code>move_min()</code>	Минимальное значение в окне
<code>move_max()</code>	Максимальное значение в окне
<code>move_median()</code>	Медиана в окне
<code>move_sum()</code>	Сумма в окне

Сейчас мы удалим ранее созданные скользящие статистики и вычислим скользящее среднее с шириной окна 4 с помощью функции `move_mean()`. Обратите внимание, что она работает немного иначе, чем функция `rolling().mean()`. С помощью параметра `window` можно задать ширину окна. Параметр `min_count` работает следующим образом: если количество непропущенных значений в окне меньше `min_count`, окну присваивается значение `NaN`. По умолчанию для параметра `min_count` установлено значение `None`, т. е. для параметра `min_count` устанавливается значение параметра `window`. Заодно посмотрим, как происходят вычисления скользящих средних под капотом.

```
# удалим переменные по логику 'rolling'
data.drop(data.filter(regex='rolling').columns,
           axis=1, inplace=True)

# вычисляем скользящее среднее шириной 4 с помощью функции
# move_mean() пакета bottleneck
data['rolling_mean4_bottleneck'] = bn.move_mean(
    data['sales'], window=4, min_count=1)

# под капотом вычисления осуществляются так
data['rolling_mean4_bottleneck_manually'] = np.NaN
data.iloc[0, 2] = np.nanmean([2400])
data.iloc[1, 2] = np.nanmean([2400, 2800])
data.iloc[2, 2] = np.nanmean([2400, 2800, 2500])
data.iloc[3, 2] = np.nanmean([2400, 2800, 2500, 2890])
data.iloc[4, 2] = np.nanmean([2800, 2500, 2890, 2610])
data.iloc[5, 2] = np.nanmean([2500, 2890, 2610, 2500])
data.iloc[6, 2] = np.nanmean([2890, 2610, 2500, 2750])
data.iloc[7, 2] = np.nanmean([2610, 2500, 2750, 2700])
data.iloc[8, 2] = np.nanmean([2500, 2750, 2700, np.nan])
data.iloc[9, 2] = np.nanmean([2750, 2700, np.nan, np.nan])
data.iloc[10, 2] = np.nanmean([2700, np.nan, np.nan])
data.iloc[11, 2] = np.nanmean([np.nan, np.nan, np.nan])
data
```

	<code>sales</code>	<code>rolling_mean4_bottleneck</code>	<code>rolling_mean4_bottleneck_manually</code>
2018-01-09	2400.0	2400.000000	2400.000000
2018-01-10	2800.0	2600.000000	2600.000000
2018-01-11	2500.0	2566.666667	2566.666667
2018-01-12	2890.0	2647.500000	2647.500000
2018-01-13	2610.0	2700.000000	2700.000000
2018-01-14	2500.0	2625.000000	2625.000000
2018-01-15	2750.0	2687.500000	2687.500000
2018-01-16	2700.0	2640.000000	2640.000000
2018-01-17	NaN	2650.000000	2650.000000
2018-01-18	NaN	2725.000000	2725.000000
2018-01-19	NaN	2700.000000	2700.000000
2018-01-20	NaN	NaN	NaN

Чтобы функция `move_mean()` работала так же, как и функция `rolling().mean()`, применяем ее с лагом, равным значению `min_count`. Параметры `periods` и `min_count` будут аналогами параметров `periods` и `min_periods` конструкции `shift(periods=1) + rolling(min_periods=1).mean()`.

```
# вычисляем скользящее среднее шириной 4 с лагом 1
# при помощи функции move_mean() пакета bottleneck
data['rolling_mean4_btl_sh'] = data[['sales']].shift(
    periods=1).apply(lambda x: bn.move_mean(
        x, window=4, min_count=1))
# для сравнения вычислим скользящее среднее с шириной
# окна 4, лагом 1 и минимальным количеством наблюдений
# для вычисления статистики 1
data['rolling_mean4'] = data[['sales']].shift(1).rolling(
    min_periods=1, window=4).mean()
data
```

	<code>sales</code>	<code>rolling_mean4_bottleneck</code>	<code>rolling_mean4_bottleneck_manually</code>	<code>rolling_mean4_btl_sh</code>	<code>rolling_mean4</code>
2018-01-09	2400.0	2400.000000	2400.000000	NaN	NaN
2018-01-10	2800.0	2600.000000	2600.000000	2400.000000	2400.000000
2018-01-11	2500.0	2566.666667	2566.666667	2600.000000	2600.000000
2018-01-12	2890.0	2647.500000	2647.500000	2566.666667	2566.666667
2018-01-13	2610.0	2700.000000	2700.000000	2647.500000	2647.500000
2018-01-14	2500.0	2625.000000	2625.000000	2700.000000	2700.000000
2018-01-15	2750.0	2687.500000	2687.500000	2625.000000	2625.000000
2018-01-16	2700.0	2640.000000	2640.000000	2687.500000	2687.500000
2018-01-17	NaN	2650.000000	2650.000000	2640.000000	2640.000000
2018-01-18	NaN	2725.000000	2725.000000	2650.000000	2650.000000
2018-01-19	NaN	2700.000000	2700.000000	2725.000000	2725.000000
2018-01-20	NaN	NaN	NaN	2700.000000	2700.000000

Мы можем написать удобную функцию `moving_stats()` для вычисления различных скользящих статистик разными способами. Здесь мы можем дополнительно выбрать веса, коэффициенты сезонности для вычисления скользящих статистик.

```
# функция для создания скользящих статистик
def moving_stats(series, alpha=1, seasonality=1,
                  periods=1, min_periods=1, window=4,
                  aggfunc='mean', fillna=None):
    """
    Создает скользящие статистики.

Параметры
-----
alpha: int, по умолчанию 1
    Коэффициент авторегрессии.
seasonality: int, по умолчанию 1
    Коэффициент сезонности.
periods: int, по умолчанию 1
    Порядок лага, с которым вычисляем скользящие
    статистики.
min_periods: int, по умолчанию 1
    Минимальное количество наблюдений в окне для
    вычисления скользящих статистик.
window: int, по умолчанию 4
    Ширина окна. Не должна быть меньше
    горизонта прогнозирования.
aggfunc: string, по умолчанию 'mean'
    Агрегирующая функция.
fillna: int, по умолчанию None
    Стратегия импутации пропусков.
    """

# задаем размер окна для определения
# диапазона коэффициентов
size = window if window != -1 else len(series) - 1
# задаем диапазон значений коэффициентов в виде списка
alpha_range = [alpha ** i for i in range(0, size)]
# задаем минимально требуемое количество наблюдений
# в окне для вычисления скользящего среднего с
# поправкой на сезонность
min_required_len = max(
    min_periods - 1, 0) * seasonality + 1

# функция для получения лагов в соответствии
# с заданной сезонностью
def get_required_lags(series):
    # возвращает вычисленные лаги в соответствии
    # с заданной сезонностью
    return pd.Series(series.values[::-1]
                     [:seasonality])

# функция для вычисления скользящей статистики
def aggregate_window(series):
    tmp_series = get_required_lags(series)
    size = len(tmp_series)
```

```

tmp = tmp_series * alpha_range[-size:]

if aggfunc == 'median':
    return tmp.to_frame().agg(
        lambda x: np.nanmedian(
            np.abs(x - np.nanmedian(x))))
else:
    return tmp.agg(aggfunc)

# собственно вычисление скользящих статистик
features = series.shift(periods=periods).rolling(
    window=seasonality * window
    if window != -1 else len(series) - 1,
    min_periods=min_required_len).aggregate(
        aggregate_window)

# имплементируем пропуски
if fillna is not None:
    features.fillna(fillna, inplace=True)
return features

```

Давайте удалим ранее созданные скользящие средние и вычислим с помощью функции `moving_stats()` скользящее среднее с шириной окна 4 обоими способами – с лагом, равным горизонту прогнозирования, и лагом 1.

```

# удалим переменные по паттерну 'rolling_mean'
data.drop(data.filter(regex='rolling_mean').columns,
           axis=1, inplace=True)
# вычислим скользящее среднее с шириной окна 4 обоими способами:
# 1 - с шириной окна 4 и лагом 4
# 2 - с шириной окна 4 и лагом 1
data['rolling_mean4_frst_method'] = moving_stats(
    data['sales'], window=4, periods=4)
data['rolling_mean4_scnd_method'] = moving_stats(
    data['sales'], window=4, periods=1, min_periods=1)
data

```

	sales	rolling_mean4_frst_method	rolling_mean4_scnd_method
2018-01-09	2400.0	NaN	NaN
2018-01-10	2800.0	NaN	2400.000000
2018-01-11	2500.0	NaN	2600.000000
2018-01-12	2890.0	NaN	2566.666667
2018-01-13	2610.0	2400.000000	2647.500000
2018-01-14	2500.0	2600.000000	2700.000000
2018-01-15	2750.0	2566.666667	2625.000000
2018-01-16	2700.0	2647.500000	2687.500000
2018-01-17	NaN	2700.000000	2640.000000
2018-01-18	NaN	2625.000000	2650.000000
2018-01-19	NaN	2687.500000	2725.000000
2018-01-20	NaN	2640.000000	2700.000000

Начинающего пользователя может сбить с толку такое большое количество параметров, поэтому можно обойтись более простым вариантом функции. Ниже приведен пример более простой функции для вычисления скользящих статистик. В ней реализован обычный режим вычислений и быстрый режим вычислений (на основе функций библиотеки bottleneck).

```
# более простая (и быстрая) функция
# для создания скользящих статистик
def fast_moving_stats(series, periods=1, min_count=1,
                      window=4, fast=True, fillna=None,
                      aggfunc='mean'):
    """
    Создает скользящие статистики.

    Параметры
    -----
    series:
        pandas.Series
    periods: int, по умолчанию 1
        Порядок лага, с которым вычисляем скользящие
        статистики.
    min_count: int, по умолчанию 1
        Минимальное количество наблюдений в окне для
        вычисления скользящих статистик.
    window: int, по умолчанию 4
        Ширина окна. Не должна быть меньше
        горизонта прогнозирования.
    fast: bool, по умолчанию None
        Режим вычислений скользящих статистик.
    fillna: int, по умолчанию None
        Стратегия импутации пропусков.
    aggfunc: string, по умолчанию 'mean'
        Стратегия импутации пропусков.
    """

    if fast:
        def shift(xs, n):
            return np.concatenate((np.full(n, np.nan),
                                  xs[:-n]))

        arr = series.values
        arr = shift(xs=arr, n=periods)
        if aggfunc == 'mean':
            arr = bn.move_mean(
                arr, window=window, min_count=min_count)
        if aggfunc == 'std':
            arr = bn.move_std(
                arr, window=window, min_count=min_count)
        if aggfunc == 'sum':
            arr = bn.move_sum(
                arr, window=window, min_count=min_count)
        if aggfunc == 'median':
            arr = bn.move_median(
                arr, window=window, min_count=min_count)

        features = pd.Series(arr)
        features.index = series.index
    else:
```

```

if aggfunc == 'mean':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).mean()
if aggfunc == 'std':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).std()
if aggfunc == 'sum':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).sum()
if aggfunc == 'median':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).median()

# имплементируем пропуски
if fillna is not None:
    features.fillna(fillna, inplace=True)

return features

```

Давайте вновь вычислим скользящее среднее с шириной окна 4 обоими способами, используя уже функцию `fast_moving_stats()`.

```

# вычислим скользящее среднее обоими способами:
# 1 - с шириной окна 4 и лагом 1
# 2 - с шириной окна 4 и лагом 4
data['roll_mean4_frst_method_fast'] = fast_moving_stats(
    data['sales'], window=4, periods=4,
    fast=True, fillna=0, aggfunc='mean')
data['roll_mean4_scnd_method_fast'] = fast_moving_stats(
    data['sales'], window=4, periods=1, min_count=1,
    fast=True, fillna=0, aggfunc='mean')
data

```

	sales	rolling_mean4_frst_method	rolling_mean4_scnd_method	roll_mean4_frst_method_fast	roll_mean4_scnd_method_fast
2018-01-09	2400.0	NaN	NaN	0.000000	0.000000
2018-01-10	2800.0	NaN	2400.000000	0.000000	2400.000000
2018-01-11	2500.0	NaN	2600.000000	0.000000	2600.000000
2018-01-12	2890.0	NaN	2566.666667	0.000000	2566.666667
2018-01-13	2610.0	2400.000000	2647.500000	2400.000000	2647.500000
2018-01-14	2500.0	2600.000000	2700.000000	2600.000000	2700.000000
2018-01-15	2750.0	2566.666667	2625.000000	2566.666667	2625.000000
2018-01-16	2700.0	2647.500000	2687.500000	2647.500000	2687.500000
2018-01-17	NaN	2700.000000	2640.000000	2700.000000	2640.000000
2018-01-18	NaN	2625.000000	2650.000000	2625.000000	2650.000000
2018-01-19	NaN	2687.500000	2725.000000	2687.500000	2725.000000
2018-01-20	NaN	2640.000000	2700.000000	2640.000000	2700.000000

Теперь создадим серию со 100 000 000 значений, сравним обычный и быстрый режимы вычислений.

```
# создаем серию со 100 000 000 значений
series = pd.Series(np.random.randn(100000000))

%%time
roll_mean = fast_moving_stats(series, window=4,
                               min_count=1, fast=False)
```

CPU times: user 4.68 s, sys: 2.6 s, total: 7.27 s
Wall time: 7.29 s

```
%%time
roll_mean = fast_moving_stats(series, window=4,
                               min_count=1, fast=True)
```

CPU times: user 530 ms, sys: 386 ms, total: 916 ms
Wall time: 917 ms

Библиотека Polars также позволяет вычислить различные скользящие статистики.

Метод	Описание
.rolling_mean()	Среднее значение в окне
.rolling_std()	Стандартное отклонение в окне
.rolling_var()	Дисперсия в окне
.rolling_min()	Минимальное значение в окне
.rolling_max()	Максимальное значение в окне
.rolling_quantile()	Оценка для заданного процентиля / выборочного квантиля в окне
.rolling_median()	Медиана в окне
.rolling_sum()	Сумма в окне
.rolling_apply()	Применение пользовательской функции к значениям окна
.rolling_skew()	Коэффициент асимметрии в окне

Давайте с помощью метода `.rolling_mean()` вычислим скользящее среднее с шириной окна 4 обоими способами в Polars, предварительно избавившись от ранее созданных переменных. Ширина окна задается с помощью параметра `window_size`, а минимальное количество наблюдений, требуемое для вычисления статистики, – с помощью параметра `min_periods`.

```
# избавимся от ранее созданных переменных
polars_data = polars_data.select(['date', 'sales'])
```

```
# вычислим скользящее среднее обоими способами:  
# 1 - с шириной окна 4 и лагом 4  
# 2 - с шириной окна 4 и лагом 1  
polars_data = polars_data.with_columns([  
    pl.col('sales').shift(4).rolling_mean(  
        window_size=4, min_periods=1).alias(  
            'rolling_mean4_frst_method'),  
    pl.col('sales').shift(1).rolling_mean(  
        window_size=4, min_periods=1).alias(  
            'rolling_mean4_scnd_method')  
])  
polars_data
```

shape: (12, 4)

date	sales	rolling_mean4_frst_method	rolling_mean4_scnd_method
date	f64	f64	f64
2018-01-09	2400.0	null	null
2018-01-10	2800.0	null	2400.0
2018-01-11	2500.0	null	2600.0
2018-01-12	2890.0	null	2566.666667
2018-01-13	2610.0	2400.0	2647.5
2018-01-14	2500.0	2600.0	2700.0
2018-01-15	2750.0	2566.666667	2625.0
2018-01-16	2700.0	2647.5	2687.5
2018-01-17	null	2700.0	2640.0
2018-01-18	null	2625.0	2650.0
2018-01-19	null	2687.5	2725.0
2018-01-20	null	2640.0	2700.0

Вновь можно написать функцию для вычисления скользящих статистик в Polars.

```
# пишем функцию, вычисляющую скользящие  
# статистики в Polars  
def polars_moving_stats(pl_data,  
                        pl_series,  
                        periods=1,  
                        min_periods=1,  
                        window_size=4,  
                        out_column='Mov_stat',  
                        aggfunc='mean',  
                        strategy=None):  
    """  
    Вычисляет скользящие статистики в Polars.  
    Параметры  
    -----  
    pl_data:  
        Polars.DataFrame  
    pl_series:  
        Polars.Series  
    periods: int, по умолчанию 1  
    Порядок лага, с которым вычисляем скользящие статистики.
```

```

min_periods: int, по умолчанию 1
    Минимальное количество наблюдений в окне для
    вычисления скользящих статистик.
window_size: int, по умолчанию 4
    Ширина окна. Не должна быть меньше
    горизонта прогнозирования.
out_column: string, по умолчанию 'Mov_stat'
    Название результирующего столбца.
aggfunc: string, по умолчанию 'mean'
    Агрегирующая функция.
strategy: string, по умолчанию None
    Стратегия импутации пропусков.
"""

if aggfunc == 'mean':
    pl_data = pl_data.with_column(pl.col(pl_series).shift(
        periods=periods).rolling_mean(
        window_size=window_size,
        min_periods=min_periods).alias(out_column))

if aggfunc == 'min':
    pl_data = pl_data.with_column(pl.col(pl_series).shift(
        periods=periods).rolling_min(
        window_size=window_size,
        min_periods=min_periods).alias(out_column))

if aggfunc == 'max':
    pl_data = pl_data.with_column(pl.col(pl_series).shift(
        periods=periods).rolling_max(
        window_size=window_size,
        min_periods=min_periods).alias(out_column))

if aggfunc == 'median':
    pl_data = pl_data.with_column(pl.col(pl_series).shift(
        periods=periods).rolling_median(
        window_size=window_size,
        min_periods=min_periods).alias(out_column))

if aggfunc == 'sum':
    pl_data = pl_data.with_column(pl.col(pl_series).shift(
        periods=periods).rolling_sum(
        window_size=window_size,
        min_periods=min_periods).alias(out_column))

if aggfunc == 'std':
    pl_data = pl_data.with_column(pl.col(pl_series).shift(
        periods=periods).rolling_std(
        window_size=window_size,
        min_periods=min_periods).alias(out_column))

pl_data = pl_data.fill_null(strategy=strategy)
return pl_data

# вычислим скользящее среднее с шириной окна 4 и лагом 1
polars_data = polars_moving_stats(
    pl_data=polars_data,

```

```
pl_series='sales',
periods=1,
min_periods=1,
window_size=4,
out_column='sales_roll_mean4_scnd_method',
aggfunc='mean',
strategy='zero')
polars_data
```

shape: (12, 5)

date	sales	rolling_mean4_frst_method	rolling_mean4_scnd_method	sales_roll_mean4_scnd_method
date	f64	f64	f64	f64
2018-01-09	2400.0	0.0	0.0	0.0
2018-01-10	2800.0	0.0	2400.0	2400.0
2018-01-11	2500.0	0.0	2600.0	2600.0
2018-01-12	2890.0	0.0	2566.666667	2566.666667
2018-01-13	2610.0	2400.0	2647.5	2647.5
2018-01-14	2500.0	2600.0	2700.0	2700.0
2018-01-15	2750.0	2566.666667	2625.0	2625.0
2018-01-16	2700.0	2647.5	2687.5	2687.5
2018-01-17	0.0	2700.0	2640.0	2640.0
2018-01-18	0.0	2625.0	2650.0	2650.0
2018-01-19	0.0	2687.5	2725.0	2725.0
2018-01-20	0.0	2640.0	2700.0	2700.0

Завершая разговор про лаги, разности на лагах и скользящие статистики, отметим, что если вы применяете перекрестную проверку расширяющимся/скользящим окном без гэпа / с гэпом для оценки качества модели, то лаги, разности на лагах, скользящие статистики должны заново создаваться на каждой итерации перекрестной проверки. В каждой итерации для получения значений лагов, разностей и скользящих статистик в проверочной выборке используются только данные обучающей выборки. Поэтому в этом случае пишут собственный класс или собственную функцию перекрестной проверки на основе класса `TimeSeriesSplit`. Помимо того что такая функция заново создает лаги, разности и скользящие статистики на каждой итерации, она должна выводить подробную информацию о том, что происходит в каждой итерации перекрестной проверки: временные рамки обучающей и проверочной выборок, индексы наблюдений, попавших в обучающую и проверочную выборки, метрику качества и визуализацию прогнозов модели для проверочной выборки. Давайте напишем такую функцию.

Отключим экспоненциальное представление и напишем функцию перекрестной проверки расширяющимся/скользящим окном без гэпа / с гэпом с формированием лагов и скользящих статистик без протечек (формируем специальную защиту).

```
# отключаем экспоненциальное представление
pd.set_option('display.float_format',
               lambda x: '%.5f' % x)
```

```
# пишем функцию перекрестной проверки расширяющимся/
# скользящим окном без гэпа /с гэлом
# с формированием лагов и скользящих
# статистик без промежек
def timeseries_cv_with_lags_and_moving_stats(
    data, y_data, model, lags_range=None, moving_stats_range=None,
    aggfunc='mean', seasonality=1, print_cv_scheme=True,
    print_features=True, visualize=True, last_n_train=5,
    max_train_size=None, test_size=None, n_splits=3,
    gap=0, fillna=None):
    """
    Выполняет перекрестную проверку расширяющимся/
    скользящим окном без гэпа / с гэлом с
    формированием лагов и скользящих статистик.
    
```

Параметры

`data: pandas.DataFrame`

Массив признаков.

`y_data: pandas.Series`

Массив меток.

`model:`

Модель-регрессор: либо `CatBoostRegressor`, либо
класс-регрессор библиотеки `sklearn`.

`lags_range:`

Диапазон значений, значение – количество
периодов для лагов.

`moving_stats_range:`

Диапазон значений, значение – ширина окна
для скользящей статистики.

`aggfunc: string`, по умолчанию 'mean'

Агрегирующая функция для вычисления
скользящей статистики.

`seasonality: int`, по умолчанию 1

Коэффициент сезонности для скользящей статистики.

`print_cv_scheme: bool`, по умолчанию True

Печать схемы перекрестной проверки.

`print_features: bool`, по умолчанию True

Печать признаков.

`visualize: bool`, по умолчанию True

Визуализация прогнозов.

`last_n_train: int`, 5

Вывод n наблюдений обучающей выборки
при визуализации прогнозов.

`n_splits: int`, по умолчанию 3

Количество разбиений на обучающую и тестовую
выборки.

`max_train_size: int`, по умолчанию None

Максимальный размер обучающей выборки.

`test_size: int`, по умолчанию None

Максимальный размер тестовой выборки
(определяется горизонтом прогнозирования).

`gap: int`, по умолчанию 0

Размер гэпа.

`fillna: string`, по умолчанию None

Стратегия импутации пропусков
в обучающей выборке.

"""

```
if min(lags_range) < test_size:
    warnings.warn(
        "Количество периодов для лагов задавайте\n" +
        "равным или больше горизонта прогнозирования.\n")

if min(moving_stats_range) < test_size:
    warnings.warn(
        "Ширина окна для скользящих статистик задавайте\n" +
        "равной или больше горизонта прогнозирования.\n")

# создаем экземпляр класса TimeSeriesSplit
tscv = TimeSeriesSplit(
    max_train_size=max_train_size,
    test_size=test_size,
    n_splits=n_splits,
    gap=gap)

# создаем пустой список, куда записываем значения RMSE
rmse_lst = []

# запускаем цикл перекрестной проверки
for cnt, (train_index, test_index) in enumerate(
    tscv.split(data), 1):
    X_train, X_test = (data.iloc[train_index],
                        data.iloc[test_index])
    y_train, y_test = (y_data.iloc[train_index],
                       y_data.iloc[test_index])

    y_test_N = y_test.copy()

    # значения в тестовом массиве меняем NaN
    y_test_N[:] = np.NaN
    # конкatenируем обучающий и тестовый массивы меток
    tmp_target = pd.concat([y_train, y_test_N])

    # конкatenируем обучающий и тестовый
    # массивы признаков
    concat_data = pd.concat([X_train, X_test])

    # печатаем схему валидации и сконкатенированный
    # массив меток
    if print_cv_scheme:
        print("-----")
        print("TRAIN:",
              [X_train.index[0].strftime('%Y-%m-%d'),
               X_train.index[-1].strftime('%Y-%m-%d')]),
        print("TEST:",
              [X_test.index[0].strftime('%Y-%m-%d'),
               X_test.index[-1].strftime('%Y-%m-%d')])
        print("\nОбщее кол-во наблюдений: %d" % (
            len(X_train) + len(X_test)))
        print("Обучающий набор: %d" % (len(X_train)))
        print("Тестовый набор: %d" % (len(X_test)))

    if print_features:
        print(f"\nЗащита:\n{tmp_target}\n")
```

```

# формируем лаги, количество периодов должно быть
# равно или превышать значение горизонта
# прогнозирования (если меньше, получим
# значения NaN)
if lags_range is not None:
    for i in lags_range:
        concat_data[f'Lag_{i}'] = tmp_target.shift(i)

# формируем скользящие статистики, ширина окна
# должна быть равна или превышать значение
# горизонта прогнозирования (если меньше,
# получим значения NaN)
if moving_stats_range is not None:
    for i in moving_stats_range:
        concat_data[f'Moving_{aggfunc}_{i}'] = \
            moving_stats(tmp_target, window=i, aggfunc=aggfunc,
                         seasonality=seasonality)

# печатаем сконкатенированный массив признаков
# с новыми переменными - лагами и скользящими
# статистиками
if print_features:
    pattern = concat_data.columns.str.contains(
        'Lag|Moving_')
    feat = concat_data.columns[pattern]
    print(f"Добав. признаки:\n{concat_data[feat]}\n")

# сортируем столбцы для воспроизведимости
# (для CatBoost порядок генерации признаков
# влияет на результат)
concat_data = concat_data.sort_index(axis=1)

# снова выделяем обучающий и тестовый
# массивы признаков
X_train = concat_data[:-test_size]
X_test = concat_data[-test_size:]

# заполняем пропуски в обучающей выборке
if fillna == 'zero':
    X_train = X_train.fillna(0, axis=0)
if fillna == 'mean':
    X_train = X_train.fillna(X_train.mean(), axis=0)

# если модель - CatBoostRegressor
if model.__class__.__name__ == 'CatBoostRegressor':
    # создаем массив индексов категориальных
    # признаков для CatBoost
    categorical_features_indices = np.where(
        X_train.dtypes == object)[0]
    # формируем обучающий пул
    train_pool = Pool(
        X_train, y_train,
        cat_features=categorical_features_indices)
    # обучаем модель
    model.fit(train_pool)

```

```
# в противном случае
else:
    model.fit(X_train, y_train)

# получаем прогнозы
predictions = model.predict(X_test)
predictions = pd.Series(predictions)
predictions.index = X_test.index

# вычисляем RMSE на тестовой выборке
# в текущей итерации
rmse = mean_squared_error(
    y_test, predictions, squared=False)
# добавим найденное в данной итерации
# значение RMSE в список
rmse_lst.append(rmse)

print(f"\nRMSE={rmse:.3f} на {cnt}-й итерации\n")

# визуализируем прогнозы
if visualize:
    # задаем размер графика
    plt.figure(figsize=(8, 4))
    # настраиваем ориентацию меток оси x
    plt.xticks(rotation=90)
    # строим графики для обучающих данных,
    # тестовых данных, прогнозов модели
    plt.plot(y_train.iloc[-last_n_train:], 
              label='обучающая выборка')
    plt.plot(predictions,
              color='red',
              label='прогнозы')
    plt.plot(y_test,
              color='green',
              label='тестовая выборка')
    # задаем координатную сетку
    plt.grid()
    # задаем легенду
    plt.legend()
    plt.show()

# расчет среднего значения RMSE
rmse_mean = np.mean(rmse_lst)
# выведем среднее значение RMSE
print(f"Среднее значение RMSE={rmse_mean:.3f}")

if print_features:
    # печатаем список признаков (порядок генерации
    # признаков может повлиять на результат CatBoost)
    feat_lst = concat_data.columns.tolist()
    print(f"\nСписок признаков:\n{feat_lst}")
```

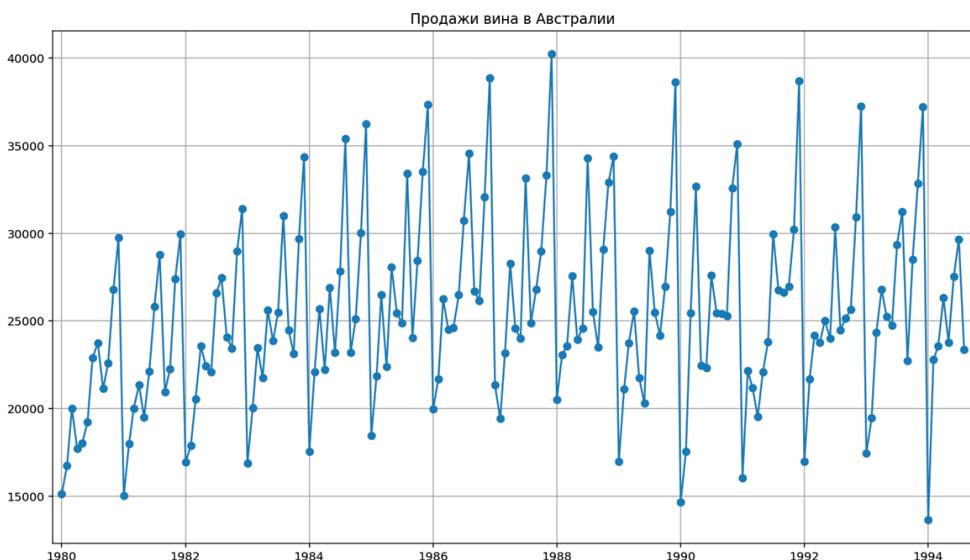
Давайте загрузим набор данных о ежемесячных продажах вина в Австралии с января 1980 года по август 1994 года включительно.

```
# загружаем набор о ежемесячных продажах вина в Австралии
wine_data = pd.read_csv('Data/monthly_australian_wine_sales.csv',
                       index_col=['month'],
                       parse_dates=['month'],
                       date_parser=lambda col: pd.to_datetime(
                           col, format='%Y-%m-%d'))
wine_data.head()
```

month	sales
1980-01-01	15136
1980-02-01	16733
1980-03-01	20016
1980-04-01	17708
1980-05-01	18019

Визуализируем временной ряд.

```
# визуализируем временной ряд
# задаем размер графика
fig, ax = plt.subplots(figsize=(14, 8))
# строим график
ax.plot(wine_data['sales'],
         marker='o')
# задаем заголовок графика
ax.set_title("Продажи вина в Австралии")
# задаем начало оси x с отступом
ax.margins(x=0.01)
# задаем координатную сетку
ax.grid();
```



Задача заключается в том, чтобы прогнозировать продажи вина на 6 месяцев вперед. Таким образом, горизонт прогнозирования составляет у нас 6 месяцев. Метрика качества – RMSE. Данные уже отсортированы по дате.

Теперь сформируем массив признаков и массив меток, создадим модель CatBoost.

```
# создаем массив меток и массив признаков
y_wine_data = wine_data.pop('sales')

# создаем модель CatBoost
ctbst_model = CatBoostRegressor(
    random_seed=42,
    n_estimators=600,
    learning_rate=0.05,
    loss_function='MAE',
    depth=9,
    logging_level='Silent')
```

Давайте применим перекрестную проверку расширяющимся окном с формированием лагов 6, 8 и 10 и скользящих средних с шириной окна 6 и 8 для продаж вина в Австралии.

```
# применяем функцию перекрестной проверки расширяющимся
# окном с формированием скользящих статистик и лагов
# на каждой итерации
timeseries_cv_with_lags_and_moving_stats(
    wine_data, y_wine_data, model=ctbst_model,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    print_features=True,
    aggfunc='mean', test_size=6, n_splits=3)
```

TRAIN: ['1980-01-01', '1993-02-01'] TEST: ['1993-03-01', '1993-08-01']

Общее кол-во наблюдений: 164

Обучающий набор: 158

Тестовый набор: 6

Защита:

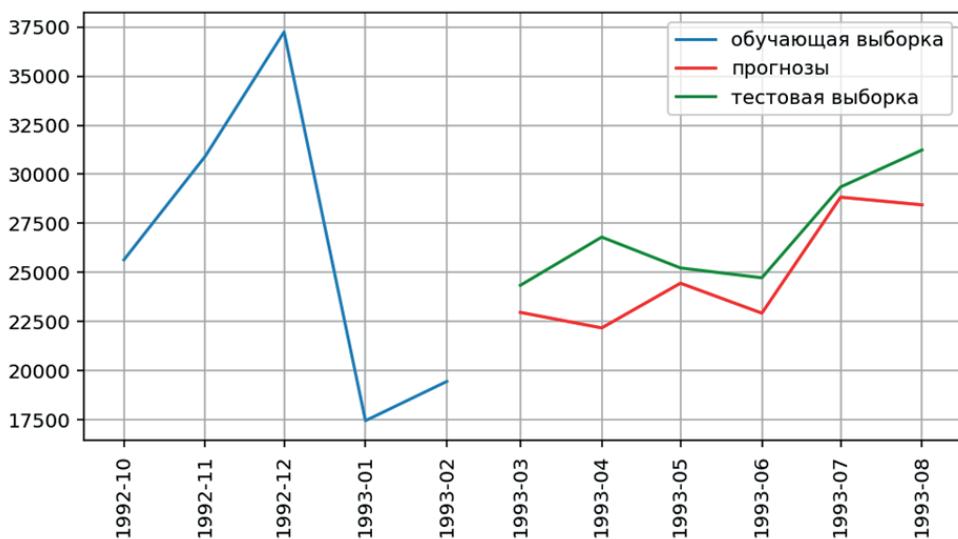
```
month
1980-01-01    15136.00000
1980-02-01    16733.00000
1980-03-01    20016.00000
1980-04-01    17708.00000
1980-05-01    18019.00000
...
1993-04-01      NaN
1993-05-01      NaN
1993-06-01      NaN
1993-07-01      NaN
1993-08-01      NaN
Name: sales, Length: 164, dtype: float64
```

Добав. признаки:

month	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
1980-01-01	NaN	NaN	NaN	NaN	NaN
1980-02-01	NaN	NaN	NaN	15136.00000	15136.00000
1980-03-01	NaN	NaN	NaN	15934.50000	15934.50000
1980-04-01	NaN	NaN	NaN	17295.00000	17295.00000
1980-05-01	NaN	NaN	NaN	17398.25000	17398.25000
...
1993-04-01	25650.00000	24488.00000	24019.00000	26148.40000	25769.42857
1993-05-01	30923.00000	25156.00000	30345.00000	26273.00000	25983.00000
1993-06-01	37240.00000	25650.00000	24488.00000	24723.00000	26148.40000
1993-07-01	17466.00000	30923.00000	25156.00000	18464.50000	26273.00000
1993-08-01	19463.00000	37240.00000	25650.00000	19463.00000	24723.00000

[164 rows x 5 columns]

RMSE=2419.324 на 1-й итерации



TRAIN: ['1980-01-01', '1993-08-01'] TEST: ['1993-09-01', '1994-02-01']

Общее кол-во наблюдений: 170

Обучающий набор: 164

Тестовый набор: 6

Зашита:

month	
1980-01-01	15136.00000
1980-02-01	16733.00000
1980-03-01	20016.00000
1980-04-01	17708.00000
1980-05-01	18019.00000
...	

```

1993-10-01      NaN
1993-11-01      NaN
1993-12-01      NaN
1994-01-01      NaN
1994-02-01      NaN
Name: sales, Length: 170, dtype: float64

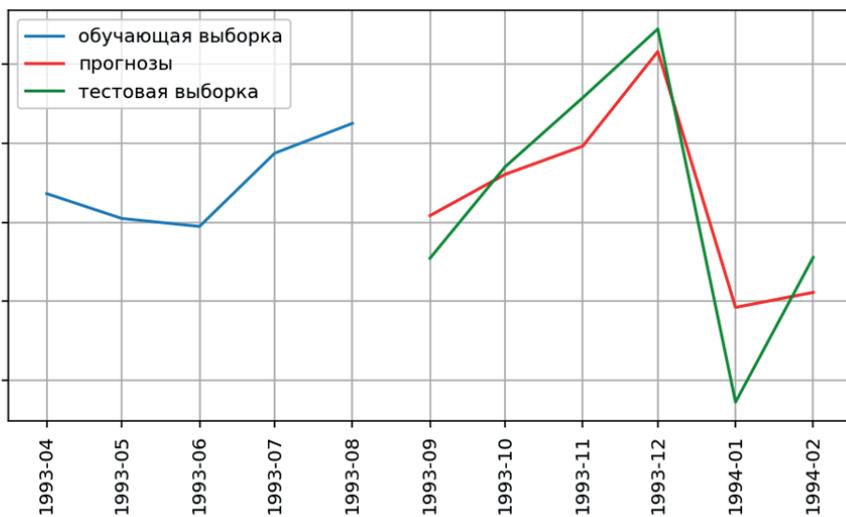
```

Добавленные признаки:

month	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
1980-01-01	NaN	NaN	NaN	NaN	NaN
1980-02-01	NaN	NaN	NaN	15136.00000	15136.00000
1980-03-01	NaN	NaN	NaN	15934.50000	15934.50000
1980-04-01	NaN	NaN	NaN	17295.00000	17295.00000
1980-05-01	NaN	NaN	NaN	17398.25000	17398.25000
...
1993-10-01	26805.00000	19463.00000	37240.00000	27473.20000	25883.00000
1993-11-01	25236.00000	24352.00000	17466.00000	27640.25000	26953.00000
1993-12-01	24735.00000	26805.00000	19463.00000	28441.66667	27473.20000
1994-01-01	29356.00000	25236.00000	24352.00000	30295.00000	27640.25000
1994-02-01	31234.00000	24735.00000	26805.00000	31234.00000	28441.66667

[170 rows x 5 columns]

RMSE=3148.286 на 2-й итерации



TRAIN: ['1980-01-01', '1994-02-01'] TEST: ['1994-03-01', '1994-08-01']

Общее кол-во наблюдений: 176

Обучающий набор: 170

Тестовый набор: 6

Задача:

month

```

1980-01-01 15136.00000
1980-02-01 16733.00000
1980-03-01 20016.00000
1980-04-01 17708.00000
1980-05-01 18019.00000
...
1994-04-01      NaN
1994-05-01      NaN
1994-06-01      NaN
1994-07-01      NaN
1994-08-01      NaN
Name: sales, Length: 176, dtype: float64

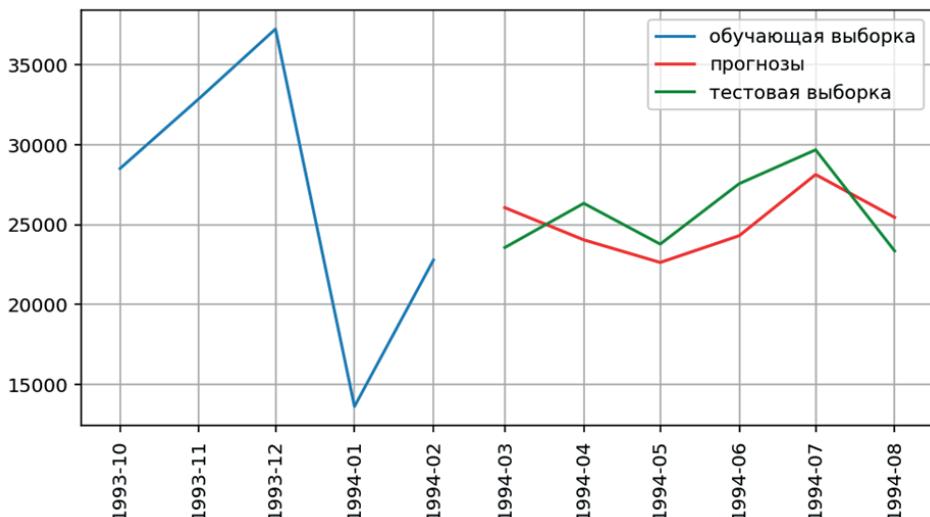
```

Добав. признаки:

month	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
1980-01-01	NaN	NaN	NaN	NaN	NaN
1980-02-01	NaN	NaN	NaN	15136.00000	15136.00000
1980-03-01	NaN	NaN	NaN	15934.50000	15934.50000
1980-04-01	NaN	NaN	NaN	17295.00000	17295.00000
1980-05-01	NaN	NaN	NaN	17398.25000	17398.25000
...
1994-04-01	28496.00000	31234.00000	24735.00000	26997.40000	26992.14286
1994-05-01	32857.00000	22724.00000	29356.00000	26622.75000	26285.16667
1994-06-01	37198.00000	28496.00000	31234.00000	24544.66667	26997.40000
1994-07-01	13652.00000	32857.00000	22724.00000	18218.00000	26622.75000
1994-08-01	22784.00000	37198.00000	28496.00000	22784.00000	24544.66667

[176 rows x 5 columns]

RMSE=2235.679 на 3-й итерации



Среднее значение RMSE=2601.096

Список признаков:

```
['Lag_10', 'Lag_6', 'Lag_8', 'Moving_mean_6', 'Moving_mean_8']
```

16.4.4. Взвешенные скользящие статистики

Кроме обычных скользящих статистик можно создавать взвешенные скользящие статистики, чтобы назначать более ранним/поздним наблюдениям в окне более низкие/высокие веса. Обычно используются две формулы, отличающиеся знаменателем (используем либо количество наблюдений в окне, либо сумму весов наблюдений в окне):

$$WMA = \frac{A_1 w_1 + A_2 w_2 + \dots + A_n w_n}{n},$$

или

$$WMA = \frac{A_1 w_1 + A_2 w_2 + \dots + A_n w_n}{\sum w_i}.$$

Заново загрузим *example.csv*.

```
# загружаем данные
data = load_data()
```

Теперь задаем массив весов длиной с ширину окна (в данном случае возьмем ширину окна 4) и вычисляем взвешенные скользящие средние с шириной окна 4, в знаменателе возьмем количество наблюдений в окне.

```
# задаем массив весов длиной с ширину окна
weights = np.array([0.7, 0.8, 0.9, 1])
# вычисляем взвешенные скользящие средние с шириной окна 4,
# в знаменателе - количество наблюдений в окне
data['weighted_rolling_mean4'] = data['sales'].rolling(
    window=4).apply(lambda x: np.mean(weights * x))
data
```

	sales	weighted_rolling_mean4
2018-01-09	2400	NaN
2018-01-10	2800	NaN
2018-01-11	2500	NaN
2018-01-12	2890	2265.00
2018-01-13	2610	2292.75
2018-01-14	2500	2227.75
2018-01-15	2750	2277.75
2018-01-16	2700	2250.50
2018-01-17	2250	2157.50
2018-01-18	2350	2115.00
2018-01-19	2550	2088.75
2018-01-20	3000	2187.50

Давайте вручную вычислим взвешенные скользящие средние с шириной окна 4, в знаменателе – количество наблюдений в окне.

```
# вручную вычисляем взвешенные скользящие средние
# с шириной окна 4, в знаменателе - количество
# наблюдений в окне
data['weighted_rolling_mean4_manually'] = np.NaN
data.iloc[3, 2] = (2400 * 0.7 + 2800 * 0.8 +
                  2500 * 0.9 + 2890 * 1) / 4
data.iloc[4, 2] = (2800 * 0.7 + 2500 * 0.8 +
                  2890 * 0.9 + 2610 * 1) / 4
data.iloc[5, 2] = (2500 * 0.7 + 2890 * 0.8 +
                  2610 * 0.9 + 2500 * 1) / 4
data.iloc[6, 2] = (2890 * 0.7 + 2610 * 0.8 +
                  2500 * 0.9 + 2750 * 1) / 4
data.iloc[7, 2] = (2610 * 0.7 + 2500 * 0.8 +
                  2750 * 0.9 + 2700 * 1) / 4
data.iloc[8, 2] = (2500 * 0.7 + 2750 * 0.8 +
                  2700 * 0.9 + 2250 * 1) / 4
data.iloc[9, 2] = (2750 * 0.7 + 2700 * 0.8 +
                  2250 * 0.9 + 2350 * 1) / 4
data.iloc[10, 2] = (2700 * 0.7 + 2250 * 0.8 +
                   2350 * 0.9 + 2550 * 1) / 4
data.iloc[11, 2] = (2250 * 0.7 + 2350 * 0.8 +
                   2550 * 0.9 + 3000 * 1) / 4
data
```

	sales	weighted_rolling_mean4	weighted_rolling_mean4_manually
2018-01-09	2400	NaN	NaN
2018-01-10	2800	NaN	NaN
2018-01-11	2500	NaN	NaN
2018-01-12	2890	2265.00	2265.00
2018-01-13	2610	2292.75	2292.75
2018-01-14	2500	2227.75	2227.75
2018-01-15	2750	2277.75	2277.75
2018-01-16	2700	2250.50	2250.50
2018-01-17	2250	2157.50	2157.50
2018-01-18	2350	2115.00	2115.00
2018-01-19	2550	2088.75	2088.75
2018-01-20	3000	2187.50	2187.50

Теперь вычисляем взвешенные скользящие средние с шириной окна 4, в знаменателе возьмем сумму весов наблюдений в окне.

```
# вычисляем взвешенные скользящие средние с шириной окна 4,
# в знаменателе - сумма весов наблюдений в окне
data['weighted_rolling_mean4_oth'] = data['sales'].rolling(
    window=4).apply(lambda x: np.sum(weights * x) /
                    np.sum(weights))
data
```

	sales	weighted_rolling_mean4	weighted_rolling_mean4_manually	weighted_rolling_mean4_oth
2018-01-09	2400	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN
2018-01-12	2890	2265.00	2265.00	2664.705882
2018-01-13	2610	2292.75	2292.75	2697.352941
2018-01-14	2500	2227.75	2227.75	2620.882353
2018-01-15	2750	2277.75	2277.75	2679.705882
2018-01-16	2700	2250.50	2250.50	2647.647059
2018-01-17	2250	2157.50	2157.50	2538.235294
2018-01-18	2350	2115.00	2115.00	2488.235294
2018-01-19	2550	2088.75	2088.75	2457.352941
2018-01-20	3000	2187.50	2187.50	2573.529412

Давайте вручную вычислим взвешенные скользящие средние с шириной окна 4, в знаменателе – сумма весов наблюдений в окне.

```
# вручную вычисляем взвешенные скользящие средние
# с шириной окна 4, в знаменателе - сумма весов
# наблюдений в окне
data['weighted_rolling_mean4_oth_manually'] = np.NaN
sum_of_weights = (0.7 + 0.8 + 0.9 + 1)
data.iloc[3, 4] = (2400 * 0.7 + 2800 * 0.8 +
                  2500 * 0.9 + 2890 * 1) / sum_of_weights
data.iloc[4, 4] = (2800 * 0.7 + 2500 * 0.8 +
                  2890 * 0.9 + 2610 * 1) / sum_of_weights
data.iloc[5, 4] = (2500 * 0.7 + 2890 * 0.8 +
                  2610 * 0.9 + 2500 * 1) / sum_of_weights
data.iloc[6, 4] = (2890 * 0.7 + 2610 * 0.8 +
                  2500 * 0.9 + 2750 * 1) / sum_of_weights
data.iloc[7, 4] = (2610 * 0.7 + 2500 * 0.8 +
                  2750 * 0.9 + 2700 * 1) / sum_of_weights
data.iloc[8, 4] = (2500 * 0.7 + 2750 * 0.8 +
                  2700 * 0.9 + 2250 * 1) / sum_of_weights
data.iloc[9, 4] = (2750 * 0.7 + 2700 * 0.8 +
                  2250 * 0.9 + 2350 * 1) / sum_of_weights
data.iloc[10, 4] = (2700 * 0.7 + 2250 * 0.8 +
                   2350 * 0.9 + 2550 * 1) / sum_of_weights
data.iloc[11, 4] = (2250 * 0.7 + 2350 * 0.8 +
                   2550 * 0.9 + 3000 * 1) / sum_of_weights
data
```

	<code>sales</code>	<code>weighted_rolling_mean4</code>	<code>weighted_rolling_mean4_manually</code>	<code>weighted_rolling_mean4_oth</code>	<code>weighted_rolling_mean4_oth_manually</code>
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN
2018-01-12	2890	2265.00	2265.00	2664.705882	2664.705882
2018-01-13	2610	2292.75	2292.75	2697.352941	2697.352941
2018-01-14	2500	2227.75	2227.75	2620.882353	2620.882353
2018-01-15	2750	2277.75	2277.75	2679.705882	2679.705882
2018-01-16	2700	2250.50	2250.50	2647.647059	2647.647059
2018-01-17	2250	2157.50	2157.50	2538.235294	2538.235294
2018-01-18	2350	2115.00	2115.00	2488.235294	2488.235294
2018-01-19	2550	2088.75	2088.75	2457.352941	2457.352941
2018-01-20	3000	2187.50	2187.50	2573.529412	2573.529412

Чтобы получить взвешенные скользящие средние на тесте, создаем защиту от утечек в виде значений NaN для тестовой части и вычисляем статистики с лагом, равным горизонту прогнозирования.

```
# отключаем предупреждения
import warnings
warnings.filterwarnings('ignore')

# значения в наблюдениях, которые будут приходиться
# на тест (последние 4 наблюдения), заменяем
# на значения NaN
HORIZON = 4
data['sales'].iloc[-HORIZON:] = np.NaN

# вычисляем взвешенное скользящее среднее с шириной
# окна 4 и лагом 4, в знаменателе - количество
# наблюдений в окне
data['weighted_rolling_mean4_sh'] = data['sales'].shift(
    periods=4).rolling(window=4, min_periods=1).apply(
        lambda x: np.mean(weights * x))

# вычисляем взвешенное скользящее среднее с шириной
# окна 4 и лагом 4, в знаменателе - сумма весов
# наблюдений в окне
data['weighted_rolling_mean4_oth_sh'] = data['sales'].shift(
    periods=4).rolling(window=4, min_periods=1).apply(
        lambda x: np.sum(weights * x) / np.sum(weights))
data
```

weighted_rolling_mean4_manually	weighted_rolling_mean4_oth	weighted_rolling_mean4_oth_manually	weighted_rolling_mean4_sh	weighted_rolling_mean4_oth_sh
NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN
2265.00000	2664.70588	2664.70588	NaN	NaN
2292.75000	2697.35294	2697.35294	2400.00000	705.88235
2227.75000	2620.88235	2620.88235	2480.00000	1458.82353
2277.75000	2679.70588	2679.70588	2313.33333	2041.17647
2250.50000	2647.64706	2647.64706	2265.00000	2664.70588
2157.50000	2538.23529	2538.23529	2292.75000	2697.35294
2115.00000	2488.23529	2488.23529	2227.75000	2620.88235
2088.75000	2457.35294	2457.35294	2277.75000	2679.70588
2187.50000	2573.52941	2573.52941	2250.50000	2647.64706

16.4.5. Групповые скользящие статистики

Заново загрузим *example.csv*.

```
# загружаем данные
data = load_data()
```

Создадим столбец *class* с двумя категориями.

```
# создаем столбец Class с двумя категориями
data['Class'] = pd.Series(['A', 'B', 'A', 'B', 'A', 'B',
                           'B', 'B', 'B', 'B', 'A', 'B'],
                           index=data.index)
data
```

	sales	class
2018-01-09	2400	A
2018-01-10	2800	B
2018-01-11	2500	A
2018-01-12	2890	B
2018-01-13	2610	A
2018-01-14	2500	B
2018-01-15	2750	B
2018-01-16	2700	B
2018-01-17	2250	B
2018-01-18	2350	B
2018-01-19	2550	A
2018-01-20	3000	B

Теперь отсортируем данные по категориям переменной *class*.

Обратите внимание, категории должны быть отсортированы по датам.

```
# отсортируем по переменной Class
data = data.sort_values('class', ascending=True)
data
```

	sales	class
2018-01-09	2400	A
2018-01-11	2500	A
2018-01-13	2610	A
2018-01-19	2550	A
2018-01-10	2800	B
2018-01-12	2890	B
2018-01-14	2500	B
2018-01-15	2750	B
2018-01-16	2700	B
2018-01-17	2250	B
2018-01-18	2350	B
2018-01-20	3000	B

Вычислим групповые скользящие средние с шириной окна 4 и лагом 1.

```
# вычислим групповые скользящие средние
# с шириной окна 4 и лагом 1
data['groupby_rolling_mean'] = data.groupby(
    'class')['sales'].transform(lambda s: s.shift(
        periods=1).rolling(window=4, min_periods=1).agg('mean'))
data
```

	sales	class	groupby_rolling_mean
2018-01-09	2400	A	NaN
2018-01-11	2500	A	2400.00000
2018-01-13	2610	A	2450.00000
2018-01-19	2550	A	2503.33333
2018-01-10	2800	B	NaN
2018-01-12	2890	B	2800.00000
2018-01-14	2500	B	2845.00000
2018-01-15	2750	B	2730.00000
2018-01-16	2700	B	2735.00000
2018-01-17	2250	B	2710.00000
2018-01-18	2350	B	2550.00000
2018-01-20	3000	B	2512.50000

Теперь выясним, как вычислить групповые скользящие средние в наборе новых данных или тестовом наборе данных. Давайте создадим набор новых

данных, нам неизвестна зависимая переменная, зато известны категории переменной *class*.

```
# создаем набор новых данных
new_data = pd.DataFrame(
    {'sales': [np.NaN, np.NaN, np.NaN, np.NaN],
     'class': ['A', 'A', 'A', 'B']},
    index=pd.date_range(start='2018-01-21', periods=4))
new_data
```

	sales	class
2018-01-21	NaN	A
2018-01-22	NaN	A
2018-01-23	NaN	A
2018-01-24	NaN	B

Вручную вычислим групповые скользящие средние в наборе новых данных.

```
# вручную вычислим групповые скользящие
# средние в наборе новых данных
new_data['groupby_rolling_mean'] = np.NaN
new_data.iloc[0, 2] = (2550 + 2610 + 2500 + 2400) / 4
new_data.iloc[1, 2] = (2550 + 2610 + 2500) / 3
new_data.iloc[2, 2] = (2550 + 2610) / 2
new_data.iloc[3, 2] = (3000 + 2350 + 2250 + 2700) / 4
new_data
```

	sales	class	groupby_rolling_mean
2018-01-21	NaN	A	2515.000000
2018-01-22	NaN	A	2553.333333
2018-01-23	NaN	A	2580.000000
2018-01-24	NaN	B	2575.000000

date	sales	class
2018-01-09	2400	A
2018-01-11	2500	A
2018-01-13	2610	A
2018-01-19	2550	A
2018-01-10	2800	B
2018-01-12	2890	B
2018-01-14	2500	B
2018-01-15	2750	B
2018-01-16	2700	B
2018-01-17	2250	B
2018-01-18	2350	B
2018-01-20	3000	B

Групповые скользящие средние для набора новых данных (тестового набора) мы считаем по более поздним наблюдениям исторического набора (обучающего набора) с уменьшением ширины окна

При вычислении групповых скользящих средних для набора новых данных (тестового набора) мы не используем значения продаж в историческом наборе (обучающем наборе)

Теперь выполним конкатенацию исторического набора и набора новых данных и удалим переменную `groupby_rolling_mean`. Зависимая переменная в последние 4 наблюдениях, которые будут у нас приходиться на новые данные, состоит из значений NaN.

```
# конкatenируем исторический набор и набор новых данных
df = pd.concat([data, new_data], axis=0)
# удалим переменную groupby_rolling_mean
df.drop('groupby_rolling_mean', axis=1, inplace=True)
df
```

date	sales	class
2018-01-09	2400.0	A
2018-01-11	2500.0	A
2018-01-13	2610.0	A
2018-01-19	2550.0	A
2018-01-10	2800.0	B
2018-01-12	2890.0	B
2018-01-14	2500.0	B
2018-01-15	2750.0	B
2018-01-16	2700.0	B
2018-01-17	2250.0	B
2018-01-18	2350.0	B
2018-01-20	3000.0	B
2018-01-21	NaN	A
2018-01-22	NaN	A
2018-01-23	NaN	A
2018-01-24	NaN	B

Пишем функцию вычисления групповых скользящих средних `groupby_moving_stats()`.

```
# функция для создания групповых скользящих статистик
def groupby_moving_stats(df, by, target, min_periods=1,
                         window=4, aggfunc='mean',
                         offset=False, start_date=None):
    """
    Создает групповые скользящие статистики.

Параметры
-----
df: pandas.DataFrame
    Датафрейм Pandas, наблюдения, приходящиеся
    на историческую (обучающую) часть, должны
    быть отсортированы по категориям
    группирующей переменной.
by: string
    Имя группирующего столбца.
target: string
    Имя агрегируемого столбца.
min_periods: int, по умолчанию 1
    Минимальное количество наблюдений в окне для
    вычисления скользящих статистик.
window: int, по умолчанию 4
    Ширина окна.
offset: int, по умолчанию 0
    Смещение.
aggfunc: string, по умолчанию 'mean'
    Агрегирующая функция.
offset: bool, по умолчанию True
    Вычисляет групповые скользящие
    статистики со смещением по времени.
start_date: datetime
    Стартовая дата для смещения.
"""
if offset:
    df = df[df.index >= start_date]

# если не является списком
if type(by) != list:
    # преращаем в список
    by = [by]

df = df.groupby(by)[target].transform(
    lambda x: x.shift(1).rolling(
        window=window,
        min_periods=min_periods).agg(aggfunc))
df.fillna(0, inplace=True)
return df
```

Применим нашу функцию для вычисления некоторых групповых скользящих статистик к набору, полученному в результате конкатенации.

```
# создаем список агрегирующих функций
aggfunc_lst = ['mean', 'median', 'sum']
```

```
# вычисляем групповые скользящие статистики с
# помощью функции groupby_moving_stats()
for aggfunc in aggfunc_lst:
    df[f"groupby_rolling_{aggfunc}"] = groupby_moving_stats(
        df, by='class', target='sales',
        min_periods=1, window=4, aggfunc=aggfunc)
df
```

	sales	class	groupby_rolling_mean	groupby_rolling_median	groupby_rolling_sum
2018-01-09	2400.00000	A	0.00000	0.00000	0.00000
2018-01-11	2500.00000	A	2400.00000	2400.00000	2400.00000
2018-01-13	2610.00000	A	2450.00000	2450.00000	4900.00000
2018-01-19	2550.00000	A	2503.33333	2500.00000	7510.00000
2018-01-10	2800.00000	B	0.00000	0.00000	0.00000
2018-01-12	2890.00000	B	2800.00000	2800.00000	2800.00000
2018-01-14	2500.00000	B	2845.00000	2845.00000	5690.00000
2018-01-15	2750.00000	B	2730.00000	2800.00000	8190.00000
2018-01-16	2700.00000	B	2735.00000	2775.00000	10940.00000
2018-01-17	2250.00000	B	2710.00000	2725.00000	10840.00000
2018-01-18	2350.00000	B	2550.00000	2600.00000	10200.00000
2018-01-20	3000.00000	B	2512.50000	2525.00000	10050.00000
2018-01-21	NaN	A	2515.00000	2525.00000	10060.00000
2018-01-22	NaN	A	2553.33333	2550.00000	7660.00000
2018-01-23	NaN	A	2580.00000	2580.00000	5160.00000
2018-01-24	NaN	B	2575.00000	2525.00000	10300.00000

16.4.6. Расширяющиеся статистики

Теперь поговорим о расширяющихся статистиках. В рамках подхода «расширяющееся окно» библиотека pandas вычисляет статистику по «окну» данных, представляющему определенный период времени. Затем окно охватывает больший интервал времени и статистика постоянно вычисляется для каждого нового окна до тех пор, пока окно охватывает даты временного ряда. Библиотека pandas непосредственно поддерживает расширяющиеся оконные функции, предлагая метод `.expanding()` объекта Series и объекта DataFrame.

Заметим, что расширяющееся среднее используется не только для конструирования признаков, но и в качестве прогнозной модели (когда прогноз – расширяющееся среднее n наблюдений).

Давайте создадим расширяющееся среднее. Здесь у нас не будет фиксированной ширины окна, поскольку окно каждый раз расширяется, у нас будет только параметр `min_periods` – минимальное количество наблюдений в окне для вычисления расширяющегося среднего.

Заново загрузим `example.csv`.

```
# загружаем данные
data = load_data()
```

создаем расширяющееся среднее

```
data['expanding_mean'] = data['sales'].expanding().mean()
data
```

	sales	expanding_mean
2018-01-09	2400	2400.000000
2018-01-10	2800	2600.000000
2018-01-11	2500	2566.666667
2018-01-12	2890	2647.500000
2018-01-13	2610	2640.000000
2018-01-14	2500	2616.666667
2018-01-15	2750	2635.714286
2018-01-16	2700	2643.750000
2018-01-17	2250	2600.000000
2018-01-18	2350	2575.000000
2018-01-19	2550	2572.727273
2018-01-20	3000	2608.333333

Удаляем переменную *expanding_mean* и в последних 4 наблюдениях, которые будут у нас соответствовать тестовой выборке, значения переменной *sales* заменяем на значения NaN (таким образом, у нас горизонт прогнозирования равен 4).

удаляем переменную expanding_mean

```
data.drop('expanding_mean', axis=1, inplace=True)
# в последних 4 наблюдениях, которые будут соответствовать
# тестовой выборке, значения переменной sales заменяем
# на значения NaN
data.iloc[-4:] = np.NaN
data
```

	sales
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0
2018-01-17	NaN
2018-01-18	NaN
2018-01-19	NaN
2018-01-20	NaN

Теперь напишем функцию для вычисления расширяющихся статистик `expanding_stats()`.

```
# функция для создания расширяющихся статистик
def expanding_stats(series, min_periods=1,
                     periods=1, aggfunc='mean'):
    """
    Создает скользящие статистики.

Параметры
-----
min_periods: int, по умолчанию 1
    Минимальное количество наблюдений
    в окне для вычисления
    расширяющихся статистик.
periods: int, по умолчанию 1
    Порядок лага, с которым вычисляем
    расширяющиеся статистики (не должен
    быть меньше горизонта
    прогнозирования).
aggfunc: string, по умолчанию 'mean'
    Агрегирующая функция.
"""

# вычисление расширяющихся статистик
features = series.shift(
    periods=periods).expanding(
    min_periods=min_periods).agg(aggfunc)
features.fillna(0, inplace=True)
return features
```

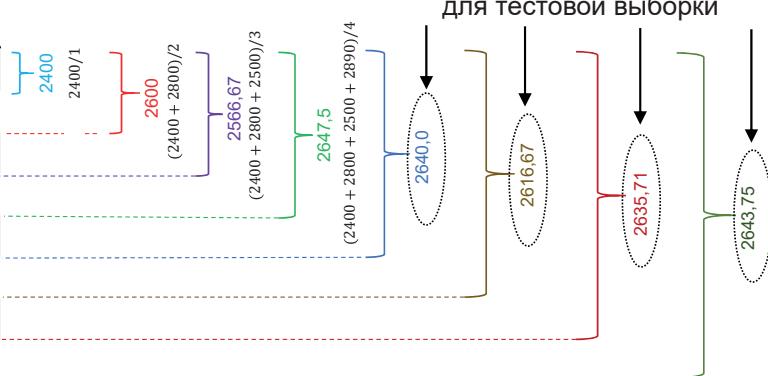
Чтобы получить расширяющиеся средние, не используя информацию тестовой выборки, мы должны их вычислить с лагом, равным горизонту прогнозирования (или выше). Давайте так и сделаем. Вычисляем расширяющиеся средние с лагом 4, т. е. с лагом, равным горизонту прогнозирования.

```
# вычисляем расширяющиеся средние с лагом 4, т. е.
# с лагом, равным горизонту прогнозирования
data['expanding_mean'] = expanding_stats(
    data['sales'],
    periods=4,
    min_periods=1,
    aggfunc='mean')
data
```

	sales	expanding_mean
2018-01-09	2400.0	0.000000
2018-01-10	2800.0	0.000000
2018-01-11	2500.0	0.000000
2018-01-12	2890.0	0.000000
2018-01-13	2610.0	2400.000000
2018-01-14	2500.0	2600.000000
2018-01-15	2750.0	2566.666667
2018-01-16	2700.0	2647.500000
2018-01-17	NaN	2640.000000
2018-01-18	NaN	2616.666667
2018-01-19	NaN	2635.714286
2018-01-20	NaN	2643.750000

	sales
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0
2018-01-17	NaN
2018-01-18	NaN
2018-01-19	NaN
2018-01-20	NaN

ЭТИ значения используем
для тестовой выборки



16.4.7. Групповые статистики

Заново загрузим *example.csv*.

```
# загружаем данные
data = load_data()
```

Создадим столбец *class* с тремя категориями и количественный столбец *income*.

```
# создаем столбец Class с тремя категориями
data['class'] = pd.Series(
    ['A', 'B', 'A', 'B', 'A', 'B',
     'B', 'B', 'B', 'B', 'A', 'C'],
    index=data.index)

# создаем количественный столбец income
data['income'] = pd.Series(
    [4500, 5000, 5100, 4800, 5300, 5200,
     4800, 4950, 5050, 4850, 4700, 5030],
    index=data.index)
```

Разбиваем набор на обучающую и тестовую выборки, взглянем на них.

```
# разбиваем набор на обучающую и тестовую выборки
train, test = data[0:len(data)-4], data[len(data)-4:]
```

```
# смотрим обучающую выборку
train
```

	sales	class	income
2018-01-09	2400	A	4500
2018-01-10	2800	B	5000
2018-01-11	2500	A	5100
2018-01-12	2890	B	4800
2018-01-13	2610	A	5300
2018-01-14	2500	B	5200
2018-01-15	2750	B	4800
2018-01-16	2700	B	4950

```
# смотрим тестовую выборку
test
```

	sales	class	income
2018-01-17	2250	B	5050
2018-01-18	2350	B	4850
2018-01-19	2550	A	4700
2018-01-20	3000	C	5030

Обратите внимание: в тестовой выборке есть категория C, которая отсутствует в обучающей выборке.

Теперь напишем функцию вычисления групповых статистик `grouped_stats()`.

```
# пишем функцию вычисления групповых статистик
def grouped_stats(df_treat, df_calc, var,
                  by, func='mean', fillna=None):
```

```
"""
Создаем групповые статистики.

Параметры
-----
df_treat: pandas.DataFrame
    Датафрейм Pandas, для которого
    вычисляем статистики.
df_calc: pandas.DataFrame
    Датафрейм Pandas, на основе которого
    вычисляются статистики.
var: string
    Имя агрегируемого столбца.
by: string
    Имя группирующего столбца.
aggfunc: string, по умолчанию 'mean'
    Агрегирующая функция.
fillna, int, по умолчанию -1
    Стратегия импутации пропусков.
"""

# если не является списком
if type(by) != list:
    # превращаем в список
    by = [by]
# задаем имя
name = '{0}_{1}_mean'.format(var, by)
# получаем статистики по каждой комбинации
# категорий признаков в списке
grp = df_calc.groupby(by)[[var]].agg(func)

# присваиваем имя
grp.columns = [name]
# записываем результаты в исходный датафрейм
var = pd.merge(df_treat[by], grp, left_on=by,
               right_index=True, how='left')[name]

# импутируем пропуски
if fillna is not None:
    var.fillna(fillna, inplace=True)

return var
```

Давайте вычислим групповые статистики – средние, минимумы, максимумы, медианы, стандартные отклонения, разности, средние абсолютные отклонения, медианные абсолютные отклонения переменной *income* по группам – категориям переменной *class*. Сначала вычислим их в обучающей выборке. Здесь мы будем придерживаться консервативного сценария, вычисляя соответствующую групповую статистику на всей обучающей выборке, однако поскольку более ранние данные могут просто устареть, не отражать изменившуюся ситуацию, всегда полезно попробовать вычислить статистику по более позднему периоду обучающей выборки (например, вычислить групповое

среднее по последнему году или последним 2 годам обучающей выборки, которая включает 5 лет наблюдений).

```
# вычисляем групповые средние, минимумы, максимумы,
# медианы, стандартные отклонения, разности, средние
# абсолютные отклонения, медианные абсолютные
# отклонения на обучающей выборке
train['mean_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='mean', fillna=train['income'].mean())
train['min_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='min', fillna=train['income'].mean())
train['max_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='max', fillna=train['income'].mean())
train['median_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='median', fillna=train['income'].mean())
train['std_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='std', fillna=train['income'].mean())
train['diff_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func=np.ptp, fillna=train['income'].mean())
train['mad_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='mad', fillna=train['income'].mean())
train['mdad_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func=lambda x: np.nanmedian(
        np.abs(x - np.nanmedian(x, axis=0))), axis=0),
    fillna=train['income'].mean())
train
```

income	mean_inc_by_class	min_inc_by_class	max_inc_by_class	median_inc_by_class	std_inc_by_class	diff_inc_by_class	mad_inc_by_class	mdad_inc_by_class
4500	4966.66667	4500	5300	5100.00000	416.33320	800	311.11111	200.00000
5000	4950.00000	4800	5200	4950.00000	165.83124	400	120.00000	150.00000
5100	4966.66667	4500	5300	5100.00000	416.33320	800	311.11111	200.00000
4800	4950.00000	4800	5200	4950.00000	165.83124	400	120.00000	150.00000
5300	4966.66667	4500	5300	5100.00000	416.33320	800	311.11111	200.00000
5200	4950.00000	4800	5200	4950.00000	165.83124	400	120.00000	150.00000
4800	4950.00000	4800	5200	4950.00000	165.83124	400	120.00000	150.00000
4950	4950.00000	4800	5200	4950.00000	165.83124	400	120.00000	150.00000

Теперь вычислим групповые статистики в тестовой выборке. Здесь используем групповые статистики, вычисленные на обучающей выборке.

```

# вычисляем групповые средние, минимумы, максимумы,
# медианы, стандартные отклонения, разности, средние
# абсолютные отклонения, медианные абсолютные
# отклонения на тестовой выборке
test['mean_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func='mean', fillna=train['income'].mean())
test['min_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func='min', fillna=train['income'].mean())
test['max_inc_by_class'] = grouped_stats(
    train, train, var='income', by='class',
    func='max', fillna=train['income'].mean())
test['median_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func='median', fillna=train['income'].mean())
test['std_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func='std', fillna=train['income'].mean())
test['diff_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func=np.ptp, fillna=train['income'].mean())
test['mad_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func='mad', fillna=train['income'].mean())
train['mdad_inc_by_class'] = grouped_stats(
    test, train, var='income', by='class',
    func=lambda x: np.nanmedian(
        np.abs(x - np.nanmedian(x, axis=0)), axis=0),
    fillna=train['income'].mean())
test

```

income	mean_inc_by_class	min_inc_by_class	max_inc_by_class	median_inc_by_class	std_inc_by_class	diff_inc_by_class	mad_inc_by_class	mdad_inc_by_class
5050	4950.00000	4800.00000	5200.00000	4950.00000	165.83124	400.00000	120.00000	150.00000
4850	4950.00000	4800.00000	5200.00000	4950.00000	165.83124	400.00000	120.00000	150.00000
4700	4966.66667	4500.00000	5300.00000	5100.00000	416.33320	800.00000	311.11111	200.00000
5030	4956.25000	4956.25000	4956.25000	4956.25000	4956.25000	4956.25000	4956.25000	4956.25000

16.4.8. Признаки даты и времени

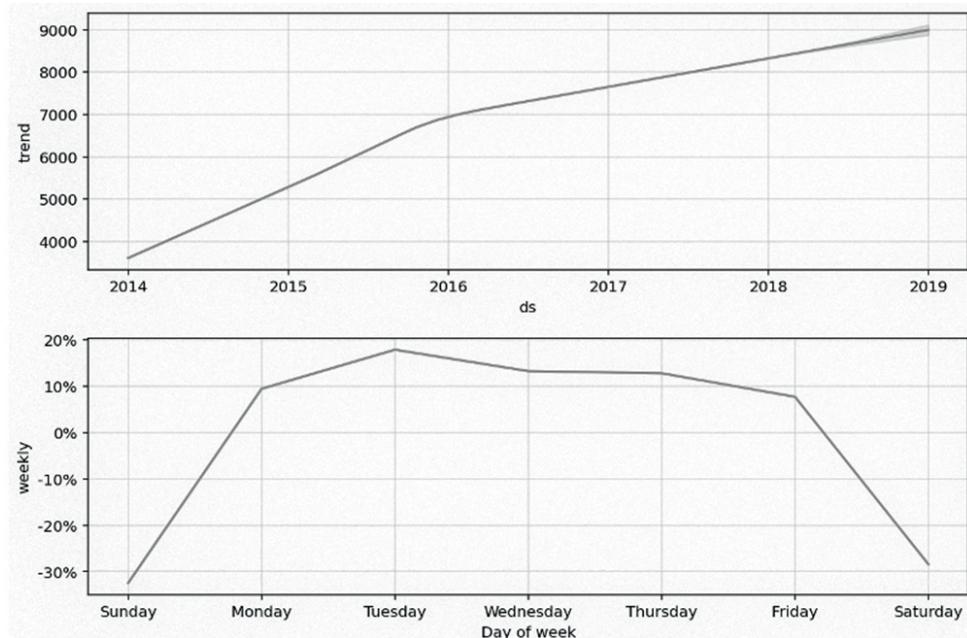
Как правило, столбец с датой или столбец с датой и временем не является переменной для моделирования, переменные нужно выделить из него самостоятельно, например выделить признаки даты (года, порядковые номера кварталов, порядковые номера месяцев, порядковые номера дней года, порядковые номера дней месяца, порядковые номера недель, порядковые номера дней недели) и признаки времени (части суток, часы, минуты) и затем подать на вход модели. Многие признаки даты и времени имеют циклический характер (минуты, часы, дни недели, месяцы, кварталы). К признакам даты еще можно отнести бинарные признаки типа праздничный день / непраздничный день, будний день / выходной день.

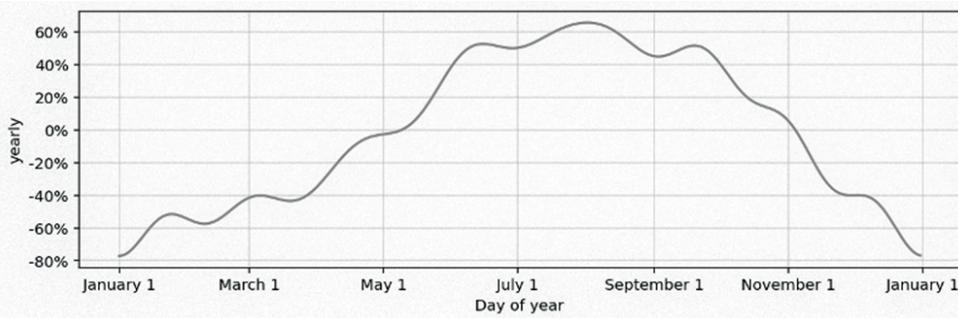
Для моделирования сезонностей полезны следующие признаки даты и времени:

- годовая сезонность (продажи мороженого, пик продаж – в летние месяцы и максимальный спад продаж – в зимние месяцы) – порядковый номер дня года, порядковый номер или строковое название месяца, порядковый номер или строковое название сезона, комбинация порядкового номера месяца и порядкового номера дня месяца, комбинация порядкового номера недели года и порядкового номера дня недели;
- квартальная сезонность (частота обращений к налоговым консультантам, пик – первый месяц каждого квартала) – порядковый номер квартала, порядковый номер дня квартала, начало/конец квартала;
- месячная сезонность (продажи в дни выдачи зарплат в моногородах, городах с 2–3 крупными предприятиями – каждого 1-го и каждого 15-го числа месяца) – порядковый номер дня месяца, начало/середина/конец месяца;
- недельная сезонность (объем поездок на велосипедах – подъем в будние дни, снижение в выходные дни) – порядковый номер или строковое название дня недели, индикатор выходного дня, комбинация порядкового номера недели года и порядкового номера дня недели;
- суточная сезонность (пик пассажиропотока в утренние часы, спад в середине дня и подъем в вечерние часы) – порядковый номер часа, часть суток.

Анализ различных типов сезонности удобно выполнять с помощью метода `.plot_components()` библиотеки Prophet.

```
# смотрим графики компонентов
fig = model.plot_components(forecast)
plt.show()
```





Импортируем необходимые библиотеки, модули, классы, функции и загрузим набор с ежедневными продажами лука, на котором проиллюстрируем создание признаков даты.

```
# импортируем библиотеки numpy, pandas,
# polars, requests
import numpy as np
import pandas as pd
import polars as pl
import requests

# импортируем модули json и datetime
import json
import datetime

# импортируем функции ceil() и reduce()
from math import ceil
from functools import reduce

# импортируем необходимые классы и функции
# библиотеки sklearn
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

# импортируем класс CatBoostRegressor
# библиотеки catboost
from catboost import CatBoostRegressor

# импортируем класс ExponentialSmoothing
# библиотеки statsmodels
from statsmodels.tsa.api import ExponentialSmoothing

# импортируем необходимые классы sktime
from sktime.forecasting.ets import AutoETS
from sktime.forecasting.base import ForecastingHorizon

# импортируем модуль pyplot библиотеки matplotlib
import matplotlib.pyplot as plt
```

```
# настраиваем визуализацию
%config InlineBackend.figure_format = 'retina'

# записываем датафрейм на основе CSV-файла,
# содержащего ежедневные продажи лука
data = pd.read_csv('Data/X5_ARIMA.csv')
data
```

	day	rto
0	2018-01-01	5641.51000
1	2018-01-02	17525.38000
2	2018-01-03	22780.76000
3	2018-01-04	23590.37000
4	2018-01-05	29344.19000
...
1091	2020-12-27	81329.60000
1092	2020-12-28	96236.12000
1093	2020-12-29	105614.55000
1094	2020-12-30	148178.60000
1095	2020-12-31	88948.67000

Вспомним, чтобы с переменной *day* (дата продаж) работать как с переменной, содержащей даты, ей нужно присвоить тип *datetime*.

```
# переменной day присваиваем тип datetime,
# чтобы работать с этой переменной как с датами
data['day'] = pd.to_datetime(data['day'])
```

Выделяем из переменной *date_start* года, порядковые номера кварталов, порядковые номера месяцев, порядковые номера дней года, порядковые номера дней месяца, порядковые номера дней недели, порядковые номера недель. Для этого мы можем использовать либо связку *pd.DatetimeIndex()* и атрибута, либо связку аксессора *.dt* и атрибута. Обратите внимание: день недели можно вычислить двумя способами – по американской системе (неделя начинается с воскресенья) и по российской системе (неделя начинается с понедельника). Номер недели тоже можно вычислить двумя способами, '*%U*' представляет номер недели в году (воскресенье как первый день недели) в виде десятичного числа, заполненного нулями. В рамках этого способа все дни нового года, предшествующие первому воскресенью, считаются нулевой неделей. Кроме того, путем нехитрых манипуляций и простых самостоятельно написанных функций мы можем создать индикатор выходного дня, название дня недели, название месяца, сезон, порядковый номер недели в месяце, порядковый номер декады в месяце.

```
# увеличиваем количество отображаемых столбцов
pd.set_option('display.max_columns', 50)
```

```
# выделяем из переменной day ГОДА
data['year'] = pd.DatetimeIndex(data['day']).year
data['year_alter'] = data['day'].dt.year

# выделяем из переменной day ПОРЯДКОВЫЕ НОМЕРА
# КВАРТАЛОВ от 1 до 4
data['quarter'] = pd.DatetimeIndex(data['day']).quarter
data['quarter_alter'] = data['day'].dt.quarter

# выделяем из переменной day ПОРЯДКОВЫЕ НОМЕРА
# МЕСЯЦЕВ от 1 до 12
data['month'] = pd.DatetimeIndex(data['day']).month
data['month_alter'] = data['day'].dt.month

# выделяем из переменной day ПОРЯДКОВЫЕ
# НОМЕРА ДНЕЙ ГОДА от 1 до 365
data['dayofyear'] = pd.DatetimeIndex(data['day']).dayofyear
data['dayofyear_alter'] = data['day'].dt.dayofyear

# выделяем из переменной day ПОРЯДКОВЫЕ НОМЕРА
# ДНЕЙ МЕСЯЦА от 1 до 31
data['dayofmonth'] = pd.DatetimeIndex(data['day']).day
data['dayofmonth_alter'] = data['day'].dt.day

# выделяем из переменной day ПОРЯДКОВЫЕ
# НОМЕРА ДНЕЙ НЕДЕЛИ
# от 0 до 6, где 0 - понедельник, 6 - воскресенье
# (амер. система)
# от 1 до 7, где 1 - понедельник, 7 - воскресенье
# (росс. система)
data['dayofweek_usa'] = pd.DatetimeIndex(data['day']).dayofweek
data['dayofweek_usa_alter'] = data['day'].dt.dayofweek
data['dayofweek_russia'] = (data['dayofweek_usa'] + 7) - 6

# выделяем из переменной day ПОРЯДКОВЫЕ НОМЕРА НЕДЕЛЬ
# либо с 1 по 52, либо с 00 до 52
data['week'] = data['day'].dt.isocalendar().week
data['week_alter'] = data['day'].dt.strftime('%U')

# создаем переменную - обычный или выходной день
data['weekend'] = np.where(
    data['dayofweek_usa'].isin([5, 6]), 1, 0)

# пишем функцию, которая извлекает название дня недели
def dayNameFromWeekday(weekday):
    days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']
    return days[weekday]

# создаем переменную - название дня недели
data['weekday_name'] = data['dayofweek_usa'].apply(
    lambda x: dayNameFromWeekday(x))
data['weekday_name_alter'] = data['day'].dt.day_name()

# создаем переменную - название месяца
data['month_name'] = data['day'].dt.strftime('%b')
data['month_name_alter'] = data['day'].dt.month_name()
```

```

# пишем функцию для создания переменной - сезон
def get_season(month):
    if (month > 11 or month < 3):
        return 'WINTER'
    elif (month == 3 or month <= 5):
        return 'SPRING'
    elif (month >= 6 and month < 9):
        return 'SUMMER'
    else:
        return 'FALL'

# создаем переменную - сезон
data['season'] = data['month'].apply(
    lambda x: get_season(x))

# пишем функцию, вычисляющую ПОРЯДКОВЫЙ НОМЕР
# НЕДЕЛИ В МЕСЯЦЕ
def week_of_month(dt):
    first_day = dt.replace(day=1)
    dom = dt.day
    adjusted_dom = dom + first_day.weekday()
    return int(ceil(adjusted_dom / 7.0))

# получаем ПОРЯДКОВЫЙ НОМЕР НЕДЕЛИ В МЕСЯЦЕ
data['week_of_month'] = data['day'].apply(
    week_of_month).values

# пишем функцию вычисления декад
def create_decades(series):
    lst = list()
    for d in series:
        res = ceil(d.day / 10.5)
        lst.append(res)
    return lst

# получаем ПОРЯДКОВЫЙ НОМЕР ДЕКАДЫ В МЕСЯЦЕ
data['decades'] = create_decades(data['day'])

# смотрим результаты
data

```

day	rto	year	year_alter	quarter	quarter_alter	month	month_alter	dayofyear	dayofyear_alter	dayofmonth	dayofmonth_alter	dayofweek_usa
0	2018-01-01	5641.51	2018	1	1	1	1	1	1	1	1	0
1	2018-01-02	17525.38	2018	1	1	1	1	2	2	2	2	1
2	2018-01-03	22780.76	2018	1	1	1	1	3	3	3	3	2
3	2018-01-04	23590.37	2018	1	1	1	1	4	4	4	4	3
4	2018-01-05	29344.19	2018	1	1	1	1	5	5	5	5	4
...
1091	2020-12-27	81329.60	2020	4	4	12	12	362	362	27	27	6
1092	2020-12-28	96236.12	2020	4	4	12	12	363	363	28	28	0
1093	2020-12-29	105614.55	2020	4	4	12	12	364	364	29	29	1
1094	2020-12-30	148178.60	2020	4	4	12	12	365	365	30	30	2
1095	2020-12-31	88948.67	2020	4	4	12	12	366	366	31	31	3

384 ❖ План предварительной подготовки данных

day	dayofweek_usa_alter	dayofweek_russia	week	week_alter	weekend	weekday_name	weekday_name_alter	month_name	month_name_alter	season
0 2018-01-01		0	1	1	00	0	Monday	Monday	Jan	January WINTER
1 2018-01-02		1	2	1	00	0	Tuesday	Tuesday	Jan	January WINTER
2 2018-01-03		2	3	1	00	0	Wednesday	Wednesday	Jan	January WINTER
3 2018-01-04		3	4	1	00	0	Thursday	Thursday	Jan	January WINTER
4 2018-01-05		4	5	1	00	0	Friday	Friday	Jan	January WINTER
...
1091 2020-12-27		6	7	52	52	1	Sunday	Sunday	Dec	December WINTER
1092 2020-12-28		0	1	53	52	0	Monday	Monday	Dec	December WINTER
1093 2020-12-29		1	2	53	52	0	Tuesday	Tuesday	Dec	December WINTER
1094 2020-12-30		2	3	53	52	0	Wednesday	Wednesday	Dec	December WINTER
1095 2020-12-31		3	4	53	52	0	Thursday	Thursday	Dec	December WINTER

day	week_of_month	decades
0 2018-01-01		1 1
1 2018-01-02		1 1
2 2018-01-03		1 1
3 2018-01-04		1 1
4 2018-01-05		1 1
...
1091 2020-12-27		4 3
1092 2020-12-28		5 3
1093 2020-12-29		5 3
1094 2020-12-30		5 3
1095 2020-12-31		5 3

Теперь создадим индикаторы – начало года, конец года, начало квартала, конец квартала, начало месяца, конец месяца.

```
# выделяем из переменной day
# ИНДИКАТОР - начало года
data['year_start'] = data['day'].dt.is_year_start
# выделяем из переменной day
# ИНДИКАТОР - конец года
data['year_end'] = data['day'].dt.is_year_end
# выделяем из переменной day
# ИНДИКАТОР - начало квартала
data['qrt_start'] = data['day'].dt.is_quarter_start
# выделяем из переменной day
# ИНДИКАТОР - конец квартала
data['qrt_end'] = data['day'].dt.is_quarter_end
# выделяем из переменной day
# ИНДИКАТОР - начало месяца
data['month_start'] = data['day'].dt.is_month_start
# выделяем из переменной day
# ИНДИКАТОР - конец месяца
data['month_end'] = data['day'].dt.is_month_end
# смотрим результаты
data
```

	day	year_start	year_end	qrt_start	qrt_end	month_start	month_end	leap_year
0	2018-01-01	True	False	True	False	True	False	False
1	2018-01-02	False	False	False	False	False	False	False
2	2018-01-03	False	False	False	False	False	False	False
3	2018-01-04	False	False	False	False	False	False	False
4	2018-01-05	False	False	False	False	False	False	False
...
1091	2020-12-27	False	False	False	False	False	False	True
1092	2020-12-28	False	False	False	False	False	False	True
1093	2020-12-29	False	False	False	False	False	False	True
1094	2020-12-30	False	False	False	False	False	False	True
1095	2020-12-31	False	True	False	True	False	True	True

При создании индикаторов мы не учитываем, на какой день – будний или выходной – приходится начало/конец месяца, начало/конец квартала. Кроме того, нам может потребоваться информация о конкретных днях месяца, например мы знаем, что выплата зарплаты и поставка товара влияют на продажи, зарплата выплачивается 1-го и 15-го числа каждого месяца, а поставка товара осуществляется в последний четверг каждого месяца. Для создания таких признаков мы можем воспользоваться классами для смещения дат.

Мы можем извлечь первый и 16-й календарные дни каждого месяца.

```
# задаем частоту - первый и 16-й календарные
# дни каждого месяца
frst_sixteenth_days = pd.tseries.offsets.SemiMonthBegin(
    day_of_month=16)
# создаем датафрейм с датами согласно частоте
frst_sixteenth_days_of_month = pd.DataFrame(
    {'day': pd.date_range('2018-01-01', '2020-12-31',
                          freq=frst_sixteenth_days),
     'frst_sixteenth_days_of_month': 1})
# смотрим первые 5 наблюдений
frst_sixteenth_days_of_month.head()
```

	day	frst_sixteenth_days_of_month
0	2018-01-01	1
1	2018-01-16	1
2	2018-02-01	1
3	2018-02-16	1
4	2018-03-01	1

Теперь извлечем 16-й и последний календарные дни каждого месяца.

```
# задаем частоту - 16-й и последний календарные
# дни каждого месяца
sixteenth_last_days = pd.tseries.offsets.SemiMonthEnd(
    day_of_month=16)
```

```
# создаем датафрейм с датами согласно частоте
sixteenth_last_days_of_month = pd.DataFrame(
    {'day': pd.date_range('2018-01-01', '2020-12-31',
                          freq=sixteenth_last_days),
     'sixteenth_last_days_of_month': 1})
# смотрим первые 5 наблюдений
sixteenth_last_days_of_month.head()
```

	day	sixteenth_last_days_of_month
0	2018-01-16	1
1	2018-01-31	1
2	2018-02-16	1
3	2018-02-28	1
4	2018-03-16	1

Теперь извлечем последний четверг каждого месяца.

```
# задаем частоту - последний четверг каждого месяца
last_thurs = pd.offsets.LastWeekOfMonth(weekday=3)
# создаем датафрейм с датами согласно частоте
last_thursday_of_month = pd.DataFrame(
    {'day': pd.date_range('2018-01-01', '2020-12-31',
                          freq=last_thurs),
     'last_thursday_of_month': 1})
# смотрим первые 5 наблюдений
last_thursday_of_month.head()
```

	day	last_thursday_of_month
0	2018-01-25	1
1	2018-02-22	1
2	2018-03-29	1
3	2018-04-26	1
4	2018-05-31	1

Извлечем первый рабочий день каждого месяца.

```
# создаем датафрейм с датами, частота -
# первый рабочий день каждого месяца
frst_work_day_of_month = pd.DataFrame(
    {'day': pd.date_range('2018-01-01', '2020-12-31', freq='BMS'),
     'frst_work_day_of_month': 1})
# смотрим первые 5 наблюдений
frst_work_day_of_month.head()
```

day	frst_work_day_of_month
0 2018-01-01	1
1 2018-02-01	1
2 2018-03-01	1
3 2018-04-02	1
4 2018-05-01	1

```
# создаем датафрейм с датами, частота -
# последний рабочий день каждого месяца
last_work_day_of_month = pd.DataFrame(
    {'day': pd.date_range('2018-01-01', '2020-12-31', freq='BM'),
     'last_work_day_of_month': 1})
# смотрим первые 5 наблюдений
last_work_day_of_month.head()
```

day	last_work_day_of_month
0 2018-01-31	1
1 2018-02-28	1
2 2018-03-30	1
3 2018-04-30	1
4 2018-05-31	1

Добавим последний четверг каждого месяца, первый рабочий день каждого месяца, последний рабочий день каждого месяца в исходный набор. Для этого к датафрейму `data` присоединяем датафреймы `last_thursday_of_month`, `frst_work_day_of_month`, `last_work_day_of_month`.

```
# к датафрейму data присоединяем датафреймы
# last_thursday_of_month, frst_work_day_of_month,
# last_work_day_of_month
dfs = [data, last_thursday_of_month,
       frst_work_day_of_month,
       last_work_day_of_month]
data = reduce(lambda left, right: pd.merge(
    left, right, how='left',
    on='day'), dfs)
# заменим пропуски нулями
data.fillna(0, inplace=True)
# преобразуем в тип int
for col in ['last_thursday_of_month',
            'frst_work_day_of_month',
            'last_work_day_of_month']:
    data[col] = data[col].astype(int)
data
```

388 ❖ План предварительной подготовки данных

decades	year_start	year_end	qrt_start	qrt_end	month_start	month_end	leap_year	last_thursday_of_month	frst_work_day_of_month	last_work_day_of_month
1	True	False	True	False	True	False	False	0	1	0
1	False	False	False	False	False	False	False	0	0	0
1	False	False	False	False	False	False	False	0	0	0
1	False	False	False	False	False	False	False	0	0	0
1	False	False	False	False	False	False	False	0	0	0
...
3	False	False	False	False	False	False	True	0	0	0
3	False	False	False	False	False	False	True	0	0	0
3	False	False	False	False	False	False	True	0	0	0
3	False	False	False	False	False	False	True	0	0	0
3	False	True	False	True	False	True	True	1	0	1

Года, порядковые номера кварталов, порядковые номера месяцев, порядковые номера дней года, порядковые номера дней месяца, порядковые номера дней недели, порядковые номера недель, индикаторы начала/конца года/квартала/месяца можно извлекать не только из столбца с датами, но и из индекса DatetimeIndex.

```
# загружаем данные, сразу прочитав столбец
# с датами как индекс и выполнив парсинг дат
data = pd.read_csv('Data/X5_ARIMA.csv',
                    index_col=['day'],
                    parse_dates=['day'])
```

data

day	rto
2018-01-01	5641.51
2018-01-02	17525.38
2018-01-03	22780.76
2018-01-04	23590.37
2018-01-05	29344.19
...	...
2020-12-27	81329.60
2020-12-28	96236.12
2020-12-29	105614.55
2020-12-30	148178.60
2020-12-31	88948.67

```
# выделяем из индекса ГОДА
data['year'] = data.index.year
# выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА
# КВАРТАЛОВ от 1 до 4
data['quarter'] = data.index.quarter
# выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА
# МЕСЯЦЕВ от 1 до 12
data['month'] = data.index.month
# выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА МЕСЯЦЕВ
```

```

# ДЕНЬ ГОДА от 1 до 365
data['dayofyear'] = data.index.dayofyear
# выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА
# ДЕНЬ МЕСЯЦА от 1 до 31
data['dayofmonth'] = data.index.day
# выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА
# ДЕНЬ НЕДЕЛИ от 0 до 6,
# где 0 - понедельник, 6 - воскресенье
data['dayofweek'] = data.index.dayofweek
# выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА НЕДЕЛЬ
# либо с 1 по 52, либо с 00 до 52
data['week'] = data.index.week
data['week_alter'] = data.index.strftime('%U')
# выделяем из индекса ИНДИКАТОР - начало года
data['year_start'] = data.index.is_year_start
# выделяем из индекса ИНДИКАТОР ВЫСОКОСНОГО ГОДА
data['leap_year'] = data.index.is_leap_year
# получаем из индекса, превращенного в серию временных
# меток, ПОРЯДКОВЫЙ НОМЕР НЕДЕЛИ В МЕСЯЦЕ
data['week_of_month'] = pd.Series(data.index).apply(
    week_of_month).values
# получаем из индекса, превращенного в серию временных
# меток, ПОРЯДКОВЫЙ НОМЕР ДЕКАДЫ В МЕСЯЦЕ
data['decades'] = create_decades(pd.Series(data.index))
# смотрим результаты
data

```

	rto	year	quarter	month	dayofyear	dayofmonth	dayofweek	week	week_alter	year_start	leap_year	week_of_month	decades
day													
2018-01-01	5641.51	2018	1	1	1	1	0	1	00	True	False	1	1
2018-01-02	17525.38	2018	1	1	2	2	1	1	00	False	False	1	1
2018-01-03	22780.76	2018	1	1	3	3	2	1	00	False	False	1	1
2018-01-04	23590.37	2018	1	1	4	4	3	1	00	False	False	1	1
2018-01-05	29344.19	2018	1	1	5	5	4	1	00	False	False	1	1
...
2020-12-27	81329.60	2020	4	12	362	27	6	52	52	False	True	4	3
2020-12-28	96236.12	2020	4	12	363	28	0	53	52	False	True	5	3
2020-12-29	105614.55	2020	4	12	364	29	1	53	52	False	True	5	3
2020-12-30	148178.60	2020	4	12	365	30	2	53	52	False	True	5	3
2020-12-31	88948.67	2020	4	12	366	31	3	53	52	False	True	5	3

Кроме того, полезно попробовать представить значения признаков даты в формате десятичной дроби.

```

# выделяем из индекса КВАРТАЛЫ
# в формате десятичной дроби
data['frac_quarter'] = data.index.quarter / 4
# выделяем из индекса МЕСЯЦЫ
# в формате десятичной дроби
data['frac_month'] = data.index.month / 12
# выделяем из индекса ДНИ ГОДА
# в формате десятичной дроби
data['frac_dayofyear'] = data.index.dayofyear / 365
# выделяем из индекса ДНИ МЕСЯЦА
# в формате десятичной дроби
data['frac_dayofmonth'] = data.index.day / 31

```

```
# выделяем из индекса ДНИ НЕДЕЛИ
# в формате десятичной дроби
data['frac_dayofweek'] = (
    (data['dayofweek'] + 7) - 6) / 7
data
```

day		frac_quarter	frac_month	frac_dayofyear	frac_dayofmonth	frac_dayofweek
0	2018-01-01	0.25	0.083333	0.002740	0.032258	0.142857
1	2018-01-02	0.25	0.083333	0.005479	0.064516	0.285714
2	2018-01-03	0.25	0.083333	0.008219	0.096774	0.428571
3	2018-01-04	0.25	0.083333	0.010959	0.129032	0.571429
4	2018-01-05	0.25	0.083333	0.013699	0.161290	0.714286
...
1091	2020-12-27	1.00	1.000000	0.991781	0.870968	1.000000
1092	2020-12-28	1.00	1.000000	0.994521	0.903226	0.142857
1093	2020-12-29	1.00	1.000000	0.997260	0.935484	0.285714
1094	2020-12-30	1.00	1.000000	1.000000	0.967742	0.428571
1095	2020-12-31	1.00	1.000000	1.002740	1.000000	0.571429

Кроме того, можно воспользоваться производственным календарем и добавить информацию о праздниках и предпраздничных днях. Сейчас мы проиллюстрируем, как добавлять признаки на основе производственного календаря, размещенного в файле <https://github.com/d10xa/holidays-calendar/blob/master/json/calendar.json>. Файл обновляется ежегодно, содержит информацию о праздниках и выходных ('holidays'), сокращенных рабочих днях ('reg-holidays') и днях двух локдаунов ('nowork') в период с января 2011 года по 31 декабря 2022 года.

calendar.json

```
{
    "holidays": [
        "2011-01-01",
        "2011-01-02",
        .
        .
        .
        "2022-12-25",
        "2022-12-31"
    ],
    "preholidays": [
        "2011-02-22",
        "2011-03-05",
        .
        .
        .
        "2022-03-05",
        "2022-11-03"
    ],
    "nowork": [
        "2020-03-30",
        "2020-03-31",
        .
        .
        .
        "2021-11-02",
        "2021-11-03"
    ]
}
```

Давайте запишем в две переменные стартовую дату – 1 января 2017 года и последнюю дату – 31 декабря 2021 года, а затем на их основе создадим датафрейм `calendar_df`, охватывающий период с 1 января 2017 года по 31 декабря 2021 года.

```
# создаем стартовую дату - 1 января 2017 года
start_date = datetime.date(year=2017, month=1, day=1)
# создаем последнюю дату - 31 декабря 2021 года
end_date = datetime.date(year=2021, month=12, day=31)
# создаем датафрейм на основе стартовой и последней дат
calendar_df = pd.DataFrame(
    pd.date_range(start=start_date,
                  end=end_date,
                  freq='D'), columns=['date'])
# посмотрим первую и последнюю метки времени датафрейма
print(f"Min: {calendar_df['date'].min()}" +
      f"\nMax: {calendar_df['date'].max()}")
# взглянем на первые 5 наблюдений
calendar_df.head()
```

```
Min: 2017-01-01 00:00:00
Max: 2021-12-31 00:00:00
```

	date
0	2017-01-01
1	2017-01-02
2	2017-01-03
3	2017-01-04
4	2017-01-05

Теперь загружаем данные производственного календаря из файла *calendar.json* с *holidays* – датами выходных дней и праздничных дней, *preholidays* – датами сокращенных рабочих дней, *nowork* – датами локдаунов в период COVID-пандемии.

```
# загружаем данные производственного календаря с
# holidays – датами выходных дней и праздничных дней,
# preholidays – датами сокращенных рабочих дней,
# nowork – датами локдаунов в период COVID-пандемии
with open('Data/calendar.json') as json_file:
    json_data = json.load(json_file)
```

Сейчас мы создадим пустой датафрейм, сохраним в него даты выходных и праздничных дней из JSON-файла и создадим столбец-флаг *is_holiday* со значениями – единицами.

```
# создаем пустой датафрейм
holidays_df = pd.DataFrame()
# сохраним в датафрейм даты выходных и праздничных дней
holidays_df['date'] = pd.to_datetime(
    json_data['holidays'])
# создаем столбец-флаг is_holiday со значениями – единицами
holidays_df['is_holiday'] = 1
holidays_df.head(15)
```

	date	is_holiday
0	2011-01-01	1
1	2011-01-02	1
2	2011-01-03	1
3	2011-01-04	1
4	2011-01-05	1
5	2011-01-06	1
6	2011-01-07	1
7	2011-01-08	1
8	2011-01-09	1
9	2011-01-10	1
10	2011-01-15	1
11	2011-01-16	1
12	2011-01-22	1
13	2011-01-23	1
14	2011-01-29	1

Опять создадим пустой датафрейм, сохраним в него даты сокращенных рабочих дней из JSON-файла и создадим столбец-флаг *is_preholiday* со значениями – единицами.

```
# создаем пустой датафрейм
preholydys_df = pd.DataFrame()
# сохраняем в датафрейм даты сокращенных рабочих дней
preholydys_df['date'] = pd.to_datetime(
    json_data['preholidays'])
# создаем столбец-флаг is_preholiday со значениями - единицами
preholydys_df['is_preholiday'] = 1
preholydys_df.head(15)
```

	date	is_preholiday
0	2011-02-22	1
1	2011-03-05	1
2	2011-11-03	1
3	2012-02-22	1
4	2012-03-07	1
5	2012-04-28	1
6	2012-05-12	1
7	2012-06-09	1
8	2012-12-29	1
9	2013-02-22	1
10	2013-03-07	1
11	2013-04-30	1
12	2013-05-08	1
13	2013-06-11	1
14	2013-12-31	1

Объединяем датафреймы *calendar_df*, *holydays_df* и *preholydys_df*, в результате чего получаем датафрейм, охватывающий период с 1 января 2017 года по 31 декабря 2021 года, с флагом праздничных и выходных дней и флагом сокращенных рабочих дней.

```
# присоединяем к датафрейму calendar_df датафреймы holydays_df
# и preholydys_df, в результате чего получаем датафрейм,
# охватывающий период с 1 января 2017 года по 31 декабря
# 2021 года, с флагом праздничных и выходных дней
# и флагом сокращенных рабочих дней
dfs = [calendar_df, holydays_df, preholydys_df]
calendar_df = reduce(lambda left, right: pd.merge(
    left, right, how='left',
    on='date'), dfs)
calendar_df.head(15)
```

	date	is_holiday	is_preholiday
0	2017-01-01	1.0	NaN
1	2017-01-02	1.0	NaN
2	2017-01-03	1.0	NaN
3	2017-01-04	1.0	NaN
4	2017-01-05	1.0	NaN
5	2017-01-06	1.0	NaN
6	2017-01-07	1.0	NaN
7	2017-01-08	1.0	NaN
8	2017-01-09	NaN	NaN
9	2017-01-10	NaN	NaN
10	2017-01-11	NaN	NaN
11	2017-01-12	NaN	NaN
12	2017-01-13	NaN	NaN
13	2017-01-14	1.0	NaN
14	2017-01-15	1.0	NaN

Пропуски в столбцах-флагах заполняем нулями, а сами значения столбцов переводим в целочисленный формат.

```
# пропуски в столбцах-флагах заполняем нулями,
# а сами значения столбца переводим
# в целочисленный формат
lst = ['is_holiday', 'is_preholiday']
calendar_df[lst] = calendar_df[lst].fillna(0).astype('int')
calendar_df.head(15)
```

	date	is_holiday	is_preholiday
0	2017-01-01	1	0
1	2017-01-02	1	0
2	2017-01-03	1	0
3	2017-01-04	1	0
4	2017-01-05	1	0
5	2017-01-06	1	0
6	2017-01-07	1	0
7	2017-01-08	1	0
8	2017-01-09	0	0
9	2017-01-10	0	0
10	2017-01-11	0	0
11	2017-01-12	0	0
12	2017-01-13	0	0
13	2017-01-14	1	0
14	2017-01-15	1	0

Давайте посмотрим на выходные, праздничные дни и сокращенные рабочие дни в феврале 2021 года.

```
# посмотрим на выходные, праздничные дни и
# сокращенные рабочие дни в феврале 2021
calendar_df[(calendar_df['date'] >= '2021-02-01') &
             (calendar_df['date'] < '2021-02-28')]
```

	date	is_holiday	is_preholiday
1492	2021-02-01	0	0
1493	2021-02-02	0	0
1494	2021-02-03	0	0
1495	2021-02-04	0	0
1496	2021-02-05	0	0
1497	2021-02-06	1	0
1498	2021-02-07	1	0
1499	2021-02-08	0	0
1500	2021-02-09	0	0
1501	2021-02-10	0	0
1502	2021-02-11	0	0
1503	2021-02-12	0	0
1504	2021-02-13	1	0
1505	2021-02-14	1	0
1506	2021-02-15	0	0
1507	2021-02-16	0	0
1508	2021-02-17	0	0
1509	2021-02-18	0	0
1510	2021-02-19	0	0
1511	2021-02-20	0	1
1512	2021-02-21	1	0
1513	2021-02-22	1	0
1514	2021-02-23	1	0
1515	2021-02-24	0	0
1516	2021-02-25	0	0
1517	2021-02-26	0	0
1518	2021-02-27	1	0

сокращенный рабочий день

выходной день

перенесенный выходной день

праздничный выходной день

Теперь приведем пример того, как можно выделить признаки времени – порядковый номер трети суток, порядковый номер четверти суток, порядковый номер часа, порядковый номер 30-минутного интервала, порядковый номер 15-минутного интервала, порядковый номер минуты, часть суток (утро, день, вечер, ночь).

```
# записываем датафрейм на основе CSV-файла,
# содержащего даты и время
data = pd.read_csv('Data/pickup_dates.csv')
data['pickup_datetime'] = pd.to_datetime(
    data['pickup_datetime'])
data.head()
```

```
pickup_datetime
0 2016-03-14 17:24:00
1 2016-12-06 00:43:00
2 2016-01-19 11:35:00
3 2016-06-04 19:32:00
4 2016-03-26 13:30:00

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР ПОЛОВИНЫ СУТОК
data['half_of_day'] = data['pickup_datetime'].apply(
    lambda x: x.hour // 12)

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР ТРЕТИ СУТОК
data['third_of_day'] = data['pickup_datetime'].apply(
    lambda x: x.hour // 8)

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР ЧЕТВЕРТИ СУТОК
data['quarter_of_day'] = data['pickup_datetime'].apply(
    lambda x: x.hour // 6)

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР ЧАСА
data['hour'] = pd.DatetimeIndex(
    data['pickup_datetime']).hour
data['hour_alter'] = data['pickup_datetime'].dt.hour

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР 30-МИНУТНОГО ИНТЕРВАЛА В ЧАСЕ
data['30_min_interval'] = data['pickup_datetime'].apply(
    lambda x: x.minute // 30)

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР 15-МИНУТНОГО ИНТЕРВАЛА В ЧАСЕ
data['15_min_interval'] = data['pickup_datetime'].apply(
    lambda x: x.minute // 15)

# выделяем из переменной pickup_datetime
# ПОРЯДКОВЫЙ НОМЕР МИНУТЫ
data['minute'] = pd.DatetimeIndex(
    data['pickup_datetime']).minute
data['minute_alter'] = data['pickup_datetime'].dt.minute

# пишем функцию для создания переменной - ЧАСТЬ СУТОК
def get_part_of_day(hour):
    return (
        'morning' if 5 <= hour <= 11
        else
        'afternoon' if 12 <= hour <= 17
        else
        'evening' if 18 <= hour <= 23
        else
        'night'
    )
```

```
# создаем переменную - ЧАСТЬ СУТОК
data['part_of_day'] = data['hour'].apply(
    lambda x: get_part_of_day(x))
data
```

	pickup_datetime	half_of_day	third_of_day	quarter_of_day	hour	hour_alter	30_min_interval	15_min_interval	minute	minute_alter	part_of_day
0	2016-03-14 17:24:00	1	2	2	17	17	0	1	24	24	afternoon
1	2016-12-06 00:43:00	0	0	0	0	0	1	2	43	43	night
2	2016-01-19 11:35:00	0	1	1	11	11	1	2	35	35	morning
3	2016-06-04 19:32:00	1	2	3	19	19	1	2	32	32	evening
4	2016-03-26 13:30:00	1	1	2	13	13	1	2	30	30	afternoon
5	2016-01-30 22:01:00	1	2	3	22	22	0	0	1	1	evening

Теперь покажем, как можно создавать признаки даты и времени в Polars.

```
# записываем датафрейм Polars на основе CSV-файла,
# содержащего ежедневные продажи лука
polars_data = pl.read_csv('Data/X5_ARIMA.csv',
                           parse_dates=True)
polars_data

shape: (1096, 2)
```

day	rto
date	f64
2018-01-01	5641.51
2018-01-02	17525.38
2018-01-03	22780.76
2018-01-04	23590.37
2018-01-05	29344.19
2018-01-06	33302.34
2018-01-07	27007.21
2018-01-08	29806.28
2018-01-09	16021.67
2018-01-10	37215.22
2018-01-11	35629.0
2018-01-12	45641.58
...	...
2020-12-20	63515.62
2020-12-21	74395.04
2020-12-22	63112.09
2020-12-23	69393.19
2020-12-24	73657.01
2020-12-25	78295.61
2020-12-26	79701.82
2020-12-27	81329.6
2020-12-28	96236.12
2020-12-29	105614.55
2020-12-30	148178.6
2020-12-31	88948.67

```
# создаем признаки даты
```

```
polars_data = polars_data.with_columns([
    (pl.col('day').dt.year()).alias('year'),
    (pl.col('day').dt.quarter()).alias('quarter'),
    (pl.col('day').dt.month()).alias('month'),
    (pl.col('day').dt.strftime('%b')).alias('month_name'),
    (pl.col('day').dt.week()).alias('week'),
    (pl.col('day').dt.strftime('%U')).alias('week_alter'),
    (pl.col('day').dt.weekday()).alias('weekday'),
    (pl.col('day').dt.day()).alias('dayofmonth'),
    (pl.col('day').dt.ordinal_day().alias('dayofyear'))
])
polars_data = polars_data.with_columns([
    (pl.col('weekday').apply(lambda x: dayNameFromWeekday(x))).alias(
        'weekday_name'),
    (pl.col('day').apply(lambda x: week_of_month(x))).alias('week_of_month'),
    (pl.col('month').apply(lambda x: get_season(x))).alias('season'),
    (pl.when(pl.col('weekday') > 5).then(1).otherwise(0)).alias('weekend')
])
polars_data
```

day	rto	year	quarter	month	month_name	week	week_alter	weekday	dayofmonth	dayofyear	weekday_name	week_of_month	season	weekend	
date	f64	i32	u32	u32	str	u32	str	u32	u32	u32	str	i64	str	i64	
2018-01-01	5641.51	2018		1	1	"Jan"	1	"01"	0	1	1	"Monday"	1	"WINTER"	0
2018-01-02	17525.38	2018		1	1	"Jan"	1	"01"	1	2	2	"Tuesday"	1	"WINTER"	0
2018-01-03	22780.76	2018		1	1	"Jan"	1	"01"	2	3	3	"Wednesday"	1	"WINTER"	0
2018-01-04	23590.37	2018		1	1	"Jan"	1	"01"	3	4	4	"Thursday"	1	"WINTER"	0
2018-01-05	29344.19	2018		1	1	"Jan"	1	"01"	4	5	5	"Friday"	1	"WINTER"	0
2018-01-06	33302.34	2018		1	1	"Jan"	1	"01"	5	6	6	"Saturday"	1	"WINTER"	0
2018-01-07	27007.21	2018		1	1	"Jan"	1	"02"	6	7	7	"Sunday"	1	"WINTER"	1
2018-01-08	29806.28	2018		1	1	"Jan"	2	"02"	0	8	8	"Monday"	2	"WINTER"	0
2018-01-09	16021.67	2018		1	1	"Jan"	2	"02"	1	9	9	"Tuesday"	2	"WINTER"	0
2018-01-10	37215.22	2018		1	1	"Jan"	2	"02"	2	10	10	"Wednesday"	2	"WINTER"	0
2018-01-11	35629.0	2018		1	1	"Jan"	2	"02"	3	11	11	"Thursday"	2	"WINTER"	0
2018-01-12	45641.58	2018		1	1	"Jan"	2	"02"	4	12	12	"Friday"	2	"WINTER"	0
...	
2020-12-20	63515.62	2020	4	12	"Dec"	51	"51"	6	20	355	"Sunday"	3	"WINTER"	1	
2020-12-21	74395.04	2020	4	12	"Dec"	52	"51"	0	21	356	"Monday"	4	"WINTER"	0	
2020-12-22	63112.09	2020	4	12	"Dec"	52	"51"	1	22	357	"Tuesday"	4	"WINTER"	0	
2020-12-23	69393.19	2020	4	12	"Dec"	52	"51"	2	23	358	"Wednesday"	4	"WINTER"	0	
2020-12-24	73657.01	2020	4	12	"Dec"	52	"51"	3	24	359	"Thursday"	4	"WINTER"	0	
2020-12-25	78295.61	2020	4	12	"Dec"	52	"51"	4	25	360	"Friday"	4	"WINTER"	0	
2020-12-26	79701.82	2020	4	12	"Dec"	52	"51"	5	26	361	"Saturday"	4	"WINTER"	0	
2020-12-27	81329.6	2020	4	12	"Dec"	52	"52"	6	27	362	"Sunday"	4	"WINTER"	1	
2020-12-28	96236.12	2020	4	12	"Dec"	53	"52"	0	28	363	"Monday"	5	"WINTER"	0	
2020-12-29	105614.55	2020	4	12	"Dec"	53	"52"	1	29	364	"Tuesday"	5	"WINTER"	0	
2020-12-30	148178.6	2020	4	12	"Dec"	53	"52"	2	30	365	"Wednesday"	5	"WINTER"	0	
2020-12-31	88948.67	2020	4	12	"Dec"	53	"52"	3	31	366	"Thursday"	5	"WINTER"	0	

```
# записываем датафрейм Polars на основе CSV-файла,
# содержащего даты и время
```

```
polars_data = pl.read_csv('Data/pickup_dates.csv',
                          parse_dates=True)
# присваиваем переменной pickup_datetime
# mun Datetime и нужный формат
polars_data = polars_data.with_column(
    pl.col('pickup_datetime').str.strptime(
        pl.Datetime, fmt='%d.%m.%Y %H:%M'))
```

polars_data

shape: (6, 1)

```
pickup_datetime
datetime[us]
2016-03-14 17:24:00
2016-06-12 00:43:00
2016-01-19 11:35:00
2016-04-06 19:32:00
2016-03-26 13:30:00
2016-01-30 22:01:00

# создаем признаки времени
polars_data = polars_data.with_columns([
    (pl.col('pickup_datetime').apply(
        lambda x: x.hour // 12)).alias('half_of_day'),
    (pl.col('pickup_datetime').apply(
        lambda x: x.hour // 8)).alias('third_of_day'),
    (pl.col('pickup_datetime').apply(
        lambda x: x.hour // 6)).alias('quarter_of_day'),
    (pl.col('pickup_datetime').dt.hour()).alias('hour'),
    (pl.col('pickup_datetime').apply(
        lambda x: x.minute // 30)).alias('30_min_interval'),
    (pl.col('pickup_datetime').apply(
        lambda x: x.minute // 15)).alias('15_min_interval'),
    (pl.col('pickup_datetime').dt.minute()).alias('minute')
])
polars_data = polars_data.with_column(
    (pl.col('hour').apply(
        lambda x: get_part_of_day(x))).alias('part_of_day'))
polars_data
```

shape: (6, 9)

	pickup_datetime	half_of_day	third_of_day	quarter_of_day	hour	30_min_interval	15_min_interval	minute	part_of_day
	datetime[us]	i64	i64	i64	u32	i64	i64	u32	str
2016-03-14 17:24:00		1	2	2	17	0	1	24	"afternoon"
2016-06-12 00:43:00		0	0	0	0	1	2	43	"night"
2016-01-19 11:35:00		0	1	1	11	1	2	35	"morning"
2016-04-06 19:32:00		1	2	3	19	1	2	32	"evening"
2016-03-26 13:30:00		1	1	2	13	1	2	30	"afternoon"
2016-01-30 22:01:00		1	2	3	22	0	0	1	"evening"

16.4.9. Компоненты ряда Фурье

Для моделирования сезонностей, помимо календарных признаков, можно использовать компоненты ряда Фурье.

Давайте напишем функцию, вычисляющую компоненты ряда Фурье.

```
# пишем функцию, которая вычисляет компоненты ряда Фурье
def fourier(series_index, period=None, order=None,
            mods=None, out_column=None):
    """
    Вычисляет компоненты ряда Фурье.
```

Параметры

series_index:

 Индекс временного ряда

period:

 Период сезонности, задаваемый в зависимости
 от частоты ряда, должен быть ≥ 2

order:

 Верхняя граница порядка компонент ряда Фурье для включения,
 должна быть ≥ 1 и $\leq \text{ceil}(\text{period}/2)$

mods:

 Альтернативный и точный способ задать количество
 используемых гармоник, например `mods=[1, 3, 4]`
 означает, что будет использован синус первого порядка
 и синус и косинус второго порядка,
 `mods` должен быть ≥ 1 и $< \text{period}$

out_column:

 Названия признаков

Замечания

Для понимания, как вычисляются компоненты ряда Фурье, см.
<https://otexts.com/fpp2/useful-predictors.html#fourier-series>

* Параметр `period` отвечает за моделирование сезонности.

* Параметры `order` и `mods` определяют количество гармоник.

Параметр `order` – это упрощенный вариант параметра `mods`.
Например, `order=2` можно записать как `mods=[1, 2, 3]`,
если `period` > 4 , и как `mods=[1, 2, 3]`,
если $3 \leq \text{period} \leq 4$.

"""

```
def _get_column_name(mod):
    if out_column is None:
        return f"Fourier_{mod}"
    else:
        return f"{out_column}_{mod}"

if period < 2:
    raise ValueError("Period should be at least 2")

if order is not None and mods is None:
    if order < 1 or order > ceil(period / 2):
        raise ValueError("Order should be within" +
                         "[1, ceil(period/2)] range")
    mods = [mod for mod in range(
        1, 2 * order + 1) if mod < period]
elif mods is not None and order is None:
    if min(mods) < 1 or max(mods) >= period:
        raise ValueError("Every mod should be within " +
                         "[1, int(period)) range")
else:
    raise ValueError("There should be exactly one" +
                     "option set: order or mods")

features = pd.DataFrame(index=series_index)
elapsed = np.arange(features.shape[0]) / period
```

```

for mod in mods:
    order = (mod + 1) // 2
    is_cos = mod % 2 == 0
    features[_get_column_name(mod)] = np.sin(
        2 * np.pi * order * elapsed + np.pi / 2 * is_cos)
return features

```

Допустим, у нас есть ежедневный временной ряд длиной в год. Мы создадим индекс, состоящий из 365 временных меток.

```

# создаем индекс, состоящий из 365 временных меток (т. е. длиной в год)
indx = pd.date_range(start='2021-01-01', periods=365)
indx

DatetimeIndex(['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04',
               '2021-01-05', '2021-01-06', '2021-01-07', '2021-01-08',
               '2021-01-09', '2021-01-10',
               ...
               '2021-12-22', '2021-12-23', '2021-12-24', '2021-12-25',
               '2021-12-26', '2021-12-27', '2021-12-28', '2021-12-29',
               '2021-12-30', '2021-12-31'],
              dtype='datetime64[ns]', length=365, freq='D')

```

Теперь вычислим компоненты ряда Фурье для моделирования годовой сезонности вплоть до третьего порядка: синус и косинус первого порядка, синус и косинус второго порядка, синус и косинус третьего порядка. Для соответствующей сезонности указываем соответствующий период.

Ниже приведена таблица, которая позволит подобрать период для соответствующей сезонности.

Сезонный цикл

Частота ряда	Неделя	Месяц	Квартал	Год
Ежеквартальный				4
Ежемесячный			3	12
Еженедельный		4.348125	13.044375	52.1775
Ежедневный	7	30.436875	91.310625	365.2425

Мы имеем дело с ежедневным рядом. Если мы предполагаем, что наш ряд демонстрирует годовую сезонность, то для `period` задаем значение 365,24. С помощью параметра `order` мы подбираем верхнюю границу порядка компонент ряда Фурье, по сути, количество гармоник в зависимости от паттерна годовой сезонности. Хорошим компромиссным значением является 3. Чем больше значение параметра `order`, тем более сложный, более изменчивый паттерн годовой сезонности мы предполагаем.

```

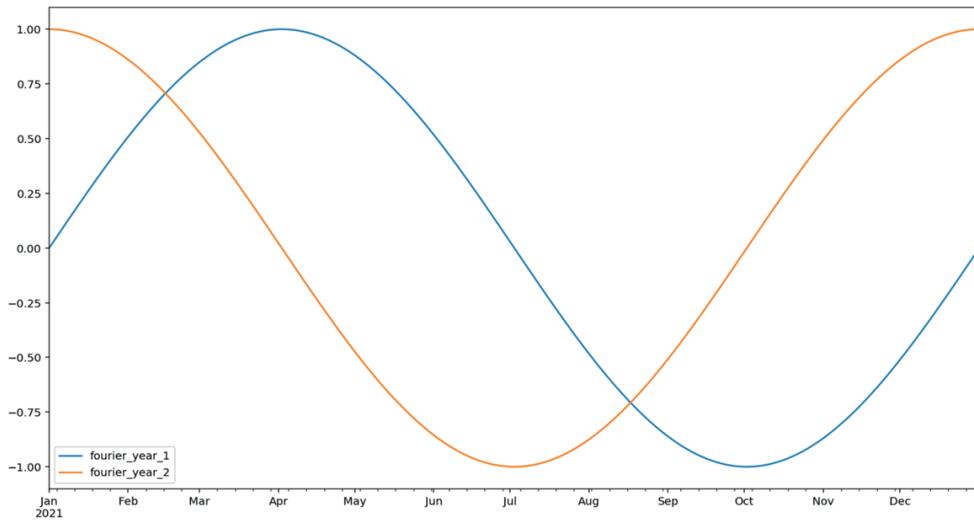
# вычисляем компоненты ряда Фурье для моделирования
# годовой сезонности вплоть до третьего порядка:
# синус и косинус первого порядка, синус и
# косинус второго порядка, синус и косинус
# третьего порядка
fourier_year = fourier(indx,
                       out_column='fourier_year',
                       period=365.24, order=3)
fourier_year.head()

```

	fourier_year_1	fourier_year_2	fourier_year_3	fourier_year_4	fourier_year_5	fourier_year_6
2021-01-01	0.00000	1.00000	0.00000	1.00000	0.00000	1.00000
2021-01-02	0.01720	0.99985	0.03440	0.99941	0.05159	0.99867
2021-01-03	0.03440	0.99941	0.06876	0.99763	0.10303	0.99468
2021-01-04	0.05159	0.99867	0.10303	0.99468	0.15421	0.98804
2021-01-05	0.06876	0.99763	0.13719	0.99054	0.20497	0.97877

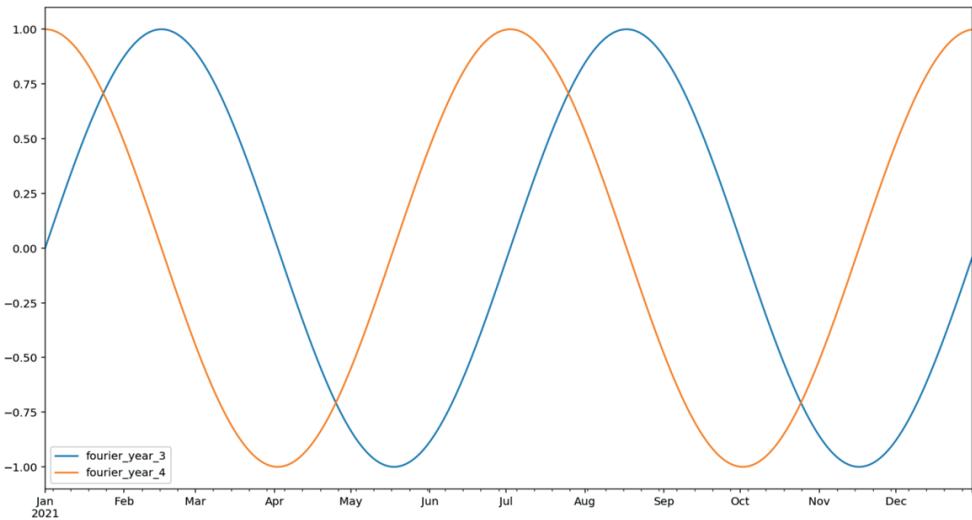
Давайте визуализируем компоненты ряда Фурье – синус и косинус *первого* порядка для моделирования годовой сезонности.

```
# визуализируем компоненты ряда Фурье – синус и косинус первого
# порядка для моделирования годовой сезонности
fourier_year_lst = ['fourier_year_1', 'fourier_year_2']
for i in fourier_year_lst:
    fourier_year[i].plot(figsize=(15, 8), legend=True)
```



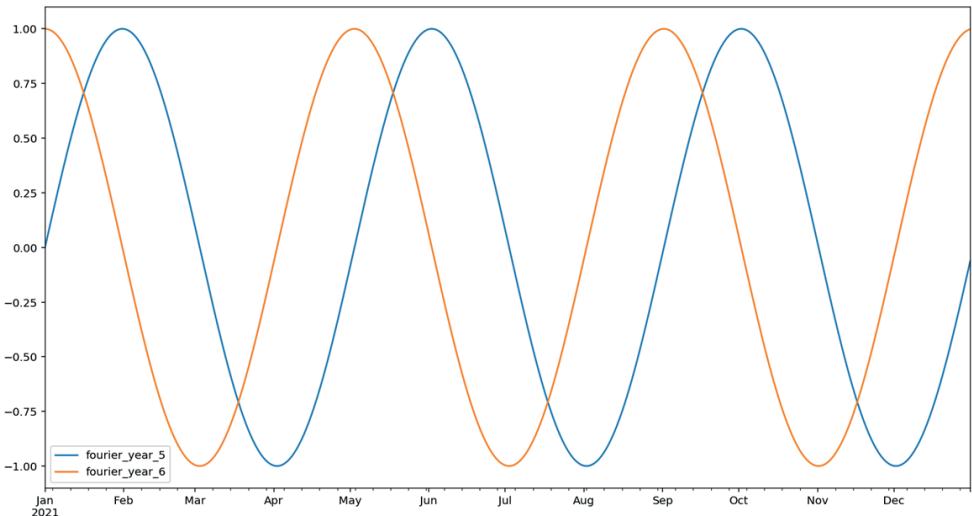
Теперь визуализируем компоненты ряда Фурье – синус и косинус *второго* порядка для моделирования годовой сезонности.

```
# визуализируем компоненты ряда Фурье – синус и косинус второго
# порядка для моделирования годовой сезонности
fourier_year_lst = ['fourier_year_3', 'fourier_year_4']
for i in fourier_year_lst:
    fourier_year[i].plot(figsize=(15, 8), legend=True)
```



Теперь визуализируем компоненты ряда Фурье – синус и косинус *третьего* порядка для моделирования годовой сезонности.

```
# визуализируем компоненты ряда Фурье – синус и косинус третьего
# порядка для моделирования годовой сезонности
fourier_year_lst = ['fourier_year_5', 'fourier_year_6']
for i in fourier_year_lst:
    fourier_year[i].plot(figsize=(15, 8), legend=True)
```



Видим, что с увеличением порядка увеличивается частота колебаний волны и моделируется более сложный паттерн годовой сезонности.

Теперь с помощью параметра `mod` выборочно создадим компоненты ряда Фурье для моделирования годовой сезонности. Возьмем синус первого порядка, косинус и синус второго порядка.

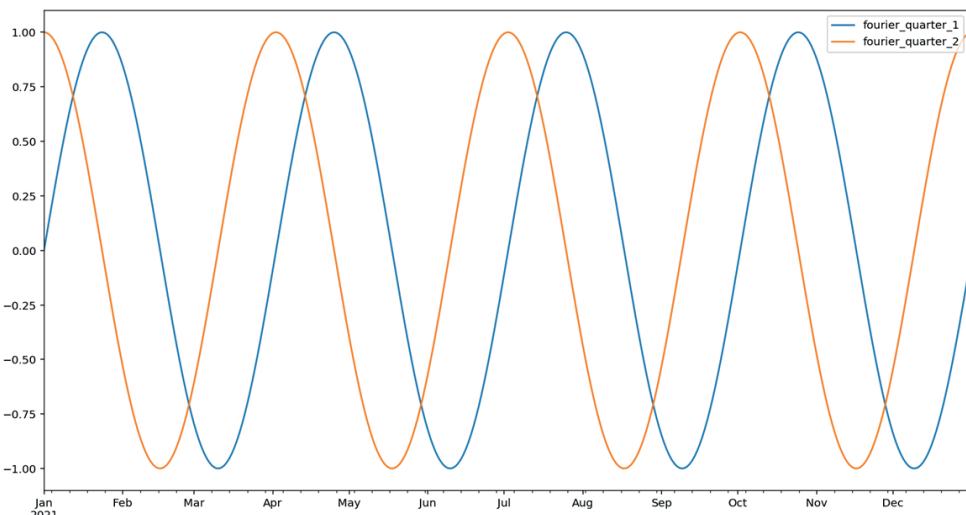
```
# вычисляем компоненты ряда Фурье для моделирования
# годовой сезонности, берем синус первого
# порядка, косинус и синус второго порядка
fourier_year = fourier(indx,
                       out_column='fourier_year',
                       period=365.24, mods=[1, 3, 4])
fourier_year.head()
```

	fourier_year_1	fourier_year_3	fourier_year_4
2021-01-01	0.00000	0.00000	1.00000
2021-01-02	0.01720	0.03440	0.99941
2021-01-03	0.03440	0.06876	0.99763
2021-01-04	0.05159	0.10303	0.99468
2021-01-05	0.06876	0.13719	0.99054

Давайте вычислим и визуализируем компоненты ряда Фурье первого порядка для моделирования квартальной сезонности.

```
# вычисляем компоненты ряда Фурье первого порядка
# для моделирования квартальной сезонности
fourier_quarter = fourier(indx,
                           out_column='fourier_quarter',
                           period=91.31, order=1)

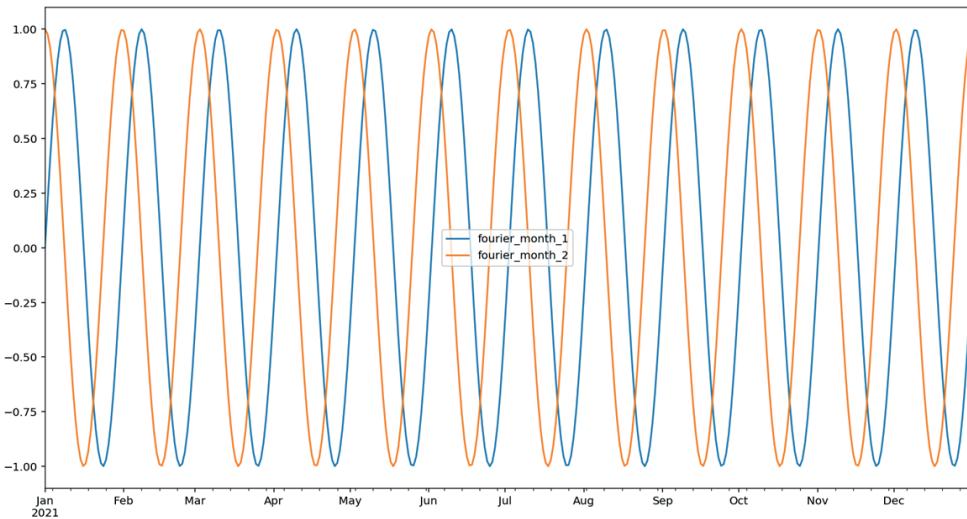
# визуализируем члены ряда Фурье первого порядка
# для моделирования квартальной сезонности
fourier_quarter_lst = fourier_quarter.columns.tolist()
for i in fourier_quarter_lst:
    fourier_quarter[i].plot(figsize=(15, 8), legend=True)
```



Вычислим и визуализируем компоненты ряда Фурье первого порядка для месячной сезонности.

```
# вычисляем компоненты ряда Фурье первого порядка
# для моделирования месячной сезонности
fourier_month = fourier(idx,
                        out_column='fourier_month',
                        period=30.44, order=1)

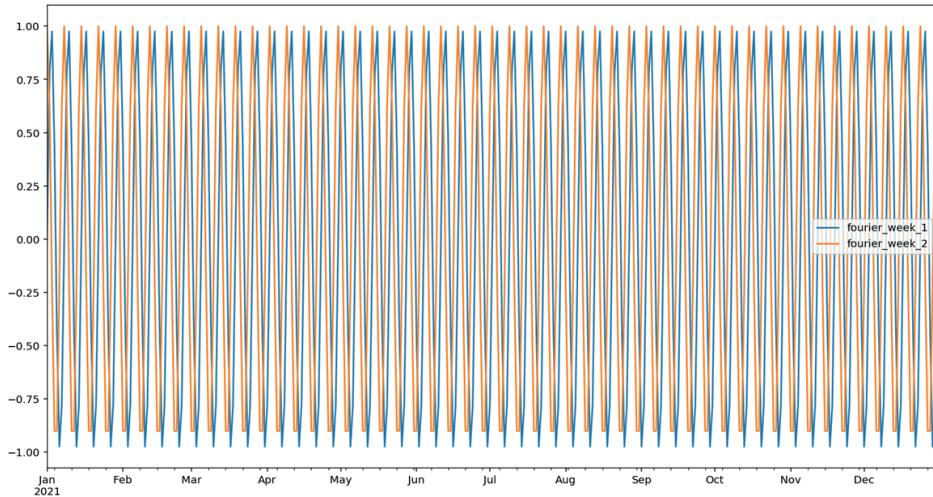
# визуализируем члены ряда Фурье первого порядка
# для моделирования месячной сезонности
fourier_month_lst = fourier_month.columns.tolist()
for i in fourier_month_lst:
    fourier_month[i].plot(figsize=(15, 8), legend=True)
```



Вычислим и визуализируем компоненты ряда Фурье первого порядка для недельной сезонности.

```
# вычисляем компоненты ряда Фурье первого порядка
# для моделирования недельной сезонности
fourier_week = fourier(idx,
                        out_column='fourier_week',
                        period=7, order=1)

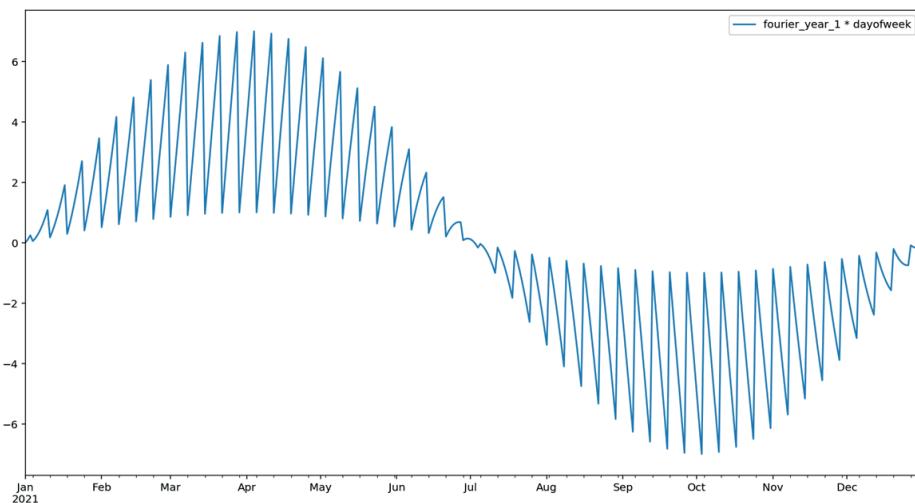
# визуализируем компоненты ряда Фурье первого порядка
# для моделирования недельной сезонности
fourier_week_lst = fourier_week.columns.tolist()
for i in fourier_week_lst:
    fourier_week[i].plot(figsize=(15, 8), legend=True)
```



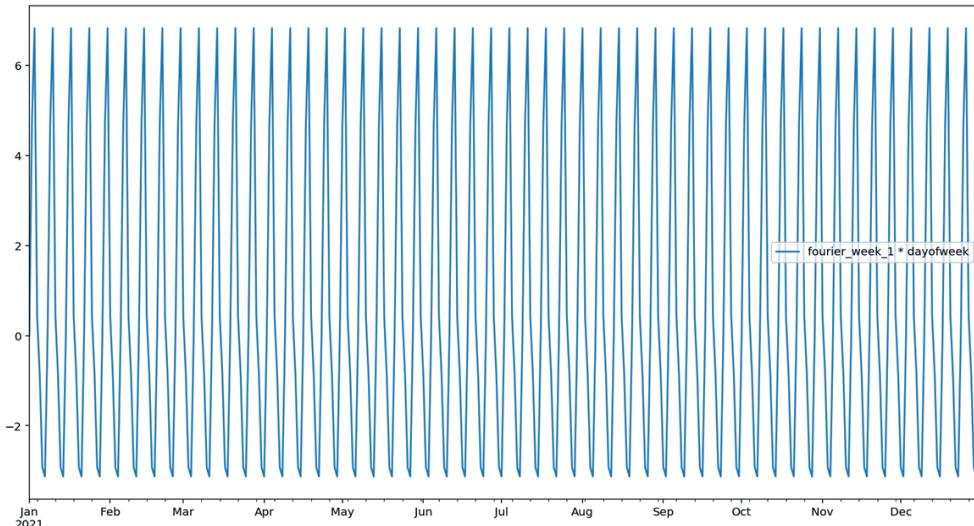
Кроме того, для моделирования смесей сезонностей полезно создавать произведения компонент ряда Фурье и календарных признаков (поскольку значения некоторых календарных признаков вычисляются с 0, позаботьтесь о том, чтобы они начинались с 1).

```
# создаем произведение компонент ряда Фурье и дня недели
fourier_year['fourier_year_1 * dayofweek'] = (
    fourier_year['fourier_year_1'] * (fourier_year.index.dayofweek + 1))
fourier_week['fourier_week_1 * dayofweek'] = (
    fourier_week['fourier_week_1'] * (fourier_week.index.dayofweek + 1))

# визуализируем произведение синуса первого порядка
# с периодом 365.24 (для годовой сезонности) и
# дня недели
fourier_year['fourier_year_1 * dayofweek'].plot(
    figsize=(15, 8), legend=True);
```



```
# визуализируем произведение синуса первого порядка
# с периодом 7 (для недельной сезонности) и
# дня недели
fourier_week['fourier_week_1 * dayofweek'].plot(
    figsize=(15, 8), legend=True);
```



16.4.10. Признаки на основе тройного экспоненциального сглаживания

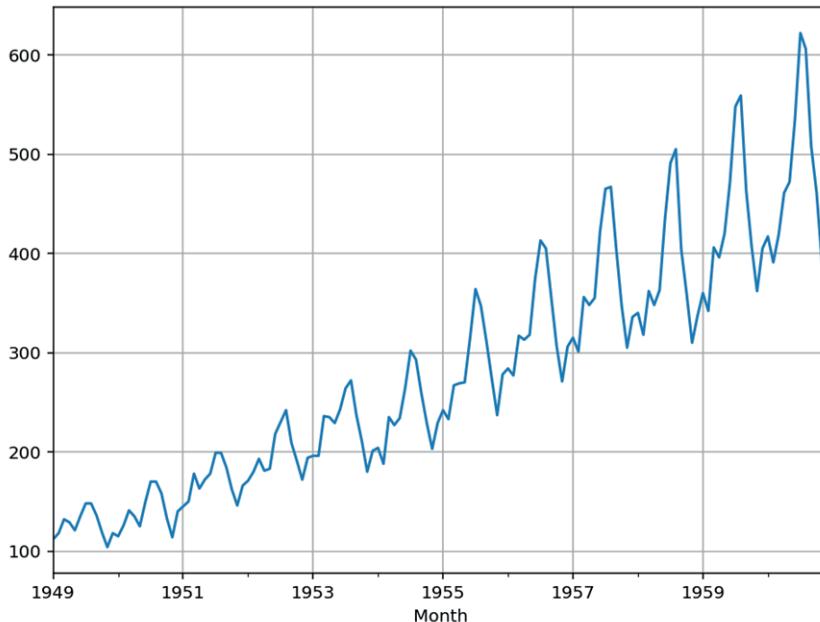
Модель тройного экспоненциального сглаживания можно использовать в качестве самостоятельной прогнозной модели, а можно применить для создания признаков, в свою очередь, эти признаки можно подать на вход другой модели, например модели градиентного бустинга. Данный подход используется в сети магазинов Walmart. Давайте загрузим ежемесячные данные о количестве авиапассажиров.

```
# загружаем ежемесячные данные
# о количестве авиапассажиров
df = pd.read_csv('Data/AirPassengers.csv',
                  header=0, index_col=0,
                  parse_dates=True)
df.head()
```

#Passengers	
Month	
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

Давайте визуализируем временной ряд.

```
# визуализируем ряд
plt.figure(figsize=(8, 6))
df['#Passengers'].plot()
plt.grid()
```



Мы видим аддитивный тренд и мультипликативную сезонность.

Сейчас напишем функцию конструирования календарных признаков.

```
# пишем функцию конструирования
# календарных признаков
def create_calendar_vars(df):

    # создаем переменную - месяц
    df['month'] = df.index.month

    # пишем функцию для создания переменной - сезон
    def get_season(month):
```

```

if (month > 11 or month < 3):
    return 'WINTER'
elif (month == 3 or month <= 5):
    return 'SPRING'
elif (month >= 6 and month < 9):
    return 'SUMMER'
else:
    return 'FALL'

# создаем переменную - сезон
df['season'] = df['month'].apply(
    lambda x: get_season(x))
# создаем переменную - квартал
df['quarter'] = df.index.quarter

return df

```

Создаем календарные признаки – месяц, сезон и квартал.

```

# создаем календарные признаки
df = create_calendar_vars(df)
df.head()

```

	#Passengers	month	season	quarter
Month				
1949-01-01	112	1	WINTER	1
1949-02-01	118	2	WINTER	1
1949-03-01	132	3	SPRING	1
1949-04-01	129	4	SPRING	2
1949-05-01	121	5	SPRING	2

Пишем функцию для разбиения на обучающие и тестовые массив признаков и массив меток.

```

# пишем функцию для разбиения на обучающие и тестовые
# массив признаков и массив меток
def fast_train_test_split(data, target, test_size):
    """

```

Разбивает набор данных на обучающие и тестовые
массив признаков и массив меток.

Параметры

data: pandas.DataFrame

Набор данных.

target: string

Имя зависимой переменной.

test_size:

Размер тестовой выборки

(горизонт прогнозирования).

"""

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(target, axis=1),
```

```

        data[target],
        test_size=test_size,
        shuffle=False)
    return X_train, X_test, y_train, y_test
    
```

Задаем горизонт прогнозирования – 36 месяцев.

```
# задаем горизонт прогнозирования
HORIZON = 36
```

Создаем с помощью нашей функции обучающие и тестовые массив признаков и массив меток.

```
# формируем обучающие и тестовые
# массив признаков и массив меток
X_train, X_test, y_train, y_test = fast_train_test_split(
    df, '#Passengers', HORIZON)
```

Теперь мы воспользуемся классом `ExponentialSmoothing` библиотеки `statsmodels` и создадим признак – обучим модель тройного экспоненциального сглаживания, параметры `smoothing_level`, `smoothing_trend` и `smoothing_seasonal` уже предварительно подобраны. В качестве значений признака для обучающей выборки используем сглаженные значения, полученные с помощью атрибута `fittedvalues`. В качестве значений признака для тестовой выборки используем прогнозы, полученные с помощью метода `forecast()`. Для понимания, как вычисляются сглаженные значения и прогнозы для различных сочетаний тренда и сезонности в моделях тройного экспоненциального сглаживания, можно воспользоваться таксономией.

Сезонность			
Тренд	Аддитивная	Мультипликативная	
Без тренда	$\ell_t = \alpha(y_t - l_{t-L}) + (1 - \alpha)\ell_{t-1}$ $b_t =$ $I_t = \gamma(y_t - \ell_{t-1}) + (1 - \gamma)l_{t-L}$ $S_t = \ell_{t-1} + b_{t-L}$ $F_{t+m} = \ell_t + I_{t-L+1+(m-1)}$	$\ell_t = \alpha(y_t/l_{t-L}) + (1 - \alpha)\ell_{t-1}$ $b_t =$ $I_t = \gamma(y_t/\ell_{t-1}) + (1 - \gamma)l_{t-L}$ $S_t = \ell_{t-1} \cdot l_{t-L}$ $F_{t+m} = \ell_t \cdot I_{t-L+1+(m-1)}$	
Аддитивный	$\ell_t = \alpha(y_t - l_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$ $I_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1 - \gamma)l_{t-L}$ $S_t = \ell_{t-1} + b_{t-1} + l_{t-L}$ $F_{t+m} = \ell_t + mb_t + I_{t-L+1+(m-1)}$	$\ell_t = \alpha(y_t/l_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$ $I_t = \gamma(y_t/(\ell_{t-1} + b_{t-1})) + (1 - \gamma)l_{t-L}$ $S_t = (\ell_{t-1} + b_{t-1}) \cdot l_{t-L}$ $F_{t+m} = (\ell_t + mb_t) \cdot I_{t-L+1+(m-1)}$	
Аддитивный с затуханием	$\ell_t = \alpha(y_t - l_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$ $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)\phi b_{t-1}$ $I_t = \gamma(y_t - \ell_{t-1} - \phi b_{t-1}) + (1 - \gamma)l_{t-L}$ $S_t = \ell_{t-1} + \phi b_{t-1} + l_{t-L}$ $F_{t+m} = \ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t + I_{t-L+1+(m-1)}$	$\ell_t = \alpha(y_t/l_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$ $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)\phi b_{t-1}$ $I_t = \gamma(y_t/(\ell_{t-1} + \phi b_{t-1})) + (1 - \gamma)l_{t-L}$ $S_t = (\ell_{t-1} + \phi b_{t-1}) \cdot l_{t-L}$ $F_{t+m} = (\ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t) \cdot I_{t-L+1+(m-1)}$	
Мультипликативный	$\ell_t = \alpha(y_t - l_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$ $b_t = \beta(\ell_t/l_{t-1}) + (1 - \beta)b_{t-1}$ $I_t = \gamma(y_t - \ell_{t-1} \cdot b_{t-1}) + (1 - \gamma)l_{t-L}$ $S_t = (\ell_{t-1} \cdot b_{t-1}) + l_{t-L}$ $F_{t+m} = \ell_t \cdot b_t^m + I_{t-L+1+(m-1)}$	$\ell_t = \alpha(y_t/l_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$ $b_t = \beta(\ell_t/l_{t-1}) + (1 - \beta)b_{t-1}$ $I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1})) + (1 - \gamma)l_{t-L}$ $S_t = (\ell_{t-1} \cdot b_{t-1}) \cdot l_{t-L}$ $F_{t+m} = \ell_t \cdot b_t^m \cdot I_{t-L+1+(m-1)}$	
Мультипликативный с затуханием	$\ell_t = \alpha(y_t - l_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$ $b_t = \beta(\ell_t/l_{t-1}) + (1 - \beta)b_{t-1}^\phi$ $I_t = \gamma(y_t - \ell_{t-1} \cdot b_{t-1}^\phi) + (1 - \gamma)l_{t-L}$ $S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) + l_{t-L}$ $F_{t+m} = (\ell_t \cdot (b_t^{(\phi+\phi^2+\dots+\phi^m)}) + I_{t-L+1+(m-1)}$	$\ell_t = \alpha(y_t/l_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$ $b_t = \beta(\ell_t/l_{t-1}) + (1 - \beta)b_{t-1}^\phi$ $I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1}^\phi)) + (1 - \gamma)l_{t-L}$ $S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) \cdot l_{t-L}$ $F_{t+m} = (\ell_t \cdot (b_t^{(\phi+\phi^2+\dots+\phi^m)})) \cdot I_{t-L+1+(m-1)}$	

где:

- ℓ_t – значение уровня для момента времени t ;
- b_t – значение тренда для момента времени t ;
- α – слаживающая константа для уровня;
- β – слаживающая константа для тренда;
- γ – слаживающая константа для сезонности;
- ϕ – коэффициент затухания;
- I_t – значение индекса сезонности для момента времени t ;
- L – количество периодов;
- m – количество периодов, на которое делается прогноз.

```
# обучаем модель тройного экспоненциального сглаживания,
# параметры уже предварительно подобраны
triple = ExponentialSmoothing(
    y_train,
    trend='additive',
    seasonal='multiplicative',
    seasonal_periods=12,
    freq='MS').fit(
        smoothing_level=0,
        smoothing_trend=0.4,
        smoothing_seasonal=1,
        optimized=False)
# получаем сглаженные значения и прогнозы
train_preds = triple.fittedvalues
test_preds = triple.forecast(HORIZON)
```

Создаем признаки на основе тройного экспоненциального сглаживания.

```
# создаем признаки на основе тройного
# экспоненциального сглаживания
X_train['ets_feature'] = train_preds
X_test['ets_feature'] = test_preds
X_train
```

	month	season	quarter	ets_feature
Month				
1949-01-01	1	WINTER	1	112.61211
1949-02-01	2	WINTER	1	119.28980
1949-03-01	3	SPRING	1	134.16424
1949-04-01	4	SPRING	2	131.82007
1949-05-01	5	SPRING	2	124.30647
...
1957-08-01	8	SUMMER	3	422.67438
1957-09-01	9	SUMMER	3	370.43622
1957-10-01	10	FALL	4	319.25754
1957-11-01	11	FALL	4	282.69892
1957-12-01	12	WINTER	4	319.16250

108 rows × 4 columns

Создаем массив индексов категориальных признаков.

```
# создаем массив индексов категориальных признаков
categorical_features_indices = np.where(
    X_train.dtypes == np.object)[0]
categorical_features_indices
array([1])
```

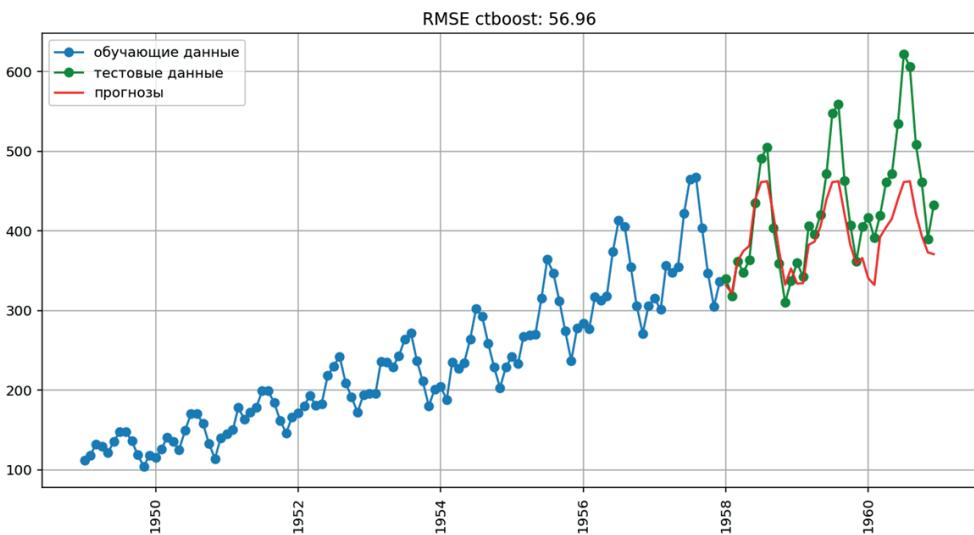
Теперь обучаем модель CatBoost, получаем прогнозы и визуализируем их.

```
# обучаем модель CatBoost
ctbst = CatBoostRegressor(
    random_seed=42,
    logging_level='Silent')
ctbst.fit(
    X_train, y_train,
    cat_features=categorical_features_indices,
    plot=False)

# получаем прогнозы CatBoost
ctbst_predictions = ctbst.predict(X_test)
ctbst_predictions = pd.Series(ctbst_predictions,
                             index=y_test.index)

# визуализируем прогнозы
rmse_ctbst = mean_squared_error(
    y_test, ctbst_predictions, squared=False)
# задаем размер графика
plt.figure(figsize=(12, 6))

# задаем заголовок графика
plt.title(f'RMSE ctboost: {rmse_ctbst:.2f}')
# настраиваем ориентацию меток оси x
plt.xticks(rotation=90)
# строим графики для обучающих данных,
# тестовых данных и прогнозов
plt.plot(y_train, marker='o',
         label='обучающие данные')
plt.plot(y_test, color='green', marker='o',
         label='тестовые данные')
plt.plot(ctbst_predictions, color='red',
         label='прогнозы')
# задаем координатную сетку
plt.grid()
# задаем легенду
plt.legend()
plt.show();
```



Для генерации признака на основе тройного экспоненциального сглаживания мы использовали класс `ExponentialSmoothing` библиотеки `statsmodels`, а еще можно было воспользоваться классом `AutoETS` библиотеки `sktime` со встроенной оптимизацией параметров сглаживания. Обратите внимание, что библиотека `sktime` требует, чтобы индексом временного ряда был `PeriodIndex`, а зависимая переменная была вещественной. Давайте теперь воспользуемся этим классом для создания признаков и снова построим модель `CatBoost`.

```
# переведем DatetimeIndex в PeriodIndex, поскольку
# sktime не работает с DatetimeIndex
y_train.index = y_train.index.to_period('M')
y_train = y_train.astype('float64')
y_test.index = y_test.index.to_period('M')

# создаем прогнозную модель
forecaster = AutoETS(auto=True, sp=12, n_jobs=-1)

# создаем горизонты прогнозирования
fh_tr = ForecastingHorizon(
    y_train.index, is_relative=False)
fh_tst = ForecastingHorizon(
    y_test.index, is_relative=False)

# обучаем модель тройного экспоненциального
# сглаживания с помощью класса AutoETS
# библиотеки sktime
forecaster.fit(y_train)

# получаем прогнозы
sk_tr_preds = forecaster.predict(fh_tr)
sk_tst_preds = forecaster.predict(fh_tst)
```

```
# PeriodIndex массива прогнозов
# переводим в DatetimeIndex
sk_tr_preds.index = sk_tr_preds.index.to_timestamp()
sk_tst_preds.index = sk_tst_preds.index.to_timestamp()

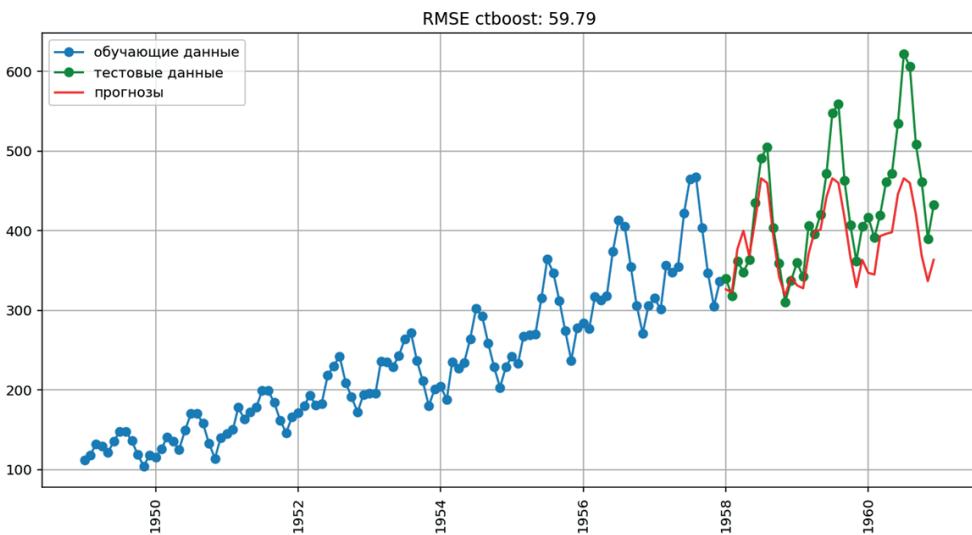
# создаем признаки на основе тройного
# экспоненциального сглаживания
X_train['ets_feature'] = sk_tr_preds
X_test['ets_feature'] = sk_tst_preds

# выполним обратные преобразования массивов меток
# из PeriodIndex в DatetimeIndex
y_train.index = y_train.index.to_timestamp()
y_test.index = y_test.index.to_timestamp()

# обучаем модель CatBoost
ctbst = CatBoostRegressor(
    random_seed=42,
    logging_level='Silent')
ctbst.fit(
    X_train, y_train,
    cat_features=categorical_features_indices,
    plot=False)

# получаем прогнозы CatBoost
ctbst_predictions = ctbst.predict(X_test)
ctbst_predictions = pd.Series(ctbst_predictions,
                             index=y_test.index)

# визуализируем прогнозы
rmse_ctbst = mean_squared_error(
    y_test, ctbst_predictions, squared=False)
# задаем размер графика
plt.figure(figsize=(12, 6))
# задаем заголовок графика
plt.title(f'RMSE ctboost: {rmse_ctbst:.2f}')
# настраиваем ориентацию меток оси x
plt.xticks(rotation=90)
# строим графики для обучающих данных,
# тестовых данных и прогнозов
plt.plot(y_train, marker='o',
         label='обучающие данные')
plt.plot(y_test, color='green', marker='o',
         label='тестовые данные')
plt.plot(ctbst_predictions, color='red',
         label='прогнозы')
# задаем координатную сетку
plt.grid()
# задаем легенду
plt.legend()
plt.show();
```



16.4.11. Прогнозирование тренда, детрендинг и добавление тренда к прогнозам

Мы получили прогнозы, но видим, что амплитуда сезонных колебаний остается неизменной, находится на одном и том же уровне. Надо передать градиентному бустингу информацию о тренде. Для этого выполним удаление тренда (детрендинг) с последующим восстановлением. Процедура состоит из 5 этапов.

1. На обучающем массиве признаков, в котором единственный признак – номер периода, и обучающем массиве меток (собственно временном ряде) обучаем линейную модель (например, модель линейной регрессии). Прогнозами у нас будут значения тренда для обучающего ряда.
2. Если у нас сезонность является мультипликативной, то значения обучающего массива меток делим на значения тренда для обучающей выборки и получаем обучающий массив меток без тренда. Если у нас сезонность является аддитивной, то из значений обучающего массива меток вычитаем значения тренда для обучающего ряда и получаем обучающий массив меток без тренда.
3. Вычисляем значения тренда для тестового ряда. Берем последнее значение тренда для обучающего ряда и к нему прибавляем второй коэффициент линейной регрессии, отвечающий за шаг, получаем стартовое значение. Создаем пустой список, в который будем записывать значения тренда для тестового ряда. Пишем цикл for, на каждой итерации добавляем в созданный список стартовое значение, прибавляем к стартовому значению шаг и обновляем его. Количество итераций определяется горизонтом прогнозирования (длиной тестового ряда). Обратите внимание, что здесь мы нигде не используем значения тестового ряда, мы помним, что тестовый ряд – это прообраз новых данных, о котором мы ничего не знаем.

4. На обучающем массиве признаков и обучающем массиве меток без тренда обучаем модель градиентного бустинга. Затем получаем прогнозы для тестового массива признаков.
5. Наконец, выполняем восстановление тренда. Если у нас была аддитивная сезонность, мы к прогнозам градиентного бустинга для тестового ряда прибавляем значения тренда для тестового ряда (потому что ранее вычитали тренд). Если у нас была мультипликативная сезонность, мы умножаем прогнозы градиентного бустинга для тестового ряда на значения тренда для тестового ряда (потому что ранее делили на тренд).

Давайте напишем функцию прогнозирования тренда.

```
# пишем функцию прогнозирования тренда
def forecast_trend(tr_target,
                    calc_trend_for_tst_target=False,
                    test_size=4,
                    freq='MS'):
```

Прогнозирует тренд с помощью линейной регрессии
для обучающего и тестового временных рядов.

Параметры

tr_target: pandas.Series

Обучающий массив меток.

calc_trend_for_tst_target, bool, по умолчанию False

Вычисляет тренд для тестового ряда.

test_size=4

Задает горизонт прогнозирования (определяется
размером тестового ряда).

freq: string, по умолчанию 'MS'

Частота для временных меток тестового ряда.

Возвращает

tr_trend_pred, coefs: значения тренда для
обучающего временного ряда,
коэффициенты линейной регрессии
(если вычисление тренда для тестового
ряда не задано)

tr_trend_pred, tst_trend_pred: значения тренда
для обучающего временного ряда, значения
тренда для тестового временного ряда
(если задано вычисление тренда
для тестового ряда)

"""

```
# создаем конвейер для прогнозирования тренда
regressor = make_pipeline(
    PolynomialFeatures(degree=1, include_bias=True),
    LinearRegression(fit_intercept=False))
```

```
# при прогнозировании тренда признаком является
# номер периода, а зависимой переменной -
# значение временного ряда
```

```

# зададим длину обучающего временного ряда
n_timepoints = len(tr_target)
# формируем массив признаков
X = np.arange(n_timepoints).reshape(-1, 1)
# обучаем конвейер и получаем коэффициенты
regressor.fit(X, tr_target)
coefs = regressor.named_steps['linearregression'].coef_
# получаем прогноз тренда с помощью нашего конвейера
tr_trend_pred = regressor.predict(X)
tr_trend_pred = pd.Series(
    tr_trend_pred,
    index=tr_target.index)

if calc_trend_for_tst_target is False:
    pass
else:
    # задаем стартовое значение, к последнему
    # значению тренда для обучающего ряда
    # прибавляем второй коэффициент
    # линейной регрессии, отвечающий за шаг
    start = tr_trend_pred[-1] + coefs[1]
    # задаем количество итераций
    count = test_size
    # задаем шаг
    step = coefs[1]
    # задаем пустой список, в который будем
    # записывать значения тренда
    # для тестового ряда
    numbers = list()
    # получаем значения тренда для тестового ряда
    for i in range(count):
        numbers.append(start)
        start += step
    # переводим значения тренда для тестового
    # временного ряда в серию
    tst_trend_pred = pd.Series(numbers)
    # формируем индекс для тестового временного ряда
    future_dates = pd.date_range(
        start=tr_target.index[-1],
        periods=test_size + 1,
        freq=freq,
        closed='right')
    tst_trend_pred.index = future_dates

if calc_trend_for_tst_target is False:
    return tr_trend_pred, coefs
else:
    return tr_trend_pred, tst_trend_pred

```

Давайте получим значения тренда для обучающего и тестового массивов меток.

```

# получаем значения тренда для обучающего
# и тестового массивов меток
y_train_trend_pred, y_test_trend_pred = forecast_trend(
    y_train,

```

```
calc_trend_for_tst_target=True,  
test_size=HORIZON)
```

Взглянем на первые 5 предсказанных значений тренда для обучающего и тестового массивов меток.

```
# выведем первые 5 предсказанных значений тренда  
# для обучающего массива меток  
y_train_trend_pred.head()
```

```
Month  
1949-01-01    97.53500  
1949-02-01    100.02777  
1949-03-01    102.52054  
1949-04-01    105.01331  
1949-05-01    107.50607  
Freq: MS, dtype: float64
```

```
# выведем первые 5 предсказанных значений тренда  
# для тестового массива меток  
y_test_trend_pred.head()
```

```
1958-01-01    366.75407  
1958-02-01    369.24684  
1958-03-01    371.73961  
1958-04-01    374.23237  
1958-05-01    376.72514  
Freq: MS, dtype: float64
```

Теперь получим значения обучающего массива меток без тренда, для этого делим значения обучающего массива меток на значения тренда для обучающего массива меток (поскольку у нас – мультиплексивная сезонность).

```
# получаем значения обучающего массива меток без тренда,  
# для этого делим значения обучающего массива меток на  
# значения тренда для обучающего массива меток  
y_train_detrended = y_train / y_train_trend_pred  
# выведем первые 5 значений обучающего  
# массива меток без тренда  
y_train_detrended.head()
```

```
Month  
1949-01-01    1.14831  
1949-02-01    1.17967  
1949-03-01    1.28755  
1949-04-01    1.22842  
1949-05-01    1.12552  
Freq: MS, dtype: float64
```

Теперь мы пишем функцию, с помощью которой обучаем модель CatBoost, получаем прогнозы, умножаем их на значения тренда для тестового массива меток, поскольку ранее значения обучающего ряда делили на значения тренда (помните, что мы получили прогнозы, ни разу не обращаясь к тестовому массиву меток) и визуализируем полученные прогнозы.

```

# пишем функцию, которая строит модель CatBoost
# и визуализирует прогнозы
def train_and_evaluate(model, tr, y_tr, tst, y_tst,
                       y_tr_trend_pred, y_tst_trend_pred,
                       cat_feat):
    """
    Обучает модель CatBoost, вычисляет прогнозы для
    тестовой выборки и строит график прогнозов.

Параметры
-----
tr: pandas.DataFrame
    Обучающий массив признаков.
y_tr: pandas.Series
    Обучающий массив меток.
tst:
    Тестовый массив признаков.
y_tst:
    Тестовый массив меток.
y_tr_trend_pred:
    Массив предсказанных значений тренда для обучающей выборки.
y_tst_trend_pred:
    Массив предсказанных значений тренда для тестовой выборки.
cat_feat:
    Массив индексов категориальных признаков.
"""

# обучаем модель CatBoost
model.fit(
    tr, y_tr,
    cat_features=cat_feat,
    plot=False)

# получаем прогнозы CatBoost
ctbst_predictions = (model.predict(tst) *
                     y_tst_trend_pred)

ctbst_predictions = pd.Series(ctbst_predictions,
                             index=y_tst.index)

# визуализируем прогнозы

# вычислим ошибку
rmse_ctbst = mean_squared_error(
    y_tst,
    ctbst_predictions,
    squared=False)
# задаем размер графика
plt.figure(figsize=(12, 6))
# задаем заголовок графика
plt.title(f'RMSE ctboost: {rmse_ctbst:.2f}')
# настраиваем ориентацию меток оси x
plt.xticks(rotation=90)

```

```

# поскольку для обучения использовали обучающий
# массив меток без тренда, для визуализации
# восстанавливаем исходный обучающий массив меток
y_tr = y_tr * y_tr_trend_pred
# строим графики для обучающих данных,
# тестовых данных и прогнозов
plt.plot(y_tr, marker='o',
          label='обучающие данные')
plt.plot(y_tst, color='green', marker='o',
          label='тестовые данные')
plt.plot(ctbst_predictions, color='red',
          label='прогнозы')
# задаем координатную сетку
plt.grid()
# задаем легенду
plt.legend()
plt.show();

```

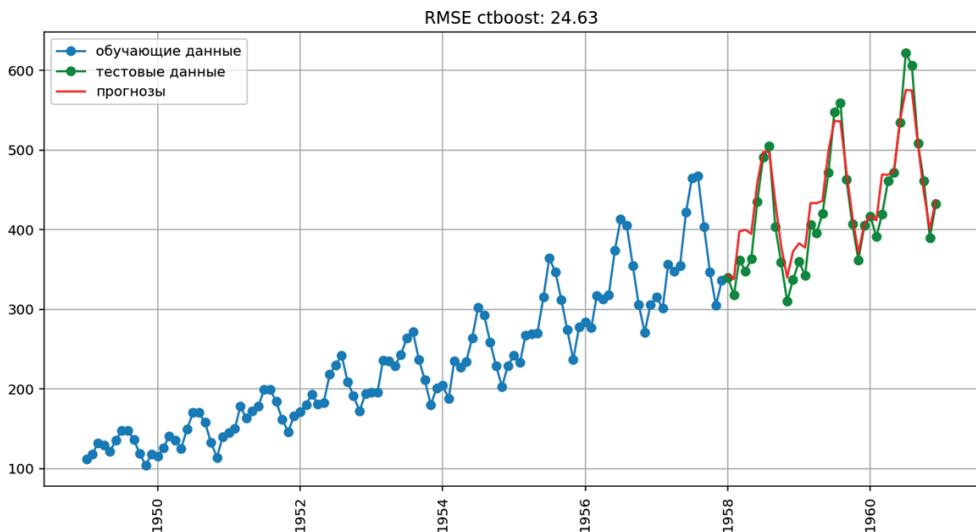
Давайте с помощью нашей функции обучим модель CatBoost и визуализируем прогнозы.

```

# создаем модель CatBoost
ctbst = CatBoostRegressor(random_seed=42,
                           logging_level='Silent')

# обучаем и оцениваем модель CatBoost
train_and_evaluate(
    ctbst, X_train, y_train_detrended, X_test, y_test,
    y_train_trend_pred, y_test_trend_pred,
    categorical_features_indices)

```



Бустинг «увидел» тренд и качество прогнозов значительно улучшилось.

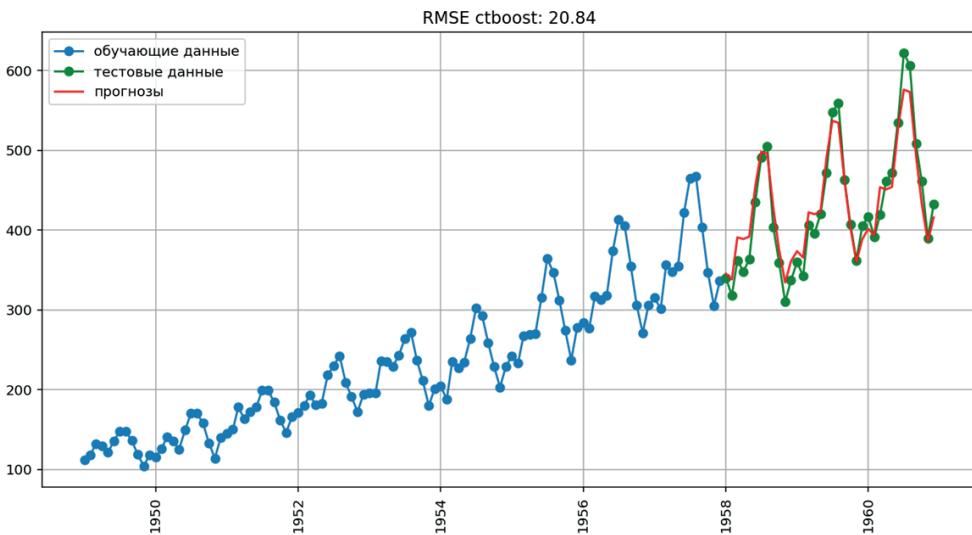
16.4.12. Добавление тренда, компонент ряда Фурье, произведения компоненты ряда Фурье и тренда в качестве признаков

Теперь удалим переменные на основе тройного экспоненциального сглаживания, вместо них добавим переменные на основе спрогнозированного тренда и снова построим модель CatBoost.

```
# удаляем переменную на основе тройного
# экспоненциального сглаживания
X_train.drop('ets_feature', axis=1, inplace=True)
X_test.drop('ets_feature', axis=1, inplace=True)

# добавляем переменную - тренд
X_train['trend'] = y_train_trend_pred
X_test['trend'] = y_test_trend_pred

# обучаем и оцениваем модель CatBoost
train_and_evaluate(
    ctbst, X_train, y_train_detrended,
    X_test, y_test,
    y_train_trend_pred, y_test_trend_pred,
    categorical_features_indices)
```



Качество прогнозов тоже заметно улучшилось.
А сейчас добавим компоненты ряда Фурье первого порядка для годовой сезонности и опять построим модель CatBoost.

Сначала получаем индекс для объединенного набора, чтобы на его основе сразу вычислить компоненты ряда Фурье для обучающей и тестовой выборок.

```
# получаем индекс
indx = pd.concat([X_train, X_test], axis=0).index
# создаем члены ряда Фурье первого порядка для годовой сезонности
fourier_year = fourier(indx,
                       out_column='fourier_year',
                       period=12, order=1)
fourier_year.head()
```

fourier_year_1 fourier_year_2

Month		
1949-01-01	0.00000	1.00000
1949-02-01	0.50000	0.86603
1949-03-01	0.86603	0.50000
1949-04-01	1.00000	0.00000
1949-05-01	0.86603	-0.50000

Выделяем признаки – компоненты ряда Фурье для обучающей выборки.

```
# выделим признаки – члены ряда Фурье для обучающей выборки
fourier_year_train = fourier_year.iloc[:,-HORIZON]
fourier_year_train.head()
```

fourier_year_1 fourier_year_2

Month		
1949-01-01	0.00000	1.00000
1949-02-01	0.50000	0.86603
1949-03-01	0.86603	0.50000
1949-04-01	1.00000	0.00000
1949-05-01	0.86603	-0.50000

Выделяем признаки – компоненты ряда Фурье для тестовой выборки.

```
# выделим признаки – члены ряда Фурье для тестовой выборки
fourier_year_test = fourier_year.iloc[-HORIZON:]
fourier_year_test.head()
```

fourier_year_1 fourier_year_2

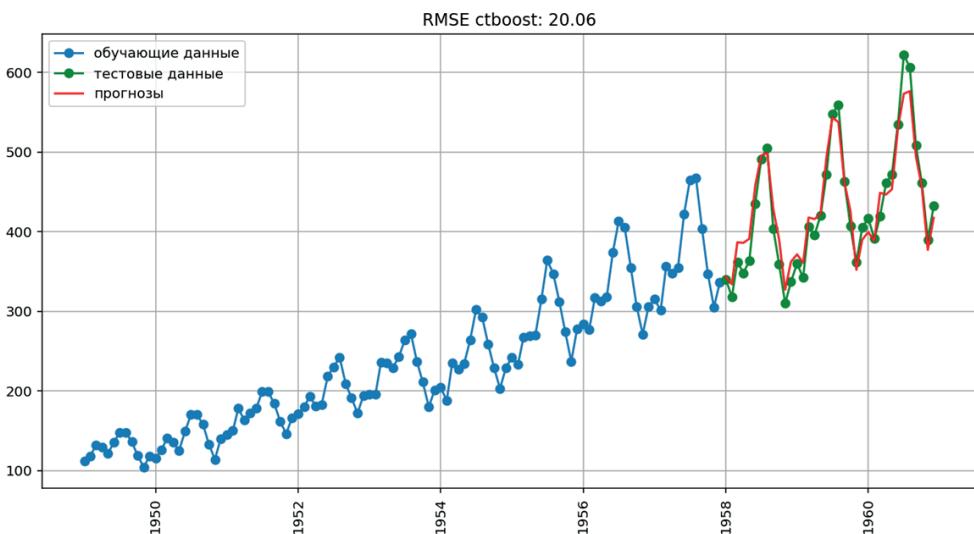
Month	fourier_year_1	fourier_year_2
1958-01-01	-0.00000	1.00000
1958-02-01	0.50000	0.86603
1958-03-01	0.86603	0.50000
1958-04-01	1.00000	-0.00000
1958-05-01	0.86603	-0.50000

Теперь компоненты ряда Фурье для обучающей выборки добавляем в обучающую выборку, а компоненты ряда Фурье для тестовой выборки – в тестовую выборку.

```
# добавляем члены ряда Фурье для обучающей выборки в обучающую выборку
X_train = pd.concat([X_train, fourier_year_train], axis=1)
# добавляем члены ряда Фурье для тестовой выборки в тестовую выборку
X_test = pd.concat([X_test, fourier_year_test], axis=1)
```

Итак, строим модель CatBoost и оцениваем ее качество.

```
# обучаем и оцениваем модель CatBoost
train_and_evaluate(
    ctbst, X_train, y_train_detrended,
    X_test, y_test,
    y_train_trend_pred, y_test_trend_pred,
    categorical_features_indices)
```

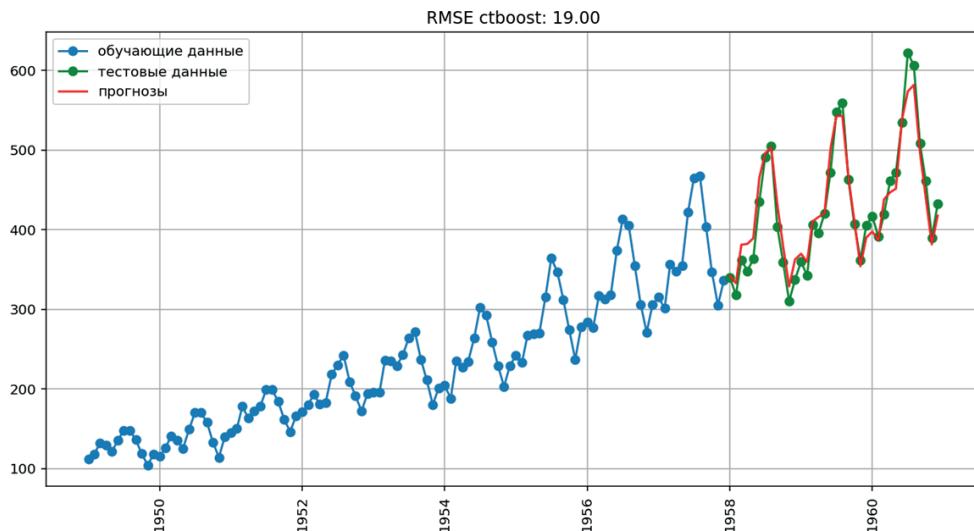


Качество прогнозов осталось практически прежним.

Теперь добавим произведение тренда и члена ряда Фурье – косинуса первого порядка. Подобные переменные, представляющие собой произведения тренда и члена ряда Фурье, используются под капотом библиотеки Greykite. Напомним, что произведения членов ряда Фурье и календарных признаков также могут улучшить качество как линейной модели, так и модели на основе ансамбля деревьев и они тоже используются под капотом библиотеки Greykite.

```
# добавляем переменную - произведение тренда
# и компоненты ряда Фурье - косинуса первого порядка
X_train['trend * fourier_year'] = (y_train_trend_pred *
                                     X_train['fourier_year_2'])
X_test['trend * fourier_year'] = (y_test_trend_pred *
                                    X_test['fourier_year_2'])

# обучаем и оцениваем модель CatBoost
train_and_evaluate(
    ctbst, X_train, y_train_detrended, X_test, y_test,
    y_train_trend_pred, y_test_trend_pred,
    categorical_features_indices)
```



Качество прогнозов немного улучшилось.

А теперь посмотрим, какое качество прогнозов даст модель тройного экспоненциального слгаживания сама по себе. Возьмем прогнозы, полученные с помощью класса ExponentialSmoothing библиотеки statsmodels.

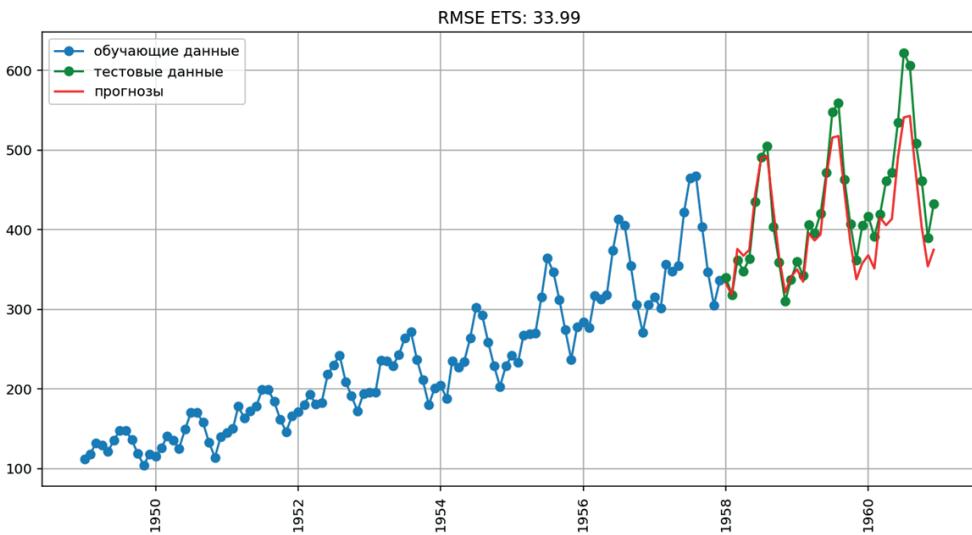
```
# визуализируем прогнозы ETS
# вычислим ошибку
rmse_ets = mean_squared_error(
    y_test,
    test_preds,
    squared=False)

# задаем размер графика
```

```

plt.figure(figsize=(12, 6))
# задаем заголовок графика
plt.title(f'RMSE ETS: {rmse_ets:.2f}')
# настраиваем ориентацию меток оси x
plt.xticks(rotation=90)
# строим графики для обучающих данных,
# проверочных данных и прогнозов
plt.plot(y_train, marker='o',
          label='обучающие данные')
plt.plot(y_test, color='green', marker='o',
          label='проверочные данные')
plt.plot(test_preds, color='red',
          label='прогнозы')
# задаем координатную сетку
plt.grid()
# задаем легенду
plt.legend()
plt.show();

```



Модель тройного экспоненциального сглаживания оказалась менее точной, чем градиентный бустинг.

16.4.13. Добавление информации о светлом/темном времени суток, времени восхода и времени заката

Для успешного решения ряда задач может помочь информация о светлом/темном времени суток, времени восхода и времени заката. Например, такие признаки могут быть полезны в задачах прогнозирования сумм снятий в банкоматах.

Давайте напишем функцию, которая возвращает информацию о длине светлого/темного времени суток, местном времени восхода и местном времени заката для заданного временного периода (только для часовых поясов России), используя Sunset and sunrise times API.

```
# пишем функцию, которая возвращает информацию о длине светлого/темного
# времени суток, местном времени восхода и местном времени заката для
# заданного временного периода, используя Sunset and sunrise times API
def calculate_sunrise_sunset(coordinates, start_date, end_date, tz):
    """
    Вычисляет продолжительность светлого/темного времени суток, времени
    восхода и времени заката для заданного временного периода
    с помощью Sunset and sunrise times API
    https://sunrise-sunset.org/api
    (реализация только для часовых поясов России)

Параметры
-----
coordinates: Координаты
start_date: Стартовая дата периода
end_date: Конечная дата периода
tz: Часовой пояс, возможные значения:
    'KALT' - калининградское время МСК-1 (UTC+2)
    'MSK' - московское время МСК (UTC+3)
    'SAMT' - самарское время МСК+1 (UTC+4)
    'YEKT' - екатеринбургское время МСК+2 (UTC+5)
    'OMST' - омское время МСК+3 (UTC+6)
    'KRAT' - красноярское время МСК+4 (UTC+7)
    'IRKT' - иркутское время МСК+5 (UTC+8)
    'YAKT' - якутское время МСК+6 (UTC+9)
    'VLAT' - владивостокское время МСК+7 (UTC+10)
    'MAGT' - магаданское время МСК+8 (UTC+11)
    'PETT' - камчатское время МСК+9 (UTC+12)

Возвращает
-----
date, darktime, daytime, sunrise, sunset: дату, продолжительность
    темного времени суток, продолжительность светлого времени
    суток, местное время восхода, местное время заката
"""

output_dataframe = pd.DataFrame(
    columns=['date', 'darktime', 'daytime', 'sunrise', 'sunset'])
true_date = datetime.datetime.strptime(start_date, '%Y-%m-%d')
url = 'https://api.sunrise-sunset.org/json'
params = {
    'lat': coordinates[0],
    'lng': coordinates[1],
    'date': true_date.strftime('%Y-%m-%d'),
    'formatted': 0
}

# создаем словарь, ключи - строковые названия
# часовыx поясов, значения - смещения
tz_lst = ['KALT', 'MSK', 'SAMT', 'YEKT', 'OMST', 'KRAT',
          'IRKT', 'YAKT', 'VLAT', 'MAGT', 'PETT']
```

```

offset_lst = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
d = dict(zip(tz_lst, offset_lst))
while true_date <= datetime.datetime.strptime(end_date, '%Y-%m-%d'):
    fetcher = requests.get(url, params=params)
    status = fetcher.status_code
    if status != 200:
        print('Ошибка получения данных:', fetcher.reason)
        break
    fetcher_dict = fetcher.json()
    # получаем продолжительность светлого времени суток
    daytime = datetime.timedelta(
        seconds=fetcher_dict['results'][0]['day_length'])
    # получаем продолжительность темного времени суток
    darktime = datetime.timedelta(days=1) - daytime
    # извлекаем время восхода и время заката в UTC
    sunrise = fetcher_dict['results'][0]['sunrise']
    sunset = fetcher_dict['results'][0]['sunset']
    # преобразовываем время восхода и время заката в местное время
    sunrise = (datetime.datetime.strptime(
        sunrise, '%Y-%m-%dT%H:%M:%S%z').replace(tzinfo=None) +
        daytime.timedelta(hours=d[tz])))
    sunset = (datetime.datetime.strptime(
        sunset, '%Y-%m-%dT%H:%M:%S%z').replace(tzinfo=None) +
        daytime.timedelta(hours=d[tz])))

    output_dataframe.loc[len(output_dataframe)] = (
        true_date, darktime, daytime, sunrise, sunset)
    true_date += datetime.timedelta(days=1)
    params['date'] = true_date.strftime('%Y-%m-%d')

return output_dataframe

```

Давайте зададим координаты Санкт-Петербурга и получим информацию о продолжительности светлого/темного времени суток, местном времени восхода и времени заката для периода с 1 по 10 июля 2017 года.

```

# задаем координаты Санкт-Петербурга
spb_coords = [59.563178, 30.188478]
# получаем информацию о продолжительности светлого/темного
# времени суток, местном времени восхода и местном времени
# заката для нашего временного отрезка
spb_df = calculate_sunrise_sunset(spb_coords,
                                    '2017-07-01',
                                    '2017-07-10',
                                    'MSK')
spb_df

```

	date	darktime	daytime	sunrise	sunset
0	2017-07-01	0 days 05:19:55	0 days 18:40:05	2017-07-01 03:43:06	2017-07-01 22:23:11
1	2017-07-02	0 days 05:21:46	0 days 18:38:14	2017-07-02 03:44:13	2017-07-02 22:22:27
2	2017-07-03	0 days 05:23:46	0 days 18:36:14	2017-07-03 03:45:24	2017-07-03 22:21:38
3	2017-07-04	0 days 05:25:55	0 days 18:34:05	2017-07-04 03:46:39	2017-07-04 22:20:44
4	2017-07-05	0 days 05:28:13	0 days 18:31:47	2017-07-05 03:47:59	2017-07-05 22:19:46
5	2017-07-06	0 days 05:30:40	0 days 18:29:20	2017-07-06 03:49:22	2017-07-06 22:18:42
6	2017-07-07	0 days 05:33:15	0 days 18:26:45	2017-07-07 03:50:50	2017-07-07 22:17:35
7	2017-07-08	0 days 05:35:59	0 days 18:24:01	2017-07-08 03:52:21	2017-07-08 22:16:22
8	2017-07-09	0 days 05:38:49	0 days 18:21:11	2017-07-09 03:53:55	2017-07-09 22:15:06
9	2017-07-10	0 days 05:41:48	0 days 18:18:12	2017-07-10 03:55:33	2017-07-10 22:13:45

Взглянем на типы переменных.

```
# смотрим типы переменных
spb_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
----  --          --          --      
 0   date        10 non-null    datetime64[ns]
 1   darktime    10 non-null    timedelta64[ns]
 2   daytime     10 non-null    timedelta64[ns]
 3   sunrise     10 non-null    datetime64[ns]
 4   sunset      10 non-null    datetime64[ns]
dtypes: datetime64[ns](3), timedelta64[ns](2)
memory usage: 480.0 bytes
```

Переведем продолжительность светлого/темного времени суток в часы, создадим переменные – час восхода, час заката и разницу между закатом и восходом и вновь взглянем на типы переменных.

```
# переводим продолжительность светлого/темного времени суток в часы
spb_df['daytime'] = spb_df['daytime'].dt.total_seconds() / 60 / 60
spb_df['darktime'] = spb_df['darktime'].dt.total_seconds() / 60 / 60
# выделяем час восхода, час заката и разницу между закатом и восходом
spb_df['sunrise_hour'] = spb_df['sunrise'].dt.hour
spb_df['sunset_hour'] = spb_df['sunset'].dt.hour
spb_df['diff'] = spb_df['sunset_hour'] - spb_df['sunrise_hour']
```

```
# смотрим типы переменных
spb_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10 entries, 0 to 9
```

```
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
 ...   ...        
 0   date        10 non-null     datetime64[ns]
 1   darktime    10 non-null     float64 
 2   daytime     10 non-null     float64 
 3   sunrise     10 non-null     datetime64[ns]
 4   sunset      10 non-null     datetime64[ns]
 5   sunrise_hour 10 non-null     int64   
 6   sunset_hour 10 non-null     int64   
 7   diff         10 non-null     int64   
dtypes: datetime64[ns](3), float64(2), int64(3)
memory usage: 720.0 bytes
```

Теперь взглянем на сам датафрейм.

```
# смотрим результаты
spb_df
```

	date	darktime	daytime	sunrise	sunset	sunrise_ho
0	2017-07-01	5.33194	18.66806	2017-07-01 03:43:06	2017-07-01 22:23:11	
1	2017-07-02	5.36278	18.63722	2017-07-02 03:44:13	2017-07-02 22:22:27	
2	2017-07-03	5.39611	18.60389	2017-07-03 03:45:24	2017-07-03 22:21:38	
3	2017-07-04	5.43194	18.56806	2017-07-04 03:46:39	2017-07-04 22:20:44	
4	2017-07-05	5.47028	18.52972	2017-07-05 03:47:59	2017-07-05 22:19:46	
5	2017-07-06	5.51111	18.48889	2017-07-06 03:49:22	2017-07-06 22:18:42	
6	2017-07-07	5.55417	18.44583	2017-07-07 03:50:50	2017-07-07 22:17:35	
7	2017-07-08	5.59972	18.40028	2017-07-08 03:52:21	2017-07-08 22:16:22	

16.4.14. Добавление информации о погодных условиях (минимальной температуре, максимальной температуре, высоте снежного покрова и т.д)

Часто у нас имеется временной ряд и его необходимо обогатить данными о погодных условиях. В этом нам может помочь функция пакета WorldWeatherOnline historical weather data API wrapper.

Допустим, у нас есть временной ряд продаж в Москве с 16 апреля по 12 августа 2021 года. Обогатим его информацией о погодных условиях. Нам нужно задать частоту ряда (например, можно задать информацию о погоде за каждые 24 часа, т.е. за каждые сутки, за каждые 4 часа, за каждый час), стартовую и конечную даты, API-ключ и географическую точку.

```
# импортируем функцию загрузки погодных данных
from wwo_hist import retrieve_hist_data

# задаем частоту ряда, стартовую и
# конечную даты, API-ключ и локацию
frequency = 24
```

430 ❖ План предварительной подготовки данных

```

start_date = '16-APR-2021'
end_date = '12-AUG-2021'
api_key = 'ваш-апи-ключ'
location = ['moscow']

# загружаем датафрейм с погодными данными
hist_weather_data = retrieve_hist_data(api_key,
                                         location,
                                         start_date,
                                         end_date,
                                         frequency,
                                         location_label=False,
                                         export_csv=False,
                                         store_df=True)

hist_weather_data = hist_weather_data[0]
hist_weather_data

```

Retrieving weather data for moscow

```

Currently retrieving data for moscow: from 2021-04-16 to 2021-04-30
Time elapsed (hh:mm:ss.ms) 0:00:00.273999
Currently retrieving data for moscow: from 2021-05-01 to 2021-05-31
Time elapsed (hh:mm:ss.ms) 0:00:00.673437
Currently retrieving data for moscow: from 2021-06-01 to 2021-06-30
Time elapsed (hh:mm:ss.ms) 0:00:01.086673
Currently retrieving data for moscow: from 2021-07-01 to 2021-07-31
Time elapsed (hh:mm:ss.ms) 0:00:01.530547
Currently retrieving data for moscow: from 2021-08-01 to 2021-08-12
Time elapsed (hh:mm:ss.ms) 0:00:01.732210

```

	date_time	maxtempC	mintempC	totalSnow_cm	sunHour	uvIndex	moon_illumination	moonrise	moonset	sunrise	sunset	DewPointC	FeelsLikeC	
0	2021-04-16	12	6	0.0	8.9	3		29	07:03 AM	No moonset	05:22 AM	07:38 PM	4	6
0	2021-04-17	14	8	0.0	13.4	4		35	07:31 AM	12:40 AM	05:20 AM	07:40 PM	1	8
0	2021-04-18	11	6	0.0	11.1	2		42	08:09 AM	01:46 AM	05:17 AM	07:42 PM	-0	6
0	2021-04-19	9	-0	0.0	8.9	2		49	09:02 AM	02:39 AM	05:15 AM	07:44 PM	3	4
0	2021-04-20	7	-1	0.0	9.0	1		56	10:09 AM	03:21 AM	05:12 AM	07:46 PM	1	1
...
0	2021-08-08	25	19	0.0	14.5	6		1	03:48 AM	08:56 PM	04:48 AM	08:21 PM	16	22
0	2021-08-09	31	20	0.0	14.5	6		7	05:10 AM	09:14 PM	04:50 AM	08:19 PM	16	26
0	2021-08-10	27	21	0.0	10.7	5		14	06:34 AM	09:27 PM	04:52 AM	08:16 PM	16	25
0	2021-08-11	23	19	0.0	9.8	5		21	08:00 AM	09:39 PM	04:54 AM	08:14 PM	18	21
0	2021-08-12	26	19	0.0	11.6	5		28	09:26 AM	09:49 PM	04:56 AM	08:12 PM	15	23

	HeatIndexC	WindChillC	WindGustKmph	cloudcover	humidity	precipMM	pressure	tempC	visibility	winddirDegree	windspeedKmph	location
9	6	28	78	72	1.4	1021	12	9	60		21	moscow
11	8	25	45	54	0.0	1025	14	10	67		20	moscow
8	6	24	60	55	0.1	1023	11	10	79		18	moscow
7	4	22	99	74	1.0	1025	9	8	81		17	moscow
4	1	30	100	78	2.1	1021	7	8	86		22	moscow
...
22	22	11	30	72	10.7	1014	25	9	234		8	moscow
26	25	7	26	62	1.6	1017	31	10	130		5	moscow
25	24	14	62	62	2.2	1016	27	9	113		10	moscow
22	21	12	82	83	11.6	1015	23	9	157		9	moscow
23	22	14	58	69	6.1	1011	26	10	222		11	moscow

Датафрейм содержит следующие переменные:

- *date_time* – дата и время;
- *maxtempC* – максимальная температура в °C;
- *mintempC* – минимальная температура в °C;
- *totalSnow_cm* – высота снежного покрова в сантиметрах;
- *sunHour* – количество солнце-часов;
- *uvIndex* – ультрафиолетовый индекс;
- *moon_illumination* – освещенность Луны в процентах;
- *moonrise* – восход Луны;
- *moonset* – закат Луны;
- *sunrise* – восход Солнца;
- *sunset* – закат Солнца;
- *DewPointC* – точка росы в градусах;
- *FeelsLikeC* – ощущаемая температура в градусах;
- *HeatIndexC* – индекс жары в градусах;
- *WindChillC* – ветро-холодовой индекс в градусах;
- *WindGustKmph* – скорость порыва ветра (до 20 секунд) в км/ч
- *cloudcover* – облачность в процентах;
- *humidity* – влажность в процентах;
- *precipMM* – атмосферные осадки в мм;
- *pressure* – атмосферное давление в гектопаскалях;
- *tempC* – температура в °visibility – дальность видимости в километрах;
- *winddirDegree* – направление ветра в градусах;
- *windspeedKmph* – скорость порыва ветра (до 20 секунд) в км/ч;
- *location* – географическая точка.

Можно загрузить данные о погоде по нескольким городам.

```
# загрузим данные по нескольким городам
location_list = ['bryansk', 'ufa', 'nizhny_novgorod']

d = []

for location in location_list:
    hist_weather_data = retrieve_hist_data(api_key,
                                             [location],
                                             start_date,
                                             end_date,
                                             frequency,
                                             location_label=False,
                                             export_csv=False,
                                             store_df=True)
    hist_weather_data = hist_weather_data[0]
    d.append(hist_weather_data)
```

Retrieving weather data for bryansk

```
Currently retrieving data for bryansk: from 2021-04-16 to 2021-04-30
Time elapsed (hh:mm:ss.ms) 0:00:00.329954
Currently retrieving data for bryansk: from 2021-05-01 to 2021-05-31
Time elapsed (hh:mm:ss.ms) 0:00:00.776495
Currently retrieving data for bryansk: from 2021-06-01 to 2021-06-30
Time elapsed (hh:mm:ss.ms) 0:00:01.179878
Currently retrieving data for bryansk: from 2021-07-01 to 2021-07-31
Time elapsed (hh:mm:ss.ms) 0:00:01.599630
```

432 ❖ План предварительной подготовки данных

Currently retrieving data for bryansk: from 2021-08-01 to 2021-08-12
Time elapsed (hh:mm:ss.ms) 0:00:01.797744

Retrieving weather data for ufa

Currently retrieving data for ufa: from 2021-04-16 to 2021-04-30

Time elapsed (hh:mm:ss.ms) 0:00:00.302362

Currently retrieving data for ufa: from 2021-05-01 to 2021-05-31

Time elapsed (hh:mm:ss.ms) 0:00:00.752032

Currently retrieving data for ufa: from 2021-06-01 to 2021-06-30

Time elapsed (hh:mm:ss.ms) 0:00:01.181727

Currently retrieving data for ufa: from 2021-07-01 to 2021-07-31

Time elapsed (hh:mm:ss.ms) 0:00:01.617010

Currently retrieving data for ufa: from 2021-08-01 to 2021-08-12

Time elapsed (hh:mm:ss.ms) 0:00:01.814976

Retrieving weather data for nizhny_novgorod

Currently retrieving data for nizhny_novgorod: from 2021-04-16 to 2021-04-30

Time elapsed (hh:mm:ss.ms) 0:00:00.369983

Currently retrieving data for nizhny_novgorod: from 2021-05-01 to 2021-05-31

Time elapsed (hh:mm:ss.ms) 0:00:00.830206

Currently retrieving data for nizhny_novgorod: from 2021-06-01 to 2021-06-30

Time elapsed (hh:mm:ss.ms) 0:00:01.274360

Currently retrieving data for nizhny_novgorod: from 2021-07-01 to 2021-07-31

Time elapsed (hh:mm:ss.ms) 0:00:01.721867

Currently retrieving data for nizhny_novgorod: from 2021-08-01 to 2021-08-12

Time elapsed (hh:mm:ss.ms) 0:00:01.961067

формируем единый датафрейм

df = pd.concat(d)

df

	date_time	maxtempC	mintempC	totalSnow_cm	sunHour	uvIndex	moon_illumination	moonrise	moonset	sunrise	sunset	DewPointC	FeelsLikeC	
0	2021-04-16	13	7	0.0	8.8	3		29	07:31 AM	No moonset	05:41 AM	07:45 PM	7	8
0	2021-04-17	11	6	0.0	9.9	2		35	08:02 AM	12:37 AM	05:39 AM	07:47 PM	7	8
0	2021-04-18	6	4	0.0	8.8	2		42	08:42 AM	01:40 AM	05:37 AM	07:49 PM	4	3
0	2021-04-19	8	5	0.0	8.9	2		49	09:35 AM	02:33 AM	05:35 AM	07:51 PM	5	3
0	2021-04-20	9	3	0.0	8.9	2		56	10:40 AM	03:17 AM	05:32 AM	07:52 PM	4	2
...
0	2021-08-08	32	18	0.0	14.5	7		1	03:17 AM	08:33 PM	04:20 AM	07:58 PM	12	25
0	2021-08-09	32	22	0.0	14.5	7		7	04:40 AM	08:50 PM	04:22 AM	07:56 PM	14	27
0	2021-08-10	24	20	0.0	10.6	5		14	06:05 AM	09:03 PM	04:24 AM	07:54 PM	15	24
0	2021-08-11	22	20	0.0	12.5	5		21	07:32 AM	09:14 PM	04:26 AM	07:51 PM	17	21
0	2021-08-12	33	21	0.0	14.5	7		28	08:58 AM	09:24 PM	04:28 AM	07:49 PM	12	28

HeatIndexC	WindChillC	WindGustKmph	cloudcover	humidity	precipMM	pressure	tempC	visibility	winddirDegree	windspeedKmph	location
10	8	18	75	83	9.4	1014	13	8	102	11	bryansk
9	8	19	72	84	1.1	1020	11	10	55	12	bryansk
6	3	19	94	90	5.4	1018	6	7	129	12	bryansk
7	3	31	97	91	21.1	1018	8	8	71	20	bryansk
5	2	25	100	91	4.1	1013	9	7	57	17	bryansk
...
25	25	22	28	49	0.0	1016	32	10	141	13	nizhny_novgorod
27	26	12	25	48	0.0	1017	32	10	251	7	nizhny_novgorod
24	23	10	67	61	0.1	1018	24	10	91	6	nizhny_novgorod
23	21	13	65	76	3.0	1017	22	10	102	7	nizhny_novgorod
28	27	29	42	45	0.3	1011	33	10	141	19	nizhny_novgorod

17. Отбор признаков

Отбор признаков – это отбор подмножества важных признаков для последующего построения модели машинного обучения. Он необходим по ряду следующих причин:

- улучшение обобщающей способности модели из-за уменьшения проклятия размерности⁶ и переобучения (удаление избыточных и нерелевантных переменных делает модель менее сложной и менее восприимчивой к случайным возмущениям обучающих данных, которые сложной моделью будут восприняты за сигнал);
- сокращение времени обучения;
- простые модели легче интерпретировать;
- простые модели легче имплементировать.

Обычно выделяют задачу-минимум и задачу-максимум. Задача-минимум – сократить признаковое пространство без потери качества. Задача-максимум – сократить признаковое пространство и при этом улучшить качество модели. Разумеется, нас интересует качество на независимом, тестовом наборе.

Как правило, методы отбора признаков характеризуются высокими вычислительными затратами, один и тот же метод отбора признаков для разных методов машинного обучения может дать разные подмножества признаков. Не существует одного оптимального метода отбора признаков. Вопрос «какой метод отбора признаков является лучшим?» тождествен вопросу «какой метод машинного обучения является лучшим?». Поэтому каждый раз берем задачу и пробуем разные методы отбора признаков. Важный признак является релевантным (связан с зависимой переменной) и неизбыточным (дает новую информацию, еще не объясненную другими признаками).

Существует три типа методов отбора признаков:

- методы-фильтры (filter methods);
- методы-обертки (wrapper methods);
- встроенные методы (embedded methods).

Кроме того, можно выделить гибридные подходы, сочетающие методы-обертки и встроенные методы. Кстати, именно они на практике часто дают наилучшие результаты.

Методы-фильтры основываются только на статистиках признаков, не используют методы машинного обучения, таким образом, не зависят от моделей машинного обучения. Обычно такие методы являются менее вычислительно затратными. Вместе с тем эти методы менее эффективны по сравнению с методами-обертками и встроенными методами с точки зрения получения подмножества признаков, дающего наилучшее качество модели. В их основе – следующая процедура:

⁶ Речь идет об экспоненциальном росте числа данных, необходимого для заполнения пространства при увеличении размерности. Например, восемь точек плотно заполняют одномерное пространство, но становятся более разрозненными при увеличении размерности. В 100-мерном пространстве они будут похожи на удаленные галактики. Для удачного подбора модели требуется плотно заполненное пространство признаков.

- получаем оценку каждого признака по отдельности с точки зрения определенного критерия (например, с точки зрения IV), таким образом, отбор является одномерным;
- ранжируем признаки по полученным оценкам;
- выбираем признаки с наименее высокой оценкой.

Методы-фильтры не годятся для отбора сильных переменных. Они могут отбирать избыточные переменные, поскольку преимущественно являются одномерными и не учитывают взаимосвязи между признаками (за исключением методов-фильтров на основе корреляции). Мы оцениваем, как признак работает сам по себе, а не в сочетании с другими признаками. Методы-фильтры хороши для быстрого мониторинга и удаления наименее релевантных признаков (константных, дублирующихся признаков) на самом раннем этапе отбора признаков. Примерами методов-фильтров может быть отбор признаков по:

- информационному значению (чем выше значение IV, тем выше прогнозная сила);
- взаимной информации (насколько информация, содержащаяся в признаке, снижает неопределенность относительно зависимой переменной; если признак и целевая переменная не зависят друг от друга, взаимная информация равна 0);
- дисперсии (низкая оценка дисперсии может указывать на почти константный признак);
- критерию хи-квадрат или F-критерию (чем выше значение хи-квадрат/F-критерий и ниже p -значение, тем переменная важнее);
- коэффициенту корреляции (есть корреляция с зависимой переменной – хороший признак, есть корреляция с другим признаком – избыточный признак);
- метрике (чем выше AUC, тем важнее признак).

Методы-обертки используют методы машинного обучения, таким образом, зависят от моделей машинного обучения. В их основе – следующая процедура:

- находим подмножество признаков;
- строим модель машинного обучения на этом подмножестве признаков;
- оцениваем качество модели;
- повторяем.

Задача заключается в том, чтобы найти подмножество признаков, которое дает наилучшее качество модели. Здесь нужно определиться с критерием поиска и критерием остановки поиска. Примерами методов-оберток может быть прямое включение признаков, обратное исключение признаков, исчерпывающий поиск.

Метод прямого включения – это итеративный метод, который начинает работу с модели без признаков. На каждой итерации мы добавляем наименее значимый признак (который имеет наименьшее p -значение). Процедура прямого включения заканчивается, когда больше нет признаков с p -значениями, меньшими, чем заранее заданный порог для добавления переменных, или когда включено заданное количество признаков. Метод обратного исключения – это итеративный метод, который начинает работу с модели, включающей все признаки. На каждой итерации мы удаляем наименее значимый признак (ко-

торый имеет наибольшее p -значение). Процедура обратного включения заканчивается, когда все оставшиеся признаки имеют p -значения, меньшие, чем заранее заданный порог для исключения переменных, или когда исключено заданное количество признаков.

Если метод прямого включения включает по одному признаку за итерацию, метод обратного исключения исключает по одному признаку за итерацию, то метод исчерпывающего поиска ищет наилучшее подмножество признаков по всем возможным комбинациям признаков.

Первый недостаток методов-оберток заключается в том, что эти методы являются более вычислительно затратными и иногда их невозможно использовать, когда признаков очень много, а вычислительные ресурсы ограничены. Второй недостаток методов-оберток – произвольность критерия остановки, который устанавливает сам пользователь.

Вместе с тем методы-обертки, в отличие от методов-фильтров, чаще позволяют найти подмножество признаков, дающее наилучшее качество модели. Это обусловлено тем, что в отличие от методов-фильтров они могут учитывать взаимодействия переменных. Вы должны помнить, что некоторые признаки могут не обладать прогнозной силой по отдельности, но в сочетании с другими переменными приобретают ее.

Разумеется, с помощью метода-обертки мы не можем получить подмножество признаков, которое даст наилучшее качество при использовании разных моделей машинного обучения. Например, подмножество признаков, найденное с помощью логистической регрессии, не гарантирует наилучшего качества при построении градиентного бустинга на этом подмножестве. В то же время подмножество признаков, отобранное с помощью градиентного бустинга, может дать хорошее качество при построении на этом подмножестве случайного леса. Поэтому, выбирая метод-обертку, нужно помнить о том, какую модель машинного обучения мы потом будем строить на отобранном подмножестве признаков.

Встроенные методы выполняют отбор признаков в ходе обучения модели машинного обучения. В их основе – следующая процедура:

- обучаем модель машинного обучения;
- вычисляем важности признаков;
- удаляем неважные признаки (в случайном лесе или бустинге для этого задается определенный порог, ниже которого считаем признаки неважными, в методе LASSO неважные признаки просто получат нулевые коэффициенты и будут исключены из модели).

Эти методы менее вычислительно затратны, чем методы-обертки, поскольку они обучают модель машинного обучения один раз. Они могут дать лучшее качество, чем методы-фильтры, потому что учитывают взаимодействия между признаками. В то же время они ограничены методами машинного обучения, на основе которых построены. Примерами встроенных методов является отбор признаков с помощью регуляризации LASSO, отбор признаков на основе важностей на основе уменьшения неоднородности для случайного леса, пермутированных важностей (изначально предназначались для случайного леса, но можно применять для любой модели машинного обучения с учителем) и т. д.

Отбор признаков – это тоже модель, либо использующая вычисление статистик признаков (в случае с методами-фильтрами), либо использующая парамет-

тры моделей машинного обучения (в случае с методами-обертками и встроеннымными методами), поэтому модели отбора признаков строим на обучающей выборке, настраиваем гиперпараметры на проверочной выборке и получаем итоговую оценку качества на тестовой.

Автор готовит отдельную книгу, посвященную отбору признаков, поэтому в этом разделе сосредоточимся на методах-фильтрах, разберем кейс применения метода-фильтра и встроенного метода для решения соревнования BNP Paribas Cardif Claims Management <https://www.kaggle.com/competitions/bnp-paribas-cardif-claims-management/overview> и рассмотрим кейс, который является эталонным примером отбора признаков в ИЦ «Гевисста» (на примере соревнования Porto Seguro's Safe Driver Prediction с Kaggle <https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/overview>).

17.1. Методы-фильтры

Итак, методы-фильтры основываются только на статистиках признаков. Мы оцениваем, как признак работает сам по себе, а не в сочетании с другими признаками. Методы-фильтры хороши для быстрого мониторинга и удаления наименее релевантных признаков (константных, дублирующихся признаков) на самом раннем этапе отбора признаков.

17.1.1. Класс VarianceThreshold

Один из таких полезных методов-фильтров – метод отбора признаков на основе дисперсии, реализованный в классе VarianceThreshold. Он удаляет все признаки с низкой дисперсией.

```
from sklearn.feature_selection import VarianceThreshold(threshold=0.0)
```

Признаки с дисперсией ниже этого порога удаляются. По умолчанию оставляем все признаки с ненулевой дисперсией, т.е. константные признаки будут удалены

Давайте импортируем необходимые библиотеки и загрузим данные соревнования Santander Customer Satisfaction с Kaggle <https://www.kaggle.com/c/santander-customer-satisfaction>.

```
# импортируем необходимые библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold

# загружаем данные
data = pd.read_csv('Data/santander_train.csv')
```

Удаляем идентификатор и разбиваем набор данных на обучающую и тестовую выборки.

```
# удаляем ID
data.drop('ID', axis=1, inplace=True)
```

```
# разбиваем набор данных на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('TARGET', axis=1), data['TARGET'],
    test_size=0.3, random_state=42)
# смотрим количество наблюдений и признаков
print(X_train.shape, X_test.shape)

((53214, 369), (22806, 369))
```

Теперь создаем экземпляр класса `VarianceThreshold` и обучаем, по сути вычисляем дисперсии признаков.

```
# создаем экземпляр класса VarianceThreshold
sel = VarianceThreshold(threshold=0)
# обучаем модель, вычисляем дисперсии признаков
sel.fit(X_train);
```

У классов, выполняющих отбор признаков в библиотеке scikit-learn (эти классы еще называют селекторами), существует метод `.get_support()`, который возвращает булев вектор, указывающий, будет сохранена переменная или нет. Значение `True` говорит, что признак будет сохранен, значение `False` – что признак будет удален.

```
# с помощью метода .get_support() получаем булев вектор, указывающий,
# будет отобран признак или нет
# взглянем, сколько признаков не являются константами
sum(sel.get_support())
```

323

```
# еще можно так
len(X_train.columns[sel.get_support()])
```

323

```
# смотрим статус каждого признака
for i, j in zip(X_train.columns, sel.get_support()):
    print(i, j)

var3 True
var15 True
imp_ent_var16_ult1 True
imp_op_var39_comer_ult1 True
imp_op_var39_comer_ult3 True
imp_op_var40_comer_ult1 True
imp_op_var40_comer_ult3 True
imp_op_var40_efect_ult1 True
imp_op_var40_efect_ult3 True
imp_op_var40_ult1 True
. . .
```

Теперь вычислим количество признаков-констант и выведем список этих признаков-констант.

```
# теперь давайте вычислим количество признаков-констант
# и выведем эти признаки
print([
    len([
        x for x in X_train.columns
        if x not in X_train.columns[sel.get_support()]
    ])
])
[x for x in X_train.columns if x not in X_train.columns[sel.get_support()]]
```

46

```
['ind_var2_0',
 'ind_var2',
 'ind_var18_0',
 'ind_var18',
 'ind_var27_0',
 'ind_var28_0',
 'ind_var28',
 'ind_var27',
 'ind_var41',
 'ind_var46_0',
 'ind_var46',
 'num_var18_0',
 'num_var18',
 'num_var27_0',
 'num_var28_0',
 'num_var28',
 'num_var27',
 'num_var41',
 'num_var46_0',
 'num_var46',
 'saldo_var18',
 'saldo_var28',
 'saldo_var27',
 'saldo_var41',
 'saldo_var46',
 'delta_imp_amort_var18_1y3',
 'delta_imp_reemb_var33_1y3',
 'delta_num_reemb_var33_1y3',
 'imp_amort_var18_hace3',
 'imp_amort_var18_ult1',
 'imp_amort_var34_hace3',
 'imp_reemb_var13_hace3',
 'imp_reemb_var33_hace3',
 'imp_reemb_var33_ult1',
 'imp_trasp_var17_out_hace3',
 'imp_trasp_var33_out_hace3',
 'num_var2_0_ult1',
 'num_var2_ult1',
 'num_reemb_var13_hace3',
 'num_reemb_var33_hace3',
 'num_reemb_var33_ult1',
 'num_trasp_var17_out_hace3',
 'num_trasp_var33_out_hace3',
 'saldo_var2_ult1',
```

```
'saldo_medio_var13_medio_hace3',
'saldo_medio_var29_hace3']
```

Можем убедиться, что у выбранных признаков – по одному уникальному значению.

```
# давайте убедимся, что у выбранных переменных –
# по одному уникальному значению
cols = X_train.columns[sel.get_support()]
const_feat = [x for x in X_train.columns if x not in cols]
for col in const_feat:
    print(col, X_train[col].nunique())

ind_var2_0 1
ind_var2 1
ind_var18_0 1
ind_var18 1
ind_var27_0 1
ind_var28_0 1
ind_var28 1
ind_var27 1
ind_var41 1
ind_var46_0 1
ind_var46 1
num_var18_0 1
num_var18 1
num_var27_0 1
num_var28_0 1
num_var28 1
num_var27 1
num_var41 1
num_var46_0 1
num_var46 1
saldo_var18 1
saldo_var28 1
saldo_var27 1
saldo_var41 1
saldo_var46 1
delta_imp_amort_var18_1y3 1
delta_imp_reemb_var33_1y3 1
delta_num_reemb_var33_1y3 1
imp_amort_var18_hace3 1
imp_amort_var18_ult1 1
imp_amort_var34_hace3 1
imp_reemb_var13_hace3 1
imp_reemb_var33_hace3 1
imp_reemb_var33_ult1 1
imp_trasp_var17_out_hace3 1
imp_trasp_var33_out_hace3 1
num_var2_0_ult1 1
num_var2_ult1 1
num_reemb_var13_hace3 1
num_reemb_var33_hace3 1
num_reemb_var33_ult1 1
num_trasp_var17_out_hace3 1
num_trasp_var33_out_hace3 1
saldo_var2_ult1 1
```

```
saldo_medio_var13_medio_hace3 1  
saldo_medio_var29_hace3 1
```

Теперь с помощью метода `.transform()` преобразовываем наборы, по сути, удаляем переменные-константы из выборок.

```
# теперь преобразовываем наборы, по сути, удаляем  
# переменные-константы из выборок  
X_train = sel.transform(X_train)  
X_test = sel.transform(X_test)  
  
# смотрим количество наблюдений и признаков  
print(X_train.shape, X_test.shape)  
  
((53214, 323), (22806, 323))
```

Однако можно удалить константные признаки, не прибегая к классу `VarianceThreshold`. Давайте снова загрузим данные и разобьем их на обучающую и тестовую выборки.

```
# снова загружаем данные  
data = pd.read_csv('Data/santander_train.csv')  
  
# удаляем ID  
data.drop('ID', axis=1, inplace=True)  
  
# разбиваем набор данных на обучающую и тестовую выборки  
X_train, X_test, y_train, y_test = train_test_split(  
    data.drop('TARGET', axis=1),  
    data['TARGET'],  
    test_size=0.3,  
    random_state=42)  
  
# смотрим количество наблюдений и признаков  
X_train.shape, X_test.shape  
  
((53214, 369), (22806, 369))
```

Определяем константные признаки с помощью метода `.std()`, а потом удаляем их с помощью метода `.drop()`.

```
# определяем константные признаки с помощью метода .std()  
constant_features = [  
    feat for feat in X_train.columns if X_train[feat].std() == 0  
]  
len(constant_features)  
  
# удаляем эти признаки из выборок с помощью метода .drop()  
X_train.drop(labels=constant_features, axis=1, inplace=True)  
X_test.drop(labels=constant_features, axis=1, inplace=True)  
  
# смотрим количество наблюдений и признаков  
print(X_train.shape, X_test.shape)  
  
((53214, 323), (22806, 323))
```

17.1.2. Классы SelectPercentile и SelectKBest

Классы `SelectKBest` и `SelectPercentile` позволяют отбирать признаки на основе ряда статистик: взаимной информации, значения Фишера, значения хи-квадрат.

Взаимная информация измеряет взаимную зависимость двух переменных X и Y . Другими словами, она определяет, насколько точно мы можем определить переменную Y , зная переменную X . Здесь нас интересует, насколько совместное распределение $P(X, Y)$ похоже на произведение маргинальных распределений $P(X)P(Y)$. Если X и Y независимы, то взаимная информация будет равна нулю. Формула взаимной информации приведена ниже:

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(x, y) \log \frac{P(x, y)}{P(x)P(y)}.$$

Значение Фишера измеряет зависимость двух переменных и используется для категориальных переменных. Зависимая переменная должна быть бинарной. Значения переменных должны быть неотрицательными и обычно представляют собой абсолютные или относительные частоты. Значение Фишера для i -го признака вычисляется по формуле:

$$S_i = \frac{\sum_{k=1}^K n_j (\mu_{ij} - \mu_i)^2}{\sum_{k=1}^K n_j \rho_{ij}^2},$$

где μ_{ij} и ρ_{ij} – это среднее и дисперсия i -го признака в j -м классе соответственно, n_j – это количество наблюдений в j -м классе и μ_i – это среднее i -го признака. Признаки с высокими значениями Фишера характеризуются тем, что наблюдения одного и того же класса имеют схожие значения, а наблюдения разных классов будут сильно различаться по своим значениям.

Критерий хи-квадрат Пирсона проверяет нулевую гипотезу о независимости двух категориальных переменных друг от друга. Допустим, у нас есть признак *Семейное положение* с категориями Холост и Женат и зависимая переменная *Наличие просрочки* с классами Нет просрочки и Есть просрочка. Нулевая гипотеза звучит так: категории признака не отличаются друг от друга с точки зрения распределения классов зависимой переменной.

Строится двухходовая таблица сопряженности, где строки являются категориями признака *Семейное положение*, а столбцы – категориями зависимой переменной *Наличие просрочки*. Для каждой ячейки таблицы фиксируем наблюдаемую частоту O (от *observed*). Затем для каждой ячейки фиксируем ожидаемую частоту E (от *expected*) согласно нулевой гипотезе. В итоге для каждой ячейки вычисляем квадрат разности между наблюдаемой и ожидаемой частотами, поделенный на ожидаемую частоту. Складываем результаты, вычисленные по каждой ячейке, и получаем значение хи-квадрат (χ^2). Таким образом, критерий хи-квадрат Пирсона – это показатель, вычисляющий степень суммарного расхождения наблюдаемых (реальных) и ожидаемых частот. Подробнее можно посмотреть раздел 2.5 *SciPy*.

И критерий хи-квадрат, и критерий Фишера из-за использования p -значений чувствительны к размеру выборки, поэтому на больших выборках большинство признаков будут значимыми. Вы должны помнить, что низкое p -значение может указывать не на высокую прогнозную силу переменной, а просто на тот факт, что мы работаем с большим набором данных. Поэтому полезнее смотреть не на сами p -значения, а сравнивать эти p -значения между собой.

Разница между классами `SelectKBest` и `SelectPercentile` лишь в том, что класс `SelectKBest` отбирает признаки согласно k наивысшим значениям статистик, а класс `SelectPercentile` отбирает признаки согласно заданному процентилю наивысших значений статистик. Основное применение классов – сокращение признакового пространства для линейных моделей без потери качества.

```
from sklearn.feature_selection import SelectKBest(score_func=f_classif, k=10)
    задает k отбираемых признаков → k=10
from sklearn.feature_selection import SelectPercentile(score_func=f_classif, percentile=10)
    задает процент оставляемых признаков → percentile=10
```

задает вызываемую функцию, которая принимает массив признаков и массив меток и возвращает либо пару массивов (массив значений статистики и p -значений), либо один массив значений статистики. По умолчанию используется значение Фишера для классификации (`f_classif`)

17.1.3. Одномерный отбор признаков по метрике

Процедура одномерного отбора признаков по метрике работает следующим образом:

- строим модель дерева решений / модель логистической регрессии / модель линейной регрессии с одним интересующим признаком;
- получаем прогнозы с помощью модели;
- получаем AUC-ROC или RMSE (могут быть и другие метрики) для признака;
- ранжируем признаки по убыванию AUC-ROC или возрастанию RMSE и отбираем лучшие.

Проиллюстрируем отбор признаков по AUC-ROC на примере данных соревнования Santander Customer Satisfaction с Kaggle <https://www.kaggle.com/c/santander-customer-satisfaction>.

Импортируем необходимые библиотеки и загрузим данные соревнования.

```
# импортируем необходимые библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
# увеличиваем количество строк вывода
pd.set_option('display.max_rows', 400)
# загружаем данные
data = pd.read_csv('Data/santander_train.csv')
```

Удаляем идентификатор и разбиваем набор данных на обучающую и тестовую выборки.

```
# удаляем ID
data.drop('ID', axis=1, inplace=True)
```

```
# разбиваем набор данных на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('TARGET', axis=1),
    data['TARGET'],
    test_size=0.3,
    random_state=42)
# смотрим количество наблюдений и признаков
print(X_train.shape, X_test.shape)

((53214, 369), (22806, 369))
```

Пишем функцию `importance_auc()`, которая проранжирует наши признаки по убыванию AUC-ROC (будут использованы модель логистической регрессии и перекрестная проверка), не забыв при этом импортировать необходимые для нее классы и функции.

```
# импортируем необходимые классы и функции
# для нашей функции importance_auc()
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

# пишем функцию, вычисляющую AUC для модели с одним признаком
def importance_auc(train, y_train, imp_strategy):
    # задаем количество десятичных разрядов
    pd.set_option('display.precision', 3)
    # создаем копию обучающего набора
    train_copy = train.copy()
    # создаем список признаков
    col_list = train_copy.select_dtypes(include=['number']).columns
    # создаем пустой список, в который будем записывать значения AUC
    auc_list = []
    # вычисляем AUC для модели с одним признаком,
    # используя перекрестную проверку
    for i in col_list:
        pipe = Pipeline([('imputer', SimpleImputer(strategy=imp_strategy)),
                        ('scaler', StandardScaler()),
                        ('logreg', LogisticRegression(solver='liblinear'))])
        auc = cross_val_score(pipe, train_copy[[i]], y_train,
                              scoring='roc_auc', cv=5).mean()
        # добавляем значение AUC в список
        auc_list.append(auc)
    # превращаем список со значениями AUC в серию
    auc_values = pd.Series(auc_list)
    # метками индекса будут имена признаков
    auc_values.index = col_list
    # сортируем по убыванию
    auc_values = auc_values.sort_values(ascending=False)
    return auc_values
```

Применяем нашу функцию `importance_auc()` и смотрим результаты.

```
# применяем функцию, вычисляя значения AUC
auc_values = importance_auc(X_train, y_train, 'median')
auc_values
```

var15	0.698
saldo_var30	0.697
num_meses_var5_ult3	0.694
num_var30	0.682
saldo_var42	0.682
saldo_var5	0.677
saldo_medio_var5_hace2	0.677
saldo_medio_var5_ult3	0.676
saldo_medio_var5_ult1	0.676
ind_var30	0.675
saldo_medio_var5_hace3	0.672
num_var42	0.672
ind_var5	0.667
num_var5	0.667
num_var35	0.648
num_var4	0.648
var36	0.619
var38	0.585
...	...

Давайте отберем столбцы с AUC-ROC выше 0.5 и вычислим количество отобранных столбцов.

```
# отбираем столбцы с AUC выше 0.5
select_columns = auc_values.index[auc_values > 0.5].tolist()
# вычисляем количество отобранных столбцов
len(select_columns)
```

233

Здесь необходимо учитывать, что одномерный отбор по AUC-ROC не учитывает корреляции между признаками, корреляции признаков с зависимой переменной, дисперсию признаков, поэтому одномерный отбор будет полезнее после удаления константных признаков, дублирующих признаков.

17.2. ПРИМЕНЕНИЕ МЕТОДА-ФИЛЬТРА И ВСТРОЕННОГО МЕТОДА ДЛЯ ОТБОРА ПРИЗНАКОВ (НА ПРИМЕРЕ СОРЕВНОВАНИЯ BNP PARIBAS CARDIF CLAIMS MANAGEMENT С KAGGLE)

Давайте возьмем данные соревнования BNP Paribas Cardif Claims Management <https://www.kaggle.com/c/bnp-paribas-cardif-claims-management/overview> с Kaggle. Импортируем необходимые библиотеки, классы и функции и загрузим исторический набор данных.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/paribas_train.csv', sep=';')
data.head()
```

ID	target	v1	v2	v3	v4	v5	v6	v7	v8	...	v122	v123	v124	v125	v126	v127	v128	
0	3	1	1.335739	8.727474	C	3.921026	7.915266	2.599278	3.176895	0.012941	...	8.000000	1.989780	0.035754	AU	1.804126	3.113719	2.024285
1	4	1	NaN	NaN	C	NaN	9.191265	NaN	NaN	2.301630	...	NaN	NaN	0.598896	AF	NaN	NaN	1.957825
2	5	1	0.943877	5.310079	C	4.410969	5.326159	3.979592	3.928571	0.019645	...	9.333333	2.477596	0.013452	AE	1.773709	3.922193	1.120468
3	6	1	0.797415	8.304757	C	4.225930	11.627438	2.097700	1.987549	0.171947	...	7.018256	1.812795	0.002267	CJ	1.415230	2.954381	1.990847
4	8	1	NaN	NaN	C	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	Z	NaN	NaN	NaN

Взглянем на количество наблюдений и количество признаков.

```
# смотрим количество наблюдений
# и количество признаков
print(data.shape)
```

(114321, 133)

Как видим, данные анонимизированы. У нас – 133 переменные и примерно 114 000 наблюдений.

Удалим идентификатор и создадим пустые списки, в которые запишем для каждой переменной количество уникальных значений, количество пропусков, тип переменной.

```
# удаляем переменную ID
data.drop('ID', axis=1, inplace=True)

# создаем пустые списки, в которые для каждой переменной
# записываем количество уникальных значений, количество
# пропусков, тип переменной
nunique_list = []
miss_list = []
type_list = []
for col in data.columns:
    nunique_list.append(data[col].nunique())
    miss_list.append(data[col].isnull().sum())
    type_list.append(data[col].dtypes)
```

Увеличиваем количество выводимых строк и создаем датафрейм с информацией о количестве уникальных значений, количестве пропусков, типе переменной на основе заполненных списков.

```
# увеличиваем количество выводимых строк
pd.set_option('display.max_rows', 140)
# создаем датафрейм с информацией о количестве уникальных
# значений, количестве пропусков, типе переменной
feat_labels = data.columns
summary = np.array([nunique_list, miss_list, type_list])
columns = ['nunique', 'missing', 'type']
results = pd.DataFrame(summary.T,
                       index=feat_labels,
                       columns=columns)
results
```

	nunique	missing	type
target	2	0	int64
v1	64487	49832	float64
v2	64524	49796	float64
v3	3	3457	object
v4	64524	49796	float64
v5	65671	48624	float64
v6	64487	49832	float64
v7	64489	49832	float64
v8	65688	48619	float64
v9	64451	49851	float64

Видим, что у нас есть пропуски, помимо количественных переменных есть несколько категориальных, при этом некоторые категориальные переменные являются высококардинальными.

Импутируем пропуски значением вне диапазона.

```
# импутируем пропуски
data.fillna(-9999, inplace=True)
```

Разбиваем набор на обучающую и тестовую выборки.

```
# разбиваем набор на обучающую и тестовую
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis=1),
    data['target'],
    test_size=0.3,
    stratify=data['target'],
    random_state=42)
```

Увеличиваем количество отображаемых столбцов и выведем первые пять строк матрицы корреляций для количественных признаков.

```
# увеличиваем количество отображаемых столбцов
pd.set_option('display.max_columns', 150)
# выводим матрицу корреляций
corr = X_train.corr()
corr.head()
```

	v1	v2	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15	v16
v1	1.000000	0.999315	0.999315	0.947288	1.000000	1.000000	0.947393	0.999619	0.022242	0.999924	0.022906	1.000000	0.005391	0.999924	0.998909
v2	0.999315	1.000000	1.000000	0.947958	0.999315	0.999315	0.948062	0.998934	0.022263	0.999239	0.022928	0.999315	0.005398	0.999239	0.998224
v4	0.999315	1.000000	1.000000	0.947958	0.999315	0.999315	0.948062	0.998934	0.022262	0.999239	0.022926	0.999315	0.005397	0.999239	0.998224
v5	0.947288	0.947958	0.947958	1.000000	0.947289	0.947288	0.999898	0.946916	0.021939	0.947214	0.022639	0.947288	0.005573	0.947214	0.946224
v6	1.000000	0.999315	0.999315	0.947289	1.000000	1.000000	0.947393	0.999619	0.022242	0.999924	0.022906	1.000000	0.005391	0.999924	0.998909

Видим, что у нас очень много идеально скоррелированных количественных признаков. Введем критерий важности количественного признака: важным является признак, у которого будет наименьшая сумма корреляций с остальными признаками.

```
# выделяем наиболее важные количественные признаки - те,
# у которых сумма корреляций с остальными была наименьшей
for col in corr.columns:
    print(col, corr[col].sum(axis=0))

v1 99.07717708071425
v2 99.05777616046353
v4 99.05772612661347
v5 96.02502215200258
v6 99.07718573728903
v7 99.07719145780698
v8 96.03328950620445
v9 99.05227072633997
v10 7.596528760721377
v11 99.07493693361639
v12 7.689686069140725
v13 99.07716443574643
v14 3.008223831271456
v15 99.07491902615293
v16 98.9893482534167
v17 99.0577192817849
v18 99.07718093373879
v19 99.06449911186405
v20 99.07131668968174
v21 3.4438744609608825
. . .
```

Давайте выполним отбор признаков. Отберем признаки, у которых сумма корреляций с остальными признаками меньше 10, и сформируем список. По сути, мы выполнили отбор на основе метода-фильтра, критерием здесь у нас стала корреляция. Давайте выведем этот список важных переменных на основе корреляций.

```
# формируем список важных переменных на основе корреляций
corr_select_cols = ['v10', 'v12', 'v14', 'v21', 'v34', 'v38',
                     'v40', 'v50', 'v62', 'v72', 'v114', 'v129']
```

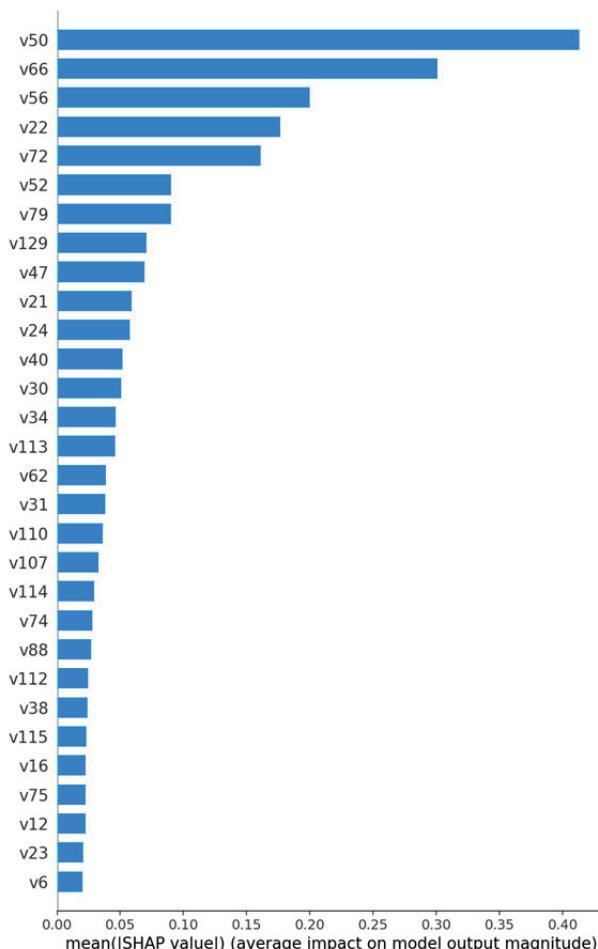
Теперь получаем индексы категориальных признаков, формируем обучающий пул, обучаем модель CatBoost, вычисляем важности признаков по SHAP и выводим график 30 наиболее важных признаков на основе усредненных значений SHAP. Значения SHAP являются адаптацией теории игр для применения к моделям машинного обучения, вычисляются для каждого признака и каждого наблюдения. Они измеряют величину вклада, который признак вносит в выход модели для определенного наблюдения. Все гиперпараметры были предварительно найдены с помощью обычного поиска по сетке на основе комбинированной проверки, запущенной на обучающей выборке (для экономии места не приводится).

```
# формируем массив индексов категориальных признаков
categorical_features_ind = np.where(X_train.dtypes != float)[0]
categorical_features_ind

array([ 2,  21,  23,  29,  30,  37,  46,  51,  55,  61,  65,  70,  71,
       73,  74,  78,  90, 106, 109, 111, 112, 124, 128])
```

```
# формируем обучающий пул
train_pool = Pool(
    X_train,
    y_train,
    cat_features=categorical_features_ind)
# создаем экземпляр класса CatBoostClassifier
clf = CatBoostClassifier(learning_rate=0.08,
                        iterations=1200,
                        random_strength=0.15,
                        random_seed=0,
                        model_size_reg=0.1,
                        logging_level='Silent')

# обучаем модель
clf.fit(train_pool)
# вычисляем важности по SHAP
shap_values = clf.get_feature_importance(train_pool, 'ShapValues')
shap_values = shap_values[:, :-1]
# выводим график 30 наиболее важных признаков по SHAP
shap.summary_plot(shap_values, X_train, plot_type='bar', max_display=30)
```



Давайте сформируем список важных переменных на основе значений SHAP, но только так, чтобы они не дублировались с переменными, отобранными на основе корреляций. Мы выполнили отбор признаков с помощью встроенного метода, воспользовавшись важностями на основе SHAP.

```
# формируем список важных переменных на основе SHAP
shap_select_cols = ['v66', 'v56', 'v22', 'v79', 'v52', 'v24',
                     'v47', 'v113', 'v74', 'v31', 'v110', 'v30']
```

Теперь объединяем два списка в один.

```
# объединяем два созданных списка в один
select_cols = corr_select_cols + shap_select_cols
```

Теперь мы готовим посылку для Kaggle, дополнительно добавив парных взаимодействий на основе признаков со строковыми и целочисленными значениями, являющимися наиболее важными по SHAP.

```
# загружаем наборы
train = pd.read_csv('Data/paribas_train.csv')
test = pd.read_csv('Data/paribas_test.csv')

# формируем массив меток и массив признаков
labels = train.pop('target')

# сохраняем ID набора новых данных
test_id = test['ID']

# удаляем ID
train.drop('ID', axis=1, inplace=True)
test.drop('ID', axis=1, inplace=True)

# имплементируем пропуски
train.fillna(-9999, inplace=True)
test.fillna(-9999, inplace=True)

# формируем новые массивы признаков
train = train[select_cols]
test = test[select_cols]

# добавляем парные взаимодействия на основе признаков
# со строковыми и целочисленными значениями,
# являющимися наиболее важными по SHAP
train['v129 + v66'] = train.apply(
    lambda x: f"{x['v129']} + {x['v66']}",
    axis=1)

train['v66 + v56'] = train.apply(
    lambda x: f"{x['v66']} + {x['v56']}",
    axis=1)

train['v56 + v22'] = train.apply(
    lambda x: f"{x['v56']} + {x['v22']}",
    axis=1)

train['v22 + v79'] = train.apply(
    lambda x: f"{x['v22']} + {x['v79']}",
    axis=1)

train['v79 + v52'] = train.apply(
    lambda x: f"{x['v79']} + {x['v52']}",
    axis=1)
```

```
train['v52 + v24'] = train.apply(  
    lambda x: f"{x['v52']} + {x['v24']}" ,  
    axis=1)  
  
train['v24 + v47'] = train.apply(  
    lambda x: f"{x['v24']} + {x['v47']}" ,  
    axis=1)  
  
train['v66 + v72'] = train.apply(  
    lambda x: f"{x['v66']} + {x['v72']}" ,  
    axis=1)  
  
test['v129 + v66'] = test.apply(  
    lambda x: f"{x['v129']} + {x['v66']}" ,  
    axis=1)  
  
test['v66 + v56'] = test.apply(  
    lambda x: f"{x['v66']} + {x['v56']}" ,  
    axis=1)  
  
test['v56 + v22'] = test.apply(  
    lambda x: f"{x['v56']} + {x['v22']}" ,  
    axis=1)  
  
test['v22 + v79'] = test.apply(  
    lambda x: f"{x['v22']} + {x['v79']}" ,  
    axis=1)  
  
test['v79 + v52'] = test.apply(  
    lambda x: f"{x['v79']} + {x['v52']}" ,  
    axis=1)  
  
test['v52 + v24'] = test.apply(  
    lambda x: f"{x['v52']} + {x['v24']}" ,  
    axis=1)  
  
test['v24 + v47'] = test.apply(  
    lambda x: f"{x['v24']} + {x['v47']}" ,  
    axis=1)  
  
test['v66 + v72'] = test.apply(  
    lambda x: f"{x['v66']} + {x['v72']}" ,  
    axis=1)  
  
# формируем массив индексов категориальных признаков  
cat_features_ids = np.where(train.dtypes != float)[0]  
  
# формируем обучающий пул  
train_pool = Pool(train, labels, cat_features=cat_features_ids)  
  
# создаем экземпляр класса CatBoostClassifier  
clf_full = CatBoostClassifier(learning_rate=0.06,  
                             iterations=1600,  
                             random_strength=0.15,  
                             model_size_reg=0.1,  
                             random_seed=0,  
                             logging_level='Silent')
```

```
# обучаем модель
clf_full.fit(train_pool)

# вычисляем вероятности
proba = clf_full.predict_proba(test)[:, 1]

# формируем посылку для Kaggle (10-е место на привате)
pd.DataFrame({'ID': test_id, 'PredictedProb': proba}).to_csv(
    'subm_paribas.csv', index=False)
```

Private Score	Public Score
0.42940	0.43203

Получаем результат, соответствующий примерно 10-му месту на приватном лидерборде.

17.3. Комбинирование нескольких методов для отбора признаков (на примере соревнования Porto Seguro's Safe Driver Prediction с Kaggle)

Ничто так не испортит острые ощущения от покупки нового автомобиля, как стоимость страховки. Переживания становятся еще более болезненными, когда ты знаешь, что являешься хорошим водителем. Кажется несправедливым, что вам нужно платить так много, если вы годами осторожничали на дороге. С этим полностью согласна одна из крупнейших в Бразилии компаний по страхованию автомобилей Porto Seguro. Неточности в прогнозах страховых компаний по страхованию автомобилей увеличивают стоимость страховки для хороших водителей и снижают стоимость для плохих.

В этом соревновании вам предстоит построить модель, которая прогнозирует вероятность того, что водитель продлит автостраховку в следующем году. Хотя Porto Seguro использует машинное обучение в течение последних 20 лет, они надеются, что сообщество Kaggle по машинному обучению предложит новые, более эффективные методы. Более точный прогноз позволит им еще больше адаптировать свои цены и, надеюсь, сделает автострахование более доступным для большего числа водителей.

С этой преамбулы начинается соревнование Porto Seguro's Safe Driver Prediction <https://www.kaggle.com/c/porto-seguro-safe-driver-prediction> с Kaggle, на нем мы и проиллюстрируем, как отбор признаков позволяет попасть в топ-20. Оптимизирующей метрикой будет нормализованный коэффициент Джини. Для ее оптимизации вполне подойдет максимизация AUC-ROC.

Импортируем необходимые библиотеки.

```
# загружаем необходимые библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.model_selection import (train_test_split,
                                      cross_val_score,
                                      cross_validate)
```

```
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import GridSearchCV
from catboost import CatBoostClassifier, Pool
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from rfpimp import (feature_dependence_matrix,
                    plot_dependence_heatmap,
                    plot_corr_heatmap)
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

# отключаем экспоненциальное представление и увеличиваем
# максимальное количество отображаемых столбцов
pd.set_option('display.float_format', lambda x: '%.8f' % x)
pd.set_option('display.max_columns', 60)
```

Теперь загружаем исторический набор. У нас – 58 переменных и примерно 595 000 наблюдений.

```
# загружаем данные
data = pd.read_csv('Data/porto_seguro_train.csv')
data.head()
```

	<code>id</code>	<code>target</code>	<code>ps_ind_01</code>	<code>ps_ind_02_cat</code>	<code>ps_ind_03</code>	<code>ps_ind_04_cat</code>	<code>ps_ind_05_cat</code>	<code>ps_ind_06_bin</code>	<code>ps_ind_07_bin</code>	<code>ps_ind_08_bin</code>	<code>ps_ind_09_bin</code>	<code>ps_ind_10_bin</code>
0	7	0	2	2	5	1	0	0	1	0	0	0
1	9	0	1	1	7	0	0	0	0	1	0	0
2	13	0	5	4	9	1	0	0	0	1	0	0
3	16	0	0	1	2	0	0	1	0	0	0	0
4	17	0	0	2	0	1	0	1	0	0	0	0

Удалим идентификатор, убедимся в отсутствии пропусков и посмотрим типы данных.

```
# удаляем id
data.drop('id', axis=1, inplace=True)
# смотрим пропуски и типы данных
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 595212 entries, 0 to 595211
Data columns (total 58 columns):
 #   Column           Non-Null Count  Dtype  
 ---  -- 
 0   target          595212 non-null   int64  
 1   ps_ind_01        595212 non-null   int64  
 2   ps_ind_02_cat    595212 non-null   int64  
 3   ps_ind_03        595212 non-null   int64  
 4   ps_ind_04_cat    595212 non-null   int64  
 5   ps_ind_05_cat    595212 non-null   int64  
 6   ps_ind_06_bin    595212 non-null   int64  
 7   ps_ind_07_bin    595212 non-null   int64  
 8   ps_ind_08_bin    595212 non-null   int64  
 9   ps_ind_09_bin    595212 non-null   int64
```

```

10 ps_ind_10_bin    595212 non-null  int64
11 ps_ind_11_bin    595212 non-null  int64
12 ps_ind_12_bin    595212 non-null  int64
13 ps_ind_13_bin    595212 non-null  int64
14 ps_ind_14        595212 non-null  int64
15 ps_ind_15        595212 non-null  int64
16 ps_ind_16_bin    595212 non-null  int64
17 ps_ind_17_bin    595212 non-null  int64
18 ps_ind_18_bin    595212 non-null  int64
19 ps_reg_01         595212 non-null  float64
20 ps_reg_02         595212 non-null  float64
21 ps_reg_03         595212 non-null  float64
22 ps_car_01_cat    595212 non-null  int64
23 ps_car_02_cat    595212 non-null  int64
24 ps_car_03_cat    595212 non-null  int64
25 ps_car_04_cat    595212 non-null  int64
26 ps_car_05_cat    595212 non-null  int64
27 ps_car_06_cat    595212 non-null  int64
28 ps_car_07_cat    595212 non-null  int64
29 ps_car_08_cat    595212 non-null  int64
30 ps_car_09_cat    595212 non-null  int64
31 ps_car_10_cat    595212 non-null  int64
32 ps_car_11_cat    595212 non-null  int64
33 ps_car_11         595212 non-null  int64
34 ps_car_12         595212 non-null  float64
35 ps_car_13         595212 non-null  float64
36 ps_car_14         595212 non-null  float64
37 ps_car_15         595212 non-null  float64
38 ps_calc_01         595212 non-null  float64
39 ps_calc_02         595212 non-null  float64
40 ps_calc_03         595212 non-null  float64
41 ps_calc_04         595212 non-null  int64
42 ps_calc_05         595212 non-null  int64
43 ps_calc_06         595212 non-null  int64
44 ps_calc_07         595212 non-null  int64
45 ps_calc_08         595212 non-null  int64
46 ps_calc_09         595212 non-null  int64
47 ps_calc_10         595212 non-null  int64
48 ps_calc_11         595212 non-null  int64
49 ps_calc_12         595212 non-null  int64
50 ps_calc_13         595212 non-null  int64
51 ps_calc_14         595212 non-null  int64
52 ps_calc_15_bin     595212 non-null  int64
53 ps_calc_16_bin     595212 non-null  int64
54 ps_calc_17_bin     595212 non-null  int64
55 ps_calc_18_bin     595212 non-null  int64
56 ps_calc_19_bin     595212 non-null  int64
57 ps_calc_20_bin     595212 non-null  int64
dtypes: float64(10), int64(48)
memory usage: 263.4 MB

```

Формально пропусков нет, однако некоторые участники соревнования рассматривали значения -1 в переменных как пропуски и применяли разные стратегии импутации, но существенного прироста качества это не давало.

Разбиваем набор на обучающую и тестовую выборки.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis=1),
    data['target'],
    test_size=0.3,
    stratify=data['target'],
    random_state=42)
```

Теперь нужно вычислить важности признаков на основе информационного выигрыша с помощью деревьев градиентного бустинга LightGBM. Чтобы оценки важностей были надежными, нужно настроить гиперпараметры градиентного бустинга и постараться максимизировать метрику AUC-ROC.

Сначала ищем оптимальный темп обучения для зафиксированного количества деревьев с помощью обычного поиска по сетке.

```
# создаем экземпляр класса LGBMClassifier
lgbm_model = LGBMClassifier(random_state=42,
                            n_estimators=300)

# задаем сетку гиперпараметров
param_grid = {
    'learning_rate': [0.01, 0.05, 0.1]
}

# создаем экземпляр класса GridSearchCV, передав
# конвейер, сетку гиперпараметров и указав
# количество блоков перекрестной проверки
gs = GridSearchCV(lgbm_model,
                   param_grid,
                   scoring='roc_auc',
                   cv=5)

# выполняем поиск по всем значениям сетки
gs.fit(X_train, y_train);

# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров: {}".format(
    gs.best_params_))
# смотрим наилучшее значение AUC
print("Наилучшее значение AUC: {:.3f}".format(
    gs.best_score_))
```

Наилучшие значения гиперпараметров: {'learning_rate': 0.01}
Наилучшее значение AUC: 0.633

Теперь с найденным оптимальным темпом обучения для зафиксированного количества деревьев ищем с помощью обычного поиска по сетке оптимальные значения гиперпараметров `lambda_l1`, `bagging_fraction`, `feature_fraction`. Гиперпараметр `lambda_l1` задает штрафной коэффициент перед L1-нормой вектора весов листьев (по умолчанию 0). Гиперпараметр `bagging_fraction` задает случайный отбор наблюдений без возвращения. Он может принимать значения от 0 до 1 (по умолчанию 1, т. е. не используется). Гипер-

параметр `feature_fraction` задает случайный отбор признаков для каждого дерева. Он может принимать значения от 0 до 1 (по умолчанию 1, т. е. не используется).

Особенность этого соревнования заключалась в том, что большее значение `lambda_l1` улучшало результат.

```
# создаем экземпляр класса LGBMClassifier
lgbm_model2 = LGBMClassifier(random_state=42,
                             n_estimators=300,
                             learning_rate=0.01)

# задаем сетку гиперпараметров
param_grid2 = {
    'lambda_l1': [0, 10],
    'bagging_fraction': [0.5, 1],
    'feature_fraction': [0.5, 1]
}

# создаем экземпляр класса GridSearchCV, передав
# конвейер, сетку гиперпараметров и указав
# количество блоков перекрестной проверки
gs2 = GridSearchCV(lgbm_model2,
                    param_grid2,
                    scoring='roc_auc',
                    cv=5)

# выполняем поиск по всем значениям сетки
gs2.fit(X_train, y_train);

# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров: {}".format(
    gs2.best_params_))
# смотрим наилучшее значение AUC
print("Наилучшее значение AUC: {:.3f}".format(
    gs2.best_score_))

Наилучшие значения гиперпараметров: {'bagging_fraction': 0.5,
                                         'feature_fraction': 1,
                                         'lambda_l1': 10}
Наилучшее значение AUC: 0.634
```

Теперь вычислим важности признаков на основе информационного выигрыша. Если говорить упрощенно: информационный выигрыш – это уменьшение функции потерь, получаемое в результате применения данного признака в качестве признака, по которому происходит разбиение узла дерева. Для надежности воспользуемся 5-блочной перекрестной проверкой: построим пять моделей LightGBM, вычислим на основе каждой модели важности признаков и усредним.

```
# создаем экземпляр класса LGBMClassifier
model_all_features = LGBMClassifier(
    random_state=42, learning_rate=0.01,
    n_estimators=300, bagging_fraction=0.5,
    feature_fraction=1, lambda_l1=10,
    importance_type='gain')
```

```

# выполняем перекрестную проверку и сохраняем результат
# с помощью функции cross_validate()
output = cross_validate(
    model_all_features, X_train, y_train, cv=5,
    scoring='roc_auc', return_estimator=True)

# создаем список fi, в который будем сохранять
# важности признаков, и сохраняем в него важности,
# рассчитанные для каждой из моделей
fi = []
for estimator in output['estimator']:
    fi.append(estimator.feature_importances_)

# преобразовываем список в датафрейм, индексы в котором
# будут именами наших переменных
fi = pd.DataFrame(
    np.array(fi).T,
    columns=['importance ' + str(idx)
              for idx in range(len(fi))],
    index=X_train.columns)

# вычисляем усредненные важности и добавляем столбец с ними
fi['mean_importance'] = fi.mean(axis=1)
# смотрим полученный датафрейм
fi

```

	importance 0	importance 1	importance 2	importance 3	importance 4	mean_importance
ps_ind_01	3634.11843014	3172.45801353	4035.87575197	3279.06987143	2702.83714104	3364.87184162
ps_ind_02_cat	1180.81135607	1882.19693899	1775.92964697	2375.23713636	2278.47166729	1898.52934914
ps_ind_03	13521.97072601	12791.99010944	11509.53043318	11959.67374229	12108.93555784	12378.42011375
ps_ind_04_cat	1130.56435776	1201.73512697	1314.90494204	904.47342062	798.09325933	1069.95422134
ps_ind_05_cat	16222.12306643	16123.02086258	15932.37066174	16057.97703266	17996.74093056	16466.44651079
ps_ind_06_bin	7848.01708078	7344.12867975	7628.59964228	6914.00448751	8448.97758007	7636.74549408
ps_ind_07_bin	1684.69169283	2116.86437941	1721.08208227	1947.20034266	1518.87499571	1797.74269857
ps_ind_08_bin	797.62602758	294.66852856	602.52638626	445.62016964	395.38190889	507.16460419
ps_ind_09_bin	2757.77236462	2649.19727755	2396.16355896	2329.34756851	3267.23337460	2679.94282885
ps_ind_10_bin	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
ps_ind_11_bin	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
ps_ind_12_bin	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
ps_ind_13_bin	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
ps_ind_14	0.00000000	0.00000000	0.00000000	9.70009995	10.99250031	4.13852005
ps_ind_15	8445.80825377	7259.77261543	7811.67089462	8883.37784529	9142.92332029	8308.71058588
ps_ind_16_bin	3594.88074923	3509.95378923	3273.56424665	1337.86866140	2532.19668770	2849.69282684
ps_ind_17_bin	14058.08708429	13347.51616716	11947.90276051	13520.95722485	12732.10676146	13121.31399965
ps_ind_18_bin	12.81997013	115.63325977	126.38013935	71.77380085	108.96996021	87.11542606
ps_reg_01	5313.95518112	5947.96767426	5856.15704250	5880.09585094	4874.63846684	5574.56284313
ps_reg_02	5457.37255430	6061.99634600	4475.55018902	5560.93921518	5275.02103043	5366.17586699
ps_reg_03	13175.25240374	13797.13337278	15403.81610060	14623.83505630	13464.07024002	14092.82143469
ps_car_01_cat	3048.85453653	3279.79331398	2844.39936447	4449.36163282	3023.27647734	3329.13706503
ps_car_02_cat	731.62481499	842.38870907	640.21889114	471.59033108	882.89957762	713.74446478
ps_car_03_cat	4622.82691002	5898.57451391	6476.14354229	4333.57600927	5972.97319651	5460.81883440
ps_car_04_cat	1345.02040720	849.83740425	447.30497980	607.29100323	1341.97853994	918.28646688
ps_car_05_cat	596.99902201	178.20328045	572.34245872	423.83154249	322.36361980	418.74798470

ps_car_06_cat	743.44324923	800.87292004	653.12492228	706.45074177	680.93659067	716.96568480
ps_car_07_cat	8949.11175203	8697.74657440	8095.87302923	7388.46882868	9609.84132004	8548.20830088
ps_car_08_cat	9.82437992	38.43260050	57.59858036	0.00000000	98.43255997	40.85762415
ps_car_09_cat	1700.02721024	776.02667904	1109.60206079	1339.81728792	912.65734291	1167.62611618
ps_car_10_cat	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
ps_car_11_cat	1549.07591629	1560.96250820	1420.49286890	1562.57766438	1961.22403717	1610.86659899
ps_car_11	1633.93474150	1846.69763565	1256.76338482	1244.98663664	1234.70482016	1443.41744375
ps_car_12	635.36775637	607.98078775	775.08933735	371.59271049	796.89216185	637.38455076
ps_car_13	38443.67880917	37352.31745291	38156.90388393	41013.99278545	39509.88973284	38895.35653286
ps_car_14	3294.28320217	2851.42403460	3094.54500341	3439.44725466	3251.40136528	3186.22017202
ps_car_15	2038.48745918	3657.93068504	2475.40875006	2183.72364187	2958.50813818	2662.81173487
ps_calc_01	698.90069008	527.43793917	470.01878119	602.04200172	944.04997349	648.48987713
ps_calc_02	637.03396845	645.58579874	299.59928131	505.12669897	575.13375092	532.49589968
ps_calc_03	880.55225801	929.55533314	653.85595226	358.40529919	690.98340845	702.67045021
ps_calc_04	264.42786074	149.28655910	373.20282984	347.31832933	447.77858019	316.40283184
ps_calc_05	385.40248775	360.96417141	415.62386179	613.87396288	580.15139914	471.20317659
ps_calc_06	410.08880901	200.53073978	364.16458941	266.48433924	194.00913095	287.05552168
ps_calc_07	450.10324001	113.19445133	150.43614864	356.91799974	382.50057173	290.63048229
ps_calc_08	449.09108925	134.11524010	334.73910904	409.93175125	201.58351994	305.89214191
ps_calc_09	549.81587887	558.50692987	288.15635061	496.33184910	586.82808638	495.92781897
ps_calc_10	978.49474096	598.23471832	585.14284086	815.43182039	640.98062992	723.65695009
ps_calc_11	203.24930859	481.55242968	600.03047752	217.78520155	483.35039282	397.19356203
ps_calc_12	287.20431042	470.84547949	580.42349195	334.98449945	278.20474100	390.33250446
ps_calc_13	264.16829872	331.12765169	449.34706974	336.49524927	206.20591927	317.46883774
ps_calc_14	909.32447147	407.63785839	767.45191336	938.87422347	1165.17498064	837.69268947
ps_calc_15_bin	18.50564003	86.36724043	11.13369989	33.72865009	0.00000000	29.94704609
ps_calc_16_bin	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
ps_calc_17_bin	31.14730024	41.15370941	8.08759975	117.66150093	19.89355946	43.58873396
ps_calc_18_bin	0.00000000	0.00000000	105.73935986	0.00000000	28.69632006	26.88713598
ps_calc_19_bin	12.93319988	0.00000000	9.70132017	123.88676071	0.00000000	29.30425615
ps_calc_20_bin	0.00000000	26.59291983	19.27118015	90.37017012	0.00000000	27.24685402

Здесь мы видим, что некоторые признаки имеют нулевые или очень низкие важности. Речь идет о переменных `ps_ind_10_bin`, `ps_ind_11_bin`, `ps_ind_12_bin`, `ps_ind_13_bin`, `ps_ind_14`, `ps_calc_16_bin`, `ps_car_10_cat`. Эти признаки можно рассматривать для удаления первыми.

Теперь записываем серию, в которой индексные метки – признаки, значения – усредненные важности на основе информационного выигрыша. Затем сортируем индексные метки (признаки) по возрастанию усредненных важностей.

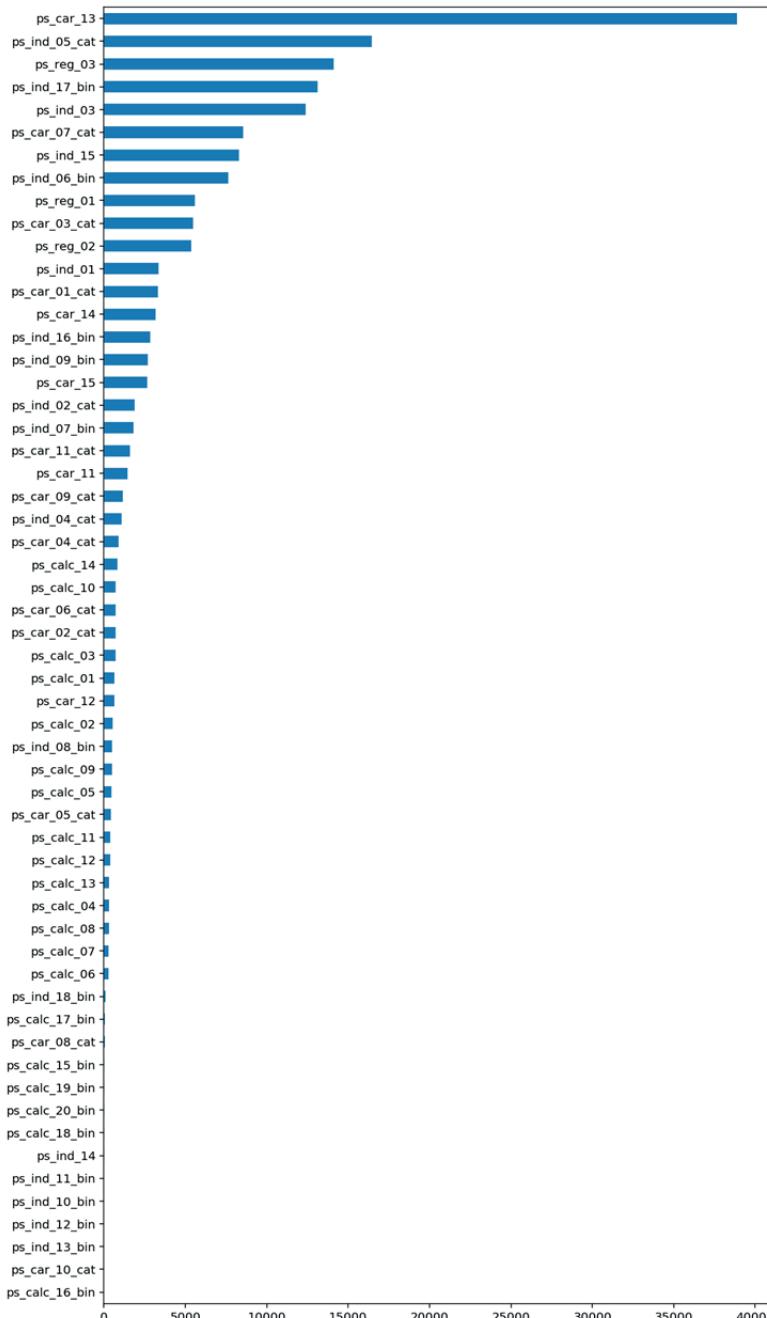
```
# записываем серию, в которой индексные метки - признаки,
# значения - важности
features = fi['mean_importance']
# сортируем индексные метки по возрастанию важностей
features = features.sort_values(ascending=True)
features

ps_calc_16_bin      0.00000000
ps_car_10_cat       0.00000000
```

```
ps_ind_13_bin      0.00000000
ps_ind_12_bin      0.00000000
ps_ind_10_bin      0.00000000
ps_ind_11_bin      0.00000000
ps_ind_14          4.13852005
ps_calc_18_bin     26.88713598
ps_calc_20_bin     27.24685402
ps_calc_19_bin     29.30425615
ps_calc_15_bin     29.94704609
ps_car_08_cat      40.85762415
ps_calc_17_bin     43.58873396
ps_ind_18_bin      87.11542606
ps_calc_06          287.05552168
ps_calc_07          290.63048229
ps_calc_08          305.89214191
ps_calc_04          316.40283184
ps_calc_13          317.46883774
ps_calc_12          390.33250446
ps_calc_11          397.19356203
ps_car_05_cat      418.74798470
ps_calc_05          471.20317659
ps_calc_09          495.92781897
ps_ind_08_bin      507.16460419
ps_calc_02          532.49589968
ps_car_12          637.38455076
ps_calc_01          648.48987713
ps_calc_03          702.67045021
ps_car_02_cat      713.74446478
ps_car_06_cat      716.96568480
ps_calc_10          723.65695009
ps_calc_14          837.69268947
ps_car_04_cat      918.28646688
ps_ind_04_cat      1069.95422134
ps_car_09_cat      1167.62611618
ps_car_11          1443.41744375
ps_car_11_cat      1610.86659899
ps_ind_07_bin      1797.74269857
ps_ind_02_cat      1898.52934914
ps_car_15          2662.81173487
ps_ind_09_bin      2679.94282885
ps_ind_16_bin      2849.69282684
ps_car_14          3186.22017202
ps_car_01_cat      3329.13706503
ps_ind_01          3364.87184162
ps_reg_02          5366.17586699
ps_car_03_cat      5460.81883440
ps_reg_01          5574.56284313
ps_ind_06_bin      7636.74549408
ps_ind_15          8308.71058588
ps_car_07_cat      8548.20830088
ps_ind_03          12378.42011375
ps_ind_17_bin      13121.31399965
ps_reg_03          14092.82143469
ps_ind_05_cat      16466.44651079
ps_car_13          38895.35653286
Name: mean_importance, dtype: float64
```

Визуализируем график усредненных важностей на основе информационного выигрыша.

```
# выводим график усредненных важностей
features.plot.barh(figsize=(10, 20));
```



Давайте создадим список из индекса серии. Признаки в этом списке отсортированы по возрастанию усредненной важности на основе информационного выигрыша.

```
# создаем список признаков
features = list(features.index)
features

['ps_calc_16_bin', 'ps_car_10_cat', 'ps_ind_13_bin', 'ps_ind_12_bin', 'ps_ind_10_bin',
 'ps_ind_11_bin', 'ps_ind_14', 'ps_calc_18_bin', 'ps_calc_20_bin', 'ps_calc_19_bin',
 'ps_calc_15_bin', 'ps_car_08_cat', 'ps_calc_17_bin', 'ps_ind_18_bin', 'ps_calc_06',
 'ps_calc_07', 'ps_calc_08', 'ps_calc_04', 'ps_calc_13', 'ps_calc_12', 'ps_calc_11',
 'ps_car_05_cat', 'ps_calc_05', 'ps_calc_09', 'ps_ind_08_bin', 'ps_calc_02', 'ps_car_12',
 'ps_calc_01', 'ps_calc_03', 'ps_car_02_cat', 'ps_car_06_cat', 'ps_calc_10',
 'ps_calc_14', 'ps_car_04_cat', 'ps_ind_04_cat', 'ps_car_09_cat', 'ps_car_11',
 'ps_car_11_cat', 'ps_ind_07_bin', 'ps_ind_02_cat', 'ps_car_15', 'ps_ind_09_bin',
 'ps_ind_16_bin', 'ps_car_14', 'ps_car_01_cat', 'ps_ind_01', 'ps_reg_02',
 'ps_car_03_cat', 'ps_reg_01', 'ps_ind_06_bin', 'ps_ind_15', 'ps_car_07_cat',
 'ps_ind_03', 'ps_ind_17_bin', 'ps_reg_03', 'ps_ind_05_cat', 'ps_car_13']
```

Вычисляем оценку AUC-ROC, усредненную по пяти проверочным блокам перекрестной проверки (пяти моделям со всеми признаками). Это будет у нас оценка AUC-ROC для модели со всеми признаками.

```
# вычисляем оценку AUC-ROC, усредненную по пяти проверочным
# блокам перекрестной проверки (пяти моделям со всеми
# признаками)
auc_score_all = output['test_score'].mean()
auc_score_all
```

0.6337963788722549

Теперь попробуем гибридный подход к отбору признаков. В нем мы сочтем встроенный метод (вычисление важностей на основе информационного выигрыша) с методом-оберткой (рекурсивным жадным удалением признаков). Мы уже сформировали список признаков, отсортированных по мере возрастания усредненной важности, и сейчас пойдем по нему, строя модель, каждый раз исключая признак из списка. Предварительно мы зададим пороговое значение разницы между AUC-ROC модели со всеми признаками (только что вычислили) и AUC-ROC модели с удаленным признаком. Если значение разницы является отрицательным, значит, удаление признака повысило AUC-ROC. Например, оценка AUC-ROC модели со всеми признаками равна 0,80, а оценка AUC-ROC модели с конкретным удаленным признаком равна 0,81, разница составляет $0,80 - 0,81 = -0,01$. Если значение разницы меньше порогового значения, то удаляем признак. Например, мы задали пороговое значение 0,01. Разница AUC-ROC составляет $-0,01$, что меньше порогового значения, тогда удаляем признак. Если значение разницы больше или равно пороговому значению, то сохраняем признак. Например, оценка AUC-ROC модели со всеми признаками равна 0,80, а оценка AUC-ROC модели с конкретным удаленным признаком равна 0,76, разница составляет $0,80 - 0,76 = 0,04$, что больше порогового значения, сохраняем признак. Нас, конечно, будут интересовать при-

знаки, дающие отрицательную разницу, такие переменные будут первыми кандидатами на удаление.

Помним, что для надежности лучше использовать оценку не одной модели, а нескольких моделей в рамках процедуры перекрестной проверки. Кроме того, помните, что пороговое значение разницы между AUC-ROC модели со всеми признаками и AUC-ROC модели с удаленным признаком – это гиперпараметр, который настраивается либо на проверочной выборке, либо на проверочных блоках перекрестной проверки, итоговую оценку качества модели, построенную на наборе с отобранными признаками (а пространство этих отобранных признаков будет зависеть от порогового значения разницы) нужно получать на независимой тестовой выборке. Здесь мы перебирали разные пороговые значения (для экономии этап пропущен), если используется одно пороговое значение, исходя из каких-то априорных знаний, можно обойтись без независимой тестовой выборки.

```
# задаем пороговое значение разницы AUC
tol = 0.0001
print("выполнение рекурсивного удаления признаков")
# создаем список, в который будем
# записывать удаляемые признаки
features_to_remove = []
# создаем список, в который будем
# записывать значение AUC
auc_score_mean_list = []
# создаем список, в который будем
# записывать разницу AUC
diff_auc_list = []

# задаем счетчик для оценки прогресса
count = 1

# итерируем по всем признакам, признаки упорядочены по
# возрастанию важности на основе информационного выигрыша
for feature in features:
    print()
    print("проверяемый признак: ", feature, " признак ", count,
          " из ", len(features))
    count = count + 1

    # создаем экземпляр класса LGBMClassifier
    model = LGBMClassifier(
        random_state=42, learning_rate=0.01,
        n_estimators=300, bagging_fraction=0.5,
        feature_fraction=1, lambda_l1=10)

    # обучаем модели со всеми признаками минус уже удаленные признаки
    # (берем их из списка удаляемых признаков) и оцениваемый признак
    auc_scores = cross_val_score(
        model,
        X_train.drop(features_to_remove + [feature], axis=1),
        y_train,
        scoring='roc_auc',
        cv=5)
```

```
# вычисляем, усредненный по проверочным блокам
# перекрестной проверки
auc_score_mean = auc_scores.mean()

# печатаем усредненное значение AUC
print("AUC модели после удаления={}".format(auc_score_mean))

# добавляем усредненное значение AUC в список
auc_score_mean_list.append(auc_score_mean)

# печатаем AUC модели со всеми признаками
# (опорное значение AUC)
print("AUC модели со всеми признаками={}".format(auc_score_all))

# определяем разницу AUC (если отрицательное значение
# - удаление признака улучшило AUC)
diff_auc = auc_score_all - auc_score_mean

# записываем разницу AUC в список
diff_auc_list.append(diff_auc)

# сравниваем разницу AUC с порогом, заданным заранее
# если разница AUC больше или равна порогу, сохраняем
if diff_auc >= tol:
    print("Разница AUC={}".format(diff_auc))
    print("сохраняем: ", feature)
    print

# если разница AUC меньше порога, удаляем
else:
    print("Разница AUC={}".format(diff_auc))
    print("удаляем: ", feature)
    print
    # если разница AUC меньше порога и мы удаляем признак,
    # мы в качестве нового опорного значения AUC задаем
    # значение AUC для модели с оставшимися признаками
    auc_score_all = auc_score_mean

    # добавляем удаляемый признак в список
    features_to_remove.append(feature)

# формируем датафрейм
df = pd.DataFrame({'feature': features,
                   'auc_score_mean': auc_score_mean_list,
                   'diff_auc_score': diff_auc_list})
# цикл завершен, вычисляем количество
# удаленных признаков
print("ВЫПОЛНЕНО!!")
print("общее количество признаков для удаления: ",
      len(features_to_remove))
# определяем признаки, которые мы хотим сохранить (не удаляем)
features_to_keep = [x for x in features
                     if x not in features_to_remove]
print("общее количество признаков для сохранения: ",
      len(features_to_keep))
```

выполнение последовательного удаления признаков

проверяемый признак: ps_calc_16_bin признак 1 из 57
AUC модели после удаления=0.6337963788722549
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=0.0
удаляем: ps_calc_16_bin

проверяемый признак: ps_cagr_10_cat признак 2 из 57
AUC модели после удаления=0.6337963788722549
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=0.0
удаляем: ps_cagr_10_cat

проверяемый признак: ps_ind_13_bin признак 3 из 57
AUC модели после удаления=0.6337963788722549
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=0.0
удаляем: ps_ind_13_bin

проверяемый признак: ps_ind_12_bin признак 4 из 57
AUC модели после удаления=0.6337963788722549
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=0.0
удаляем: ps_ind_12_bin

проверяемый признак: ps_ind_10_bin признак 5 из 57
AUC модели после удаления=0.6337963788722549
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=0.0
удаляем: ps_ind_10_bin

проверяемый признак: ps_ind_11_bin признак 6 из 57
AUC модели после удаления=0.6337963788722549
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=0.0
удаляем: ps_ind_11_bin

проверяемый признак: ps_ind_14 признак 7 из 57
AUC модели после удаления=0.6338067550591004
AUC модели со всеми признаками=0.6337963788722549
Разница AUC=-1.0376186845517665e-05
удаляем: ps_ind_14

проверяемый признак: ps_calc_18_bin признак 8 из 57
AUC модели после удаления=0.6338111094806743
AUC модели со всеми признаками=0.6338067550591004
Разница AUC=-4.3544215738711145e-06
удаляем: ps_calc_18_bin

проверяемый признак: ps_calc_20_bin признак 9 из 57
AUC модели после удаления=0.633827714658789
AUC модели со всеми признаками=0.6338111094806743
Разница AUC=-1.660517811463702e-05
удаляем: ps_calc_20_bin

проверяемый признак: ps_calc_19_bin признак 10 из 57
AUC модели после удаления=0.6338229642834915
AUC модели со всеми признаками=0.633827714658789
Разница AUC=4.7503752974575875e-06
удаляем: ps_calc_19_bin

проверяемый признак: ps_calc_15_bin признак 11 из 57
AUC модели после удаления=0.6338110813518851
AUC модели со всеми признаками=0.6338229642834915
Разница AUC=1.188293160636622e-05
удаляем: ps_calc_15_bin

проверяемый признак: ps_cag_08_cat признак 12 из 57
AUC модели после удаления=0.6337359989043597
AUC модели со всеми признаками=0.6338110813518851
Разница AUC=7.508244752540971e-05
удаляем: ps_cag_08_cat

проверяемый признак: ps_calc_17_bin признак 13 из 57
AUC модели после удаления=0.6337827940871303
AUC модели со всеми признаками=0.6337359989043597
Разница AUC=-4.6795182770620336e-05
удаляем: ps_calc_17_bin

• • • • •

проверяемый признак: ps_ind_17_bin признак 54 из 57
AUC модели после удаления=0.6321386404840369
AUC модели со всеми признаками=0.6346921252310002
Разница AUC=0.0025534847469632638
сохраняем: ps_ind_17_bin

проверяемый признак: ps_reg_03 признак 55 из 57
AUC модели после удаления=0.6341076985623825
AUC модели со всеми признаками=0.6346921252310002
Разница AUC=0.000584426668617688
сохраняем: ps_reg_03

проверяемый признак: ps_ind_05_cat признак 56 из 57
AUC модели после удаления=0.6297811483108824
AUC модели со всеми признаками=0.6346921252310002
Разница AUC=0.004910976920117771
сохраняем: ps_ind_05_cat

проверяемый признак: ps_cag_13 признак 57 из 57
AUC модели после удаления=0.6319399615646538
AUC модели со всеми признаками=0.6346921252310002
Разница AUC=0.002752163666346341
сохраняем: ps_cag_13
ВЫПЛНЕНО!!
общее количество признаков для удаления: 35
общее количество признаков для сохранения: 22

Большая часть признаков, в названии которых упоминается calc, имеют отрицательные важности и рекомендуются к удалению. Обязательно нужно попробовать построить модель без этих признаков.

Выведем получившийся датафрейм.

```
# выводим получившийся датафрейм
df.sort_values(by='diff_auc_score', ascending=False)
```

	feature	auc_score_mean	diff_auc_score
55	ps_ind_05_cat	0.62978115	0.00491098
52	ps_ind_03	0.63029787	0.00439426
56	ps_car_13	0.63193996	0.00275216
53	ps_ind_17_bin	0.63213864	0.00255348
50	ps_ind_15	0.63253407	0.00215805
51	ps_car_07_cat	0.63291583	0.00177629
48	ps_reg_01	0.63336871	0.00125154
45	ps_ind_01	0.63378883	0.00083142
44	ps_car_01_cat	0.63392532	0.00069493
47	ps_car_03_cat	0.63395877	0.00066147
39	ps_ind_02_cat	0.63401722	0.00062519
34	ps_ind_04_cat	0.63380038	0.00060725
54	ps_reg_03	0.63410770	0.00058443
46	ps_reg_02	0.63409663	0.00052361
36	ps_car_11	0.63396734	0.00044029
41	ps_ind_09_bin	0.63430118	0.00034123
35	ps_car_09_cat	0.63409074	0.00031689
29	ps_car_02_cat	0.63405267	0.00026078
40	ps_car_15	0.63442188	0.00022052
24	ps_ind_08_bin	0.63378164	0.00021228
38	ps_ind_07_bin	0.63451221	0.00013019
30	ps_car_06_cat	0.63419859	0.00011486
42	ps_ind_16_bin	0.63454606	0.00009635
22	ps_calc_05	0.63399118	0.00008280
11	ps_car_08_cat	0.63373600	0.00007508
32	ps_calc_14	0.63442409	0.00005813
21	ps_car_05_cat	0.63407398	0.00002005
33	ps_car_04_cat	0.63440763	0.00001646
10	ps_calc_15_bin	0.63381108	0.00001188
9	ps_calc_19_bin	0.63382296	0.00000475
13	ps_ind_18_bin	0.63378256	0.00000024
0	ps_calc_16_bin	0.63379638	0.00000000
5	ps_ind_11_bin	0.63379638	0.00000000
1	ps_car_10_cat	0.63379638	0.00000000
4	ps_ind_10_bin	0.63379638	0.00000000
2	ps_ind_13_bin	0.63379638	0.00000000
3	ps_ind_12_bin	0.63379638	0.00000000
19	ps_calc_12	0.63397520	-0.00000220
23	ps_calc_09	0.63399392	-0.00000273
7	ps_calc_18_bin	0.63381111	-0.00000435
16	ps_calc_08	0.63392509	-0.00000566
6	ps_ind_14	0.63380676	-0.00001038

8	ps_calc_20_bin	0.63382771	-0.00001661
17	ps_calc_04	0.63394824	-0.00002314
18	ps_calc_13	0.63397300	-0.00002476
14	ps_calc_06	0.63381810	-0.00003554
12	ps_calc_17_bin	0.63378279	-0.00004680
25	ps_calc_02	0.63404081	-0.00004689
27	ps_calc_01	0.63419725	-0.00007062
49	ps_ind_06_bin	0.63469213	-0.00007188
43	ps_car_14	0.63462025	-0.00007419
26	ps_car_12	0.63412663	-0.00008582
15	ps_calc_07	0.63391944	-0.00010134
28	ps_calc_03	0.63431345	-0.00011620
20	ps_calc_11	0.63409404	-0.00011884
31	ps_calc_10	0.63448222	-0.00016877
37	ps_car_11_cat	0.63464241	-0.00023478

Можно попробовать подход, в ходе которого строим модели LightGBM, увеличивая глубину, и смотрим важности признаков. Здесь нам важно понять, как быстро признаки «включаются в работу», т. е. начинают использоваться в качестве признаков расщепления. Чем раньше, т. е. чем меньше глубина использования признака, тем важнее признак.

```
# еще один подход - смотрим, как меняются важности
# признаков по мере увеличения глубины: наиболее
# важные признаки - те, которые начинают
# использоваться раньше остальных

# задаем сетку значений глубины
max_depth_grid = [1, 2, 3, 4, 5]

# создаем список fi, в который будем сохранять
# важности признаков, и сохраняем в него важности,
# рассчитанные для каждой из моделей
fi = []

# обучаем модели с разными значениями глубины, получаем
# важности и записываем важности в список
for max_depth in max_depth_grid:
    model_all_features = LGBMClassifier(
        random_state=42,
        learning_rate=0.01,
        n_estimators=300,
        bagging_fraction=0.5,
        feature_fraction=1,
        lambda_l1=10,
        max_depth=max_depth,
        importance_type='gain')
    model_all_features.fit(X_train, y_train)
    fi.append(model_all_features.feature_importances_)
```

```

# преобразовываем список в датафрейм, индексы в котором
# будут именами наших переменных
fi = pd.DataFrame(
    np.array(fi).T,
    columns=['importance ' + str(idx)
              for idx in range(len(fi))],
    index=X_train.columns)

# вычисляем усредненные важности и добавляем столбец с ними
fi['mean_importance'] = fi.mean(axis=1)
# сортируем по убыванию усредненных важностей
fi = fi.sort_values(by='mean_importance', ascending=False)
# смотрим полученный датафрейм
fi

```

	importance 0	importance 1	importance 2	importance 3	importance 4	mean_importance
ps_car_13	31088.76501465	37646.84114456	41829.26031256	43967.54304695	46634.65979491	40233.41386273
ps_ind_05_cat	11652.49598694	17385.41012573	19248.93219471	19829.36675119	20479.29731715	17719.10047514
ps_ind_17_bin	13619.29296875	16078.80482960	15924.80874634	16155.20823884	16395.42095292	15634.70714729
ps_reg_03	10162.90495300	12752.14356613	14365.13713789	14954.14613691	15789.47207206	13604.76077320
ps_car_07_cat	7655.76197815	9674.39733505	10612.05427265	10625.81039953	10304.01973176	9774.40874343
ps_ind_06_bin	6724.82598877	8012.71405029	8784.96511269	8746.07821178	10020.12641358	8457.74195542
ps_ind_03	831.29901123	5254.63282585	9482.69581079	11901.62035066	13185.44850903	8131.13930151
ps_car_03_cat	2305.92500305	5675.29155350	6248.71047497	6779.32622671	6998.71988058	5601.59462776
ps_ind_15	0.00000000	3759.04252434	5811.22193480	7761.13614535	8936.56874725	5253.59387035
ps_reg_02	2386.90000916	4291.70919228	4320.85382652	5384.68808004	6524.61982181	4581.75418596
ps_reg_01	0.00000000	1973.42809677	3397.28571320	4956.82016042	6432.46748034	3352.00029087
ps_ind_07_bin	1957.69799805	3247.97988892	3260.49017334	3454.27051258	2097.31960863	2803.55163630
ps_ind_16_bin	0.00000000	1911.37630463	3157.28998375	3319.97046995	3710.17561281	2419.76247423
ps_car_01_cat	0.00000000	933.99000168	2103.05275506	2887.38082376	3980.37131802	1980.95897970
ps_ind_09_bin	0.00000000	339.66689301	1429.61061049	1844.03826141	3141.73607612	1351.01036820
ps_ind_01	0.00000000	0.00000000	896.75059605	1805.29361260	2907.39888167	1121.88861806
ps_car_04_cat	726.89100647	1549.85709381	1086.00623417	1062.52384758	1142.33369184	1113.52237477
ps_car_14	0.00000000	89.67190170	929.72722626	1707.75405788	2494.83665672	1044.39796851
ps_car_15	0.00000000	149.88369942	451.31661892	1682.20131731	2652.91240441	987.26280801
ps_ind_08_bin	0.00000000	290.63089752	694.75669861	1580.55943489	386.26834249	590.44307470
ps_ind_02_cat	0.00000000	14.55609989	426.72060204	908.44012260	1458.22494543	561.58835399
ps_car_09_cat	0.00000000	0.00000000	166.57168865	906.97482753	1392.64119148	493.23754153
ps_car_11	0.00000000	0.00000000	47.54831028	708.67777872	1221.50005210	395.54522822
ps_car_11_cat	0.00000000	0.00000000	122.83360004	278.96876937	1054.12522064	291.18551801
ps_car_02_cat	0.00000000	0.00000000	22.98900032	366.01611853	820.22939920	241.84690361
ps_calc_05	0.00000000	0.00000000	106.96292019	307.05697870	509.30712819	184.66540542
ps_calc_01	0.00000000	0.00000000	10.56719971	257.81927204	572.63541198	168.20437675
ps_calc_14	0.00000000	0.00000000	0.00000000	128.54154849	686.07907273	162.92412424

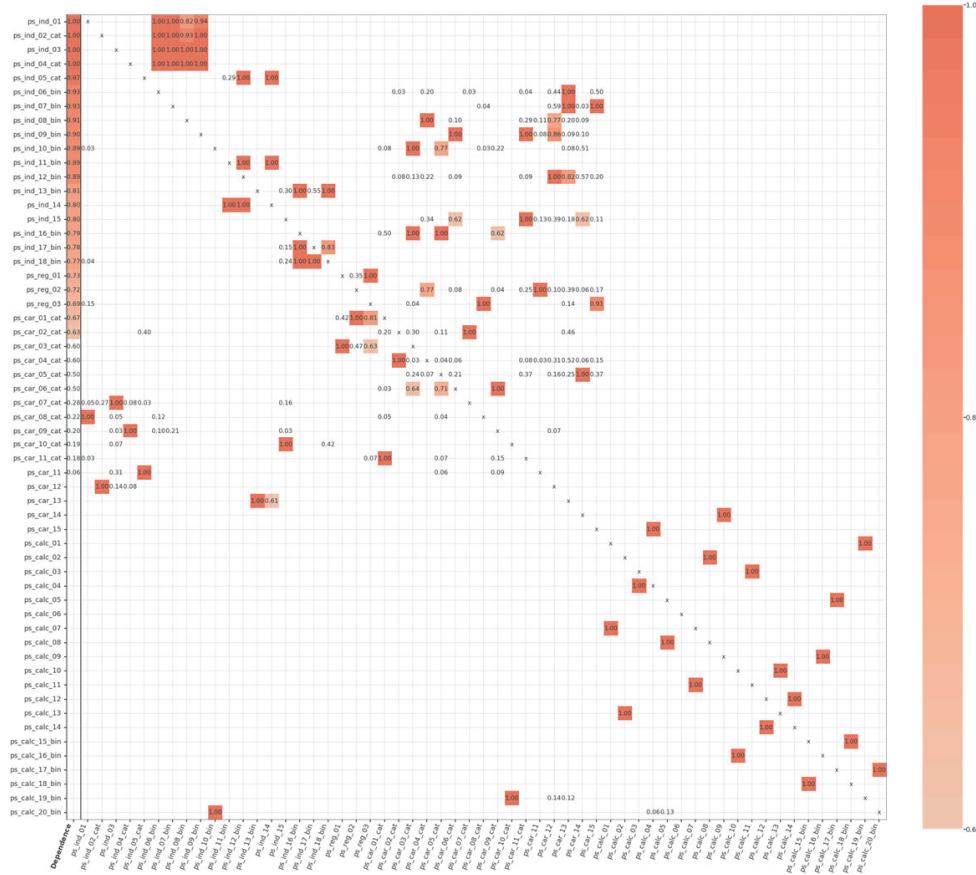
	importance 0	importance 1	importance 2	importance 3	importance 4	mean_importance
ps_calc_10	0.0000000	0.0000000	7.91657996	151.46667659	631.88851071	158.25435345
ps_car_06_cat	0.0000000	0.0000000	31.43953991	178.86615109	533.49897681	148.76093356
ps_calc_02	0.0000000	0.0000000	6.70694017	116.81970930	510.42256689	126.78984327
ps_car_12	0.0000000	0.0000000	0.0000000	84.68779945	528.57922009	122.65340391
ps_car_05_cat	0.0000000	0.0000000	22.58640957	244.49698973	315.68886486	116.55445283
ps_ind_04_cat	0.0000000	0.0000000	0.0000000	61.23967075	445.47994184	101.34392252
ps_calc_09	0.0000000	0.0000000	0.0000000	127.51931930	355.34966280	96.57379642
ps_calc_12	0.0000000	0.0000000	0.0000000	7.92763007	445.77103599	90.73973321
ps_calc_03	0.0000000	0.0000000	9.99913979	68.79207969	352.50114140	86.25847217
ps_calc_04	0.0000000	0.0000000	18.97594023	72.99546003	187.44858408	55.88399687
ps_calc_06	0.0000000	0.0000000	0.0000000	37.44561982	195.58822438	46.60676884
ps_ind_18_bin	0.0000000	0.0000000	0.0000000	66.04910088	108.33237028	34.87629423
ps_calc_13	0.0000000	0.0000000	0.0000000	14.52629995	136.44696951	30.19465389
ps_calc_11	0.0000000	0.0000000	0.0000000	24.48115945	106.74532100	26.24529609
ps_calc_08	0.0000000	0.0000000	0.0000000	23.78528953	87.95740661	22.34853923
ps_calc_15_bin	0.0000000	0.0000000	16.40506077	7.52575016	75.66593051	19.91934829
ps_calc_17_bin	0.0000000	0.0000000	0.0000000	0.0000000	58.55152035	11.71030407
ps_calc_07	0.0000000	0.0000000	0.0000000	0.0000000	44.54987234	8.90997447
ps_ind_14	0.0000000	0.0000000	0.0000000	0.0000000	24.49913979	4.89982796
ps_calc_18_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_calc_16_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_calc_19_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_car_08_cat	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_car_10_cat	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_ind_13_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_ind_12_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_ind_11_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_ind_10_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
ps_calc_20_bin	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000

Опять видим, что признаки с префиксом calc, а также признаки *ps_car_08_cat*, *ps_car_10_cat*, *ps_ind_10_bin*, *ps_ind_11_bin*, *ps_ind_12_bin*, *ps_ind_13_bin*, *ps_ind_14* начинают использоваться деревьями позже остальных (нулевые значения). Потенциально это также может говорить о низкой важности этих признаков. Примечательно, что почти все эти признаки, кроме *ps_car_08_cat* и некоторых признаков с префиксом calc, давали отрицательные разности AUC-ROC.

До этого момента мы смотрели важности, оценивающие релевантность признака, его связь.

Наконец, попробуем еще один подход, в ходе которого мы вычисляем матрицу зависимостей признаков, значениями в этой матрице будут пермутированные важности признаков, с помощью которых мы пытаемся предсказать интересующий признак. Если признак хорошо предсказывается остальными, он будет менее важен, в то же время если признак не предсказывается вообще остальными признаками, высока вероятность того, что он либо создан искусственно, либо не релевантен задаче (прогнозируем вероятность просрочки, а в качестве признаков используем переменную *Наличие домашних животных*) и, скорее всего, будет снижать качество модели. Нам потребуются функции `feature_dependence_matrix()`, `plot_dependence_heatmap()` и `plot_coggr_heatmap()` из пакета `rfimp`.

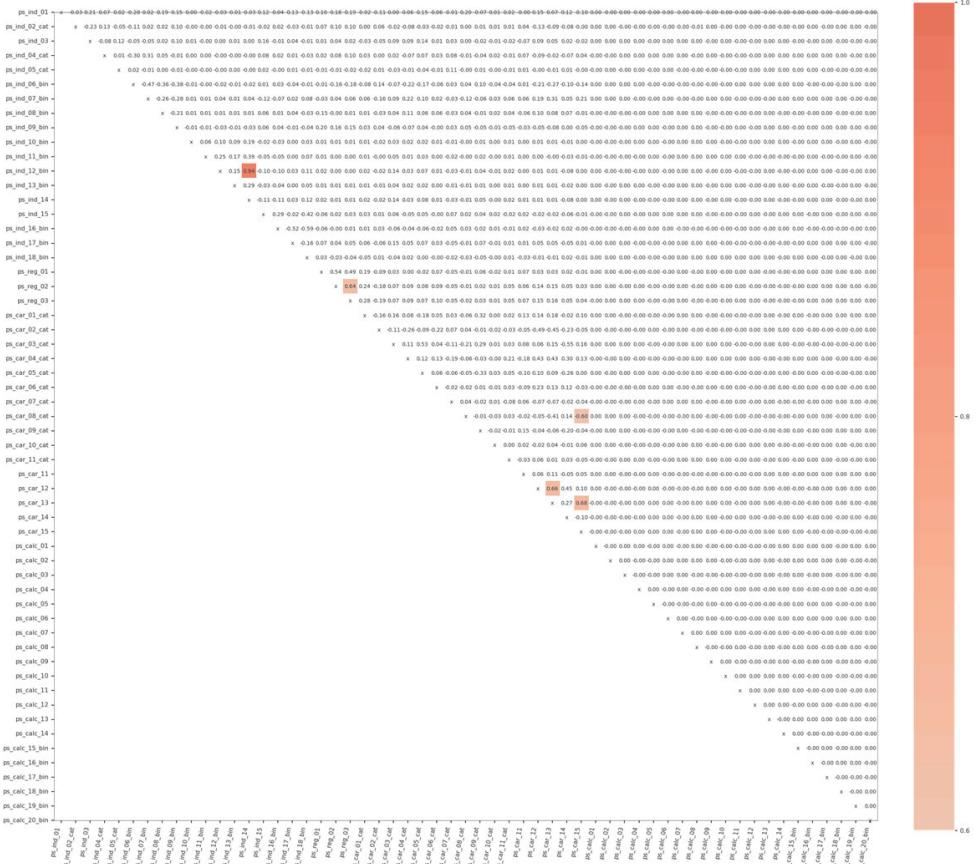
```
# вычисляем матрицу зависимостей признаков, значения - это
# пермутированные важности признаков, с помощью которых
# мы пытаемся предсказать интересующий признак
D = feature_dependence_matrix(X_train, sort_by_dependence=True)
viz = plot_dependence_heatmap(D, figsize=(18, 18))
viz
```



Видим, что в предсказании признаков совершенно не участвуют признаки с префиксом `calc`, а сами признаки с префиксом `calc` совершенно не предсказываются остальными признаками.

Давайте взглянем на матрицу корреляций на основе ранговой корреляции Спирмена, которая не предполагает линейной взаимосвязи между переменными.

```
# выводим матрицу корреляций (на основе
# ранговой корреляции Спирмена)
viz = plot_corr_heatmap(X_train,
                        figsize=(18, 18),
                        label_fontsize=8,
                        value_fontsize=7)
viz
```



Видим, что признаки с префиксом `calc` имеют нулевые корреляции с остальными признаками, не имеющими префикса `calc`.

Принимаем решение, что не будем использовать признаки с упоминанием `calc` и признаки `ps_ind_14`, `ps_car_10_cat`, `ps_ind_10_bin`, `ps_ind_11_bin`, `ps_ind_12_bin`, `ps_ind_13_bin`, `ps_ind_18_bin`.

Начинаем готовить финальное решение на базе трех моделей – LightGBM, CatBoost и XGBoost.

Начинаем с того, что пишем функцию предварительной подготовки для исторического набора и набора новых данных.

```
# пишем функцию предварительной подготовки
def preprocessing(df, lightgbm=True, newdata=False):
    # удаляем столбцы с calc в названии
    calc_columns = df.columns[df.columns.str.contains('calc')]
    df.drop(calc_columns, axis=1, inplace=True)

    # для новых данных
    if newdata:
        # записываем id набора
```

```

ident = df['id']
# удаляем id из набора
df.drop('id', axis=1, inplace=True)

# для исторических данных
else:
    # удаляем id из набора
    df.drop('id', axis=1, inplace=True)
    # формируем массив меток и массив признаков
    labels = df.pop('target')

# если готовим данные для LightGBM
if lightgbm:
    # удаляем наименее важные переменные
    some_columns = ['ps_ind_14', 'ps_car_10_cat',
                    'ps_ind_10_bin', 'ps_ind_11_bin',
                    'ps_ind_12_bin', 'ps_ind_13_bin',
                    'ps_ind_18_bin']
    df.drop(some_columns, axis=1, inplace=True)

    # записываем список столбцов с cat в названии
    cat_columns = df.columns[df.columns.str.contains('cat')]
    # формируем массив с num
    df_ = df[cat_columns]
    # столбцам с cat в названии присваиваем тип object
    # и выполняем дамми-кодирование
    for col in cat_columns:
        df[col] = df[col].astype('object')

    df = pd.get_dummies(df)
    # конкatenируем массив с исходными столбцами с cat в названии
    # и массив, к которому было применено дамми-кодирование
    df = pd.concat([df_, df], axis=1)

# в противном случае (если готовим данные
# для CatBoost и XGBoost)
else:
    # удаляем наименее важные переменные
    some_columns = ['ps_ind_14', 'ps_car_10_cat',
                    'ps_car_14', 'ps_ind_10_bin',
                    'ps_ind_11_bin', 'ps_ind_12_bin',
                    'ps_ind_13_bin', 'ps_car_11',
                    'ps_car_12']
    df.drop(some_columns, axis=1, inplace=True)

# для новых данных
if newdata:
    # возвращаем преобразованный массив
    # признаков, идентификатор
    return df, ident

# для исторических данных
else:
    # возвращаем преобразованный массив признаков,
    # массив меток
    return df, labels

```

Кратко опишем, что происходит под капотом функции предварительной подготовки.

Мы начинаем с того, что удаляем переменные с префиксом `calc`.

Если мы работаем с набором новых данных (здесь под новыми данными подразумеваются данные без зависимой переменной), то записываем `id` в переменную `ident` и удаляем `id` из набора. Если же мы работаем с историческими данными, то удаляем `id` из набора и с помощью метода `.pop()` формируем массив меток и массив признаков.

Если мы делаем предварительную подготовку данных для LightGBM, то удаляем переменные `ps_ind_14`, `ps_car_10_cat`, `ps_ind_10_bin`, `ps_ind_11_bin`, `ps_ind_12_bin`, `ps_ind_13_bin`, `ps_ind_18_bin`. Затем записываем список столбцов с `cat` в названии и формируем массив с ним. Столбцам с `cat` в названии присваиваем тип `object` и выполняем дамми-кодирование. Затем конкатенируем массив с исходными столбцами с `cat` в названии и массив, к которому было применено дамми-кодирование.

Если мы делаем предварительную подготовку данных для CatBoost и XGBoost, то удаляем переменные `ps_ind_14`, `ps_car_10_cat`, `ps_car_14`, `ps_ind_10_bin`, `ps_ind_11_bin`, `ps_ind_12_bin`, `ps_ind_13_bin`, `ps_car_11`, `ps_car_12`. Кроме того, вычисление важностей по SHAP с постепенным увеличением глубины для CatBoost и вычисление важностей по информационному выигрышу с постепенным увеличением глубины для XGBoost (для экономии места этап пропущен) показало, что можно удалить `ps_car_11` и `ps_car_12`.

Если мы работаем с новыми данными, возвращаем преобразованный массив признаков и идентификатор. Если же мы работаем с историческими данными, возвращаем преобразованный массив признаков и массив меток (массив меток, разумеется, оставляем без изменений).

Загружаем исторический набор и набор новых данных и выполняем предварительную подготовку данных для LightGBM.

```
# загружаем наборы
train = pd.read_csv('Data/porto_seguro_train.csv')
test = pd.read_csv('Data/porto_seguro_test.csv')

# выполняем предварительную подготовку
# данных для LightGBM
train, labels = preprocessing(train,
                             lightgbm=True,
                             newdata=False)
test, ident = preprocessing(test,
                           lightgbm=True,
                           newdata=True)
```

Теперь строим модель LightGBM на историческом наборе и вычисляем вероятности для набора новых данных. Гиперпараметры предварительно настраивались.

```
# создаем экземпляр класса LGBMClassifier
lightgbm_model = LGBMClassifier(random_state=42,
                                 feature_fraction=0.4,
                                 lambda_l1=8,
                                 bagging_fraction=0.1,
                                 learning_rate=0.012,
                                 n_estimators=1600)
```

```
# строим модель на всем историческом наборе
lightgbm_model.fit(train, labels)

# вычисляем вероятности для набора новых данных
lgbm_preds_prob = lightgbm_model.predict_proba(test)[:, 1]
```

Загружаем исторический набор и набор новых данных, выполняем предварительную подготовку данных для CatBoost и XGBoost.

```
# загружаем наборы
train = pd.read_csv('Data/porto_seguro_train.csv')
test = pd.read_csv('Data/porto_seguro_test.csv')

# выполняем предварительную подготовку
# данных для CatBoost и XGBoost
train, labels = preprocessing(train,
                             lightgbm=False,
                             newdata=False)
test, ident = preprocessing(test,
                           lightgbm=False,
                           newdata=True)
```

Формируем массив индексов категориальных признаков и обучающий пул.

```
# формируем массив индексов категориальных признаков
cat_features_ids = np.where(train.dtypes != float)[0]
# формируем обучающий пул
train_pool = Pool(train,
                  labels,
                  cat_features=cat_features_ids)
```

Теперь строим модель CatBoost на историческом наборе и вычисляем вероятности для набора новых данных. Гиперпараметры предварительно настраивались.

```
# создаем экземпляр класса CatBoostClassifier
catbst_model = CatBoostClassifier(
    iterations=1200,
    learning_rate=0.1,
    random_strength=0.15,
    simple_ctr='Counter:CtrBorderCount=50',
    model_size_reg=0.1,
    max_depth=4,
    random_seed=0,
    logging_level='Silent')

# обучаем модель
catbst_model.fit(train_pool)

# вычисляем вероятности
catbst_preds_prob = catbst_model.predict_proba(test)[:, 1]
```

Теперь строим модель XGBoost на историческом наборе и вычисляем вероятности для набора новых данных. Гиперпараметры предварительно настраивались.

```
# создаем экземпляр класса XGBClassifier
xgbst_model = XGBClassifier(learning_rate=0.04,
                             subsample=0.7,
                             random_state=42,
                             max_depth=4,
                             n_estimators=500)

# обучаем модель
xgbst_model.fit(train, labels)
# вычисляем вероятности
xgbst_preds_prob = xgbst_model.predict_proba(test)[:, 1]
```

Усредняем вероятности трех моделей, используя вес, при этом модель, дающая меньшее качество, берется с меньшим весом.

```
# усредняем вероятности с весами,
# учитывающими качество модели
average_prob = (catbst_preds_prob * 0.5 +
                 xgbst_preds_prob * 0.3 +
                 lgbm_preds_prob * 1) / 3
```

Формируем посылку и отправляем решение. Наш результат соответствует 11-му месту на приватном лидерборде.

```
# формируем посылку
pd.DataFrame({'id': ident, 'target': average_prob}).to_csv(
    'subm_seguro.csv', index=False)
```

Private Score	Public Score
0.29180	0.28458

18. Стандартизация

Ряд методов машинного обучения, например линейная и логистическая регрессии, кластерный анализ, нейронные сети и SVM, чувствительны к масштабу признаков. Если не привести признаки к единому масштабу, то прогноз будут определять признаки, имеющие наибольшую разрядность и соответственно наибольшую дисперсию.

Допустим, у нас есть доход в долларах и возраст. Поскольку дисперсия у дохода в долларах (разряд – десятки тысяч) обычно намного больше, чем у возраста (разряд – десятки), доход будет доминировать в решении. Взгляните на рисунок. На вертикальной (доход) и горизонтальной (возраст) осях установлен одинаковый масштаб (единичный). По рисунку ясно, что при измерении дохода в долларах (а не в тысячах или десятках тысяч долларов) решения будут почти полностью определяться доходом.

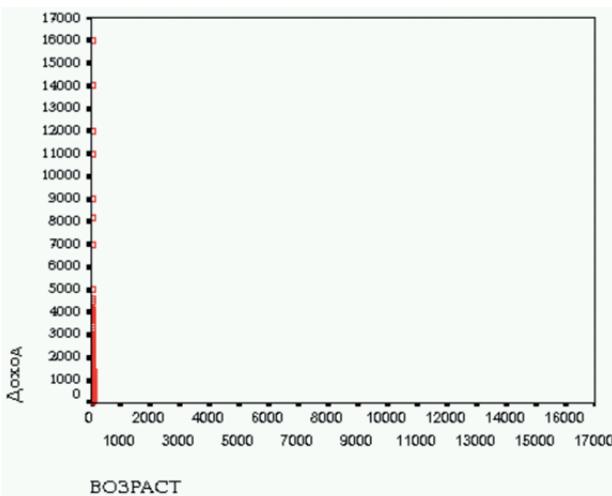


Рис. 67 Вычисления в условиях разных масштабов

Поэтому обычной практикой является преобразование признаков, с тем чтобы итоговое представление данных было более подходящим для использования вышеупомянутых алгоритмов. Часто достаточно процедуры приведения признаков к единому масштабу, которую называют масштабированием (scaling) или стандартизацией (standardization).

Для кластерного анализа он помогает сделать признаки равноправными в образовании кластеров. В методе опорных векторов единый масштаб дает признакам одинаковое влияние при вычислении расстояния между разделяющей гиперплоскостью и объектами разделяемых классов. Для регрессионного анализа единый масштаб позволяет сравнивать регрессионные коэффициенты между собой при признаках и корректно применять регуляризацию. Что значит корректно?

Штраф, который накладывается на значения коэффициентов при признаках в ходе регуляризации, будет зависеть от масштаба признаков. Представьте-

те, мы предсказываем вероятность мошенничества и у нас есть признак *Сумма транзакции*. Без регуляризации, если единица измерения признака *Сумма транзакции* определяется в долларах, то подогнанный коэффициент будет приблизительно в 100 раз больше, чем в случае, если бы единицей измерения был цент. Это обусловлено тем, что по сравнению с признаками большого масштаба признаки маленького масштаба должны иметь более высокие коэффициенты, чтобы оказывать одинаковое влияние на результат. Когда используется регуляризация, мы знаем, что лasso и гребневая регрессия существенное штрафуют большие коэффициенты, а это значит, что на признак *Сумма транзакции* будет наложен больший штраф, если единицей измерения будет доллар. Таким образом, регуляризация является предвзятой и имеет тенденцию штрафовать признаки меньшего масштаба. Чтобы решить проблему, мы как раз стандартизируем все признаки и обеспечиваем им равные условия в ходе регуляризации.

А вот деревьям решений стандартизация не нужна. Деревья вместо абсолютных значений работают с пороговыми значениями – разделяющими значениями признаков, по которым разбивают выборку наблюдений на два узла (если деревья являются бинарными, например деревья CART, QUEST) и более (если деревья могут иметь более двух потомков, например CHAID). Допустим, дерево принимает решение разбить выборку по признаку *Возраст* в значение 60, мы сравниваем все наблюдения с этим порогом, наблюдения ≤ 60 лет идут в левый узел, а наблюдения > 60 лет – в правый узел.

Стандартизация влияет на качество градиентного спуска. На рисунке, ставшем уже классическим, показано, как будет осуществлен градиентный спуск до стандартизации (слева) и после стандартизации (справа).

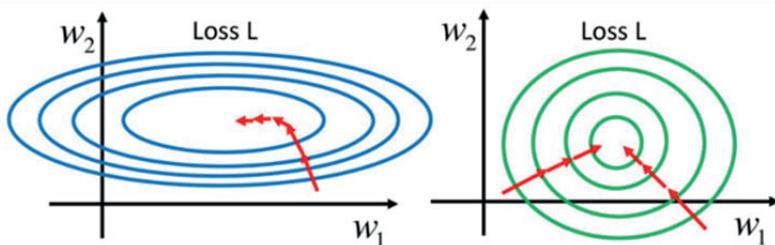


Рис. 68 Градиентный спуск (слева – стандартизация не выполнена, справа – стандартизация выполнена)

Градиентный спуск работает хорошо, если линии уровня функции похожи на круги (правая часть рисунка справа). В этом случае, откуда бы вы ни начали, вектор антиградиента будет смотреть в сторону минимума функции и будет сходиться довольно быстро. Если же линии уровня похожи на вытянутые эллипсы, то градиентный спуск будет иметь проблемы. Эллипсоподобные линии уровня обусловлены тем, что небольшим изменениям одного признака соответствуют очень большие изменения другого признака (разные масштабы признаков или признаки имеют один и тот же масштаб, но встречаются выбросы – наблюдения, значительно отличающиеся от остальных). Направление антиградиента будет слабо совпадать с направлением в сторону мини-

мума функции, и градиентный спуск будет делать много лишних шагов. Его сходимость будет медленная. И более того, есть риск расхождения градиентного спуска, если размер шага будет подобран неправильно. Именно различие в масштабе признаков и выбросы приводят к тому, что линии уровня не похожи на круги. Чтобы бороться с этой проблемой, нужно обрабатывать выбросы и затем стандартизовать признаки.

Самая простая стандартизация подразумевает, что из каждого значения переменной мы вычтем среднее значение и полученный результат разделим на стандартное отклонение:

$$\frac{x_i - \text{mean}(x)}{\text{stddev}(x)}.$$

Разумеется, в стандартизации нуждаются лишь количественные переменные.

Именно это и делает класс `StandardScaler`. В итоге мы получаем распределение со средним 0 и стандартным отклонением 1. Теперь мы можем сказать, насколько наше значение отличается от среднего в единицах стандартного отклонения.

Еще один способ масштабирования заключается в том, чтобы из каждого значения переменной вычесть минимальное значение и полученный результат разделить на ширину диапазона (разницу между минимальным и максимальным значениями):

$$\frac{x_i - \min(x)}{\max(x) - \min(x)}.$$

В итоге мы сжимаем значения переменных в диапазон от 0 до 1 (или от -1 до 1, если есть отрицательные значения). При таком способе масштабирования стандартные отклонения получают значения меньше 1. Этот способ реализован в классе `MinMaxScaler` и работает лучше в тех случаях, когда `StandardScaler` дает не очень хороший результат. Если распределение не является нормальным или стандартное отклонение является очень маленьким, `MinMaxScaler` сработает лучше. Обратите внимание на то, что данный вид стандартизации часто называется нормализацией, не путайте эту процедуру с приведением количественных переменных к нормальному распределению.

Класс `RobustScaler` похож на класс `MinMaxScaler`, но из каждого значения переменной вычитает значение, соответствующее первому квартилю, и полученный результат делит на межквартильный размах:

$$\frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}.$$

Он часто используется, когда данные содержат выбросы. Меньшая чувствительность данного вида стандартизации к выбросам дает больший диапазон преобразованных значений, чем у других видов стандартизации. Однако выбросы не удаляются, просто уменьшается их разряд. У нас были выбросы – миллионы, после стандартизации они стали тысячами, тогда как остальные числа стали единицами.

В стандартизации нуждаются лишь количественные переменные. Если речь идет о регрессионном анализе, категориальные переменные будут записаны в виде дамми-переменных со значениями 0 или 1 и дамми-переменные стандартизировать не нужно. В случае присутствия дамми-переменных рекомендуется при выполнении стандартизации количественных переменных делить не на одно, а на два стандартных отклонения, чтобы и дамми-переменные, и количественные переменные имели один и тот же масштаб (и мы могли сравнивать коэффициенты при них), стандартные отклонения дамми-переменных и количественных переменных будут примерно равны 0,5, в противном случае стандартные отклонения дамми-переменных будут равны 0,5, а стандартные отклонения количественных переменных – 1.

Обратите внимание: стандартизация не меняет форму распределения и не заменяет собой нормализацию распределения переменных, если переменная характеризовалась правосторонней асимметрией, эта асимметрия так остается и после стандартизации. Также следует отметить, что ни один из способов стандартизации не в состоянии удалить выбросы и стандартизация всегда проводится после обработки выбросов.

Теперь подробнее выясним, почему стандартизация не меняет форму распределения. Потому что, вычитая из значения среднее, мы просто изменяем его положение, сдвигаем среднее к нулю, а деля на стандартное отклонение, мы меняем масштаб. Сначала на конкретном примере посмотрим, как вычитание константы из каждого значения переменной влияет на распределение.

Допустим, у нас есть информация о весе 80 мужчин в возрасте от 19 до 24 лет со средним ростом. Мы визуализировали ее в виде гистограммы и ящиковидной диаграммы.

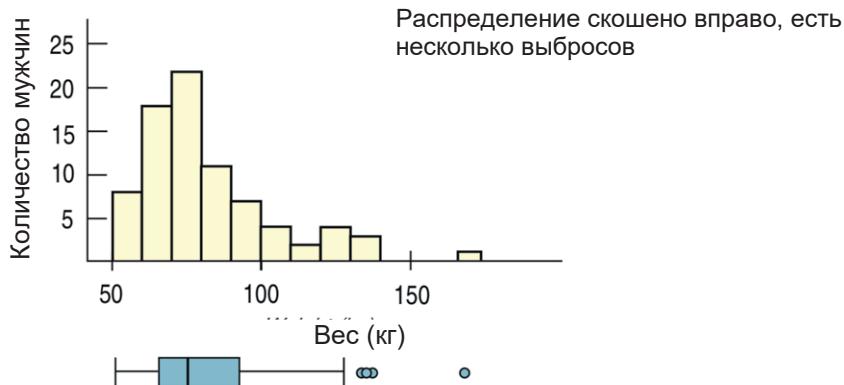


Рис. 69 Исходное распределение веса мужчин

Средний вес мужчин составляет 82,36 кг. Для данной категории мужчин Национальный институт здоровья США рекомендует максимальный вес 74 кг, но мы можем видеть, что некоторые мужчины тяжелее рекомендованного веса. Чтобы сравнить их вес с рекомендованным максимумом, мы могли бы вычесть 74 кг из веса каждого мужчины. Как изменился центр, форма и разброс распределения? Ниже видим соответствующий рисунок.

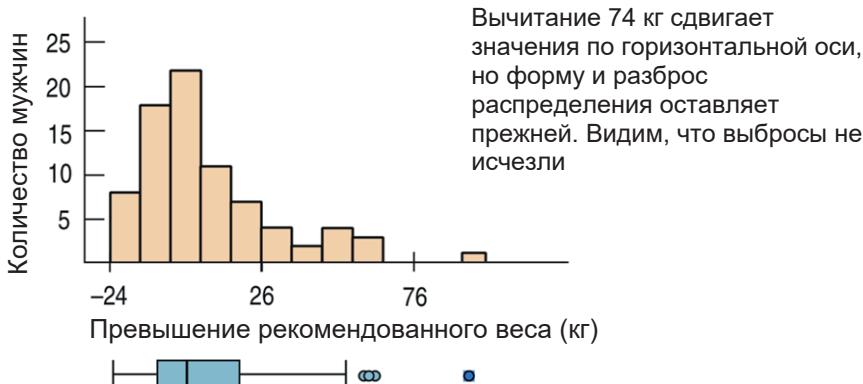


Рис. 70 Распределение веса мужчин после вычитания рекомендованного максимального веса

Средний вес мужчин составляет 82,36 кг, поэтому средний избыточный вес составляет 8,36 кг. И после вычитания 74 кг из каждого веса среднее значение нового распределения составит $82,36 - 74 = 8,36$ кг. Фактически, когда мы **сдвигаем** данные, добавляя (или вычитая) константу к каждому значению, все меры положения (центр, процентили, минимум, максимум) **увеличиваются (уменьшаются)** на ту же самую константу.

А как ситуация обстоит с разбросом? Как добавление или вычитание константы влияет на разброс распределения? Посмотрите на две гистограммы выше еще раз. При добавлении или вычитании константы каждое значение данных сдвигается одинаково (равномерно), поэтому все распределение просто сдвигается. Его форма не меняется. Ни одна из мер разброса (размах, межквартильный размах, стандартное отклонение) не изменяется. Таким образом, **добавление (или вычитание) константы к каждому значению добавляет (или вычитает) ту же самую константу к мерам положения, но оставляет меры разброса неизменными**.

Теперь посмотрим, как деление или умножение каждого значения переменной на константу влияет на распределение.

Поскольку в каждом килограмме содержится около 2,2 фунта, мы можем преобразовать веса, умножив каждое значение на 2,2. Умножение или деление каждого значения на константу изменяет единицы измерения. Ниже приводятся гистограммы двух распределений веса (слева – в килограммах, справа – в фунтах), чтобы мы могли увидеть эффект умножения.

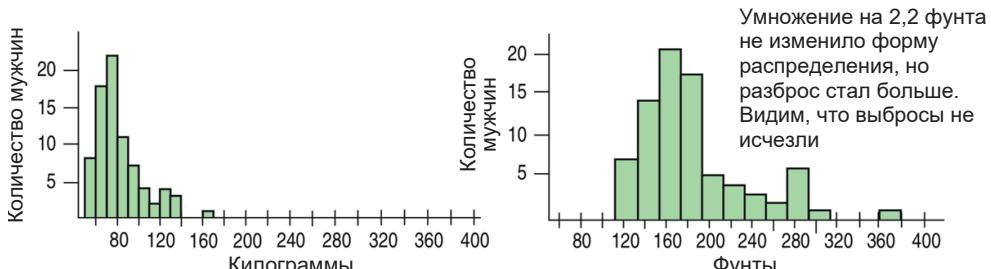


Рис. 71 Гистограмма распределения до и после умножения на константу

Что произошло с формой распределения? Хотя гистограммы не похожи друг на друга, мы видим, что форма действительно не изменилась: оба распределения являются унимодальными и смещены вправо.

Что произошло со средним? Оно умножается на 2,2. Мужчины весят в среднем 82,36 кг, что составляет 181,19 фунта. Как показывает ящиковую диаграмму, приведенная внизу, все меры положения изменяются аналогично: все они умножаются на эту же константу.

Что произошло с разбросом распределения? Взгляните на ящиковую диаграмму. Разброс в фунтах (справа) стал больше. Насколько больше? Если вы догадались, что он стал больше в 2,2 раза, то вы уже знаете, как изменятся меры разброса.

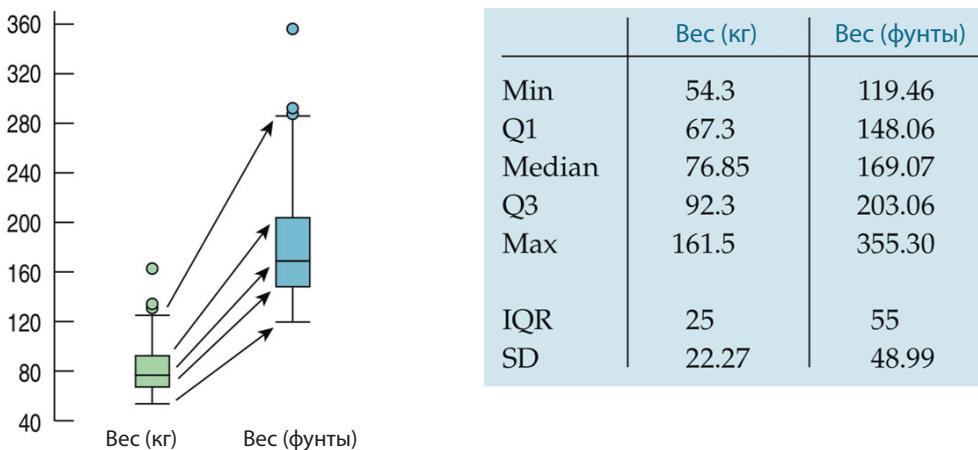


Рис. 72 Ящиковую диаграмму до и после умножения на константу

Давайте дополнительно убедимся в том, что стандартизация не меняет форму распределения и не в состоянии удалить выбросы.

Импортируем необходимые библиотеки и классы и загружаем данные.

```
# импортируем необходимые библиотеки и классы
import numpy as np
import pandas as pd
from sklearn.preprocessing import (StandardScaler,
                                   MinMaxScaler,
                                   RobustScaler)
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
import seaborn as sns

# загружаем набор данных
train = pd.read_csv('Data/Normality.csv', sep=';')
```

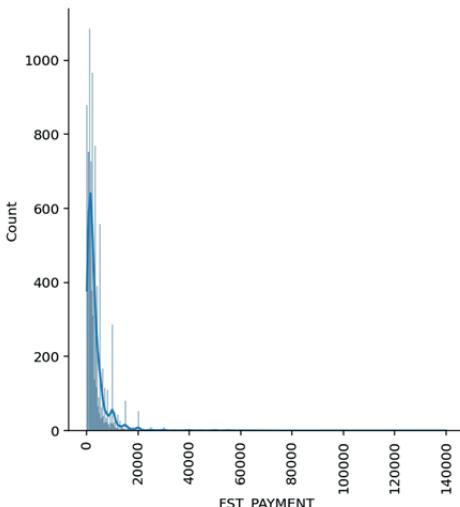
Давайте выведем гистограмму распределения, значения скоса и эксцесса, для переменной *FST_PAYMENT*.

```
# вычисляем скос, эксцесс, минимум, максимум,
# среднее, стандартное отклонение
print("Скос", train['FST_PAYMENT'].skew())
print("Эксцесс", train['FST_PAYMENT'].kurtosis())
print("Минимальное значение", train['FST_PAYMENT'].min())
print("Максимальное значение", train['FST_PAYMENT'].max())
print("Среднее значение", train['FST_PAYMENT'].mean())
print("Стандартное отклонение", train['FST_PAYMENT'].std())
```

```
# строим гистограмму распределения
sns.distplot(data=train, x='FST_PAYMENT', kde=True)
plt.xticks(rotation=90)
```

```
plt.show()
```

Скос 6.7435491347775
 Эксцесс 88.52885252034434
 Минимальное значение 0.0
 Максимальное значение 140000.0
 Среднее значение 3350.7137969219248
 Стандартное отклонение 5080.739926908223



Мы видим правостороннюю асимметрию распределения. Давайте попробуем ее устраниć с помощью различных способов стандартизации.

```
# создаем экземпляр класса StandardScaler
standardscaler = StandardScaler()
# обучаем и применяем модель стандартизации
train['FST_PAYMENT_standardscaled'] = standardscaler.fit_transform(
    train[['FST_PAYMENT']])

# создаем экземпляр класса MinMaxScaler
minmaxscaler = MinMaxScaler()
# обучаем и применяем модель стандартизации
train['FST_PAYMENT_minmaxscaled'] = minmaxscaler.fit_transform(
    train[['FST_PAYMENT']])
```

```

# создаем экземпляр класса RobustScaler
robustscaler = RobustScaler()
# обучаем и применяем модель стандартизации
train['FST_PAYMENT_robustscaled'] = robustscaler.fit_transform(
    train[['FST_PAYMENT']])

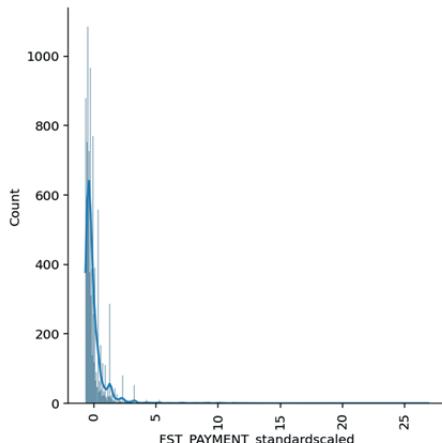
# записываем наши переменные в список
num_cols = train.columns[train.columns.str.contains('scaled')]

# по каждой переменной в списке вычисляем скос, эксцесс,
# минимум, максимум, среднее, стандартное отклонение,
# строим гистограмму распределения и график квантиль-квантиль
for col in num_cols:
    print(col)
    print("Скос", train[col].skew())
    print("Эксцесс", train[col].kurtosis())
    print("Минимальное значение", train[col].min())
    print("Максимальное значение", train[col].max())
    print("Среднее значение", train[col].mean())
    print("Стандартное отклонение", train[col].std())
    print("")

# строим гистограмму распределения
sns.displot(data=train, x=col, kde=True)
plt.xticks(rotation=90)
plt.show()

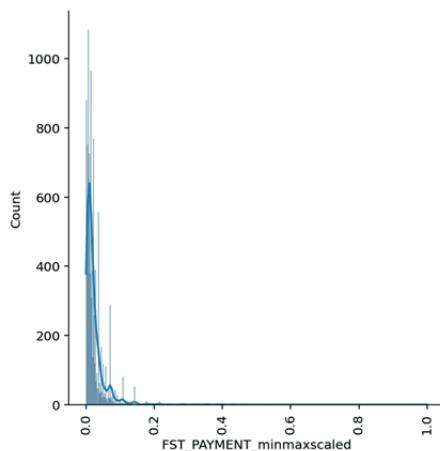
```

FST_PAYMENT_standardscaled
Скос 6.7435491347775
Эксцесс 88.52885252034432
Минимальное значение -0.6595242185440513
Максимальное значение 26.896810399168672
Среднее значение 9.524471895606669e-17
Стандартное отклонение 1.0000469252246733

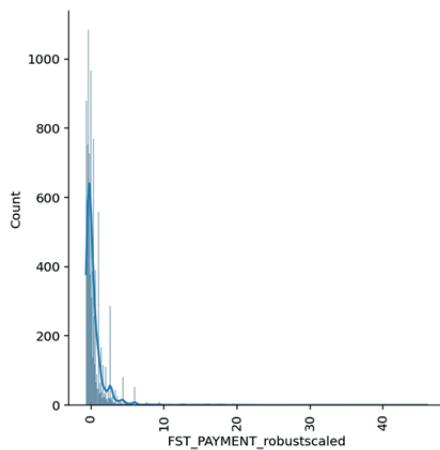


FST_PAYMENT_minmaxscaled
Скос 6.7435491347775
Эксцесс 88.52885252034437
Минимальное значение 0.0

Максимальное значение 1.0
Среднее значение 0.023933669978013664
Стандартное отклонение 0.036290999477916204



FST_PAYMENT_robustscaled
Скос 6.7435491347775
Эксцесс 88.52885252034432
Минимальное значение -0.6666666666666666
Максимальное значение 46.0
Среднее значение 0.45023793230730763
Стандартное отклонение 1.6935799756360734



Видим, что после применения различных видов стандартизации правосторонняя асимметрия сохранилась. Теперь заменим значения в первых трех наблюдениях переменной *FST_PAYMENT* на аномально большие и взглянем на результат.

```
# заменяем значения в первых трех наблюдениях
# переменной FST_PAYMENT на аномально большие
train.iloc[0, 3] = 9999999
train.iloc[1, 3] = 9999999
train.iloc[2, 3] = 9999999
```

```
# смотрим результат
train['FST_PAYMENT'].head()
```

```
0    9999999.0
1    9999999.0
2    9999999.0
3      790.0
4     1112.0
Name: FST_PAYMENT, dtype: float64
```

Теперь выполним стандартизацию с помощью класса `RobustScaler`, которую часто приписывают способность удалять выбросы, и выведем результаты.

```
# обучаем и применяем модель стандартизации
train['FST_PAYMENT_robustscaled'] = robustscaler.fit_transform(
    train[['FST_PAYMENT']])
# взглянем на результат
train['FST_PAYMENT_robustscaled'].head()
```

```
0    3332.666333
1    3332.666333
2    3332.666333
3      -0.403333
4      -0.296000
Name: FST_PAYMENT_robustscaled, dtype: float64
```

Как видим, выбросы остались, просто вместо десятков миллионов мы теперь имеем дело с тысячами.

На следующем слайде показаны способы, как можно выполнить стандартизацию с помощью готовых классов и функций, а также вручную.

Автоматически с помощью класса `StandardScaler` библиотеки `scikit-learn` ко всем столбцам

```
# импортируем класс StandardScaler
from sklearn.preprocessing import StandardScaler
# создаем экземпляр класса StandardScaler
scaler = StandardScaler()
# обучаем модель
scaler.fit(X_train)
# преобразовываем данные
X_tr_scaled = scaler.transform(X_train)
X_tst_scaled = scaler.transform(X_test)
```

Автоматически с помощью класса `StandardScaler` библиотеки `scikit-learn` к списку отобранных столбцов

```
# создаем список переменных
cust_cols = ['tenure', 'age']
```

```
# создаем экземпляр класса StandardScaler
scaler = StandardScaler()
# обучаем модель
scaler.fit(X_train[cust_cols])
# применяем модель
X_train[cust_cols] = scaler.transform(X_train[cust_cols])
X_test[cust_cols] = scaler.transform(X_test[cust_cols])
```

Вручную

```
# выделяем количественные переменные в отдельный список
num_cols = train.dtypes[train.dtypes != 'object'].index
# создаем копию обучающего набора
train_copy = train.copy()
# выполняем стандартизацию
for i in num_cols:
    train[i] = (train[i] - train[i].mean()) / train[i].std()
    test[i] = (test[i] - train_copy[i].mean()) / train_copy[i].std()
```

19. Собираем все вместе

В этом разделе рассмотрим задачу Give Me Some Credit с Kaggle <https://www.kaggle.com/c/GiveMeSomeCredit>. Файл исторических данных записан в файле *cs_training.csv*. Он содержит записи о 150 000 клиентов, классифицированных на два класса: 0 – просрочки 90+ нет (139 974 клиента) и 1 – просрочка 90+ есть (10 026 клиентов).

Список исходных переменных включает в себя:

- категориальный признак Уникальный идентификатор [*Unnamed: 0*];
- категориальную зависимую переменную Наличие просрочки 90+ по данным банка [*SeriousDlqin2yrs*];
- количественный признак Утилизация [*RevolvingUtilizationOfUnsecuredLines*];
- количественный признак Возраст клиента [*age*];
- количественный признак Количество просрочек 30–59 дней по данным БКИ [*NumberOfTime30-59DaysPastDueNotWorse*];
- количественный признак Коэффициент долговой нагрузки [*DebtRatio*];
- количественный признак Ежемесячный заработка [*MonthlyIncome*];
- количественный признак Количество кредитов [*NumberOfOpenCreditLinesAndLoans*];
- количественный признак Количество просрочек 90+ по данным БКИ [*NumberOfTimes90DaysLate*];
- категориальный признак Количество ипотечных кредитов [*NumberOfRealEstateLoansOrLines*];
- количественный признак Количество просрочек 60–89 дней по данным БКИ [*NumberOfTime60-89DaysPastDueNotWorse*];
- количественный признак Количество иждивенцев [*NumberOfDependents*].

Задача состоит в том, чтобы с помощью метода логистической регрессии построить модель прогнозирования просрочки. Критерием качества модели является оценка AUC-ROC.

Начнем с построения базовой модели.

Давайте импортируем необходимые библиотеки, модули, классы и функции.

```
# импортируем библиотеки numpy, pandas и missingno
import numpy as np
import pandas as pd
import missingno as msno
# импортируем модуль stats библиотеки scipy
from scipy import stats
# импортируем функцию train_test_split(), с помощью
# которой разбиваем данные на обучающие и тестовые
from sklearn.model_selection import train_test_split
# импортируем библиотеку seaborn
import seaborn as sns
# импортируем классы SimpleImputer, StandardScaler, Pipeline
# и функцию cross_val_score для нашей функции importance_auc()
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
# импортируем функцию roc_auc_score() для
# вычисления AUC-ROC
from sklearn.metrics import roc_auc_score
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression
```

Записываем файл исторических данных *cs_training.csv* в датафрейм *data* с помощью функции *pd.read_csv()*. С помощью *index_col* указываем индекс столбца (категориальный признак Уникальный идентификатор [*Unnamed: 0*]), который прочитываем в качестве индекса.

```
# считываем данные
data = pd.read_csv('Data/cs-training.csv', index_col='Unnamed: 0')
data.head()
```

SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30-59DaysPastDueNotWorse	DebtRatio	MonthlyIncome	NumberOfOpenCreditLinesAndLoans
1	0.766127	45	2	0.802982	9120.0	13
2	0.957151	40	0	0.121876	2600.0	4
3	0.658180	38	1	0.085113	3042.0	2
4	0.233810	30	0	0.036050	3300.0	5
5	0.907239	49	1	0.024926	63588.0	7

Посмотрим типы переменных и наличие пропусков.

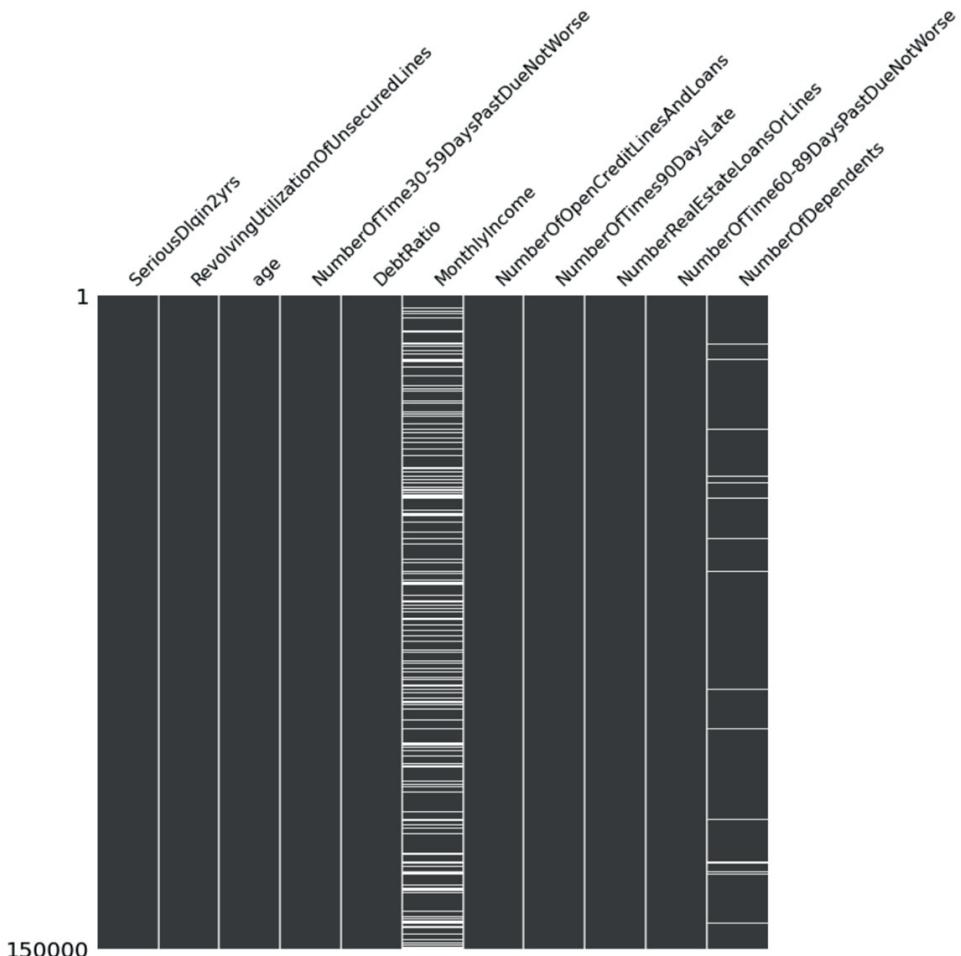
```
# смотрим типы переменных и информацию о количестве пропусков
data.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 150000 entries, 1 to 150000
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   SeriousDlqin2yrs    150000 non-null   int64  
 1   RevolvingUtilizationOfUnsecuredLines 150000 non-null   float64 
 2   age                150000 non-null   int64  
 3   NumberOfTime30-59DaysPastDueNotWorse 150000 non-null   int64  
 4   DebtRatio          150000 non-null   float64 
 5   MonthlyIncome      120269 non-null   float64 
 6   NumberOfOpenCreditLinesAndLoans        150000 non-null   int64  
 7   Numberoftimes90DaysLate               150000 non-null   int64  
 8   NumberRealEstateLoansOrLines         150000 non-null   int64  
 9   Numberoftime60-89DaysPastDueNotWorse 150000 non-null   int64  
 10  NumberofDependents                 146076 non-null   float64 
dtypes: float64(4), int64(7)
memory usage: 13.7 MB
```

Все переменные корректно определены. Переменные *MonthlyIncome* и *NumberofDependents* имеют пропуски (выделены красными рамками).

Давайте визуализируем пропуски с помощью библиотеки *missingno*.

```
# визуализируем пропуски с помощью missingno
msno.matrix(data, sparkline=False, figsize=(11, 11));
```



Выведем статистики для наших переменных, сократив количество десятичных знаков до трех.

```
# смотрим статистики для количественных переменных
pd.set_option('display.float_format', lambda x: '%.3f' % x)
data.describe()
```

	SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30-59DaysPastDueNotWorse	DebtRatio	MonthlyIncome
count	150000.000	150000.000	150000.000	150000.000	150000.000	120269.000
mean	0.067	6.048	52.295	0.421	353.005	6670.221
std	0.250	249.755	14.772	4.193	2037.819	14384.674
min	0.000	0.000	0.000	0.000	0.000	0.000
25%	0.000	0.030	41.000	0.000	0.175	3400.000
50%	0.000	0.154	52.000	0.000	0.367	5400.000
75%	0.000	0.559	63.000	0.000	0.868	8249.000
max	1.000	50708.000	109.000	98.000	329664.000	3008750.000

NumberOfOpenCreditLinesAndLoans	NumberOfTimes90DaysLate	NumberRealEstateLoansOrLines	NumberOfTime60-89DaysPastDueNotWorse	NumberOfDependents
150000.000	150000.000	150000.000	150000.000	146076.000
8.453	0.266	1.018	0.240	0.757
5.146	4.169	1.130	4.155	1.115
0.000	0.000	0.000	0.000	0.000
5.000	0.000	0.000	0.000	0.000
8.000	0.000	1.000	0.000	0.000
11.000	0.000	2.000	0.000	1.000
58.000	98.000	54.000	98.000	20.000

Переменная *RevolvingUtilizationOfUnsecuredLines* имеет значения больше 1, тогда как утилизация обычно варьирует в диапазоне от 0 до 1, исключением может быть ситуация, когда мы суммируем утилизации по нескольким картам, однако даже в таком случае утилизация редко превышает значения 3 или 4.

У переменной *age* есть значения меньше 18 лет и больше 70 лет (в США минимальный возраст для получения кредита – 18 лет, строгого ограничения верхней границы возраста для желающих оформить кредит нет, но некоторые банки устанавливают планку в 70 лет).

У переменных *NumberOfTime30-59DaysPastDueNotWorse*, *NumberOfTime60-89DaysPastDueNotWorse*, *NumberOfTimes90DaysLate*, характеризующих глубину просрочки, есть большое значение 98, которое, скорее всего, является служебным кодом, а не количеством просрочек.

Переменная *Коэффициент долговой нагрузки [DebtRatio]* имеет значения больше 1, максимальное значение равно 329 664. Здесь мы понимаем, что такое аномально большое значение не может быть обусловлено объективными причинами. Положительные значения больше 1 указывают на то, что ежемесячная сумма долговых обязательств больше ежемесячного дохода и при таком раскладе банк не выдаст кредит. Впрочем, бывают практики, когда банк суммирует значения коэффициента долговой нагрузки у клиента за разные периоды, обычно кварталы, и тогда значение может быть больше 1, но редко превышает значения больше 4. Речь идет либо об ошибках ввода данных, либо об искусственном искажении данных (что часто имеет место на соревнованиях Kaggle), либо об ошибке, произошедшей при импорте данных (например, мог произойти сдвиг десятичного разделителя: значение 0,5 стало значением 50). Поэтому применять преобразования логарифмом, квадратным корнем для борьбы с такими значениями не имеет смысла, здесь мы можем как раз применить винзоризацию (помним, что винзоризацию мы делаем уже после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки, потому что нам нужно будет вычислить нижний и верхний квантили) или попробовать сдвиг десятичного разделителя (это можно сделать до разбиения на обучающую и тестовую выборки или до цикла перекрестной проверки, нам не требуется вычислять статистики).

Переменная *MonthlyIncome* имеет аномально низкое минимальное значение (0), тогда как в Штатах кредитуют при минимальном ежемесячном доходе не менее 1200–1400\$ (умножаем минимальную почасовую ставку 7,25 \$ в час на 8-часовой рабочий день и затем на ~22 рабочих дня, получаем примерно 1300\$). Кроме того, у переменной *MonthlyIncome* есть аномально большое зна-

чение (3 008 750). Опять нам важно понять природу очень небольших и очень больших значений дохода, надо выяснить, они объективны или обусловлены ошибками ввода.

Начнем обработку данных с переменной *MonthlyIncome*. Для начала выведем 10 наибольших значений переменной *MonthlyIncome*.

```
# выведем 10 наибольших значений MonthlyIncome
data['MonthlyIncome'].nlargest(10)
```

```
73763    3008750.000
137140   1794060.000
111365   1560100.000
50640    1072500.000
122543   835040.000
123291   730483.000
93564    702500.000
96549    699530.000
119136   649587.000
37078    629000.000
Name: MonthlyIncome, dtype: float64
```

Подробнее взглянем на клиентов, у которых ежемесячный доход больше 500 000 \$.

```
# взглянем на значения MonthlyIncome больше 500 000
data[data['MonthlyIncome'] > 500000]
```

SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30-59DaysPastDueNotWorse	DebtRatio	MonthlyIncome	NumberOfOpenCreditLinesAndLoans
35974	0	0.440	64	0	582369.000	11
37079	0	0.000	83	0	629000.000	3
50641	0	0.469	44	1	1072500.000	9
73764	0	0.007	52	0	3008750.000	10
93565	0	0.072	50	0	702500.000	12
96550	0	0.064	52	0	699530.000	11
111366	0	0.164	44	0	1560100.000	12
119137	0	0.151	49	0	649587.000	8
122544	0	0.042	55	0	835040.000	8
123292	0	0.226	67	1	730483.000	23
137141	0	0.000	68	0	1794060.000	15
137427	0	0.080	61	0	562466.000	7

Видим, что данные значения дохода, скорее, носят объективный характер: у таких клиентов – возраст выше 40 лет, низкая утилизация и низкий коэффициент долговой нагрузки, нулевое количество просрочек 90+. Поэтому здесь целесообразно применить преобразования корнем, логарифмом и т. п.

Теперь взглянем на клиентов, у которых ежемесячный доход менее 1200 \$ в месяц.

```
# взглянем на значения MonthlyIncome менее 1200
data[data['MonthlyIncome'] < 1200]
```

SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30-59DaysPastDueNotWorse	DebtRatio	MonthlyIncome	NumberOfOpenCreditLinesAndLoans	
15	0	0.020	76	0	477.000	0.000	6
20	0	0.603	25	0	0.066	333.000	2
32	0	1.000	24	0	0.473	750.000	1
39	0	0.364	26	0	0.010	1000.000	1
45	0	0.369	68	0	1687.500	1.000	31
...
149883	0	0.062	77	0	714.500	1.000	3
149951	0	0.005	61	0	1940.000	0.000	10
149955	1	1.000	46	0	170.398	401.000	3
149962	1	0.920	31	1	0.177	1125.000	4
149994	0	1.000	22	0	0.000	820.000	1

5335 rows × 11 columns

Здесь у нас – пестрый состав: клиенты имеют разный возраст, разное соотношение долга к доходу, разную утилизацию. Переменная *NumberOfOpenCreditLinesAndLoans* указывает на то, что клиенты активно пользуются кредитами, что невозможно при доходе менее 1200 \$ в месяц. Здесь логично применить импутацию, объявляем все значения меньше 1200 \$ пропусками и затем заменяя их либо какой-либо статистикой (помним, что импутацию статистиками мы делаем уже после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки), либо минимально допустимым значением дохода, при котором выдается кредит. Последний сценарий импутации более популярен в банковской практике: мы ничего не знаем о доходе клиента, поэтому берем самую консервативную оценку (если предположить, что чем ниже доход, тем выше риск просрочки), а поскольку заменяя пропуски константой, то замену можно выполнить до разбиения на обучающую и тестовую выборки или до цикла перекрестной проверки.

Теперь взглянем на клиентов, у которых ежемесячный доход пропущен.

```
# взглянем на пропуски переменной MonthlyIncome
data[data['MonthlyIncome'].isnull()]
```

SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30-59DaysPastDueNotWorse	DebtRatio	MonthlyIncome	NumberOfOpenCreditLinesAndLoans	
7	0	0.306	57	0	5710.000	NaN	8
9	0	0.117	27	0	46.000	NaN	2
17	0	0.061	78	0	2058.000	NaN	10
33	0	0.083	62	0	977.000	NaN	6
42	0	0.073	81	0	75.000	NaN	7
...
149977	0	0.001	76	0	60.000	NaN	5
149978	0	0.236	29	0	349.000	NaN	3
149985	0	0.038	84	0	25.000	NaN	5
149993	0	0.872	50	0	4132.000	NaN	11
149998	0	0.246	58	0	3870.000	NaN	18

Опять видим, что клиенты имеют разный возраст, разное соотношение долга к доходу, утилизацию. Переменная *NumberOfOpenCreditLinesAndLoans* вновь указывает на то, что клиенты активно пользуются кредитами.

Заменяем пропуски в переменной *MonthlyIncome* и значения переменной *MonthlyIncome*, которые меньше 1200\$, минимально допустимым значением ежемесячного дохода – значением 1200.

Значения переменной *age* меньше 18 лет заменяют минимально допустимым значением возраста – значением 18 (вновь руководствуемся самой консервативной оценкой, если предположить, что чем ниже возраст, тем выше риск просрочки).

```
# заменяем пропуски и значения < 1200 в MonthlyIncome
# минимально допустимым значением дохода
data['MonthlyIncome'] = np.where((data['MonthlyIncome'].isnull() |
                                   (data['MonthlyIncome'] < 1200),
                                   1200,
                                   data['MonthlyIncome']))

# значения переменной age меньше 18 заменяем
# минимально допустимым значением возраста
data['age'] = np.where(data['age'] < 18, 18, data['age'])
```

Пропуски в переменной *NumberOfDependents* заменяют нулем, а значения больше 4 выделяем в отдельную категорию. Все значения переменной *RevolvingUtilizationOfUnsecuredLines* больше 2 пометим как пропуски. Здесь мы предположим, что значением переменной является утилизация по нескольким картам.

```
# пропуски в переменной NumberOfDependents заменяем нулем
data.loc[data['NumberOfDependents'].isnull(), 'NumberOfDependents'] = 0

# все значения переменной RevolvingUtilizationOfUnsecuredLines
# больше 2 пометим как пропуски
data['RevolvingUtilizationOfUnsecuredLines'] = np.where(
    data['RevolvingUtilizationOfUnsecuredLines'] > 2,
    np.NaN,
    data['RevolvingUtilizationOfUnsecuredLines'])
```

Теперь займемся конструированием признаков, не предполагающим использование статистик, поэтому его можно выполнить до разбиения на обучающую и тестовую выборки.

Создаем переменную *Ratio* – отношение количества просрочек 90+ к общему количеству просрочек – и индикаторы нулевых значений *NumberOfOpenCreditLinesAndLoans_is_0*, *NumberRealEstateLoansOr-Lines_is_0* и *RevolvingUtilizationOfUnsecuredLines_is_0*.

```
# создаем переменную Ratio - отношение количества
# просрочек 90+ к общему количеству просрочек
sum_of_delinq = (data['NumberOfTimes90DaysLate'] +
                  data['NumberOfTime30-59DaysPastDueNotWorse'] +
                  data['NumberOfTime60-89DaysPastDueNotWorse'])

cond = (data['NumberOfTimes90DaysLate'] == 0) | (sum_of_delinq == 0)
data['Ratio'] = np.where(
    cond, 0, data['NumberOfTimes90DaysLate'] / sum_of_delinq)
```

```
# создаем индикатор нулевых значений переменной
# NumberOfOpenCreditLinesAndLoans
data['NumberOfOpenCreditLinesAndLoans_is_0'] = np.where(
    data['NumberOfOpenCreditLinesAndLoans'] == 0, 'T', 'F')

# создаем индикатор нулевых значений переменной
# NumberRealEstateLoansOrLines
data['NumberRealEstateLoansOrLines_is_0'] = np.where(
    data['NumberRealEstateLoansOrLines'] == 0, 'T', 'F')

# создаем индикатор нулевых значений переменной
# RevolvingUtilizationOfUnsecuredLines
data['RevolvingUtilizationOfUnsecuredLines_is_0'] = np.where(
    data['RevolvingUtilizationOfUnsecuredLines'] == 0, 'T', 'F')
```

Теперь взглянем на уникальные значения целочисленных переменных, характеризующих глубину просрочки.

```
# смотрим уникальные значения целочисленных переменных,
# характеризующих глубину просрочки
for col in ['NumberOfTime30-59DaysPastDueNotWorse',
            'NumberOfTime60-89DaysPastDueNotWorse',
            'NumberOfTimes90DaysLate']:
    print(f'{col}:\n{data[col].unique()}\n')

NumberOfTime30-59DaysPastDueNotWorse:
[ 2  0  1  3  4  5  7 10  6 98 12  8  9 96 13 11]

NumberOfTime60-89DaysPastDueNotWorse:
[ 0  1  2  5  3 98  4  6  7  8 96 11  9]

NumberOfTimes90DaysLate:
[ 0  1  3  2  5  4 98 10  9  6  7  8 15 96 11 13 14 17 12]
```

Видим, что у переменных *NumberOfTime30-59DaysPastDueNot-Worse*, *NumberOfTime60-89DaysPastDueNotWorse*, *NumberOfTimes90-DaysLate*, характеризующих глубину просрочки, есть большие значения 96 и 98, которые скорее всего являются служебными кодами (так обычно указывают ошибки обработки запроса в БКИ). Если строится модель логистической регрессии, то такие переменные обычно превращают в категориальные с укрупнением категорий, итоговый набор категорий можно получить с помощью оптимального биннинга на основе CHAID, исключив значения 96 и 98. Допустим, мы получили 5 категорий, статистически значимо отличающихся друг от друга по зависимой переменной: '0 просочек', '1 просрочка', '2 просрочки', '3 просрочки', 'более 3 просрочек'. Наблюдения со значениями 96 и 98 можно отнести к категории 'более 3 просрочек', поскольку мы ничего не знаем о количестве просрочек для этих наблюдений и опять должны руководствоваться консервативным сценарием. Если строится модель градиентного бустинга, биннинг обычно не применяется (деревья сами выполняют его), а значения 96 и 98 кодируются единым значением вне диапазона.

Мы сейчас укрупним категории переменных, характеризующих глубину просрочки. Переменную *NumberOfDependents* тоже превратим в категори-

альную с укрупнением категорий. Как показывает практика, заемщики с 4 иждивенцами не сильно отличаются по зависимой переменной от заемщиков с 5 иждивенцами, а заемщики с 5 иждивенцами не сильно отличаются по зависимой переменной от заемщиков с 6 иждивенцами и т. д. Это опять же можно проверить с помощью метода CHAID. Кроме того, новоиспеченные категориальные переменные приведем к единому строковому формату.

```
# преобразовываем переменные в категориальные, применив
# биндинг и перевод в единый строковый формат
for col in ['NumberOfTime30-59DaysPastDueNotWorse',
            'NumberOfTime60-89DaysPastDueNotWorse',
            'NumberOfTimes90DaysLate',
            'NumberOfDependents']:
    data.loc[data[col] > 3, col] = 4
    data[col] = data[col].apply(lambda x: f"cat_{x}")

# смотрим частоты
for col in ['NumberOfTime30-59DaysPastDueNotWorse',
            'NumberOfTime60-89DaysPastDueNotWorse',
            'NumberOfTimes90DaysLate',
            'NumberOfDependents']:
    print(data[col].value_counts())
    print('')

cat_0      126018
cat_1      16033
cat_2      4598
cat_3      1754
cat_4      1597
Name: NumberOfTime30-59DaysPastDueNotWorse, dtype: int64

cat_0      142396
cat_1      5731
cat_2      1118
cat_4      437
cat_3      318
Name: NumberOfTime60-89DaysPastDueNotWorse, dtype: int64

cat_0      141662
cat_1      5243
cat_2      1555
cat_4      873
cat_3      667
Name: NumberOfTimes90DaysLate, dtype: int64

cat_0.0    90826
cat_1.0    26316
cat_2.0    19522
cat_3.0    9483
cat_4.0    3853
Name: NumberOfDependents, dtype: int64
```

Теперь создаем взаимодействия.

```

# пишем функцию, которая создает взаимодействие
# в результате конъюнкции переменных
# f1 и f2
def make_interact(df, interact_list):
    for i in lst:
        f1 = i[0]
        f2 = i[1]
        df[f1 + ' + ' + f2 + '_interact'] = (df[f1].astype(str) + ' + '
                                                + df[f2].astype(str))

# создаем список списков - список 2-факторных взаимодействий
lst = [
    ['NumberOfDependents',
     'NumberOfTime30-59DaysPastDueNotWorse'],
    ['NumberOfTime60-89DaysPastDueNotWorse',
     'NumberOfTimes90DaysLate'],
    ['NumberOfTime30-59DaysPastDueNotWorse',
     'NumberOfTime60-89DaysPastDueNotWorse'],
    ['NumberOfRealEstateLoansOrLines_is_0',
     'NumberOfTimes90DaysLate'],
    ['NumberOfOpenCreditLinesAndLoans_is_0',
     'NumberOfTimes90DaysLate']
]
# создаем взаимодействия
make_interact(data, interact_list=lst)

```

Не забываем, что у нас могут быть редкие комбинации, поэтому редкие категории нужно укрупнить.

```

# укрупняем редкие категории
interact_columns = data.columns[data.columns.str.contains('interact')]
for col in interact_columns:
    data.loc[data[col].value_counts()[data[col]].values < 55, col] = 'Other'

```

Давайте сделаем случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```

# создаем обучающий массив признаков, тестовый массив признаков,
# обучающий массив меток, тестовый массив меток
train, test, y_train, y_test = train_test_split(
    data.drop('SeriousDlqin2yrs', axis=1),
    data['SeriousDlqin2yrs'],
    test_size=.3,
    stratify=data['SeriousDlqin2yrs'],
    random_state=100)

```

Теперь выделим наиболее важные признаки, преобразования которых кардинально повлияют на качество модели. Мы просто каждый раз строим модель с одним признаком и оцениваем AUC-ROC на проверочных блоках k -блочной перекрестной проверки, запущенной на обучающей выборке. Часто такой способ используется перед построением модели логистической регрессии, признаки должны быть предварительно преобразованы в количественные (для

категориальных признаков выполняется либо дамми-кодирование, либо WoE-кодирование, либо такие признаки игнорируют). Обратите внимание, что при вычислении AUC-ROC будет важен способ импутации пропусков, если в переменной были пропуски, и способ обработки выбросов, если таковые в переменной присутствовали.

Пишем и применяем функцию `importance_auc()`, вычисляющую оценку AUC-ROC по каждому признаку.

```
# пишем функцию, вычисляющую AUC для модели с одним признаком
def importance_auc(train, y_train, imp_strategy):
    # создаем копию обучающего набора
    train_copy = train.copy()
    # создаем список переменных
    col_list = train_copy.select_dtypes(include=['number']).columns
    # создаем список, в который будем записывать auc
    auc_list = []
    for i in col_list:
        pipe = Pipeline([
            ('imputer', SimpleImputer(strategy=imp_strategy)),
            ('scaler', StandardScaler()),
            ('logreg', LogisticRegression(solver='liblinear'))
        ])
        auc = cross_val_score(pipe,
                              train_copy[[i]],
                              y_train,
                              scoring='roc_auc',
                              cv=5).mean()
        auc_list.append(auc)

    result = pd.DataFrame({'Переменная': col_list,
                           'AUC': auc_list})
    result = np.round(result.sort_values(
        by='AUC', ascending=False), 3)
    cm = sns.light_palette('yellow', as_cmap=True)
    return (result.style.background_gradient(cmap=cm))

# применяем функцию importance_auc()
importance_auc(train, y_train, 'median')
```

	Переменная	AUC
0	RevolvingUtilizationOfUnsecuredLines	0.779000
6	Ratio	0.652000
1	age	0.634000
4	NumberOfOpenCreditLinesAndLoans	0.540000
5	NumberRealEstateLoansOrLines	0.535000
3	MonthlyIncome	0.532000
2	DebtRatio	0.475000

Наиболее важными признаками стали признаки *RevolvingUtilizationOfUnsecuredLines* и *Ratio*. Обработка этих переменных существенно повлияет на качество модели.

Давайте выполним импутацию пропусков в переменной *RevolvingUtilizationOfUnsecuredLines* средним значением.

```
# выполняем импутацию пропусков в переменной
# RevolvingUtilizationOfUnsecuredLines средним
train['RevolvingUtilizationOfUnsecuredLines'].fillna(
    train['RevolvingUtilizationOfUnsecuredLines'].mean(), inplace=True)
test['RevolvingUtilizationOfUnsecuredLines'].fillna(
    train['RevolvingUtilizationOfUnsecuredLines'].mean(), inplace=True)
```

Для обработки выбросов в переменной *MonthlyIncome* применим логарифмическое преобразование, поскольку ранее мы выяснили, что выбросы носят объективную природу.

```
# выполняем логарифмическое преобразование
# переменной MonthlyIncome
train['MonthlyIncome'] = np.log(train['MonthlyIncome'].clip(0.01))
test['MonthlyIncome'] = np.log(test['MonthlyIncome'].clip(0.01))
```

Теперь займемся признаком *Коэффициент долговой нагрузки [DebtRatio]*. Допустим, у человека ежемесячный доход составляет 100 000 рублей, а ежемесячная сумма выплат по кредитным обязательствам составляет 60 000 рублей, коэффициент долговой нагрузки равен $60\ 000 / 100\ 000 = 0,6$. Обычно при значениях коэффициента 0,6 и выше (в зависимости от риска-аппетита банка) в кредите отказывают.

Выведем сводку описательных статистик по этой переменной.

```
# выводим описательные статистики
train['DebtRatio'].describe()

count      105000.000000
mean        353.036672
std         2051.316138
min         0.000000
25%        0.174913
50%        0.366826
75%        0.870187
max        329664.000000
Name: DebtRatio, dtype: float64
```

Предположим, в банке, с данными которого мы работаем, кредит не выдают при коэффициенте 0,7 и выше. Давайте с помощью функции *percentileofscore()* модуля stats библиотеки scipy найдем процентиль, соответствующий значению 0,7.

```
# допустим, мы знаем, что не выдаем кредиты с коэффициентом
# долговой нагрузки больше 0.7, найдем соответствующий этому
# значению процентиль
stats.percentileofscore(train['DebtRatio'], 0.7)
```

71.64095238095238

С помощью функции `pr.percentile()` проверим, какому числу соответствует найденный процентиль.

```
# проверим, какому числу соответствует этот процентиль  
pr.percentile(train['DebtRatio'], 71.641)
```

```
0.6999722155381308
```

Убедились, что найденный процентиль соответствует значению 0,7. В Python винзоризацию можно выполнить с помощью функции `winsorize()` подмодуля `mstats` модуля `stats` библиотеки `scipy`. С помощью параметра `limits` мы задаем кортеж из двух чисел с плавающей точкой – проценты наблюдений, которые мы приравниваем к соответствующим процентилям с обоих концов диапазона. В данном случае мы выполняем одностороннюю винзоризацию, установив только верхнюю границу. Поскольку нижнюю границу мы не меняем, первое значение кортежа приравниваем к 0, а второе значение будет равно $1 - 0,71641 = 0,28359$. Винзоризацию выполним в копии обучающего набора, чтобы не модифицировать исходный обучающий набор.

```
# создаем копию серии  
tr = train['DebtRatio'].copy()  
# выполняем винзоризацию  
debt_ratio = stats.mstats.winsorize(debt_ratio,  
                                     limits=(0, 0.28359))
```

Снова выведем сводку описательных статистик по нашей переменной.

```
# выводим описательные статистики  
debt_ratio.describe()
```

```
count    105000.000  
mean      0.390  
std       0.250  
min       0.000  
25%      0.175  
50%      0.367  
75%      0.700  
max       0.700  
Name: DebtRatio, dtype: float64
```

Видим, что максимальное значение теперь равно 0,7.

Кроме того, можно попробовать следующую схему винзоризации.

1. Задаем нижний quartиль Q_1 и верхний quartиль Q_3 .
2. Задаем межквартильный размах по формуле $k \cdot (Q_3 - Q_1)$, где k – коэффициент, наиболее часто употребляемое значение которого равно 1,5.
3. Выполняем обработку выбросов по правилу:

- ♦ если $x_i < Q_1 - k \cdot (Q_3 - Q_1)$, то заменяем x_i на $Q_1 - k \cdot (Q_3 - Q_1)$;
- ♦ если $x_i \geq Q_3 + k \cdot (Q_3 - Q_1)$, то заменяем x_i на $Q_3 + k \cdot (Q_3 - Q_1)$.

Давайте реализуем эту схему в виде собственного класса `OutlierRemover`.

```

# создаем собственный класс, выполняющий винзоризацию
class OutlierRemover(BaseEstimator, TransformerMixin):
    """
    Параметры:
    lower_quantile: float, по умолчанию 0.25
        Нижний квантиль.
    upper_quantile: float, по умолчанию 0.75
        Верхний квантиль.
    k: float, по умолчанию 1.5
        Коэффициент.
    copy: bool, по умолчанию True
        Возвращает копию.
    """

    def __init__(self, copy=True, lower_quantile=0.25,
                 upper_quantile=0.75, k=1.5):
        # все параметры для инициализации публичных атрибутов
        # должны быть заданы в методе __init__

        # публичные атрибуты
        self.copy = copy
        self.lower_quantile = lower_quantile
        self.upper_quantile = upper_quantile
        self.k = k

    def __is_numpy(self, X):
        # частный метод, который с помощью функции isinstance()
        # проверяет, является ли наш объект массивом NumPy
        return isinstance(X, np.ndarray)

    def fit(self, X, y=None):
        # fit должен принимать в качестве аргументов X и y

        # обучение модели осуществляется прямо здесь
        # создаем пустой словарь, в котором ключами
        # будут имена / целые числа, а значениями - кортежи
        self._dict = {}

        # если 1D-массив, то переводим в 2D
        if len(X.shape) == 1:
            X = X.reshape(-1, 1)

        # записываем количество столбцов
        ncols = X.shape[1]

        # записываем результат __is_numpy()
        is_np = self.__is_numpy(X)

        # если объект - массив NumPy, выполняем следующие действия:
        if is_np:
            # по каждому столбцу массива NumPy
            for col in range(ncols):
                lower = np.quantile(X[:, col], self.lower_quantile)
                upper = np.quantile(X[:, col], self.upper_quantile)
                IQR = (upper - lower) * self.k
                self._dict[col] = (lower, upper, IQR)
            # в противном случае, т. е. если объект - датафрейм pandas,

```

```
# выполняем следующие действия:
else:
    # по каждому столбцу датафрейма pandas
    for col in X.columns:
        # вычисляем и записываем в словарь
        lower = X[col].quantile(self.lower_quantile)
        upper = X[col].quantile(self.upper_quantile)
        IQR = (upper - lower) * self.k
        self._dict[col] = (lower, upper, IQR)

    # fit возвращаем self
    return self

def transform(self, X):
    # transform принимает в качестве аргумента только X

    # выполняем копирование массива во избежание
    # предупреждения SettingWithCopyWarning
    # "A value is trying to be set on a copy of
    # a slice from a DataFrame (Происходит попытка изменить
    # значение в копии среза данных датафрейма)"
    if self.copy:
        X = X.copy()

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # записываем результат __is_numpy()
    is_np = self.__is_numpy(X)

    # применяем преобразование к X
    # если объект - массив NumPy, выполняем следующие действия:
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # заменяем
            X[:, col] = np.where(
                X[:, col] < (self._dict[col][0] - self._dict[col][2]),
                self._dict[col][0] - self._dict[col][2],
                X[:, col])
            X[:, col] = np.where(
                X[:, col] >= (self._dict[col][1] + self._dict[col][2]),
                self._dict[col][1] + self._dict[col][2],
                X[:, col])

    # в противном случае, т. е. если объект - датафрейм pandas,
    # выполняем следующие действия:
else:
    # по каждому столбцу датафрейма pandas
    for col in X.columns:
        # заменяем
        X[col] = np.where(
```

```

X[col] < (self._dict[col][0] - self._dict[col][2]),
self._dict[col][0] - self._dict[col][2],
X[col])
X[col] = np.where(
    X[col] >= (self._dict[col][1] + self._dict[col][2]),
    self._dict[col][1] + self._dict[col][2],
    X[col])
# transform возвращает X
return X

```

Сейчас мы выполним винзоризацию значений признака *DebtRatio* и создадим новые признаки на основе биннинга переменных *DebtRatio* и *RevolvingUtilizationOfUnsecuredLines*.

```

# выполняем винзоризацию переменной DebtRatio
# с помощью нашего класса OutlierRemover
rem = OutlierRemover(lower_quantile=0,
                      upper_quantile=0.8,
                      k=1.15)
rem.fit(train[['DebtRatio']])

train['DebtRatio'] = rem.transform(train[['DebtRatio']])
test['DebtRatio'] = rem.transform(test[['DebtRatio']])

# выполняем биннинг переменных DebtRatio и
# RevolvingUtilizationOfUnsecuredlines
bins = np.round(np.arange(0, 1.05, 0.05), 2).tolist()
bins[0] = -1
for col in ['DebtRatio', 'RevolvingUtilizationOfUnsecuredLines']:
    train[col+'_bins'] = pd.cut(
        train[col], bins, labels=bins[1:]).astype('float')
    train.loc[(train[col] > 1), col+'_bins'] = 2
    test[col+'_bins'] = pd.cut(
        test[col], bins, labels=bins[1:]).astype('float')
    test.loc[(test[col] > 1), col+'_bins'] = 2

# преобразовываем DebtRatio_bins в категориальную переменную
train['DebtRatio_bins'] = train['DebtRatio_bins'].apply(lambda x: f"cat_{x}")
test['DebtRatio_bins'] = test['DebtRatio_bins'].apply(lambda x: f"cat_{x}")

```

Применяем стандартизацию, чтобы привести количественные переменные к единому масштабу.

```

# выполняем стандартизацию
num_cols = train.select_dtypes(include=['number']).columns.tolist()
train_copy = train.copy()
for col in num_cols:
    train[col] = (train[col] - train[col].mean()) / train[col].std()
    test[col] = (test[col] - train_copy[col].mean()) / train_copy[col].std()

```

Проверяем, нет ли у нас пропусков, и выполняем дамми-кодирование.

```
# проверяем наличие пропусков
print(train.isnull().sum().sum())
print(test.isnull().sum().sum())
```

```
0
0
```

```
# выполняем дамми-кодирование
X_train = pd.get_dummies(train)
X_test = pd.get_dummies(test)
```

Проверяем, совпадает ли количество признаков в обучающей и тестовой выборках.

```
# проверяем количество признаков
print(X_train.shape[1], X_test.shape[1])
```

```
142 142
```

Приступаем к построению модели логистической регрессии и оцениваем ее качество на тестовой выборке.

```
# строим модель логистической регрессии
logreg = LogisticRegression(solver='liblinear').fit(X_train, y_train)
print("AUC на обучающей выборке: {:.3f}".format(
    roc_auc_score(y_train, logreg.predict_proba(X_train)[:, 1])))
print("AUC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))
```

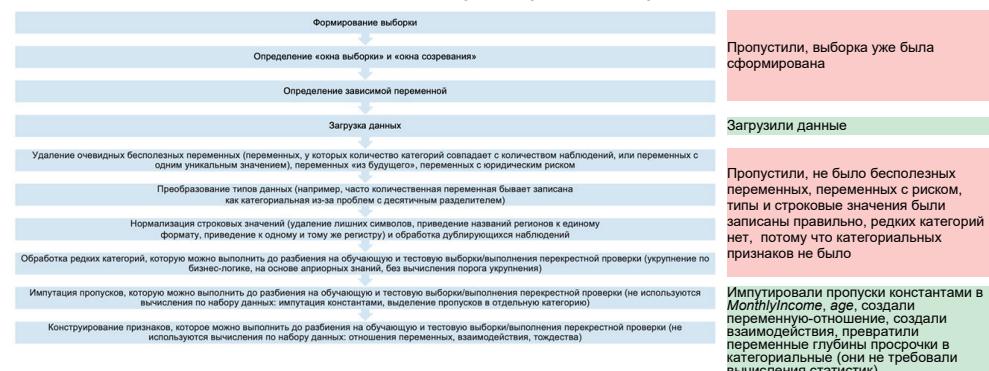
```
AUC на обучающей выборке: 0.864
```

```
AUC на тестовой выборке: 0.864
```

Важно, что предварительную подготовку мы выполняли в строгом соответствии с планом, приведенным в начале этой части.

План предварительной подготовки

До разбиения на обучающую и тестовую выборки



После разбиения на обучающую и тестовую выборки



Теперь выполним оптимизацию гиперпараметров. В качестве гиперпараметров мы будем рассматривать способы обработки конкретных признаков.

Давайте загрузим необходимые библиотеки, классы и функции.

```

# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd
# импортируем функцию train_test_split(), с помощью
# которой разбиваем данные на обучающие и тестовые
from sklearn.model_selection import train_test_split
# импортируем классы BaseEstimator и TransformerMixin,
# позволяющие написать собственные классы
from sklearn.base import BaseEstimator, TransformerMixin
# импортируем класс SimpleImputer, позволяющий
# выполнить импутацию пропусков
from sklearn.impute import SimpleImputer
# импортируем класс StandardScaler,
# позволяющий выполнить стандартизацию
from sklearn.preprocessing import StandardScaler
# импортируем класс OneHotEncoder,
# позволяющий выполнить дамми-кодирование
from sklearn.preprocessing import OneHotEncoder
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression
# импортируем функцию roc_auc_score() для вычисления AUC-ROC
from sklearn.metrics import roc_auc_score
# импортируем класс ColumnTransformer, позволяющий выполнять
# преобразования для отдельных типов столбцов
from sklearn.compose import ColumnTransformer
# импортируем класс FunctionTransformer, позволяющий
# задавать пользовательские функции
from sklearn.preprocessing import FunctionTransformer
# импортируем класс Pipeline, позволяющий создавать конвейеры
from sklearn.pipeline import Pipeline
# импортируем класс GridSearchCV, позволяющий
# выполнить поиск по сетке
from sklearn.model_selection import GridSearchCV
  
```

Поскольку ранее мы уже работали с этим набором данных, когда строили базовую модель, и уже знаем, какие преобразования, не использующие вы-

числения по всему набору, можно применить до разбиения на обучающую и тестовую выборки (или до цикла перекрестной проверки), пишем функцию предварительной подготовки данных. Предварительная подготовка некоторых переменных (замена пропусков в переменной *NumberOfDependents*, замена значений переменной *MonthlyIncome* ниже 1200\$, замена значений переменной *RevolvingUtilizationOf-UnsecuredLines* выше определенного порога на пропуски, обработка выбросов в переменной *DebtRatio*, биннинг переменной *DebtRatio*) в этой функции будет отсутствовать, поскольку для этих переменных мы попробуем разные способы обработки.

```
# пишем функцию предварительной подготовки
def preprocessing(df):

    # значения переменной age меньше 18 заменяем
    # минимально допустимым значением возраста
    df['age'] = np.where(df['age'] < 18, 18, df['age'])

    # создаем переменную Ratio - отношение количества
    # просрочек 90+ к общему количеству просрочек
    sum_of_delinq = (df['NumberOfTimes90DaysLate'] +
                      df['NumberOfTime30-59DaysPastDueNotWorse'] +
                      df['NumberOfTime60-89DaysPastDueNotWorse'])

    cond = (df['NumberOfTimes90DaysLate'] == 0) | (sum_of_delinq == 0)
    df['Ratio'] = np.where(
        cond, 0, df['NumberOfTimes90DaysLate'] / sum_of_delinq)

    # создаем индикатор нулевых значений переменной
    # NumberOfOpenCreditLinesAndLoans
    df['NumberOfOpenCreditLinesAndLoans_is_0'] = np.where(
        df['NumberOfOpenCreditLinesAndLoans'] == 0, 'T', 'F')

    # создаем индикатор нулевых значений переменной
    # NumberRealEstateLoansOrLines
    df['NumberRealEstateLoansOrLines_is_0'] = np.where(
        df['NumberRealEstateLoansOrLines'] == 0, 'T', 'F')

    # создаем индикатор нулевых значений переменной
    # RevolvingUtilizationOfUnsecuredLines
    df['RevolvingUtilizationOfUnsecuredLines_is_0'] = np.where(
        df['RevolvingUtilizationOfUnsecuredLines'] == 0, 'T', 'F')

    # преобразовываем переменные в категориальные, применив
    # биннинг и перевод в единый строковый формат
    for col in ['NumberOfTime30-59DaysPastDueNotWorse',
                'NumberOfTime60-89DaysPastDueNotWorse',
                'NumberOfTimes90DaysLate']:
        df.loc[df[col] > 3, col] = 4
        df[col] = df[col].apply(lambda x: f"cat_{x}")

    # создаем список списков - список 2-факторных взаимодействий
    lst = [
        ['NumberOfDependents',
         'NumberOfTime30-59DaysPastDueNotWorse'],
```

```

[ 'NumberOfTime60-89DaysPastDueNotWorse',
  'NumberOfTimes90DaysLate'],
[ 'NumberOfTime30-59DaysPastDueNotWorse',
  'NumberOfTime60-89DaysPastDueNotWorse'],
[ 'NumberRealEstateLoansOrLines_is_0',
  'NumberOfTimes90DaysLate'],
[ 'NumberOfOpenCreditLinesAndLoans_is_0',
  'NumberOfTimes90DaysLate']
]

# создаем взаимодействия
for i in lst:
    f1 = i[0]
    f2 = i[1]
    df[f1 + ' + ' + f2 + '_interact'] = (df[f1].astype(str) + ' + '
                                           + df[f2].astype(str))

# укрупняем редкие категории
interact_columns = df.columns[df.columns.str.contains('interact')]
for col in interact_columns:
    df.loc[df[col].value_counts()[df[col]].values < 55, col] = 'Other'

return df

```

Применяем нашу функцию.

```
# применяем нашу функцию
data = preprocessing(data)
```

Выполняем разбиение на обучающую и тестовую выборки.

```
# создаем обучающий массив признаков, обучающий массив меток,
# тестовый массив признаков, тестовый массив меток
train, test, y_train, y_test = train_test_split(
    data.drop('SeriousDlqin2yrs', axis=1),
    data['SeriousDlqin2yrs'],
    test_size=.3,
    stratify=data['SeriousDlqin2yrs'],
    random_state=100)
```

Теперь напишем собственные классы. Для наглядности мы приведем их прямо здесь (они будут даны в тетрадке к этому разделу), однако лучшей практикой является перенос классов в отдельный модуль предварительной подготовки с последующим импортом.

Начнем с класса `NumberOfDependentsReplacer`, который будет заменять пропуски в переменной `NumberOfDependents` определенным константным значением. При построении базовой модели мы заменяли пропуски в переменной `NumberOfDependents` нулем, однако можно попробовать и другие значения.

```
# создаем собственный класс NumberOfDependentsReplacer, который
# заменяет пропуски переменной NumberOfDependents
# на определенное константное значение
class NumberOfDependentsReplacer(BaseEstimator, TransformerMixin):
```

```
"""
Параметры:
    threshold: пороговое значение
    replace_value: значение,
    на которое заменяем
"""
def __init__(self, replace_value=0):
    self.replace_value = replace_value

def fit(self, X, y=None):
    return self

def transform(self, X):
    X_replaced = np.where(X.isnull(), self.replace_value, X)
    return X_replaced
```

Пишем класс `MonthlyIncomeReplacer`, который заменяет пропуски и значения переменной `MonthlyIncome` ниже заданного порога на определенное константное значение. При построении базовой модели мы заменяли пропуски и значения переменной `NumberOfDependents` ниже 1200\$ значением 1200, однако у нас может появиться внутренняя информация о том, что пропуски и значения дохода ниже 1200\$ нужно заменять не значением 1200, а, например, значением 25 000\$. Предположим, для клиентов с доходом 25 000\$ были выставлены неверные значения меньше 1200\$ или пропуски.

```
# создаем собственный класс MonthlyIncomeReplacer, который
# заменяет пропуски и значения переменной MonthlyIncome
# ниже заданного порога на определенное константное значение
class MonthlyIncomeReplacer(BaseEstimator, TransformerMixin):
    """
Параметры:
    threshold: пороговое значение
    replace_value: значение,
    на которое заменяем
"""
    def __init__(self, threshold=1200, replace_value=1200):
        self.threshold = threshold
        self.replace_value = replace_value

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_trans = np.where((X.isnull() | (X < self.threshold)),
                           self.replace_value, X)
        return X_trans
```

Пишем собственный класс `UtilizationThresholdSetter`, который заменяет значения переменной `RevolvingUtilizationOfUnsecuredLines` выше заданного порога на пропуски. При построении базовой модели мы заменяли на пропуски значения переменной `RevolvingUtilizationOfUnsecuredLines` выше 2, однако можно попробовать альтернативные варианты.

```
# создаем собственный класс UtilizationThresholdSetter, который
# заменяет значения переменной RevolvingUtilizationOfUnsecuredLines
# выше заданного порога на пропуски
class UtilizationThresholdSetter(BaseEstimator, TransformerMixin):
    """
    Параметры:
        threshold: пороговое значение
    """
    def __init__(self, threshold=2):
        self.threshold = threshold

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_trans = np.where(X > self.threshold, np.NaN, X)
        return X_trans
```

Пишем собственный класс `CustomDiscretizer`, выполняющий биннинг переменной `DebtRatio`.

```
# создаем собственный класс CustomDiscretizer,
# выполняющий биннинг переменной DebtRatio
class CustomDiscretizer(BaseEstimator, TransformerMixin):
    """
    Параметры:
        bins: список бинов.
    """
    def __init__(self, bins=np.arange(0, 1.05, 0.05)):
        self.bins = bins

    def fit(self, X, y=None):
        # fit опять бездельничает
        return self

    def transform(self, X):
        # transform выполняет всю работу: применяем преобразование
        # с помощью заданного значения параметра bins
        self.bins[0] = -1
        X_bin = np.digitize(X, self.bins).astype('object')
        return X_bin
```

Пишем класс `OutlierRemover`, выполняющий винзоризацию переменной `DebtRatio`.

```
# создаем собственный класс, выполняющий винзоризацию
class OutlierRemover(BaseEstimator, TransformerMixin):
    """
    Параметры:
        lower_quantile: float, по умолчанию 0
            Нижний квантиль.
        upper_quantile: float, по умолчанию 0.75
            Верхний квантиль.
        k: float, по умолчанию 1.5
            Коэффициент.
```

```
copy: bool, по умолчанию True
    Возвращает копию.
"""

def __init__(self, copy=True, lower_quantile=0,
             upper_quantile=0.75, k=1.5):
    # все параметры для инициализации публичных атрибутов
    # должны быть заданы в методе __init__

    # публичные атрибуты
    self.copy = copy
    self.lower_quantile = lower_quantile
    self.upper_quantile = upper_quantile
    self.k = k

def __is_numpy(self, X):
    # частный метод, который с помощью функции isinstance()
    # проверяет, является ли наш объект массивом NumPy
    return isinstance(X, np.ndarray)

def fit(self, X, y=None):
    # fit должен принимать в качестве аргументов X и y

    # обучение модели осуществляется прямо здесь
    # создаем пустой словарь, в котором ключами
    # будут имена / целые числа, а значениями - кортежи
    self._dict = {}

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # записываем результат __is_numpy()
    is_np = self.__is_numpy(X)

    # если объект - массив NumPy, выполняем следующие действия:
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            lower = np.quantile(X[:, col], self.lower_quantile)
            upper = np.quantile(X[:, col], self.upper_quantile)
            IQR = (upper - lower) * self.k
            self._dict[col] = (lower, upper, IQR)
    # в противном случае, т. е. если объект - датафрейм pandas,
    # выполняем следующие действия:
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # вычисляем и записываем в словарь
            lower = X[col].quantile(self.lower_quantile)
            upper = X[col].quantile(self.upper_quantile)
            IQR = (upper - lower) * self.k
            self._dict[col] = (lower, upper, IQR)
```

```

# fit возвращает self
return self

def transform(self, X):
    # transform принимает в качестве аргумента только X

    # выполняем копирование массива во избежание
    # предупреждения SettingWithCopyWarning
    # "A value is trying to be set on a copy of
    # a slice from a DataFrame (Происходит попытка изменить
    # значение в копии среза данных датафрейма)"
    if self.copy:
        X = X.copy()

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # записываем результатом __is_numpy__()
    is_np = self.__is_numpy__(X)

    # применяем преобразование к X
    # если объект - массив NumPy, выполняем следующие действия:
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # заменяем
            X[:, col] = np.where(
                X[:, col] < (self._dict[col][0] - self._dict[col][2]),
                self._dict[col][0] - self._dict[col][2],
                X[:, col])
            X[:, col] = np.where(
                X[:, col] >= (self._dict[col][1] + self._dict[col][2]),
                self._dict[col][1] + self._dict[col][2],
                X[:, col])

    # в противном случае, т. е. если объект - датафрейм pandas,
    # выполняем следующие действия:
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # заменяем
            X[col] = np.where(
                X[col] < (self._dict[col][0] - self._dict[col][2]),
                self._dict[col][0] - self._dict[col][2],
                X[col])
            X[col] = np.where(
                X[col] >= (self._dict[col][1] + self._dict[col][2]),
                self._dict[col][1] + self._dict[col][2],
                X[col])

    # transform возвращает X
    return X

```

Создаем списки признаков:

- список категориальных признаков;
- список количественных признаков, кроме признаков *NumberOfDependents*, *MonthlyIncome*, *DebtRatio* и *RevolvingUtilization-OfUnsecuredLines*;
- список с количественным признаком *NumberOfDependents*;
- список с количественным признаком *MonthlyIncome*;
- список с количественным признаком *DebtRatio*;
- список с количественным признаком *RevolvingUtilizationOfUnsecuredLines*.

```
# создаем список категориальных переменных
cat_columns = train.dtypes[train.dtypes == 'object'].index.tolist()
# создаем список количественных переменных
num_columns = train.dtypes[train.dtypes != 'object'].index.tolist()
# создаем список с переменной NumberOfDependents
numberofdependents = ['NumberOfDependents']
# создаем список с переменной MonthlyIncome
income = ['MonthlyIncome']
# создаем список с переменной DebtRatio
debratio = ['DebtRatio']
# создаем список с переменной RevolvingUtilizationOfUnsecuredLines
utilization = ['RevolvingUtilizationOfUnsecuredLines']
# удаляем из списка количественных переменных
# переменные NumberOfDependents, MonthlyIncome,
# DebtRatio и RevolvingUtilizationOfUnsecuredLines
num_columns = list(set(num_columns).difference(
    set(numberofdependents + income + debtratio + utilization)))
```

Создадим конвейеры для наших списков, передаем в *ColumnTransformer* и формируем итоговый конвейер.

```
# создаем конвейер для количественных переменных
num_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# создаем конвейер для переменной NumberOfDependents
numberofdependents_pipe = Pipeline([
    ('trans', NumberOfDependentsReplacer()),
    ('scaler', StandardScaler())
])

# создаем конвейер для переменной MonthlyIncome
income_pipe = Pipeline([
    ('trans', MonthlyIncomeReplacer()),
    ('imp', SimpleImputer(strategy='median')),
    ('log', FunctionTransformer(np.log, validate=False)),
    ('scaler', StandardScaler())
])

# создаем конвейер для переменной DebtRatio
debtratio_pipe = Pipeline([
    ('outl', OutlierRemover()),
    ('scaler', StandardScaler())
])
```

```

# создаем конвейер для переменной DebtRatio
debtratio_pipe2 = Pipeline([
    ('outl', OutlierRemover()),
    ('binn', CustomDiscretizer()),
    ('ohe', OneHotEncoder(sparse=False,
                          handle_unknown='ignore'))
])
# создаем конвейер для переменной RevolvingUtilizationOfUnsecuredLines
utilization_pipe = Pipeline([
    ('trans', UtilizationThresholdSetter()),
    ('imp', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])
# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([('ohe', OneHotEncoder(sparse=False,
                                             handle_unknown='ignore'))])

# создаем список трехэлементных кортежей, в котором первый
# элемент кортежа - название конвейера с преобразованиями
transformers = [('num', num_pipe, num_columns),
                ('numberofdependents', numberofdependents_pipe,
                 numberofdependents),
                ('income', income_pipe, income),
                ('utilization', utilization_pipe, utilization),
                ('debtratio', debtratio_pipe, debtratio),
                ('debtratio2', debtratio_pipe2, debtratio),
                ('cat', cat_pipe, cat_columns)]]

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)

# задаем итоговый конвейер
pipe = Pipeline([('tf', transformer),
                 ('logreg', LogisticRegression(C=0.03,
                                              solver='liblinear',
                                              random_state=42))])

```

Создаем сетку значений гиперпараметров. По сути, здесь мы перебираем разные способы обработки конкретных признаков.

```

# задаем сетку гиперпараметров
param_grid = {'tf_utilization_trans_threshold': [1.5, 1.75, 2],
              'tf_numberofdependents_trans_replace_value': [0, 1, 2, 3],
              'tf_debtratio2_binn_bins': [np.arange(0, 1.05, 0.05),
                                         np.arange(0, 1.05, 0.1)],
              'tf_income_trans_replace_value': [25000, 30000, 35000],
              'tf_debtratio_outl_upper_quantile': [0.75, 0.8, 0.85]
             }

```

Создаем экземпляр класса GridSearchCV.

```
# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество
# блоков перекрестной проверки

gs = GridSearchCV(pipe,
                  param_grid,
                  scoring='roc_auc',
                  cv=5)
```

Запускаем поиск по сетке.

```
# выполняем поиск по сетке
gs.fit(train, y_train)
# смотрим наилучшие значения гиперпараметров
print('Наилучшие значения гиперпараметров: {}'.format(gs.best_params_))
# смотрим наилучшее значение AUC-ROC
print('Наилучшее значение AUC-ROC: {:.3f}'.format(gs.best_score_))
# смотрим значение AUC-ROC на тестовой выборке
print('AUC-ROC на тестовом наборе: {:.3f}'.format(
    roc_auc_score(y_test, gs.predict_proba(test)[:, 1])))
```

Наилучшие значения гиперпараметров: {'tf_debtratio2_binn_bins': array([0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]), 'tf_debtratio_outl_upper_quantile': 0.8, 'tf_income_trans_replace_value': 35000, 'tf_numberofdependents_trans_replace_value': 3, 'tf_utilization_trans_threshold': 2}

Наилучшее значение AUC-ROC: 0.863

AUC-ROC на тестовом наборе: 0.864

Заново прочитываем все исторические данные.

```
# записываем CSV-файл в объект DataFrame
fulldata = pd.read_csv('Data/cs-training.csv', index_col='Unnamed: 0')
```

Применяем функцию предварительной подготовки.

```
# применяем функцию предварительной обработки
# ко всем историческим данным
fulldata = preprocessing(fulldata)
```

Создаем массив меток и массив признаков.

```
# создаем массив меток и массив признаков
y_fulldata = fulldata.pop('SeriousDlqin2yrs')
```

С помощью атрибута `best_params_` извлекаем наилучшие значения гиперпараметров, присваиваем итоговому конвейеру и обучаем его на всех исторических данных.

```
# записываем оптимальные значения гиперпараметров
best_params = gs.best_params_
# присваиваем итоговому конвейеру оптимальные
# значения гиперпараметров
pipe.set_params(**best_params)
# обучаем итоговый конвейер с оптимальными значениями
# гиперпараметров на всех исторических данных
```

```
pipe.fit(fulldata, y_fulldata)
# смотрим значение AUC-ROC
print('AUC-ROC на всей исторической выборке: {:.3f}'.format(
    roc_auc_score(y_fulldata, pipe.predict_proba(fulldata)[:, 1])))
```

AUC-ROC на всей исторической выборке: 0.864

Загружаем новые данные.

```
# записываем CSV-файл, содержащий новые данные,
# в объект DataFrame
newdata = pd.read_csv('Data/cs-test.csv', index_col=0)
# записываем идентификатор набора новых данных
test_id = newdata.index
```

Применяем функцию предварительной подготовки.

```
# выполняем предварительную обработку
# новых данных
newdata = preprocessor(newdata)
```

При помощи итогового конвейера с оптимальными значениями гиперпараметров, обученного на всей исторической выборке, вычисляем вероятности для новых данных и формируем посылку на Kaggle.

```
# при помощи итогового конвейера с оптимальными значениями
# гиперпараметров, обученного на всей исторической выборке,
# вычисляем вероятности для новых данных
prob = pipe.predict_proba(newdata)
# выведем вероятности для первых 5 наблюдений
prob[:5]

array([0.05477489, 0.05055286, 0.01712939, 0.09484548, 0.10469977])

# формируем посылку
pd.DataFrame({'Id': test_id, 'Probability': prob}).to_csv(
    'Data/subm_giveme.csv', index=False)
```

Часть 4

Метрики для оценки качества модели

1. Бинарная классификация

1.1. Отрицательный и положительный классы, порог отсечения

При решении бизнес-задач, сводящихся к бинарной классификации, все наблюдения делят на два класса: класс с отрицательными исходами (первый уровень зависимой переменной) и класс с положительными исходами (второй уровень зависимой переменной). Обычно положительный класс обозначает наступление какого-то важного для бизнеса события (оттока, отклика, дефолта) и является интересующим нас классом. Допустим, важным событием является отток, поэтому положительным (интересующим) классом станет класс «Уходит» (соответствует ушедшим клиентам), а отрицательным классом – класс «Остается» (соответствует оставшимся клиентам).

Разбиение на два спрогнозированных класса получают с помощью варьирования порога отсечения – порогового значения спрогнозированной вероятности положительного класса, меняющейся в интервале от 0 до 1. По умолчанию используется пороговое значение 0,5. Если вероятность положительного класса больше 0,5, то прогнозируется положительный класс; если она меньше этого порогового значения, прогнозируется отрицательный класс. Возьмем наблюдение, для которого были спрогнозированы вероятности классов 0,44 и 0,56. Нашим положительным классом является класс *Уходит*. В данном случае вероятность класса *Уходит* равна 0,56 и превышает пороговое значение 0,5, поэтому прогнозируется класс *Уходит*.

1.2. МАТРИЦА ОШИБОК

На основе фактической и спрогнозированной принадлежности наблюдений к отрицательному или положительному классу возможны четыре типа случаев.

TP (*True Positives*) – верно классифицированные положительные примеры, или истинно положительные случаи. Пример истинно положительного случая – ушедший клиент верно классифицирован как ушедший. TN (*True Negatives*) – верно классифицированные отрицательные примеры, или истинно отрицательные случаи. Пример истинно отрицательного случая – оставшийся клиент верно классифицирован как оставшийся. FN (*False Negatives*) – положительные примеры, неверно классифицированные как отрицательные (ошибка I рода). Это так называемый «ложный пропуск», когда интересующее нас событие ошибочно не обнаруживается (ложноотрицательные случаи). Пример ложноотрицательного случая – ушедший клиент ошибочно классифицирован как оставшийся.

FP (*False Positives*) – отрицательные примеры, неверно классифицированные как положительные (ошибка II рода). Это «ложная тревога», когда при отсутствии события ошибочно выносится решение о его присутствии (ложноположительные случаи). Пример ложноположительного случая – оставшийся клиент ошибочно классифицирован как ушедший.

Эти четыре типа случаев образуют следующую матрицу ошибок.

	фактический отрицательный класс	спрогнозированный отрицательный класс
фактический положительный класс	TN	FP
	FN	TP

Рис. 1 Матрица ошибок

Давайте импортируем необходимые библиотеки, функцию `confusion_matrix()`, строящую матрицу ошибок.

```
# импортируем необходимые библиотеки
import pandas as pd
import numpy as np
# из модуля sklearn.metrics импортируем
# функцию confusion_matrix()
from sklearn.metrics import confusion_matrix
```

Загрузим данные, которые состоят из трех столбцов: столбца фактических значений зависимой переменной (фактические классы 0 и 1 для задачи оттока, где класс 0 соответствует классу *Остается*, а класс 1 – классу *Уходит*), столбца спрогнозированных значений зависимой переменной (спрогнозированные классы 0 и 1 для задачи оттока), столбца с вычисленными вероятностями класса 1 зависимой переменной, – и построим матрицу ошибок.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/results.csv', sep=';')
data.head()
```

	fact	predict	probability
0	1	1	0.556452
1	1	1	0.556452
2	0	1	0.556452
3	1	1	0.921147
4	1	1	0.921147

```
# вычисляем матрицу ошибок
confusion = confusion_matrix(data['fact'],
                             data['predict'])
# печатаем матрицу ошибок
print("Матрица ошибок:\n{}".format(confusion))
```

Матрица ошибок:

```
[[464 282]
 [ 73 525]]
```



Рис. 2 Матрица ошибок с цветовым выделением

В итоге получаем матрицу ошибок. Строки – это фактические классы зависимой переменной, столбцы – спрогнозированные классы зависимой переменной.

На пересечении строки и столбца с одинаковым именем/индексом записывают количество правильно классифицированных наблюдений (для удобства выделены в таблице классификации зеленым цветом). В нашем случае на пересечении строки 0 и столбца 0 записано количество верно классифицированных оставшихся клиентов. На пересечении строки 1 и столбца 1 записано количество верно классифицированных ушедших клиентов.

На пересечении строки и столбца с разными именами/индексами записывают количество неправильно классифицированных наблюдений (для удобства выделены в таблице классификации красным цветом). В нашем случае на пересечении строки 0 и столбца 1 записано количество неверно классифицированных оставшихся клиентов (они классифицированы как ушедшие). На пересечении строки 1 и столбца 0 записано количество неверно классифицированных ушедших клиентов (они классифицированы как оставшиеся).

Представим нашу матрицу ошибок в более наглядном виде.

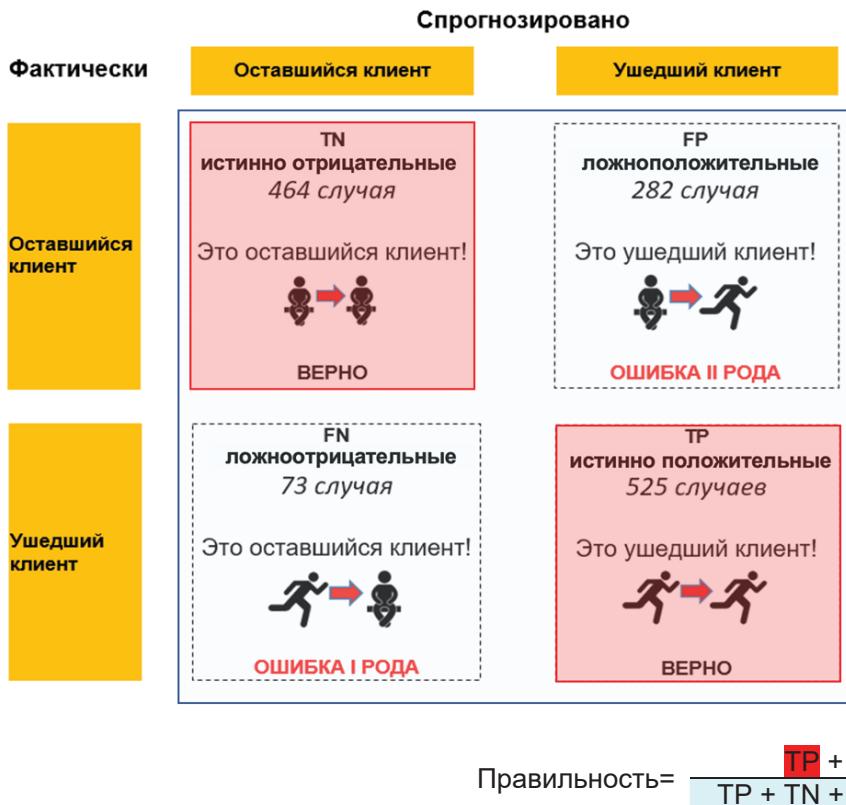


Рис. 3 Матрица ошибок с инфографикой

1.3. Доля ПРАВИЛЬНЫХ ОТВЕТОВ, ПРАВИЛЬНОСТЬ (ACCURACY)

В первую очередь мы можем вычислить *правильность* (accuracy). Ее выражают в виде следующей формулы:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} = \frac{525 + 464}{525 + 464 + 282 + 73} = 0,74.$$



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

Рис. 4 Правильность

Правильность – это количество правильно классифицированных случаев ($\text{TP} + \text{TN}$, показано на рисунке в виде областей красного цвета), поделенное на общее количество случаев ($\text{TP} + \text{TN} + \text{FP} + \text{FN}$, показано на рисунке в виде области синего цвета). Правильность еще называют долей правильных ответов.

Правильность характеризует дискриминирующую способность бинарного классификатора (способность модели отличать отрицательный класс от положительного) при конкретном пороге отсечения.

Давайте вычислим для нашего примера с оттоком правильность.

```
# из модуля sklearn.metrics импортируем
# функцию accuracy_score()
from sklearn.metrics import accuracy_score
# вычисляем правильность
acc_score = accuracy_score(data['fact'],
                           data['predict'])
# печатаем значение правильности
print("Правильность: {:.3f}".format(acc_score))
```

Правильность: 0.736

Следует помнить, что у правильности есть серьезный недостаток, она не может служить достоверной метрикой качества при работе с несбалансированными наборами данных.

Представьте, что вам требуется выяснить отклик клиентов на маркетинговое предложение. У вас есть набор данных, в котором 13 411 наблюдений соответствуют ситуации «не откликнулся» и 1812 наблюдений – «откликнулся». Другими словами, 88 % примеров относятся к классу «не откликнулся». Такие наборы данных, в которых один класс встречается гораздо чаще, чем остальные, часто называют *несбалансированными наборами данных* (*imbalanced datasets*), или *наборами данных с несбалансированными классами* (*datasets with imbalanced classes*). Теперь предположим, что вы построили модель дерева решений и получили следующую таблицу классификации.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 13411	FP 0
	Откликнулся	FN 1812	TP 0

Рис. 5 Пример несбалансированного набора данных

В итоге получаем правильность, равную 88 %, или $(13\ 411 + 0) / (13\ 411 + 0 + 1812 + 0)$, просто всегда прогнозируя класс «не откликнулся». Правильность по классу «Не откликнулся» у нас будет равна 100 %, или $13\ 411 / (13\ 411 + 0)$, а правильность по классу «Откликнулся» будет равна 0 %, или $0 / (0 + 1812)$. Таким образом, получаем высокое значение правильности при нулевом качестве прогнозирования класса «Откликнулся».

1.4. Чувствительность (SENSITIVITY)

Чувствительность – это количество истинно положительных случаев (TP, показано на рисунке в виде области красного цвета), поделенное на общее количество положительных случаев в выборке (TP + FN, показано на рисунке в виде области синего цвета). Она измеряется по формуле:

$$Se = TPR = \frac{TP}{TP + FN} = \frac{525}{525 + 73} = 0,88.$$

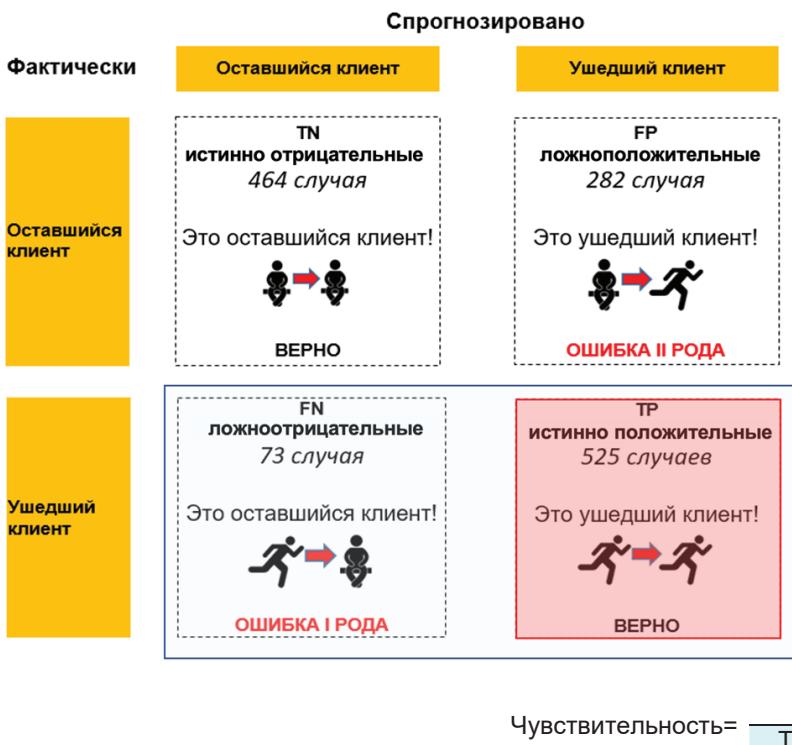


Рис. 6 Чувствительность

В нашем примере чувствительность – это способность модели правильно определять ушедших клиентов. Чувствительность – это правильность классификации для класса *Уходит*. Модель с высокой чувствительностью максимизирует долю правильно классифицированных ушедших клиентов. Чувствительность минимизирует вероятность совершения ошибки I рода, при этом увеличивая вероятность совершения ошибки II рода.

Повышая чувствительность, мы минимизируем риск классифицировать ушедшего клиента как оставшегося, но при этом увеличиваем риск классифицировать оставшегося клиента как ушедшего. Образно говоря, увеличивая чувствительность, повышаем «пессимизм» модели. Наша модель демонстрирует высокую чувствительность. Финансовые риски высокочувствительной модели заключаются в том, что мы можем впустую затратить средства на удержание какой-то части лояльных клиентов, ошибочно классифицировав как клиентов, склонных к оттоку.

Обратите внимание: у чувствительности много синонимов. Чувствительность еще называют *полнотой* (*recall*), *процентом результативных ответов* или *хит-рейтом* (*hit rate*).

Давайте вычислим для нашего примера с оттоком чувствительность.

```
# вычисляем tn, fp, fn, tp
tn, fp, fn, tp = confusion_matrix(
    data['fact'], data['predict']).ravel()
# вычисляем чувствительность
sensitivity = tp / (tp + fn)
```

```
# печатаем значение чувствительности
print("Чувствительность: {:.3f}".format(sensitivity))
```

Чувствительность: 0.878

1.5. СПЕЦИФИЧНОСТЬ (SPECIFICITY)

Специфичность – это количество истинно отрицательных случаев (TN, показано на рисунке в виде области красного цвета), поделенное на общее количество отрицательных случаев в выборке (TN + FP, показано на рисунке в виде области синего цвета). Она измеряется по формуле:

$$Sp = TNR = \frac{TN}{TN + FP} = \frac{464}{464 + 282} = 0,62.$$

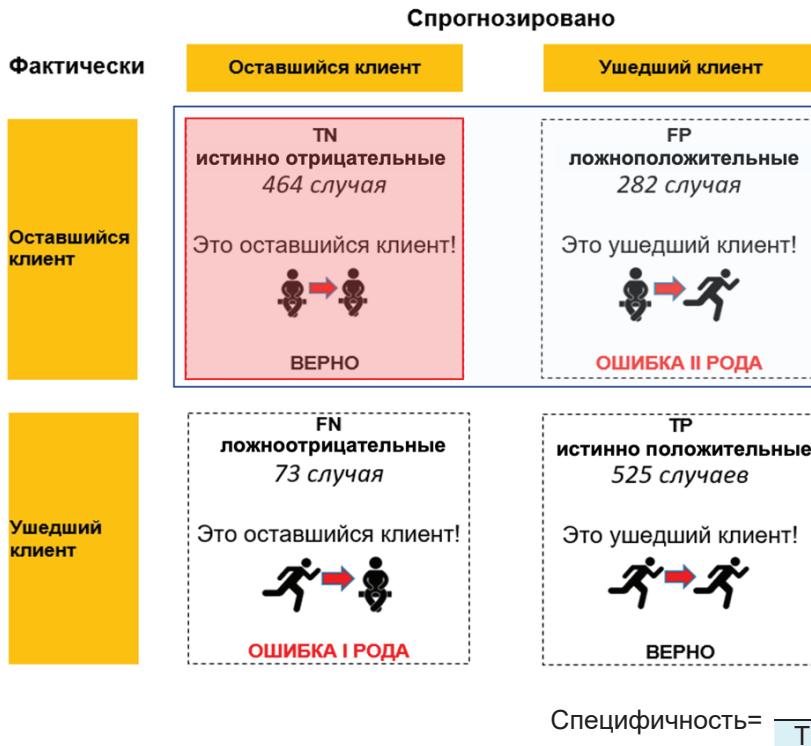


Рис. 7 Специфичность

В нашем примере специфичность – это способность модели правильно определять оставшихся клиентов. Специфичность – это правильность классификации для класса *Остается*. Модель с высокой специфичностью максимизирует долю правильно классифицированных оставшихся клиентов. Специфичность минимизирует вероятность совершения ошибки II рода, при этом увеличивая вероятность совершения ошибки I рода.

Повышая специфичность, мы минимизируем риск классифицировать оставшегося клиента как ушедшего, но при этом увеличиваем риск классифицировать ушедшего клиента как оставшегося. Образно говоря, увеличивая специфичность, повышаем «оптимизм» модели. Наша модель показывает высокий уровень специфичности. Финансовые риски высокоспецифичной модели заключаются в том, что мы можем потерять какую-то часть клиентов, которые собирались покинуть компанию, а мы их ошибочно классифицировали как лояльных.

Давайте вычислим для нашего примера с оттоком специфичность.

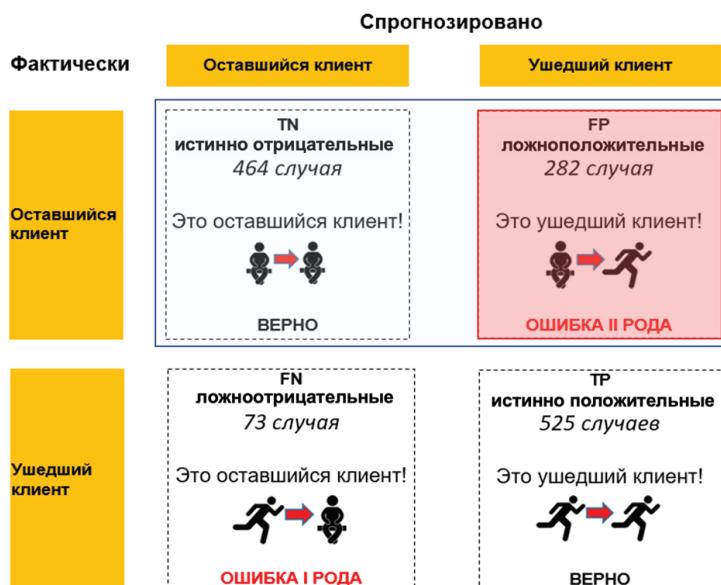
```
# вычисляем специфичность
specificity = tn / (tn + fp)
# печатаем значение специфичности
print("Специфичность: {:.3f}".format(specificity))

Специфичность: 0.622
```

1.6. 1 – СПЕЦИФИЧНОСТЬ (1 – SPECIFICITY)

Еще один показатель, который нам пригодится в будущем, – это 1 – специфичность (единица минус специфичность) – количество ложноположительных случаев (FP, показано на рисунке в виде области красного цвета), поделенное на общее количество отрицательных случаев в выборке (FP + TN, показано на рисунке в виде области синего цвета), вычисляется по формуле:

$$FPR = 1 - Sp = \frac{FP}{FP + TN} = 1 - 0,62 = \frac{282}{282 + 464} = 0,38.$$



$$\text{1 – специфичность} = \frac{FP}{FP + TN}$$

Рис. 8 1 – специфичность

В нашем примере 1 – специфичность характеризует уровень «ложных срабатываний» модели, когда оставшийся клиент классифицируется как ушедший. Давайте вычислим для нашего примера с оттоком 1 – специфичность.

```
# вычисляем 1 - специфичность
one_minus_specificity = fp / (fp + tn)
# печатаем значение 1 - специфичности
print("1 - специфичность: {:.3f}".format(
    one_minus_specificity))
```

1 - специфичность: 0.378

1.7. СБАЛАНСИРОВАННАЯ ПРАВИЛЬНОСТЬ

Ранее мы говорили о том, что легко получить высокую правильность, всегда предсказывая мажоритарный класс. Похожие недостатки имеют чувствительность и специфичность. Легко получить специфичность 100 %, отнеся все наблюдения к отрицательному классу, при этом будет чувствительность 0 %, и наоборот, если отнести все наблюдения к положительному классу, то будет специфичность 0 % и чувствительность 100 %. Часто используют сбалансированную правильность, которая вычисляется как среднее чувствительности и специфичности, в условиях дисбаланса классов она будет полезнее обычной правильности.

$$BA = \frac{TPR + TNR}{2} = \frac{0,88 + 0,62}{2} = 0,75.$$

Теперь вычислим сбалансированную правильность для несбалансированного набора данных, рассмотренного ранее.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 13411	FP 0
	Откликнулся	FN 1812	TP 0

Рис. 9 Пример несбалансированного набора данных

Видим, что она ниже обычной правильности.

$$BA = \frac{TPR + TN}{2} = \frac{0 + 1}{2} = 0,5.$$

Если классы распределены поровну, то сбалансированная правильность будет примерно равна обычной правильности.

Давайте вычислим для нашего примера с оттоком сбалансированную правильность.

```
# из модуля sklearn.metrics импортируем
# функцию balanced_accuracy_score()
from sklearn.metrics import balanced_accuracy_score
# вычисляем сбалансированную правильность
bal_acc = balanced_accuracy_score(data['fact'],
                                    data['predict'])
# печатаем значение сбалансированной правильности
print("Сбалансированная правильность: {:.3f}".format(bal_acc))
```

Сбалансированная правильность: 0.750

1.8. Точность (PRECISION)

Точность – это количество истинно положительных случаев (TP, показано на рисунке в виде области красного цвета), поделенное на общее количество предсказанных положительных случаев (TP + FP, показано на рисунке в виде области синего цвета). Она измеряется по формуле:

$$P = \frac{TP}{TP + FP} = \frac{525}{525 + 282} = 0,65.$$

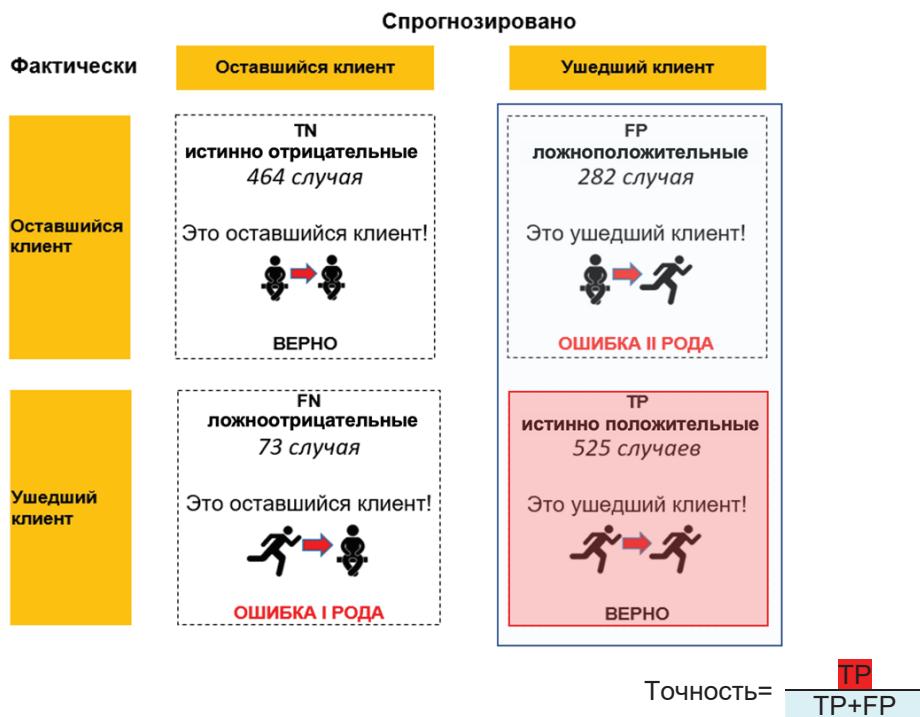


Рис. 10 Точность

Давайте вычислим для нашего примера с оттоком точность.

```
# вычисляем точность
precision = tp / (tp + fp)
# печатаем значение точности
print("Точность: {:.3f}".format(precision))
```

Точность: 0.651

1.9. СРАВНЕНИЕ ТОЧНОСТИ И ЧУВСТВИТЕЛЬНОСТИ (ПЛНОТЫ)

Точность и чувствительность (полнота) похожи тем, что обе метрики интересуют истинно положительные случаи, однако задача точности – определить долю таких случаев в выборке предсказанных положительных случаев, а задачи полноты – определить долю таких случаев в выборке фактических положительных случаев.

Точность важнее, когда вам нужно уменьшить количество ложноположительных случаев, увеличив количество ложноотрицательных случаев. Смысл в том, что ложноположительные случаи имеют более высокую цену ошибки, чем ложноотрицательные случаи.

Для простоты возьмем более образный пример из фантастического фильма. Пусть в условиях зомби-апокалипсиса отрицательным случаем будет человек, превратившийся в зомби, а положительным случаем – здоровый человек. Вы, конечно, постараетесь переместить в безопасную зону максимально возможное количество здоровых людей из общей массы, но вы ведь не хотите ошибочно поместить зомби в безопасную зону. Поэтому вы стараетесь уменьшить число ложноположительных случаев (когда зомби ошибочно были приняты за здоровых и попали в безопасную зону), увеличив количество ложноотрицательных случаев (когда здоровые люди ошибочно приняты за зомби и не попали в безопасную зону). Если по примерным расчетам мы знаем, что у нас 1000 здоровых и 300 зомби, при этом 950 фактически здоровых находятся в безопасной зоне, а 50 фактически здоровых находятся вне зоны, цена ложноположительного случая будет выше. Ниже приводится матрица ошибок для нашего фантастического примера.

TN	FP
300	0
FN	TP
50	950

Рис. 11 Матрица ошибок для фантастического примера

Допустим, у нас есть прибор, определяющий, является ли человек зомби. Если прибор имеет идеальную точность, то каждый раз, когда он сообщает «пациент здоров», мы можем доверять ему: действительно пациент здоров. Однако точность не дает никакой информации о том, можем ли мы доверять ему, когда он сообщает «пациент – зомби». Мы должны выяснить, сколько здоровых наш прибор ошибочно принял за зомби (отрицательный класс).

К примеру, у нас есть набор данных из 205 человек, 100 зомби и 105 здоровых. Наш прибор классифицирует пять пациентов как здоровых (и эти пять пациентов действительно здоровы), а всех остальных относит к зомби (отрицательному классу).

TN 100	FP 0
FN 100	TP 5

Рис. 12 Матрица ошибок при идеальной точности

Вычисляем точность и полноту. Точность будет идеальной, а полнота – очень низкой.

$$P = \frac{TP}{TP + FP} = \frac{5}{5 + 0} = 1$$

$$Se = \frac{TP}{TP + FN} = \frac{5}{5 + 100} = 0,048.$$

Пять здоровых человек попадут в безопасную зону, а 100 здоровых человек будут обречены.

Теперь представим: наш глупый прибор ВСЕГДА утверждает, что «пациент здоров».

TN 0	FP 100
FN 0	TP 105

Рис. 13 Матрица ошибок при идеальной полноте

Мы получаем идеальную полноту!

$$Se = \frac{TP}{TP + FN} = \frac{105}{105 + 0} = 1.$$

Должны ли мы заключить, что это идеальное устройство? Нет, мы должны обратиться к точности. Вычисляем точность.

$$P = \frac{TP}{TP + FP} = \frac{105}{105 + 100} = 0,51.$$

205 человек попадут в безопасную зону, из которых 105 будут здоровыми, а 100 окажутся зомби.

1.10. F-МЕРА (F-SCORE, ИЛИ F-MEASURE)

Хотя точность и полнота являются очень важными метриками, сами по себе они не дадут вам полной картины, особенно в условиях дисбаланса классов. Одним из способов подытожить их является F-мера (F-measure), которая представляет собой гармоническое среднее точности и полноты, то есть обратное значение от среднего значения обратных значений точности и полноты:

$$F = \left(\frac{\text{точность}^{-1} + \text{полнота}^{-1}}{2} \right)^{-1} = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}}.$$

Здесь стоит отметить, что точность и полнота являются «естественными» обратными величинами в том смысле, что у них одинаковый числитель (количество истинно положительных случаев), а знаменатели различны.

Рассмотренный вариант вычисления F-меры называют сбалансированной F-мерой, поскольку она придает одинаковый вес точности и полноте, или F_1 -мерой (в нижнем индексе указываем величину β).

В более общем виде F-меру записывают с помощью следующей формулы:

$$F_\beta = \left(1 + \beta^2 \right) \cdot \frac{\text{точность} \times \text{полнота}}{(\beta^2 \cdot \text{точность}) + \text{полнота}}.$$

Чаще всего величина β принимает три значения:

- 1, точность и полнота имеют одинаковый вес ($F = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}}$);
- 0,5, точность имеет вес, в 2 раза больший веса полноты
($F_{0,5} = 1,25 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}}$);
- 2, полноте нужно присвоить вес, в 2 раза больший веса точности ($F_2 = 5 \times \frac{\text{точность} \times \text{полнота}}{4 \times \text{точность} + \text{полнота}}$).

Вернемся к F_1 -мере. Поскольку F_1 -мера учитывает точность и полноту, то для бинарной классификации несбалансированных данных она может быть лучшей метрикой, чем правильность. Для нашего примера с оттоком она будет равна

$$F_1 = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}} = 2 \times \frac{0,65 \times 0,88}{0,65 + 0,88} = 2 \times \frac{0,572}{1,53} = 0,747.$$

```
# из модуля sklearn.metrics импортируем функцию f1_score()
from sklearn.metrics import f1_score
# вычисляем F1-меру
f1 = f1_score(data['fact'], data['predict'])
# печатаем F1-меру
print("F1-мера: {:.3f}".format(f1))
```

F1-мера: 0.747

Теперь вычислим F_1 -меру для нашего несбалансированного набора данных.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 13 411	FP 0
	Откликнулся	FN 1812	TP 0

Рис. 14 Пример несбалансированного набора данных

Вспомним, что в этом примере мы получаем правильность, равную 88 %, или $(13\ 411 + 0) / (13\ 411 + 0 + 1812 + 0)$, просто всегда прогнозируя класс «не откликнулся». Правильность по классу *Не откликнулся* у нас будет равна 100 %, или $13\ 411 / (13\ 411 + 0)$, а правильность по классу *Откликнулся* (т. е. полнота) будет равна 0 %, или $0 / (0 + 1812)$. Точность будет также равна 0 %, или $0 / (0 + 0)$. Таким образом, получаем высокое значение правильности при нулевом качестве прогнозирования класса *Откликнулся* (при нулевой полноте) и нулевой точности. И вот здесь как раз объективную картину даст F_1 -мера. Нетрудно догадаться, что она будет равна 0. F_1 -мера действительно дает более лучшее представление о качестве модели, чем правильность в условиях дисбаланса классов.

Вместе с тем недостаток F_1 -меры заключается в том, что в отличие от правильности ее труднее интерпретировать. Кроме того, F_1 -мера не принимает во внимание истинно отрицательные случаи.

А теперь вычислим F_1 -меру для одного из примеров с зомби. У нас есть набор данных из 205 человек, 100 зомби и 105 здоровых. Отрицательным случаем будет человек, превратившийся в зомби, а положительным случаем – здоровый человек. Наш прибор классифицирует пять пациентов как здоровых (и эти пять пациентов действительно здоровы), а всех остальных относит к зомби (отрицательному классу).

TN 100	FP 0
FN 100	TP 5

Рис. 15 Матрица ошибок при идеальной точности

Мы помним, что точность будет идеальной, а полнота – очень низкой. Пять здоровых человек попадут в безопасную зону, а 100 здоровых человек будут обречены.

$$P = \frac{TP}{TP + FP} = \frac{5}{5 + 0} = 1; \quad Se = \frac{TP}{TP + FN} = \frac{5}{5 + 100} = 0,048.$$

Пять здоровых человек попадут в безопасную зону, а 100 здоровых человек будут обречены.

F_1 -мера будет равна 0,05.

$$F_1 = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}} = 2 \times \frac{1 \times 0,048}{1 + 0,048} = 2 \times \frac{0,048}{1,048} = 0,09.$$

А какой результат получился бы, если бы мы использовали вместо гармонического среднего арифметическое среднее? Тогда F_1 -мера была бы неправдоподобно высокой.

$$F_1 = \frac{\text{точность} + \text{полнота}}{2} = \frac{1 + 0,048}{2} = 0,524.$$

Здесь проявляется важное свойство гармонического среднего. Когда значения точности и полноты значительно отличаются друг от друга, то гармоническое среднее, в отличие от арифметического среднего, будет ближе к минимальному значению этих двух чисел.

Ниже приведены 3D-визуализации арифметического среднего и гармонического среднего.

```
# импортируем библиотеку matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
# включаем режим 'retina', если у вас экран Retina
%config InlineBackend.figure_format = 'retina'
# импортируем конструктор Axes3D
from mpl_toolkits.mplot3d import Axes3D

# подготавливаем значения осей для построения 3D-поверхности:
# пятьдесят одна точка от 0.01 до 102 с шагом 2,
# для избежания ошибки деления на ноль мы задали
# ненулевое начальное значение
x = np.arange(0.01, 102, 2)
y = np.arange(0.01, 102, 2)

# meshgrid позволяет создать сетку координат
# для построения 3D-поверхности
X, Y = np.meshgrid(x, y)

# ось Z - арифметическое среднее X и Y
Z = (X + Y) / 2

# создаем рисунок, задав его размер
fig = plt.figure(figsize=(10, 7))

# график будет отображаться как subplot
ax = fig.add_subplot(111, projection='3d')

# вызываем функцию построения поверхности, используя
# градиентную заливку (параметр cmap)
ax.plot_surface(X, Y, Z, cmap='Spectral_r')

# задаем названия осей
ax.set(xlabel='x', ylabel='y')
```

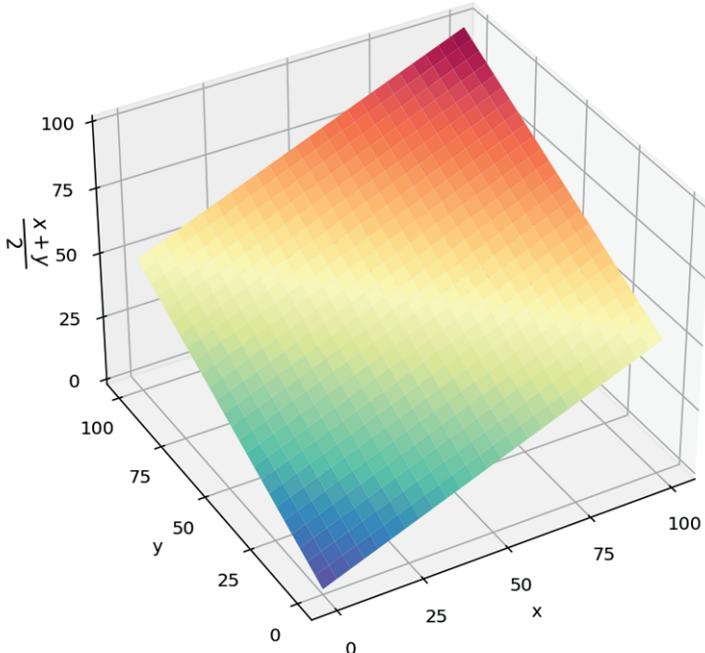
```
# в случае оси Z используем возможности LaTeX для получения
# красивого отображения формулы функции, корректируем размер
# шрифта с помощью параметра fontsize
ax.set_zlabel(r'$\frac{x + y}{2}$', fontsize=16)

# задаем значения делений осей
ticks = [0, 25, 50, 75, 100]
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_zticks(ticks)

# задаем начальное положение рисунка
# с помощью углов отображения
ax.view_init(35, 240)

# задаем заголовок
ax.set_title("Арифметическое среднее");
```

Арифметическое среднее



```
# теперь ось Z - гармоническое среднее
Z = 1 / ((1 / X + 1 / Y) / 2)

# создаем еще один рисунок для отображения
# 3D-поверхности второй функции
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='Spectral_r')
```

```

# задаем названия осей
ax.set(xlabel='x', ylabel='y')

# в случае оси Z используем возможности LaTeX для получения
# красивого отображения формулы функции, корректируем размер
# шрифта с помощью параметра fontsize
ax.set_zlabel(r'$\frac{1}{(1/x + 1/y)/2}$', fontsize=16)

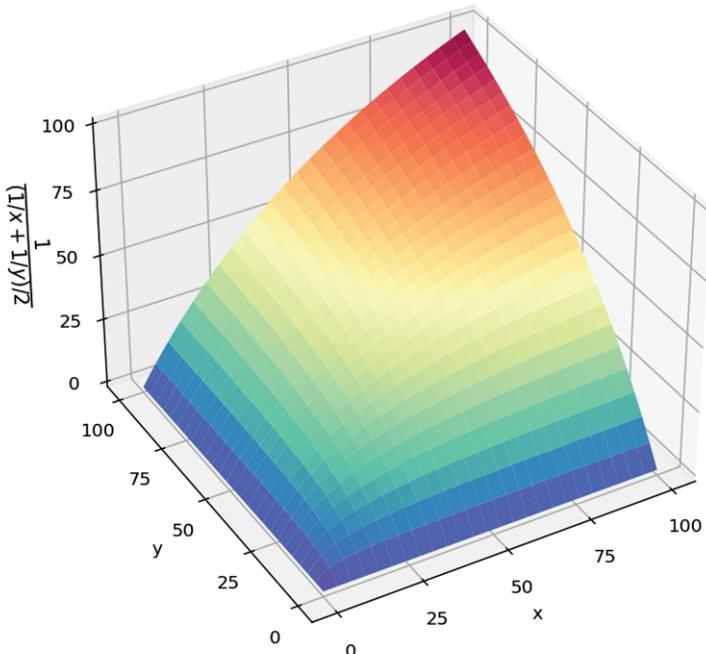
# задаем значения делений осей
ticks = [0, 25, 50, 75, 100]
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_zticks(ticks)

# задаем начальное положение рисунка
# с помощью углов отображения
ax.view_init(35, 240)

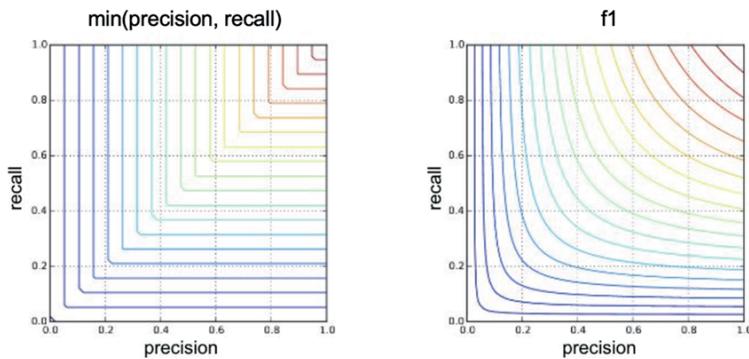
# задаем заголовок
ax.set_title("Гармоническое среднее");

```

Гармоническое среднее



На рисунке ниже приводятся линии уровня некоторых функций от двух переменных.

Рис. 16 Линии уровня функций $\min(\text{precision}, \text{recall})$ и f_1

Видно, что линии уровня гармонического среднего сильно похожи на «уголки», т. е. на линии функции $\min()$, это вынуждает при максимизации функционала сильнее «тянуть вверх» меньшее значение. Если, например, точность очень мала, то увеличение полноты даже в два раза не сильно поменяет значение F-меры.

1.11. ВАРЬИРОВАНИЕ ПОРОГА ОТСЕЧЕНИЯ

Вернемся к нашей матрице ошибок. Вспомним, что получаемая матрица ошибок будет зависеть от выбранного порога отсечения. По умолчанию выводится матрица ошибок для порога отсечения 0,5.

		Спрогнозировано	
		Оставшийся клиент	Ушедший клиент
Фактически	Оставшийся клиент	TN истинно отрицательные 464 случая	FP ложноположительные 282 случая
	Ушедший клиент	Это оставшийся клиент!  ВЕРНО	Это ушедший клиент!  ОШИБКА II РОДА
Ушедший клиент	Оставшийся клиент	FN ложноотрицательные 73 случая	TP истинно положительные 525 случаев
	Ушедший клиент	Это оставшийся клиент!  ОШИБКА I РОДА	Это ушедший клиент!  ВЕРНО

Рис. 17 Матрица ошибок для порога отсечения 0,5

Меняя пороговое значение, с которым сравнивается спрогнозированная вероятность положительного класса, мы будем получать разные результаты классификации и, соответственно, разные значения правильности, чувствительности, специфичности, 1 – специфичности, точности и F-меры.

Давайте снизим пороговое значение до 0,3. Теперь класс *Уходит* прогнозируется, когда вероятность класса *Уходит* превышает пороговое значение 0,3.



Рис. 18 Матрица ошибок для порога отсечения 0,3

Снизив пороговое значение, повышаем чувствительность – способность модели правильно классифицировать ушедших клиентов. При этом специфичность (способность модели правильно классифицировать оставшихся клиентов) у нас снизится. Давайте проверим это утверждение.

Вычисляем чувствительность и специфичность.

$$Se = TPR = \frac{TP}{TP + FN} = \frac{549}{549 + 49} = 0,92.$$

$$Sp = TNR = \frac{T}{TN + FP} = \frac{413}{413 + 333} = 0,55.$$

Действительно, по сравнению с исходной моделью, использующей порог отсечения 0,5, чувствительность повысилась (с 0,88 до 0,92), а специфичность снизилась (с 0,62 до 0,55).

```

# задаем порог
mythreshold = 0.3
# получаем прогнозы согласно новому порогу
predictions = (data['probability'] >= mythreshold).astype(int)
# вычисляем матрицу ошибок
confusion = confusion_matrix(data['fact'], predictions)
# вычисляем чувствительность: TP / (TP + FN)
se = confusion[1][1] / (confusion[1][1] + confusion[1][0])
# вычисляем специфичность: TN / (TN + FP)
sp = confusion[0][0] / (confusion[0][0] + confusion[0][1])
# печатаем матрицу ошибок
print(f"Матрица ошибок для порога {mythreshold}:\n{confusion}")
# печатаем значение чувствительности
print(f"Чувствительность: {se:.3f}")
# печатаем значение специфичности
print(f"Специфичность: {sp:.3f}")

```

Матрица ошибок для порога 0.3:

[[413 333]

[49 549]]

Чувствительность: 0.918

Специфичность: 0.554

А сейчас повысим пороговое значение до 0,7. Теперь класс Уходит прогнозируется, когда вероятность класса Уходит превышает пороговое значение 0,7.



Рис. 19 Матрица ошибок для порога отсечения 0,7

Увеличив пороговое значение, снижаем чувствительность – способность модели правильно классифицировать ушедших клиентов. При этом специфичность (способность модели правильно классифицировать оставшихся клиентов) у нас повысится.

Вычисляем чувствительность и специфичность.

$$Se = TPR = \frac{TP}{TP + FN} = \frac{106}{106 + 492} = 0,18.$$

$$Sp = TNR = \frac{TN}{TN + FP} = \frac{729}{729 + 17} = 0,98.$$

Действительно, по сравнению с исходной моделью, использующей порог отсечения 0,5, чувствительность снизилась (с 0,88 до 0,18), а специфичность повысилась (с 0,62 до 0,98).

```
# задаем порог
mythreshold = 0.7
# получаем прогнозы согласно новому порогу
predictions = (data['probability'] >= mythreshold).astype(int)
# вычисляем матрицу ошибок
confusion = confusion_matrix(data['fact'], predictions)
# вычисляем чувствительность: TP / (TP + FN)
se = confusion[1][1] / (confusion[1][1] + confusion[1][0])
# вычисляем специфичность: TN / (TN + FP)
sp = confusion[0][0] / (confusion[0][0] + confusion[0][1])
# печатаем матрицу ошибок
print(f"Матрица ошибок для порога {mythreshold}:\n{confusion}")
# печатаем значение чувствительности
print(f"Чувствительность: {se:.3f}")
# печатаем значение специфичности
print(f"Специфичность: {sp:.3f}")
```

Матрица ошибок для порога 0.7:

```
[[729 17]
 [492 106]]
```

Чувствительность: 0.177

Специфичность: 0.977

По итогам этих экспериментов становится легко понять, что мы можем получить чувствительность 100 %, просто установив порог 0 и приписав всех клиентов к положительному классу – ушедшим клиентам (таким образом, чувствительность имеет наименьший оптимальный порог отсечения), или специфичность 100 %, просто установив порог 1 и приписав всех клиентов к отрицательному классу – оставшимся клиентам (таким образом, специфичность имеет наибольший оптимальный порог отсечения).

```
# задаем порог
mythreshold = 0
# получаем прогнозы согласно новому порогу
predictions = (data['probability'] >= mythreshold).astype(int)
# вычисляем матрицу ошибок
confusion = confusion_matrix(data['fact'], predictions)
```

```
# вычисляем чувствительность: TP / (TP + FN)
se = confusion[1][1] / (confusion[1][1] + confusion[1][0])
# вычисляем специфичность: TN / (TN + FP)
sp = confusion[0][0] / (confusion[0][0] + confusion[0][1])
# печатаем матрицу ошибок
print(f"Матрица ошибок для порога {mythreshold}:\n{confusion}")
# печатаем значение чувствительности
print(f"Чувствительность: {se:.3f}")
# печатаем значение специфичности
print(f"Специфичность: {sp:.3f}")
```

Матрица ошибок для порога 0:

```
[[ 0 746]
 [ 0 598]]
```

Чувствительность: 1.000

Специфичность: 0.000

```
# задаем порог
mythreshold = 1
# получаем прогнозы согласно новому порогу
predictions = (data['probability'] >= mythreshold).astype(int)
# вычисляем матрицу ошибок
confusion = confusion_matrix(data['fact'], predictions)
# вычисляем чувствительность: TP / (TP + FN)
se = confusion[1][1] / (confusion[1][1] + confusion[1][0])
# вычисляем специфичность: TN / (TN + FP)
sp = confusion[0][0] / (confusion[0][0] + confusion[0][1])
# печатаем матрицу ошибок
print(f"Матрица ошибок для порога {mythreshold}:\n{confusion}")
# печатаем значение чувствительности
print(f"Чувствительность: {se:.3f}")
# печатаем значение специфичности
print(f"Специфичность: {sp:.3f}")
```

Матрица ошибок для порога 1:

```
[[746  0]
 [598  0]]
```

Чувствительность: 0.000

Специфичность: 1.000

1.12. Коэффициент Мэттьюса (MATTHEWS CORRELATION COEFFICIENT или MCC)

Коэффициент корреляции Мэттьюса (Matthews correlation coefficient, или MCC) используется в машинном обучении в качестве критерия качества бинарной классификации. Он был предложен биохимиком Брайаном Мэттьюсом в 1975 году. Коэффициент учитывает истинно отрицательные, истинно положительные, ложноотрицательные и ложноположительные случаи и обычно рассматривается в качестве сбалансированной метрики, которую можно использовать в условиях дисбаланса классов. MCC, по сути, является коэффициентом корреляции между фактической и спрогнозированной бинарной классификациями. Он возвращает значение от -1 до +1. Значение +1 соответствует идеальной согласованности между фактическими и спрогнозированными значениями.

зированными значениями, значение 0 означает, что прогноз не лучше случайного угадывания, а значение –1 указывает на полную рассогласованность между фактическими и спрогнозированными значениями. MCC также известен как ϕ -коэффициент. MCC связан со статистикой хи-квадрат для таблицы сопряженности 2×2 :

$$|MCC| = \sqrt{\frac{\chi^2}{n}},$$

где n – общее количество наблюдений.

MCC можно напрямую вычислить на основе матрицы ошибок по формуле:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$

Давайте вычислим для нашего примера с оттоком коэффициент Мэттьюса.

```
# из модуля sklearn.metrics импортируем
# функцию matthews_corrcoef()
from sklearn.metrics import matthews_corrcoef
# вычисляем коэффициент Мэттьюса
matt_corr = matthews_corrcoef(data['fact'], data['predict'])
# печатаем коэффициент Мэттьюса
print("Коэффициент Мэттьюса: {:.3f}".format(matt_corr))
```

Коэффициент Мэттьюса: 0.507

Теперь вычислим коэффициент Мэттьюса для нашего несбалансированного набора.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 13411	FP 0
	Откликнулся	FN 1812	TP 0

Рис. 20 Пример несбалансированного набора данных

В вышеприведенном примере MCC не определен (поскольку FP и TP будут равны 0, знаменатель равен 0). В scikit-learn обычно в случае нулевых значений показателей, участвующих в вычислении коэффициента, используют очень маленькое положительное значение, поэтому получим значение, близкое к нулю.

Дэвид Чикко в своей статье «Ten quick tips for machine learning in computational biology» («Десять быстрых советов для машинного обучения в вычислительной биологии») высказал мнение, что для оценки качества бинарной

классификации коэффициент корреляции Мэттьюса более информативен, чем другие показатели матрицы ошибок (такие как F_1 -мера и правильность), поскольку он принимает во внимание взаимоотношение между четырьмя компонентами матрицы ошибок (истинно отрицательными, истинно положительными, ложноотрицательными и ложноположительными случаями).

Чикко рассуждает следующим образом. У вас есть очень несбалансированный тестовый набор, состоящий из 100 наблюдений, 95 из которых являются наблюдениями положительного класса и лишь пять являются наблюдениями отрицательного класса. Вы допустили некоторые ошибки при разработке и обучении модели, и теперь у вас есть алгоритм, который всегда предсказывает положительный класс. Представим, что вы не знаете об этой проблеме.

Применяя свою модель к несбалансированному тестовому набору, вы получаете следующие значения для матрицы ошибок: $TP = 95$, $FP = 5$, $TN = 0$, $FN = 0$.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 0	FP 5
	Откликнулся	FN 0	TP 95

Рис. 21 Матрица ошибок для первого примера Дэвида Чикко

Эти значения дадут правильность 95 % и F_1 -меру 97,44 %. Взглянув на эти чрезмерно оптимистичные оценки, вы будете очень счастливы и будете думать, что ваш алгоритм машинного обучения отлично работает. Очевидно, что вы ошибаетесь.

Напротив, чтобы избежать этих опасных вводящих в заблуждение иллюзий, существует еще один показатель качества, который вы можете использовать: коэффициент корреляции Мэттьюса.

Принимая во внимание долю каждого класса в матрице ошибок, коэффициент корреляции Мэттьюса будет высоким только в том случае, если ваша модель хорошо классифицирует как наблюдения отрицательного класса, так и наблюдения положительного класса.

В вышеприведенном примере MCC не определен (поскольку TN и FN будут равны 0, знаменатель равен 0). Взглянув на это значение вместо правильности или F_1 -меры, вы сможете заметить, что ваш классификатор движется в неправильном направлении, и вы поймете, что есть проблемы, которые нужно решить.

Теперь, продолжает свои рассуждения Чикко, рассмотрим другой пример. Вы выполнили классификацию для того же самого набора данных и получили следующие значения для матрицы ошибок: $TP = 90$, $FP = 4$, $TN = 1$, $FN = 5$.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 1	FP 4
	Откликнулся	FN 5	TP 90

Рис. 22 Матрица ошибок для второго примера Дэвида Чикко

В этом примере классификатор хорошо классифицирует наблюдения положительного класса, но не смог правильно распознать наблюдения отрицательного класса. Вновь F_1 -мера и правильность будут чрезвычайно высокими: правильность 91 %, а F_1 -мера 95,24 %. Аналогично предыдущему случаю, если бы исследователь анализировал только эти два показателя, не рассматривая MCC, он ошибочно полагал бы, что модель выполняет свои задачи достаточно хорошо, и у него была бы иллюзия успеха.

Проверка коэффициента корреляции Мэттьюса вновь будет иметь решающее значение. В этом примере значение MCC будет равно 0,14, а это указывает на то, что алгоритм работает аналогично случайному угадыванию. Действуя подобно сигналу тревоги, MCC сможет проинформировать аналитика о том, что модель работает плохо.

Обратите внимание, что F_1 -мера зависит от того, какой класс определен в качестве положительного. В первом примере значение F_1 -меры высоко, потому что положительным классом был мажоритарный класс. Инвертирование положительного и отрицательного классов приведет к следующей матрице ошибок: TP = 0, FP = 0, TN = 5, FN = 95.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 5	FP 0
	Откликнулся	FN 95	TP 0

Рис. 23 Матрица ошибок для первого примера Дэвида Чикко после инвертирования меток классов

Это даст F_1 -меру 0 %.

По этим причинам, резюмирует Чикко, мы настоятельно рекомендуем оценивать качество модели с помощью коэффициента корреляции Мэттьюса (MCC) вместо правильности и F_1 -меры для любой задачи бинарной классификации.

Таким образом, мы выяснили, что МСС не зависит от того, какой класс является положительным, и это дает ему преимущество перед F1-мерой, когда имеет место неправильное определение положительного класса.

1.13. Каппа Коэна (Cohen's kappa)

Каппа Коэна измеряет согласованность между двумя экспертами, классифицирующими n объектов по s взаимоисключающим категориям. В качестве экспертов выступают фактические и спрогнозированные метки классов зависимой переменной. Ее часто используют вместо правильности в случае несбалансированных классов:

$$\kappa = \frac{p_o - p_e}{1 - p_e},$$

где

p_o – относительная наблюдаемая согласованность (идентична правильности);

p_1 – ожидаемая вероятность случайной согласованности.

Если между экспертами наблюдается полная согласованность, то $\kappa = 1$. Если согласованность между двумя экспертами случайна, то $\kappa = 0$. Статистика может быть отрицательной, когда согласованность хуже случайной.

Вычислим каппу Коэна для нашего примера с оттоком.

		Эксперт B	
		Остается	Уходит
Эксперт A	Остается	TN 464	FP 282
	Уходит	FN 73	TP 525

Рис. 24 Матрица ошибок

Вычисляем наблюдаемую согласованность.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} = \frac{525 + 464}{525 + 464 + 282 + 73} = 0,74.$$

Вычисляем вероятность случайной согласованности.

Эксперт А (строки – фактические классы) отнес к классу *Остается* 746 наблюдений ($464 + 282$), к классу *Уходит* – 588 наблюдений ($73 + 525$). Таким образом, эксперт А отнес наблюдения к классу *Остается* в 55,5 % случаев ($746 / 1344$).

Эксперт В (столбцы – спрогнозированные классы) отнес к классу *Остается* 537 наблюдений ($464 + 73$), к классу *Уходит* – 807 наблюдений ($282 + 525$). Таким образом, эксперт В отнес наблюдения к классу *Остается* в 40 % случаев ($537 / 1344$).

Ожидаемая вероятность того, что оба эксперта отнесли наблюдения к классу *Остается* случайно, составляет

$$p_{\text{Остается}} = \frac{TN + FP}{TN + FP + FN + TP} \times \frac{TN + FN}{TN + FP + FN + TP} = 0,555 \times 0,40 = 0,22.$$

Ожидаемая вероятность того, что оба эксперта отнесли наблюдения к классу *Уходит* случайно, составляет

$$p_{\text{Уходит}} = \frac{FN + TP}{TN + FP + FN + TP} \times \frac{FP + TP}{TN + FP + FN + TP} = 0,445 \times 0,60 = 0,267.$$

Вычисляем общую вероятность случайной согласованности:

$$p_e = p_{\text{Остается}} + p_{\text{Уходит}} = 0,22 + 0,267 = 0,487.$$

Вычисляем каппу Коэна:

$$\kappa = \frac{p_o - p_e}{1 - p_e} = \frac{0,736 - 0,487}{1 - 0,487} = \frac{0,249}{0,513} = 0,48.$$

При $\kappa > 0,75$ согласованность считается высокой, при $0,4 < \kappa < 0,75$ – хорошей, в противном случае – плохой.

Давайте автоматически вычислим для нашего примера с оттоком каппу Коэна.

```
# из модуля sklearn.metrics импортируем
# функцию cohen_kappa_score()
from sklearn.metrics import cohen_kappa_score
# вычисляем каппу Коэна
cohen_kappa = cohen_kappa_score(data['fact'], data['predict'])
# печатаем коэффициент Каппы Коэна
print("Каппа Коэна: {:.3f}".format(cohen_kappa))
```

Каппа Коэна: 0.483

Теперь вычислим каппу Коэна для несбалансированного набора данных.

		Спрогнозированные классы	
		Не откликнулся	Откликнулся
Фактические классы	Не откликнулся	TN 13411	FP 0
	Откликнулся	FN 1812	TP 0

Рис. 25 Пример несбалансированного набора данных

Вычисляем наблюдаемую согласованность.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} = \frac{0 + 13411}{13411 + 0 + 1812 + 0} = 0,88.$$

Вычисляем вероятность случайной согласованности.

$$p_{\text{остается}} = \frac{TN + FP}{TN + FP + FN + TP} \times \frac{TN + FN}{TN + FP + FN + TP} = 0,88 \times 1 = 0,88.$$

$$p_{\text{выходит}} = \frac{FN + TP}{TN + FP + FN + TP} \times \frac{FP + TP}{TN + FP + FN + TP} = 0,12 \times 0 = 0.$$

Вычисляем общую вероятность случайной согласованности:

$$p_e = p_{\text{Ostrtsf}} + p_{\text{Uhottt}} = 0,88 + 0 = 0,88.$$

Вычисляем каппу Коэна:

$$\kappa = \frac{p_o - p_e}{1 - p_e} = \frac{0,88 - 0,88}{1 - 0,88} = \frac{0}{0,12} = 0.$$

Видим, что каппа Коэна, как и коэффициент Мэттьюса с F_1 -мерой, равна 0 и может использоваться для объективной оценки качества модели в случае дисбаланса классов.

1.14. ROC-КРИВАЯ (ROC CURVE) И ПЛОЩАДЬ под ROC-КРИВОЙ (AUC-ROC)

Оптимальная модель должна обладать 100%-ной чувствительностью и 100%-ной специфичностью, но, как мы выяснили выше, добиться этого, как правило, невозможно. Повысшая чувствительность, неизбежно снижаем специфичность, и наоборот, понижая чувствительность, неизбежно повышаем специфичность. Поэтому на практике строится ROC-кривая (англ. receiver operating characteristic – рабочая характеристика приёмника). Речь идет о кривой соотношений истинно положительных случаев (чувствительности) и ложноположительных случаев (1 – специфичности) для различных пороговых значений спрогнозированной вероятности интересующего класса. С ее помощью можно выбрать такой порог, который дает оптимальное соотношение чувствительности и 1 – специфичности. Вместо 1 – специфичности можно использовать специфичность. Понятие «оптимальности» зависит от того, какая бизнес-задача стоит перед моделером.

Кроме того, если правильность – это показатель дискриминирующей способности модели (способности модели отличать отрицательный класс от положительного) для конкретного порога отсечения, то ROC-кривая позволяет судить о дискриминирующей способности модели для разных порогов отсечения.

Сам термин «receiver operating characteristic» пришел из теории обработки сигналов времен Второй мировой войны. После атаки на Перл Харбор в 1941 году, когда самолеты японцев были сначала ошибочно приняты за стаю перелетных птиц, а потом за грузовой конвой транспортных самолетов,

перед инженерами-электротехниками и инженерами по радиолокации была поставлена задача увеличить точность распознавания вражеских объектов по радиолокационному сигналу.

2.5 Weighted Combination Criteria

Suppose it is possible to assign a certain number w as a weighting factor representing the importance of a false alarm relative to a hit. Since $P_{SN}(A)$ is the probability of a hit, and $P_N(A)$ the probability of a false alarm, it would then be reasonable to find a criterion A which maximizes the quantity

$$P_{SN}(A) - wP_N(A). \quad (9)$$

But this quantity can be written as

$$\int_A [f_{SN}(X) - wf_N(X)] dX \quad (10)$$

where the integration is taken over the sample points X listed in A . To maximize this integral, one would list in A every sample for which the integrand was not negative. Solving that inequality for w , one sees that A should contain those sample points X for which

$$\ell(X) = \frac{f_{SN}(X)}{f_N(X)} \geq w. \quad (11)$$

Thus the desired criterion A is simply $A(w)$, and so it is a ratio criterion.

2.6 Neyman-Pearson Criteria

If it is critically important to keep the probability of a false alarm $P_N(A)$ below a certain level k , then it would be reasonable to choose from among such criteria that one which maximizes the probability of a hit. Thus Neyman and Pearson proposed as a type of optimum criterion any criterion A_k for which

- (1) $P_N(A_k) \leq k$, and
- (2) $P_{SN}(A_k)$ is a maximum for all the criteria A with the property $P_N(A) \leq k$.

The A_k type criterion can also be expressed as a ratio criterion. This can be made plausible as follows. To begin with, it is necessary to consider only those criteria A for which $P_N(A) = k$, because A will be taken as large as possible in order to meet condition (2). Now consider the curve given parametrically by the equations

$$X = X(\beta) = P_N(A(\beta))$$

and

$$Y = Y(\beta) = P_{SN}(A(\beta)). \quad (12)$$

This curve will be called the Receiver Operating Characteristic (briefly, ROC) curve, for a receiver whose output is likelihood ratio and with which ratio criteria are being used.

The ROC curve passes through the points $(0, 0)$ and $(1, 1)$, the first at $\beta = \infty$, the second at $\beta = 0$. At $\beta = 0$, $\ell(X) \geq \beta = 0$ for all X , so $A(0)$ consists of all possible samples. Thus the observer will report that every sample is drawn from SN , so he will be certain to make a false alarm and to make a hit. (This assumes that the samples will not be drawn exclusively from one of the populations.)

Рис. 26 Одна из первых статей, посвященных ROC-кривой

Допустим, у нас есть 20 наблюдений, из которых 12 наблюдений относятся к отрицательному классу, а 8 наблюдений – к положительному классу. Отрицательный класс – «хорошие» заемщики, положительный класс – «плохие» заемщики. С помощью бинарного классификатора мы получили следующие спрогнозированные вероятности положительного класса (рисунок ниже).

№	фактический класс	спрогнозированная вероятность положительного класса
1	N	0,18
2	N	0,24
3	N	0,32
4	N	0,33
5	N	0,4
6	N	0,53
7	N	0,58
8	N	0,59
9	N	0,6
10	N	0,7
11	N	0,75
12	N	0,85
13	P	0,52
14	P	0,72
15	P	0,73
16	P	0,79
17	P	0,82
18	P	0,88
19	P	0,9
20	P	0,92

Рис. 27 Исходные спрогнозированные вероятности положительного класса

Построение ROC-кривой происходит следующим образом.

- Сначала сортируем все наблюдения по убыванию спрогнозированной вероятности положительного класса.
- Берем единичный квадрат на координатной плоскости. Значения оси абсцисс будут значениями 1 – специфичности (цена деления оси задается значением $1/\text{neg}$), а значения оси ординат будут значениями чувствительности (цена деления оси задается значением $1/\text{pos}$). При этом pos – это количество наблюдений положительного класса, а neg – количество наблюдений отрицательного класса.
- Задаем точку с координатами $(0, 0)$ и для каждого отсортированного наблюдения x :
 - если x принадлежит положительному классу, двигаемся на $1/\text{pos}$ вверх;
 - если x принадлежит отрицательному классу, двигаемся на $1/\text{neg}$ вправо.

Спрогнозированные вероятности положительного класса, отсортированные по убыванию

№	фактический класс	спрогнозированная вероятность положительного класса
20	P	0.92
19	P	0.9
18	P	0.88
12	N	0.85
17	P	0.82
16	P	0.79
11	N	0.75
15	P	0.73
14	P	0.72
10	N	0.7
9	N	0.6
8	N	0.59
7	N	0.58
6	N	0.53
13	P	0.52
5	N	0.4
4	N	0.33
3	N	0.32
2	N	0.24
1	N	0.18

Цена деления 1/8, поскольку у нас 8 наблюдений положительного класса

Построение ROC-кривой вручную

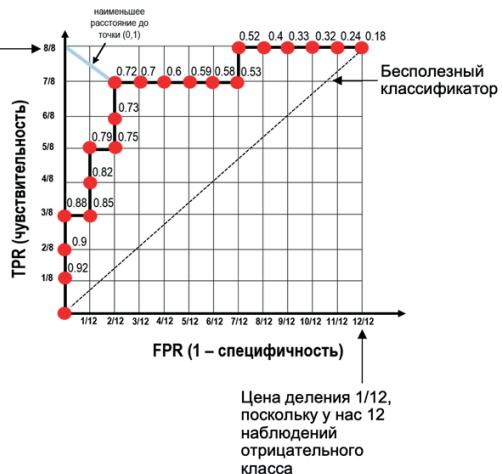


Рис. 28 Построение ROC-кривой вручную

Построив ROC-кривую, мы убеждаемся, что она действительно представляет собой кривую соотношений истинно положительных случаев (чувствительности) и ложноположительных случаев (1 – специфичности) для различных пороговых значений спрогнозированной вероятности интересующего класса. График часто дополняют диагональной линией, проведенной под углом 45 градусов из точки с координатами (0, 0) в точку с координатами (1, 1). Эта линия соответствует бесполезному классификатору, предсказывающему классы случайным образом.

Вместо 1 – специфичности можно отложить специфичность, но тогда произойдет инверсия шкалы: 12/12, 11/12, ..., 1/12, что не очень удобно для интерпретации.

Значение вероятности положительного класса, при котором ROC-кривая находится на минимальном расстоянии от верхнего левого угла – точки с координатами (0,1), дает наибольшую правильность классификации. В данном случае таким значением будет значение 0,72.

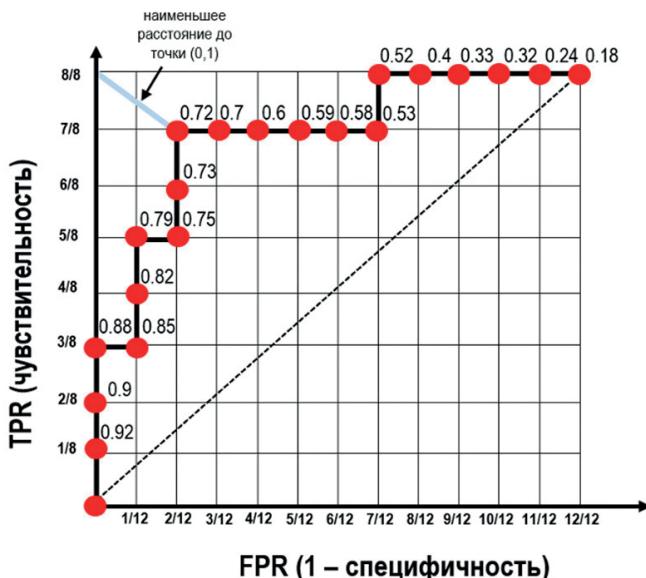


Рис. 29 Порог, находящийся на наименьшем расстоянии от точки (0,1), дает наилучшую правильность (accuracy)

По мере построения ROC-кривой для каждого значения вероятности положительного класса записываем соответствующие ей пары значений 1 – специфичности и чувствительности (координаты).

№	Фактический класс	Спрогнозированная вероятность положительного класса	1 – специфичность, или FPR (при данном пороге вероятности из 12 отрицательных наблюдений n наблюдений будут неверно классифицированы как положительные)	Чувствительность, или TPR (при данном пороге вероятности из 8 положительных наблюдений n наблюдений будут верно классифицированы как положительные)
20	P	0.92	0	1/8
19	P	0.9	0	2/8
18	P	0.88	0	3/8
12	N	0.85	1/12	3/8
17	P	0.82	1/12	4/8
16	P	0.79	1/12	5/8
11	N	0.75	2/12	5/8
15	P	0.73	2/12	6/8
14	P	0.72	2/12	7/8
10	N	0.7	3/12	7/8
9	N	0.6	4/12	7/8
8	N	0.59	5/12	7/8
7	N	0.58	6/12	7/8
6	N	0.53	7/12	7/8
13	P	0.52	7/12	8/8
5	N	0.4	8/12	8/8
4	N	0.33	9/12	8/8
3	N	0.32	10/12	8/8
2	N	0.24	11/12	8/8
1	N	0.18	12/12	8/8

Рис. 30 Отсортированные спрогнозированные вероятности положительного класса и соответствующие значения FPR и TPR

По рисунку выше видим, что при пороге 0,72 мы правильно классифицируем 83,3 % отрицательных (10 из 12 отрицательных, потому что согласно столбцу «FPR» только 2 из 12 отрицательных будут неверно классифицированы) и 87,5 % положительных (7 из 8 положительных согласно столбцу «TPR»). Таким образом, мы правильно классифицируем 17 человек из 20, т. е. правильность составляет 85 %. Убедились, что порог 0,72 является порогом, при котором мы получаем наибольшую правильность классификации.

На практике бывают ситуации, когда при упорядочении наблюдений по убыванию вероятности положительного класса два наблюдения, принадлежащих разным классам, получают одинаковые вероятности.

Для таких наблюдений построение ROC-кривой осуществляется иначе. Вообще говоря, здесь могут быть две стратегии.

Первая стратегия, которую называют «пессимистичной», заключается в том, чтобы поместить в начало такой последовательности сначала отрицательный пример, а затем положительный (двигаемся вправо и затем вверх, получаем нижний L-образный сегмент).

Вторая стратегия, которую называют «оптимистичной», заключается в том, чтобы поместить в начало такой последовательности сначала положительный пример, а затем отрицательный (двигаемся вверх и затем вправо, получаем верхний L-образный сегмент).

Компромиссная стратегия, применяющаяся на практике, заключается в усреднении пессимистичного и оптимистичного сегментов. Усреднением будет диагональ, проведенная в прямоугольнике, образованном этими двумя наблюдениями.

Отсортированные спрогнозированные вероятности положительного класса, двум наблюдениям разных классов присвоены одинаковые вероятности

Nº	фактический класс	спрогнозированная вероятность положительного класса
20	P	0.92
19	P	0.9
18	P	0.88
12	N	0.85
17	P	0.82
16	P	0.79
11	N	0.75
15	P	0.75
14	P	0.72
10	N	0.7
9	N	0.6
8	N	0.59
7	N	0.55
6	I	0.53
13	P	0.52
5	N	0.4
4	N	0.33
3	N	0.32
2	N	0.24
1	N	0.18

Построение ROC-кривой вручную в том случае, когда двум наблюдениям, относящимся к разным классам, присвоены одинаковые вероятности

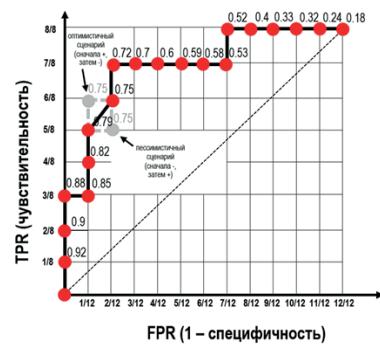


Рис. 31 Построение ROC-кривой вручную в том случае, когда двум наблюдениям, относящимся к разным классам, присвоены одинаковые вероятности

ROC-кривая является монотонной и неубывающей функцией. Когда по мере движения из нижнего левого угла в верхний правый угол пороговое значение уменьшается (т. е. уменьшается вероятность положительного класса, позволяющая отнести наблюдение к положительному классу), у нас происходит монотонное увеличение чувствительности и 1 – специфичности.

В идеале бы сначала встретить только наблюдения положительного класса, а потом только наблюдения отрицательного класса: из нижнего левого угла все время идти вверх, затем, добравшись до верхнего левого угла, все время идти вправо и достичь верхнего правого угла (идеальное ранжирование). Тогда получим Г-образную ROC-кривую (идеальный классификатор) и охватим максимальную площадь под ней.

Итак, при идеальном качестве модели график ROC-кривой проходит через верхний левый угол. В этом случае доля истинно положительных примеров составляет 100 %, а доля ложноположительных примеров равна 0 %. Поэтому чем ближе кривая к верхнему левому углу, тем выше дискриминирующая способность модели.

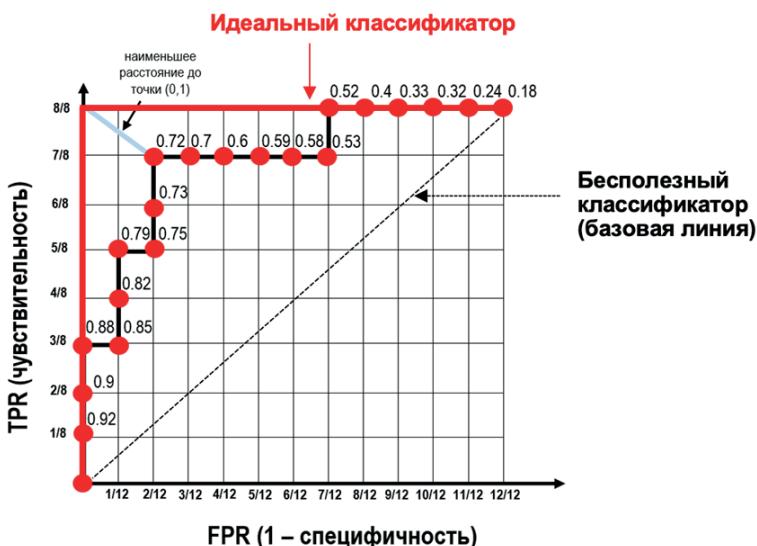


Рис. 32 ROC-кривая для идеального классификатора

Давайте для нашего игрушечного примера из 20 наблюдений построим ROC-кривую.

Сначала создаем массив из наших 20 фактических значений (классов) зависимой переменной.

```
# создаем массив фактических значений (классов)
# зависимой переменной
classes = пр.аггай([1, 1, 1, 0, 1, 1, 0, 1, 1, 0,
                     0, 0, 0, 0, 1, 0, 0, 0, 0, 0])
```

Создаем массив спрогнозированных вероятностей положительного класса.

```
# создаем массив спрогнозированных вероятностей
# положительного класса
пргфа = пр.аггай([0.92, 0.9, 0.88, 0.85, 0.82,
                    0.79, 0.75, 0.73, 0.72, 0.7,
                    0.6, 0.59, 0.58, 0.53, 0.52,
                    0.4, 0.33, 0.32, 0.24, 0.18])
```

Давайте напишем собственную функцию, которая построит ROC-кривую.

```
# пишем функцию, которая будет строить ROC-кривую
def _binary_clf_curve(y_true, y_score, sample_weight=None):
    """
    Вычисляет количество истинно положительных и
    ложноположительных для каждого порогового значения
    вероятности положительного класса; предполагается, что
    наблюдение положительного класса всегда имеет метку 1

    Параметры
    -----
    y_true : одномерный массив формы [n_samples]
        Фактические метки (классы) зависимой переменной.

    y_score : одномерный массив формы [n_samples]
        Спрогнозированные вероятности положительного класса.

    sample_weight : одномерный массив формы (n_samples,),
        по умолчанию None
        Веса наблюдений.

    Возвращает
    -----
    fps : одномерный массив
        Количество ложноположительных случаев (fps), индекс i
        фиксирует количество наблюдений отрицательного класса,
        которые получили оценку >= thresholds[i].
        Общее количество наблюдений отрицательного класса равно
        fps[-1] (таким образом, количество истинно отрицательных
        случаев определяется по формуле fps[-1] - fps).

    tps : одномерный массив
        Количество истинно положительных случаев (tps), индекс i
        фиксирует количество наблюдений положительного класса,
        которые получили оценку >= thresholds[i].
        Общее количество наблюдений положительного класса равно
        tps[-1] (таким образом, количество ложноотрицательных
        случаев определяется по формуле tps[-1] - tps).

    thresholds : одномерный массив
        Пороговые значения спрогнозированной вероятности
        положительного класса, отсортированные по убыванию.
    """

    # если для sample_weight задано не None
    if sample_weight is not None:
        # формируем булеву маску: True, если вес не равен 0,
        # False в противном случае
        nonzero_weight_mask = sample_weight != 0
        # с помощью булевой маски удаляем из массива
        # меток метки с нулевыми весами
        y_true = y_true[nonzero_weight_mask]
        # с помощью булевой маски удаляем из массива вероятностей
        # положительного класса вероятности с нулевыми весами
        y_score = y_score[nonzero_weight_mask]
```

```
# с помощью булевой маски удаляем
# из массива весов нулевые веса
sample_weight = sample_weight[nonzero_weight_mask]

# получаем индексы вероятностей положительного класса,
# отсортированных по убыванию
desc_score_indices = np.argsort(y_score)[::-1]
# сортируем вероятности с помощью индексов
y_score = y_score[desc_score_indices]
# сортируем метки с помощью индексов
y_true = y_true[desc_score_indices]

# если для sample_weight задано не None
if sample_weight is not None:
    # сортируем веса с помощью индексов
    weight = sample_weight[desc_score_indices]
# в противном случае
else:
    # все веса равны 1.0
    weight = 1.0

# функция np.diff() возвращает n-ю разность элементов массива
# (по умолчанию n = 1), которая также может быть вычислена
# вдоль указанной оси или осей, с помощью np.ufunc() получаем
# индексы порогов
distinct_indices = np.where(np.diff(y_score))[0]
# вычисляем индекс последнего порога
end = np.array([y_true.size - 1])
# добавляем этот индекс в конец массива с индексами порогов
threshold_indices = np.hstack((distinct_indices, end))

# получаем пороговые значения вероятности положительного класса
# и количество истинно положительных случаев
thresholds = y_score[threshold_indices]
tps = np.cumsum(y_true * weight)[threshold_indices]

# получаем количество ложноположительных случаев
if sample_weight is not None:
    # (1 - y_true) = количество ложноположительных
    # наблюдений в каждом индексе
    fps = np.cumsum((1 - y_true) * weight)[threshold_indices]
else:
    # (1 + threshold_indices) = количество положительных наблюдений
    # в каждом индексе, таким образом, количество положительных
    # наблюдений минус количество истинно положительных =
    # количество ложноположительных
    fps = (1 + threshold_indices) - tps

return fps, tps, thresholds
```

Применяем нашу функцию, получаем количество истинно положительных случаев и количество ложноположительных случаев для каждого порога вероятности положительного класса.

```
# применяем нашу функцию _binary_clf_curve()
fps, tps, thresholds = _binary_clf_curve(classes, proba)
print("пороги:\n{thresholds}\n")
print("количество ложноположительных:\n{fps}\n")
print("количество истинно положительных:\n{tps}\n")

пороги:
[0.92 0.9 0.88 0.85 0.82 0.79 0.75 0.73 0.72 0.7 0.6 0.59 0.58 0.53
 0.52 0.4 0.33 0.32 0.24 0.18]

количество ложноположительных:
[ 0.  0.  0.  1.  1.  2.  2.  2.  3.  4.  5.  6.  7.  7.  8.  9. 10.
 11. 12.]

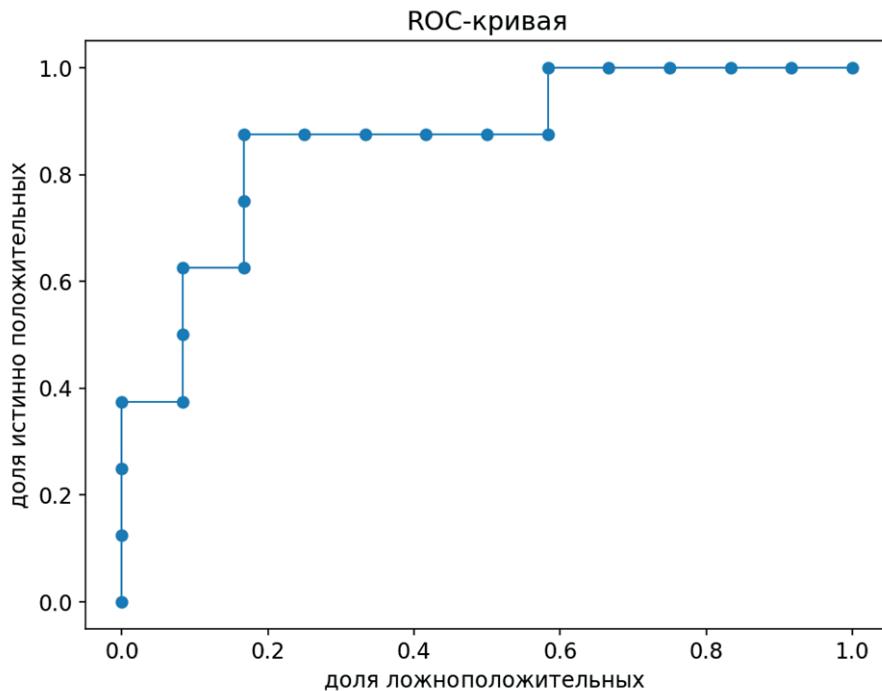
количество истинно положительных:
[1. 2. 3. 3. 4. 5. 5. 6. 7. 7. 7. 7. 7. 8. 8. 8. 8. 8.]
```

Давайте абсолютные частоты переведем в относительные и визуализируем результаты.

```
# преобразовываем количество в долю, добавляем 0
# к истинно положительным и ложноположительным,
# чтобы ROC-кривая брала начало в точке (0, 0)
fpr = np.hstack((0, fps / fps[-1]))
tpr = np.hstack((0, tps / tps[-1]))
print("доля ложноположительных:\n{fpr}\n")
print("доля истинно положительных:\n{tpr}\n")

# строим ROC-кривую
plt.rcParams['figure.figsize'] = 8, 6
plt.rcParams['font.size'] = 12

fig = plt.figure()
plt.plot(fpr, tpr, marker='o', lw=1)
plt.xlabel("доля ложноположительных")
plt.ylabel("доля истинно положительных")
plt.title("ROC-кривая")
plt.show()
```



Для более глубокого понимания подробнее рассмотрим все, что происходит под капотом нашей функции `_binary_clf_curve()`.

Сначала надо получить индексы вероятностей положительного класса, отсортированных по убыванию.

```
# получаем индексы вероятностей положительного класса,
# отсортированных по убыванию
desc_score_indices = np.argsort(proba)[::-1]
desc_score_indices

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

С помощью индексов сортируем вероятности и метки.

```
# сортируем вероятности с помощью индексов
y_score = proba[desc_score_indices]
y_score

array([ 0.92,  0.9 ,  0.88,  0.85,  0.82,  0.79,  0.75,  0.73,  0.72,  0.7 ,  0.6 ,
       0.59,  0.58,  0.53,  0.52,  0.4 ,  0.33,  0.32,  0.24,  0.18])

# сортируем метки с помощью индексов
y_true = classes[desc_score_indices]
y_true

array([1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0])
```

Теперь нам надо получить индексы порогов. Однако здесь есть сложность: в реальности у нас часто могут быть одинаковые вероятности и нам нужно получить уникальные пороговые значения, отличающиеся друг от друга. На помощь приходит функция `np.diff()`, которая возвращает n -ю разность элементов массива вероятностей (по умолчанию $n = 1$). Например, первые три значения $-0.02, -0.02, -0.03$ получены так: из 0.9 вычли 0.92, из 0.88 вычли 0.9, из 0.85 вычли 0.88.

```
# функция np.diff() возвращает n-ю разность элементов массива
# (по умолчанию n = 1), которая также может быть вычислена
# вдоль указанной оси или осей
np.diff(y_score)
```

```
аггай([-0.02, -0.02, -0.03, -0.03, -0.03, -0.04, -0.02, -0.01, -0.02,
       -0.1, -0.01, -0.01, -0.05, -0.01, -0.12, -0.07, -0.01, -0.08,
       -0.06])
```



Рис. 33 Иллюстрация работы функции `np.where()`

Функция `np.where()` вернет индексы только тех значений, которые отличаются от нуля. В нашем случае она вернет индексы всех значений.

```
# с помощью np.where() получаем индексы уникальных порогов
distinct_indices = np.where(np.diff(y_score))[0]
distinct_indices
```

```
аггай([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
```

Посмотрим, как сработали бы функции `np.diff()` и `np.where()` в случае равных вероятностей.

```
аггай([0.92, 0.88, 0.88, 0.85, 0.82,
       0.79, 0.75, 0.73, 0.72, 0.7 ,
       0.6 , 0.59, 0.58, 0.53, 0.52,
       0.4 , 0.33, 0.32, 0.24, 0.18])
```

Функция `np.diff()` вернет разности.

```
аггай([-0.04, 0, -0.03, -0.03, -0.03, -0.04, -0.02, -0.01,
       -0.02, -0.1, -0.01, -0.01, -0.05, -0.01, -0.12, -0.07,
       -0.01, -0.08, -0.06])
```

Функция `np.where()` вернет индексы только тех значений, которые отличаются от нуля.

```
аггай([ 0, 2, 3, 4, 5, 6, 7, 8, 9, 10,
       11, 12, 13, 14, 15, 16, 17, 18])
```

Возвращаемся к нашему примеру. Теперь вычисляем индекс последнего порога и добавляем его в конец массива с индексами порогов.

```
# вычисляем индекс последнего порога
end = np.array([y_true.size - 1])
end

array([19])

# добавляем этот индекс в конец массива с индексами порогов
threshold_indices = np.hstack((distinct_indices, end))
threshold_indices

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Наконец, получаем пороговые значения вероятности положительного класса, значение чувствительности (количество истинно положительных) и 1 – специфичности (количество ложноположительных) для каждого порога.

```
# получаем пороговые значения вероятности положительного класса
thresholds = y_score[threshold_indices]
thresholds

array([0.92, 0.9 , 0.88, 0.85, 0.82, 0.79, 0.75, 0.73, 0.72, 0.7 , 0.6 ,
       0.59, 0.58, 0.53, 0.52, 0.4 , 0.33, 0.32, 0.24, 0.18])

# получаем количество истинно положительных случаев для каждого порога
tps = np.cumsum(y_true)[threshold_indices]
tps

array([1, 2, 3, 3, 4, 5, 5, 6, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8])

# получаем количество ложноположительных случаев для каждого порога
fps = (1 + threshold_indices) - tps
fps

array([ 0,  0,  0,  1,  1,  2,  2,  3,  4,  5,  6,  7,  7,  8,  9, 10, 11, 12])
```

Еще раз взглянем, как вычислены чувствительность и 1 – специфичность. Чувствительность – это количество истинно положительных наблюдений в каждом индексе. $(1 + \text{threshold_indices})$ – это, по сути, количество положительных наблюдений в каждом индексе, таким образом, количество положительных наблюдений минус количество истинно положительных равно количеству ложноположительных.

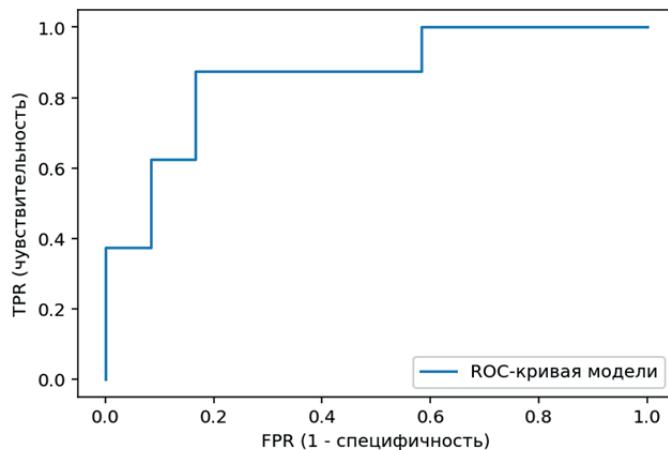
Теперь с помощью функции `roc_curve()` библиотеки scikit-learn мы построим ROC-кривую автоматически.

```
from sklearn.metrics import roc_curve(y_true, Фактические классы зависимой переменной
                                         y_score, Вероятности положительного класса
                                         pos_label=None) зависимой переменной, значения
уверенности, значения решающей
функции
```

Метка положительного класса зависимой переменной. Когда задано `pos_label=None`, а `y_true` принимает значения $\{-1, 1\}$ или $\{0, 1\}$, то метка положительного класса устанавливается равной 1, в противном случае будет выдана ошибка

```
# из модуля sklearn.metrics импортируем функцию roc_curve()
from sklearn.metrics import roc_curve

# вычисляем значения FPR и TPR для всех возможных
# порогов отсечения, передав функции roc_curve()
# в качестве аргументов фактические значения
# зависимой переменной и вероятности
fpr, tpr, thresholds = roc_curve(classes, proba)
# создаем заголовок ROC-кривой
plt.plot(fpr, tpr, label="ROC-кривая модели")
# задаем название для оси x
plt.xlabel("FPR (1 - специфичность)")
# задаем название для оси y
plt.ylabel("TPR (чувствительность)")
# задаем расположение легенды
plt.legend(loc=4);
```



Теперь разберем случай, когда наблюдениям присвоены веса.

Давайте создадим массив весов и опять построим ROC-кривую с помощью нашей функции `_binary_clf_curve()`.

```
# задаем веса наблюдений
sample_weights = np.array([1, 0, 1, 1, 1, 1, 1, 5, 1, 10,
                           10, 1, 1, 1, 1, 5, 1, 1, 1, 10])

# применяем нашу функцию _binary_clf_curve()
fps, tps, thresholds = _binary_clf_curve(
    classes, proba,
    sample_weight=sample_weights)
print("пороги:\n{thresholds}\n")
print("количество ложноположительных:\n{fps}\n")
print("количество истинно положительных:\n{tps}\n")

пороги:
[0.92 0.88 0.85 0.82 0.79 0.75 0.73 0.72 0.7 0.6 0.59 0.58 0.53 0.52
 0.4 0.33 0.32 0.24 0.18]

количество ложноположительных:
[ 0  0  1  1  2  2 12 22 23 24 25 25 30 31 32 33 43]
количество истинно положительных:
[ 1  2  2  3  4  4  9 10 10 10 10 11 11 11 11 11]
```

Вновь абсолютные частоты переведем в относительные и визуализируем результаты.

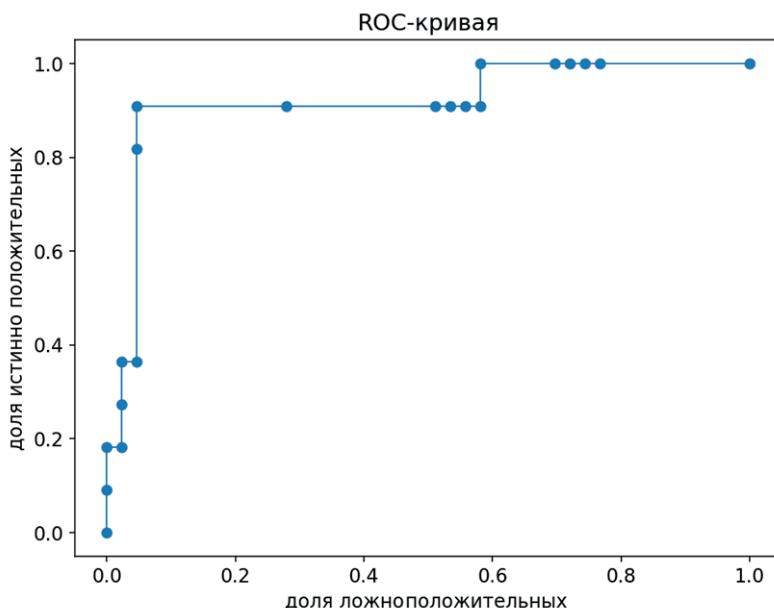
```
# преобразовываем количество в долю, добавляем 0
# к истинно положительным и ложноположительным,
# чтобы ROC-кривая брала начало в точке (0, 0)
fpr = np.hstack((0, fps / fps[-1]))
tpr = np.hstack((0, tps / tps[-1]))
print("доля ложноположительных:\n{}\n".format(fpr))
print("доля истинно положительных:\n{}\n".format(tpr))

# строим ROC-кривую
plt.rcParams['figure.figsize'] = 8, 6
plt.rcParams['font.size'] = 12

fig = plt.figure()
plt.plot(fpr, tpr, marker='o', lw=1)
plt.xlabel("доля ложноположительных")
plt.ylabel("доля истинно положительных")
plt.title("ROC-кривая")
plt.show()

доля ложноположительных:
[0.          0.          0.          0.02325581 0.02325581 0.02325581
 0.04651163 0.04651163 0.04651163 0.27906977 0.51162791 0.53488372
 0.55813953 0.58139535 0.58139535 0.69767442 0.72093023 0.74418605
 0.76744186 1.          ]]

доля истинно положительных:
[0.          0.09090909 0.18181818 0.18181818 0.27272727 0.36363636
 0.36363636 0.81818182 0.90909091 0.90909091 0.90909091 0.90909091
 0.90909091 0.90909091 1.          1.          1.          1.
 1.          1.          ]]
```



Посмотрим, что происходит под капотом.

Сначала мы создаем булеву маску: `True`, если вес наблюдения не равен 0, `False` в противном случае.

```
# создаем булеву маску: True, если вес не равен 0,
# False в противном случае
nonzero_weight_mask = sample_weights != 0
nonzero_weight_mask

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

С помощью булевой маски из массива меток удаляем метки с нулевыми весами, из массива вероятностей – вероятности с нулевыми весами, из массива весов – нулевые веса.

```
# с помощью булевой маски удаляем из массива
# меток метки с нулевыми весами
y_true = classes[nonzero_weight_mask]
y_true

array([1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0])

# с помощью булевой маски удаляем из массива вероятностей
# положительного класса вероятности с нулевыми весами
y_score = proba[nonzero_weight_mask]
y_score

array([0.92, 0.88, 0.85, 0.82, 0.79, 0.75, 0.73, 0.72, 0.7 , 0.6 , 0.59,
       0.58, 0.53, 0.52, 0.4 , 0.33, 0.32, 0.24, 0.18])

# с помощью булевой маски удаляем
# из массива весов нулевые веса
smpl_weights = sample_weights[nonzero_weight_mask]
smpl_weights

array([ 1,  1,  1,  1,  1,  5,  1, 10, 10,  1,  1,  1,  1,  5,  1,  1,  1, 10])
```

Теперь получаем индексы вероятностей положительного класса, отсортированных по убыванию.

```
# получаем индексы вероятностей положительного класса,
# отсортированных по убыванию
desc_score_indices = np.argsort(y_score)[::-1]
desc_score_indices

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
```

С помощью индексов сортируем вероятности, метки и веса.

```
# сортируем вероятности с помощью индексов
y_score = y_score[desc_score_indices]
y_score
```

```
array([0.92, 0.88, 0.85, 0.82, 0.79, 0.75, 0.73, 0.72, 0.7 , 0.6 , 0.59,
       0.58, 0.53, 0.52, 0.4 , 0.33, 0.32, 0.24, 0.18])

# сортируем метки с помощью индексов
y_true = y_true[desc_score_indices]
y_true

array([1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0])

# сортируем веса с помощью индексов
weights = smpl_weights[desc_score_indices]
weights

array([ 1,  1,  1,  1,  1,  5,  1, 10, 10,  1,  1,  1,  1,  5,  1,  1,  1, 10])
```

Теперь получаем индексы порогов.

```
# с помощью np.where() получаем индексы уникальных порогов
distinct_indices = np.where(np.diff(y_score))[0]
distinct_indices

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17])
```

Теперь вычисляем индекс последнего порога и добавляем его в конец массива с индексами порогов.

```
# вычисляем индекс последнего порога
end = np.array([y_true.size - 1])
end

array([18])

# добавляем этот индекс в конец массива с индексами порогов
threshold_indices = np.hstack((distinct_indices, end))
threshold_indices

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
```

Получаем пороговые значения вероятности положительного класса, значение чувствительности (количество истинно положительных) и 1 – специфичности (количество ложноположительных) для каждого порога.

```
# получаем пороговые значения вероятности положительного класса
thresholds = y_score[threshold_indices]
thresholds

array([0.92, 0.88, 0.85, 0.82, 0.79, 0.75, 0.73, 0.72, 0.7 , 0.6 , 0.59,
       0.58, 0.53, 0.52, 0.4 , 0.33, 0.32, 0.24, 0.18])

# получаем количество истинно положительных случаев для каждого порога
tps = np.cumsum(y_true * weights)[threshold_indices]
tps

array([ 1,  2,  2,  3,  4,  4,  9, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11])
```

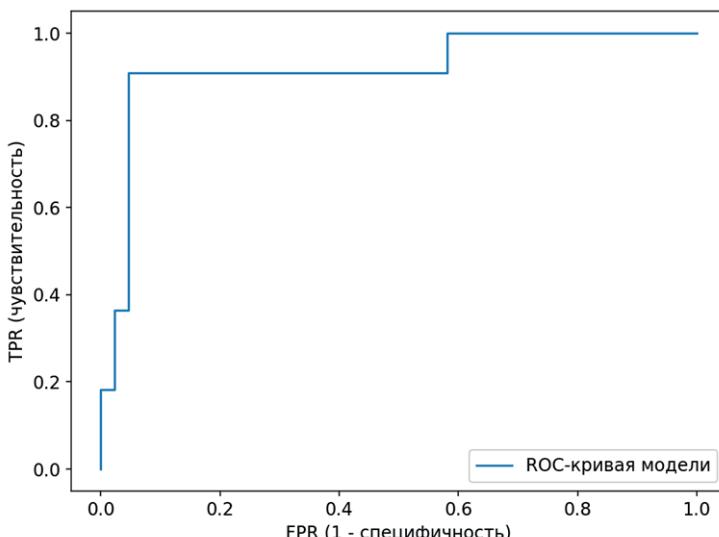
```
# получаем количество ложноположительных случаев для каждого порога
fps = np.cumsum((1 - y_true) * weights)[threshold_indices]
fps

array([ 0,  0,  1,  1,  2,  2,  2, 12, 22, 23, 24, 25, 25, 30, 31, 32, 33, 43])
```

Еще раз взглянем, как вычислены чувствительность и $1 - \text{специфичность}$. Чувствительность – это количество взвешенных истинно положительных наблюдений в каждом индексе. $(1 - y_{\text{true}})$ – это количество ложноположительных наблюдений – случаев, когда наблюдение класса 0 классифицируется как наблюдение класса 1 (как у нас и дано в круглых скобках). Таким образом, $1 - \text{специфичность}$ – это количество взвешенных ложноположительных наблюдений в каждом индексе.

Теперь построим ROC-кривую автоматически с помощью функции `roc_curve()`.

```
# вычисляем значения FPR и TPR для всех возможных
# порогов отсечения, передав функции roc_curve()
# в качестве аргументов фактические значения
# зависимой переменной, вероятности и веса
fpr, tpr, thresholds = roc_curve(
    classes, proba, sample_weight=sample_weights)
# создаем заголовок ROC-кривой
plt.plot(fpr, tpr, label="ROC-кривая модели")
# задаем название для оси x
plt.xlabel("FPR (1 - специфичность)")
# задаем название для оси y
plt.ylabel("TPR (чувствительность)")
# задаем расположение легенды
plt.legend(loc=4);
```



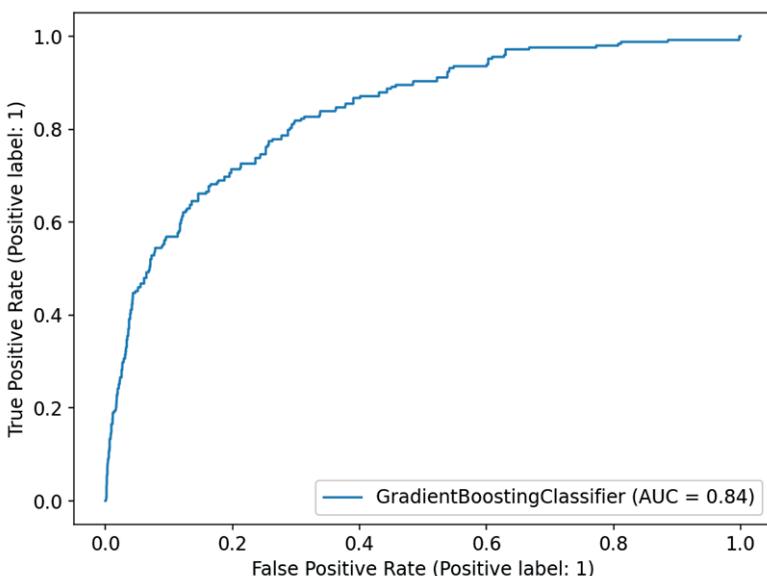
На практике пользуются более коротким путем. С помощью методов `.from_estimator()` и `.from_predictions()` класса `RocCurveDisplay` можно построить ROC-кривую на основе модели и на основе прогнозов модели соответственно.

```
# импортируем необходимые функции и классы
from sklearn.model_selection import train_test_split
from sklearn.metrics import RocCurveDisplay
from sklearn.ensemble import GradientBoostingClassifier

# записываем CSV-файл в объект DataFrame
df = pd.read_csv('Data/StateFarm.csv', sep=';')

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    df.drop('Response', axis=1),
    df['Response'],
    test_size=0.3,
    stratify=df['Response'],
    random_state=42)

# создаем экземпляр класса GradientBoostingClassifier
boost = GradientBoostingClassifier(
    subsample=0.8, random_state=42)
# обучаем модель
boost.fit(X_train, y_train)
# строим ROC-кривую для модели
RocCurveDisplay.from_estimator(boost, X_test, y_test);
```



Нередко можно получить ROC-кривую, часть которой лежит выше базовой линии, а часть – ниже базовой линии. Такое нередко бывает, когда классы не являются линейно разделимыми, а при этом применяется линейная модель.

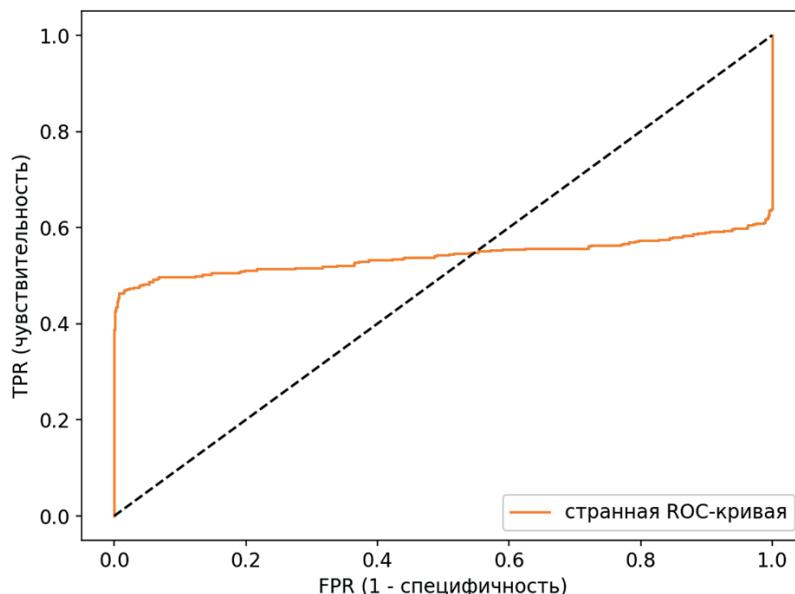
В примере ниже как раз приводится данная ситуация, в результате для всех наблюдений независимо от своего класса были получены низкие вероятности в диапазоне от 0,25 до 0,39 (меньше 0,5).

```
# записываем CSV-файл в объект DataFrame
weird_data = pd.read_csv('Data/weird_roc.csv', sep=',')
weird_data.tail(10)
```

	fact	prob
1473	1	0.366743
1474	1	0.295093
1475	1	0.306020
1476	1	0.270486
1477	1	0.368694
1478	1	0.271503
1479	1	0.350956
1480	1	0.355348
1481	1	0.283862
1482	1	0.295863

```
# строим ROC-кривую
fpr, tpr, thresholds = roc_curve(weird_data['fact'],
                                  weird_data['prob'])

plt.plot(fpr, tpr)
plt.xlabel("FPR (1 - специфичность)")
plt.ylabel("TPR (чувствительность)")
plt.plot(fpr, tpr, label="странныя ROC-кривая")
plt.plot([0, 1], [0, 1], "k--")
plt.legend(loc=4);
```



Несмотря на свой странный вид, ROC-кривая является по-прежнему монотонной и неубывающей. По мере уменьшения порогового значения мы движемся от низких значений чувствительности и 1 – специфичности к высоким значениям этих показателей.

На собеседованиях часто задают вопрос *Какие из этих кривых могут быть ROC-кривыми?* из теста Александра Дьяконова: <https://docs.google.com/forms/d/e/1FAIpQLSfrZOU9TaDWlxvBabf8saK-unmijfOHwkANpARNCrVQ-g3KyQ/viewform>.

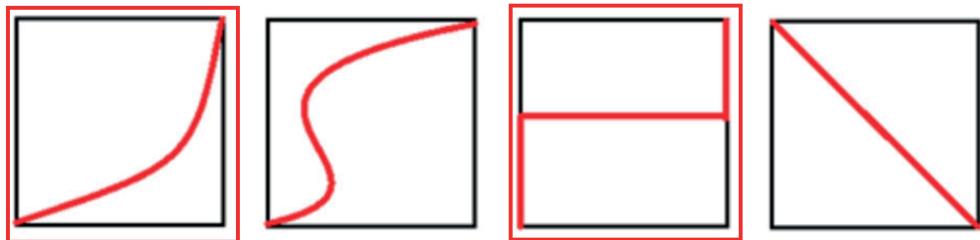


Рис. 34 Не все кривые являются ROC-кривыми. Источник: <https://docs.google.com/forms/d/e/1FAIpQLSfrZOU9TaDWlxvBabf8saK-unmijfOHwkANpARNCrVQ-g3KyQ/viewform>

Правильные ответы выделены красной рамкой. Здесь опять же нужно вспомнить, что ROC-кривая является монотонной и неубывающей функцией.

Визуальное сравнение двух и более ROC-кривых не всегда позволяет выявить наиболее эффективную модель. Для сравнения двух и более ROC-кривых сравниваются площади под кривыми. Площадь под ROC-кривой часто обозначают как AUC или AUC-ROC (Area Under ROC Curve). Она меняется от 0 до 1. Чем больше значение AUC-ROC, тем выше качество модели. Значение 0,5 соответствует случайному угадыванию (соответствует базовой линии – диагональной линии, проведенной из точки (0,0) – нижнего левого угла единичного квадрата в точку (1,1) – верхний правый угол). Значение менее 0,5 говорит, что классификатор работает хуже случайного угадывания. В этой ситуации можно инвертировать метки классов и получить классификатор лучше случайного, то есть ROC-кривая преобразованного классификатора будет лежать выше диагонали.

Поэтому для удобства принимают, что ROC-кривая всегда лежит выше диагонали или совпадает с ней, тогда AUC-ROC будет принимать значения в диапазоне от 0,5 до 1.

Обычно считают, что значение AUC-ROC от 0,9 до 1 соответствует отличной дискриминирующей способности модели, 0,8–0,9 – очень хорошей, 0,7–0,8 – хорошей, 0,6–0,7 – средней, 0,5–0,6 – неудовлетворительной. В нашем случае AUC-ROC равен 0,865, 83 клетки под ROC-кривой делим на общее количество клеток ($12 \times 8 = 96$).

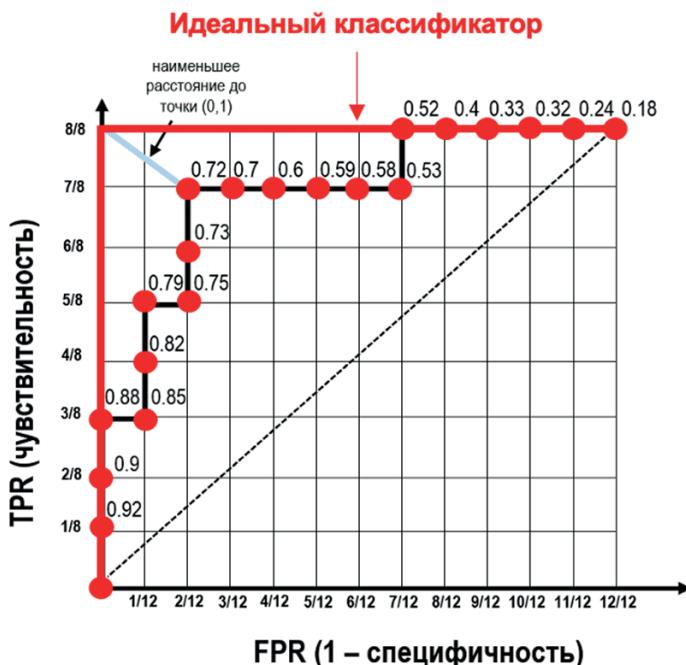


Рис. 35 Площадь под ROC-кривой можно посчитать по клеткам

Давайте для нашего игрушечного примера из 20 наблюдений вычислим AUC-ROC с помощью собственной функции `_goc_auc_score()`.

```
# пишем функцию, которая будет вычислять AUC-ROC
def _goc_auc_score(y_true, y_score):
    """
    Вычисляет AUC-ROC

    Параметры
    -----
    y_true : одномерный массив формы [n_samples]
        Фактические метки (классы) зависимой переменной.
    y_score : одномерный массив формы [n_samples]
        Спрогнозированные вероятности положительного класса.

    Возвращает
    -----
    auc : float
    """

    # убедимся, что зависимая переменная является бинарной
    if np.unique(y_true).size != 2:
        raise ValueError("Лишь два класса должно быть в y_true. Значение "
                         "'AUC-ROC не определяется в данном случае.'")

```

```
# получаем с помощью функции _binary_clf_curve()
# значения 1 - специфичности и чувствительности
fps, tps, _ = _binary_clf_curve(y_true, y_score)
# переходим колям
tpr = tps / tps[-1]
fpr = fps / fps[-1]
# вычисляем AUC с помощью метода трапеций;
# добавляем 0 для обеспечения соответствия длины
zero = np.array([0])
tpr_diff = np.hstack((np.diff(tpr), zero))
fpr_diff = np.hstack((np.diff(fpr), zero))
auc = np.dot(tpr, fpr_diff) + np.dot(tpr_diff, fpr_diff) / 2
return auc
```

Теперь вычисляем значение AUC-ROC, передав нашей функции `_gosc_auc_score()` в качестве аргументов фактические значения зависимой переменной и вероятности.

```
# вычисляем значение AUC-ROC, передав нашей функции
# _gosc_auc_score() в качестве аргументов фактические
# значения зависимой переменной и вероятности
auc_gosc = _gosc_auc_score(classes, proba)
# печатаем AUC-ROC
```

```
auc_gosc
```

```
0.8645833333333333
```

Затем автоматически вычислим AUC-ROC с помощью функции `roc_auc_score()`.

```
from sklearn.metrics import roc_auc_score(y_true, ← Фактические классы зависимой переменной
                                            y_score) ← Вероятности положительного
                                            класса зависимой переменной
```



```
# импортируем функцию gosc_auc_score()
from sklearn.metrics import gosc_auc_score
# вычисляем значение AUC-ROC, передав функции
# gosc_auc_score() в качестве аргументов фактические
# значения зависимой переменной и вероятности
auc_gosc = gosc_auc_score(classes, proba)
# печатаем AUC-ROC
```

```
auc_gosc
```

```
0.8645833333333333
```

В обоих случаях оценка AUC-ROC была вычислена с помощью численного метода трапеций. Это обозначает, что для вычисления AUC-ROC мы должны сложить площади прямоугольных и треугольных участков под кривой. Как правило, ROC-кривая будет состоять из прямоугольных и треугольных участков, на рисунке ниже прямоугольный участок показан слева, а треугольный участок – справа.

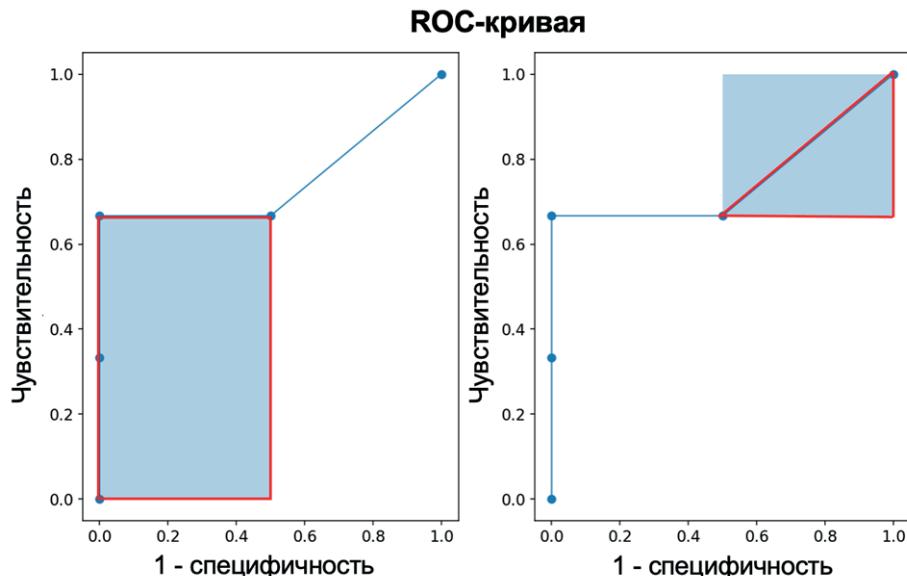


Рис. 36 Прямоугольные и треугольные участки ROC-кривой

Для прямоугольных участков высота – это TPR, а ширина – это разность FPR. Таким образом, общая площадь всех прямоугольников – это скалярное произведение TPR и разностей FPR. Для треугольных участков высота – это разность TPR, а ширина – это разность FPR. Таким образом, общая площадь всех прямоугольников – это скалярное произведение разностей TPR и разностей FPR. Однако в случае треугольных участков только половина каждого прямоугольника лежит ниже ROC-кривой, поэтому площадь прямоугольника мы должны разделить на 2, чтобы получить площадь треугольника. Поэтому общая площадь будет равна сумме скалярного произведения TPR и разностей FPR и скалярного произведения разностей TPR и разностей FPR, поделенного на 2. Это мы и видим в программном коде функции `_goc_auc_score()`, вычисляющей AUC-ROC.

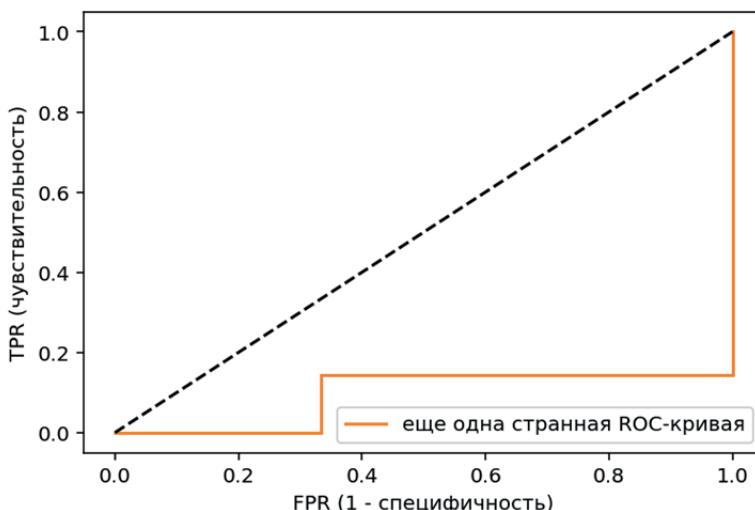
```
# вычисляем AUC с помощью метода трапеций
# добавляем 0 для обеспечения соответствия длины
zero = np.append([0])
tpr_diff = np.hstack((np.diff(tpr), zero))
fpr_diff = np.hstack((np.diff(fpr), zero))
auc = np.dot(tpr, fpr_diff) + np.dot(tpr_diff, fpr_diff) / 2
```

Выше мы говорили, что если значение AUC-ROC меньше 0,5, то в этой ситуации можно инвертировать метки классов и получить классификатор лучше случайного, то есть ROC-кривая преобразованного классификатора будет лежать выше диагонали. Давайте убедимся в этом. Для этого создаем массив фактических меток и массив вероятностей положительного класса, при котором можно получить значение AUC-ROC меньше 0,5.

```
# иногда можно получить AUC-ROC меньше 0.5
cl = np.array([1, 1, 1, 0, 1, 1, 0, 1, 1, 0])
pr = np.array([0.45, 0.41, 0.51, 0.58, 0.86,
              0.42, 0.72, 0.18, 0.44, 0.95])
auc_goc = roc_auc_score(cl, pr)
auc_goc

0.09523809523809525

# строим ROC-кривую
fpr, tpr, thresholds = roc_curve(cl, pr)
plt.plot(fpr, tpr)
plt.xlabel("FPR (1 - специфичность)")
plt.ylabel("TPR (чувствительность)")
plt.plot(fpr, tpr, label="еще одна странная ROC-кривая")
plt.plot([0, 1], [0, 1], "k--")
plt.legend(loc=4);
```

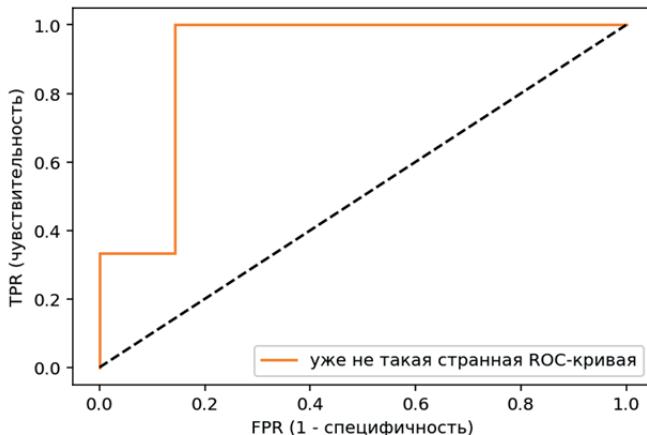


Применим инверсию меток классов и вновь построим ROC-кривую.

```
# выполняем инверсию меток классов
invert_cl = np.where(cl == 1, 0, 1)
auc_goc = roc_auc_score(invert_cl, pr)
auc_goc

0.9047619047619049

# строим ROC-кривую
fpr, tpr, thresholds = roc_curve(invert_cl, pr)
plt.plot(fpr, tpr)
plt.xlabel("FPR (1 - специфичность)")
plt.ylabel("TPR (чувствительность)")
plt.plot(fpr, tpr, label="уже не такая странная ROC-кривая")
plt.plot([0, 1], [0, 1], "k--")
plt.legend(loc=4);
```



Помимо метода трапеций, есть и другие способы вычислить AUC-ROC.

AUC-ROC классификатора C – это вероятность того, что классификатор C присвоит случайно отобранному наблюдению положительного класса более высокий ранг, чем случайно отобранному наблюдению отрицательного класса (если пренебречь вероятностью того, что ранг обоих будет одинаковым). Таким образом, $AUC(C) = P[C(x^+) > C(x^-)]$. Именно это определение AUC-ROC мы давали в самом начале книги. AUC-ROC позволяет нам выяснить, насколько хорошо мы ранжируем наших клиентов, например располагая клиентов от самых «хороших» к самым «плохим» и выбирая порог отсечения для кредитной политики. Давайте проверим эту вероятностную интерпретацию AUC-ROC на нашем игрушечном примере из 20 наблюдений.

Импортируем класс `RandomState`. Мы будем проводить эксперимент со случайнym извлечением наблюдений, и для воспроизводимости нам нужно будет задать стартовое значение генератора псевдослучайных чисел.

```
# импортируем класс RandomState
from numpy.random import RandomState
```

Создаем массив меток зависимой переменной и массив спрогнозированных вероятностей положительного класса.

```
# создаем массив меток классов зависимой переменной
classes = np.array([1, 1, 1, 0, 1,
                    1, 0, 1, 1, 0,
                    0, 0, 0, 0, 1,
                    0, 0, 0, 0, 0])

# создаем массив спрогнозированных вероятностей
# положительного класса
probas = np.array([0.92, 0.9, 0.88, 0.85, 0.82,
                   0.79, 0.75, 0.73, 0.72, 0.7,
                   0.6, 0.59, 0.58, 0.53, 0.52,
                   0.4, 0.33, 0.32, 0.24, 0.18])
```

Записываем вероятности положительных и отрицательных примеров.

```
# записываем вероятности положительных
# и отрицательных примеров
pos = proba[np.where(classes == 1)]
neg = proba[np.where(classes == 0)]
```

Извлекаем случайным образом положительные и отрицательные примеры и вычисляем долю случаев, когда положительные примеры имели более высокое значение вероятности положительного класса, чем отрицательные.

```
# задаем стартовое значение генератора
# случайных чисел для воспроизводимости
seed = 14
# извлекаем случайным образом положительные и
# отрицательные примеры и вычисляем долю случаев,
# когда положительные примеры получили более
# высокую вероятность, чем отрицательные
size = 200000
random_pos = RandomState(seed).choice(pos, size=size)
random_neg = RandomState(seed).choice(neg, size=size)
p = np.sum(random_pos > random_neg) / size
p
```

0.86439

Пришли к практически тому же самому значению, которое получили ранее, разделив количество клеток под ROC-кривой на общее количество клеток.

Однако недостаток такой интерпретации заключается в том, что мы пре-небрегаем часто встречающейся ситуацией равенства вероятностей, поэтому правильнее сказать, что AUC-ROC равен доле пар объектов вида (наблюдение класса 1, наблюдение класса 0), которые алгоритм верно упорядочил в соот-ветствии с формулой:

$$\frac{\sum_1^{n_i} \sum_1^{n_j} S(x_i, x_j)}{n_i \times n_j}.$$

В этой формуле x – ответ алгоритма для наблюдения (при этом это может быть не только вероятность, но и целое число). Наблюдения положительного класса имеют нижний индекс i , наблюдения отрицательного класса имеют нижний индекс j . Важнейшим компонентом формулы является правило скоринга:

$$S(x_i, x_j) = \begin{cases} 1, & x_i > x_j \\ \frac{1}{2}, & x_i = x_j \\ 0, & x_i < x_j \end{cases}.$$

По сути, числитель дроби в формуле представляет собой сумму количеств j -х наблюдений отрицательного класса, лежащих ниже каждого i -го наблюдения положительного класса. Каждое такое количество мы берем по каждому i -му наблюдению положительного класса в последовательности, отсортированной

по мере убывания вероятности положительного класса. Знаменатель дроби – это произведение количества i -х наблюдений положительного класса и j -х наблюдений отрицательного класса.

Давайте воспользуемся этой формулой. Для этого вернемся к нашему примеру, состоящему из 8 наблюдений положительного класса и 12 наблюдений отрицательного класса. Мы сразу можем сказать, что у нас будет $8 \times 12 = 96$ пар. 96 идет в знаменатель дроби нашей формулы на предыдущем слайде.

Теперь в этом наборе, отсортированном по убыванию вероятности положительного класса, мы берем наблюдение положительного класса под номером 20 (самая верхняя строка таблицы) и каждый раз образовываем пару с наблюдением отрицательного класса, лежащим ниже его. У нас будет 12 пар, 12 раз наблюдение положительного класса под номером 20 было проранжировано выше наблюдений отрицательного класса 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2 и 1, и в соответствии со скоринговым правилом мы складываем «единички» и получаем $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 12$. Записываем число 12 напротив наблюдения 20. По сути, мы подсчитали количество наблюдений отрицательного класса, лежащих ниже нашего наблюдения положительного класса под номером 20.

Отсортированные спрогнозированные вероятности положительного класса

№	фактический класс	спрогнозированная вероятность положительного класса	скоринговое правило	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
20	P	0,92	0	12
19	P	0,9	0	
18	P	0,88	0	
12	N	0,85	1	
17	P	0,82	0	
16	P	0,79	0	
11	N	0,75	1	
15	P	0,73	0	
14	P	0,72	0	
10	N	0,7	1	
9	N	0,6	1	
8	N	0,59	1	
7	N	0,58	1	
6	N	0,53	1	
13	P	0,52	0	
5	N	0,4	1	
4	N	0,33	1	
3	N	0,32	1	
2	N	0,24	1	
1	N	0,18	1	

Рис. 37 Вычисляем AUC-ROC по формуле $\frac{\sum_1^{n_i} \sum_1^{n_j} S(x_i, x_j)}{n_i \times n_j}$: считаем «единички», когда наше наблюдение положительного класса № 20 было проранжировано выше наблюдения отрицательного класса

наблюдение положительного класса № 20 было проранжировано выше наблюдения отрицательного класса

Аналогичный процесс повторяем для каждого наблюдения положительного класса, опускаясь вниз.

Отсортированные спрогнозированные вероятности положительного класса

№	фактический класс	спрогнозированная вероятность положительного класса	скоринговое правило	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
20	P	0,92	0	12
19	P	0,9	0	12
18	P	0,88	0	12
12	N	0,85	1	
17	P	0,82	0	11
16	P	0,79	0	11
11	N	0,75	1	
15	P	0,73	0	10
14	P	0,72	0	10
10	N	0,7	1	
9	N	0,6	1	
8	N	0,59	1	
7	N	0,58	1	
6	N	0,53	1	
13	P	0,52	0	5
5	N	0,4	1	
4	N	0,33	1	
3	N	0,32	1	
2	N	0,24	1	
1	N	0,18	1	

Рис. 38 Вычисляем AUC-ROC по формуле $\frac{\sum_{i=1}^{n_p} \sum_{j=1}^{n_n} S(x_i, x_j)}{n_p \times n_n}$: вычисляем количество «единичек» для каждого наблюдения положительного класса

Затем суммируем полученные значения и сумму делим на произведение количества наблюдений положительного класса и количества наблюдений отрицательного класса.

В итоге получаем:

$$AUC_ROC = \frac{1}{8 \times 12} (12 + 12 + 12 + 11 + 11 + 10 + 10 + 5) = \frac{83}{96} = 0,865.$$

Из 96 пар 83 мы упорядочили правильно, 83 поделили на 96 и получили AUC-ROC 0,865.

вычислим AUC вручную

```
denominator = np.bincount(classes)[0] * np.bincount(classes)[1]
numerator = (12 + 12 + 12 + 11 + 11 + 10 + 10 + 5)
```

```
manually_calculated_auc = numerator / denominator
print("AUC-ROC {:.4f}".format(manually_calculated_auc))
```

AUC-ROC 0.8646

Здесь же поясним бизнес-смысль AUC-ROC.

Допустим, мы задали порог 0,72, клиентам с вероятностями выше порога не выдаем кредит, а клиентам с вероятностями ниже порога выдаем кредит. Однако здесь у нас среди «хороших» заемщиков попался один «плохой» заемщик (наблюдение 13). Мы понесем потери. Среди «плохих» заемщиков окажутся два «хороших» заемщика (наблюдения 11 и 12). Мы также понесем потери, теперь уже из-за упущенной выгоды.

Отсортированные спрогнозированные вероятности положительного класса

№	фактический класс	спрогнозированная вероятность положительного класса	скоринговое правило $S(x_i, x_j)$ = $\begin{cases} 1, & x_i > x_j \\ \frac{1}{2}, & x_i = x_j \\ 0, & x_i < x_j \end{cases}$	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
20	P	0,92	0	12
19	P	0,9	0	12
18	P	0,88	0	12
12	N	0,85	1	
17	P	0,82	0	11
16	P	0,79	0	11
11	N	0,75	1	
15	P	0,73	0	10
14	P	0,72	0	10
10	N	0,7	1	
9	N	0,6	1	
8	N	0,59	1	
7	N	0,58	1	
6	N	0,53	1	
13	P	0,52	0	5
5	N	0,4	1	
4	N	0,33	1	
3	N	0,32	1	
2	N	0,24	1	
1	N	0,18	1	

Рис. 39 Бизнес-смысль AUC-ROC

В идеале мы бы хотели, чтобы все наблюдения положительного класса находились в верхней части, а все наблюдения отрицательного класса находились в нижней части таблицы, то есть чтобы все наблюдения положительного класса были проранжированы выше, чем наблюдения отрицательного класса. Взглядните на рисунок ниже.

Отсортированные спрогнозированные вероятности положительного класса идеальный случай

№	фактический класс	спрогнозированная вероятность положительного класса	скоринговое правило	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
20	P	0,95	$S(x_i, x_j) = \begin{cases} 1, & x_i > x_j \\ \frac{1}{2}, & x_i = x_j \\ 0, & x_i < x_j \end{cases}$	12
19	P	0,91		12
18	P	0,84		12
12	P	0,81		12
17	P	0,80		12
16	P	0,78		12
11	P	0,74		12
15	P	0,72		12
14	N	0,69		1
10	N	0,68		1
9	N	0,65		1
8	N	0,59		1
7	N	0,57		1
6	N	0,55		1
13	N	0,53		1
5	N	0,51		1
4	N	0,49		1
3	N	0,45		1
2	N	0,41		1
1	N	0,38		1

Рис. 40 Идеальный случай

В этом случае значение AUC-ROC будет равно 1. Тогда, выбрав порог (в данном случае – 0,69), мы надежно отделим наблюдения отрицательного класса от наблюдений положительного класса, например отделим «хороших» заемщиков от «плохих».

$$AUC_ROC = \frac{1}{8 \times 12} (12 + 12 + 12 + 12 + 12 + 12 + 12 + 12) = \frac{96}{96} = 1.$$

Из 96 пар 96 мы классифицировали правильно, 96 поделили на 96 и получили оценку AUC-ROC, равную 1.

```
# модифицируем массив спрогнозированных
# вероятностей положительного класса
probva = np.array([0.95, 0.91, 0.84, 0.81, 0.80,
                   0.78, 0.74, 0.72, 0.69, 0.68,
                   0.65, 0.59, 0.57, 0.55, 0.53,
                   0.51, 0.49, 0.45, 0.41, 0.38])
```

```
# вычислим AUC вручную
denominator = np.bincount(classes)[0] * np.bincount(classes)[1]
numerator = (12 + 12 + 12 + 12 + 12 + 12 + 12 + 12)
manually_calculated_auc = numerator / denominator
print("AUC-ROC {0:.4f}".format(manually_calculated_auc))
```

AUC-ROC 1.0000

Отдельно разберем случай равенства вероятностей. Для наблюдений положительного класса под номерами 20 и 19 (2 самые верхние строки таблицы) по-прежнему будет 12 наблюдений отрицательного класса, лежащих ниже его. А вот для наблюдения положительного класса под номером 18 мы берем уже 11,5 наблюдения, поскольку такую же вероятность имеет наблюдение отрицательного класса под номером 12. Взгляните на рисунок ниже.

Отсортированные спрогнозированные вероятности положительного класса случай равенства вероятностей

№	фактический класс	спрогнозированная вероятность положительного класса	скоринговое правило $S(x_i, x_j) = \begin{cases} 1, & x_i > x_j \\ \frac{1}{2}, & x_i = x_j \\ 0, & x_i < x_j \end{cases}$	количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
20	P	0,92	0	12
19	P	0,9	0	12
18	P	0,88	0,5	11,5
12	N	0,88	0,5	11,5
17	P	0,82	0	11
16	P	0,79	0	11
11	N	0,75	1	
15	P	0,73	0	10
14	P	0,72	0	10
10	N	0,7	1	
9	N	0,6	1	
8	N	0,59	1	
7	N	0,58	1	
6	N	0,53	1	
13	P	0,52	0	5
5	N	0,4	1	
4	N	0,33	1	
3	N	0,32	1	
2	N	0,24	1	
1	N	0,18	1	

Рис. 41 Случай равенства вероятностей

В этом случае значение AUC-ROC будет равно 0,859.

$$AUC_ROC = \frac{1}{8 \times 12} (12 + 12 + 11,5 + 11 + 11 + 10 + 10 + 5) = \frac{82,5}{96} = 0,859.$$

Из 96 пар 82,5 мы классифицировали правильно, 82,5 поделили на 96 и получили AUC-ROC 0,859.

```
# модифицируем массив спрогнозированных вероятностей
# положительного класса
proba = np.array([0.92, 0.9, 0.88, 0.88, 0.82,
                  0.79, 0.75, 0.73, 0.72, 0.7,
                  0.6, 0.59, 0.58, 0.53, 0.52,
                  0.4, 0.33, 0.32, 0.24, 0.18])

# вычислим AUC вручную
denominator = np.bincount(classes)[0] * np.bincount(classes)[1]
numerator = (12 + 12 + 11.5 + 11 + 11 + 10 + 10 + 5)
manually_calculated_auc = numerator / denominator
print("AUC-ROC {:.4f}".format(manually_calculated_auc))

AUC-ROC 0.8594
```

Здесь же отметим: AUC-ROC не зависит от преобразования спрогнозированных вероятностей. Мы можем возвести их в квадрат или поделить на 2, AUC-ROC не изменится, так как зависит не от самих спрогнозированных вероятностей, а от порядка ранжирования наблюдений.

Поскольку при вычислении AUC-ROC для нас важен лишь порядок ранжирования наблюдений, мы можем сделать неверные прогнозы и при этом получить идеальное значение AUC-ROC. Например, у нас есть 6 наблюдений, три наблюдения отрицательного класса и три наблюдения положительного класса. Три наблюдения положительного класса получили низкие вероятности меньше 0,5, и при пороге отсечения 0,5 мы получили 50 % неверных прогнозов, однако AUC-ROC будет равен 1.

```
# 50 % неверных прогнозов и AUC-ROC 1
cl = np.array([1, 1, 1, 0, 0, 0])
pr = np.array([0.45, 0.40, 0.38, 0.35, 0.33, 0.3])
predictions = np.where(pr >= 0.5, 1, 0)
print("auc-гос", gos_auc_score(cl, pr))
print("правильность", accuracy_score(cl, predictions))

auc-гос 1.0
правильность 0.5
```

Приведем еще один пример, когда получаем идеальную оценку AUC-ROC и при этом низкую правильность.

```
# 83 % неверных прогнозов и AUC-ROC 1
cls = np.array([1, 1, 1, 1, 1, 0])
prb = np.array([0.45, 0.43, 0.38, 0.36, 0.33, 0.3])
preds = np.where(prb >= 0.5, 1, 0)
print("auc-гос", gos_auc_score(cls, prb))
```

```
print("правильность", accuracy_score(cls, preds))

auc-gos 1.0
правильность 0.1666666666666666
```

Давайте разберем несколько популярных задач по AUC-ROC с собеседований (часто их берут из вышеупомянутого теста Александра Дьяконова по AUC-ROC).

Задача 1: может ли сумма двух алгоритмов с $AUC-ROC = 0,5$ иметь $AUC-ROC = 1$ (имеется в виду, что мы суммируем оценки, полученные разными алгоритмами)?

Интуитивно понятно, что да, может. Это знакомая ситуация, когда ответы алгоритмов (вероятности) при объединении взаимно дополняют друга, усиливают друг друга, в таких случаях еще говорят «возникает комплементарность».

Пусть у нас будет ситуация с идеальным балансом классов.

```
# создаем массив меток зависимой переменной
# (идеальный баланс)
cl = np.hstack([np.ones(10), np.zeros(10)])
cl

array([1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Нам необходимо, чтобы первый классификатор идеально прогнозировал ровно половину наблюдений положительного класса, а второй классификатор идеально прогнозировал другую половину наблюдений положительного класса. Та самая комплементарность! При этом наблюдения отрицательного класса должны получить такие вероятности, чтобы при сложении они были меньше сложенных вероятностей для наблюдений положительного класса.

```
# создаем массив вероятностей первого классификатора
pr = np.hstack([np.ones(5), np.zeros(5), .25 * np.ones(10)])
pr

array([1. , 1. , 1. , 1. , 1. , 0. , 0. , 0. , 0. , 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25])

# создаем массив вероятностей второго классификатора
pr2 = np.hstack([np.zeros(5), np.ones(5), .25 * np.ones(10)])
pr2

array([0. , 0. , 0. , 0. , 1. , 1. , 1. , 1. , 1. , 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25])
```

Давайте взглянем на AUC-ROC классификаторов и итоговый AUC-ROC.

```
# смотрим AUC классификаторов
print(
    roc_auc_score(cl, pr),
    roc_auc_score(cl, pr2),
    roc_auc_score(cl, pr + pr2)
)
```

0.5 0.5 1.0

Изобразим более наглядно.

Рис. 42 Случай, когда два классификатора с AUC-ROC 0,5 в сумме дают AUC-ROC 1

В коде это выглядит следующим образом.

```
# изобразим более наглядно
results = np.column_stack((cl, pr, pr2, pr + pr2))
results
```

```
array([[1. , 1. , 0. , 1. ],
       [1. , 1. , 0. , 1. ],
       [1. , 1. , 0. , 1. ],
       [1. , 1. , 0. , 1. ]],
```

```
[1. , 1. , 0. , 1. ],
[1. , 0. , 1. , 1. ],
[1. , 0. , 1. , 1. ],
[1. , 0. , 1. , 1. ],
[1. , 0. , 1. , 1. ],
[1. , 0. , 1. , 1. ],
[1. , 0. , 1. , 1. ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ],
[0. , 0.25, 0.25, 0.5 ]])
```

Задача 2. В тестовой выборке 10 объектов, известно, что AUC-ROC < 1. Какое максимальное значение может быть у AUC-ROC?

Чтобы максимизировать AUC-ROC, нам нужно максимизировать долю пар объектов, которые алгоритм верно упорядочил. Сумма попарных сравнений имеет максимальное значение при пяти наблюдениях положительного класса и пяти наблюдениях отрицательного класса. Количество попарных сравнений будет в этом случае равно $5 * 5 = 25$. Поскольку мы знаем, что AUC-ROC меньше 1, то для максимизации AUC-ROC при этих условиях нужно, чтобы алгоритм верно упорядочил все пары, кроме одной ($24 * 1$), для которой ответы алгоритма совпадают ($1 * 0.5$). Тогда формула получается такая: $(24 * 1 + 1 * 0.5) / 25 = 0.98$.

Задача 3. Задача XOR (первый класс – $(-1, -1), (+1, +1)$, второй – $(-1, +1), (+1, -1)$) решается линейным алгоритмом (оценка – линейная комбинация признаков). Чему равно максимально возможное значение AUC-ROC?

Линейные методы классификации строят очень простую разделяющую поверхность – гиперплоскость. Самый известный игрушечный пример, в котором классы нельзя без ошибок поделить гиперплоскостью (то есть прямой, если это двумерное пространство), получил название «задача XOR» (the XOR problem).

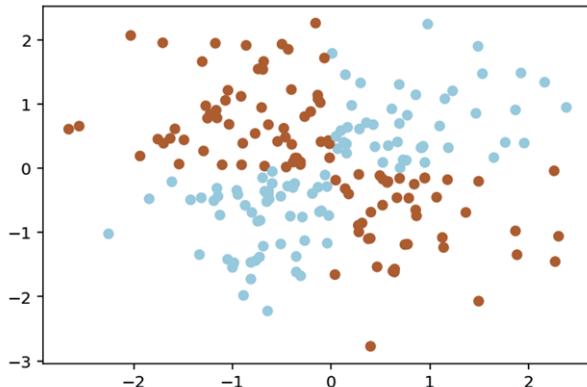
XOR – это «исключающее ИЛИ», булева функция со следующей таблицей истинности:

Input 1	Input 2	Output
0	0	0
0	1	1
1	1	0
1	0	1

XOR дал имя простой задаче бинарной классификации, в которой классы представлены вытянутыми по диагоналям и пересекающимися облаками точек.

```
# пример задачи XOR
rng = np.random.RandomState(0)
X = rng.randn(200, 2)
y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)
```

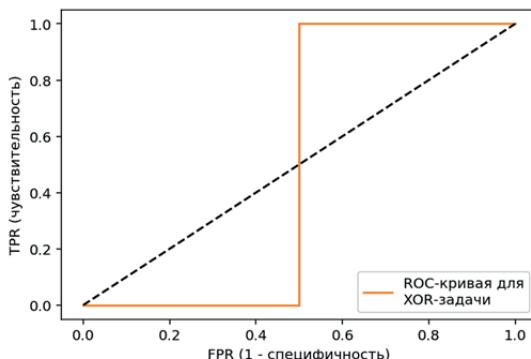
```
plt.scatter(X[:, 0],
            X[:, 1],
            s=30,
            c=y,
            cmap=plt.cm.Paired);
```



Очевидно, нельзя провести прямую так, чтобы без ошибок отделить один класс от другого. Поэтому линейные модели плохо справляются с такой задачей. Давайте убедимся в этом. Получим для наших наблюдений вероятности положительного класса с помощью логистической регрессии и вычислим AUC-ROC.

```
# импортируем класс SGDClassifier
from sklearn.linear_model import SGDClassifier
# создаем массив признаков
X = np.array([[-1, -1],
              [+1, +1],
              [-1, +1],
              [+1, -1]])
# создаем массив меток
y = [0, 0, 1, 1]
# строим логистическую регрессию
logreg = SGDClassifier(loss='log',
                        max_iter=500,
                        random_state=42)
logreg.fit(X, y)
# печатаем вероятности, коэффициенты и AUC-ROC
print("вероятности:", np.round(
    logreg.predict_proba(X)[:, 1], 3))
print("коэффициенты:", logreg.coef_)
print("auc_goc:", roc_auc_score(
    y, logreg.predict_proba(X)[:, 1]))
вероятности: [0.999 0. 0.199 0.144]
коэффициенты: [[-4.47126728 -4.27824711]]
auc_goc: 0.5
# строим ROC-кривую для XOR-задачи
fpr, tpr, thresholds = roc_curve(y, logreg.predict_proba(X)[:, 1])
plt.plot(fpr, tpr)
plt.xlabel("FPR (1 - специфичность)")
```

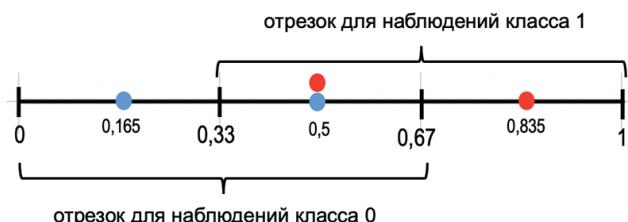
```
plt.ylabel("TPR (чувствительность)")
plt.plot(fpr, tpr, label="ROC-кривая для\nпXOR-задачи")
plt.plot([0, 1], [0, 1], "k-")
plt.legend(loc=4);
```



Задача 4. На ответах (оценках) алгоритма объекты класса 0 распределены равномерно на отрезке $[0, 2/3]$, а ответы класса 1 – равномерно на отрезке $[1/3, 1]$. Чему равен AUC-ROC?

Переведем обыкновенные дроби в десятичные: $1/3 \approx 0,33$, $2/3 \approx 0,67$.

Визуализируем оценки на отрезке от 0 до 1.



Сопоставим меткам вероятности.

Метки	Вероятности
0	0,165
0	0,5
1	0,5
1	0,835

Отсортируем по убыванию вероятностей.

Метки	Вероятности
1	0,835
1	0,5
0	0,5
0	0,165

Вычисляем AUC-ROC. Подсчитываем количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса, и делим на произведение количества наблюдений отрицательного класса и количества наблюдений положительного класса.

Метки	Вероятности	Количество наблюдений отрицательного класса, лежащих ниже соответствующего наблюдения положительного класса
1	0,835	2
1	0,5	
0	0,5	1,5
0	0,165	

$$AUC_ROC = \frac{1}{2 \times 2} (2 + 1,5) = \frac{3,5}{4} = 0,875.$$

Алгоритм верно упорядочил все пары ($4 * 1$), кроме одной, для которой ответы алгоритма совпадают ($1 * 0,5$). Тогда формула получается такая: $(3 * 1 + 1 * 0,5) / 4 = 0,875$.

проверяем решение

```
cl = np.array([0, 0, 1, 1])
pr = np.array([0.165, 0.5, 0.5, 0.835])
roc_auc_score(cl, pr)
```

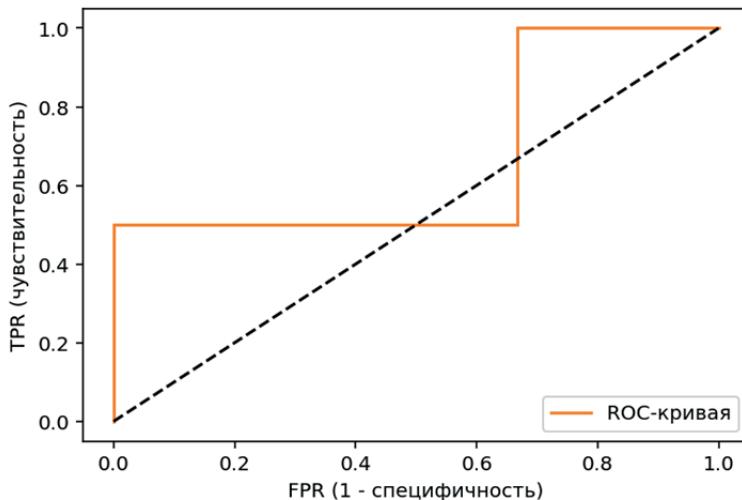
0.875

Задача 5. Объекты нулевого класса получили оценки 0,1, 0,4, 0,5, а первого – 0,2, 0,8. Чему равен AUC-ROC?

У нас три наблюдения отрицательного класса, и цена деления для оси x будет $1/3$, два наблюдения положительного класса и цена деления для оси y будет $1/2$. Сортируем объекты по убыванию вероятности.

Фактический класс	Спрогнозированная вероятность положительного класса
P	0,8
N	0,5
N	0,4
P	0,2
N	0,1

Берем единичный квадрат на координатной плоскости. Из точки (0,0) поднимаемся один раз вверх на $1/2$, потом два раза вправо на $1/3$, один раз вверх на $1/2$ и один раз вправо на $1/3$. У нас шесть клеток, 4 клетки под ROC-кривой, получаем $4 / 6 = 2/3$.



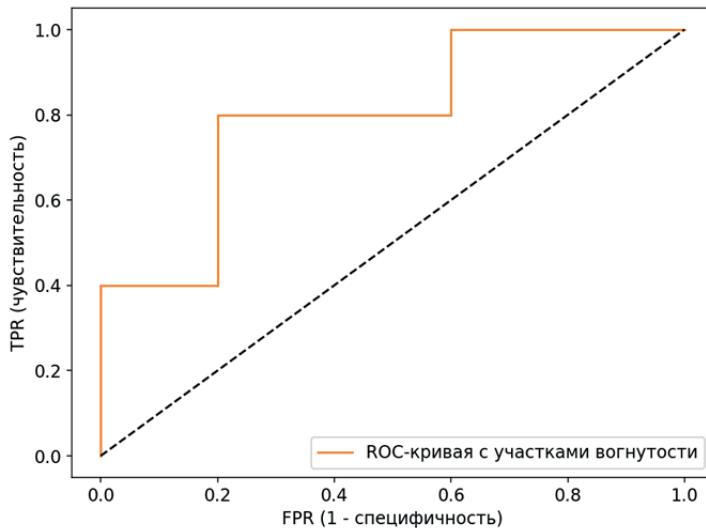
Задача 6. В каких случаях мы можем получить вогнутые участки ROC-кривой?

Если ROC-кривая имеет вогнутость, это означает, что существуют два пороговых значения, между которыми наши вероятности дают качество хуже, чем случайное угадывание. На рисунке внизу справа мы встречаем наблюдение отрицательного класса (0,7) между двумя наблюдениями положительного класса (0,8 и 0,6) и наблюдение отрицательного класса (0,3) между наблюдением отрицательного класса (0,4) и наблюдением положительного класса (0,2).

```
# создаем массив меток и массив вероятностей
cl = np.array([1, 1, 0, 1, 1, 0, 0, 1, 0, 0])
pr = np.array([0.9, 0.8, 0.7, 0.6, 0.5,
              0.4, 0.3, 0.2, 0.1, 0.05])
roc_auc_score(cl, pr)
```

0.80

```
# строим ROC-кривую с участками вогнутости
fpr, tpr, thresholds = roc_curve(cl, pr)
plt.plot(fpr, tpr)
plt.xlabel("FPR (1 - специфичность)")
plt.ylabel("TPR (чувствительность)")
string = "ROC-кривая с участками вогнутости"
plt.plot(fpr, tpr, label=string)
plt.plot([0, 1], [0, 1], "k--")
plt.legend(loc=4);
```



Давайте визуализируем участки вогнутости ROC-кривой.

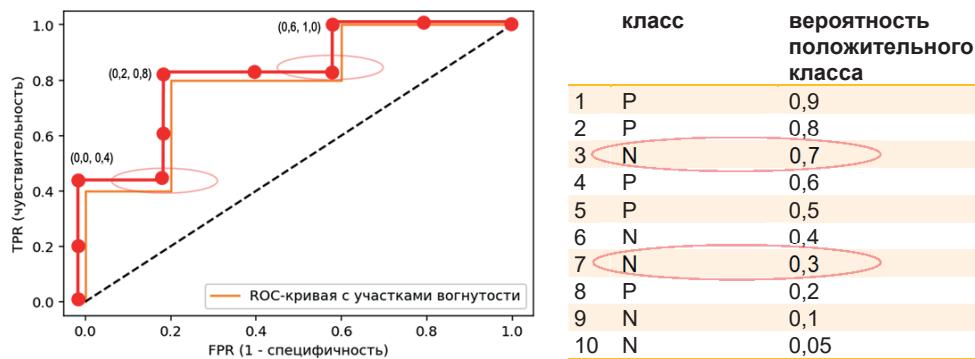


Рис. 43 Визуализация участков вогнутости ROC-кривой

Существует методика «ремонта» участков вогнутости ROC-кривой, к которым нужно относиться с осторожностью. Например, мы можем вернуться к случайному угадыванию между этими пороговыми значениями или (что практически эквивалентно) сделать оценки классификатора в этом интервале константными. Например, мы можем задать вероятность 0,6 для наблюдений с 3 по 5 и вероятность 0,3 для наблюдений с 6 по 8.

Nº	Фактический класс	Спрогнозированная вероятность положительного класса
1	P	0,9
2	P	0,8
3	N	0,6

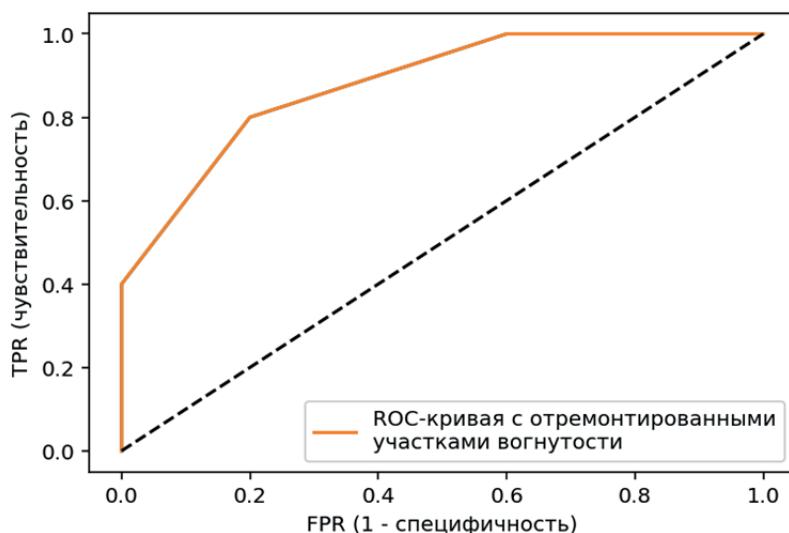
№	Фактический класс	Спрогнозированная вероятность положительного класса
4	P	0,6
5	P	0,6
6	N	0,3
7	N	0,3
8	P	0,3
9	N	0,1
10	N	0,05

Рис. 44 «Ремонт» ROC-кривой

```
# меняем вероятности, «ремонтируя» ROC-кривую
рг = пр.аггай([0.9, 0.8, 0.6, 0.6, 0.6,
                 0.3, 0.3, 0.3, 0.1, 0.05])
roc_auc_score(cl, рг)
```

0.88

```
# строим ROC-кривую с отремонтированными
# участками вогнутости
fрг, tрг, thresholds = roc_curve(cl, рг)
plt.plot(fрг, tрг)
plt.xlabel("FPR (1 - специфичность)")
plt.ylabel("TPR (чувствительность)")
string = ("ROC-кривая с отремонтированными\n" +
          "участками вогнутости")
plt.plot([0, 1], [0, 1], 'k--')
plt.legend(loc=4);
```



Давайте визуализируем «отремонтированные» участки вогнутости ROC-кривой.

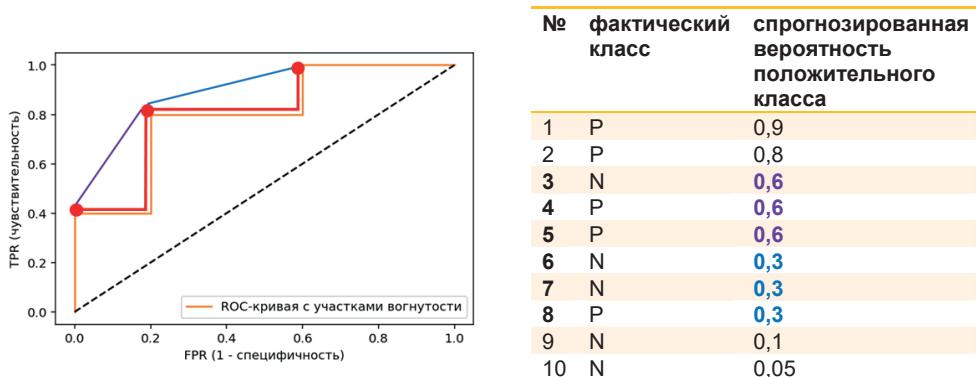


Рис. 45 Визуализация «отремонтированных» участков вогнутости ROC-кривой

Однако обратите внимание, что AUC выпуклой кривой будет больше (0,88), чем AUC исходной кривой (0,80). Процесс превращения ROC-кривой в более выпуклую кривую вносит положительное смещение оцененного значения AUC относительно фактического значения AUC при настройке большой выборки. Таким образом, в ходе оценки дискриминирующей способности важно иметь в виду, что оценка AUC, возникшая в результате «ремонта», является статистикой, смещенной вверх.

Задача 7. Есть большая выборка с бинарным классификатором, баланс классов 50 % / 50 %, и есть такая же выборка, но случайным образом превращенная в несбалансированную: 95 % / 5 %. Будет ли значение AUC-ROC на таких выборках отличаться друг от друга или будет примерно одинаковым?

При достаточно большом размере выборки изменение баланса классов практически не повлияет на AUC-ROC, поскольку TPR и FPR являются дробными значениями, вычисляемыми внутри одной строки (фактического класса). TPR вычисляется внутри строки, являющейся фактическим положительным классом, FPR – внутри строки, являющейся фактическим отрицательным классом, таким образом, они не зависят от баланса классов. Поэтому, в отличие от правильности, AUC-ROC устойчива к дисбалансу классов, информация об исходных распределениях классов и их размерах не используется, и для нас важен лишь порядок ранжирования объектов. И наоборот, метрики, вычисляемые сразу по обеим строкам – фактическим классам, например F-мера, точность, будут чувствительны к дисбалансу.

Исследователи Джесси Дэвис и Марк Гоадрич в своей работе «The relationship between Precision-Recall and ROC curves» («Взаимосвязь между PR-кривыми и ROC-кривыми»)⁷ отмечают, что «ROC-кривые могут чрезмерно оптимистично оценивать качество работы алгоритма в случае дисбаланса классов. [...] PR-кривые, часто используемые в информационном поиске, можно использовать в качестве альтернативы ROC-кривым для задач с дисбалансом классов».

Исследователи Такайя Сейто и Марк Ремсмайер в своей работе «The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets» («График PR-кривой более информативен, чем гра-

⁷ <https://www.biostat.wisc.edu/~page/rocpr.pdf>.

фик ROC-кривой для оценки качества бинарных классификаторов на несбалансированных наборах данных»)⁸ привели доказательства того, что ROC-кривая, в отличие от PR-кривой, не чувствительна к изменению распределения классов.

Они сгенерировали пять выборок с различными распределениями скоринговых баллов, то есть рассмотрели пять различных по качеству классификаторов:

- 1) случайный (random);
- 2) плохой (poor early retrieval);
- 3) хороший (good early retrieval);
- 4) отличный (excellent);
- 5) идеальный (perfect).

Далее для двух типов выборок, сбалансированной и несбалансированной, были построены четыре типа кривых для оценки качества классификации: ROC-кривые, CROC-кривые, Cost Curves (CC) и PR-кривые. Оказалось, что кривые ROC, CROC и CC показывают одинаковое качество классификаторов на сбалансированной и несбалансированной выборках. И только PR-кривые показывали отличия и, в частности, плохое качество классификаторов на несбалансированной выборке.

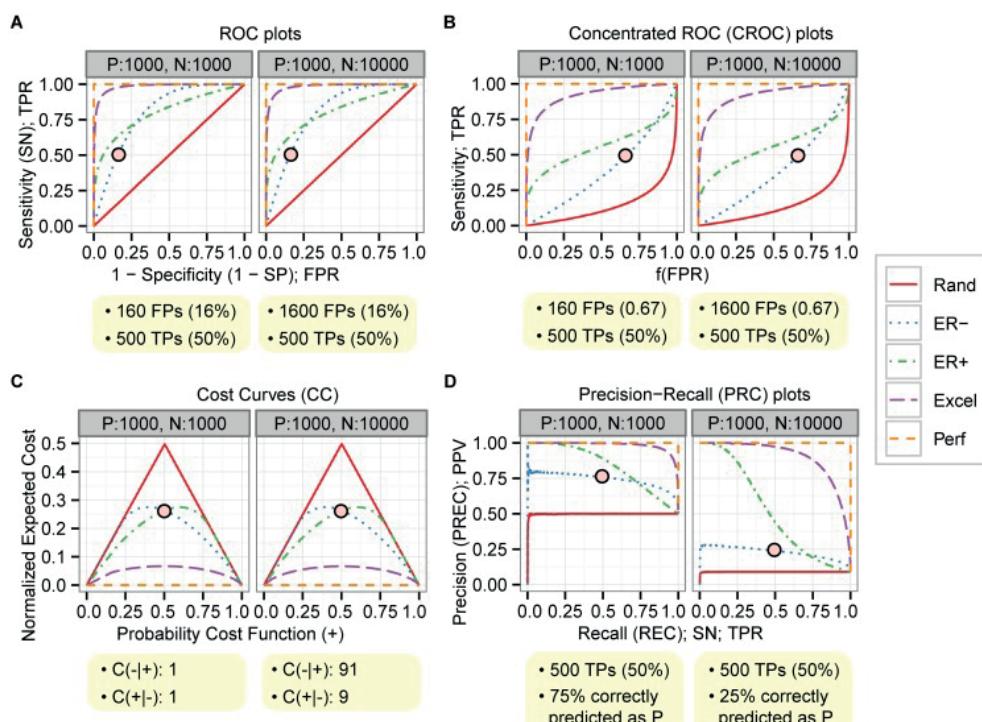


Рис. 46 Результаты эксперимента Сейто–Ремсмайера

На первый взгляд кажется, что мы можем без всяких преград использовать AUC для сравнения двух классификаторов. Эта идея основана на неявном

⁸ <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0118432>.

предположении, что AUC для обоих классификаторов был получен способом, который не зависит от распределения выходных значений решающей функции (то есть распределения оценок).

Однако Дэвид Хэнд в своей статье (*D. J. Hand. Measuring classifier performance: a coherent alternative to the area under the ROC curve. Machine Learning, 77:103–123, 2009*) показывает, что это не так: «*AUC оценивает классификатор, используя метрику, которая сама зависит от классификатора. То есть AUC оценивает разные классификаторы, используя разные метрики*». И далее он приводит следующую аналогию: «*Это похоже на то, как если бы кто-то измерил рост человека А, используя линейку, откалиброванную в дюймах, и рост человека В, используя линейку, откалиброванную в сантиметрах, и решил, что кто-то из них выше, просто сравнив числа, игнорируя тот факт, что использовались разные единицы измерения*».

При одинаковом AUC у разных моделей (соответственно, с разными ROC-кривыми) будет разное распределение стоимостей ошибочной классификации. Проще говоря, мы можем вычислить AUC для классификатора А и получить 0,7, а затем вычислить AUC для классификатора В и получить тот же AUC 0,7, но это не обязательно означает, что их качество аналогично.

В статье Дэвида Хэнда «*Measuring classifier performance: a coherent alternative to the area under the ROC curve*» можно найти очень хорошее интуитивное объяснение проблемы, а также строгий математический анализ с последующим предложенным решением.

Вы должны понимать, что «площадь под ROC-кривой» не равна «интересующей нас площади». Приведем пример, когда мы получили две совершенно различные ROC-кривые с одинаковым значением AUC-ROC.

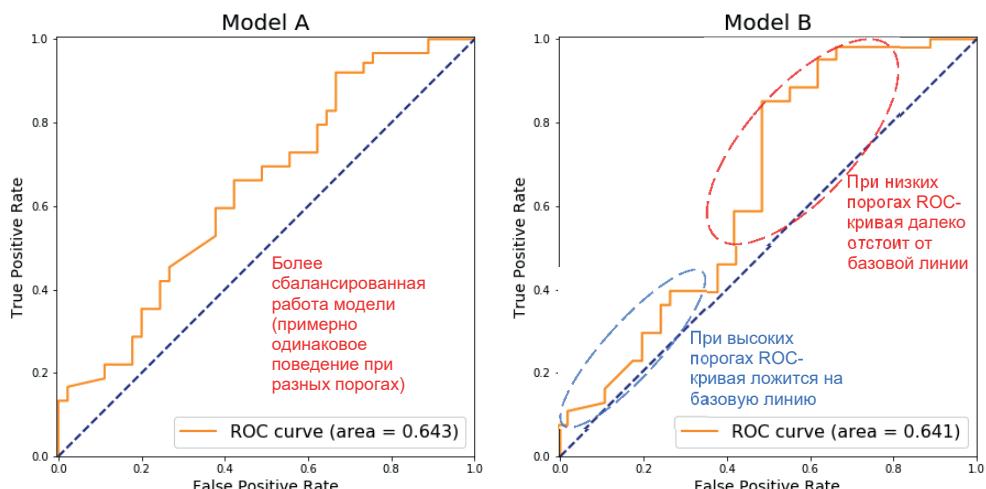


Рис. 47 Разные ROC-кривые с одинаковым значением AUC-ROC

Модель А лучше, чем модель В?

Обе модели имеют очень схожее значение AUC, но модель А является более последовательной с точки зрения соотношения доли истинно положительных

случаев и доли ложноположительных случаев (для всех порогов), в то время как для модели В соотношение между долей истинно положительных случаев и долей ложноположительных случаев сильно зависит от выбора порога – при высоких порогах ROC-кривая практически ложится на базовую линию, а при низких порогах ROC-кривая далеко отстоит от базовой линии.

В некоторых случаях минимизация доли ложноположительных случаев будет важнее, чем максимизация доли истинно положительных случаев, а в некоторых ситуациях потребуется обратное. Все зависит от того, как будет использоваться наша модель.

При вычислении AUC доля ложноположительных случаев и доля истинно положительных случаев получают одинаковые веса, и данный факт не позволяет нам выбрать модель, которая соответствует нашей конкретной ситуации.

Приведем еще пример, смотрите рисунок ниже.

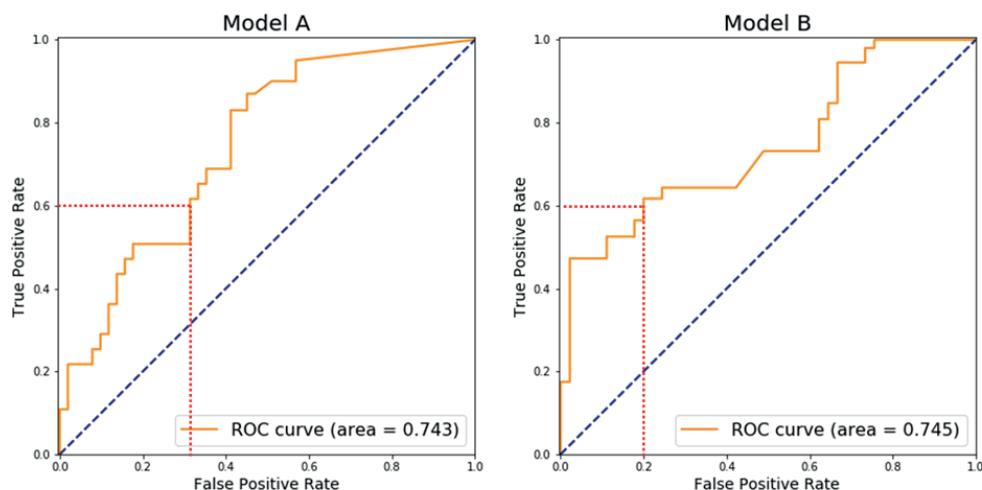


Рис. 48 Еще один пример разных ROC-кривых с одинаковым значением AUC-ROC

Какая модель лучше, А или В?

Это зависит от нашей предметной области и способа использования модели.

Допустим, мы анализируем ROC-кривую для модели А. Если мы решим, что у нас доля истинно положительных случаев должна быть равна, по крайней мере 60 %, мы должны будем признать, что у модели также будет доля ложноположительных случаев 30 %.

Теперь анализируем ROC-кривую для модели В. Мы можем достичь доли истинно положительных случаев, по меньшей мере, 60 % и доли ложноположительных случаев 20 %.

Если минимизация доли ложноположительных случаев является более важной задачей, то модель В предпочтительнее, чем модель А, даже если они имеют очень схожее значение AUC.

На практике мы всегда, помимо оценки AUC, будем анализировать экономический эффект от внедрения модели. Для этого мы берем исторические данные, где есть информация о прибыльности клиента (у нас может быть положительная и отрицательная прибыльность), и нам нужно показать пре-

имущество новой модели над существующей. У нас есть отсечка, ниже которой даем кредиты, выше – не даем (выше балл – выше риск дефолта). Здесь работает логика: «если бы вы не выдали кредит людям, у которых скорбалл по старой модели ниже отсечки, а по новой – выше, то не потеряли бы на них X рублей». Выделяем клиентов, у которых скорбалл по новой модели выше отсечки, а по старой – ниже. Потом выделяем клиентов, у которых скорбалл по новой модели ниже отсечки, а по старой – выше. Сравниваем прибыльности от этих групп клиентов, получаем экономический эффект от внедрения модели.

AUC-ROC можно легко распространить на случай многоклассовой классификации. Начнем с подхода 'ovg', «один против остальных».

Допустим, у нас – три класса 'apple', 'pear' и 'orange'.

Обучаем три классификатора:

- бинарный классификатор 1: 'apple' против ['pear', 'orange'], наблюдения с классом 'apple' считаем положительным классом, а наблюдения с классами 'pear', 'orange' считаем отрицательным классом;
- бинарный классификатор 2: 'pear' против ['apple', 'orange'], наблюдения с классом 'pear' считаем положительным классом, а наблюдения с классами 'apple', 'orange' считаем отрицательным классом;
- бинарный классификатор 3: 'orange' против ['pear', 'apple'], наблюдения с классом 'orange' считаем положительным классом, а наблюдения с классами 'pear', 'apple' считаем отрицательным классом.

Получаем три вероятности классов.

Вычисляем AUC-ROC для трех моделей и усредняем.

Этой схеме соответствует комбинация настроек `multi_class='ovr'` и `average='macro'` для функции `roc_auc_score()`.

Давайте убедимся в этом. Сначала импортируем необходимые классы и функции.

```
# импортируем необходимые классы и функции
from sklearn.preprocessing import LabelEncoder,
                                    label_binarize
from sklearn.linear_model import LogisticRegression
from itertools import combinations
```

Создаем игрушечные массив признаков и массив меток.

```
# создаем игрушечные данные
X_trn = np.array([[4.2, 1.5],
                  [1.4, 2.1],
                  [3.1, 0.5],
                  [1.3, 2.2],
                  [6.9, 4.5],
                  [7.9, 7.1]])

y_trn = np.array(['apple', 'pear', 'apple',
                  'orange', 'pear', 'apple'])
```

Строковые метки преобразовываем в целочисленные.

```
# строковые метки преобразовываем в целочисленные
le = LabelEncoder()
y_trn = le.fit_transform(y_trn)
y_trn

array([0, 2, 0, 1, 2, 0])
```

Теперь создаем и обучаем модель логистической регрессии.

```
# создаем и обучаем модель логистической регрессии
logreg = LogisticRegression()
logreg.fit(X_trn, y_trn);
```

Получаем вероятности трех классов.

```
# получаем вероятности классов
proba = logreg.predict_proba(X_trn)
proba

array([[0.68374283, 0.05426589, 0.26199129],
       [0.21630564, 0.39917286, 0.3845215 ],
       [0.63632408, 0.09994339, 0.26373253],
       [0.19683739, 0.42100499, 0.38215762],
       [0.70556819, 0.01338598, 0.28104583],
       [0.56121542, 0.01220895, 0.42657563]])
```

Вычисляем AUC-ROC с комбинацией настроек `multi_class='ovr'` и `average='macro'`.

```
# вычисляем AUC-ROC по схеме one-vs-rest
# с average='macro'
logreg_roc_auc = roc_auc_score(
    y_trn,
    logreg.predict_proba(X_trn),
    multi_class='ovr', average='macro')
print("AUC-ROC: {:.3f}".format(logreg_roc_auc))
```

AUC-ROC: 0.764

Смотрим, что происходит под капотом.

Сначала получаем уникальные метки классов.

```
# получаем уникальные метки классов
classes = np.unique(y_trn)
classes

array([0, 1, 2])
```

Вычисляем количество классов.

```
# вычисляем количество классов
n_classes = len(classes)
n_classes
```

Выполняем бинаризацию массива меток. Каждый класс получает свой столбец.

```
# бинаризуем массив меток
y_trn_binarized = label_binarize(y_trn, classes=classes)
y_trn_binarized

array([[1, 0, 0],
       [0, 0, 1],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [1, 0, 0]])
```

Теперь мы создаем массив `auc_scores` из нулей с количеством элементов, равным количеству классов, в него будем записывать оценки AUC-ROC моделей.

```
# создаем массив auc_scores из нулей с количеством элементов,
# равным количеству классов, в него будем записывать
# оценки AUC-ROC моделей
auc_scores = np.zeros((n_classes,))
```

В цикле `for` для каждого класса записываем метки из столбца бинаризованного массива меток, соответствующего этому классу, и вероятности – из столбца массива вероятностей, соответствующего этому классу, вычисляем оценку AUC-ROC и записываем эту оценку в ранее созданный массив `auc_scores`.

```
# для каждого класса...
for c in range(n_classes):
    # записываем метки
    y_true_c = y_trn_binarized[:, c]
    print(f"фактические метки классов:{\n{y_true_c}}")
    # записываем вероятности
    y_score_c = proba[:, c]
    print(f"вероятности положительного класса (класса {c}):{\n{y_score_c}}")
    print("")
    # вычисляем AUC-ROC и записываем в массив auc_scores
    auc_scores[c] = roc_auc_score(y_true_c, y_score_c, sample_weight=None)
```

фактические метки классов:
[1 0 1 0 0 1]
вероятности положительного класса (класса 0):
[0.68374283 0.21630564 0.63632408 0.19683739 0.70556819 0.56121542]

фактические метки классов:
[0 0 0 1 0 0]
вероятности положительного класса (класса 1):
[0.05426589 0.39917286 0.09994339 0.42100499 0.01338598 0.01220895]

фактические метки классов:
[0 1 0 0 1 0]
вероятности положительного класса (класса 2):
[0.26199129 0.3845215 0.26373253 0.38215762 0.28104583 0.42657563]

Смотрим оценки AUC-ROC и усредняем.

```
# смотрим оценки AUC-ROC
print("оценки AUC-ROC: {auc_scores}")
# усредняем оценки
ovr_macro_average_auc_score = np.mean(auc_scores)
print("итоговая оценка AUC-ROC: {ovr_macro_average_auc_score: .3f}")

оценки AUC-ROC: [0.66666667 1.          0.625      ]
итоговая оценка AUC-ROC: 0.764
```

Теперь вычислим AUC-ROC с комбинацией настроек `multi_class='ovr'` и `average='weighted'`.

```
# вычисляем AUC-ROC по схеме one-vs-rest
# с average='weighted'
logreg_roc_auc = roc_auc_score(
    y_trn,
    logreg.predict_proba(X_trn),
    multi_class='ovr', average='weighted')
print("AUC-ROC: {:.3f}".format(logreg_roc_auc))

AUC-ROC: 0.708
```

Смотрим, что происходит под капотом.

Сначала вычисляем веса классов. Вес каждого класса вычисляется как количество наблюдений положительного класса в соответствующем столбце бинаризованного массива меток.

бинаризованный массив меток

класс 0	класс 1	класс 2
1	0	0
0	0	1
1	0	0
0	1	0
0	0	1
1	0	0

The diagram shows a 6x3 binary matrix. Red arrows point from the column headers 'класс 0', 'класс 1', and 'класс 2' to the numbers 3, 1, and 2 respectively at the bottom, indicating the count of positive observations for each class.

```
# вычисляем вес каждого класса как количество наблюдений
# положительного класса в соответствующем столбце
# бинаризованного массива меток
average_weight = np.sum(y_trn_binarized, axis=0)
print("веса классов: {average_weight}")
print("")

веса классов: [3 1 2]
```

Опять мы создаем массив `auc_scores` из нулей с количеством элементов, равным количеству классов, в него будем записывать оценки AUC-ROC моделей.

```
# создаем массив auc_scores из нулей с количеством элементов,
# равным количеству классов, в него будем записывать
# оценки AUC-ROC моделей
auc_scores = np.zeros((n_classes,))
```

Вновь в цикле `for` для каждого класса записываем метки из столбца бинаризованного массива меток, соответствующего этому классу, и вероятности – из столбца массива вероятностей, соответствующего этому классу, вычисляем оценку AUC-ROC и записываем эту оценку в ранее созданный массив `auc_scores`.

```
# для каждого класса...
for c in range(n_classes):
    # записываем метки
    y_true_c = y_trn_binarized[:, c]
    print(f"фактические метки классов:{\n{y_true_c}}")
    # записываем вероятности
    y_score_c = proba[:, c]
    print(f"вероятности положительного класса (класса {c}):{\n{y_score_c}}")
    print("")
    # вычисляем AUC-ROC и записываем в массив auc_scores
    auc_scores[c] = roc_auc_score(y_true_c, y_score_c, sample_weight=None)
```

фактические метки классов:

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

вероятности положительного класса (класса 0):

```
[0.68374283 0.21630564 0.63632408 0.19683739 0.70556819 0.56121542]
```

фактические метки классов:

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

вероятности положительного класса (класса 1):

```
[0.05426589 0.39917286 0.09994339 0.42100499 0.01338598 0.01220895]
```

фактические метки классов:

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

вероятности положительного класса (класса 2):

```
[0.26199129 0.3845215 0.26373253 0.38215762 0.28104583 0.42657563]
```

Смотрим оценки AUC-ROC и усредняем с учетом весов классов.

```
# смотрим оценки AUC-ROC
print(f"оценки AUC-ROC: {auc_scores}")
# усредняем оценки с учетом весов классов
ovr_weighted_average_auc_score = np.average(auc_scores,
                                              weights=average_weight)
print(f"итоговая оценка AUC-ROC: {ovr_weighted_average_auc_score: .3f}")

оценки AUC-ROC: [0.66666667 1.          0.625      ]
итоговая оценка AUC-ROC: 0.708
```

Ниже приведен программный код, который подробно иллюстрирует происходящее под капотом функции `np.average(auc_scores, weights=average_weight)`.

```
# под капотом pr.average(auc_scores, weights=average_weight)
# происходит следующее
ovr_weighted_average_auc_score_alter = (
    average_weight[0] * auc_scores[0] +
    average_weight[1] * auc_scores[1] +
    average_weight[2] * auc_scores[2]) / sum(average_weight)
print(f"итоговая оценка AUC-ROC: {ovr_weighted_average_auc_score_alter: .3f}")

итоговая оценка AUC-ROC: 0.708
```

Теперь зададим веса наблюдений и опять вычислим AUC-ROC с комбинацией настроек `multi_class='ovr'` и `average='weighted'`.

```
# задаем веса наблюдений
obs_weights = [1, 2, 1, 2, 1, 3]

# вычисляем AUC-ROC по схеме one-vs-rest
# с average='weighted' и весами наблюдений
logreg_roc_auc = roc_auc_score(
    y_trn,
    logreg.predict_proba(X_trn),
    multi_class='ovr', average='weighted',
    sample_weight=obs_weights)
print("AUC-ROC: {:.3f}".format(logreg_roc_auc))

AUC-ROC: 0.743
```

Смотрим, что происходит под капотом.

Сначала меняем форму массива весов.

```
# меняем форму массива весов
obs_weights_reshaped = np.reshape(obs_weights, (-1, 1))
obs_weights_reshaped

array([[1],
       [2],
       [1],
       [2],
       [1],
       [3]])
```

Умножаем значения бинаризованного массива меток на веса и получаем новый массив.

```
# умножаем значения бинаризованного массива меток на веса
obs_weights_reshaped = np.multiply(y_trn_binarized,
                                    obs_weights_reshaped)
obs_weights_reshaped

array([[1, 0, 0],
       [0, 0, 2],
       [1, 0, 0],
       [0, 2, 0],
       [0, 0, 1],
       [3, 0, 0]])
```

Теперь вычисляем вес каждого класса как сумму взвешенных наблюдений положительного класса в соответствующем столбце нового массива.

```
# вычисляем вес каждого класса как сумму взвешенных
# наблюдений положительного класса
# в соответствующем столбце
average_weight = np.sum(obs_weights_reshaped, axis=0)
print(f"веса классов: {average_weight}")

веса классов: [5 2 3]
```

Создаем массив `auc_scores` из нулей с количеством элементов, равным количеству классов, в него будем записывать оценки AUC-ROC моделей.

```
# создаем массив auc_scores из нулей с количеством элементов,
# равным количеству классов, в него будем записывать
# оценки AUC-ROC моделей
auc_scores = np.zeros((n_classes,))
```

Вновь в цикле `for` для каждого класса записываем метки из столбца бинаризованного массива меток, соответствующего этому классу, и вероятности – из столбца массива вероятностей, соответствующего этому классу, вычисляем оценку AUC-ROC и записываем эту оценку в ранее созданный массив `auc_scores`.

```
# для каждого класса...
for c in range(n_classes):
    # записываем метки
    y_true_c = y_trn_binarized[:, c]
    print(f"фактические метки классов:\n{y_true_c}")
    # записываем вероятности
    y_score_c = proba[:, c]
    print(f"вероятности положительного класса (класса {c}): \n{y_score_c}")
    print("")
    # вычисляем AUC-ROC и записываем в массив auc_scores
    auc_scores[c] = roc_auc_score(
        y_true_c, y_score_c, sample_weight=obs_weights)
```

фактические метки классов:

[1 0 1 0 0 1]

вероятности положительного класса (класса 0):

[0.68374283 0.21630564 0.63632408 0.19683739 0.70556819 0.56121542]

фактические метки классов:

[0 0 0 1 0 0]

вероятности положительного класса (класса 1):

[0.05426589 0.39917286 0.09994339 0.42100499 0.01338598 0.01220895]

фактические метки классов:

[0 1 0 0 1 0]

вероятности положительного класса (класса 2):

[0.26199129 0.3845215 0.26373253 0.38215762 0.28104583 0.42657563]

Смотрим оценки AUC-ROC и усредняем с учетом весов классов.

```
# смотрим оценки AUC-ROC
print(f"оценки AUC-ROC: {auc_scores}")
# усредняем оценки с учетом весов классов
ovr_cust_weighted_average_auc_score = np.average(
    auc_scores, weights=average_weight)
print(f"итоговая оценка AUC-ROC: {ovr_cust_weighted_average_auc_score: .3f}")

оценки AUC: [0.8           1.           0.47619048]
итоговая оценка AUC:  0.743
```

Теперь посмотрим, как вычислить AUC-ROC в рамках подхода 'ovo', «один против одного».

В рамках подхода «один против одного» («one versus one» или ovo) мы решаем $n_{\text{classes}} \times (n_{\text{classes}} - 1)/2$ задач, в каждой отделяем i -й класс от j -го.

Допустим, у нас – три класса 'apple', 'pear' и 'orange'.

Обучаем три классификатора:

- бинарный классификатор 1: 'apple' против 'orange';
- бинарный классификатор 2: 'apple' против 'pear';
- бинарный классификатор 3: 'orange' против 'pear'.

Получаем три вероятности классов.

Затем мы создаем пары 'apple' vs 'orange', 'apple' vs 'pear', 'orange' vs 'pear'. Берем пару 'apple' vs 'orange'. Класс 'apple' объявляем положительным классом, а класс 'orange' – отрицательным классом. Для этого берем метки для пары 'apple' vs 'orange', все метки класса 'apple' получают значение 1 (или True), а все метки класса 'orange' получают значение 0 (или False). Берем эти метки, вероятности для класса 'apple' и вычисляем AUC-ROC. Теперь класс 'apple' объявляем отрицательным классом, а класс 'orange' – положительным классом. Для этого берем метки для пары 'apple' vs 'orange', все метки класса 'apple' получают значение 0 (или False), а все метки класса 'orange' получают значение 1 (или True). Берем эти метки, вероятности для класса 'orange' и вычисляем AUC-ROC. Затем берем эти две оценки AUC-ROC и усредняем, получаем среднее значение AUC-ROC для пары 'apple' vs 'orange'.

Давайте это поясним визуально на примере пары 'apple' vs 'orange'.

Помним, что наши строковые метки мы превратили в целочисленные.

y_trn
apple
pear
apple
orange
pear
apple

→

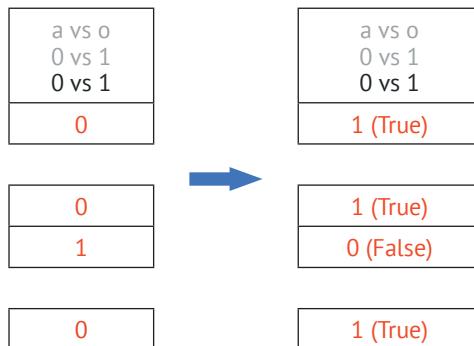
y_trn
0
2
0
1
2
0

У нас – три пары.

ind	X_trn		y_trn	a vs o 0 vs 1 0 vs 1	a vs p 0 vs 2 0 vs 1	o vs p 1 vs 2 0 vs 1
0	4.2	1.5	0	0	0	
1	1.4	2.1	2		1	1
2	3.1	0.5	0	0	0	
3	1.3	2.2	1		1	0
4	6.9	4.5	2		1	1
5	7.9	7.1	0	0	0	

Берем пару 'apple' vs 'orange'.

Класс 'apple' объявляем положительным классом, а класс 'orange' – отрицательным классом. Все метки, относящиеся к 'apple', кодируем 1 (True), а все метки, относящиеся к 'orange', кодируем 0 (False).

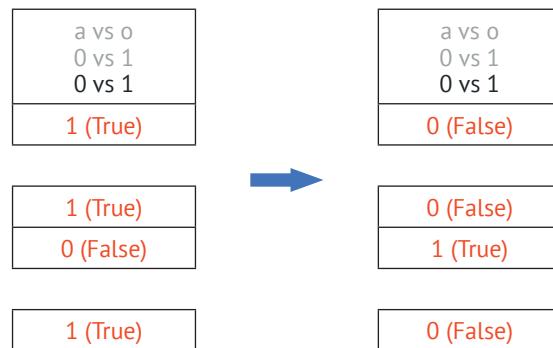


Из массива вероятностей берем столбец с вероятностями для класса 'apple' (столбец с индексом 0).

```
apple      orange      pear
 0          1          2
array([[0.68374283, 0.05426589, 0.26199129],
       [0.21630564, 0.39917286, 0.3845215 ],
       [0.63632408, 0.09994339, 0.26373253],
       [0.19683739, 0.42100499, 0.38215762],
       [0.70556819, 0.01338598, 0.28104583],
       [0.56121542, 0.01220895, 0.42657563]])
```

На основе полученных меток и вероятностей вычисляем AUC-ROC. Допустим, он равен 1.

Теперь класс 'apple' объявляем отрицательным классом, а класс 'orange' – положительным классом. Все метки, относящиеся к 'apple', кодируем 0 (False), а все метки, относящиеся к 'orange', кодируем 1 (True).



Из массива вероятностей берем столбец с вероятностями для класса 'orange' (столбец с индексом 1).

```
apple      orange      pear
0          1           2
array([[0.68374283, 0.05426589, 0.26199129],
       [0.21630564, 0.39917286, 0.3845215 ],
       [0.63632408, 0.09994339, 0.26373253],
       [0.19683739, 0.42100499, 0.38215762],
       [0.70556819, 0.01338598, 0.28104583],
       [0.56121542, 0.01220895, 0.42657563]])
```

На основе полученных меток и вероятностей вычисляем AUC-ROC. Допустим, он равен 1.

У нас две оценки AUC-ROC, равные 1, усредняем их и получаем среднее значение AUC-ROC для пары 'apple' vs 'orange', равное 1.

Аналогичную процедуру повторяем для остальных пар. В итоге получим среднее значение AUC-ROC для каждой пары. Три средних значения AUC-ROC усредняем и получаем итоговую оценку AUC-ROC.

Давайте вычислим AUC-ROC с комбинацией настроек `multi_class='ovo'` и `average='macro'`.

```
# вычисляем AUC-ROC по схеме one-vs-one
# с average='macro'
logreg_roc_auc = roc_auc_score(
    y_trn,
    logreg.predict_proba(X_trn),
    multi_class='ovo', average='macro')
print("AUC: {:.3f}".format(logreg_roc_auc))
```

AUC: 0.778

Смотрим, что происходит под капотом.

Сначала записываем количество пар классов согласно формуле `n_classes*(n_classes-1)/2`.

```
# записываем количество пар классов
n_pairs = n_classes * (n_classes - 1) // 2
n_pairs
```

Создаем массив `pair_auc_scores` из нулей с количеством элементов, равным количеству пар, в него будем записывать средние оценки AUC-ROC для пар.

```
# создаем массив pair_auc_scores из нулей с количеством
# элементов, равным количеству пар, в него будем
# записывать средние оценки AUC-ROC для пар
pair_auc_scores = np.zeros(n_pairs)
```

С помощью цикла `for` вычисляем среднее значение AUC-ROC для каждой пары классов вида А-В. Сначала вычисляем AUC-ROC, объявив класс А положительным, а класс В – отрицательным. Затем вычисляем AUC-ROC, объявив класс В положительным, а класс А – отрицательным.

```
# создаем массив pair_auc_scores из нулей с количеством
# элементов, равным количеству пар, в него будем
# записывать средние оценки AUC-ROC для пар
pair_auc_scores = np.zeros(n_pairs)

# вычисляем среднее значение AUC-ROC
# для каждой пары классов вида A-B,
# сначала вычисляем AUC-ROC, объявив класс A
# положительным, а класс B - отрицательным,
# затем вычисляем AUC-ROC, объявив класс B
# положительным, а класс A - отрицательным
for ix, (a, b) in enumerate(combinations(classes, 2)):
    a_mask = y_trn == a
    b_mask = y_trn == b
    ab_mask = np.logical_or(a_mask, b_mask)

    a_true = a_mask[ab_mask]
    b_true = b_mask[ab_mask]

    print(f"сравниваем класс {a} с классом {b}")
    a_true_auc_score = roc_auc_score(a_true, proba[ab_mask, a])
    print(f"метки классов, когда класс {a} -\n"
          f"положительный класс:\n{a_true}")
    print(f"вероятности для класса {a}:\n{proba[ab_mask, a]}")
    print(f"AUC-ROC, когда класс {a} - положительный класс: {a_true_auc_score}")
    print("")
    b_true_auc_score = roc_auc_score(b_true, proba[ab_mask, b])
    print(f"метки классов, когда класс {b} -\n"
          f"положительный класс:\n{b_true}")
    print(f"вероятности для класса {b}:\n{proba[ab_mask, b]}")
    print(f"AUC-ROC, когда класс {b} - положительный класс: {b_true_auc_score}\n")
    pair_auc_scores[ix] = (a_true_auc_score + b_true_auc_score) / 2
    print(f"усредненная оценка AUC-ROC для пары: {pair_auc_scores[ix]}")
    print("-----")
```

сравниваем класс 0 с классом 1
метки классов, когда класс 0 -
положительный класс:
[True True False True]
вероятности для класса 0:
[0.68374283 0.63632408 0.19683739 0.56121542]
AUC-ROC, когда класс 0 - положительный класс: 1.0

метки классов, когда класс 1 -
положительный класс:
[False False True False]
вероятности для класса 1:
[0.05426589 0.09994339 0.42100499 0.01220895]
AUC-ROC, когда класс 1 - положительный класс: 1.0

усредненная оценка AUC-ROC для пары: 1.0

сравниваем класс 0 с классом 2
метки классов, когда класс 0 -
положительный класс:
[True False True False True]
вероятности для класса 0:
[0.68374283 0.21630564 0.63632408 0.70556819 0.56121542]
AUC-ROC, когда класс 0 - положительный класс: 0.5

метки классов, когда класс 2 -
положительный класс:
[False True False True False]
вероятности для класса 2:
[0.26199129 0.3845215 0.26373253 0.28104583 0.42657563]
AUC-ROC, когда класс 2 - положительный класс: 0.6666666666666667

усредненная оценка AUC-ROC для пары: 0.583333333333334

сравниваем класс 1 с классом 2
метки классов, когда класс 1 -
положительный класс:
[False True False]
вероятности для класса 1:
[0.39917286 0.42100499 0.01338598]
AUC-ROC, когда класс 1 - положительный класс: 1.0

метки классов, когда класс 2 -
положительный класс:
[True False True]
вероятности для класса 2:
[0.3845215 0.38215762 0.28104583]
AUC-ROC, когда класс 2 - положительный класс: 0.5

усредненная оценка AUC-ROC для пары: 0.75

Давайте взглянем на массив со средними оценками AUC-ROC для пар и усредним эти средние оценки.

```
# смотрим массив со средними оценками AUC-ROC для пар
print("усредненные оценки AUC-ROC для пар: {pair_auc_scores}")
# усредняем средние оценки AUC-ROC для пар
ovo_macro_average_auc_score = np.average(pair_auc_scores,
                                           weights=None)
print(f"итоговая оценка AUC-ROC: {ovo_macro_average_auc_score: .3f}")
```

усредненные оценки AUC-ROC для пар: [1. 0.58333333 0.75]
итоговая оценка AUC-ROC: 0.778

Давайте вычислим AUC-ROC с комбинацией настроек `multi_class='ovo'` и `average='weighted'`.

```
# вычисляем AUC-ROC по схеме one-vs-one
# с average='weighted'
logreg_roc_auc = roc_auc_score(
    y_trn,
    logreg.predict_proba(X_trn),
    multi_class='ovo', average='weighted')
print("AUC: {:.3f}".format(logreg_roc_auc))
```

AUC: 0.764

Смотрим, что происходит под капотом.

Нам нужно получить веса пар. Для каждой пары сравниваемых классов мы все метки, относящиеся к сравниваемым классам, кодируем 1 (`True`), а все метки, относящиеся к другим классам, кодируем 0 (`False`). Затем подсчитываем сумму значений 1 по столбцу и делим на количество наблюдений.

The diagram illustrates the transformation of the target variable `y_trn` from categorical labels to binary weights. On the left, a vertical list of labels includes "apple", "pear", "apple", "orange", "pear", and "apple". An arrow points to the right, where a vertical list of binary weights shows 0, 2, 0, 1, 2, and 0 respectively. This represents the mapping where each class in a pair is assigned a weight of 1, while others are assigned 0.

<code>y_trn</code>
apple
pear
apple
orange
pear
apple

<code>y_trn</code>
0
2
0
1
2
0

<code>a vs o</code>	<code>a vs p</code>	<code>o vs p</code>
1	1	0
0	1	1
1	1	0
1	0	1
0	1	1
1	1	0

$4/6 = 0,67$ $5/6 = 0,83$ $3/6 = 0,5$

Когда мы вычислим средние значения AUC для каждой пары классов, мы усредним их с учетом полученных весов пар. Давайте посмотрим все это в программном коде.

```
# создаем массив prevalence с количеством элементов,
# равным количеству классов, в него будем
# записывать веса пар
prevalence = np.zeros(n_pairs)

# создаем массив pair_auc_scores из нулей с количеством
# элементов, равным количеству пар, в него будем
# записывать средние оценки AUC-ROC для пар
pair_auc_scores = np.zeros(n_pairs)
```

```

# вычисляем среднее значение AUC-ROC
# для каждой пары классов вида A-B,
# сначала вычисляем AUC-ROC, объявив класс A
# положительным, а класс B - отрицательным,
# затем вычисляем AUC-ROC, объявив класс B
# положительным, а класс A - отрицательным
for ix, (a, b) in enumerate(combinations(classes, 2)):
    a_mask = y_trn == a
    b_mask = y_trn == b
    ab_mask = np.logical_or(a_mask, b_mask)

    # получаем вес класса, вес - это среднее, взятое
    # по положительным наблюдениям (наблюдениям с меткой 1)
    # в соответствующем столбце бинаризованного массива меток,
    # к положительным наблюдениям причисляем наблюдения,
    # принадлежащие сравниваемым классам в паре,
    # к отрицательным - наблюдения остальных классов
    prevalence[ix] = np.average(ab_mask)

a_true = a_mask[ab_mask]
b_true = b_mask[ab_mask]

print(f"сравниваем класс {a} с классом {b}\n")
print(f"вес пары {a} vs {b}: {prevalence[ix]}\n")
a_true_auc_score = goc_auc_score(a_true, proba[ab_mask, a])
print(f"метки классов, когда класс {a} -\n"
      f"положительный класс:\n{a_true}")
print(f"вероятности для класса {a}:\n{proba[ab_mask, a]}")
print(f"AUC-ROC, когда класс {a} - положительный класс: {a_true_auc_score}")
print("")
b_true_auc_score = goc_auc_score(b_true, proba[ab_mask, b])
print(f"метки классов, когда класс {b} -\n"
      f"положительный класс:\n{b_true}")
print(f"вероятности для класса {b}:\n{proba[ab_mask, b]}")
print(f"AUC-ROC, когда класс {b} - положительный класс: {b_true_auc_score}\n")
pair_auc_scores[ix] = (a_true_auc_score + b_true_auc_score) / 2
print(f"усредненная оценка AUC-ROC для пары: {pair_auc_scores[ix]}")
print("-----")

```

сравниваем класс 0 с классом 1
 вес пары 0 vs 1: 0.6666666666666666

метки классов, когда класс 0 -
 положительный класс:
 [True True False True]
 вероятности для класса 0:
 [0.68374283 0.63632408 0.19683739 0.56121542]
 AUC-ROC, когда класс 0 - положительный класс: 1.0

метки классов, когда класс 1 -
 положительный класс:
 [False False True False]
 вероятности для класса 1:
 [0.05426589 0.09994339 0.42100499 0.01220895]
 AUC-ROC, когда класс 1 - положительный класс: 1.0

усредненная оценка AUC-ROC для пары: 1.0

сравниваем класс 0 с классом 2
вес пары 0 vs 2: 0.83333333333334

метки классов, когда класс 0 -
положительный класс:

[True False True False True]

вероятности для класса 0:

[0.68374283 0.21630564 0.63632408 0.70556819 0.56121542]

AUC-ROC, когда класс 0 - положительный класс: 0.5

метки классов, когда класс 2 -
положительный класс:

[False True False True False]

вероятности для класса 2:

[0.26199129 0.3845215 0.26373253 0.28104583 0.42657563]

AUC-ROC, когда класс 2 - положительный класс: 0.6666666666666667

усредненная оценка AUC-ROC для пары: 0.5833333333333334

сравниваем класс 1 с классом 2
вес пары 1 vs 2: 0.5

метки классов, когда класс 1 -
положительный класс:

[False True False]

вероятности для класса 1:

[0.39917286 0.42100499 0.01338598]

AUC-ROC, когда класс 1 - положительный класс: 1.0

метки классов, когда класс 2 -
положительный класс:

[True False True]

вероятности для класса 2:

[0.3845215 0.38215762 0.28104583]

AUC-ROC, когда класс 2 - положительный класс: 0.5

усредненная оценка AUC-ROC для пары: 0.75

смотрим веса пар

print(f"веса пар: {prevalence}")

смотрим массив со средними оценками AUC-ROC для пар

print(f"усредненные оценки AUC-ROC для пар: {pair_auc_scores}")

усредняем средние оценки AUC-ROC для пар

ovo_weighted_average_auc_score = np.average(pair_auc_scores,
weights=prevalence)

print(f"итоговая оценка AUC-ROC: {ovo_weighted_average_auc_score: .3f}")

веса пар: [0.66666667 0.83333333 0.5]

усредненные оценки AUC-ROC для пар: [1. 0.58333333 0.75]

итоговая оценка AUC-ROC: 0.764

1.15. PR-КРИВАЯ (PR CURVE) И ПЛОЩАДЬ ПОД PR-КРИВОЙ (AUC-PR)

По аналогии с ROC-кривой (кривой чувствительности и 1 – специфичности) можно построить PR-кривую. PR-кривая – это кривая соотношений точности и полноты для различных пороговых значений спрогнозированной вероятности интересующего класса. При построении ROC-кривой мы по оси x откладывали 1 – специфичность, а по оси y – чувствительность. При построении PR-кривой по оси x откладывают полноту, а по оси y – точность.

Мы помним: чем ближе ROC-кривая подходит к верхнему левому углу, тем лучше классификатор. С PR-кривой все обстоит иначе. Чем ближе PR-кривая подходит к верхнему правому углу, тем лучше классификатор. Чем больше модель сохраняет высокое значение точности при одновременном увеличении полноты, тем лучше. Мы также можем вычислить площадь под PR-кривой, этот показатель называют AUC-PR или average precision (средняя точность, AP).

Пример кривой точности–полноты



Рис. 49 Кривая точности–полноты

Ранее мы говорили, что базовая линия ROC-кривой для любого классификатора лежит на отрезке прямой с концами $(0; 0)$ и $(1; 1)$, то есть является диагональю квадрата координатной плоскости. Базовая линия соответствует бесполезному классификатору, который ранжирует объекты случайным образом. Поэтому AUC-ROC для любого бесполезного классификатора всегда будет равна 0,5. Для PR-кривой базовая линия также имеет форму прямой, однако положение этой линии зависит от дисбаланса выборки: базовая линия PR-кривой проходит через точки $(0; d)$ и $(1; d)$, где d – это доля наблюдений миноритарного (положительного) класса в выборке или $P/(N + P)$. Это обусловлено тем, что бесполезный классификатор расставляет метки 0 и 1 случайным образом. Мы получаем вероятности 0,5, и при случайном присваивании меток наша матрица сводится к тому, что $TP = P$, $FP = N$, и поэтому формула Точность = $TP/(TP + FP)$ превращается в формулу Точность = $P/(P + N) = d$, а полнота принимает значение 1. Чем больше дисбаланс, тем ниже будет располагаться базовая линия. Таким образом, базовые линии PR-кривых у разных классификаторов не обязаны совпадать.

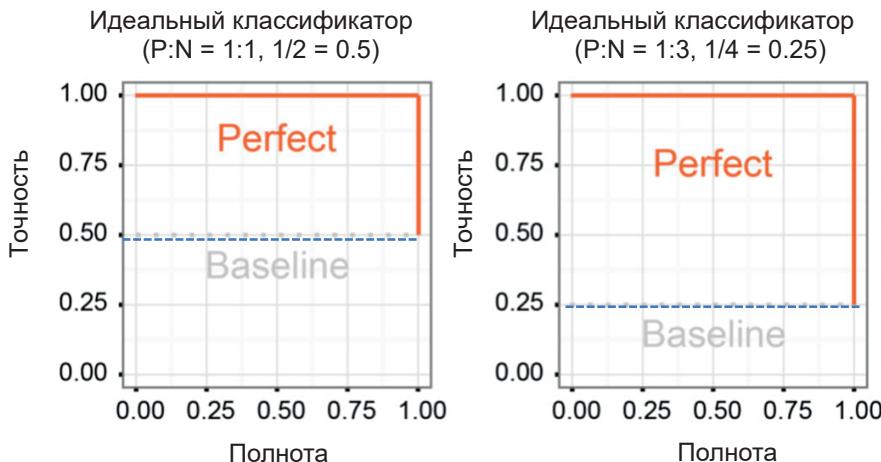


Рис. 50 Кривая точности–полноты с разными базовыми линиями из-за разного баланса классов

Допустим, базовая линия равна 0,2, т.е. проходит через значение оси у, равное 0,2. Если оценка AUC-PR модели равна 0,5, то модель работает лучше случайного угадывания. Если оценка AUC-PR модели равна 0,2, то модель работает на уровне случайного угадывания. Если оценка AUC-PR модели равна 0,1, то модель работает хуже случайного угадывания.

На собеседованиях часто спрашивают, какие координаты будет иметь последняя точка кривой точности–полноты. Здесь тоже нетрудно догадаться, что у нее будут координаты $(1; P/(N + P))$, даже не помня про то, как строится базовая линия для графика кривой точности–полноты. Когда мы перебираем пороги и, наконец, переходим к минимальному значению порога (крайняя правая точка на графике PR-кривой), мы все наблюдения классифицируем как наблюдения положительного класса, количество истинно положительных становится равным количеству наблюдений положительного класса. Таким образом, точность в минимальном пороге становится равной доле наблюдений миноритарного (положительного) класса в выборке. Опять можно показать: Точность = $TP/(TP + FP)$, для минимального порога $TP = P$, $FP = N$, получаем Точность = $P/(P + N)$.

Давайте напишем функцию `_precision_recall_curve()`, которая строит график кривой точности–полноты. При этом в ходе написания этой функции мы воспользуемся ранее написанной функцией `_binary_clf_curve()`.

```
# пишем функцию, которая будет строить PR-кривую
def _precision_recall_curve(y_true, y_score, sample_weight=None):
    """
    
```

Вычисляет пары значений точности и полноты для каждого порогового значения вероятности положительного класса.

Параметры

`y_true` : одномерный массив формы [n_samples]

Фактические метки (классы) зависимой переменной.

```

y_score : одномерный массив формы [n_samples]
Спрогнозированные вероятности положительного класса.

sample_weight : одномерный массив формы (n_samples,),
    по умолчанию None
Веса наблюдений.

Возвращает
-----
precision : одномерный массив
    Значения точности.

recall : одномерный массив
    Значения полноты (чувствительности).

thresholds : одномерный массив
    Пороговые значения спрогнозированной вероятности
    положительного класса, отсортированные по возрастанию.

"""
# вычисляем количество истинно положительных случаев
# для каждого порога, количество ложноположительных
# случаев для каждого порога, пороги
fps, tps, thresholds = _binary_clf_curve(
    y_true, y_score, sample_weight=sample_weight
)

# вычисляем общее количество предсказанных
# положительных случаев
ps = tps + fps
# вычисляем точность
precision = np.divide(tps, ps, where=(ps != 0))

# вычисляем полноту,
# когда в y_true нет меток положительного класса,
# полнота задается равной 1 для всех порогов
# tps[-1] == 0 <=> y_true == all negative labels
if tps[-1] == 0:
    warnings.warn(
        "Положительный класс не найден в y_true, "
        "recall приравнивается к 1 для всех порогов."
    )
    recall = np.ones_like(tps)
else:
    recall = tps / tps[-1]

# меняем порядок, потому что полнота уменьшается
sl = slice(None, None, -1)
return (np.hstack((precision[sl], 1)),
        np.hstack((recall[sl], 0)),
        thresholds[sl])

```

Теперь на данных StateFarm построим модель градиентного бустинга, вычислим вероятности положительного класса для тестовой выборки, а затем на основе массива фактических меток и вычисленных вероятностей с помощью нашей функции `precision_recall_curve()` построим PR-кривую.

```

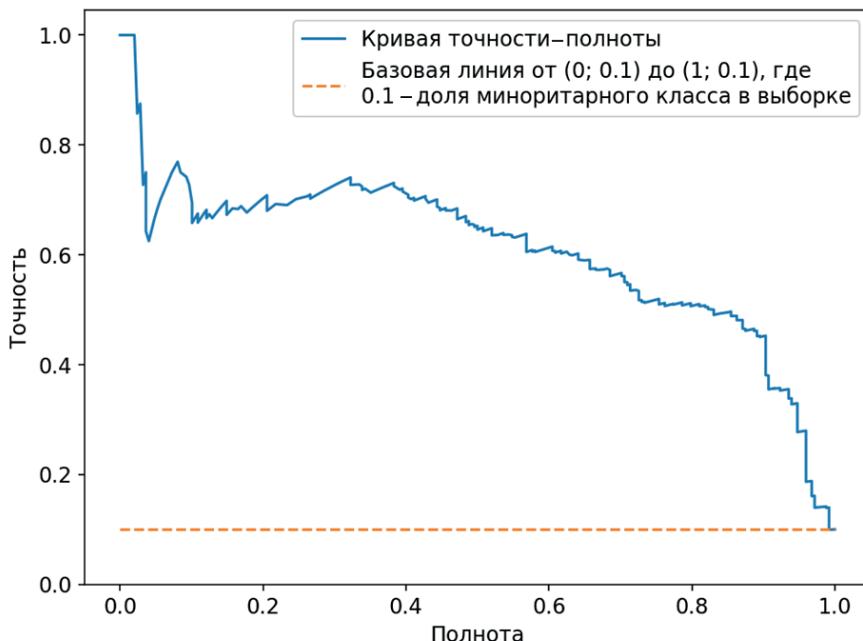
# создаем экземпляр класса GradientBoostingClassifier
boost = GradientBoostingClassifier(n_estimators=300,
                                    learning_rate=0.01,
                                    max_depth=8,
                                    random_state=42)

# обучаем модель
boost.fit(X_train, y_train)
# получаем вероятности положительного класса
# для тестовой выборки
proba = boost.predict_proba(X_test)[:, 1]

# вычисляем значения точности и полноты для всех возможных
# порогов отсечения, передав нашей функции _precision_recall_curve()
# в качестве аргументов фактические значения зависимой
# переменной и вероятности
precision, recall, thresholds = _precision_recall_curve(
    y_test, proba)

# строим график кривой точности-полноты
fig, ax = plt.subplots(figsize=(6, 6))
bsline = round(len(y_test[y_test == 1]) / len(y_test), 1)
label_text = f"Базовая линия от (0; {bsline}) до (1; {bsline}), где\n" \
            f"{bsline} - доля миноритарного класса в выборке"
ax.plot(recall, precision, label="Кривая точности-полноты")
ax.plot([0, 1], [bsline, bsline],
        linestyle="--",
        label=label_text)
ax.set_xlim(0)
ax.set_xlabel("Полнота")
ax.set_ylabel("Точность")
plt.legend(loc="best");

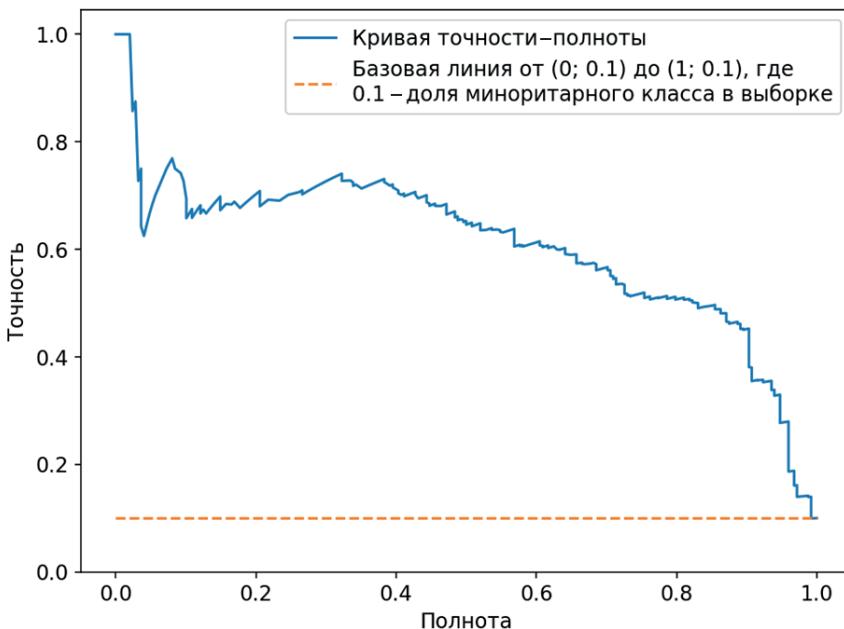
```



Теперь с помощью функции `precision_recall_curve()` библиотеки `scikit-learn` построим PR-кривую автоматически.

```
from sklearn.metrics import precision_recall_curve(y_true, ← Фактические классы зависимой переменной
probas_pred, ← Вероятности положительного
Metka положительного класса зависимой переменной. → класса зависимой переменной
Когда задано pos_label=None, а y_true принимает значения {-1, 1} или {0, 1}, то метка положительного sample_weight=None) ← Веса наблюдений
класса устанавливается равной 1, в противном случае
будет выдана ошибка
```

```
# из модуля sklearn.metrics импортируем
# функцию precision_recall_curve()
from sklearn.metrics import precision_recall_curve
# вычисляем значения точности и полноты для всех возможных
# порогов отсечения, передав функции precision_recall_curve()
# в качестве аргументов фактические значения зависимой
# переменной и вероятности
precision, recall, thresholds = precision_recall_curve(
    y_test, proba)
fig, ax = plt.subplots(figsize=(6, 6))
bsline = round(len(y_test[y_test == 1]) / len(y_test), 1)
label_text = (f"Базовая линия от (0; {bsline}) до (1; {bsline}), где\n"
              f"{bsline} - доля миноритарного класса в выборке")
ax.plot(recall, precision, label="Кривая точности-полноты")
ax.plot([0, 1], [bsline, bsline],
        linestyle="--",
        label=label_text)
ax.set_yticks([0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
ax.set_xlabel("Полнота")
ax.set_ylabel("Точность")
plt.legend(loc="best");
```



Обратите внимание, что базовая линия проходит через точки (0; 0.1) и (1; 0.1), где 0.1 – это доля наблюдений миноритарного класса в выборке. Давайте вычислим долю миноритарного класса в выборке.

```
# вычислим долю миноритарного класса в выборке  
y_test.value_counts(normalize=True)[1]
```

0.100040338846309

Теперь убедимся, что для бесполезного классификатора точность будет равна доле миноритарного класса в выборке, а формула Точность = $TP/(TP + FP)$ в матрице ошибок превращается в формулу Точность = $P/(P + N)$.

```
# импортируем класс DummyClassifier  
from sklearn.dummy import DummyClassifier  
dummy_clf = DummyClassifier(strategy='uniform',  
                             random_state=42)  
  
# обучаем бесполезную модель  
dummy_clf.fit(X_train, y_train)  
  
# вычисляем вероятности  
random_proba = dummy_clf.predict_proba(X_test)[:, 1]  
# для случайного угадывания вероятность положительного  
# класса будет равна 0.5, поэтому у нас будет  
# единственный порог 0.5  
threshold = 0.5  
  
# получаем прогнозы согласно порогу  
predictions = (random_proba >= threshold)  
  
# вычисляем матрицу ошибок  
confusion = confusion_matrix(y_test, predictions)  
# вычисляем чувствительность:  $TP/(TP+FN)$   
se = confusion[1][1] / (confusion[1][1] + confusion[1][0])  
# вычисляем точность  
p_rec = confusion[1][1] / (confusion[1][1] + confusion[0][1])  
# печатаем матрицу ошибок  
print(f"Матрица ошибок для порога {threshold}:\n{confusion}")  
# печатаем распределение классов  
neg = y_test.value_counts()[0]  
pos = y_test.value_counts()[1]  
print(f"Распределение классов: {neg}:{pos}")  
# печатаем значение чувствительности  
print(f"Чувствительность: {se:.3f}")  
# печатаем значение точности  
print(f"Точность: {p_rec:.3f}")  
  
Матрица ошибок для порога 0.5:  
[[ 0 2231]  
 [ 0 248]]  
Распределение классов: 2231:248  
Чувствительность: 1.000  
Точность: 0.100
```

Вернемся к нашей модели градиентного бустинга. Если мы посмотрим значение точности для минимального порога, мы получим то же самое значение и ту же самую матрицу ошибок.

```

# задаем минимальный порог
mythreshold = round(thresholds[0], 3)
# получаем прогнозы согласно порогу
predictions = (proba >= mythreshold)
# вычисляем матрицу ошибок
confusion = confusion_matrix(y_test, predictions)
# вычисляем чувствительность: TP/(TP+FN)
se = confusion[1][1] / (confusion[1][1] + confusion[1][0])
# вычисляем точность
prec = confusion[1][1] / (confusion[1][1] + confusion[0][1])
# печатаем матрицу ошибок
print(f"Матрица ошибок для порога {mythreshold}:\n{confusion}")
# печатаем распределение классов
neg = y_test.value_counts()[0]
pos = y_test.value_counts()[1]
print(f"Распределение классов: {neg}:{pos}")
# печатаем значение чувствительности
print(f"Чувствительность: {se:.3f}")
# печатаем значение точности
print(f"Точность: {prec:.3f}")

```

Матрица ошибок для порога 0.008:

```
[[ 0 2231]
 [ 0 248]]
```

Распределение классов: 2231:248

Чувствительность: 1.000

Точность: 0.100

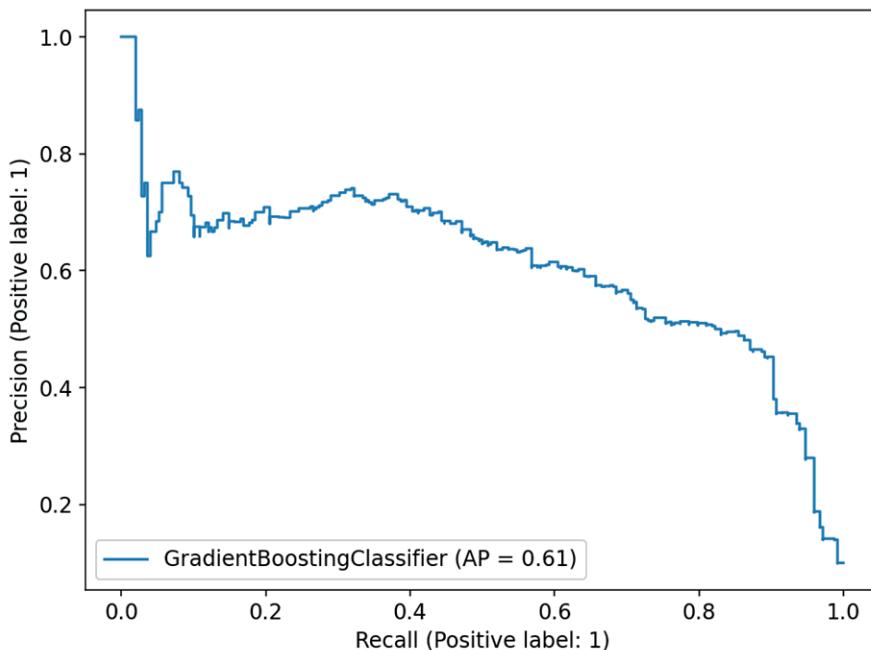
С помощью методов `.from_estimator()` и `.from_predictions()` класса `RocCurveDisplay` можно построить PR-кривую на основе модели и на основе прогнозов модели соответственно. PR-кривую можно построить без интерполяции в виде ступенчатого графика (строится по умолчанию для соответствия метрике AP) и с интерполяцией. Кроме построения PR-кривой выводится значение AUC-PR (или AP).

```

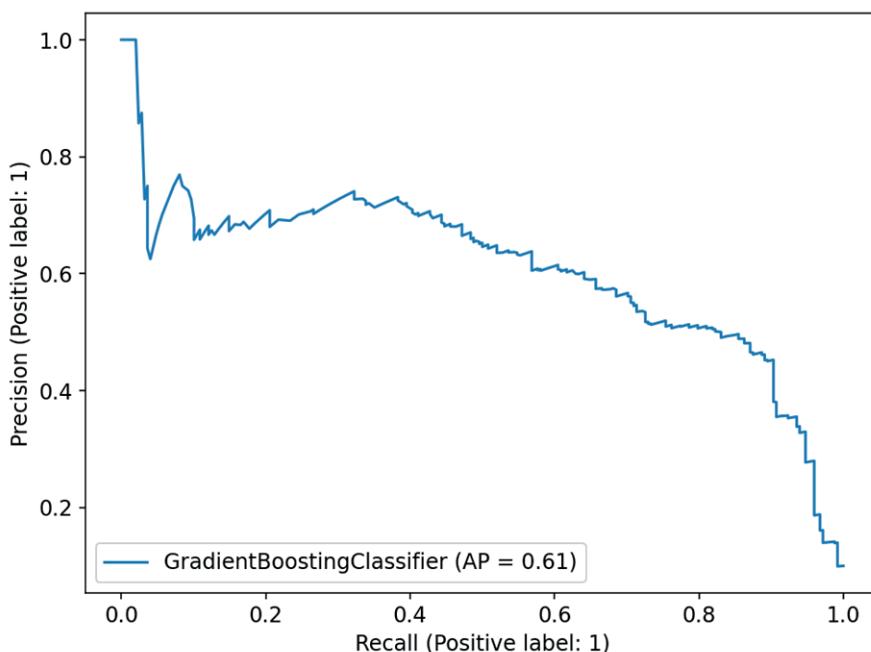
# импортируем класс PrecisionRecallDisplay
from sklearn.metrics import PrecisionRecallDisplay

# строим PR-кривую для модели по умолчанию (без интерполяции)
PrecisionRecallDisplay.from_estimator(boost, X_test, y_test);

```



```
# строим PR-кривую для модели с интерполяцией  
PrecisionRecallDisplay.from_estimator(boost, X_test, y_test,  
drawstyle='default');
```



На основе массива фактических меток и вычисленных вероятностей с помощью функции `average_precision_score()` вычислим AUC-PR.

```
# импортируем функцию average_precision_score()
from sklearn.metrics import average_precision_score
# вычисляем значение AUC-PR, передав функции
# average_precision_score() в качестве аргументов
# фактические значения зависимой переменной и вероятности
aucpr = average_precision_score(y_test, proba)
# печатаем AUC-PR
print("AUC-PR: {:.3f}".format(aucpr))
```

Как взаимосвязаны ROC-кривая и PR-кривая? Исследователи Джесси Дэвис и Марк Годрич в своей работе «The relationship between Precision-Recall and ROC curves» («Взаимосвязь между PR-кривыми и ROC-кривыми») вывели, что кривая доминирует в ROC-пространстве тогда и только тогда, когда она доминирует в PR-пространстве. Однако, как удалось показать исследователям, алгоритм, который оптимизирует AUC-ROC, не гарантирует оптимизацию AUC-PR.

Как уже было сказано выше, ROC-кривая, в отличие от PR-кривой, менее чувствительна к дисбалансу. Давайте рассмотрим поведение PR-кривых и ROC-кривых на несбалансированном наборе данных *creditcard.csv*. Он содержит информацию о 284 807 транзакциях по кредитным картам: 0 – легальная транзакция (284 315 легальных транзакций) и 1 – мошенническая транзакция (492 мошеннические транзакции). Переменные *V1*, *V2*, ... *V28* – это главные компоненты, полученные по результатам анализа главных компонент. Переменная *Time* – количество секунд, прошедшее от момента первого наблюдения (первой транзакции). Переменная *Amount* – это сумма транзакции. Переменная *Class* является зависимой и обозначает тип транзакции: 0 – легальная транзакция, 1 – мошенническая транзакция.

Итак, импортируем необходимые классы и загружаем данные.

```
# импортируем необходимые классы
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

# загружаем данные
df = pd.read_csv('Data/creditcard.csv')
df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.1285
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.1671
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.3276
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.6473
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.2060

В данных есть столбец *Time* (*Время*), в котором записано количество секунд, прошедшее от момента первого наблюдения. Создадим столбец *hour* (*час суток*) (начиная с полуночи).

```
# создаем новую переменную – час суток после полуночи
df['hour'] = df['Time'].apply(
    lambda x: np.ceil(float(x) / 3600) % 24)
```

612 ❖ Метрики для оценки качества модели

Взглянем на разбивку легальных/мошеннических транзакций по часам при помощи сводной таблицы.

```
# увеличиваем количество отображаемых столбцов
pd.options.display.max_columns = 30
# строим разбивку на легальные/мошеннические
# транзакции по часам
df.pivot_table(values='Amount',
               index='Class',
               columns='hour',
               aggfunc='count')
```

hour	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0	21.0	
Class	0	10919	7687	4212	3269	3476	2185	2979	4093	7219	10266	15824	16593	16804	15400	15350	16545	16434	16435	16135	17003	15632	16739
	1	21	6	10	57	17	23	11	9	23	9	16	8	53	17	17	23	26	22	29	33	19	18

Взглянем на процент мошеннических транзакций.

```
print("Мошеннические транзакции составляют {}% от наших данных.".format(
    df['Class'].value_counts(normalize=True)[1] * 100))
```

Мошеннические транзакции составляют 0.1727485630620034% от наших данных.

Давайте сформируем обучающий массив признаков, обучающий массив меток, тестовый массив признаков, тестовый массив меток, выполним стандартизацию и выведем информацию о распределении классов.

```
# формируем массивы признаков и массивы меток,
# выполняем предварительную подготовку данных
x_train, x_test, y_train, y_test = train_test_split(
    pd.concat([df.loc[:, 'V1': 'Amount'], df.loc[:, 'Time']], axis=1),
    df['Class'], stratify=df['Class'], test_size=0.35, random_state=1)

# выполним стандартизацию
scaler = StandardScaler()
scaler.fit(x_train[['Amount']])
x_train['Amount'] = scaler.transform(x_train[['Amount']])
x_test['Amount'] = scaler.transform(x_test[['Amount']])

# выведем информацию о распределении классов
print(f"Распределение классов в y_train:\n{y_train.value_counts()}")
print("")
print(f"Распределение классов в y_test:\n{y_test.value_counts()}")
```

Распределение классов в y_train:

```
0    184804
1      320
Name: Class, dtype: int64
```

Распределение классов в y_test:

```
0    99511
1     172
Name: Class, dtype: int64
```

Существует два распространенных подхода к решению проблемы крайне несбалансированных данных: присвоение весов и семпллинг.

Первый подход предлагает использовать в ходе обучения разные стоимости ошибочной классификации. Ошибкам отнесения к классам зависимой переменной мы можем назначить разные цены в зависимости от ценности класса. Например, предположим, что пациент относится к одному из двух классов: *Здоров* (отрицательный класс) и *Болен* (положительный класс). Ошибочное отнесение больного пациента к классу *Здоров*, вероятно, является ошибкой, имеющей более высокую цену, чем ошибочное отнесение здорового пациента к классу *Болен*. Итак, мы определяем, что стоимость ошибочной классификации наблюдений миноритарного класса выше, и пытаемся минимизировать общую стоимость ошибочной классификации.

Второй подход заключается в использовании семпллинга. Мы можем удалить некоторое количество примеров мажоритарного класса (данную технику называют андерсемплингом) или увеличить количество примеров миноритарного класса (эта техника называется оверсемплингом). Удалить примеры мажоритарного класса или увеличить количество примеров миноритарного класса можно случайным образом или по специальным правилам. Можно также сочетать андерсемплинг и оверсемплинг.

Обратите внимание, что присвоение весов и любой семпллинг, будь то андерсемплинг или оверсемплинг, случайный или по определенным правилам, мы осуществляем только на обучающей выборке, а затем смотрим, как модель, обученная по взвешенным/семплированным данным, работает на тестовых данных, не подвергшихся взвешиванию/семпллингу. Мы должны помнить, что при работе с новыми данными у нас не будет никакой зависимой переменной, и мы не можем вновь применить взвешивание/семпллинг, у нас есть лишь модель, построенная на взвешенных/семплированных обучающих данных. Семпллинг выполняется с помощью различных классов библиотеки *imbalanced-learn*. Следует помнить, что изменение пропорций классов с помощью весов и семпллинга – это самый последний инструмент в арсенале средств аналитика. Здесь эти две техники приведены только для иллюстрации поведения ROC-кривой и PR-кривой при разных весах миноритарного класса и разных пропорциях классов. Проблему дисбаланса классов нужно попытаться решить сперва за счет выбора метрики, максимально адекватной решаемой задаче, подбора порога для отнесения наблюдений к тому или иному классу, конструирования новых признаков, обогащения данными из других источников, настройки гиперпараметров и ансамблирования моделей. И только когда все традиционные способы улучшения качества модели уже испробованы, можно применить специальные техники устранения дисбаланса классов. Часто ни одна из техник взвешивания и семплирования не дает существенного прироста качества.

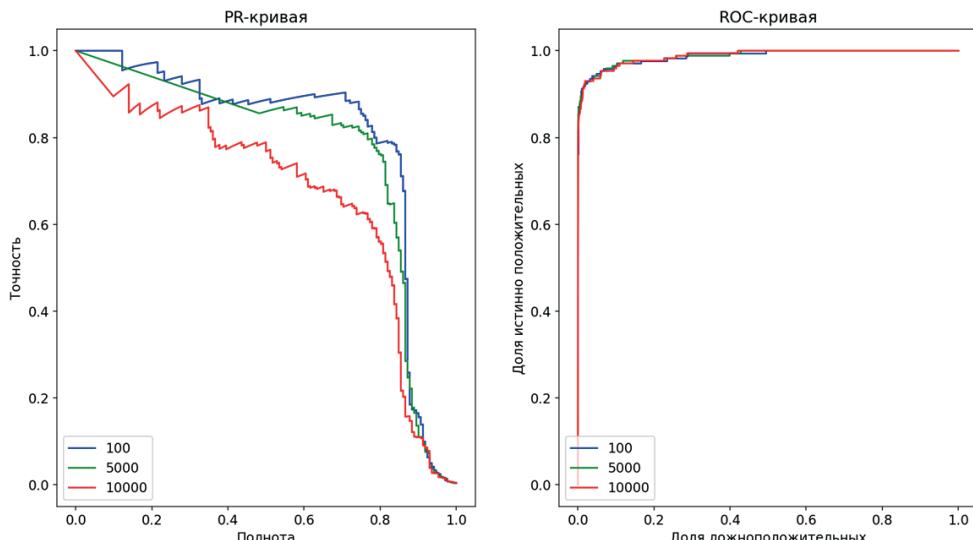
Теперь перейдем к построению модели. Мы воспользуемся логистической регрессией. Построим ROC-кривую и PR-кривую для различных значений веса миноритарного класса (мошеннических транзакций) и различных пропорций классов. Для изменения пропорций классов мы воспользуемся андерсемплингом при помощи класса *RandomUnderSampler*. Параметр *sampling_strategy* этого класса позволяет задавать желаемые пропорции классов в виде словаря.

```
# строим модели с разным весом миноритарного класса
fig = plt.figure(figsize=(15, 8))
ax1 = fig.add_subplot(1, 2, 1)
ax1.set_xlim([-0.05, 1.05])
ax1.set_ylim([-0.05, 1.05])
ax1.set_xlabel("Полнота")
ax1.set_ylabel("Точность")
ax1.set_title("PR-кривая")

ax2 = fig.add_subplot(1, 2, 2)
ax2.set_xlim([-0.05, 1.05])
ax2.set_ylim([-0.05, 1.05])
ax2.set_xlabel("Доля ложноположительных")
ax2.set_ylabel("Доля истинно положительных")
ax2.set_title("ROC-кривая")

for w, k in zip([100, 5000, 10000], 'bgr'):
    lr_model = LogisticRegression(class_weight={0:1, 1:w},
                                    solver='liblinear')
    lr_model.fit(x_train, y_train)
    pred_prob = lr_model.predict_proba(x_test)[:, 1]
    p, r, _ = precision_recall_curve(y_test, pred_prob)
    tpr, fpr, _ = roc_curve(y_test, pred_prob)
    ax1.plot(r, p, c=k, label=w)
    ax2.plot(tpr, fpr, c=k, label=w)
ax1.legend(loc='lower left')
ax2.legend(loc='lower left')

plt.show()
```



```
# импортируем класс RandomUnderSampler
from imblearn.under_sampling import RandomUnderSampler

# строим модели с разным распределением классов
fig = plt.figure(figsize=(15, 8))
ax1 = fig.add_subplot(1, 2, 1)
```

```

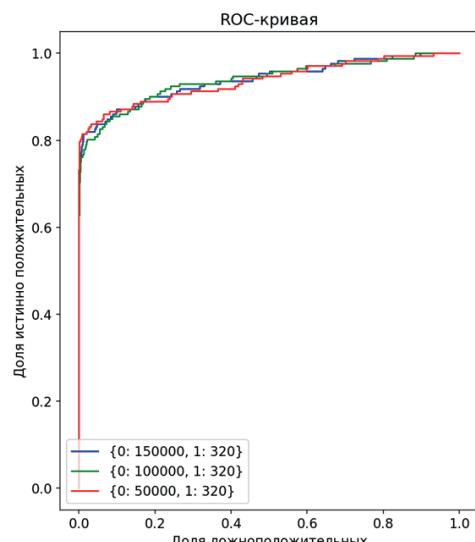
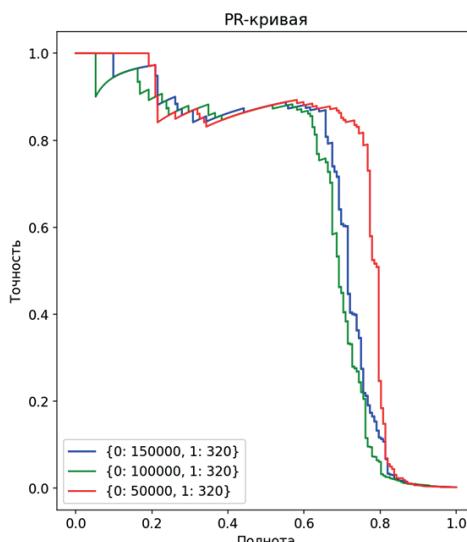
ax1.set_xlim([-0.05, 1.05])
ax1.set_ylim([-0.05, 1.05])
ax1.set_xlabel("Полнота")
ax1.set_ylabel("Точность")
ax1.set_title("PR-кривая")

ax2 = fig.add_subplot(1, 2, 2)
ax2.set_xlim([-0.05, 1.05])
ax2.set_ylim([-0.05, 1.05])
ax2.set_xlabel("Доля ложноположительных")
ax2.set_ylabel("Доля истинно положительных")
ax2.set_title("ROC-кривая")

for s, k in zip([{0:150000, 1:320}, {0:100000, 1:320}, {0:50000, 1:320}], 'bgr'):
    # создаем модель андерсемплинга
    rus = RandomUnderSampler(sampling_strategy=s,
                               random_state=42)
    # выполняем андерсемплинг обучающих данных
    # (проверочные или тестовые данные
    # в сэмплировании не участвуют!)
    x_train_resampled, y_train_resampled = rus.fit_resample(
        x_train, y_train)
    # создаем модель логистической регрессии
    lr_model = LogisticRegression(solver='liblinear')
    # обучаем модель логистической регрессии
    # сэмплированных данных
    lr_model.fit(x_train_resampled, y_train_resampled)
    pred_prob = lr_model.predict_proba(x_test)[:, 1]
    p, r, _ = precision_recall_curve(y_test, pred_prob)
    tpr, fpr, _ = roc_curve(y_test, pred_prob)
    ax1.plot(r, p, c=k, label=s)
    ax2.plot(tpr, fpr, c=k, label=s)
ax1.legend(loc='lower left')
ax2.legend(loc='lower left')

plt.show()

```



Напомним, для хорошего классификатора PR-кривая должна быть как можно ближе к правому верхнему краю графика, а для ROC-кривой – к левому верхнему.

Несмотря на то что PR-кривая и ROC-кривая используют одни и те же данные, т. е. фактические метки классов и спрогнозированные вероятности для меток классов, мы видим различия на этих графиках.

ROC-кривую не следует использовать для анализа в случае сильно несбалансированных данных из-за того, что доля ложноположительных случаев (отношение ложноположительных случаев к общему количеству отрицательных случаев в выборке) не падает существенно при огромном общем количестве отрицательных случаев в выборке.

А вот точность (отношение истинно положительных случаев к общему количеству предсказанных положительных случаев, которое складывается из истинно положительных и ложноположительных случаев) очень чувствительна к числу ложноположительных случаев, и на нее не влияет большое общее количество отрицательных случаев в выборке.

На странице https://github.com/dariyasydykova/open_projects/tree/master/ROC_animation вы найдете полезные визуализации, иллюстрирующие, как меняется наша способность отличать один класс от другого по мере увеличения AUC-ROC и AUC-PR, как на AUC-ROC и AUC-PR влияет дисбаланс классов и т. д.

1.16. КРИВАЯ ЛОРЕНЦА (LORENZ CURVE) И КОЭФФИЦИЕНТ ДЖИНИ (GINI COEFFICIENT)

Кривая Лоренца – график, характеризующий неравномерность распределения доходов среди населения. По оси абсцисс откладывается доля населения, а по оси ординат – доля доходов в обществе в процентном отношении.

Допустим, в компании работают 4 человека с суммарным доходом 10 000\$. Равномерное распределение дохода – это 2500\$ + 2500\$ + 2500\$ + 2500\$, неравномерное – 0\$ + 1000\$ + 2000\$ + 7000\$. Теперь оценим неравномерность для случая 6000\$ + 1000\$ + 2000\$ + 1000\$. Упорядочим сотрудников по возрастанию дохода (1000\$, 1000\$, 2000\$, 6000\$). Построим кривую Лоренца в координатах [процент сотрудников, процент дохода этих сотрудников] – идем по всем сотрудникам и откладываем точки. Для первого – [25 %, 10 %] – это сколько он составляет процентов от всего штата сотрудников и сколько процентов составляет его доход, для второго – [50 %, 20 %] – это сколько они составляют процентов от всего штата и сколько процентов их доход, для первых трёх – [75 %, 40 %], для всех – [100 %, 100 %]. На рисунке ниже приведена кривая Лоренца для нашего примера.

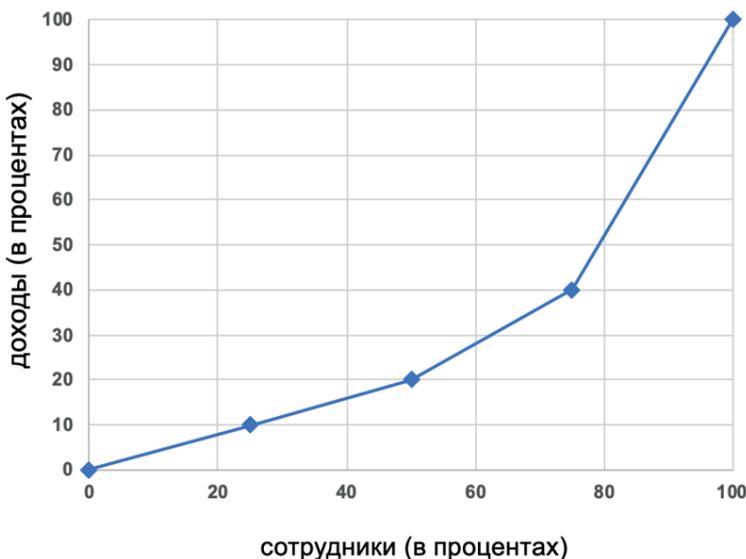


Рис. 51 Пример кривой Лоренца

Теперь выясним связь между кривой Лоренца и коэффициентом Джини.

Изначально коэффициент Джини был статистическим показателем степени расслоения общества относительно какого-либо экономического признака (годовой доход, имущество, недвижимость).

Коэффициент Джини вычисляется как отношение площади фигуры, образованной кривой Лоренца и линией равенства, к площади треугольника, образованного линией равенства и кривой неравенства.

На рисунке ниже построенная кривая Лоренца показана синим цветом. Линия равенства, которая соответствует равномерному распределению дохода, – красная диагональ. Кривая неравенства, которая соответствует неравномерному распределению дохода, – фиолетовая. Площадь А – это площадь, ограниченная кривой Лоренца и линией равенства. Площадь В – это площадь, ограниченная кривой неравенства и кривой Лоренца. Площадь А, поделённая на площадь А + В (площадь треугольника под диагональю – линией равенства), и есть коэффициент Джини.

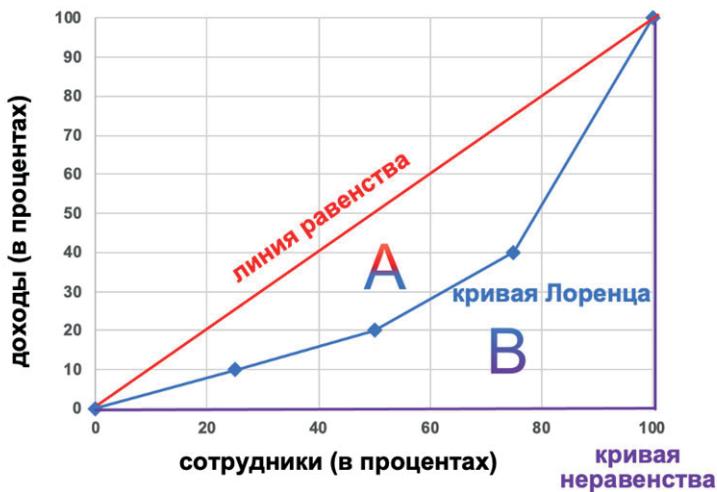
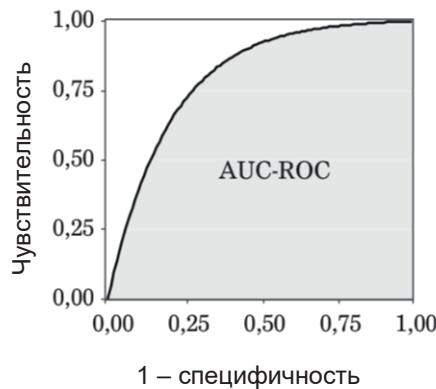
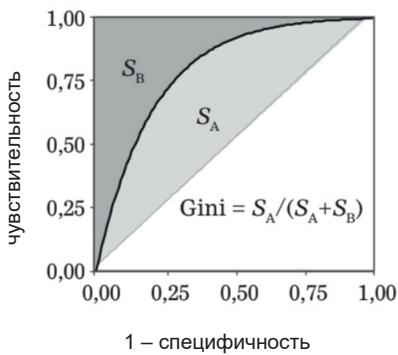


Рис. 52 Коэффициент Джини – это площадь А, поделенная на площадь А + В

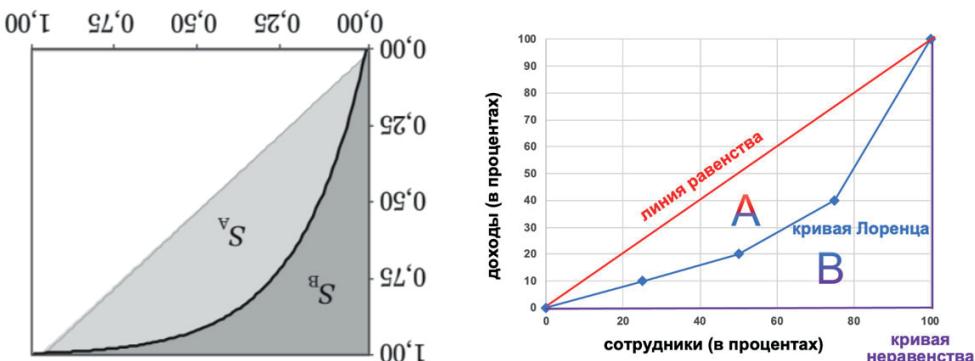
Коэффициент Джини тесно связан с AUC-ROC.
Допустим, у нас есть ROC-кривая.



Площадь между диагональю и ROC-кривой обозначим как S_A . Площадь между ROC-кривой и кривой идеального классификатора (Г-образной кривой) обозначим как S_B .



Мы могли бы перевернуть наш пример и представить в таком виде, по аналогии с примером, разобранным в самом начале этого раздела.



Аналогично тому, как в ранее разобранном примере индекс Джини равен площади А, поделённой на площадь А + В (площадь треугольника под диагональю – линией равенства), индекс Джини равен площади S_A между ROC-кривой и диагональю, поделённой на площадь $S_A + S_B$ (всю площадь треугольника под диагональю – линией бесполезного классификатора, которая равна 0,5):

$$Gini = \frac{S_A}{S_A + S_B} = \frac{AUC_{ROC} - 0,5}{0,5}.$$

Чтобы избавиться от дроби, умножаем числитель и знаменатель на 2:

$$Gini = \frac{2 \times (AUC_{ROC} - 0,5)}{2 \times 0,5} = 2AUC_{ROC} - 1.$$

Таким образом, индекс Джини показывает, во сколько раз увеличение площади под кривой при применении нашей модели меньше увеличения площади под кривой при применении идеальной модели, когда эти площади сравниваются с площадью под диагональю (всегда равна 0,5).

Подробнее о взаимосвязи между AUC-ROC и коэффициентом Джини можно прочитать в статье <https://habr.com/ru/company/ods/blog/350440/>.

На собеседованиях нередко задают следующую задачу.

Если AUC-ROC увеличился на 0,1, то Gini...

- уменьшился на 0,1;
- остался неизменным;
- увеличился на 0,1;
- нет правильного ответа.

Среди ответов нет правильного, поскольку из формулы коэффициента Джини $2 \times AUC_ROC - 1$ становится ясно, что при увеличении AUC-ROC на 0,1 Джини увеличивается на 0,2.

Давайте вычислим коэффициент Джини для нашего примера с оттоком.

```
# вычисляем коэффициент Джини
auc = 0.865
print((2 * auc) - 1)
print((auc - 0.5) / 0.5)
```

```
0.73
0.73
```

1.17. CAP-кривая (CAP curve)

В банковской практике часто используется CAP-кривая (cumulative accuracy profile – профиль кумулятивной достоверности). Заемщики упорядочиваются по вероятности положительного класса, и для каждого значения вероятности положительного класса определяется доля всех и доля «плохих» заемщиков, у которых значение вероятности меньше или равно текущему. Полученные пары чисел наносятся на координатную плоскость, по оси абсцисс откладывается доля всех заемщиков, а по оси ординат – доля «плохих» заемщиков для каждого значения вероятности. У эффективной модели CAP-кривая должна быстро возрастать в начале оси абсцисс. Если CAP-кривая близка к диагонали (доли «плохих» и «хороших» заемщиков возрастают одинаково), это означает, что модель бесполезна. Кроме того, мы можем определить для себя идеальную CAP-кривую. Например, у нас распределение классов составляет 90 % «хороших» и 10 % «плохих». Идеальная модель уже в первом дециле (в первых 10 % наблюдений) определяет 100 % «плохих» заемщиков. Другой пример, у нас распределение классов составляет 96 % «хороших» и 4 % «плохих». Идеальная модель в первых 4 % наблюдений определяет 100 % «плохих» заемщиков.

Давайте загрузим данные (нам нужны фактические метки классов и спрогнозированные вероятности положительного класса), напишем функцию для построения CAP-кривой и воспользуемся ей. У функции – три параметра: фактические метки классов, вероятности положительного класса и заданный процент всех наблюдений, в котором мы смотрим охваченную долю наблюдений положительного класса.

```
# загружаем данные
res = pd.read_csv('Data/Give_me_some_credit.csv', sep=';')
res.head()
```

SeriousDlqin2yrs	Prob
0	0 0.066
1	1 0.296
2	0 0.010
3	0 0.109
4	1 0.916

```
# импортируем integrate
from scipy import integrate

# пишем функцию, которая строит CAP-кривую
def capcurve(y_values, y_preds_proba, percent):
    """
    Автор
    https://github.com/APavlides/cap_curve
```

Строит CAP-кривую.

Параметры

```
-----
y_values : одномерный массив формы [n_samples]
Фактические метки (классы) зависимой переменной.
y_preds_proba : одномерный массив формы [n_samples]
Спрогнозированные вероятности положительного класса.
percent: float
Заданный процент всех наблюдений, в котором мы смотрим
охваченную долю наблюдений положительного класса.
"""
```

```
# вычисляем количество наблюдений положительного класса
num_pos_obs = np.sum(y_values)
# вычисляем общее количество наблюдений
num_count = len(y_values)
# вычисляем долю наблюдений положительного класса
rate_pos_obs = float(num_pos_obs) / float(num_count)
# задаем координаты трех точек для идеальной модели
ideal = np.array([[0, 0], [rate_pos_obs, 1], [1, 1]])
# делим значение последовательности (индекс наблюдения)
# на количество наблюдений минус единица, по сути, для
# каждого значения вероятности получаем долю всех
# наблюдений, которую и будем откладывать по оси x
x = np.arange(num_count) / float(num_count - 1)
# конкатенируем фактические метки классов
# и вероятности положительного класса
y_cap = np.c_[y_values, y_preds_proba]
# упорядочиваем по убыванию вероятности положительного класса
y_cap_sort = y_cap[y_cap[:, 1].astype(float).argsort()][::-1]
# делим накопленную сумму по фактическим меткам классов
# на количество наблюдений положительного класса, по сути,
# для каждого значения вероятности получаем долю наблюдений
# положительного класса, которую и будем откладывать по оси y
y = np.cumsum(y_cap_sort[:, 0]) / float(num_pos_obs)
```

```
# добавляем первую точку кривой (0,0) : для xx=0 получаем yy=0
y = np.append([0], y[0: num_count - 1])
# задаем количество наблюдений, умножим общее
# количество наблюдений на процент
row_index = int(np.trunc(num_count * percent))

# вычисляем процент наблюдений положительного класса для
# заданного процента всех наблюдений
val_y1 = y[row_index]
val_y2 = y[row_index + 1]
if val_y1 == val_y2:
    val = val_y1 * 1.0
else:
    val_x1 = x[row_index]
    val_x2 = x[row_index + 1]
    val = (val_y1 + ((val_x2 - percent) /
                      (val_x2 - val_x1)) * (val_y2 - val_y1))

# округляем процент
val = round(val, 2)

# вычисляем площадь идеальной модели
sigma_ideal = (1 * x[num_pos_obs - 1] / 2 +
                (x[num_count - 1] - x[num_pos_obs]) * 1)
# вычисляем площадь рабочей модели,
# интегрируя по формуле Симпсона
sigma_model = integrate.simps(y, x)
# вычисляем площадь случайной модели,
# интегрируя по формуле Симпсона
sigma_random = integrate.simps(x, x)

# вычисляем коэффициент Джини
gini_value = ((sigma_model - sigma_random) /
               (sigma_ideal - sigma_random))
# округляем коэффициент Джини
gini_value = round(gini_value, 2)

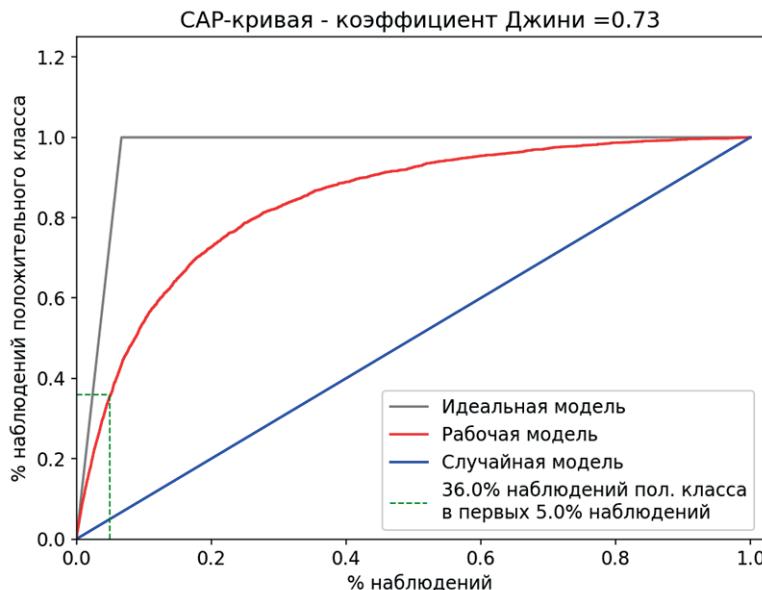
# задаем области Figure и Axes
fig, ax = plt.subplots(nrows=1, ncols=1)
# строим кривые идеальной, рабочей и случайной моделей
ax.plot(ideal[:, 0], ideal[:, 1], color='grey',
        label="Идеальная модель")
ax.plot(x, y, color='red', label="Рабочая модель")
ax.plot(x, x, color='blue', label="Случайная модель")
string = str(val * 100) + "% наблюдений пол. класса\nв первых " +
         str(percent * 100) + "% наблюдений"
ax.plot([percent, percent], [0.0, val], color='green',
        linestyle='--', linewidth=1)
ax.plot([0, percent], [val, val], color='green',
        linestyle='--', linewidth=1, label=string)
# задаем диапазоны значений осей
plt.xlim(0, 1.02)
plt.ylim(0, 1.25)
# задаем заголовок диаграммы
plt.title("CAP-кривая - коэффициент Джини =" + str(gini_value))
# задаем подписи осей
plt.xlabel("% наблюдений")
```

```

plt.ylabel("% наблюдений положительного класса")
# задаем расположение легенды
plt.legend(loc='lower right')

# применяем нашу функцию
carccurve(res['SeriousDlqin2yrs'], res['Prob'], 0.5)

```



В данном случае идеальная модель – модель, которая в первых 0,067 % наблюдений выделяет 100 % наблюдений положительного класса. Наша модель в первых 5 % наблюдений выявляет 36 % наблюдений положительного класса.

1.18. СТАТИСТИКА КОЛМОГОРОВА–СМИРНОВА (KOLMOGOROV–SMIRNOV STATISTIC)

В кредитном скоринге статистика Колмогорова–Смирнова (КС) измеряет максимальную абсолютную разницу между кумулятивными функциями распределения «хороших» и «плохих» заемщиков. Она вычисляется по формуле:

$$KS = \max_s |F_B(s) - F_G(s)|,$$

где $F_B(s)$ и $F_G(s)$ – эмпирические кумулятивные распределения скорингового балла для «плохих» и «хороших» заемщиков.

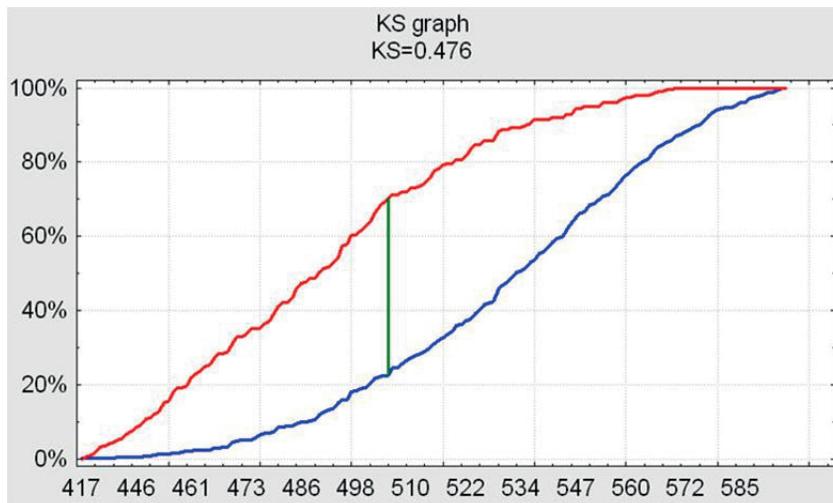


Рис. 53 Статистика Колмогорова–Смирнова

Слабость этой метрики заключается в том, что разница измеряется только в одной точке (которая может и не совпадать с ожидаемым порогом отсечения), а не по всему диапазону скоринговых баллов. Точка достижения максимально-го значения КС обычно находится ближе к середине диапазона. Поэтому если предполагаемый порог скоринговой карты находится в нижней (начальной) или верхней (конечной) части диапазона, то данная метрика не может быть хорошим критерием для сравнения карт. В таких случаях лучше сравнивать отклонение в предполагаемом пороге отсечения. Изменения в определенном бине повлекут изменение KS.

Теоретически КС может принимать значения от 0 до 100, однако на практике она обычно оказывается в диапазоне от 25 до 75. Обычно руководствуются правилом:

- меньше 20 – скоринговая карта непригодна к применению;
- 20–40 – среднее качество скоринговой карты;
- 41–50 – хорошее качество скоринговой карты;
- 51–60 – очень хорошее качество скоринговой карты;
- 61–75 – отличное качество скоринговой карты;
- больше 75 – вероятно, были допущены ошибки в разработке скоринговой карты.

KS-статистику можно вычислить с помощью функции `ks_2samp()` библиотеки `scipy`, в которой реализован двухвыборочный тест Колмогорова–Смирнова.

```
# вычислим KS с помощью функции ks_2samp()
from scipy.stats import ks_2samp
ks_2samp(res.loc[res['SeriousDlqin2yrs'] == 0, 'Prob'],
          res.loc[res['SeriousDlqin2yrs'] == 1, 'Prob'])

Ks_2sampResult(statistic=0.576005231999611, pvalue=0.0)
```

Давайте напишем и применим упрощенный вариант функции `ks_2samp()`.

```
# пишем упрощенный вариант функции ks_2samp()
def ks_2samp_simple(data1, data2):
    """
    Параметры
    -----
    data1, data2: одномерные массивы - массив вероятностей
    положительного класса для наблюдений отрицательного
    класса и массив вероятностей положительного
    класса для наблюдений положительного класса
    """

    # превращаем объекты в массивы NumPy
    data1, data2 = map(np.asarray, (data1, data2))
    # записываем количество наблюдений отрицательного класса
    n1 = data1.shape[0]
    # записываем количество наблюдений положительного класса
    n2 = data2.shape[0]
    # сортируем массивы NumPy с вероятностями по возрастанию
    data1 = np.sort(data1)
    data2 = np.sort(data2)
    # конкатенируем массивы NumPy с вероятностями по оси строк
    data_all = np.concatenate([data1, data2])
    # вычисляем кумулятивную функцию распределения наблюдений
    # отрицательного класса
    cdf1 = np.searchsorted(data1, data_all, side='right') / (1.0 * n1)
    # вычисляем кумулятивную функцию распределения наблюдений
    # положительного класса
    cdf2 = (np.searchsorted(data2, data_all, side='right')) / (1.0 * n2)
    # находим максимальную абсолютную разницу между
    # кумулятивными функциями распределения
    d = np.max(np.abs(cdf1 - cdf2))
    return d

# применяем упрощенный вариант функции ks_2samp()
ks_2samp_simple(res.loc[res['SeriousDlqin2yrs'] == 0, 'Prob'],
                 res.loc[res['SeriousDlqin2yrs'] == 1, 'Prob'])
```

0.576005231999611

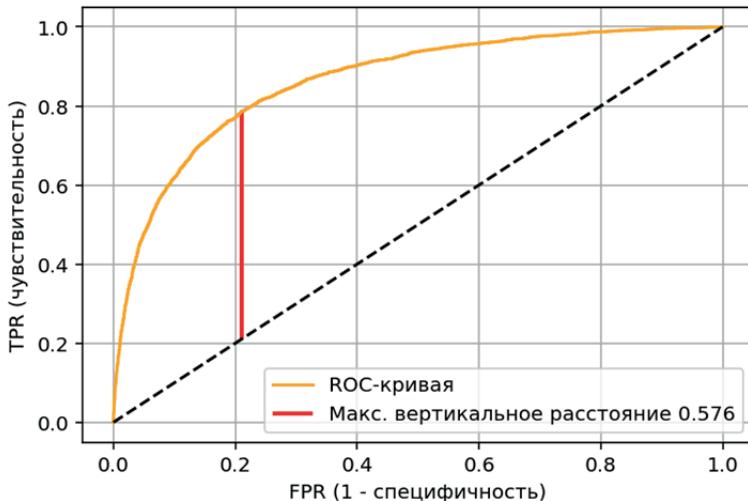
Статистика Колмогорова–Смирнова равна максимальному вертикальному расстоянию между ROC-кривой и диагональю (если предположить, что ROC-кривая лежит над диагональю). Давайте убедимся в этом.

```
# вычисляем значения 1 - специфичности, чувствительности
# для всех пороговых значений
fpr, tpr, thresholds = roc_curve(res['SeriousDlqin2yrs'], res['Prob'])
# максимальное вертикальное расстояние и его индекс
max_distance = np.max(tpr - fpr)
indx = np.argmax(tpr - fpr)
# строим ROC-кривую, красная вертикальная черта - KS
plt.grid()
plt.vlines(x=fpr[indx],
            ymin=tpr[indx] - max_distance,
            ymax=tpr[indx], linewidth=2, color='red',
            label="Макс. вертикальное расстояние {:.3f}".format(max_distance))
```

```

plt.xlabel("FPR (1 - специфичность)")
plt.ylabel("TPR (чувствительность)")
plt.plot(fpr, tpr, label='ROC-кривая', color='orange')
plt.plot([0, 1], [0, 1], 'k--')
plt.legend(loc=4);

```



Видим, что KS действительно равен вертикальной линии между ROC-кривой и диагональю.

1.19. Биномиальный тест (BINOMIAL TEST)

Биномиальный тест проверяет достоверность оценки вероятности дефолта (PD) для конкретного диапазона скорингового балла, при этом предполагается независимость событий возникновения дефолта заемщиков. В случае если события дефолта являются независимыми, их количество подчиняется биномиальному распределению. Биномиальный тест сравнивает две гипотезы:

- нулевая гипотеза: оценка PD для данного диапазона скорингового балла корректна;
- альтернативная гипотеза: оценка PD для данного диапазона скорингового балла занижена (вероятность дефолта недооценена).

Вероятность возникновения заданного количества «плохих» заемщиков X среди n заемщиков в данном диапазоне скорингового балла определяется по формуле биномиального теста

$$P(X) = \binom{n}{X} PD^X (1-PD)^{n-X} = \frac{n!}{X!(n-X)!} PD^X (1-PD)^{n-X}.$$

Предположим, у нас есть диапазон скорингового балла с $PD = 0.1$. В последующий период было найдено, что 3 заемщика попали в этот диапазон, из них двое оказались «хорошими», а один – «плохим». Можем ли мы на уровне значимости 0,05 отклонить нулевую гипотезу о том, что PD равна 0,1?

Вероятность получить 3 «плохих» заемщиков равна

$$P(X = 3) = \frac{3!}{3!(3-3)!} (0,1)^3 (0,9)^{3-3} = 1 \times 0,001 \times 1 = 0,001.$$

Вероятность получить 2 «плохих» заемщиков и 1 «хорошего» заемщика равна

$$P(X = 2) = \frac{3!}{2!(3-2)!} (0,1)^2 (0,9)^{3-2} = 3 \times 0,01 \times 0,9 = 0,027.$$

Вероятность получить 2 «плохих» заемщиков и более составляет:

$$P(X \geq 2) = P(X = 2) + P(X = 3) = 0,027 + 0,001 = 0,028.$$

0,028 ниже уровня значимости 0,05.

Вероятность получить 1 «плохого» заемщика и 2 «хороших» заемщиков равна

$$P(X = 1) = \frac{3!}{1!(3-1)!} (0,1)^1 (0,9)^{3-1} = 3 \times 0,1 \times 0,81 = 0,243.$$

Вероятность получить 1 «плохого» заемщика и более составляет:

$$P(X \geq 1) = P(X = 1) + P(X = 2) + P(X = 3) = 0,243 + 0,027 + 0,001 = 0,271.$$

0,271 превышает уровень значимости 0,05.

Вероятность того, что мы не получим ни одного «плохого» заемщика, равна

$$P(X = 0) = \frac{3!}{0!(3-0)!} (0,1)^0 (0,9)^{3-0} = 1 \times 1 \times 0,729 = 0,729.$$

0,729 превышает уровень значимости 0,05.

Итоговая таблица биномиального распределения для нашего случая выглядит так:

$P(X = 0) = 0,729$	← Вероятность того, что мы не получим ни одного «плохого» заемщика
$P(X = 1) = 0,243$	← Вероятность получить одного «плохого» заемщика
$P(X = 2) = 0,027$	← Вероятность получить двух «плохих» заемщиков
$P(X = 3) = 0,001$	← Вероятность получить трех «плохих» заемщиков

} Вероятность получить одного «плохого» заемщика и более

Таким образом, на уровне значимости 0,05 у нас нет оснований отклонить нулевую гипотезу, если сегмент с $PD\ 0,1$ не содержит «плохих» заемщиков, содержит 1 «плохого» заемщика, однако отклоняем ее, если наш сегмент содержит 2 «плохих» заемщиков и более.

Применим биномиальный тест для нашего примера в Python.

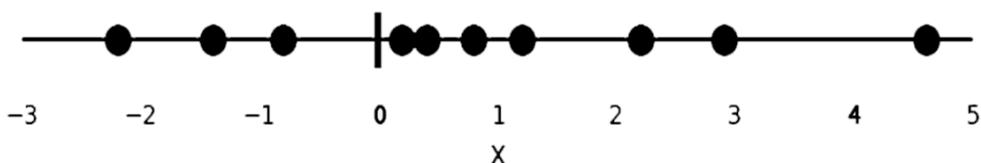
```
# выполняем биномиальный тест для нашего примера
from scipy.stats import binom_test
binom_test(1, 3, 0.1)

0.2709999999999999
```

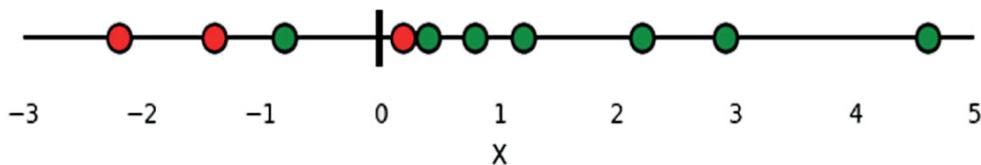
Вероятность получить 1 «плохого» заемщика и более составляет 0,271 (0,243 + 0,027 + 0,001).

1.20. Логистическая функция потерь (LOGISTIC LOSS)

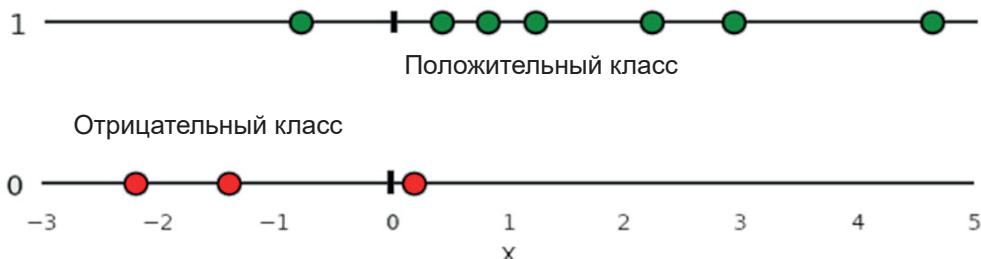
Представим, у нас есть 10 значений признака x .



Давайте обозначим их некоторыми цветами (красным и зеленым). Это и будут наши метки классов.



Пусть зеленые будут положительным классом, а красные – отрицательным.



Поскольку речь идет о бинарной классификации, мы можем сформулировать нашу задачу в виде вопроса «данная точка имеет зеленый цвет?» или, еще лучше, «какова вероятность того, что эта точка имеет зеленый цвет?» («какова вероятность того, что эта точка относится к положительному классу?»), ведь нам нужна степень уверенности модели в зеленом цвете рассматриваемой точки. В идеале, у зеленых точек вероятность быть зелеными должна быть 1, в то время как у красных точек вероятность быть зелеными должна быть 0.

Если мы обучаем модель классификации, она будет предсказывать вероятность зеленого цвета для каждой из наших точек. Учитывая все то, что мы зна-

ем о цвете точек, как мы можем оценить, насколько хороши (или плохи) наши полученные вероятности? Вот именно этим оцениванием и будет заниматься наша функция потерь! У нее будут высокие значения для плохих прогнозов и низкие значения для хороших прогнозов. В задаче бинарной классификации функцией потерь будет логистическая функция потерь.

Взглянем на формулу логистической функции потерь для бинарной классификации:

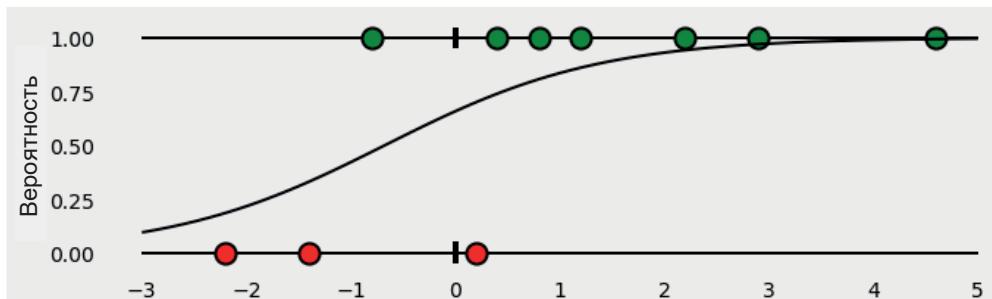
$$\text{logloss} = -\frac{1}{N} \sum_{i=1}^N [y_i \log p_i + (1-y_i) \log (1-p_i)].$$

Здесь y_i – это метка класса для i -й точки (1 для зеленых точек и 0 для красных точек) и p_i – это вероятность i -й точки быть зеленой, мы берем метки и вероятности по каждой из всех N точек.

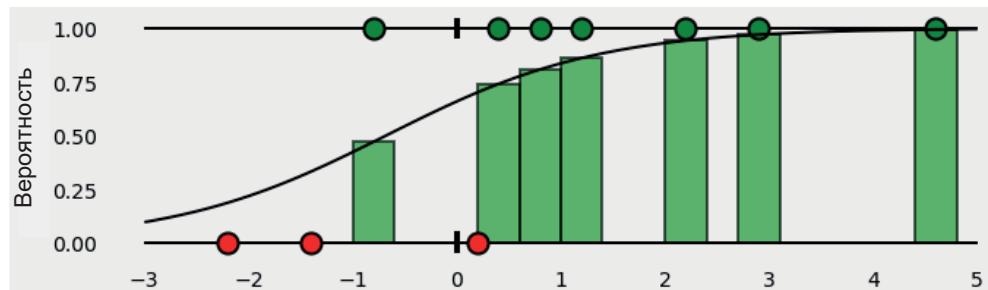
Что можно увидеть сразу? Функция не может принимать отрицательные значения. Чтобы увидеть это, обратите внимание на два момента:

- все отдельные члены в сумме являются отрицательными, поскольку мы берем логарифмы чисел в диапазоне от 0 до 1;
- впереди суммы стоит знак минуса.

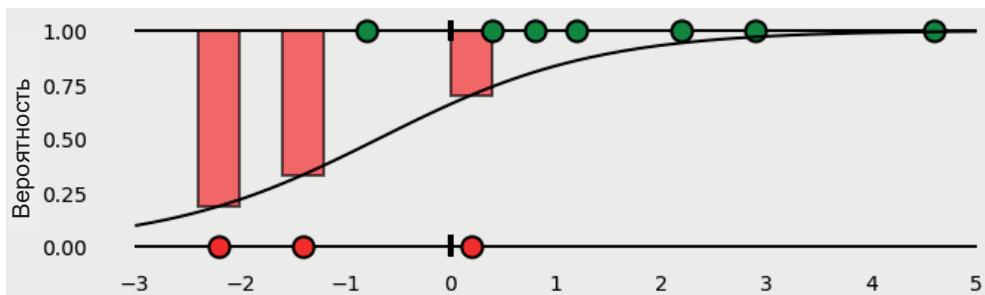
Теперь давайте обучим модель логистической регрессии, чтобы классифицировать наши точки. Обученная модель – сигмоида, представляющая собой вероятность того, что точка будет зеленой для любого заданного значения x .



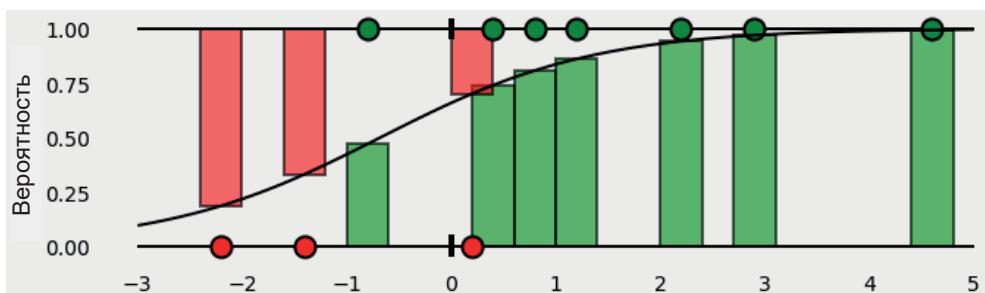
Итак, вычислим вероятности для всех точек, принадлежащих положительному классу (зеленые точки). Это будут зеленые столбики под сигмоидой в соответствующих точках.



Теперь вычислим вероятности для всех точек, принадлежащих отрицательному классу (красные точки). Это будут красные столбики над сигмоидой в соответствующих точках.

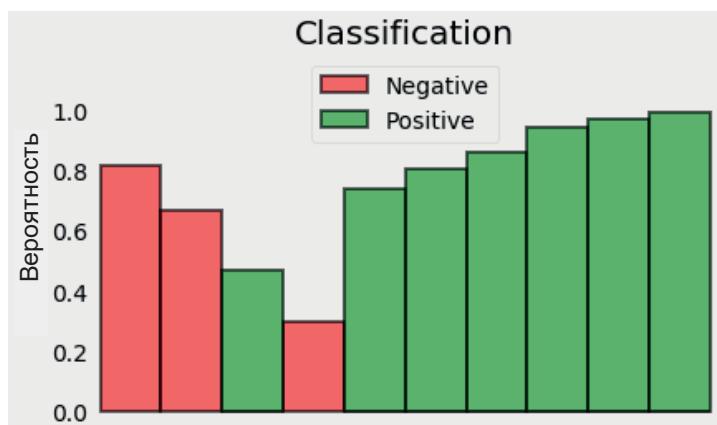


А сейчас все совместим.



Столбики – это вероятности, вычисленные для наших точек.

Столбики, свисающие сверху, не очень понятны, давайте изменим их расположение.



Поскольку мы пытаемся вычислить функцию потерь, нам нужно штрафовать за неправильные прогнозы. Если вероятность, связанная с фактическим классом, равна 0,999, нам нужно, чтобы потеря в этом случае была равна нулю. И наоборот, если эта вероятность мала, скажем, 0,01, нам нужно, чтобы потеря была большой!

Оказывается, для этой цели нам достаточно хорошо подходит (отрицательный) логарифм вероятностей (поскольку логарифм значений от 0 до 1 является отрицательным числом, мы берем отрицательный логарифм, чтобы получить положительное значение функции потерь для удобства интерпретации).

График ниже дает нам ясную картину: если вероятность фактического класса становится ближе к нулю, потеря возрастает экспоненциально. Например, если вероятность равна 0,99, то отрицательный логарифм будет равен 0,004, а вот если вероятность будет равна 0,0001, то отрицательный логарифм будет равен 4.

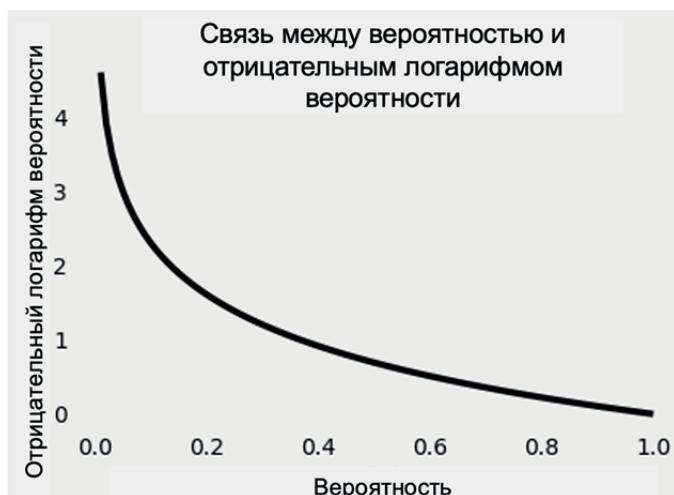


Рис. 54 Связь между вероятностью и отрицательным логарифмом вероятности

Отсюда хорошо видна особенность логистической функции потерь. Она заключается в том, что логистическая функция потерь крайне сильно штрафует за уверенность классификатора в неверном ответе. Ошибка на одном объекте может дать существенное ухудшение общей ошибки на выборке. У идеального классификатора значение логистической функции потерь будет равно точно 0.

Итак, давайте возьмем (отрицательный) логарифм вероятностей – это соответствующие потери, вычисленные для наших точек. После усреднения получаем значение логистической функции потерь.

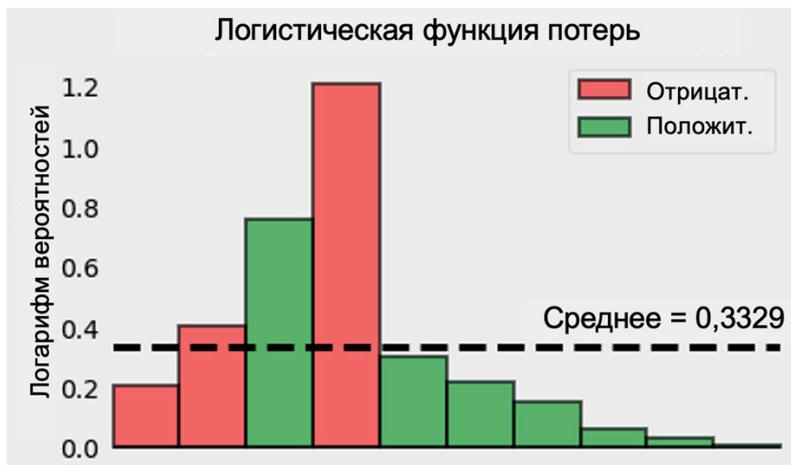


Рис. 55 Получение итогового значения логистической функции потерь

Обратите внимание, что логистическая функция потерь, вычисленная по формуле $-\frac{1}{N} \sum_{i=1}^N [y_i \log p_i + (1-y_i) \log(1-p_i)]$, будет неопределенной для вероятностей 0 и 1, поэтому для решения проблемы применяется правило минимакса $\max(\text{eps}, \min(1 - \text{eps}, p))$, где eps обычно равен $1e-15$, вероятность 0 будет преобразована в $1e-15$ (~ 0.0000000000000001), а вероятность 1 будет преобразована в $1 - 1e-15$ (~ 0.9999999999999999).

Вычислим для нашего примера с оттоком значение логистической функции потерь.

```
# импортируем функцию log_loss
from sklearn.metrics import log_loss
# вычисляем logloss
logloss = log_loss(data['fact'], data['probability'])
# печатаем logloss
print("Логистическая функция потерь: {:.3f}".format(logloss))
```

Логистическая функция потерь: 0.532

Можно реализовать собственную функцию, вычисляющую логистическую функцию потерь.

```
# пишем функцию, вычисляющую logloss
def LogLoss(fact, prob):
    eps = 1e-15
    prob = np.clip(prob, eps, 1 - eps)
    logloss = (-1 / len(prob)) * np.sum(
        fact * np.log(prob) + (1 - fact) * np.log(1 - prob))
    return logloss
```

```
# применяем нашу функцию
logloss = LogLoss(data['fact'], data['probability'])
# печатаем logloss
print("Логистическая функция потерь: {:.3f}".format(logloss))
```

Логистическая функция потерь: 0.532

На собеседованиях часто задают вопрос, чтобы было бы, если бы мы отказались от использования логарифма в формуле. Отказ от логарифма приведет к тому, что мы будем получать по каждому наблюдению вместо отрицательных положительные результаты, при умножении положительной суммы на -1 мы получим отрицательный числитель, и при делении на количество наблюдений, являющееся положительным числом, у нас будет очень низкое, отрицательное значение функции потерь, которое не удобно для интерпретации.

Давайте в нашей самописной функции модифицируем формулу, избавившись от логарифма, и заново вычислим логистическую функцию потерь.

```
# пишем функцию, вычисляющую logloss без логарифма
def LogLoss_without_log(fact, prob):
    logloss = (-1 / len(prob)) * np.sum(
        fact * prob + (1 - fact) * (1 - prob))
    return logloss

# применяем нашу функцию
logloss = LogLoss_without_log(data['fact'], data['probability'])
# печатаем logloss
print("Измененная логистическая функция потерь: {:.3f}".format(logloss))
```

Измененная логистическая функция потерь: -0.653

Как и говорили выше, получаем отрицательное значение логистической функции потерь.

2. Регрессия

2.1. R^2 , КОЭФФИЦИЕНТ ДЕТЕРМИНАЦИИ (R-SQUARE, COEFFICIENT OF DETERMINATION)

Если совсем просто, то R^2 показывает, насколько линия регрессии лучше простой горизонтальной линии, проведенной по среднему значению зависимой переменной. На рисунке ниже синяя линия – это данные, для которых мы пытаемся построить модель регрессии, черная линия – линия регрессии, представляющая собой график уравнения $y = 6x - 5$ (регрессионный коэффициент 6 и константа -5).

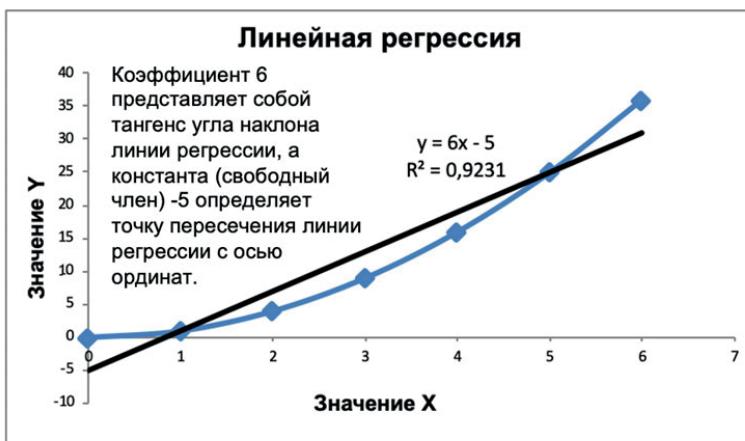


Рис. 56 Уравнение регрессии и его интерпретация

На рисунке ниже горизонтальная красная линия – это среднее значение зависимости переменной. Так выглядит самый простой прогноз. С этой красной линией мы и сравниваем нашу линию регрессии.



Рис. 57 Прогноз средним

В нашем случае фактические значения зависимой переменной – это значения признака, возведенные в квадрат, а спрогнозированные значения зависимой переменной получены с помощью уравнения $y = 6x - 5$.

Таблица 1 Таблица фактических и спрогнозированных значений зависимой переменной

Значение предиктора x_i	Фактическое значение зависимой переменной y_i $y_i = x_i^2$	Спрогнозированное значение зависимой переменной \hat{y}_i $\hat{y}_i = 6x_i - 5$
0	0	-5
1	1	1
2	4	7
3	9	13
4	16	19
5	25	25
6	36	31
		Среднее
		13

R^2 (или R -квадрат) вычисляется как единица минус отношение остаточной суммы квадратов отклонений (RSS) к общей сумме квадратов отклонений (TSS):

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{RSS}{TSS}$$

Остаточная сумма квадратов отклонений (residual sum of squares) – это сумма квадратов отклонений фактических значений зависимой переменной от спрогнозированных
 Общая сумма квадратов отклонений (total sum of squares) – это сумма квадратов отклонений фактических значений зависимой переменной от ее среднего значения

где

\bar{y}_i – среднее значение зависимой переменной;

\hat{y}_i – спрогнозированное значение зависимой переменной;

y_i – фактическое значение зависимой переменной.

Для вычисления общей суммы квадратов отклонений нам нужно:

- найти среднее значение зависимой переменной;
- из каждого фактического значения зависимой переменной вычесть среднее значение зависимой переменной;
- возвести разность в квадрат;
- сложить квадраты разностей.



Рис. 58 Вычисление общей суммы квадратов отклонений

Для вычисления остаточной суммы квадратов отклонений нам нужно:

- найти спрогнозированное значение зависимой переменной;
- из каждого фактического значения зависимой переменной вычесть спрогнозированное значение зависимой переменной;
- возвести разность в квадрат;
- сложить квадраты разностей.

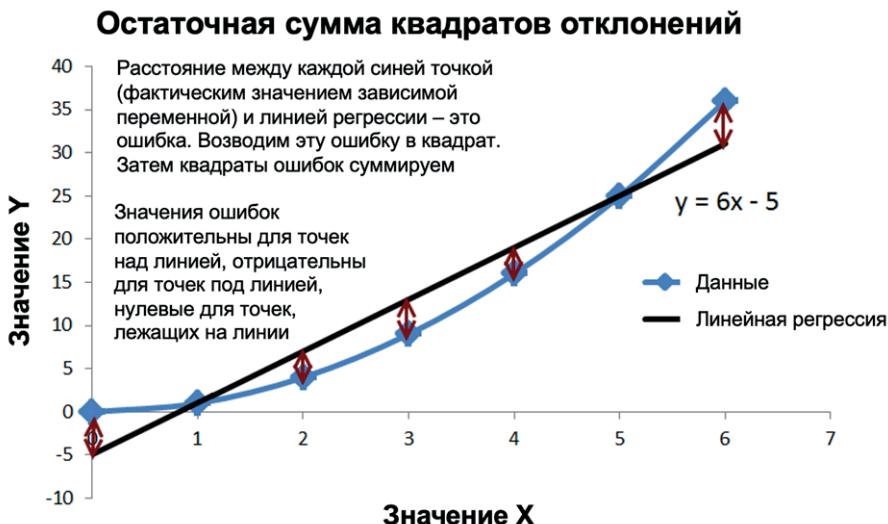


Рис. 59 Вычисление остаточной суммы квадратов отклонений

Таким образом, R^2 измеряет долю дисперсии зависимой переменной, объясненную моделью.

Давайте вычислим R -квадрат для нашего игрушечного примера.

Таблица 2 Таблица фактических и спрогнозированных значений зависимой переменной, квадратов отклонений фактических значений зависимой переменной от спрогнозированных, квадратов отклонений фактических значений зависимой переменной от ее среднего значения

Значение признака x_i	Фактическое значение зависимой переменной y_i , $y_i = x_i^2$	Спрогнозированное значение зависимой переменной \hat{y}_i $\hat{y}_i = 6x_i - 5$	Остаток или отклонение (разность между фактическим и спрогнозированным значениями зависимой переменной) $(y_i - \hat{y}_i)$	Квадраты отклонений фактических значений зависимой переменной от спрогнозированных $(y_i - \hat{y}_i)^2$	Квадраты отклонений фактических значений зависимой переменной от ее среднего значения $(y_i - \bar{y})^2$
0	0	-5	5	25	169
1	1	1	0	0	144
2	4	7	-3	9	81
3	9	13	-4	16	16
4	16	19	-3	9	9
5	25	25	0	0	144
6	36	31	5	25	529

	Среднее			Остаточная сумма квадратов	Общая сумма квадратов
	13			84	1092

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{RSS}{TSS} = 1 - \frac{84}{1092} = 0,9231.$$

Обычно пишут, что значение R^2 может принимать значение от 0 до 1 (чем больше, тем лучше). В случае если модель идеально описывает ряд данных, RSS становится равной нулю, в результате чего R^2 становится равен единице.

Вычислим R^2 для случая, когда мы идеально прогнозируем.

Таблица 3 Таблица фактических и спрогнозированных значений зависимой переменной, квадратов отклонений фактических значений зависимой переменной от спрогнозированных, квадратов отклонений фактических значений зависимой переменной от ее среднего значения для идеального случая

Значение признака x_i	Фактическое значение зависимой переменной $y_i = x_i^2$	Спрогнозированное значение зависимой переменной \hat{y}_i $\hat{y}_i = x_i^2$	Остаток или отклонение (разность между фактическим и спрогнозированным значениями зависимой переменной) $(y_i - \hat{y}_i)$	Квадраты отклонений фактических значений зависимой переменной от спрогнозированных $(y_i - \hat{y}_i)^2$	Квадраты отклонений фактических значений зависимой переменной от ее среднего значения $(y_i - \bar{y})^2$
0	0	0	0	0	169
1	1	1	0	0	144
2	4	4	0	0	81
3	9	9	0	0	16
4	16	16	0	0	9
5	25	25	0	0	144
6	36	36	0	0	529

	Среднее			Остаточная сумма квадратов	Общая сумма квадратов
	13			0	1092

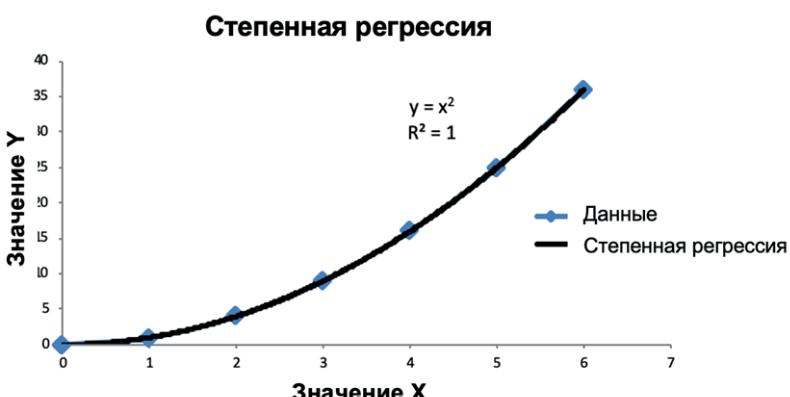


Рис. 60 Идеальная линия регрессии

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{RSS}{TSS} = 1 - \frac{0}{1092} = 1.$$

Если же модель ряд совсем не описывает, а представляет собой просто прямую линию, проведенную по среднему значению зависимой переменной, то R^2 становится равным нулю.

Вычислим R^2 для случая, когда мы просто прогнозируем средним значением зависимой переменной.

Таблица 4 Таблица фактических и спрогнозированных значений зависимой переменной, квадратов отклонений фактических значений зависимой переменной от спрогнозированных, квадратов отклонений фактических значений зависимой переменной от ее среднего значения для случая, когда прогнозируем средним значением зависимой переменной

Значение признака x_i	Фактическое значение зависимой переменной y_i $y_i = x_i^2$	Спрогнозированное значение зависимой переменной \hat{y}_i $\hat{y}_i = \bar{y}$	Остаток или отклонение (разность между фактическим и спрогнозированным значениями зависимой переменной) $(y_i - \hat{y}_i)$	Квадраты отклонений фактических значений зависимой переменной от спрогнозированных $(y_i - \hat{y}_i)^2$	Квадраты отклонений фактических значений зависимой переменной от ее среднего значения $(y_i - \bar{y})^2$
0	0	13	-13	169	169
1	1	13	-12	144	144
2	4	13	-9	81	81
3	9	13	-4	16	16
4	16	13	3	9	9
5	25	13	12	144	144
6	36	13	23	529	529

	Среднее			Остаточная сумма квадратов	Общая сумма квадратов
	13			1092	1092

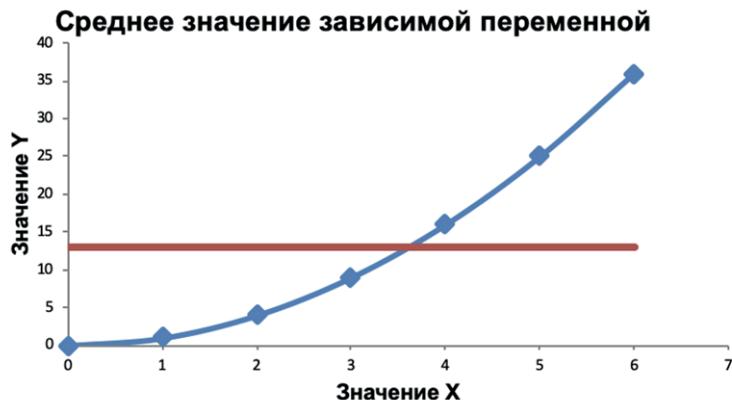


Рис. 61 Наша линия регрессии – линия, проведенная по среднему значению зависимой переменной

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{RSS}{TSS} = 1 - \frac{1092}{1092} = 0.$$

Однако если простое среднее значение зависимой переменной приближает лучше, чем наша линия регрессии (можно просто в качестве константного прогноза взять медиану зависимой переменной), это говорит о крайней неадекватности модели и R^2 может стать отрицательным.

Вычислим R^2 для случая, когда мы прогнозируем медианой зависимой переменной.

Таблица 5 Таблица фактических и спрогнозированных значений зависимой переменной, квадратов отклонений фактических значений зависимой переменной от спрогнозированных, квадратов отклонений фактических значений зависимой переменной от ее среднего значения для случая, когда прогнозируем медианой зависимой переменной

Значение признака x_i	Фактическое значение зависимой переменной y_i $y_i = x_i^2$	Спрогнозированное значение зависимой переменной \hat{y}_i $\hat{y}_i = median(y)$	Остаток или отклонение (разность между фактическим и спрогнозированным значениями зависимой переменной) $(y_i - \hat{y}_i)$	Квадраты отклонений фактических значений зависимой переменной от спрогнозированных $(y_i - \hat{y}_i)^2$	Квадраты отклонений фактических значений зависимой переменной от ее среднего значения $(y_i - \bar{y})^2$
0	0	9	-9	81	169
1	1	9	-8	64	144
2	4	9	-5	25	81
3	9	9	0	0	16
4	16	9	7	49	9
5	25	9	16	256	144
6	36	9	27	729	529

	Среднее			Остаточная сумма квадратов	Общая сумма квадратов
	13			1204	1092

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{RSS}{TSS} = 1 - \frac{1204}{1092} = -0,103.$$

R^2 может быть отрицательным, когда на константу и коэффициент наложены ограничения так, чтобы линия регрессии согласно этим ограничениям работала хуже, чем горизонтальная линия, проведенная по среднему значению зависимой переменной, а также для нелинейных моделей (например, в градиентном бустинге, когда прогнозная функция предсказывает хуже, чем простое среднее значение зависимой переменной).

Загрузим данные, которые состоят из двух столбцов – столбца фактических значений зависимой переменной (фактические значения задолженности по кредитной карте), столбца спрогнозированных значений зависимой переменной (спрогнозированные значения задолженности по кредитной карте).

```
# импортируем необходимые библиотеки
import numpy as np
import pandas as pd
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/results2.csv', sep=';')
data
```

	fact	pred
0	1.21	0.888070
1	7.09	7.903147
2	1.88	1.200113
3	0.24	1.053039
4	0.55	0.867723
5	0.08	0.139170
6	1.06	0.911489
7	0.12	1.113673
8	1.19	0.918506
9	1.63	0.948829

Вычислим R^2 .

```
# вычисляем сумму квадратов отклонений
# фактических значений зависимой переменной
# от ее среднего значения
TSS = ((data['fact'] - data['fact'].mean()) ** 2).sum()
# вычисляем сумму квадратов отклонений фактических
# значений зависимой переменной от спрогнозированных
RSS = ((data['fact'] - data['pred']) ** 2).sum()
# вычисляем R-квадрат
R_squared = 1 - (RSS / TSS)
R_squared
```

0.907322249223145

Основная проблема применения R^2 заключается в том, что его значение увеличивается (не уменьшается) от добавления в модель новых переменных, даже если эти переменные никакого отношения к зависимой переменной не имеют!

Интуитивная причина, по которой включение дополнительного признака в модель не может уменьшить значение R^2 , заключается в следующем: минимизация остаточной суммы квадратов эквивалентна максимизации R^2 . Когда включен дополнительный признак, у нас всегда есть возможность присвоить ему нулевой коэффициент, оставив спрогнозированные значения и R^2 без изменений. Единственный способ, при котором процедура оптимизации даст ненулевой коэффициент, заключается в увеличении R^2 . Поэтому сравнение моделей с разным количеством признаков с помощью коэффициента детерминации некорректно. R^2 может выступать лишь индикатором того, насколько модель адекватна. Низкие значения коэффициента детерминации могут указывать на то, что модель особого смысла не имеет, но при этом высокие значения не говорят о том, что перед нами хорошая модель.

Если нужно сравнить модели с разным количеством признаков, можно использовать альтернативные показатели, например скорректированный коэффициент детерминации.

$$R_{adj}^2 = 1 - \frac{RSS / (n-k)}{TSS / (n-1)} = 1 - (1 - R^2) \cdot \frac{(n-1)}{n-k} \leq R^2.$$

Он накладывает штраф за дополнительно включенные признаки, где n – количество наблюдений, а k – количество параметров.

Скорректированный R^2 может быть отрицательным, его значение всегда будет меньше или равно значению R^2 . В отличие от R^2 , скорректированный R^2 увеличивается только тогда, когда увеличение R^2 (из-за включения нового признака) больше, чем можно было бы ожидать в силу случайности. Чаще всего его используют для сравнения альтернативных моделей на этапе отбора признаков при построении модели.

2.2. МЕТРИКИ КАЧЕСТВА, КОТОРЫЕ ЗАВИСЯТ ОТ МАСШТАБА ДАННЫХ (RMSE, MSE, MAE, MdAE, RMSLE, MSLE)

Существуют метрики качества, которые зависят от масштаба данных (*scale-dependent metrics*).

Есть несколько часто используемых метрик качества, которые зависят от масштаба данных. Они полезны при сравнении разных методов на одном и том же наборе данных, но не должны использоваться, например, при сравнении наборов данных разных масштабов. Самые часто используемые метрики качества, зависящие от масштаба данных, основаны на абсолютной, квадратичной, квадратичной логарифмированной ошибках. Речь идет о таких метриках, как RMSE, MSE, MAE, MdAE, RMSLE, MSLE.

Давайте поговорим о каждой метрике по порядку.

2.2.1. Среднеквадратичная ошибка (mean squared error, MSE)

$$\frac{1}{n} \sum_{t=1}^n (A_t - F_t)^2,$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Давайте вычислим MSE для нашего примера вручную с помощью самостоятельно написанной функции и автоматически.

```
# пишем функцию для вычисления MSE
def mse(actual, predicted):
    return np.mean((actual - predicted) ** 2)
```

```
# вручную вычисляем MSE
MSE = mse(data['fact'], data['pred'])
print("MSE: %.3f" % MSE)

MSE: 0.354
```

В библиотеке scikit-learn функция `mean_squared_error` позволяет вычислить RMSE (если параметр `squared=False`) и MSE (если параметр `squared=True`).

```
# автоматически вычисляем MSE
from sklearn.metrics import mean_squared_error
MSE = mean_squared_error(data['fact'], data['pred'])
print("MSE: %.3f" % MSE)

MSE: 0.354
```

2.2.2. Корень из среднеквадратичной ошибки (root mean squared error, RMSE)

$$RMSE = \sqrt{\frac{1}{n} \sum_{t=1}^n (A_t - F_t)^2},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели. RMSE часто используется вместо MSE, для того чтобы получить ошибку такой же размерности, что и у интересующей нас переменной.

Давайте вычислим RMSE для нашего примера вручную с помощью самостоятельно написанной функции и автоматически.

```
# пишем функцию для вычисления RMSE
def rmse(actual, predicted):
    return np.sqrt(np.mean((actual - predicted) ** 2))

# вручную вычисляем RMSE
RMSE = rmse(data['fact'], data['pred'])
print("RMSE: %.3f" % RMSE)

RMSE: 0.595

# автоматически вычисляем RMSE
RMSE = mean_squared_error(data['fact'], data['pred'],
                           squared=False)
print("RMSE: %.3f" % RMSE)

RMSE: 0.595
```

2.2.3. Средняя абсолютная ошибка (mean absolute error, MAE)

$$MAE = \frac{1}{n} \sum_{t=1}^n |A_t - F_t|,$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели.

Давайте вычислим MAE для нашего примера вручную с помощью самостоятельно написанной функции и автоматически.

```
# пишем функцию для вычисления MAE
def mae(actual, predicted):
    return np.mean(np.abs((actual - predicted)))

# вручную вычисляем MAE
MAE = mae(data['fact'], data['pred'])
print("MAE: %.3f" % MAE)

MAE: 0.510

# автоматически вычисляем MAE
from sklearn.metrics import mean_absolute_error
MAE = mean_absolute_error(data['fact'], data['pred'])
print("MAE: %.3f" % MAE)

MAE: 0.510
```

2.2.4. Медианная абсолютная ошибка (median absolute error, MdAE)

$$MdAE = \text{median}(|A_t - F_t|),$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t .

Чем меньше значение метрики, тем лучше качество модели.

Давайте вычислим MdAE для нашего примера вручную с помощью самостоятельно написанной функции.

```
# пишем функцию для вычисления MdAE
def mdae(actual, predicted):
    return np.median(np.abs(actual - predicted))

# вручную вычисляем MdAE
MdAE = mdae(data['fact'], data['pred'])
print("MdAE: %.3f" % MdAE)

MdAE: 0.501
```

2.2.5. Сравнение RMSE, MAE, MdAE

RMSE и MAE показывают усредненную ошибку прогноза модели, MdAE показывает медианную ошибку прогноза модели. При этом все три метрики имеют ту же единицу измерения, что и предсказываемая величина (т. е. зависят от единицы измерения предсказываемой величины).

Все три метрики могут варьировать от 0 до ∞ и игнорируют направление колебания данных. Меньшие значения этих метрик указывают на лучшее качество модели.

Все три метрики являются симметричными, т. е. одинаково штрафуют заниженные и завышенные прогнозы.

Метрики не становятся огромными и не перестают определяться, когда фактические значения приближаются к нулю или равны нулю.

Главное ограничение метрик заключается в том, что, будучи усредненными (как RMSE и MAE) или взятыми по медиане (как MdAE), они не показывают максимального отклонения прогноза от действительности, хотя часто именно максимальная ошибка важна для исследователя.

В RMSE ошибки возводятся в квадрат перед их усреднением, поэтому RMSE придает относительно большой вес крупным ошибкам. Несомненным преимуществом MAE является то, что модули не увеличивают в разы ошибки. Поэтому оценка с помощью MAE является более robustной, устойчивой к большим ошибкам прогнозов. Она будет более полезной, если нас интересует, как мы прогнозируем в среднем, несмотря на некоторые большие ошибки прогноза. При решении задачи регрессии с метрикой качества MAE при помощи ансамбля вместо усреднения нескольких алгоритмов полезно брать их медиану – это, как правило, повышает качество. RMSE будет более полезной метрикой, когда для нас особенно нежелательны большие ошибки прогноза. Поэтому все будет зависеть от задачи. MdAE будет еще более robustной, чем MAE, что уже может стать проблемой: из-за того, что метрику всегда будет интересовать медиана разностей, можно получить идеальное значение метрики при наличии очень больших ошибок. Именно поэтому MAE более популярна, чем MdAE.

Приведем примеры, на которых как раз проиллюстрируем чувствительность метрик RMSE, MAE и MdAE к большим значениям ошибок.

Нам нужно создать наборы с различными распределениями ошибок. Поскольку RMSE, MAE и MdAE являются симметричными метриками, без разницы, какой является ошибка – отрицательной или положительной. Положительная ошибка – это когда фактическое значение больше прогноза (недооценка).

Фактическое значение 20, прогноз 10, $20 - 10 = 10$. Отрицательная ошибка – это когда фактическое значение меньше прогноза (переоценка). Фактическое значение 10, прогноз 20, $10 - 20 = -10$. Сейчас мы создадим наборы с разными распределениями положительных ошибок.

Сначала создадим набор данных с равномерно распределенными небольшими положительными ошибками.

```
# создаем набор с равномерно распределенными
# небольшими ошибками
exmpl = pd.DataFrame(
    {'Actual': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
     'Forecast': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]})

exmpl['error'] = exempl['Actual'] - exempl['Forecast']
exmpl['error^2'] = (exmpl['Actual'] - exempl['Forecast']) ** 2
```

	Actual	Forecast	error	error^2
0	3	1	2	4
1	4	2	2	4
2	5	3	2	4
3	6	4	2	4
4	7	5	2	4
5	8	6	2	4
6	9	7	2	4
7	10	8	2	4
8	11	9	2	4
9	12	10	2	4

```
# печатаем значения метрик
print("RMSE", mean_squared_error(
    exempl['Actual'], exempl['Forecast'], squared=False))
print("MAE", mean_absolute_error(
    exempl['Actual'], exempl['Forecast']))
print("MdAE", mdae(
    exempl['Actual'], exempl['Forecast']))
```

RMSE 2.0
MAE 2.0
MdAE 2.0

Видим, что при равномерно распределенных ошибках значения всех трех метрик одинаковы.

Теперь создадим набор с небольшим варьированием размеров положительных ошибок.

```
# создаем набор с небольшим варьированием
# размеров положительных ошибок
exmpl2 = pd.DataFrame(
    {'Actual': [2, 3, 4, 5, 6, 9, 10, 11, 12, 13],
     'Forecast': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]})

exmpl2['error'] = exempl2['Actual'] - exempl2['Forecast']
exmpl2['error^2'] = (exmpl2['Actual'] - exempl2['Forecast']) ** 2
exmpl2
```

	Actual	Forecast	error	error^2
0	2	1	1	1
1	3	2	1	1
2	4	3	1	1
3	5	4	1	1
4	6	5	1	1
5	9	6	3	9
6	10	7	3	9
7	11	8	3	9
8	12	9	3	9
9	13	10	3	9

```
# печатаем значения метрик
print("RMSE", mean_squared_error(
    exempl2['Actual'], exempl2['Forecast'], squared=False))
print("MAE", mean_absolute_error(
    exempl2['Actual'], exempl2['Forecast']))
print("MdAE", mdae(
    exempl2['Actual'], exempl2['Forecast']))
```

RMSE 2.23606797749979

MAE 2.0

MdAE 2.0

При небольшом варьировании размеров ошибок RMSE немного возрастает.
Создаем набор с одним очень большим значением положительной ошибки (выбросом).

```
# создаем набор с одним очень большим значением
# положительной ошибки (выбросом)
exmpl3 = pd.DataFrame(
    {'Actual': [1, 2, 3, 4, 5, 6, 7, 8, 9, 30],
     'Forecast': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]})

exmpl3['error'] = exempl3['Actual'] - exempl3['Forecast']
exmpl3['error^2'] = (exmpl3['Actual'] - exempl3['Forecast']) ** 2
exmpl3
```

	Actual	Forecast	error	error^2
0	1	1	0	0
1	2	2	0	0
2	3	3	0	0
3	4	4	0	0
4	5	5	0	0
5	6	6	0	0
6	7	7	0	0
7	8	8	0	0
8	9	9	0	0
9	30	10	20	400

```
# печатаем значения метрик
print("RMSE", mean_squared_error(
    exempl3['Actual'], exempl3['Forecast'], squared=False))
print("MAE", mean_absolute_error(
    exempl3['Actual'], exempl3['Forecast']))
print("MdAE", mdae(
    exempl3['Actual'], exempl3['Forecast']))

RMSE 6.324555320336759
MAE 2.0
MdAE 0.0
```

На наличие одной очень большой ошибки сильнее всего отреагировала RMSE, она увеличилась. Оценка MAE осталась неизменной, а вот MdAE стала идеальной, произошло то, о чём мы писали выше: мы получили идеальное значение метрики при наличии очень большой ошибки.

Приведем еще пример. У нас есть данные по обороту за 10 месяцев:

100, 100, 100, 100, 100, 100, 100, 100, 100, 100.

При этом нам важно, как мы прогнозируем в среднем, игнорируя отдельные большие ошибки.

Мы строим две модели. Используем две метрики для оценки прогнозов – RMSE и MAE.

Первая модель дала прогнозы:

300, 300, 300, 300, 300, 300, 300, 300, 300, 300.

Во всех наблюдениях прогнозы отличаются от фактических значений, но больших ошибок нет.

Вторая модель дала прогнозы:

100, 100, 100, 100, 100, 100, 100, 100, 100, 900.

Для 9 наблюдений прогнозы не отличаются от фактических значений, но в 10-м наблюдении у нас большая ошибка.

Вычисляем RMSE, MAE и MdAE для обеих моделей.

	RMSE	MAE	MdAE
Модель 1	200,0	200,0	200
Модель 2	284,604989	90,0	0

С точки зрения RMSE лучшей является первая модель. С точки зрения MAE лучшей является вторая модель. С точки зрения MdAE лучшей является вторая модель, она просто идеальна с точки зрения MdAE. Вспоминаем, что нам важно, как мы прогнозируем *в среднем*, игнорируя отдельные большие ошибки. Поэтому для нас приоритетной метрикой будет MAE и лучшей будет вторая модель. Вторая модель, хотя и один раз серьезно ошиблась, в среднем оказалась точнее: фактический суммарный оборот составляет 1000, первая модель предсказала 3000, вторая модель предсказала 1900.

```
# создаем массив фактических значений
actual = np.array([100] * 10)

# создаем массивы прогнозов
pred_1 = np.array([300] * 10)
pred_2 = np.array([100] * 9 + [1000])

# автоматически вычисляем RMSE
RMSE_model1 = mean_squared_error(actual, pred_1, squared=False)
RMSE_model2 = mean_squared_error(actual, pred_2, squared=False)
# автоматически вычисляем MAE
MAE_model1 = mean_absolute_error(actual, pred_1)
MAE_model2 = mean_absolute_error(actual, pred_2)
# автоматически вычисляем MDAE
MdAE_model1 = mdae(actual, pred_1)
MdAE_model2 = mdae(actual, pred_2)

# создаем датафрейм, строки - модели, столбцы - метрики
data_dict = {'RMSE': [RMSE_model1, RMSE_model2],
             'MAE': [MAE_model1, MAE_model2],
             'MdAE': [MdAE_model1, MdAE_model2]}
df = pd.DataFrame(data_dict, index=['model1', 'model2'])
df
```

	RMSE	MAE	MdAE
model1	200.000	200.000	200.000
model2	284.605	90.000	0.000

2.2.6. Корень из среднеквадратичной логарифмической ошибки (root mean squared logarithmic error, RMSLE)

$$RMSLE = \sqrt{\frac{1}{n} \sum_{t=1}^n (\log(F_t + 1) - \log(A_t + 1))^2},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели.

Давайте вычислим RMSLE для нашего примера с задолженностью по кредитной карте вручную с помощью самостоятельно написанной функции и автоматически.

В библиотеке scikit-learn функция `mean_squared_log_error` позволяет вычислить RMSLE (если параметр `squared=False`) и MSLE (если параметр `squared=True`). Программный код функции выглядит следующим образом:

```
mean_squared_error(
    np.log1p(y_true),
    np.log1p(y_pred),
    squared=True)

# пишем функцию для вычисления RMSLE
def rmsle(actual, predicted):
    return np.sqrt(np.mean((np.log(predicted + 1) -
                           np.log(actual + 1)) ** 2))

# вручную вычисляем RMSLE
RMSLE = rmsle(data['fact'], data['pred'])
print("RMSLE: %.3f" % RMSLE)
```

RMSLE: 0.302

```
# автоматически вычисляем RMSLE
from sklearn.metrics import mean_squared_log_error
RMSLE = np.sqrt(mean_squared_log_error(
    data['fact'], data['pred']))
print("RMSLE: %.3f" % RMSLE)
```

RMSLE: 0.302

Особенность RMSLE заключается в том, что она штрафует прогнозные значения, оказавшиеся меньше фактических (недооценка или положительная ошибка), сильнее, чем прогнозные значения, оказавшиеся больше фактических.

Давайте сравним RMSE, MSLE и RMSLE в случаях недооценки и переоценки.

```

# зададим фактическое и спрогнозированные
# значения для случая недооценки
actual1 = 1000
predicted1 = 600

# зададим фактическое и спрогнозированные
# значения для случая переоценки
actual2 = 1000
predicted2 = 1400

# сравним поведение RMSE, MSLE и RMSLE
# в случаях недооценки и переоценки
RMSE1 = np.sqrt(np.sum((actual1 - predicted1) ** 2))
MSLE1 = mean_squared_log_error([actual1], [predicted1])
RMSLE1 = rmsle(actual1, predicted1)
RMSE2 = np.sqrt(np.sum((actual2 - predicted2) ** 2))
MSLE2 = mean_squared_log_error([actual2], [predicted2])
RMSLE2 = rmsle(actual2, predicted2)

# создаем датафрейм, строки - модели, столбцы - метрики
data_dict = {'RMSE': [RMSE1, RMSE2],
             'MSLE': [MSLE1, MSLE2],
             'RMSLE': [RMSLE1, RMSLE2]}
df = pd.DataFrame(data_dict, index=['прогноз меньше фактического значения',
                                     'прогноз больше фактического значения'])
df

```

	RMSE	MSLE	RMSLE
прогноз меньше фактического значения	400.000	0.260	0.510
прогноз больше фактического значения	400.000	0.113	0.336

← В отличие от RMSE за недооценку MSLE и RMSLE штрафуют сильнее

Теперь, помимо наших наборов с различными распределениями положительных ошибок, мы еще создадим несколько наборов.

Создадим набор с одной большой отрицательной и одной большой положительной ошибками (выбросами).

```

# создаем набор с одной большой положительной и одной
# большой отрицательной ошибками (выбросами)
exmpl4 = pd.DataFrame(
    {'Actual': [1, 2, 3, 4, 5, 6, 25, 8, 9, 30],
     'Forecast': [1, 2, 3, 4, 5, 6, 45, 8, 9, 10]})

exmpl4['error'] = exempl4['Actual'] - exempl4['Forecast']
exmpl4['error^2'] = (exmpl4['Actual'] - exempl4['Forecast']) ** 2
exmpl4

```

	Actual	Forecast	error	error^2
0	1	1	0	0
1	2	2	0	0
2	3	3	0	0
3	4	4	0	0
4	5	5	0	0
5	6	6	0	0
6	25	45	-20	400
7	8	8	0	0
8	9	9	0	0
9	30	10	20	400

Создадим набор данных с равномерно распределенными небольшими отрицательными ошибками.

```
# создаем набор с равномерно распределенными
# небольшими отрицательными ошибками
exmpl5 = pd.DataFrame(
    {'Actual': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
     'Forecast': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]})
exmpl5['error'] = exempl5['Actual'] - exempl5['Forecast']
exmpl5['error^2'] = (exempl5['Actual'] - exempl5['Forecast']) ** 2
exmpl5
```

	Actual	Forecast	error	error^2
0	1	3	-2	4
1	2	4	-2	4
2	3	5	-2	4
3	4	6	-2	4
4	5	7	-2	4
5	6	8	-2	4
6	7	9	-2	4
7	8	10	-2	4
8	9	11	-2	4
9	10	12	-2	4

Создадим набор с небольшим варьированием размеров отрицательных ошибок.

```
# создаем набор с небольшим варьированием
# размеров отрицательных ошибок
exmpl6 = pd.DataFrame({
    'Actual': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Forecast': [2, 3, 4, 5, 6, 9, 10, 11, 12, 13]})
exmpl6['error'] = exempl6['Actual'] - exempl6['Forecast']
exmpl6['error^2'] = (exmpl6['Actual'] - exempl6['Forecast']) ** 2
exmpl6
```

	Actual	Forecast	error	error^2
0	1	2	-1	1
1	2	3	-1	1
2	3	4	-1	1
3	4	5	-1	1
4	5	6	-1	1
5	6	9	-3	9
6	7	10	-3	9
7	8	11	-3	9
8	9	12	-3	9
9	10	13	-3	9

Создаем набор с одним очень большим значением отрицательной ошибки (выбросом).

```
# создаем набор с одним очень большим значением
# отрицательной ошибки (выбросом)
exmpl7 = pd.DataFrame({
    'Actual': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Forecast': [1, 2, 3, 4, 5, 6, 7, 8, 9, 30]})
exmpl7['error'] = exempl7['Actual'] - exempl7['Forecast']
exmpl7['error^2'] = (exmpl7['Actual'] - exempl7['Forecast']) ** 2
exmpl7
```

	Actual	Forecast	error	error^2
0	1	1	0	0
1	2	2	0	0
2	3	3	0	0
3	4	4	0	0
4	5	5	0	0
5	6	6	0	0
6	7	7	0	0
7	8	8	0	0
8	9	9	0	0
9	10	30	-20	400

В отличие от RMSE, RMSLE более устойчива к выбросам. Давайте посмотрим поведение обеих метрик на наборах с различными распределениями ошибок.

```
# вычисляем RMSE и RMSLE
RMSE = rmse(exmpl['Actual'], exempl['Forecast'])
RMSE2 = rmse(exmpl2['Actual'], exempl2['Forecast'])
RMSE3 = rmse(exmpl3['Actual'], exempl3['Forecast'])
RMSE4 = rmse(exmpl4['Actual'], exempl4['Forecast'])
RMSE5 = rmse(exmpl5['Actual'], exempl5['Forecast'])
RMSE6 = rmse(exmpl6['Actual'], exempl6['Forecast'])
RMSE7 = rmse(exmpl7['Actual'], exempl7['Forecast'])

RMSLE = rmsle(exmpl['Actual'], exempl['Forecast'])
RMSLE2 = rmsle(exmpl2['Actual'], exempl2['Forecast'])
RMSLE3 = rmsle(exmpl3['Actual'], exempl3['Forecast'])
RMSLE4 = rmsle(exmpl4['Actual'], exempl4['Forecast'])
RMSLE5 = rmsle(exmpl5['Actual'], exempl5['Forecast'])
RMSLE6 = rmsle(exmpl6['Actual'], exempl6['Forecast'])
RMSLE7 = rmsle(exmpl7['Actual'], exempl7['Forecast'])

# создаем датафрейм, строки - модели, столбцы - метрики
data_dict = {'RMSE': [RMSE, RMSE2, RMSE3, RMSE4, RMSE5, RMSE6, RMSE7],
             'RMSLE': [RMSLE, RMSLE2, RMSLE3, RMSLE4, RMSLE5, RMSLE6, RMSLE7]}
df = pd.DataFrame(
    data_dict,
    index=['равномерно распределенные положительные ошибки',
           'небольшое варьирование размеров положительных ошибок',
           'одна большая положительная ошибка',
           'одна большая положительная и одна большая отрицательная',
           'равномерно распределенные отрицательные ошибки',
           'небольшое варьирование размеров отрицательных ошибок',
           'одна большая отрицательная ошибка'])
df
```

	RMSE	RMSLE
равномерно распределенные положительные ошибки	2.000000	0.362787
небольшое варьирование размеров положительных ошибок	2.236068	0.281486
одна большая положительная ошибка	6.324555	0.327641
одна большая положительная и одна большая отрицательная	8.944272	0.374033
равномерно распределенные отрицательные ошибки	2.000000	0.362787
небольшое варьирование размеров отрицательных ошибок	2.236068	0.281486
одна большая отрицательная ошибка	6.324555	0.327641

Действительно, по сравнению с RMSE метрика RMSLE более устойчива к выбросам. Интересно отметить, что при равномерно распределенных ошибках метрика RMSLE возрастает сильнее, чем при одной большой ошибке.

2.3. МЕТРИКИ КАЧЕСТВА НА ОСНОВЕ ПРОЦЕНТНЫХ ОШИБОК (MAPE, MdAPE, sMAPE, sMdAPE, WAPE, WMAPE, RMSPE, RMdSPE)

Процентная ошибка определяется как $p_t = 100e_t/Y_t$, где e_t – разница между фактическим значением и прогнозом для t -го наблюдения или момента времени t и Y_t – фактическое значение для t -го наблюдения или момента времени t . Преимущество процентных ошибок заключается в том, что они не зависят от масштаба и поэтому часто используются для сравнения эффективности прогнозов по разным наборам данных. Наиболее часто используемые метрики – MAPE, MdAPE, sMAPE, sMdAPE, RMSPE, RMdSPE. Они популярны в прогнозировании временных рядов.

2.3.1. Средняя абсолютная процентная ошибка (mean absolute percentage error, MAPE)

$$MAPE = \frac{100\%}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right|, \quad \text{или} \quad MAPE = \frac{1}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right|,$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели. Показатель можно измерять в долях или процентах. Если у вас получилось, например, что MAPE = 11,4 %, это говорит о том, что ошибка составила 11,4 % от фактических значений. MAPE не зависит от единицы измерения предсказываемой величины.

Допустим, у нас есть фактические данные продаж и прогнозы с января по октябрь. Измерим MAPE для каждого месяца.

	Январь	Февраль	Март	Апрель	Май	Июнь	Июль	Август	Сентябрь	Октябрь
Фактические продажи	30	20	20	30	10	30	1	0,1	0	1
Прогноз	30	30	10	300	20	20	2	1	1	0
MAPE	0	0,5	0,5	9	1	0,33	1	9	4.503600e+15	1

Рис. 62 Поведение MAPE при различных ситуациях

В таблице видны недостатки MAPE.

Если фактическое значение временного ряда равно 1 или меньше 1, то в знаменателе окажется очень маленькое число и значение MAPE резко возрастет. Если

фактическое значение временного ряда будет равно 0, метрика MAPE будет либо не определена, либо огромна (у нас или происходит деление на ноль и выдается ошибка, или ноль заменяется на очень маленькое значение, близкое к 0, однако чем оно меньше, тем больше MAPE). Посмотрите на прогнозы с июля по октябрь. Несмотря на то что прогнозы довольно близки к фактическим значениям, метрика MAPE принимает большие значения. Некоторые программы при вычислении MAPE просто отбрасывают нулевые значения, что не является хорошей практикой.

Теперь посмотрите на прогнозы для мая и июня. MAPE по-разному относится к положительным и отрицательным ошибкам. За переоценку, т. е. отрицательные ошибки ($A_t < F_t$, например $10 < 20$), MAPE штрафует сильнее. Это обусловлено тем, что для слишком низких прогнозов ошибка не может превышать 100 %, но для слишком высоких прогнозов нет верхнего предела ошибки.

Допустим, фактическое значение равно 10, а наш точечный прогноз составил 20. Какой будет ошибка MAPE в этом случае? Подставим эти значения в формулу, чтобы получить $|(10 - 20) / 10| = 1 = 100\%$. А какой была бы ошибка, если бы фактическое значение было равно 30? Очевидно, что $|(30 - 20) / 30| = 0,333 = 33,3\%$. Вроде бы ошибка та же, но по объективным причинам она во втором случае составляет 33,3 % от фактического значения, а не 100 %. Действительно, MAPE жестче относится к случаям завышенных прогнозов, чем заниженных (подтвердили, что отрицательные ошибки штрафуются сильнее, чем положительные). Ряд исследователей считают такое поведение «асимметрией», одной из проблем MAPE. Другие исследователи спорят с первыми, указывая на то, что не важно, какие у нас ошибки – отрицательные или положительные, просто в рассматриваемом случае меняются фактические значения, а когда фактические значения являются одинаковыми, метрика MAPE будет одинаковой для обеих ошибок. Допустим, фактическое значение равно 20, а наш точечный прогноз составил 30. Какой будет ошибка MAPE в этом случае? Подставим эти значения в формулу, чтобы получить $|(20 - 30) / 20| = 0,5 = 50\%$. А какой была бы ошибка, если бы прогноз был равен 10? Видим, что ошибка составит $|(20 - 10) / 20| = 0,5 = 50\%$. Вместе с тем отметим, если MAPE используется для сравнения точности прогнозных моделей, он будет систематически выбирать модель, прогнозы которой занижены.

Теперь посмотрите на прогнозы для апреля и августа. В первом случае ошибка прогноза составляет 270, а во втором случае ошибка прогноза составляет 0,9, но в обоих случаях MAPE = 900,0 %.

Давайте напишем собственную функцию для вычисления MAPE.

```
# напишем функцию для вычисления MAPE
def custom_mape(actual, forecast):
    mape = np.mean(np.abs((actual - forecast) / actual))
    return mape
```

Допустим, у нас есть фактическое значение 50 и прогноз 60. С помощью нашей функции вычислим MAPE.

```
# вычисляем MAPE
custom_mape(50, 60)
```

Если мы вручную попробуем вычислить MAPE, когда фактическое значение равно 0, из-за 0 в знаменателе у нас произойдет деление на ноль, и мы получим ошибку.

```
# вручную вычисляем MAPE, когда фактическое значение равно 0,
# из-за 0 в знаменателе у нас - деление на ноль, и получаем ошибку
MAPE = np.mean(np.abs((0 - 1) / 0)) * 100
print("MAPE: %.3f" % MAPE)

ZeroDivisionError                                     Traceback (most recent call last)
<ipython-input-60-f8c02fd7568b> in <module>
      1 # вручную вычисляем MAPE
      2 MAPE = np.mean(
----> 3     np.abs((0 - 1) / 0)) * 100
      4 print("MAPE: %.3f" % MAPE)
ZeroDivisionError: division by zero
```

Теперь мы опять вычислим MAPE, когда фактическое значение равно 0, но вместо 0 подставим маленькое значение, близкое к 0, однако чем оно меньше, тем больше MAPE.

```
# вручную вычисляем MAPE, когда фактическое значение равно 0,
# вместо 0 подставляем маленькое значение, близкое к 0,
# однако чем оно меньше, тем больше MAPE
MAPE = np.mean(np.abs((0 - 1) / 0.000000001)) * 100
print("MAPE: %.3f" % MAPE)

MAPE: 100000000000.000
```

В библиотеке scikit-learn для вычисления MAPE используется аналогичный подход.

```
# рассмотрим функцию для вычисления MAPE в scikit-learn
def sklearn_mape(actual, forecast):
    # задаем эпсилон - очень маленькое значение
    # (2.220446049250313e-16)
    epsilon = np.finfo(np.float64).eps
    # в знаменателе берем максимальное значение из массива
    # двух чисел - модуля числа и эпсилон
    output = np.abs(forecast - actual) / np.maximum(
        np.abs(actual), epsilon)
    # усредняем
    mape = np.average(output)
    return mape

# вычисляем MAPE с помощью нашей функции sklearn_mape(),
# когда фактическое значение равно 0
sklearn_mape(0, 1)

4503599627370496.0

# вычислим MAPE с помощью функции mean_absolute_percentage_error(),
# когда фактическое значение равно 0
mean_absolute_percentage_error([0], [1])

4503599627370496.0
```

Выведем таблицу с поведением MAPE для различных случаев.

```
# отключаем экспоненциальное представление
pd.set_option('display.float_format', lambda x: '%.3f' % x)

# вычисляем значения MAPE
mape1 = sklearn_mape(30, 30)
mape2 = sklearn_mape(20, 30)
mape3 = sklearn_mape(20, 10)
mape4 = sklearn_mape(30, 300)
mape5 = sklearn_mape(10, 20)
mape6 = sklearn_mape(30, 20)
mape7 = sklearn_mape(1, 2)
mape8 = sklearn_mape(2, 1)
mape9 = sklearn_mape(0.1, 1)
mape10 = sklearn_mape(1, 0.1)
mape11 = sklearn_mape(0, 1)
mape12 = sklearn_mape(1, 0)

# создаем датафрейм
data_dict = {'MAPE': [mape1, mape2, mape3, mape4, mape5, mape6,
                      mape7, mape8, mape9, mape10, mape11, mape12]}
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                          'фактическое=20, прогноз=30 (переоценка)',
                          'фактическое=20, прогноз=10 (недооценка)',
                          'фактическое=30, прогноз=300 (большая ошибка)',
                          'фактическое=10, прогноз=20 (переоценка)',
                          'фактическое=30, прогноз=20 (недооценка)',
                          'фактическое=1, прогноз=2 (переоценка)',
                          'фактическое=2, прогноз=1 (недооценка)',
                          'фактическое=0.1, прогноз=1 (переоценка)',
                          'фактическое=1, прогноз=0.1 (недооценка)',
                          'фактическое=0, прогноз=1 (переоценка)',
                          'фактическое=1, прогноз=0 (недооценка)'])

df
```

MAPE	
фактическое=30, прогноз=30 (идеально)	0.000
фактическое=20, прогноз=30 (переоценка)	0.500
фактическое=20, прогноз=10 (недооценка)	0.500
фактическое=30, прогноз=300 (большая ошибка)	9.000
фактическое=10, прогноз=20 (переоценка)	1.000
фактическое=30, прогноз=20 (недооценка)	0.333
фактическое=1, прогноз=2 (переоценка)	1.000
фактическое=2, прогноз=1 (недооценка)	0.500
фактическое=0.1, прогноз=1 (переоценка)	9.000
фактическое=1, прогноз=0.1 (недооценка)	0.900
фактическое=0, прогноз=1 (переоценка)	4503599627370496.000
фактическое=1, прогноз=0 (недооценка)	1.000

Ошибки одинаковы и штрафуются одинаково (одинаковые фактические значения)

Выброс увеличивает ошибку

Ошибки одинаковы, а штрафуются по-разному (одинаковые прогнозы)

Резкое возрастание, когда фактическое значение временного ряда равно 1 или меньше 1, огромное значение для нулевого фактического значения

Анализируя таблицу, мы видим резкое возрастание MAPE, когда фактическое значение равно 1 или меньше 1, и наблюдаем огромное значение MAPE для нулевого фактического значения. Мы также видим, что MAPE возрастает в случае большой ошибки. Наконец, мы видим ситуации, когда одинаковые ошибки получают разные значения MAPE, а также ситуации, когда одинаковые ошибки получают одинаковые значения MAPE.

Выше мы говорили: если MAPE используется для сравнения точности прогнозных моделей, он будет систематически выбирать метод, прогнозы которого слишком занижены. Давайте убедимся в этом.

Сейчас мы создадим два набора – набор с преимущественно положительными ошибками (заниженными прогнозами) и набор с преимущественно отрицательными ошибками (занятыми прогнозами), а затем вычислим MAPE для них.

```
# создаем набор с преимущественно положительными
# ошибками (заниженные прогнозы)
exmpl_underest = pd.DataFrame(
    {'Actual': [3, 4, 3, 6, 7, 8, 9, 10, 11, 12],
     'Forecast': [1, 2, 5, 4, 5, 6, 7, 8, 9, 12]})

exmpl_underest['error'] = (exmpl_underest['Actual'] -
                           exmpl_underest['Forecast'])
exmpl_underest['error^2'] = (exmpl_underest['Actual'] -
                            exmpl_underest['Forecast']) ** 2

exmpl_underest
```

	Actual	Forecast	error	error^2
0	3	1	2	4
1	4	2	2	4
2	3	5	-2	4
3	6	4	2	4
4	7	5	2	4
5	8	6	2	4
6	9	7	2	4
7	10	8	2	4
8	11	9	2	4
9	12	12	0	0

```
# создаем набор с преимущественно отрицательными
# ошибками (занятые прогнозы)
exmpl_overest = pd.DataFrame(
    {'Actual': [1, 2, 5, 4, 5, 6, 7, 8, 9, 12],
     'Forecast': [3, 4, 3, 6, 7, 8, 9, 10, 11, 12]})

exmpl_overest['error'] = (exmpl_overest['Actual'] -
                           exmpl_overest['Forecast'])
exmpl_overest['error^2'] = (exmpl_overest['Actual'] -
                            exmpl_overest['Forecast']) ** 2

exmpl_overest
```

	Actual	Forecast	error	error^2
0	1	3	-2	4
1	2	4	-2	4
2	5	3	2	4
3	4	6	-2	4
4	5	7	-2	4
5	6	8	-2	4
6	7	9	-2	4
7	8	10	-2	4
8	9	11	-2	4
9	12	12	0	0

```
# вычисляем MAPE для заниженных прогнозов
mape_for_underest = sklearn_mape(exmpl_underest['Actual'],
                                  exmpl_underest['Forecast'])

# вычисляем MAPE для завышенных прогнозов
mape_for_overest = sklearn_mape(exmpl_overest['Actual'],
                                  exmpl_overest['Forecast'])

# печатаем значения MAPE
print("MAPE для заниженных прогнозов: %.3f" % mape_for_underest)
print("MAPE для завышенных прогнозов: %.3f" % mape_for_overest)

MAPE для заниженных прогнозов: 0.331
MAPE для завышенных прогнозов: 0.539
```

Для набора с заниженными прогнозами мы получаем меньшее значение MAPE.

Еще одна проблема, связанная с MAPE и другими процентными ошибками, которую часто упускают из виду, заключается в том, что они предполагают, что единица измерения имеет *естественную нулевую точку* (точку отсчета). Например, процентная ошибка не имеет смысла при измерении точности прогнозов температуры по шкале Фаренгейта или Цельсия, потому что температура имеет *условную нулевую точку*. Ноль по шкале Фаренгейта определяется по самоподдерживающейся температуре смеси воды, льда и хлорида аммония (соответствует примерно $-17,8^{\circ}\text{C}$). Ноль шкалы Цельсия установлен таким образом, что температура тройной точки воды равна $0,01^{\circ}\text{C}$. Таким образом, MAPE нельзя применять к шкале интервалов, которая состоит из одинаковых интервалов и имеет условную нулевую точку, но можно применять к шкале отношений, которая отличается от шкалы интервалов тем, что имеет естественную нулевую точку. Шкалы большинства физических величин (длина, масса, сила, возраст, рост, давление, скорость и др.) являются шкалами отношений.

В случае разницы в объемах продаж (например, у вас могут быть высокие продажи быстро распродаваемого товара и гораздо меньшие объемы продаж медленно распродаваемого товара) метрика MAPE может дезориентировать

аналитика. Она мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

Допустим, у нас есть данные продаж по двум товарам, с помощью какой-то модели получили прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	101 000 000	101 120 000
MAPE	0,2	0,01	0,105
Ошибка прогноза	20 000	1 000 000	1 020 000

На уровне прогнозов отдельных товаров мы видим, что большая ошибка прогноза (1 000 000) получает меньшее значение MAPE (0,01). Видим, что MAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

Теперь переходим к итоговым значениям. Фактически мы продаем на 100 100 000 (сто миллионов сто тысяч). Мы прогнозируем продажи на 101 120 000 (сто один миллион сто двадцать тысяч). Ошибка составляет 1 020 000 (один миллион двадцать тысяч). В итоге MAPE составит 0,105.

```
# вычисляем MAPE для первой модели
print("MAPE_1й_товар: %.3f" % sklearn_mape(100000, 120000))
print("MAPE_2й_товар: %.3f" % sklearn_mape(100000000, 101000000))
actual = np.array([100000, 100000000])
forecast = np.array([120000, 101000000])
print("MAPE итого: %.3f" % sklearn_mape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

MAPE_1й_товар: 0.200
MAPE_2й_товар: 0.010
MAPE итого: 0.105
суммарная ошибка прогноза: 1020000
```

С помощью еще одной модели получили новые прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	110 000	105 000 000	105 110 000
MAPE	0,1	0,05	0,075
Ошибка прогноза	10 000	5 000 000	5 010 000

На уровне прогнозов отдельных товаров мы видим, что большая ошибка прогноза (5 000 000) получает меньшее значение MAPE (0,05). Вновь видим, что MAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

Теперь переходим к итоговым значениям. Как и прежде, по факту мы продаем на 100 100 000 (сто миллионов сто тысяч). Мы прогнозируем продажи на 105 110 000 (сто пять миллионов сто десять тысяч). Ошибка составляет 5 010 000 (пять миллионов десять тысяч). В итоге MAPE составит 0,075. Это простое среднее значений MAPE для отдельных продуктов: $(0,1 + 0,05) / 2 = 0,075$.

```
# вычисляем MAPE для второй модели
print("MAPE_1й_товар: %.3f" % sklearn_mape(100000, 110000))
print("MAPE_2й_товар: %.3f" % sklearn_mape(100000000, 105000000))
actual = np.array([100000, 100000000])
forecast = np.array([110000, 105000000])
print("MAPE итого: %.3f" % sklearn_mape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

MAPE_1й_товар: 0.100
MAPE_2й_товар: 0.050
MAPE итого: 0.075
суммарная ошибка прогноза: 5010000
```

Таким образом, при большей ошибке прогноза мы получили меньшее итоговое значение MAPE. Чтобы избежать этого, нужно использовать метрику WAPE. Она взвешивает ошибку с учетом объемов продаж.

Еще приведем пример. Допустим, у нас есть данные продаж по двум товарам, с помощью какой-то модели получили прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	100 020 000	100 140 000
MAPE	0,2	0,0002	0,1001
Ошибка прогноза	20 000	20 000	40 000

Видим, что в обоих случаях ошибка прогноза является одинаковой, но при ошибочном прогнозе товара с меньшим объемом продаж метрика MAPE резко возрастает, а при ошибочном прогнозе товара с большим объемом продаж метрика MAPE почти идеальна. Опять выходит, что MAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

```
# еще один пример
print("MAPE_1й_товар: %.5f" % sklearn_mape(100000, 120000))
print("MAPE_2й_товар: %.5f" % sklearn_mape(100000000, 100020000))
actual = np.array([100000, 100000000])
forecast = np.array([120000, 100020000])
print("MAPE итого: %.5f" % sklearn_mape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

MAPE_1й_товар: 0.20000
MAPE_2й_товар: 0.00020
MAPE итого: 0.10010
суммарная ошибка прогноза: 40000
```

2.3.2. Медианная абсолютная процентная ошибка (median absolute percentage error, MdAPE)

$$MdAPE = \text{median} \left(\left| \frac{A_t - F_t}{A_t} \right| \right),$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t .

Чем меньше значение метрики, тем лучше качество модели. MdAPE наследует все вышеописанные недостатки MAPE.

Давайте напишем собственную функцию для вычисления MdAPE.

```
# рассмотрим функцию для вычисления MDAPE
def mdape(actual, forecast):
    # задаем эпсилон - очень маленькое значение
    # (2.220446049250313e - 16)
    epsilon = np.finfo(np.float64).eps
    # в знаменателе берем максимальное значение из массива
    # двух чисел - модуля числа и эпсилон
    output = np.abs(forecast - actual) / np.maximum(
        np.abs(actual), epsilon)
    # усредняем
    mdape = np.median(output)
    return mdape
```

Выведем таблицу с поведением MdAPE для различных случаев.

```
# вычисляем значения MdAPE
mdape1 = mdape(30, 30)
mdape2 = mdape(20, 30)
mdape3 = mdape(20, 10)
mdape4 = mdape(30, 300)
mdape5 = mdape(10, 20)
mdape6 = mdape(30, 20)
mdape7 = mdape(1, 2)
mdape8 = mdape(2, 1)
mdape9 = mdape(0.1, 1)
mdape10 = mdape(1, 0.1)
mdape11 = mdape(0, 1)
mdape12 = mdape(1, 0)

# создаем датафрейм
data_dict = {'MdAPE': [mdape1, mdape2, mdape3, mdape4, mdape5, mdape6,
                       mdape7, mdape8, mdape9, mdape10, mdape11, mdape12]}
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                          'фактическое=20, прогноз=30 (переоценка)',
                          'фактическое=20, прогноз=10 (недооценка)',
                          'фактическое=30, прогноз=300 (большая ошибка)',
```

```
'фактическое=10, прогноз=20 (переоценка)',  
'фактическое=30, прогноз=20 (недооценка)',  
'фактическое=1, прогноз=2 (переоценка)',  
'фактическое=2, прогноз=1 (недооценка)',  
'фактическое=0.1, прогноз=1 (переоценка)',  
'фактическое=1, прогноз=0.1 (недооценка)',  
'фактическое=0, прогноз=1 (переоценка)',  
'фактическое=1, прогноз=0 (недооценка)'])
```

df

	MdAPE
фактическое=30, прогноз=30 (идеально)	0.000
фактическое=20, прогноз=30 (переоценка)	0.500
фактическое=20, прогноз=10 (недооценка)	0.500
фактическое=30, прогноз=300 (большая ошибка)	9.000
фактическое=10, прогноз=20 (переоценка)	1.000
фактическое=30, прогноз=20 (недооценка)	0.333
фактическое=1, прогноз=2 (переоценка)	1.000
фактическое=2, прогноз=1 (недооценка)	0.500
фактическое=0.1, прогноз=1 (переоценка)	9.000
фактическое=1, прогноз=0.1 (недооценка)	0.900
фактическое=0, прогноз=1 (переоценка)	4503599627370496.000
фактическое=1, прогноз=0 (недооценка)	1.000

Ошибки одинаковы и штрафуются одинаково (одинаковые фактические значения)

Выброс увеличивает ошибку

Ошибки одинаковы, а штрафуются по-разному (одинаковые прогнозы)

Резкое возрастание, когда фактическое значение временного ряда равно 1 или меньше 1, огромное значение для нулевого фактического значения

Анализируя таблицу, мы видим, что MdAPE ведет себя аналогично MAPE.

2.3.3. Симметричная средняя абсолютная процентная ошибка (symmetric mean absolute percentage error, SMAPE)

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(|A_t| + |F_t|)/2}, \quad \text{или} \quad SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(|A_t| + |F_t|)/2};$$

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{2 \times |F_t - A_t|}{(|A_t| + |F_t|)}, \quad \text{или} \quad SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{2 \times |F_t - A_t|}{(|A_t| + |F_t|)},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели. Интерпретация примерно такая же, как и у MAPE: какой процент составляет ошибка от фактических значений интересующей переменной.

Первым схожую формулу предложил Армстронг в 1985 году, назвав показатель «adjusted MAPE». Он не использовал абсолютные значения в знаменателе, предложив следующую формулу (позже была модифицирована Флоресом):

$$SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(A_t + F_t) / 2}.$$

Однако проблема заключалась в том, что показатель мог быть отрицательным (если $A_t + F_t < 0$) или даже не определен (если $A_t + F_t = 0$). Поэтому обще принятый вариант SMAPE предполагает использование абсолютных значений в знаменателе.

В отличие от MAPE, SMAPE имеет нижнюю и верхнюю границы. Формулы

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(|A_t| + |F_t|) / 2} \quad \text{и} \quad SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{2 \times |F_t - A_t|}{(|A_t| + |F_t|)}$$

дают значения в диапазоне от 0 % до 200 %. Однако ошибку в процентах, меняющуюся в диапазоне от 0 % до 100 %, проще интерпретировать. Именно поэтому на практике часто используют формулу, приведенную ниже (деление на 2 или умножение на 2 в знаменателе не используется):

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{|A_t| + |F_t|} \quad \text{или} \quad SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{|A_t| + |F_t|}.$$

Ее можно назвать стандартной формулой SMAPE.

Существует и третий вариант SMAPE, позволяющий получить ошибку в процентах, меняющуюся в диапазоне от 0 % до 100 %.

$$SMAPE = \frac{\sum_{t=1}^n |F_t - A_t|}{\sum_{t=1}^n (A_t + F_t)} \times 100\% \quad \text{или} \quad SMAPE = \frac{\sum_{t=1}^n |F_t - A_t|}{\sum_{t=1}^n (A_t + F_t)}.$$

Давайте напишем функции для вычисления разных вариантов SMAPE.

```
# пишем функцию для вычисления SMAPE по Армстронгу
def adjusted_mape(actual, forecast):
    return np.mean(np.abs(forecast - actual) / ((actual + forecast) / 2))
```

$$SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(A_t + F_t) / 2}.$$

```
# пишем функцию для вычисления SMAPE
# с умножением на 2 в числителе
def smape_with_multiplication_by_2(actual, forecast):
    return np.mean(2.0 * np.abs(actual - forecast) / (
        np.abs(actual) + np.abs(forecast))) * 100
```

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{2 \times |F_t - A_t|}{(|A_t| + |F_t|)} \quad \text{или} \quad SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{2 \times |F_t - A_t|}{(|A_t| + |F_t|)}.$$

```
# пишем функцию для вычисления SMAPE
# с делением на 2 в знаменателе
def smape_with_division_by_2(actual, forecast):
    return np.mean(np.abs(forecast - actual) / (
        np.abs(actual) + np.abs(forecast)) / 2) * 100
```

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(|A_t| + |F_t|) / 2} \quad \text{или} \quad SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{(|A_t| + |F_t|) / 2}.$$

```
# пишем функцию для вычисления стандартной метрики SMAPE
# (без деления на 2 в знаменателе)
def standard_smape(actual, forecast):
    return np.mean(np.abs(forecast - actual) / (
        np.abs(actual) + np.abs(forecast))) * 100
```

$$SMAPE = \frac{100\%}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{|A_t| + |F_t|} \quad \text{или} \quad SMAPE = \frac{1}{n} \sum_{t=1}^n \frac{|F_t - A_t|}{|A_t| + |F_t|}.$$

```
# пишем функцию для вычисления третьего варианта SMAPE
def third_variant_smape(actual, forecast):
    return (np.sum(np.abs(forecast - actual)) / np.sum(
        actual + forecast)) * 100
```

$$SMAPE = \frac{\sum_{t=1}^n |F_t - A_t|}{\sum_{t=1}^n (A_t + F_t)} \times 100 \% \quad \text{или} \quad SMAPE = \frac{\sum_{t=1}^n |F_t - A_t|}{\sum_{t=1}^n (A_t + F_t)}.$$

Выведем таблицу с поведением разных вариантов SMAPE для различных случаев.

```
# вычисляем значения SMAPE по Армстронгу
adjusted_mape1 = adjusted_mape(30, 30)
adjusted_mape2 = adjusted_mape(20, 30)
adjusted_mape3 = adjusted_mape(20, 10)
adjusted_mape4 = adjusted_mape(30, 300)
adjusted_mape5 = adjusted_mape(10, 20)
adjusted_mape6 = adjusted_mape(30, 20)
adjusted_mape7 = adjusted_mape(1, 2)
```

```
adjusted_mape8 = adjusted_mape(2, 1)
adjusted_mape9 = adjusted_mape(0.1, 1)
adjusted_mape10 = adjusted_mape(1, 0.1)
adjusted_mape11 = adjusted_mape(0, 1)
adjusted_mape12 = adjusted_mape(1, 0)

# вычисляем значения SMAPE с делением на 2 в знаменателе
smape1_with_div_by_2 = smape_with_division_by_2(30, 30)
smape2_with_div_by_2 = smape_with_division_by_2(20, 30)
smape3_with_div_by_2 = smape_with_division_by_2(20, 10)
smape4_with_div_by_2 = smape_with_division_by_2(30, 300)
smape5_with_div_by_2 = smape_with_division_by_2(10, 20)
smape6_with_div_by_2 = smape_with_division_by_2(30, 20)
smape7_with_div_by_2 = smape_with_division_by_2(1, 2)
smape8_with_div_by_2 = smape_with_division_by_2(2, 1)
smape9_with_div_by_2 = smape_with_division_by_2(0.1, 1)
smape10_with_div_by_2 = smape_with_division_by_2(1, 0.1)
smape11_with_div_by_2 = smape_with_division_by_2(0, 1)
smape12_with_div_by_2 = smape_with_division_by_2(1, 0)

# вычисляем значения SMAPE с умножением на 2 в числителе
smape1_with_mul_by_2 = smape_with_multiplication_by_2(30, 30)
smape2_with_mul_by_2 = smape_with_multiplication_by_2(20, 30)
smape3_with_mul_by_2 = smape_with_multiplication_by_2(20, 10)
smape4_with_mul_by_2 = smape_with_multiplication_by_2(30, 300)
smape5_with_mul_by_2 = smape_with_multiplication_by_2(10, 20)
smape6_with_mul_by_2 = smape_with_multiplication_by_2(30, 20)
smape7_with_mul_by_2 = smape_with_multiplication_by_2(1, 2)
smape8_with_mul_by_2 = smape_with_multiplication_by_2(2, 1)
smape9_with_mul_by_2 = smape_with_multiplication_by_2(0.1, 1)
smape10_with_mul_by_2 = smape_with_multiplication_by_2(1, 0.1)
smape11_with_mul_by_2 = smape_with_multiplication_by_2(0, 1)
smape12_with_mul_by_2 = smape_with_multiplication_by_2(1, 0)

# вычисляем значения стандартной метрики SMAPE
# (без деления на 2 в знаменателе)
standard_smape1 = standard_smape(30, 30)
standard_smape2 = standard_smape(20, 30)
standard_smape3 = standard_smape(20, 10)
standard_smape4 = standard_smape(30, 300)
standard_smape5 = standard_smape(10, 20)
standard_smape6 = standard_smape(30, 20)
standard_smape7 = standard_smape(1, 2)
standard_smape8 = standard_smape(2, 1)
standard_smape9 = standard_smape(0.1, 1)
standard_smape10 = standard_smape(1, 0.1)
standard_smape11 = standard_smape(0, 1)
standard_smape12 = standard_smape(1, 0)

# вычисляем значения третьего варианта метрики SMAPE
third_variant_smape1 = third_variant_smape(30, 30)
third_variant_smape2 = third_variant_smape(20, 30)
third_variant_smape3 = third_variant_smape(20, 10)
third_variant_smape4 = third_variant_smape(30, 300)
third_variant_smape5 = third_variant_smape(10, 20)
third_variant_smape6 = third_variant_smape(30, 20)
```



```
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                           'фактическое=20, прогноз=30 (переоценка)',
                           'фактическое=20, прогноз=10 (недооценка)',
                           'фактическое=30, прогноз=300 (большая ошибка)',
                           'фактическое=10, прогноз=20 (переоценка)',
                           'фактическое=30, прогноз=20 (недооценка)',
                           'фактическое=1, прогноз=2 (переоценка)',
                           'фактическое=2, прогноз=1 (недооценка)',
                           'фактическое=0.1, прогноз=1 (переоценка)',
                           'фактическое=1, прогноз=0.1 (недооценка)',
                           'фактическое=0, прогноз=1 (переоценка)',
                           'фактическое=1, прогноз=0 (недооценка)'])
```

df

	adjusted_MAPE	SMAPE_with_division_by_2	SMAPE_with_multiplication_by_2	standard_SMAPE	third.variant_SMAPE
фактическое=30, прогноз=30 (идеально)	0.000	0.000	0.000	0.000	0.000
фактическое=20, прогноз=30 (переоценка)	0.400	40.000	40.000	20.000	20.000
фактическое=20, прогноз=10 (недооценка)	0.667	66.667	66.667	33.333	33.333
фактическое=30, прогноз=300 (большая ошибка)	1.636	163.636	163.636	81.818	81.818
фактическое=10, прогноз=20 (переоценка)	0.667	66.667	66.667	33.333	33.333
фактическое=30, прогноз=20 (недооценка)	0.400	40.000	40.000	20.000	20.000
фактическое=1, прогноз=2 (переоценка)	0.667	66.667	66.667	33.333	33.333
фактическое=2, прогноз=1 (недооценка)	0.667	66.667	66.667	33.333	33.333
фактическое=1, прогноз=1 (переоценка)	1.636	163.636	163.636	81.818	81.818
фактическое=1, прогноз=0.1 (недооценка)	1.636	163.636	163.636	81.818	81.818
фактическое=0, прогноз=1 (переоценка)	2.000	200.000	200.000	100.000	100.000
фактическое=1, прогноз=0 (недооценка)	2.000	200.000	200.000	100.000	100.000

Анализируя таблицу, мы видим резкое возрастание SMAPE, когда фактическое значение равно 1 или меньше 1, и наблюдаем максимальное значение SMAPE для нулевого фактического значения. Мы также видим, что SMAPE возрастает в случае большой ошибки. Наконец, мы видим ситуации, когда одинаковые ошибки получают разные значения SMAPE, а также ситуации, когда одинаковые ошибки получают одинаковые значения SMAPE.

Ранее мы говорили, если MAPE используется для сравнения точности прогнозных моделей, она будет систематически выбирать метод, прогнозы которого слишком занижены. Посмотрим, как SMAPE ведет себя в ситуации заниженных и завышенных прогнозов.

```
# вычисляем SMAPE для заниженных прогнозов
smape_for_underest = standard_smape(exmpl_underest['Actual'],
                                         exempl_underest['Forecast'])

# вычисляем SMAPE для завышенных прогнозов
smape_for_overest = standard_smape(exmpl_overest['Actual'],
                                         exempl_overest['Forecast'])

# печатаем значения SMAPE
print("SMAPE для заниженных прогнозов: %.3f" % smape_for_underest)
print("SMAPE для завышенных прогнозов: %.3f" % smape_for_overest)
```

SMAPE для заниженных прогнозов: 19.290

SMAPE для завышенных прогнозов: 19.290

Мы получили одинаковые значения SMAPE для обоих наборов.

В случае разницы в объемах продаж (например, у вас могут быть высокие продажи быстро распродаваемого товара и гораздо меньшие объемы продаж медленно распродаваемого товара) метрика SMAPE так же, как и метрика MAPE, может дезориентировать аналитика. Она мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

Допустим, у нас есть данные продаж по двум товарам, с помощью какой-то модели получили прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	101 000 000	101 120 000
SMAPE	9,091	0,498	4,794
Ошибка прогноза	20 000	1 000 000	1 020 000

На уровне прогнозов отдельных товаров мы видим, что большая ошибка прогноза (1 000 000) получает меньшее значение SMAPE (0,498). SMAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

Теперь переходим к итоговым значениям. Фактически мы продаем на 100 100 000 (сто миллионов сто тысяч). Мы прогнозируем продажи на 101 120 000 (сто один миллион сто двадцать тысяч). Ошибка составляет 1 020 000 (один миллион двадцать тысяч). В итоге SMAPE составит 4,794. Это простое среднее значений SMAPE для отдельных продуктов: $(9,091 + 0,498) / 2 = 4,794$.

```
# вычисляем SMAPE для первой модели
print("SMAPE_1й_товар: %.3f" % standard_smape(100000, 120000))
print("SMAPE_2й_товар: %.3f" % standard_smape(100000000, 101000000))
actual = np.array([100000, 100000000])
forecast = np.array([120000, 101000000])
print("SMAPE итого: %.3f" % standard_smape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

SMAPE_1й_товар: 9.091
SMAPE_2й_товар: 0.498
SMAPE итого: 4.794
суммарная ошибка прогноза: 1020000
```

С помощью еще одной модели получили новые прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	110 000	105 000 000	105 110 000
SMAPE	4,762	2,439	3,6
Ошибка прогноза	10 000	5 000 000	5 010 000

На уровне прогнозов отдельных товаров мы видим, что большая ошибка прогноза (5 000 000) получает меньшее значение SMAPE (2,439). Вновь видим, что SMAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

Теперь переходим к итоговым значениям. Как и прежде, по факту мы проаем на 100 100 000 (сто миллионов сто тысяч). Мы прогнозируем продажи на 105 110 000 (сто пять миллионов сто десять тысяч). Ошибка составляет 5 010 000 (пять миллионов десять тысяч). В итоге SMAPE составит 3,6.

```
# вычисляем MAPE для второй модели
print("SMAPE_1й_товар: %.3f" % standard_smape(100000, 110000))
print("SMAPE_2й_товар: %.3f" % standard_smape(100000000, 105000000))
actual = np.array([100000, 100000000])
forecast = np.array([110000, 105000000])
print("SMAPE итого: %.3f" % standard_smape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))
```

```
SMAPE_1й_товар: 4.762
SMAPE_2й_товар: 2.439
SMAPE итого: 3.600
суммарная ошибка прогноза: 5010000
```

Таким образом, при большей ошибке прогноза мы получили меньшее итоговое значение SMAPE. Чтобы избежать этого, нужно использовать WAPE. Она взвешивает ошибку с учетом объемов продаж.

Еще приведем пример. Допустим, у нас есть данные продаж по двум товарам, с помощью какой-то модели получили прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	100 020 000	100 140 000
SMAPE	9,09	0,01	4,55
Ошибка прогноза	20 000	20 000	40 000

```
# еще один пример
print("SMAPE_1й_товар: %.5f" % standard_smape(100000, 120000))
print("SMAPE_2й_товар: %.5f" % standard_smape(100000000, 100020000))
actual = np.array([100000, 100000000])
forecast = np.array([120000, 100020000])
print("SMAPE итого: %.5f" % standard_smape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

SMAPE_1й_товар: 9.09091
SMAPE_2й_товар: 0.01000
SMAPE итого: 4.55045
суммарная ошибка прогноза: 40000
```

Видим, что в обоих случаях ошибка прогноза является одинаковой, но при ошибочном прогнозе товара с меньшим объемом продаж метрика SMAPE резко возрастает, а при ошибочном прогнозе товара с большим объемом продаж метрика SMAPE почти идеальна. Опять выходит, что SMAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности.

2.3.4. Симметричная медианная абсолютная процентная ошибка (symmetric median absolute percentage error, SMdAPE)

$$SMdAPE = \text{median} \left(\frac{2 \times |F_t - A_t|}{(|A_t| + |F_t|)} \right),$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t .

Чем меньше значение метрики, тем лучше качество модели.

SMdAPE наследует все вышеописанные недостатки SMAPE. Преимущество SMdAPE в том, что на нее не влияют экстремальные значения и она более устойчива, чем MAPE (Makridakis & Hibon, 2000).

Давайте напишем собственную функцию для вычисления SMdAPE и выведем таблицу с поведением SMdAPE для различных случаев.

```
# пишем функцию для вычисления SMdAPE
def smdape(actual, forecast):
    return np.median(2.0 * np.abs(actual - forecast) /
                      (np.abs(actual) + np.abs(forecast)))

# вычисляем значения SMdAPE
smdape1 = smdape(30, 30)
smdape2 = smdape(20, 30)
smdape3 = smdape(20, 10)
smdape4 = smdape(30, 300)
smdape5 = smdape(10, 20)
smdape6 = smdape(30, 20)
smdape7 = smdape(1, 2)
smdape8 = smdape(2, 1)
smdape9 = smdape(0.1, 1)
smdape10 = smdape(1, 0.1)
smdape11 = smdape(0, 1)
smdape12 = smdape(1, 0)

# создаем датафрейм
data_dict = {'SMdAPE': [smdape1, smdape2, smdape3, smdape4, smdape5,
                        smdape6, smdape7, smdape8, smdape9, smdape10,
                        smdape11, smdape12]}
```

```
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                           'фактическое=20, прогноз=30 (переоценка)',
                           'фактическое=20, прогноз=10 (недооценка)',
                           'фактическое=30, прогноз=300 (большая ошибка)',
                           'фактическое=10, прогноз=20 (переоценка)',
                           'фактическое=30, прогноз=20 (недооценка)',
                           'фактическое=1, прогноз=2 (переоценка)',
                           'фактическое=2, прогноз=1 (недооценка)',
                           'фактическое=0.1, прогноз=1 (переоценка)',
                           'фактическое=1, прогноз=0.1 (недооценка)',
                           'фактическое=0, прогноз=1 (переоценка)',
                           'фактическое=1, прогноз=0 (недооценка)'])
```

df

	SMdAPE
фактическое=30, прогноз=30 (идеально)	0.000
фактическое=20, прогноз=30 (переоценка)	0.400
фактическое=20, прогноз=10 (недооценка)	0.667
фактическое=30, прогноз=300 (большая ошибка)	1.636
фактическое=10, прогноз=20 (переоценка)	0.667
фактическое=30, прогноз=20 (недооценка)	0.400
фактическое=1, прогноз=2 (переоценка)	0.667
фактическое=2, прогноз=1 (недооценка)	0.667
фактическое=0.1, прогноз=1 (переоценка)	1.636
фактическое=1, прогноз=0.1 (недооценка)	1.636
фактическое=0, прогноз=1 (переоценка)	2.000
фактическое=1, прогноз=0 (недооценка)	2.000

Убеждаемся, что поведение SMdAPE аналогично поведению SMAPE.

2.3.5. Взвешенная абсолютная процентная ошибка (weighted absolute percentage error, WAPE)

$$WAPE = \frac{\frac{1}{n} \sum_{t=1}^n |A_t - F_t|}{\frac{1}{n} \sum_{t=1}^n A_t} = \frac{\sum_{t=1}^n |A_t - F_t|}{\sum_{t=1}^n A_t},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Метрика вычисляется следующим образом.

- Для каждого момента времени вычисляем абсолютную ошибку прогноза (прогноз вычитается из факта, и берется модуль разности).
- Суммируем абсолютные ошибки прогноза и получаем сумму абсолютных ошибок прогноза.
- Суммируем фактические значения и получаем сумму фактических значений.
- Сумму абсолютных ошибок прогноза делим на сумму фактических значений.

Чем меньше значение метрики, тем лучше качество модели.

Допустим, у нас есть фактические данные продаж и прогнозы с января по октябрь. Измерим WAPE для каждого месяца.

	Январь	Февраль	Март	Апрель	Май	Июнь	Июль	Август	Сентябрь	Октябрь
Фактические продажи	30	20	20	30	10	30	1	0,1	0	1
Прогноз	30	30	10	300	20	20	2	1	1	0
WAPE	0	0,5	0,5	9	1	0,33	1	9	4.503600e + 15	1

Рис. 63 Поведение WAPE при различных ситуациях

Мы видим, что поведение WAPE аналогично поведению MAPE.

Если фактическое значение временного ряда равно 1 или меньше 1, то в знаменателе окажется очень маленькое число и значение WAPE резко возрастет. Если фактическое значение временного ряда будет равно 0, метрика WAPE будет либо не определена, либо огромна (у нас либо происходит деление на ноль и выдается ошибка, либо ноль заменяется на очень маленькое значение, близкое к 0, однако чем оно меньше, тем больше WAPE). Посмотрите на прогнозы с июля по октябрь. Несмотря на то что прогнозы довольно близки к фактическим значениям, WAPE принимает большие значения.

Теперь посмотрите на прогнозы для мая и июня. WAPE по-разному относится к положительным и отрицательным ошибкам. За переоценку, т. е. отрицательные ошибки ($A_t < F_t$, например $10 < 20$), WAPE штрафует сильнее.

Допустим, фактическое значение равно 10, а наш точечный прогноз составил 20. Какой будет ошибка WAPE в этом случае? Подставим эти значения в формулу, чтобы получить $|10 - 20| / 10 = 1 = 100\%$. А какой была бы ошибка, если бы фактическое значение было равно 30? Очевидно, что $|30 - 20| / 30 = 0,333 = 33,3\%$. Вроде бы ошибка та же, но по объективным причинам она во втором случае составляет 33,3 % от фактического значения, а не 100 %. Действительно, WAPE жестче относится к случаям завышенных прогнозов, чем заниженных (подтвердили, что отрицательные ошибки штрафуются сильнее, чем положительные). Если WAPE используется для сравнения точности прогнозных моделей, он будет систематически выбирать модель, прогнозы которой занижены.

Теперь посмотрите на прогнозы для апреля и августа. В первом случае ошибка прогноза составляет 270, а во втором случае ошибка прогноза составляет 0,9, но в обоих случаях $WAPE = 900,0\%$.

Давайте напишем собственную функцию для вычисления WAPE и убедимся, что для набора с заниженными прогнозами мы получим меньшее значение WAPE.

```
# пишем функцию для вычисления WAPE
def wape(actual, forecast):
    epsilon = np.finfo(np.float64).eps
    return np.sum(np.abs(actual - forecast)) / np.sum(
        np.maximum(np.abs(actual), epsilon))

# вычисляем WAPE для заниженных прогнозов
wape_for_underest = wape(exmpl_underest['Actual'],
                           exempl_underest['Forecast'])

# вычисляем WAPE для завышенных прогнозов
wape_for_overest = wape(exmpl_overest['Actual'],
                           exempl_overest['Forecast'])

# печатаем значения WAPE
print("WAPE для заниженных прогнозов: %.3f" % wape_for_underest)
print("WAPE для завышенных прогнозов: %.3f" % wape_for_overest)

WAPE для заниженных прогнозов: 0.247
WAPE для завышенных прогнозов: 0.305
```

Посмотрим на поведение WAPE для различных случаев.

```
# вычисляем значения WAPE
wape1 = wape(30, 30)
wape2 = wape(20, 30)
wape3 = wape(20, 10)
wape4 = wape(30, 300)
wape5 = wape(10, 20)
wape6 = wape(30, 20)
wape7 = wape(1, 2)
wape8 = wape(2, 1)
wape9 = wape(0.1, 1)
wape10 = wape(1, 0.1)
wape11 = wape(0, 1)
wape12 = wape(1, 0)

# создаем датафрейм
data_dict = {'WAPE': [wape1, wape2, wape3, wape4, wape5,
                      wape6, wape7, wape8, wape9, wape10,
                      wape11, wape12]}
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                          'фактическое=20, прогноз=30 (переоценка)',
                          'фактическое=20, прогноз=10 (недооценка)',
                          'фактическое=30, прогноз=300 (большая ошибка)',
                          'фактическое=10, прогноз=20 (переоценка)',
                          'фактическое=30, прогноз=20 (недооценка)',
                          'фактическое=1, прогноз=2 (переоценка)',
                          'фактическое=2, прогноз=1 (недооценка)',
```

```
'фактическое=0.1, прогноз=1 (переоценка)',  
'фактическое=1, прогноз=0.1 (недооценка)',  
'фактическое=0, прогноз=1 (переоценка)',  
'фактическое=1, прогноз=0 (недооценка)']
```

df

	WAPE
фактическое=30, прогноз=30 (идеально)	0.000
фактическое=20, прогноз=30 (переоценка)	0.500
фактическое=20, прогноз=10 (недооценка)	0.500
фактическое=30, прогноз=300 (большая ошибка)	9.000
фактическое=10, прогноз=20 (переоценка)	1.000
фактическое=30, прогноз=20 (недооценка)	0.333
фактическое=1, прогноз=2 (переоценка)	1.000
фактическое=2, прогноз=1 (недооценка)	0.500
фактическое=0.1, прогноз=1 (переоценка)	9.000
фактическое=1, прогноз=0.1 (недооценка)	0.900
фактическое=0, прогноз=1 (переоценка)	4503599627370496.000
фактическое=1, прогноз=0 (недооценка)	1.000

Напомним, что в случае разницы в объемах продаж (например, у вас могут быть высокие продажи быстро распродаваемого товара и гораздо меньшие объемы продаж медленно распродаваемого товара) метрики MAPE и SMAPE могут дезориентировать аналитика. В этом случае лучше использовать WAPE.

Вернемся к знакомому примеру. У нас есть данные продаж по двум товарам, с помощью какой-то модели получили прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	101 000 000	101 120 000
WAPE	0,2	0,01	0,01019
Ошибка прогноза	20 000	1 000 000	1 020 000

На уровне прогнозирования продаж отдельных товаров WAPE ведет себя так же, как MAPE и SMAPE: большая ошибка прогноза (1 000 000) получает меньшее значение WAPE (0,01).

Теперь переходим к итоговым значениям. Фактически мы продаем на 100 100 000 (сто миллионов сто тысяч). Мы прогнозируем продажи на 101 120 000 (сто один миллион сто двадцать тысяч). Ошибка составляет 1 020 000 (один миллион двадцать тысяч). В итоге WAPE составит 0,01.

```
# вычисляем WAPE для первой модели
print("WAPE_1й_товар: %.5f" % wape(100000, 120000))
print("WAPE_2й_товар: %.5f" % wape(100000000, 101000000))
```

```
actual = np.array([100000, 100000000])
forecast = np.array([120000, 101000000])
print("WAPE итого: %.5f" % wape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

WAPE_1й_товар: 0.20000
WAPE_2й_товар: 0.01000
WAPE итого: 0.01019
суммарная ошибка прогноза: 1020000
```

Здесь прекрасно видно, что итоговое значение WAPE – это не обычное среднее, а взвешенное среднее значений WAPE для каждого товара, и оно будет близко к значению WAPE для товара с большим весом – товара 2.

```
# вес первого продукта
weight_frst_item = 100000 / (100000000 + 100000)
weight_frst_item

0.000999000999000999

# вес второго продукта
weight_scnd_item = 100000000 / (100000000 + 100000)
weight_scnd_item

0.999000999000999

# итоговое значение WAPE
WAPE_total = 0.2 * weight_frst_item + 0.01 * weight_scnd_item
print("WAPE итого: %.5f" % WAPE_total)

WAPE итого: 0.01019
```

С помощью еще одной модели получили новые прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	110 000	105 000 000	105 110 000
WAPE	0,1	0,05	0,05
Ошибка прогноза	10 000	5 000 000	5 010 000

На уровне прогнозирования продаж отдельных товаров WAPE ведет себя так же, как MAPE и SMAPE: большая ошибка прогноза (5 000 000) получает меньшее значение WAPE (0,05).

Теперь переходим к итоговым значениям. Как и прежде, по факту мы проаем на 100 100 000 (сто миллионов сто тысяч). Мы прогнозируем продажи на 105 110 000 (сто пять миллионов сто десять тысяч). Ошибка составляет 5 010 000 (пять миллионов десять тысяч). В итоге WAPE составит 0,05.

```
# вычисляем WAPE для второй модели
print("WAPE_1й_товар: %.3f" % wape(100000, 110000))
print("WAPE_2й_товар: %.3f" % wape(100000000, 105000000))
actual = np.array([100000, 100000000])
forecast = np.array([110000, 105000000])
print("WAPE итого: %.3f" % wape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

WAPE_1й_товар: 0.100
WAPE_2й_товар: 0.050
WAPE итого: 0.050
суммарная ошибка прогноза: 5010000
```

Таким образом, при большей ошибке прогноза мы получили большее итоговое значение WAPE, что вполне адекватно.

Еще приведем пример. Допустим, у нас есть данные продаж по двум товарам, с помощью какой-то модели получили прогнозы продаж по этим товарам.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	100 020 000	100 140 000
WAPE	0,2	0,0002	0,0004
Ошибка прогноза	20 000	20 000	40 000

```
# еще один пример
print("WAPE_1й_товар: %.5f" % wape(100000, 120000))
print("WAPE_2й_товар: %.5f" % wape(100000000, 100020000))
actual = np.array([100000, 100000000])
forecast = np.array([120000, 100020000])
print("WAPE итого: %.5f" % wape(actual, forecast))
print("суммарная ошибка прогноза:", sum(forecast) - sum(actual))

WAPE_1й_товар: 0.20000
WAPE_2й_товар: 0.00020
WAPE итого: 0.00040
суммарная ошибка прогноза: 40000
```

Видим, что в обоих случаях ошибка прогноза является одинаковой, но при ошибочном прогнозе товара с меньшим объемом продаж метрика WAPE резко возрастает, а при ошибочном прогнозе товара с большим объемом продаж метрика WAPE почти идеальна. Вновь видим, что WAPE мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности. Итоговое значение WAPE будет близко значению WAPE для товара с большим весом (товара 2), т. е. будет почти идеальным.

2.3.6. Средневзвешенная абсолютная процентная ошибка (weighted mean absolute percentage error, WMAPE)

$$WMAPE = \frac{\sum_{t=1}^n (w_t |A_t - F_t|)}{\sum_{t=1}^n (w_t |A_t|)},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели.

Метрика часто используется для анализа временных рядов продаж и используется, когда нужно подчеркнуть важность какого-то отдельного момента времени t для прогнозирования.

Допустим, у нас есть фактические продажи за три дня и прогнозы по ним.

	Понедельник	Вторник	Среда	ИТОГО
Фактические продажи	50	1	50	101
Прогноз	55	2	50	107
Ошибка прогноза	5	1	0	6

Вспоминаем, что в случае разницы в объемах продаж метрика MAPE может насdezориентировать. Она мягче относится к ошибкам прогноза, которые получаются для чисел большей разрядности (соответствующих товарам с большими объемами продаж). WAPE поможет устраниить данный недостаток. WMAPE позволит выделить наиболее важный или наиболее приоритетный день продаж. Например, мы считаем понедельник наиболее важным днем продаж и присваиваем ему вес 8, а остальные дни имеют вес 1.

	Понедельник	Вторник	Среда	ИТОГО
Фактические продажи	50	1	50	101
Прогноз	55	2	50	107
Ошибка прогноза	5	1	0	6
MAPE	0,1	1	0	0,367
WAPE	0,1	1	0	0,059
веса	8	1	1	
WMAPE	0,1	1	0	0,091

Ниже приведен расчет итогового значения WMAPE:

$$WMAPE = \frac{\sum_{t=1}^n (w_t |A_t - F_t|)}{\sum_{t=1}^n (w_t |A_t|)} = \frac{8*5 + 1*1 + 1*0}{8*50 + 1*1 + 1*50} = \frac{41}{451} = 0,091.$$

Давайте напишем собственную функцию для вычисления WMAPE и воспроизведем наши результаты в Python.

```
# пишем функцию для вычисления WMAPE
def wmape(a, f, w):
    """
    Вычисляет WMAPE.

    Параметры
    -----
    a: одномерный массив
        Массив фактических значений.
    f: одномерный массив
        Массив спрогнозированных значений.
    w: одномерный массив
        Массив весов.
    """

    if not isinstance((a, f, w), (list, np.ndarray)):
        a = pd.Series(a)
        f = pd.Series(f)
        w = pd.Series(w)

    num = 0.0
    for i in range(len(a)):
        num += w[i] * np.abs(a[i] - f[i])

    den = 0.0
    for i in range(len(a)):
        den += w[i] * np.abs(a[i])
    return num / den

# записываем массив фактических значений
# и массив спрогнозированных значений
actual = np.array([50, 1, 50])
forecast = np.array([55, 2, 50])

# записываем значения MAPE
mape_mon = sklearn_mape(50, 55)
mape_tue = sklearn_mape(1, 2)
mape_wed = sklearn_mape(50, 50)
mape_total = sklearn_mape(actual, forecast)

# записываем значения WAPE
wape_mon = wape(50, 55)
wape_tue = wape(1, 2)
wape_wed = wape(50, 50)
wape_total = wape(actual, forecast)

# записываем значения WMAPE
weights = [8, 1, 1]
wmape_mon = wmape(50, 55, weights[0])
```

```
wmape_tue = wmape(1, 2, weights[1])
wmape_wed = wmape(50, 50, weights[2])
wmape_total = wmape(actual, forecast, weights)

# добавляем в концы массивов итоговые значения
actual = np.insert(actual, 3, sum(actual))
forecast = np.insert(forecast, 3, sum(forecast))

# создаем датафрейм
data_dict = {'Фактические значения': actual,
             'Прогнозы': forecast,
             'Ошибка прогноза': forecast - actual,
             'MAPE': [mape_mon, mape_tue, mape_wed, mape_total],
             'WAPE': [wape_mon, wape_tue, wape_wed, wape_total],
             'веса': [weights[0], weights[1], weights[2], ''],
             'WMAPE': [wmape_mon, wmape_tue, wmape_wed, wmape_total]}
df = pd.DataFrame(data_dict, index=['понедельник',
                                     'вторник',
                                     'среда',
                                     'ИТОГО'])
df.T
```

	понедельник	вторник	среда	ИТОГО
Фактические значения	50	1	50	101
Прогнозы	55	2	50	107
Ошибка прогноза	5	1	0	6
MAPE	0.100	1.000	0.000	0.367
WAPE	0.100	1.000	0.000	0.059
веса	8	1	1	
WMAPE	0.100	1.000	0.000	0.091

Теперь смоделируем ситуацию, когда мы идеально прогнозируем продажи в понедельник (самый важный день) и плохо предсказываем по остальным дням.

	Понедельник	Вторник	Среда	ИТОГО
Фактические продажи	50	1	50	101
Прогноз	50	12	80	142
Ошибка прогноза	0	11	30	41
веса	8	1	1	
WMAPE	0	11,0	0,6	0,091

Видим, что хотя мы довольно плохо предсказываем во вторник и среду, WMAPE не возросла, потому что основное внимание уделено понедельнику, а в понедельник мы прогнозируем идеально.

```

# записываем список фактических значений
# и список спрогнозированных значений
actual = np.array([50, 1, 50])
forecast = np.array([50, 12, 80])
# записываем значения WMAPE
weights = [8, 1, 1]
wmape_mon = wmape(50, 50, weights[0])
wmape_tue = wmape(1, 12, weights[1])
wmape_wed = wmape(50, 80, weights[2])
wmape_total = wmape(actual, forecast, weights)

# добавляем в концы массивов итоговые значения
actual = np.insert(actual, 3, sum(actual))
forecast = np.insert(forecast, 3, sum(forecast))

# создаем датафрейм
data_dict = {'Фактические значения': actual,
             'Прогнозы': forecast,
             'Ошибка прогноза': forecast - actual,
             'веса': [weights[0], weights[1], weights[2], ''],
             'WMAPE': [wmape_mon, wmape_tue, wmape_wed, wmape_total]}
df = pd.DataFrame(data_dict, index=['понедельник',
                                     'вторник',
                                     'среда',
                                     'ИТОГО'])
df.T

```

	понедельник	вторник	среда	ИТОГО
Фактические значения	50	1	50	101
Прогнозы	50	12	80	142
Ошибка прогноза	0	11	30	41
веса	8	1	1	
WMAPE	0.000	11.000	0.600	0.091

2.3.7. Корень из среднеквадратичной процентной ошибки (root mean square percentage error, RMSPE)

$$RMSPE = \sqrt{\frac{1}{n} \sum_{t=1}^n \left(\frac{A_t - F_t}{A_t} \right)^2},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t ;

n – общее количество наблюдений, а для временных рядов – количество моментов времени t (определяется горизонтом прогнозирования).

Чем меньше значение метрики, тем лучше качество модели.
Поведение RMSPE во многом аналогично поведению MAPE.

```
# пишем функцию для вычисления RMSPE
def rmspe(actual, forecast):
    # задаем эпсилон - очень маленькое значение
    # (2.220446049250313e-16)
    epsilon = np.finfo(np.float64).eps
    return np.sqrt(
        np.mean(np.square((actual - forecast) / (actual + epsilon)))
    )

# вычисляем значения RMSPE
rmspe1 = rmspe(30, 30)
rmspe2 = rmspe(20, 30)
rmspe3 = rmspe(20, 10)
rmspe4 = rmspe(30, 300)
rmspe5 = rmspe(10, 20)
rmspe6 = rmspe(30, 20)
rmspe7 = rmspe(1, 2)
rmspe8 = rmspe(2, 1)
rmspe9 = rmspe(0.1, 1)
rmspe10 = rmspe(1, 0.1)
rmspe11 = rmspe(0, 1)
rmspe12 = rmspe(1, 0)

# создаем датафрейм
data_dict = {'RMSPE': [rmspe1, rmspe2, rmspe3, rmspe4, rmspe5,
                       rmspe6, rmspe7, rmspe8, rmspe9, rmspe10,
                       rmspe11, rmspe12]}
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                           'фактическое=20, прогноз=30 (переоценка)',
                           'фактическое=20, прогноз=10 (недооценка)',
                           'фактическое=30, прогноз=300 (большая ошибка)',
                           'фактическое=10, прогноз=20 (переоценка)',
                           'фактическое=30, прогноз=20 (недооценка)',
                           'фактическое=1, прогноз=2 (переоценка)',
                           'фактическое=2, прогноз=1 (недооценка)',
                           'фактическое=0.1, прогноз=1 (переоценка)',
                           'фактическое=1, прогноз=0.1 (недооценка)',
                           'фактическое=0, прогноз=1 (переоценка)',
                           'фактическое=1, прогноз=0 (недооценка)'])

df
```

	RMSPE
фактическое=30, прогноз=30 (идеально)	0.000
фактическое=20, прогноз=30 (переоценка)	0.500
фактическое=20, прогноз=10 (недооценка)	0.500
фактическое=30, прогноз=300 (большая ошибка)	9.000
фактическое=10, прогноз=20 (переоценка)	1.000
фактическое=30, прогноз=20 (недооценка)	0.333
фактическое=1, прогноз=2 (переоценка)	1.000
фактическое=2, прогноз=1 (недооценка)	0.500
фактическое=0.1, прогноз=1 (переоценка)	9.000
фактическое=1, прогноз=0.1 (недооценка)	0.900
фактическое=0, прогноз=1 (переоценка)	4503599627370496.000
фактическое=1, прогноз=0 (недооценка)	1.000

2.3.8. Корень из медианной квадратичной процентной ошибки (root median square percentage error, RMdSPE)

$$RMdSPE = \sqrt{\text{median}\left(\left(\frac{A_t - F_t}{A_t}\right)^2\right)},$$

где

A_t – фактическое значение зависимой переменной в t -м наблюдении или в момент времени t ;

F_t – спрогнозированное значение зависимой переменной в t -м наблюдении или в момент времени t .

Чем меньше значение метрики, тем лучше качество модели.

Поведение RMdSPE во многом аналогично поведению MAPE.

```
# пишем функцию для вычисления RMdSPE
def rmdspe(actual, forecast):
    # задаем эпсилон - очень маленькое значение
    # (2.220446049250313e - 16)
    epsilon = np.finfo(np.float64).eps
    return np.sqrt(
        np.median(np.square((actual - forecast) / (actual + epsilon)))
    )

# вычисляем значения RMdSPE
rmdspe1 = rmdspe(30, 30)
rmdspe2 = rmdspe(20, 30)
rmdspe3 = rmdspe(20, 10)
rmdspe4 = rmdspe(30, 300)
```

```
rmdspe5 = rmdspe(10, 20)
rmdspe6 = rmdspe(30, 20)
rmdspe7 = rmdspe(1, 2)
rmdspe8 = rmdspe(2, 1)
rmdspe9 = rmdspe(0.1, 1)
rmdspe10 = rmdspe(1, 0.1)
rmdspe11 = rmdspe(0, 1)
rmdspe12 = rmdspe(1, 0)

# создаем датафрейм
data_dict = {'RMdSPE': [rmdspe1, rmdspe2, rmdspe3, rmdspe4, rmdspe5,
                        rmdspe6, rmdspe7, rmdspe8, rmdspe9, rmdspe10,
                        rmdspe11, rmdspe12]}
df = pd.DataFrame(data_dict,
                   index=['фактическое=30, прогноз=30 (идеально)',
                          'фактическое=20, прогноз=30 (переоценка)',
                          'фактическое=20, прогноз=10 (недооценка)',
                          'фактическое=30, прогноз=300 (большая ошибка)',
                          'фактическое=10, прогноз=20 (переоценка)',
                          'фактическое=30, прогноз=20 (недооценка)',
                          'фактическое=1, прогноз=2 (переоценка)',
                          'фактическое=2, прогноз=1 (недооценка)',
                          'фактическое=0.1, прогноз=1 (переоценка)',
                          'фактическое=1, прогноз=0.1 (недооценка)',
                          'фактическое=0, прогноз=1 (переоценка)',
                          'фактическое=1, прогноз=0 (недооценка)'])
df
```

	RMdSPE
фактическое=30, прогноз=30 (идеально)	0.000
фактическое=20, прогноз=30 (переоценка)	0.500
фактическое=20, прогноз=10 (недооценка)	0.500
фактическое=30, прогноз=300 (большая ошибка)	9.000
фактическое=10, прогноз=20 (переоценка)	1.000
фактическое=30, прогноз=20 (недооценка)	0.333
фактическое=1, прогноз=2 (переоценка)	1.000
фактическое=2, прогноз=1 (недооценка)	0.500
фактическое=0.1, прогноз=1 (переоценка)	9.000
фактическое=1, прогноз=0.1 (недооценка)	0.900
фактическое=0, прогноз=1 (переоценка)	4503599627370496.000
фактическое=1, прогноз=0 (недооценка)	1.000

2.3.9. Сравнение MAPE, MdAPE, SMAPE, SMdAPE, WAPE, WMAPE, RMSPE, RMdSPE

Главным недостатком метрик на основе процентных ошибок является тот факт, что их значения становятся огромными или не определены, если $Y_t = 0$, и имеют чрезвычайно асимметричное распределение, когда значение Y_t близко к нулю. Если данные включают небольшие значения (что является обычным явлением для данных, описывающих прерывистый спрос, см. Gardner, 1990), эти метрики невозможно использовать, поскольку нулевые значения или значения, близкие к нулю, будут встречаться часто. В соревновании M3 чрезмерно больших (или бесконечных) значений MAPE удалось избежать за счет включения только положительных данных (Makridakis and Hibon, 2000). Однако это искусственное решение, которое невозможно применить на практике.

Еще одним недостатком метрик, основанных на процентных ошибках, является то, что они предполагают естественный ноль. Например, они не имеют смысла в измерении ошибки прогноза температуры по шкале Фаренгейта или Цельсия.

У MAPE и MdAPE также есть недостаток, заключающийся в том, что они жестче относятся к случаям завышенных прогнозов, чем заниженных (отрицательные ошибки штрафуются сильнее, чем положительные). Если MAPE и MdAPE используются для сравнения точности прогнозных моделей, они будут систематически выбирать метод, прогнозы которого слишком занижены. Этот факт привел к использованию так называемых «симметричных» мер (Makridakis, 1993) – SMAPE и SMdAPE.

Проблемы, возникающие из-за малых значений Y_t , могут быть менее серьезными для SMAPE и SMdAPE. Однако как бы там ни было, если значение Y_t близко к нулю, то метрика все равно будет возрастать. Кроме того, эти метрики не так «симметричны», как предполагает их название.

Некоторые авторы (например, Swanson et al., 2000) отметили, что показатели, основанные на процентных ошибках, часто сильно смещены, и поэтому преобразования (например, логарифм) могут сделать их более стабильными. См. Coleman and Swanson (2004) для ознакомления с деталями.

Метрики MAPE, SMAPE, MdAPE, SMdAPE, RMSPE, RMdSPE в случае разницы в объемах продаж могут дезориентировать аналитика. Они мягче относятся к ошибкам прогноза, которые получаются для чисел большей разрядности. При большей ошибке прогноза мы можем получить меньшее значение MAPE, SMAPE, MdAPE, SMdAPE, RMSPE, RMdSPE. Чтобы избежать этого, нужно использовать WAPE. Она взвешивает ошибку с учетом объемов продаж. Если необходимо подчеркнуть важность какого-то отдельного момента времени t для прогнозирования, нужно воспользоваться WMAPE.

Метрики RMSPE и RMdSPE демонстрируют поведение, аналогичное поведению MAPE.

Давайте посмотрим поведение метрик на наборах с различными распределениями ошибок.

вычисляем метрики

MAPE = sklearn_mape(exmpl['Actual'], exmpl['Forecast'])

MAPE2 = sklearn_mape(exmpl2['Actual'], exmpl2['Forecast'])

MAPE3 = sklearn_mape(exmpl3['Actual'], exmpl3['Forecast'])

MAPE4 = sklearn_mape(exmpl4['Actual'], exmpl4['Forecast'])

MAPE5 = sklearn_mape(exmpl5['Actual'], exmpl5['Forecast'])

MAPE6 = sklearn_mape(exmpl6['Actual'], exmpl6['Forecast'])

MAPE7 = sklearn_mape(exmpl7['Actual'], exmpl7['Forecast'])

MDAPE = mdape(exmpl['Actual'], exmpl['Forecast'])

MDAPE2 = mdape(exmpl2['Actual'], exmpl2['Forecast'])

MDAPE3 = mdape(exmpl3['Actual'], exmpl3['Forecast'])

MDAPE4 = mdape(exmpl4['Actual'], exmpl4['Forecast'])

MDAPE5 = mdape(exmpl5['Actual'], exmpl5['Forecast'])

MDAPE6 = mdape(exmpl6['Actual'], exmpl6['Forecast'])

MDAPE7 = mdape(exmpl7['Actual'], exmpl7['Forecast'])

SMAPE = standard_smape(exmpl['Actual'], exmpl['Forecast'])

SMAPE2 = standard_smape(exmpl2['Actual'], exmpl2['Forecast'])

SMAPE3 = standard_smape(exmpl3['Actual'], exmpl3['Forecast'])

SMAPE4 = standard_smape(exmpl4['Actual'], exmpl4['Forecast'])

SMAPE5 = standard_smape(exmpl5['Actual'], exmpl5['Forecast'])

SMAPE6 = standard_smape(exmpl6['Actual'], exmpl6['Forecast'])

SMAPE7 = standard_smape(exmpl7['Actual'], exmpl7['Forecast'])

SMDAPE = smdape(exmpl['Actual'], exmpl['Forecast'])

SMDAPE2 = smdape(exmpl2['Actual'], exmpl2['Forecast'])

SMDAPE3 = smdape(exmpl3['Actual'], exmpl3['Forecast'])

SMDAPE4 = smdape(exmpl4['Actual'], exmpl4['Forecast'])

SMDAPE5 = smdape(exmpl5['Actual'], exmpl5['Forecast'])

SMDAPE6 = smdape(exmpl6['Actual'], exmpl6['Forecast'])

SMDAPE7 = smdape(exmpl7['Actual'], exmpl7['Forecast'])

WAPE = wape(exmpl['Actual'], exmpl['Forecast'])

WAPE2 = wape(exmpl2['Actual'], exmpl2['Forecast'])

WAPE3 = wape(exmpl3['Actual'], exmpl3['Forecast'])

WAPE4 = wape(exmpl4['Actual'], exmpl4['Forecast'])

WAPE5 = wape(exmpl5['Actual'], exmpl5['Forecast'])

WAPE6 = wape(exmpl6['Actual'], exmpl6['Forecast'])

WAPE7 = wape(exmpl7['Actual'], exmpl7['Forecast'])

RMSPE = rmspe(exmpl['Actual'], exmpl['Forecast'])

RMSPE2 = rmspe(exmpl2['Actual'], exmpl2['Forecast'])

RMSPE3 = rmspe(exmpl3['Actual'], exmpl3['Forecast'])

RMSPE4 = rmspe(exmpl4['Actual'], exmpl4['Forecast'])

RMSPE5 = rmspe(exmpl5['Actual'], exmpl5['Forecast'])

RMSPE6 = rmspe(exmpl6['Actual'], exmpl6['Forecast'])

RMSPE7 = rmspe(exmpl7['Actual'], exmpl7['Forecast'])

RMDSPE = rmdspe(exmpl['Actual'], exmpl['Forecast'])

RMDSPE2 = rmdspe(exmpl2['Actual'], exmpl2['Forecast'])

RMDSPE3 = rmdspe(exmpl3['Actual'], exmpl3['Forecast'])

RMDSPE4 = rmdspe(exmpl4['Actual'], exmpl4['Forecast'])

RMDSPE5 = rmdspe(exmpl5['Actual'], exmpl5['Forecast'])

RMDSPE6 = rmdspe(exmpl6['Actual'], exmpl6['Forecast'])

RMDSPE7 = rmdspe(exmpl7['Actual'], exmpl7['Forecast'])

```
# создаем датафрейм, строки - модели, столбцы - метрики
data_dict = {'MAPE': [MAPE, MAPE2, MAPE3, MAPE4,
                      MAPE5, MAPE6, MAPE7],
             'MdAPE': [MDAPE, MDAPE2, MDAPE3, MDAPE4,
                        MDAPE5, MDAPE6, MDAPE7],
             'SMAPE': [SMAPE, SMAPE2, SMAPE3, SMAPE4,
                        SMAPE5, SMAPE6, SMAPE7],
             'SMdAPE': [SMdAPE, SMdAPE2, SMdAPE3, SMdAPE4,
                        SMdAPE5, SMdAPE6, SMdAPE7],
             'WAPE': [WAPE, WAPE2, WAPE3, WAPE4,
                      WAPE5, WAPE6, WAPE7],
             'RMSPE': [RMSPE, RMSPE2, RMSPE3, RMSPE4,
                        RMSPE5, RMSPE6, RMSPE7],
             'RMdSPE': [RMdSPE, RMdSPE2, RMdSPE3, RMdSPE4,
                        RMdSPE5, RMdSPE6, RMdSPE7]}

df = pd.DataFrame(
    data_dict,
    index=['равномерно распределенные положительные ошибки',
           'небольшое варьирование размеров положительных ошибок',
           'одна большая положительная ошибка',
           'одна большая положительная и одна большая отрицательная',
           'равномерно распределенные отрицательные ошибки',
           'небольшое варьирование размеров отрицательных ошибок',
           'одна большая отрицательная ошибка'])

df
```

	MAPE	MdAPE	SMAPE	SMdAPE	WAPE	RMSPE	RMdSPE
равномерно распределенные положительные ошибки	0.321	0.268	20.199	0.310	0.267	0.355	0.268
небольшое варьирование размеров положительных ошибок	0.284	0.261	16.859	0.301	0.267	0.297	0.262
одна большая положительная ошибка	0.067	0.000	5.000	0.000	0.267	0.211	0.000
одна большая положительная и одна большая отрицательная	0.147	0.000	7.857	0.000	0.430	0.329	0.000
равномерно распределенные отрицательные ошибки	0.586	0.367	20.199	0.310	0.364	0.787	0.368
небольшое варьирование размеров отрицательных ошибок	0.422	0.354	16.859	0.301	0.364	0.473	0.355
одна большая отрицательная ошибка	0.200	0.000	5.000	0.000	0.364	0.632	0.000

Видим, что MAPE и WAPE штрафуют за большую отрицательную ошибку сильнее, чем за большую положительную ошибку. SMAPE одинаково штрафует за большую отрицательную и большую положительную ошибки. Процентные метрики на основе медианы (MdAPE, SMdAPE, RMdSPE) в присутствии большой отрицательной или большой положительной ошибки становятся неадекватными, принимая идеальные значения.

2.4. МЕТРИКИ КАЧЕСТВА НА ОСНОВЕ ОТНОСИТЕЛЬНЫХ ОШИБОК (MRAE, MdRAE, GMRAE)

Альтернативный способ масштабирования – разделить каждую ошибку на ошибку, полученную с помощью другого стандартного метода прогнозирования. Пусть у нас есть относительная ошибка $r_t = e_t/e_t^*$, где e_t^* – ошибка прогноза, полученная с помощью какого-то базового метода. Обычно базовый метод – это случайное блуждание, где F_t равно последнему наблюдению. Речь идет о

таких метриках, как MRAE, MdRAE, GMRAE. Они используются преимущественно в прогнозировании временных рядов. Давайте поговорим о каждой метрике по порядку.

2.4.1. Средняя относительная абсолютная ошибка (mean relative absolute error, MRAE)

$$MRAE = \text{mean} \left(\left| \frac{e_t}{e_t^*} \right| \right),$$

где

e_t – ошибка прогноза для момента времени t ;

e_t^* – ошибка прогноза для момента времени t , полученная с помощью какого-то базового метода (модели-бенчмарка).

Чем меньше значение метрики, тем лучше качество модели.

MRAE показывает, во сколько раз наша модель оказалась хуже (или лучше), чем выбранная для сравнения (модель-бенчмарк). Здесь возможны три случая: если $MRAE = 1$, то наша модель и модель-бенчмарк одинаково хороши; если $MRAE < 1$, наша модель работает лучше, чем модель-бенчмарк, и, наконец, если $MRAE > 1$, наша модель работает хуже, чем модель-бенчмарк.

Давайте напишем собственную функцию для вычисления MRAE.

```
# пишем функцию для вычисления обычной ошибки
def _error(actual, predicted):
    return actual - predicted

# пишем функцию для вычисления наивного прогноза
def _naive_forecasting(actual, seasonality=1):
    return actual[:-seasonality]

# пишем функцию для вычисления относительной ошибки
def _relative_error(actual, predicted, benchmark=None):
    # задаем эпсилон - очень маленькое
    # значение (2.220446049250313e-16)
    epsilon = np.finfo(np.float64).eps

    if benchmark is None or isinstance(benchmark, int):
        if not isinstance(benchmark, int):
            seasonality = 1
        else:
            seasonality = benchmark
        return _error(actual[seasonality:], predicted[seasonality:]) / (
            _error(actual[seasonality:], _naive_forecasting(actual, seasonality)) + epsilon)
    return _error(actual, predicted) / (_error(actual, benchmark) + epsilon)

# пишем функцию для вычисления MRAE
def mrae(actual, predicted, benchmark=None):
    """
    Вычисляет MRAE.
    """
```

Параметры

```

actual: одномерный массив
    Массив фактических значений.
predicted: одномерный массив
    Массив спрогнозированных значений текущей модели.
benchmark: одномерный массив
    Массив спрогнозированных значений модели-бенчмарка.
"""

return np.round(np.mean(np.abs(
    _relative_error(actual, predicted, benchmark))), 3)

```

Рассмотрим пример с разными объемами продаж и вычислим MRAE.

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	120 000	101 000 000	101 120 000
MRAE			0,01
Ошибка прогноза	20 000	1 000 000	1 020 000

	Товар 1	Товар 2	Итого
Фактические продажи	100 000	100 000 000	100 100 000
Прогноз	110 000	105 000 000	105 110 000
MRAE			0,05
Ошибка прогноза	10 000	5 000 000	5 010 000

```

# создаем массив фактических значений и два массива прогнозов
a = np.array([100000, 100000000])
f = np.array([120000, 101000000])
f2 = np.array([110000, 105000000])

```

```

# печатаем значения метрик
print(mrae(a, f))
print(mrae(a, f2))

```

```

0.01
0.05

```

При большей ошибке прогноза мы получили большее значение MRAE, что вполне адекватно. При этом небольшое значение метрики говорит о том, что мы предсказываем существенно лучше модели-бенчмарка (модели наивного прогноза).

Допустим, мы прогнозируем количество туристов в тысячах, которое посетит страну в ближайшие семь месяцев. У нас есть прогнозы текущей модели и прогнозы модели-бенчмарка.

Момент времени t	Фактическое значение Y_t	Прогноз F_t	Ошибка прогноза $e_t = Y_t - F_t$	Прогноз бенчмарк-модели BF_t	Ошибка бенчмарк-прогноза $e_t^* = Y_t - BF_t$	Абсолютная относительная ошибка $r_t = \left \frac{e_t}{e_t^*} \right $
1	64	90	-26	68	-4	6,5
2	76	101	-25	75	1	25
3	35	44	-9	33	2	4,5
4	33	32	1	54	-21	0,05
5	29	31	-2	37	-8	0,25
6	35	44	-9	38	-3	3
7	47	102	-55	50	-3	18,33
Всего			-125			57,63

Давайте вычислим MRAE:

$$MRAE = \frac{6,5 + 25 + 4,5 + 0,05 + 0,25 + 3 + 18,33}{7} = \frac{57,63}{7} = 8,233.$$

```
# создаем массив фактических значений
actl = np.array([64, 76, 35, 33, 29, 35, 47])
# создаем массив прогнозов текущей модели
fcst = np.array([90, 101, 44, 32, 31, 44, 102])
# создаем массив прогнозов бенчмарк-модели
benchmark_fcst = np.array([68, 75, 33, 54, 37, 38, 50])

# вычисляем MRAE для задачи про туристов
mrae(actl, fcst, benchmark_fcst)
```

8.233

Наша модель работает хуже модели-бенчмарка.

Посмотрим поведение MRAE в наборах с различным распределением ошибок. В качестве бенчмарка возьмем модель, всегда предсказывающую среднее.

```
# вычисляем метрики
MRAE = mrae(exmpl['Actual'], exempl['Forecast'],
             np.array(exmpl['Actual'].mean() * np.ones(10)))
MRAE2 = mrae(exmpl2['Actual'], exempl2['Forecast'],
              np.array(exmpl2['Actual'].mean() * np.ones(10)))
MRAE3 = mrae(exmpl3['Actual'], exempl3['Forecast'],
              np.array(exmpl3['Actual'].mean() * np.ones(10)))
MRAE4 = mrae(exmpl4['Actual'], exempl4['Forecast'],
              np.array(exmpl4['Actual'].mean() * np.ones(10)))
MRAE5 = mrae(exmpl5['Actual'], exempl5['Forecast'],
              np.array(exmpl5['Actual'].mean() * np.ones(10)))
```

```

MRAE6 = mgae(exmpl6['Actual'], exempl6['Forecast'],
              np.array(exmpl6['Actual'].mean() * np.ones(10)))
MRAE7 = mgae(exmpl7['Actual'], exempl7['Forecast'],
              np.array(exmpl7['Actual'].mean() * np.ones(10)))

# создаем датафрейм, строки - модели, столбцы - метрики
data_dict = {'MRAE': [MRAE, MRAE2, MRAE3, MRAE4,
                      MRAE5, MRAE6, MRAE7]}

df = pd.DataFrame(
    data_dict,
    index=['равномерно распределенные положительные ошибки',
           'небольшое варьирование размеров положительных ошибок',
           'одна большая положительная ошибка',
           'одна большая положительная и одна большая отрицательная',
           'равномерно распределенные отрицательные ошибки',
           'небольшое варьирование размеров отрицательных ошибок',
           'одна большая отрицательная ошибка'])

```

df

MRAE	
равномерно распределенные положительные ошибки	1.430
небольшое варьирование размеров положительных ошибок	0.703
одна большая положительная ошибка	0.089
одна большая положительная и одна большая отрицательная	0.224
равномерно распределенные отрицательные ошибки	1.430
небольшое варьирование размеров отрицательных ошибок	1.430
одна большая отрицательная ошибка	0.444

Видим, что метрика MRAE сильнее штрафует за отрицательные ошибки.

2.4.2. Медианная относительная абсолютная ошибка (median relative absolute error, MdRAE)

$$MdRAE = \text{median} \left(\left| \frac{e_t}{e_t^*} \right| \right),$$

где

e_t – ошибка прогноза для момента времени t ;

e_t^* – ошибка прогноза для момента времени t , полученная с помощью какого-то базового метода (модели-бенчмарка).

Чем меньше значение метрики, тем лучше качество модели.

Как и MRAE, MdRAE показывает, во сколько раз наша модель оказалась хуже (или лучше), чем выбранная для сравнения (модель-бенчмарк). Здесь возможны три случая: если $MdRAE = 1$, то наша модель и модель-бенчмарк одинаково хороши; если $MdRAE < 1$, наша модель работает лучше, чем модель-бенчмарк, и, наконец, если $MdRAE > 1$, наша модель работает хуже, чем модель-бенчмарк.

Давайте напишем собственную функцию для вычисления MdRAE.

```
# пишем функцию для вычисления MdRAE
def mdrae(actual, predicted, benchmark=None):
    """
    Вычисляет MdRAE.

    Параметры
    -----
    actual: одномерный массив
        Массив фактических значений.
    predicted: одномерный массив
        Массив спрогнозированных значений текущей модели.
    benchmark: одномерный массив
        Массив спрогнозированных значений модели-бенчмарка.
    """

    return np.round(np.median(np.abs(
        _relative_error(actual, predicted, benchmark))), 3)
```

Теперь вычислим MdRAE для нашей задачи прогнозирования количества туристов.

Момент времени t	Фактическое значение Y_t	Прогноз F_t	Ошибка прогноза $e_t = Y_t - F_t$	Прогноз бенчмарк-модели BF_t	Ошибка бенчмарк-прогноза $\hat{e}_t = Y_t - BF_t$	Абсолютная относительная ошибка $r_t = \left \frac{e_t}{\hat{e}_t} \right $
1	64	90	-26	68	-4	6,5
2	76	101	-25	75	1	25
3	35	44	-9	33	2	4,5
4	33	32	1	54	-21	0,05
5	29	31	-2	37	-8	0,25
6	35	44	-9	38	-3	3
7	47	102	-55	50	-3	18,33
Всего				-125		57,63

Давайте вычислим MdRAE, нам нужно просто найти медиану абсолютных относительных ошибок.

0,05	0,25	3	4,5	6,5	18,33	25
------	------	---	-----	-----	-------	----

```
# вычисляем MdRAE для задачи про туристов
mdrae(actl, fcst, benchmark_fcst)
```

2.4.3. Средняя геометрическая относительная абсолютная ошибка (geometric mean relative absolute error, GMRAE)

$$GMRAE = \text{gmean}\left(\left|\frac{e_t}{e_t^*}\right|\right) = \sqrt[n]{\left|\frac{e_1}{e_1^*}\right| \times \dots \times \left|\frac{e_n}{e_n^*}\right|} = \sqrt[n]{\prod_{t=1}^n \left|\frac{e_t}{e_t^*}\right|} = \exp\left(\frac{1}{n} \sum_{t=1}^n \ln\left(\left|\frac{e_t}{e_t^*}\right|\right)\right),$$

где

e_t – ошибка прогноза для момента времени t ;

e_t^* – ошибка прогноза для момента времени t , полученная с помощью какого-то базового метода (модели-бенчмарка).

Чем меньше значение метрики, тем лучше качество модели.

GMRAE показывает, во сколько раз наша модель оказалась хуже (или лучше), чем выбранная для сравнения (модель-бенчмарк). Здесь возможны три случая: если $GMRAE = 1$, то наша модель и модель-бенчмарк одинаково хороши; если $GMRAE < 1$, наша модель работает лучше, чем модель-бенчмарк, и, наконец, если $GMRAE > 1$, наша модель работает хуже, чем модель-бенчмарк.

Давайте напишем собственную функцию для вычисления GMRAE.

```
# пишем функцию для вычисления геометрического среднего
def _geometric_mean(a, axis=0):
    if not isinstance(a, np.ndarray):
        log_a = np.log(np.array(a, dtype=None))
    else:
        log_a = np.log(a)
    return np.exp(log_a.mean(axis=axis))

# пишем функцию для вычисления GMRAE
def gmrae(actual, predicted, benchmark=None):
    """
    Вычисляет GMRAE.

    Параметры
    -----
    actual: одномерный массив
        Массив фактических значений.
    predicted: одномерный массив
        Массив спрогнозированных значений текущей модели.
    benchmark: одномерный массив
        Массив спрогнозированных значений модели-бенчмарка.
    """
    return np.round(_geometric_mean(np.abs(
        _relative_error(actual, predicted, benchmark))), 3)
```

Для более стабильного вычисления GMRAE применяют еще такой вариант.

```
# еще один вариант вычисления GMRAE
def gmrae2(actual, predicted, benchmark):
    """
    Вычисляет GMRAE.

    Параметры
    -----
```

```

actual: одномерный массив
    Массив фактических значений.
predicted: одномерный массив
    Массив спрогнозированных значений текущей модели.
benchmark: одномерный массив
    Массив спрогнозированных значений модели-бенчмарка.
"""
e_t = actual - predicted
e_t_bench = actual - benchmark
epsilon = np.finfo(np.float64).eps
gmrae = np.exp(np.mean(
    np.log(np.abs(e_t / (e_t_bench + epsilon)))))
return np.round(gmrae, 3)

```

Теперь вычислим GMRAE для нашей задачи прогнозирования количества туристов.

Момент времени t	Фактическое значение Y_t	Прогноз F_t	Ошибка прогноза $e_t = Y_t - F_t$	Прогноз бенчмарк-модели BF_t	Ошибка бенчмарк-прогноза $\hat{e}_t = Y_t - BF_t$	Абсолютная относительная ошибка $r_t = \left \frac{e_t}{\hat{e}_t} \right $
1	64	90	-26	68	-4	6,5
2	76	101	-25	75	1	25
3	35	44	-9	33	2	4,5
4	33	32	1	54	-21	0,05
5	29	31	-2	37	-8	0,25
6	35	44	-9	38	-3	3
7	47	102	-55	50	-3	18,33
Всего			-125			57,63

$$GMRAE = \sqrt[7]{6,5 \times 25 \times 4,5 \times 0,05 \times 0,25 \times 3 \times 18,33} = \sqrt[7]{478,8} = 2,41.$$

вычисляем GMRAE для задачи про туристов

```
print(gmrae(actl, fcst, benchmark_fcst))
print(gmrae2(actl, fcst, benchmark_fcst))
```

2.415
2.415

Наша модель работает хуже модели-бенчмарка. Теперь создадим ситуацию, когда одна из ошибок \hat{e}_t равна 0. Мы увидим, что значение метрики становится огромным.

```
# создадим ситуацию, когда одна из ошибок e_t_bench будет равна 0
# создаем массив прогнозов бенчмарк-модели
benchmark_fcst = np.array([64, 75, 33, 54, 37, 38, 50])

# вычисляем GMRAE
print(gmrae(actl, fcst, benchmark_fcst))
print(gmrae2(actl, fcst, benchmark_fcst))

507.119
507.119
```

2.4.4. Сравнение MRAE, MdRAE, GMRAE

Armstrong и Collopy (1992) рекомендуют использовать относительные абсолютные ошибки, особенно GMRAE и MdRAE. Fildes (1992) также предпочитает GMRAE, хотя он выражает его в эквивалентной (но более сложной) форме как квадратный корень из среднего геометрического квадратов относительных ошибок.

Серьезным недостатком метрик на основе относительных ошибок является то, что e_t^* может быть равна 0 – и тогда значение метрики станет огромным или бесконечным.

2.5. Относительные метрики качества (RelMAE, RelRMSE)

Вместо относительных ошибок можно использовать относительные метрики.

2.5.1. Относительная метрика MAE (relative MAE, RelMAE)

$$\text{RelMAE} = \frac{\text{MAE}}{\text{MAE}_b},$$

где

MAE – MAE рабочей модели;
 MAE_b – MAE модели-бенчмарка.

Чем меньше значение метрики, тем лучше качество модели.

Интерпретация RelMAE похожа на интерпретацию GMRAE: она показывает, во сколько раз наша модель оказалась хуже (или лучше), чем выбранная для сравнения (модель-бенчмарк). Если RelMAE = 1, то наша модель и модель-бенчмарк одинаково хороши; если RelMAE < 1, наша модель работает лучше, чем модель-бенчмарк, и, наконец, если RelMAE > 1, наша модель работает хуже, чем модель-бенчмарк.

Давайте вычислим RelMAE для нашей задачи прогнозирования количества туристов.

```
# вычисляем RelMAE для задачи про туристов
rel_mae = mae(actl, fcst) / mae(actl, benchmark_fcst)
rel_mae
```

3.0238095238095237

Наша модель работает хуже модели-бенчмарка.

2.5.2. Относительная метрика RMSE (relative RMSE, RelRMSE)

$$\text{RelRMSE} = \frac{\text{RMSE}}{\text{RMSE}_b},$$

где

RMSE – RMSE рабочей модели;

RMSE_b – RMSE модели-бенчмарка.

Чем меньше значение метрики, тем лучше качество модели.

Интерпретация RelRMSE тоже похожа на интерпретацию GMRAE: она показывает, во сколько раз наша модель оказалась хуже (или лучше), чем выбранная для сравнения (модель-бенчмарк). Если $\text{RelRMSE} = 1$, то наша модель и модель-бенчмарк одинаково хороши; если $\text{RelRMSE} < 1$, наша модель работает лучше, чем модель-бенчмарк, и, наконец, если $\text{RelRMSE} > 1$, наша модель работает хуже, чем модель-бенчмарк.

Давайте вычислим RelRMSE для нашей задачи прогнозирования количества туристов.

```
# вычисляем RelRMSE для задачи про туристов
rel_rmse = rmse(actl, fcst) / rmse(actl, benchmark_fcst)
rel_rmse
```

2.873880856345752

Наша модель работает хуже модели-бенчмарка.

2.6. МАСШТАБИРОВАННЫЕ ОШИБКИ (MASE, MdASE)

И относительные метрики, и метрики, основанные на относительных ошибках, пытаются уменьшить масштаб данных, сравнивая прогнозы с прогнозами, полученными с помощью какого-либо базового метода прогнозирования (модели-бенчмарка, обычно с помощью модели наивного прогноза). Однако у них обеих есть проблемы. Относительные ошибки имеют статистическое распределение с неопределенным средним и бесконечной дисперсией. Относительные метрики можно вычислить только при наличии нескольких прогнозов по одному и тому же ряду, поэтому их нельзя использовать для измерения качества прогнозов вне обучающей выборки на отдельном шаге горизонта прогнозирования.

Для решения этих проблем Роб Хайндман и Анне Келер в 2006 году предложили использовать масштабированные ошибки, которые прежде всего позволяют сравнивать наборы данных (временные ряды) разных масштабов.

2.6.1. Средняя абсолютная масштабированная ошибка (mean absolute scaled error, MASE)

Идея, лежащая в основе MASE, заключается в том, чтобы масштабировать ошибки на основе оценки MAE, полученной на обучающей выборке с помо-

щью методов наивного прогноза или наивного сезонного прогноза. Для временного ряда без сезонности формула MASE выглядит следующим образом:

$$MASE = \text{mean} \left(\frac{|e_j|}{\frac{1}{T-1} \sum_{t=2}^T |Y_t - Y_{t-1}|} \right) = \frac{\frac{1}{J} \sum_j |e_j|}{\frac{1}{T-1} \sum_{t=2}^T |Y_t - Y_{t-1}|}$$

вычисляем на тестовой выборке
вычисляем на обучающей выборке

Здесь числитель – средняя абсолютная ошибка, полученная на тестовой выборке. В ее основе лежит e_j – ошибка прогноза для данного момента времени в тестовой выборке (где J – количество прогнозов). Речь идет о разнице между фактическим значением (Y_j) и прогнозом (F_j) для данного момента времени в тестовой выборке: $e_j = Y_j - F_j$. Знаменатель – средняя абсолютная ошибка, полученная на обучающей выборке (здесь она задана как $t = 1 \dots T$) с помощью *одношагового метода наивного прогноза*, который использует в качестве прогноза последнее фактическое значение: $F_t = Y_{t-1}$. Проще говоря, оценку MAE, полученную с помощью прогнозов рабочей модели, делим на оценку MAE, полученную с помощью наивных прогнозов.

Для временного ряда с сезонностью формула MASE выглядит следующим образом:

$$MASE = \text{mean} \left(\frac{|e_j|}{\frac{1}{T-m} \sum_{t=m+1}^T |Y_t - Y_{t-m}|} \right) = \frac{\frac{1}{J} \sum_j |e_j|}{\frac{1}{T-m} \sum_{t=m+1}^T |Y_t - Y_{t-m}|}$$

Здесь числитель – средняя абсолютная ошибка, полученная на тестовой выборке. В ее основе лежит e_j – ошибка прогноза для данного момента времени в тестовой выборке (где J – количество прогнозов). Речь идет о разнице между фактическим значением (Y_j) и прогнозом (F_j) для данного момента времени в тестовой выборке: $e_j = Y_j - F_j$. Знаменатель – средняя абсолютная ошибка, полученная на обучающей выборке (здесь она задана как $t = 1 \dots T$) с помощью *одношагового метода наивного сезонного прогноза*, который использует в качестве прогноза последнее фактическое значение для того же времени года: $F_t = Y_{t-m}$, где m – количество периодов в полном сезонном цикле. Проще говоря, оценку MAE, полученную с помощью прогнозов рабочей модели, делим на оценку MAE, полученную с помощью наивных сезонных прогнозов.

Чем меньше значение метрики, тем лучше качество модели.

По мнению Роба Хайндмана и Анне Келер, MASE имеет ряд следующих желательных свойств.

1. *Инвариантность к масштабу:* MASE не зависит от масштаба данных, поэтому метрику можно использовать при сравнении прогнозов для наборов данных с разными масштабами.

2. Предсказуемое поведение, когда $y_t \rightarrow 0$: процентные метрики качества прогнозов типа средней абсолютной процентной ошибки (MAPE) основаны на делении на y_t , искажая распределение MAPE для значений y_t , близких к 0 или равных 0. Вспомним, что это особенно проблематично для данных, у которых шкала не имеет естественной нулевой точки (например, температура в градусах Цельсия или Фаренгейта).

3. *Симметрия:* средняя абсолютная масштабированная ошибка одинаково штрафует как положительные, так и отрицательные ошибки прогнозов, а также штрафует за ошибки при прогнозировании как маленьких, так и больших чисел.

4. *Интерпретируемость:* среднюю абсолютную масштабированную ошибку можно легко интерпретировать.

Если $MASE = 1$, то рассматриваемые прогнозы работают так же, как одношаговые прогнозы на основе обучающей выборки, полученные с помощью наивного метода.

Если $MASE < 1$, то рассматриваемые прогнозы работают лучше, чем одношаговые прогнозы на основе обучающей выборки, полученные с помощью наивного метода.

Если $MASE > 1$, то рассматриваемые прогнозы работают хуже, чем одношаговые прогнозы на основе обучающей выборки, полученные с помощью наивного метода.

Можно еще дать такую интерпретацию: MASE показывает, во сколько раз ошибка прогноза оказалась выше среднего абсолютного отклонения ряда в первых разностях.

5. *Асимптотическая нормальность:* для проверки статистической значимости разницы между двумя наборами прогнозов можно использовать критерий Диболда–Мариано для одношаговых прогнозов. Для проверки гипотезы с помощью статистики Диболда–Мариано желательно, чтобы $DM \sim N(0,1)$, где DM – значение статистики Диболда–Мариано. Philip Hans Franses (Филип Ханс Франсес) в своей статье «A note on the Mean Absolute Scaled Error» показал, что статистика Диболда–Мариано для MASE аппроксимирует это распределение, чего нельзя сказать о MAPE и SMAPE.

Таким образом, эту независимую от масштаба метрику можно использовать для сравнения методов прогноза по одному ряду, а также для сравнения качества прогнозов между рядами. Эта метрика хорошо подходит для серий с прерывистым спросом, поскольку она никогда не дает бесконечных или неопределенных значений, за исключением маловероятного случая, когда все значения в исторической выборке будут равны друг другу.

Давайте напишем собственную функцию для вычисления MASE.

```
# пишем функцию для вычисления MASE
def mase(y_true, y_pred, y_train, sp=1):
    """
    Вычисляет MASE.
```

Параметры

```

y_true: одномерный массив
    Массив фактических меток.
y_pred: одномерный массив
    Массив спрогнозированных меток.
y_train: одномерный массив
    Обучающий массив фактических меток
    (для вычисления наивного прогноза)
sp: int, значение по умолчанию 1
    Количество периодов в полном сезонном цикле.
"""

y_pred_naive = y_train[:-sp]
mae_naive = mean_absolute_error(y_train[sp:], y_pred_naive)
mae_pred = mean_absolute_error(y_true, y_pred)
epsilon = np.finfo(np.float64).eps
return np.round(mae_pred / np.maximum(mae_naive, epsilon), 3)

```

Рассмотрим пример с разными объемами продаж и вычислим MASE. У нас есть фактические значения для обучающей выборки, а также фактические значения и прогнозы для тестовой выборки.

	Товар 1	Товар 2	Итого
Фактические продажи для обучающей выборки	80 000	95 000 000	95 080 000
Фактические продажи для тестовой выборки	100 000	100 000 000	100 100 000
Прогноз для тестовой выборки	120 000	101 000 000	101 120 000
MASE			0,005
Ошибка прогноза	20 000	1 000 000	1 020 000

	Товар 1	Товар 2	Итого
Фактические продажи для обучающей выборки	80 000	95 000 000	95 080 000
Фактические продажи для тестовой выборки	100 000	100 000 000	100 100 000
Прогноз для тестовой выборки	110 000	105 000 000	105 110 000
MASE			0,026
Ошибка прогноза	10 000	5 000 000	5 010 000

При большей ошибке прогноза мы получили большее значение MASE, что вполне адекватно.

```

# создаем обучающий массив фактических меток
train_act = np.array([80000, 95000000])
# создаем тестовый массив фактических меток
test_act = np.array([100000, 100000000])

```

```
# создаем первый тестовый массив спрогнозированных меток
test_frcst = np.array([120000, 101000000])
# создаем второй тестовый массив спрогнозированных меток
test_frcst2 = np.array([110000, 105000000])

# печатаем значения метрик
print(mase(test_act, test_frcst, train_act, sp=1))
print(mase(test_act, test_frcst2, train_act, sp=1))
```

0.005
0.026

Значение MASE меньше 1 указывает на то, что наша модель прогнозирует лучше модели наивного прогноза.

Допустим, мы прогнозируем количество туристов в тысячах, которое посетит страну в ближайшие семь месяцев. У нас есть фактические значения для обучающей выборки, а также фактические значения и прогнозы для тестовой выборки.

Момент времени t	Фактическое значение для тестовой выборки Y_j	Фактическое значение для обучающей выборки Y_t	Наивный прогноз, вычисляемый на обучающей выборке Y_{t-1}	Абсолютная ошибка наивного прогноза $ Y_t - Y_{t-1} $	Прогноз модели для тестовой выборки F_j	Абсолютная ошибка прогноза $ e_j = Y_j - F_j $
1	64	54			65	1
2	76	66	54	12	67	9
3	35	25	66	41	36	1
4	33	23	25	2	34	1
5	29	19	23	4	100	71
6	35	25	19	6	36	1
7	47	37	25	12	48	1
Всего				77		85

$$\frac{1}{J} \sum_j |e_j| = \frac{85}{7} = 12,14,$$

$$\frac{1}{T-1} \sum_{t=2}^T |Y_t - Y_{t-1}| = \frac{77}{6} = 12,83,$$

$$MASE = \frac{12,14}{12,83} = 0,95.$$

```
# создаем обучающий массив фактических меток
train_actual = np.array([54, 66, 25, 23, 19, 25, 37])
# создаем тестовый массив фактических меток
test_actual = np.array([64, 76, 35, 33, 29, 35, 47])
# создаем тестовый массив спрогнозированных меток
test_forecast = np.array([65, 67, 36, 34, 100, 36, 48])
# вычисляем MASE для задачи про туристов
mase(test_actual, test_forecast, train_actual, sp=1)
```

0.946

Значение MASE, близкое к 1, указывает на то, что наша модель прогнозирует на уровне модели наивного прогноза.

Кроме того, MASE вычисляется с помощью класса `MeanAbsoluteScaledError` библиотеки прогнозирования временных рядов `sktime` (устанавливается с помощью команды `pip install sktime`).

```
# еще можно вычислить MASE с помощью класса MeanAbsoluteScaledError
# библиотеки sktime
from sktime.performance_metrics.forecasting import MeanAbsoluteScaledError
MASE = MeanAbsoluteScaledError()
MASE(test_actual, test_forecast, y_train=train_actual, sp=1)
```

0.9461966604823747

2.6.2. Медианная абсолютная масштабированная ошибка (median absolute scaled error, MdASE)

$$MdASE = \text{median} \left(\frac{|e_j|}{\frac{1}{T-1} \sum_{t=2}^T |Y_t - Y_{t-1}|} \right).$$

MdASE, в отличие от MASE, будет более устойчива к выбросам.
Давайте напишем собственную функцию для вычисления MASE.

```
# пишем функцию для вычисления MASE
def mdase(y_true, y_pred, y_train, sp=1):
    """
    Вычисляет MdASE.

    Параметры
    -----
    y_true: одномерный массив
        Массив фактических меток.
    y_pred: одномерный массив
        Массив спрогнозированных меток.
    y_train: одномерный массив
        Обучающий массив фактических меток
        (для вычисления наивного прогноза)
    sp: int, значение по умолчанию 1
        Количество периодов в полном сезонном цикле.
    """
```

```

y_pred_naive = y_train[:-sp]
mdae_naive = mdae(y_train[sp:], y_pred_naive)
mdae_pred = mdae(y_true, y_pred)
epsilon = np.finfo(np.float64).eps
return np.round(mdae_pred / np.maximum(mdae_naive, epsilon), 3)

```

Вычислим MdASE для примера с туристами.

Момент времени t	Фактическое значение для тестовой выборки y_j	Фактическое значение для обучающей выборки Y_t	Наивный прогноз, вычисляемый на обучающей выборке Y_{t-1}	Абсолютная ошибка наивного прогноза $ Y_t - Y_{t-1} $	Прогноз модели для тестовой выборки F_j	Абсолютная ошибка прогноза $ e_j = y_j - F_j $
1	64	54			65	1
2	76	66	54	12	67	9
3	35	25	66	41	36	1
4	33	23	25	2	34	1
5	29	19	23	4	100	71
6	35	25	19	6	36	1
7	47	37	25	12	48	1

Упорядочим абсолютные ошибки наивного прогноза.

2 4 6 12 12 41

Для четного количества наблюдений определяем медиану как среднее двух значений, занимающих центральное положение в ряду: $(6 + 12) / 2 = 9$.

Упорядочим абсолютные ошибки прогноза. Здесь медианой будет 1.

1 1 1 1 1 9 71

$$MdASE = \frac{1}{9} = 0,111.$$

```
# вычисляем MdASE для задачи про туристов
mdase(test_actual, test_forecast, train_actual, sp=1)
```

0.111

Кроме того, MdASE можно вычислить с помощью класса `MeanAbsoluteScaledError` библиотеки прогнозирования временных рядов `sktime`.

```
# еще можно вычислить MdASE с помощью класса MedianAbsoluteScaledError
# библиотека sktime
from sktime.performance_metrics.forecasting import MedianAbsoluteScaledError
MDASE = MedianAbsoluteScaledError()
```

```
MDASE(test_actual, test_forecast, y_train=train_actual, sp=1)
```

0.1111111111111111

2.7. КРИТЕРИЙ ДИБОЛДА–МАРИАНО

Критерий Диболда–Мариано (Diebold-Mariano test) – статистический тест, позволяющий сравнивать качество прогнозов временного ряда, полученных с помощью двух прогнозных моделей. Впервые был представлен в работе Диболда и Мариано в 1995 году, где был приведен небольшой обзор тестов такого рода.

Представим, у нас есть прогнозные значения двух моделей $\{\hat{y}_{mt}\}_{t=1}^T$ и $\{\hat{y}_{nt}\}_{t=1}^T$ для временного ряда $\{y_t\}_{t=1}^T$. Пусть $\{e_{mt}\}_{t=1}^T$ и $\{e_{nt}\}_{t=1}^T$ будут ошибками прогнозов (остатками), а $g(e)$ – это функция потерь. Нулевую гипотезу об одинаковом качестве прогнозов обеих моделей m и n можно записать так: $E[g(e_{mt})] = E[g(e_{nt})]$ или $E[d_t] = 0$, где $d_t = [g(e_{mt}) - g(e_{nt})]$ – это разность значений функции потерь (далее кратко – разность потерь) для моделей m и n . Таким образом, нулевая гипотеза об «одинаковом качестве» эквивалентна нулевой гипотезе о том, что среднее временного ряда разностей потерь в генеральной совокупности равно 0.

Рассмотрим величину

$$\sqrt{T}(\bar{d} - \mu),$$

где

$\bar{d} = \frac{1}{T} \sum_{t=1}^T d_t = \frac{1}{T} \sum_{t=1}^T [g(e_{mt}) - g(e_{nt})]$ – выборочное среднее временного ряда разностей потерь;

μ – среднее временного ряда разностей потерь в генеральной совокупности.

$f_d(0) = \frac{1}{2A} \left(\sum_{k=-\infty}^{\infty} \gamma_d(k) \right)$ – спектральная плотность временного ряда разностей

потерь на частоте 0, где $\gamma_d(k)$ – автокорреляция временного ряда разностей потерь в лаге k .

Если временной ряд разностей потерь $\{d_t\}_{t=1}^T$ ковариационно-стационарен и обладает короткой памятью, то можно показать, что

$$\begin{aligned} \sqrt{T}(\bar{d} - \mu) &\xrightarrow{d} N(0, 2\pi f_d(0)) \\ &\downarrow \\ \frac{\bar{d} - \mu}{\sqrt{\frac{2\pi f_d(0)}{T}}} &\rightarrow N(0, 1). \end{aligned}$$

Согласно нулевой гипотезе:

$$\frac{\bar{d}}{\sqrt{\frac{2\pi f_d(0)}{T}}} \rightarrow N(0,1).$$

Проиллюстрируем применение теста. Допустим, у нас есть набор фактических значений, и с помощью двух моделей получены прогнозы (в данном случае – одношаговые). Нужно выяснить, является ли качество прогнозов двух моделей одинаковым. В качестве метрики качества используем MAPE. Мы воспользуемся собственным классом `DieboldMarianoTest`. С помощью параметра `crit` можно задать метрику качества. С помощью параметра `h` мы задаем горизонт прогнозирования. Этот параметр очень важен, потому что критерий Диболда–Мариано должен принимать во внимание (благодаря своей оценке дисперсии), что чем дальше прогноз по времени, тем менее точным он, вероятно, будет, поскольку существует большая неопределенность в отношении будущего. Соответственно, прогнозы двух моделей для более длинных горизонтов будут отличаться на большую величину. С помощью параметра `seasonal_period` задаем количество периодов в полном сезонном цикле (например, 12 – для ежемесячных данных, 4 – для ежеквартальных данных), используется только при расчете MASE.

```
# импортируем класс DieboldMarianoTest
from dm_test import DieboldMarianoTest
# массив фактических значений
actual = np.array([9.5, 11.2, 12.1, 16.5, 18.1])
# массив прогнозов (одношаговых)
forecast1 = np.array([10.1, 11.1, 15.3, 16.2, 18.9])
forecast2 = np.array([10.3, 11.9, 15.5, 16.8, 18.7])
# печатаем метрики качества прогнозов
print("MAPE_1-я_модель: %.3f" % sklearn_mape(actual, forecast1))
print("MAPE_2-я_модель: %.3f" % sklearn_mape(actual, forecast2))
# печатаем результаты теста Диболда-Мариано
dmt_mape = DieboldMarianoTest(crit='MAPE', h=1, seasonal_period=1)
results = dmt_mape.db_test(actual, forecast1, forecast2)
print("Результаты теста:", results)

MAPE_1-я_модель: 0.080
MAPE_2-я_модель: 0.096
Результаты теста: dm_return(DM=-1.4548887520458251, p_value=0.21939120174583787)
```

У нас нет оснований отклонить нулевую гипотезу об одинаковом качестве прогнозов обеих моделей.

Часть 5

Другие полезные библиотеки и платформы

1. Библиотеки баейсовской оптимизации `hyperopt`, `scikit-optimize` и `optuna`

1.1. Недостатки обычного поиска по сетке и случайного поиска по сетке

Случайный поиск по сетке может привести к выбору субоптимальных значений гиперпараметров, особенно когда выбрано маленькое количество итераций. У обычного поиска также много недостатков. Он перебирает много заранее неудачных значений. Допустим, уже имеется информация, что увеличение глубины деревьев ухудшает качество случайного леса. Человек может понять, что более высокие значения глубины точно дадут плохой результат, и догадается не проверять лишний раз эти значения. Обычный поиск по сетке так делать не умеет.

Если гиперпараметров много, то размер «ячейки» приходится делать слишком крупным, и можно упустить хороший оптимум. Это часто встречающаяся ситуация, поскольку обычно с помощью поиска по сетке или случайного поиска исследователь данных осуществляет некую ручную настройку гиперпараметров для большого количества моделей, таких как дерево решений, метод опорных векторов, метод k ближайших соседей. Например, специалист нашел оптимальные настройки для метода опорных векторов, но упустил таковые для метода деревьев решений, как раз сделал размер «ячейки» крупным (проверяя глубину 4, 8, 12, а оптимальной может быть глубина 2, например в градиентном бустинге часто хороший результат дают деревья с такой глубиной). Это, в свою очередь, означает некорректность сравнения.

Следует отметить, что если включить в пространство поиска много лишних гиперпараметров, никак не влияющих на результат, то обычный поиск по сетке будет работать намного хуже при том же числе итераций, чем случайный поиск.

Чтобы уменьшить количество итераций, необходимых для поиска оптимального значения гиперпараметра или оптимальной комбинации значений гиперпараметров, были придуманы адаптивные баевесовские методы. Они выбира-

ют следующую точку для проверки, учитывая результаты на уже проверенных точках. Идея состоит в том, чтобы на каждом шаге найти компромисс между (а) исследованием регионов рядом с самыми удачными точками среди найденных и (б) исследованием регионов с большой неопределенностью, где могут находиться еще более удачные точки. Это часто называют дилеммой «explore vs exploit» (дилемма «разведка–эксплуатация») или «learning vs earning» (дилемма «познание нового – зарабатывание на старом»). Вы можете выбрать то, что знаете, и получите что-то близкое к тому, что ожидаете («эксплуатация»), или можете выбрать то, в чем вы не уверены, и, возможно, узнаете больше («исследование»). Старый ресторан или новый? Привычный маршрут домой или новый? Продолжать работать на прежней работе или поддаться уговорам хедхантеров? Яркий пример – голливудские фильмы. Поскольку прибыль в киноиндустрии снижается, это смещает предпочтение кинобоссов к эксплуатации хорошо известных кинофраншиз. Поэтому в будущем мы увидим «Мстители 10» и «Форсаж 15», а не какие-то принципиально новые фильмы о киногероях и автогонках.

Итак, в ситуациях, когда проверка каждой новой точки стоит дорого, с помощью байесовских аддитивных методов можно приблизиться к глобальному оптимуму за гораздо меньшее число шагов. Поговорим более подробно о байесовской оптимизации гиперпараметров.

1.2. ЗНАКОМСТВО С БАЙЕСОВСКОЙ ОПТИМИЗАЦИЕЙ

Представьте, у нас есть следующий график (рисунок ниже: чем ниже значение метрики RMSE, тем лучше). Где имеет смысл сконцентрировать наш поиск?

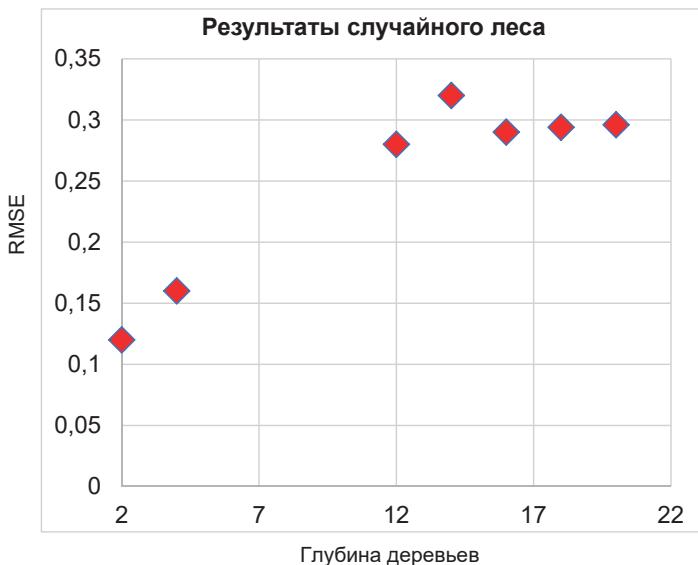


Рис.1 Зависимость RMSE от глубины деревьев

Случайный поиск и обычный поиск по сетке вообще не обратят внимания на прошлые результаты и будут продолжать поиск по всему диапазону зна-

чений, хотя очевидно, что оптимальный ответ (вероятно) лежит в небольшой области (области, соответствующей значениям небольшой глубины)!

Если же мы скажем, что поиск нужно осуществлять с помощью менее глубоких деревьев, то у нас уже есть идея баевсовской оптимизации! Мы хотим сосредоточиться на наиболее многообещающих значениях гиперпараметров, и если у нас уже есть зафиксированные оценки, то имеет смысл использовать эту информацию для нашего последующего выбора.

Баевсовские подходы, в отличие от случайного поиска или поиска по сетке, отслеживают предыдущие оценки, которые они используют для построения вероятностной модели, отображающей значения гиперпараметров в вероятность оценки целевой функции:

$$P(\text{оценка}|\text{значения гиперпараметров}).$$

В литературе эта модель называется «суррогатом» целевой функции и ее записывают как $p(y|x)$. В суррогатной оптимизации мы используем суррогатную или аппроксимационную функцию для оценки целевой функции с помощью семплирования – отбора ограниченного количества точек. Суррогатную функцию намного легче оптимизировать, чем целевую функцию, и баевсовские методы работают, находя следующую комбинацию значений гиперпараметров для оценки фактической целевой функции, выбирая гиперпараметры, которые лучше всего оптимизируют суррогатную функцию. Другими словами:

1. Строим суррогатную вероятностную модель целевой функции.
2. Находим комбинацию значений гиперпараметров, которая наилучшим образом оптимизирует суррогатную функцию.
3. Применяем эти гиперпараметры к фактической целевой функции.
4. Обновляем суррогатную модель, включающую новые результаты.
5. Повторяем шаги 2–4, пока не будет достигнуто максимальное количество итераций или пока не закончится заданное время выполнения.

Цель баевсовских рассуждений – совершать «меньше ошибок» по мере получения большего объема информации, что, собственно, эти подходы и делают, постоянно обновляя суррогатную вероятностную модель после каждой оценки целевой функции.

В целом баевсовские методы оптимизации эффективны, потому что они выбирают последующие гиперпараметры осознанным образом. Основная идея такова: потратим немного больше времени на выбор последующих гиперпараметров, чтобы сделать меньше вызовов целевой функции. На практике время, затрачиваемое на выбор последующих гиперпараметров, несущественно по сравнению со временем, потраченным на оценку целевой функции. Оценивая гиперпараметры, которые кажутся более многообещающими, исходя из прошлых результатов, баевсовские методы могут найти лучшие гиперпараметры модели, чем случайный поиск, за меньшее количество итераций.

На первых итерациях суррогат будет плохим приближением фактической целевой функции, однако уже после 5–10 итераций суррогат будет достаточно точно соответствовать фактической функции.

Можно сказать, что баевсовские методы работают во многом так же, как наше мышление: мы формируем первоначальный взгляд на мир (называемый

априорной моделью), а затем обновляем нашу модель на основе нового опыта (обновленная модель называется апостериорной). Байесовская оптимизация гиперпараметров использует этот принцип и применяет его для поиска лучших значений настроек модели!

1.3. ПОСЛЕДОВАТЕЛЬНАЯ ОПТИМИЗАЦИЯ ПО МОДЕЛИ (SEQUENTIAL MODEL-BASED OPTIMIZATION – SMBO)

Одним из видов баевской оптимизации является последовательная оптимизация по модели (Sequential model-based optimization – SMBO). Слово «последовательная» означает выполнение испытаний (trials) одно за другим, каждый раз пробуя комбинации лучших значений гиперпараметров, применяя баевские рассуждения и обновляя вероятностную (суррогатную) модель.

Существует пять аспектов последовательной оптимизации гиперпараметров по модели:

- 1) область значений гиперпараметров для поиска (scope);
- 2) целевая функция (objective function), которая принимает гиперпараметры и выдает оценку, которую мы хотим минимизировать (или максимизировать);
- 3) суррогатная модель целевой функции (surrogate function);
- 4) критерий, называемый функцией отбора (acquisition function), для оценки того, какие значения гиперпараметров выбрать в качестве последующих из результатов суррогатной модели;
- 5) история (history), состоящая из пар «значение функции – комбинация значений гиперпараметров», используемых алгоритмом для обновления суррогатной модели.

При этом есть несколько вариантов методов SMBO, которые отличаются на этапах 3–4, а конкретно отличаются тем, как они создают суррогат целевой функции и критерии, используемые для выбора последующих гиперпараметров.

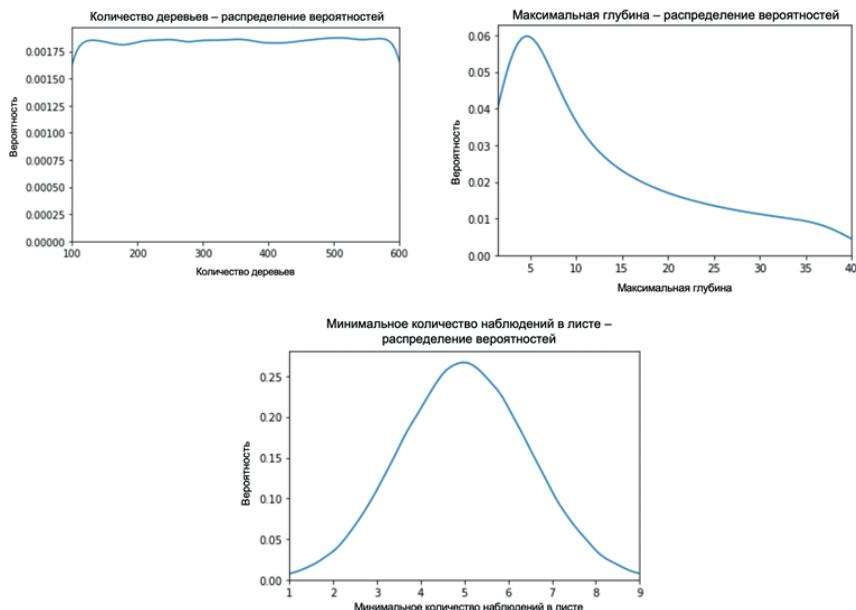
1.3.1. Область значений гиперпараметров

В ситуации случайного поиска и обычного поиска областью значений для поиска является сетка. Пример для случайного леса показан ниже:

```
hyperparameter_grid = {
    'n_estimators': [100, 200, 300, 400, 500, 600],
    'max_depth': [2, 5, 10, 15, 20, 25, 30, 35, 40],
    'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8]
}
```

Для подхода, основанного на оптимизации по модели, область состоит из *распределений вероятностей* (*probability distributions*). Как и в случае с сеткой, это позволяет нам зафиксировать значение предметной области в процессе поиска, увеличивая вероятность в тех областях, где, по нашему мнению, лежат истинные лучшие значения гиперпараметров.

Если мы захотим выразить вышеуказанную сетку в виде распределений вероятностей, она может выглядеть примерно так:



Здесь у нас – равномерное, логарифмическое и нормальное распределения. Они основаны на предшествующей практике/знаниях.

1.3.2. Целевая функция

Целевая функция принимает гиперпараметры и выдает оценку в виде отдельного вещественного числа, которое мы хотим минимизировать (или максимизировать). В качестве примера рассмотрим случай построения случайного леса для задачи регрессии. Гиперпараметры, которые мы хотим оптимизировать, показаны в сетке значений гиперпараметров ниже, а показатель, который нужно минимизировать, представляет собой корень из среднеквадратичной ошибки. Тогда наша целевая функция будет выглядеть в Python так:

```
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# функция, отображающая значения гиперпараметров
# в вещественное число
def objective(hyperparameters):

    # модель машинного обучения
    rf = RandomForestRegressor(**hyperparameters)

    # обучение
    rf.fit(X_train, y_train)

    # получение прогнозов и оценка качества
    predictions = rf.predict(X_valid)
    rmse = mean_squared_error(y_valid, predictions, squared=False)

    return rmse
```

Хотя целевая функция выглядит просто, ее вычисление является очень дорогим. Если бы целевую функцию можно было быстро вычислить, то мы могли бы попробовать каждую возможную комбинацию значений гиперпараметров (как при поиске по сетке). Если мы используем простую модель, небольшую сетку гиперпараметров и небольшой набор данных, то это может быть лучшим способом. Однако в тех случаях, когда для оценки целевой функции могут потребоваться часы или даже дни, мы хотим ограничить количество обращений к ней.

Вся концепция байесовской оптимизации по модели заключается в сокращении числа оценок целевой функции. Мы выбираем только наиболее многообещающий набор гиперпараметров для оценки, руководствуясь предыдущими результатами. Каждый последующий набор гиперпараметров выбирается на основе аппроксимации целевой функции, называемой суррогатом.

1.3.3. Суррогатная функция (вероятностная модель)

Суррогатная функция является вероятностным представлением целевой функции, построенным с использованием предыдущих оценок. Существует несколько различных форм суррогатной функции, например гауссовые процессы (Gaussian Processes – GP) и деревья оценок (Tree Parzen Estimators – TPE), предложенные Бергстра и др. в статье «Algorithms for Hyper-Parameter Optimization» («Алгоритмы оптимизации гиперпараметров»): <https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cf12577bc2619bc635690-Paper.pdf>. Эти методы отличаются тем, как они конструируют суррогатную функцию.

1.3.4. Функция отбора

Функция отбора – это критерий, в соответствии с которым выбирается следующая комбинация значений гиперпараметров. Наиболее распространенным вариантом является ожидаемое улучшение (expected improvement):

$$\text{EI}_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy.$$

Здесь y^* – пороговое значение целевой функции, x – предложенная комбинация значений гиперпараметров, y – фактическое значение целевой функции в случае использования комбинации значений гиперпараметров x , а $p(y|x)$ – суррогатная вероятностная модель, представляющая собой вероятность y при данном x . Если говорить просто и кратко, цель состоит в том, чтобы максимизировать ожидаемое улучшение относительно x . Это означает поиск лучших значений гиперпараметров в соответствии с суррогатной функцией $p(y|x)$.

Если $p(y|x)$ равно нулю везде, где $y < y^*$, то нельзя ожидать, что комбинация значений гиперпараметров x даст какое-либо улучшение. Если интеграл положителен, то это означает, что комбинация значений гиперпараметров x должна давать лучший результат, чем пороговое значение.

Ранее мы говорили, что методы SMBO отличаются тем, как они строят суррогатную модель $p(y|x)$. Деревья оценок Парзена строят модель, применяя правило Байеса.

Вместо непосредственного представления $p(y|x)$ дерева оценок Парзена используют:

$$p(y|x) = \frac{p(x|y)^* p(y)}{p(x)}$$

(правило Байеса в действии!),

где $p(y|x)$ является вероятностью значений гиперпараметров для данного значения целевой функции и, в свою очередь, выражается как

$$p(x|y) = \begin{cases} l(x), & \text{если } y < y^* \\ g(x), & \text{если } y \geq y^* \end{cases}$$

Мы создаем два разных распределения для значений гиперпараметров: $l(x)$, где значение целевой функции меньше порога $y < y^*$, и $g(x)$, где значение целевой функции больше или равно порогу $y \geq y^*$.

Давайте изменим наш рисунок со случайным лесом, добавив порог.

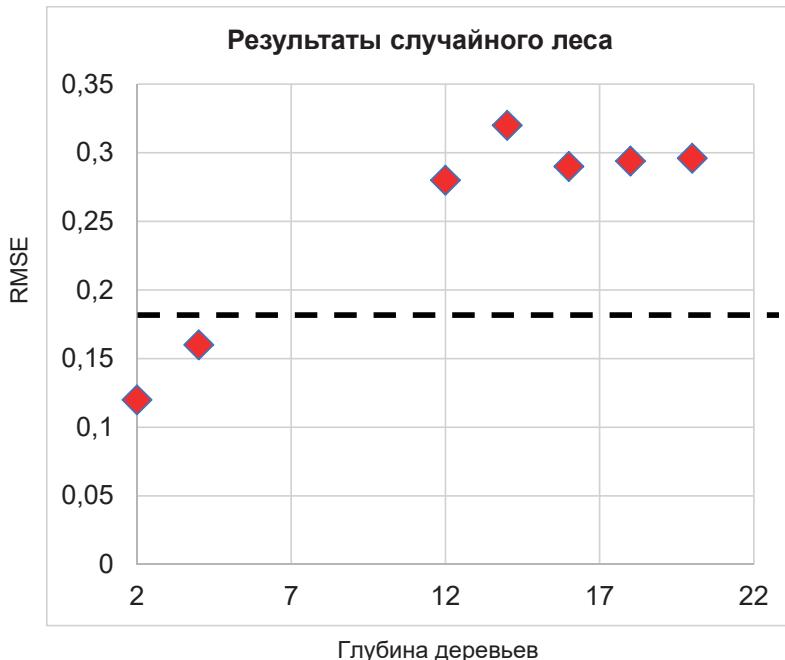


Рис. 2 Появление порога в графике зависимости RMSE от глубины деревьев

Теперь мы построим два распределения вероятностей для глубины деревьев, одно из которых использует оценки, давшие значения ниже порога, а другое — оценки, давшие значения выше порога.



Рис. 3 Два распределения вероятностей для глубины деревьев

Интуитивно кажется, что нужно извлечь значения x из $l(x)$, а не из $g(x)$, потому что это распределение основано на значениях x , которые дали более низкие оценки, чем пороговое значение.

После применения правила Байеса и выполнения нескольких замен уравнение ожидаемого улучшения (которое мы пытаемся максимизировать) становится:

$$EI_{y^*}(x) = \frac{\gamma y^* l(x) - l(x) \int_{-\infty}^{y^*} p(y) dy}{\gamma l(x) + (1-\gamma) g(x)} \propto \left(\gamma + \frac{g(x)}{l(x)} (1-\gamma) \right)^{-1}.$$

Крайний правый член – самая важная часть уравнения. Он говорит о том, что ожидаемое улучшение пропорционально отношению $l(x)/g(x)$, и поэтому, чтобы максимизировать ожидаемое улучшение, мы должны максимизировать это отношение. Наша догадка была правильной: мы должны извлечь значения гиперпараметров, которые более вероятны при $l(x)$, чем при $g(x)$!

Деревья оценок Парзена работают, извлекая комбинацию значений гиперпараметров из $l(x)$, оценивая ее с точки зрения $l(x)/g(x)$ и возвращая комбинацию, которая дает наибольшее значение согласно $l(x)/g(x)$, что соответствует наибольшему ожидаемому улучшению. Затем эти значения гиперпараметров оцениваются по целевой функции. Если суррогатная функция корректна, то при оценке эти гиперпараметры должны давать лучшее значение!

Критерий ожидаемого улучшения позволяет модели найти компромисс между разведкой и эксплуатацией. Поскольку $l(x)$ является распределением, а не единственным значением, это означает, что извлеченные значения гиперпараметров, вероятно, не гарантируют максимального ожидаемого улуч-

шения, но близки к нему. Более того, поскольку суррогат является лишь аппроксимацией целевой функции, выбранные значения гиперпараметров могут и не дать улучшения на этапе оценки и суррогатную модель нужно обновить. Это обновление выполняется на основе текущей суррогатной модели и истории оценок целевой функции.

1.3.5. История оценок целевой функции

Каждый раз, когда алгоритм предлагает новую комбинацию возможных значений гиперпараметров, он оценивает ее с помощью фактической целевой функции и записывает результат в виде пары (оценка целевой функции, комбинация гиперпараметров). Эти записи формируют *историю*. Алгоритм создает $l(x)$ и $g(x)$, используя историю, чтобы получить вероятностную модель целевой функции, которая улучшается с каждой итерацией.

Это и есть правило Байеса в действии: у нас есть первоначальная оценка суррогата целевой функции, которую мы обновляем, когда собираем больше информации. В конце концов, при достаточном количестве оценок целевой функции мы надеемся, что наша модель точно отражает целевую функцию, а значения гиперпараметров, дающие наибольшее ожидаемое улучшение, соответствуют значениям гиперпараметров, максимизирующими/минимизирующими целевую функцию.

1.3.6. Собираем все вместе

Поскольку алгоритм предлагает лучшие значения гиперпараметров для оценки, оптимизация целевой функции осуществляется намного быстрее, чем при случайном поиске или поиске по сетке, что приводит к меньшему общему количеству оценок целевой функции.

Хотя алгоритм тратит больше времени на выбор последующих значений гиперпараметров путем максимизации ожидаемого улучшения, это намного дешевле с точки зрения вычислительных затрат, чем оценка целевой функции. В статье об использовании SMBO с TPE <http://proceedings.mlr.press/v28/bergstra13.pdf> авторы сообщают, что поиск последующего набора возможных значений гиперпараметров занял несколько секунд, тогда как оценка фактической целевой функции заняла часы.

Если мы используем более информированные методы для выбора последующих значений гиперпараметров, это означает, что мы можем тратить меньше времени на оценку неправильного набора значений гиперпараметров. Кроме того, последовательная оптимизация по модели с использованием деревьев оценок Парзена способна находить лучшие значения гиперпараметров, чем случайный поиск, при том же количестве испытаний. Другими словами, мы получаем:

- сокращенное время поиска значений гиперпараметров;
- лучшие значения на тестовом наборе.

1.3.7. Реализации последовательной оптимизации по модели

К питоновским библиотекам, в которых реализована SMBO, можно отнести Spearmint <https://github.com/JasperSnoek/spearmint>, MOE <https://github.com/Yelp/MOE> и Scikit-Optimize <https://scikit-optimize.github.io/stable/>, использующие гауссов

процесс для суррогата, Hyperopt <https://github.com/hyperopt/hyperopt> и Optuna <https://github.com/optuna/optuna>, использующие деревья оценок Парзена, SMAC <https://github.com/automl/SMAC3>, использующую регрессию случайного леса. Все эти библиотеки применяют критерий ожидаемого улучшения для выбора последующих значений гиперпараметров в суррогатной модели. На сегодняшний день наиболее активно развивающейся среди библиотек байесовской оптимизации можно назвать библиотеку Optuna. Мы рассмотрим библиотеки Hyperopt, Scikit-Optimize и Optuna.

1.4. HYPEROPT

Библиотека hyperopt устанавливается с помощью команды `pip install scikit-optimize`. В ней реализован метод деревьев оценок Парзена. Преимущество его в том, что он может работать с очень разными пространствами значений гиперпараметров. По мнению автора Джеймса Бергстра, этот алгоритм достаточно хорошо решает проблему `explore-exploit` и работает лучше как обычного поиска по сетке, так и экспертного перебора в случае применения градиентного бустинга и глубокого обучения, где гиперпараметров особенно много.

Давайте загрузим необходимые нам библиотеки, классы и функции.

```
# импортируем необходимые библиотеки, классы и функции
import pandas as pd
import numpy as np
import json
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
from sklearn.base import clone
from sklearn.model_selection import (train_test_split,
                                      cross_val_score)
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import GradientBoostingClassifier
from lightgbm import LGBMClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
import hyperopt
from hyperopt import fmin, tpe, hp, Trials, space_eval
```

Основной функцией библиотеки hyperopt является функция `fmin()`. Она принимает в себя функцию и возвращает оптимальные гиперпараметры, с которыми значение функции будет минимально (если вам надо максимизировать, то нужно сменить знак вашей функции на противоположный).

Допустим, дана функция, определенная на некотором интервале, и нам нужно найти ее минимум. Иными словами, мы хотим определить такое значение аргумента, которое даст наименьшее значение функции. Тривиальный пример ниже находит такое значение x , которое минимизирует линейную функцию $y(x) = x$.

```
# рассмотрим простой пример
best = fmin(
    fn=lambda x: x, ←———— Оптимизируемая функция
    space=hp.uniform('x', 0, 1), ←———— Пространство поиска
    algo=tpe.suggest, ←———— Алгоритм поиска
    max_evals=100 ←———— Количество итераций поиска
    rstate=np.random.default_rng(123)) ←———— Стартовое значение генератора
# печатаем результат
print(best)
```

```
100%|██████████| 100/100 [00:00<00:00, 552.25trial/s, best loss: 0.0005330086077229319]
{'x': 0.0005330086077229319}
```

Функция `fmin()` сначала принимает функцию, у которой необходимо найти минимум, в данном случае мы ее здесь задали с помощью анонимной функции `lambda x: x`. На ее месте могла бы быть любая функция, возвращающая значение, например абсолютная или квадратичная функция потерь в задаче регрессии.

Параметр `space` определяет пространство поиска, и в этом примере речь идет о непрерывном диапазоне от 0 до 1, заданном `hp.uniform('x', 0, 1)`.

Пространство поиска задается весьма легко. Нужно всего лишь указать тип распределения и его границы. Приведем здесь основные типы:

`hp.choice(label, options)`, где `label` – название гиперпараметра, `options` представляет собой список или кортеж Python

равновероятный выбор из множества

`hp.randint(label, upper)`, где `label` – название гиперпараметра, `upper` – верхнее значение

случайное целое число в диапазоне от 0 до `upper`
смысл этого распределения состоит в том, что соседние целочисленные значения гиперпараметров скоррелированы сильнее, чем более удаленные друг от друга

`hp.uniform(label, low, high)`, где `label` – название гиперпараметра, `low` и `high` – это нижняя и верхняя границы диапазона поиска соответственно

равномерное непрерывное распределение

`hp.quniform(label, low, high, q)` – название гиперпараметра, `low` и `high` – это нижняя и верхняя границы диапазона поиска соответственно, `q` – корректирующий коэффициент

значение типа `round(uniform(low, high) / q) * q`
равномерное непрерывное распределение с округлением

`hp.normal(label, mu, sigma)`, где `label` – название гиперпараметра, `mu` и `sigma` – это среднее и стандартное отклонение соответственно

нормальное непрерывное распределение

`hp.lognormal(label, low, high)`, где `label` – название гиперпараметра, `low` и `high` – это нижняя и верхняя границы диапазона поиска соответственно

логнормальное непрерывное распределение

`hp.loguniform(label, low, high)`, где `label` – название гиперпараметра, `low` и `high` – это нижняя и верхняя границы диапазона поиска соответственно

логравномерное непрерывное распределение

Пространство поиска может быть списком, кортежем, словарем:

```
list_space = [
    hp.uniform('a', 0, 1),
    hp.loguniform('b', 0, 1)]

tuple_space = (
    hp.uniform('a', 0, 1),
    hp.loguniform('b', 0, 1))

dict_space = {
    'a': hp.uniform('a', 0, 1),
    'b': hp.loguniform('b', 0, 1)}
```

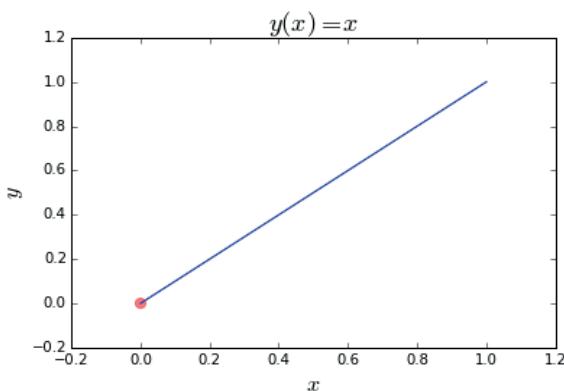
Hyperopt учится, получая значения функции, адаптируется и отбирает такие участки исходного пространства поиска, где, по его мнению, можно получить наиболее оптимальное значение функции.

Параметр `algo` обозначает алгоритм поиска, в данном случае `tpe`, означающий *деревья оценок Парзена*. В качестве значения аргумента `algo` можно задать `hyperopt.random` – случайный поиск.

Наконец, с помощью параметра `max_evals` мы определяем максимальное количество итераций поиска. Функция `fmin()` возвращает питоновский словарь значений.

Для нашего примера получаем решение `{'x': 0.0005330086077229319}`.

Ниже приведен график функции. Красная точка и есть то, что мы пытались найти.



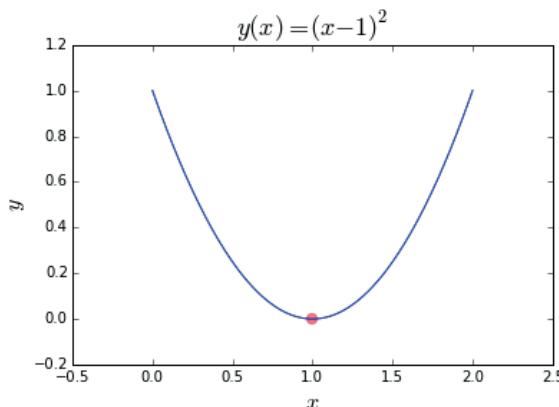
А вот более сложная функция: `lambda x: (x-1)**2`. На этот раз мы попробуем найти минимум квадратичной функции $y(x)=(x-1)^2$. Итак, изменим область поиска, чтобы включить предполагаемое оптимальное значение $x=1$ плюс некоторые субоптимальные значения, лежащие по разным сторонам от него: `hp.uniform('x', -2, 2)`.

```
# еще один пример
best = fmin(
    fn=lambda x: (x-1)**2,
    space=hp.uniform('x', -2, 2),
    algo=tpe.suggest,
```

```
max_evals=100,
rstate=np.random.default_rng(123))
print(best)

100%|██████████| 100/100 [00:00<00:00, 568.29trial/s, best loss: 3.239117511790906e-06]
{'x': 1.001799754847692}
```

Найденная точка на графике функции выглядит следующим образом:



Было бы неплохо посмотреть, что происходит внутри черного ящика Hyperopt. Объект Trials позволит нам это сделать. Он сохраняет значения функции в каждой итерации (каждом испытании). Мы можем посмотреть значения функции для интересующего гиперпараметра в данной итерации.

Сейчас мы найдем такое значение x , которое минимизирует квадратичную функцию $y(x)=x^2$.

```
# найдем такое значение x, которое минимизирует
# квадратичную функцию y(x) = x**2

# задаем пространство поиска
fspace = {
    'x': hp.uniform('x', -5, 5)
}

# оптимизируемая функция
def objective(params):
    x = params['x']
    return x**2

# записываем историю
trials = Trials()

# выполняем оптимизацию
best = fmin(fn=objective,
            space=fspace,
            algo=tpe.suggest,
            max_evals=1000,
            trials=trials,
            rstate=np.random.default_rng(123))
print("best:", best)
```

```
100%|██████████| 1000/1000 [00:03<00:00, 261.41it/s, best loss: 1.7829880251661697e-08]
best: {'x': 0.0009295936447884516}
```

Кстати, начиная с версии 0.2.6 мы можем задавать пространство поиска и внутри оптимизируемой функции.

```
# пространство поиска теперь можно задавать
# и внутри оптимизируемой функции
def objective(x: hp.uniform('x', -5, 5)):
    return x**2

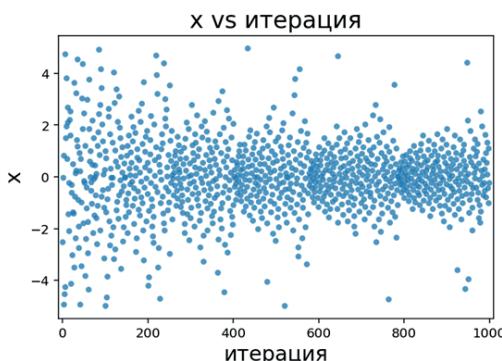
# записываем историю
trials = Trials()

# выполняем оптимизацию
best = fmin(
    fn=objective,
    space='annotated',
    algo=tpe.suggest,
    max_evals=1000,
    trials=trials,
    rstate=np.random.default_rng(123)
)
print("best:", best)
```

```
100%|██████████| 1000/1000 [00:03<00:00, 261.41it/s, best loss: 1.7829880251661697e-08]
best: {'x': 0.0009295936447884516}
```

Теперь с помощью истории оптимизации, записанной в `trials.trials`, построим две визуализации – график значение аргумента *vs* итерация и график значение функции *vs* значение аргумента. Начнем с графика значения аргумента *vs* итерация.

```
# визуализация значение аргумента vs итерация
f, ax = plt.subplots(1)
xs = [t['tid'] for t in trials.trials]
ys = [t['misc']['vals']['x'] for t in trials.trials]
ax.set_xlim(xs[0]-10, xs[-1]+10)
ax.scatter(xs, ys, s=20, linewidth=0.01, alpha=0.75)
ax.set_title('x vs итерация', fontsize=18)
ax.set_xlabel('итерация', fontsize=16)
ax.set_ylabel('x', fontsize=16)
```



Можно заметить, что изначально алгоритм выбирал значения из всего пространства поиска равномерно, но, получив со временем больше информации о том, как гиперпараметры влияют на функцию, алгоритм все больше фокусируется на областях, где, как он считает, он может достичь оптимума – значений, близких к нулю. И хотя алгоритм исследует по-прежнему все пространство решений, но делает это реже.

Теперь посмотрим на график *значение функции vs значение аргумента*.

```
# визуализация значение функции vs значение аргумента
f, ax = plt.subplots(1)
xs = [t['misc']['vals'][x] for t in trials.trials]
ys = [t['result']['loss'] for t in trials.trials]
ax.scatter(xs, ys, s=20, linewidth=0.01, alpha=0.75)
ax.set_title('значение функции vs x', fontsize=18)
ax.set_xlabel('x', fontsize=16)
ax.set_ylabel('значение функции', fontsize=16)
```



Загрузим уже знакомые расширенные данные компании StateFarm, создадим списки переменных, создадим трансформеры, список трансформеров передадим в `ColumnTransformer`, зададим итоговый конвейер.

```
# загружаем уже знакомые данные
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)

# создаем списки категориальных
# и количественных столбцов
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
```

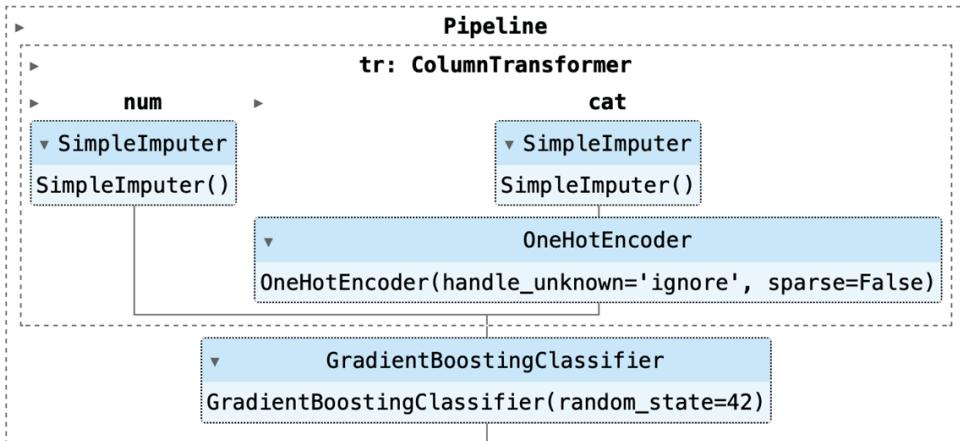
```
num_columns = X_train.select_dtypes(  
    exclude='object').columns.tolist()  
  
# создаем конвейер для количественных переменных  
num_pipe = Pipeline([('imputer', SimpleImputer())])  
  
# создаем конвейер для категориальных переменных  
cat_pipe = Pipeline([  
    ('imputer', SimpleImputer()),  
    ('ohe', OneHotEncoder(sparse=False,  
        handle_unknown='ignore'))  
])  
  
# создаем список прехэлементных кортежей, в котором  
# первый элемент кортежа - название конвейера с  
# преобразованиями для определенного типа признаков  
transformers = [('num', num_pipe, num_columns),  
    ('cat', cat_pipe, cat_columns)]  
# передаем список трансформеров в ColumnTransformer  
transformer = ColumnTransformer(transformers=transformers)  
# задаем итоговый конвейер  
pipe = Pipeline([  
    ('tr', transformer),  
    ('boost', GradientBoostingClassifier(random_state=42))  
])
```

С помощью функции `clone()` создаем клон исходного итогового конвейера. Мы создаем новый еще не обученный конвейер с теми же самыми параметрами, что у исходного конвейера. По сути, мы делаем глубокую копию конвейера без привязки к данным. Это нужно по причине того, что в ходе байесовской оптимизации гиперпараметры конвейера будут меняться, ему будут присваиваться найденные оптимальные гиперпараметры, а потом этот конвейер будет использоваться в других библиотеках оптимизации. Поэтому с помощью этой функции мы можем вернуть конвейер в исходное состояние. Это может быть полезно, когда конвейер был изменен, а заново его пересоздавать вручную не хочется.

```
# создаем клон исходного итогового конвейера  
initial_pipe = clone(pipe)
```

Давайте взглянем на исходный итоговый конвейер, раскрыв все блоки.

```
# взглянем на исходный итоговый конвейер  
pipe
```



С помощью баевской оптимизации попробуем найти оптимальную стратегию импутации для количественных признаков, оптимальную стратегию импутации для категориальных признаков и оптимальный темп обучения для модели градиентного бустинга. Пишем функцию оптимизации, задаем пространство поиска, запускаем поиск и печатаем результаты.

```

# пишем функцию, возвращающую правильность,
# усредненную по итогам перекрестной проверки
# (берем с минусом, т. к. максимизируем)
def hyperopt_objective(params):
    pipe.set_params(**params)
    return -cross_val_score(pipe, X_train, y_train, cv=5).mean()

# создаем пространство поиска
space_hyperopt = {
    'boost_learning_rate': hp.loguniform(
        'boost_learning_rate', 0.001, 0.1),
    'tr_num_imputer_strategy': hp.choice(
        'tr_num_imputer_strategy',
        ['mean', 'median', 'constant']),
    'tr_cat_imputer_strategy': hp.choice(
        'tr_cat_imputer_strategy',
        ['most_frequent', 'constant'])
}

# записываем историю
trials = Trials()

# запускаем поиск
best = fmin(fn=hyperopt_objective,
            space=space_hyperopt,
            algo=tpe.suggest,
            max_evals=20,
            trials=trials,
            rstate=np.random.default_rng(123))
  
```

```
# печатаем результаты
print("best:")
print(best)

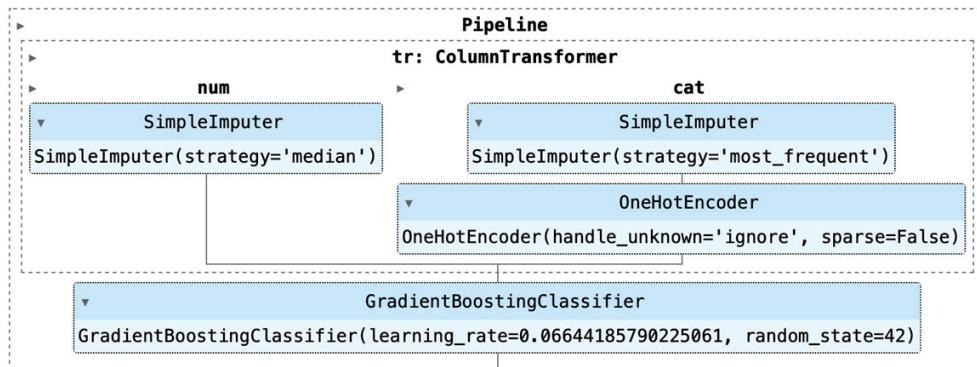
100%|██████████| 20/20 [00:53<00:00,  2.65s/trial, best loss: -0.9024978466838931]
best:
{'boost_learning_rate': 0.08228751341909112, 'transform_cat_imputer_strategy': 1,
 'transform_num_imputer_strategy': 2}
```

Можно заметить, что при использовании `hp.choice` вместо строковых значений гиперпараметров возвращаются соответствующие им индексы. Для категориальных переменных оптимальной стратегией импутации стала импутация константным значением (здесь значение 'constant' соответствует индексу 1). Для количественных переменных оптимальной стратегией импутации тоже стала импутация константным значением (здесь значение 'constant' уже соответствует индексу 2). Поэтому будьте аккуратны при интерпретации результатов.

Например, мы задали '`catboost_max_depth`': `hp.choice('catboost_max_depth', np.arange(2, 12, 2, dtype=int))`. Получили результат оптимизации `{'catboost_max_depth': 2}`. Однако речь вовсе не идет о том, что оптимальным значением глубины является значение 2. Оптимальным значением является значение с индексом 2, т. е. значение 6 из массива значений глубины `array([2, 4, 6, 8, 10])`, который мы создали с помощью функции `np.arange()`.

Взглянем на конвейер, у него будут гиперпараметры, присвоенные на последней, 20-й итерации оптимизации (итерации 19, нумерация начинается с 0).

```
# взглянем на конвейер, у него – гиперпараметры,
# присвоенные на последней, 20-й итерации
# оптимизации (итерации 19, нумерация с 0)
pipe
```



Давайте убедимся в этом, обратившись к истории испытаний: нас интересуют результаты оптимизации на итерации 19.

```
# посмотрим результаты оптимизации на итерации 19
trials.trials[19]
```

```
{'state': 2,
 'tid': 19,
```

```
'spec': None,
'result': {'loss': -0.9016365202411715, 'status': 'ok'},
'misc': {'tid': 19,
'cmd': ('domain_attachment', 'FMinIter_Domain'),
'workdir': None,
'idxs': {'boost_learning_rate': [19],
'tr_cat_imputer_strategy': [19],
'tr_num_imputer_strategy': [19]},
'vals': {'boost_learning_rate': [0.06644185790225061],
'tr_cat_imputer_strategy': [0],
'tr_num_imputer_strategy': [1]}},
'exp_key': None,
'owner': None,
'version': 0,
'book_time': datetime.datetime(2022, 6, 27, 7, 9, 48, 570000),
'refresh_time': datetime.datetime(2022, 6, 27, 7, 9, 51, 136000)}
```

Теперь вычислим правильность для тестовой выборки. С помощью функции `space_eval()` библиотеки `hyperopt` мы можем извлечь оптимальные гиперпараметры, передав в нее пространство поиска и результат функции `fmin()` (в библиотеках оптимизации `scikit-optimize` и `hyperopt` оптимальные гиперпараметры можно будет получить через атрибут `best_params` или `best_trial.params` класса-оптимизатора), передать в конвейер, обучить конвейер с этими оптимальными гиперпараметрами на обучающей выборке, получить прогнозы конвейера для тестовой выборки и сравнить их с фактическими. Поскольку эти действия мы еще будем повторять, занесем их в функцию `accuracy_tst()`.

```
# пишем функцию, вычисляющую правильность
# на тестовой выборке
def accuracy_tst(X_tr, y_tr, X_tst, y_tst,
                  opt_results, space=None,
                  optimize='optuna'):
    """
    Вычисляет правильность на тестовой выборке.

Параметры
-----
X_tr:
    Обучающий массив признаков.
y_tr:
    Обучающий массив меток.
X_tst:
    Тестовый массив признаков.
y_tst:
    Тестовый массив меток.
opt_results:
    Результаты оптимизации:
        для hyperopt - результат, возвращаемый функцией fmin(),
        для optuna - study.best_trial.params.
space:
    Пространство поиска (задается только для hyperopt).
optimize:
    Способ оптимизации, возможные значения
        'optuna' и 'hyperopt'.
    """
    if optimize == 'hyperopt':
        # Вычисляем оптимальные гиперпараметры
        best_params = space_eval(space, opt_results)
        # Создаем конвейер
        pipe = Pipeline([
            ('scaler', StandardScaler()),
            ('clf', SVC(**best_params))
        ])
        # Обучаем конвейер на обучающей выборке
        pipe.fit(X_tr, y_tr)
        # Проверяем правильность на тестовой выборке
        return pipe.score(X_tst, y_tst)
    else:
        # Вычисляем оптимальный гиперпараметр
        best_trial = optuna.study.load_study(study_name='study')
        best_params = best_trial.best_trial.params
        # Создаем конвейер
        pipe = Pipeline([
            ('scaler', StandardScaler()),
            ('clf', SVC(**best_params))
        ])
        # Обучаем конвейер на обучающей выборке
        pipe.fit(X_tr, y_tr)
        # Проверяем правильность на тестовой выборке
        return pipe.score(X_tst, y_tst)
```

```

if space is None and optimize == 'hyperopt':
    raise ValueError('Задайте пространство поиска.')

if space is not None and optimize == 'optuna':
    raise ValueError('Пространство поиска задается ' +
                     'только для hyperopt.')

if optimize == 'hyperopt':
    # записываем наилучшие значения гиперпараметров
    opt_results = space_eval(space, opt_results)

# обучаем конвейер с наилучшими значениями
# гиперпараметров на обучающей выборке
pipe.set_params(**opt_results)
pipe.fit(X_tr, y_tr)

# вычисляем прогнозы на тестовой выборке
test_predict = pipe.predict(X_tst)

# вычисляем правильность для тестовой выборки
return accuracy_score(y_tst, test_predict)

```

Итак, давайте с помощью нашей функции вычислим правильность на тестовой выборке.

```

# вычислим правильность на тестовой выборке
accuracy_tst(X_train, y_train, X_test, y_test,
              best, space=space_hyperopt,
              optimize='hyperopt')

```

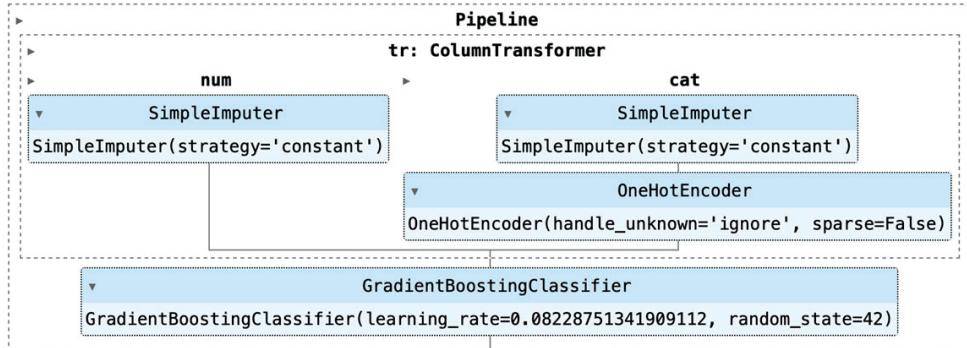
0.9031350482315113

Теперь взглянем на конвейер и убедимся, что ему присвоены найденные оптимальные значения гиперпараметров.

```

# взглянем на конвейер, ему присвоены
# оптимальные гиперпараметры
pipe

```



1.5. SCIKIT-OPTIMIZE

Библиотека scikit-optimize или skopt представляет собой еще одну реализацию баевской оптимизации гиперпараметров. Она устанавливается с помощью команды `pip install scikit-optimize`. Здесь мы будем работать с функцией `gp.minimize()` и классом `BayesSearchCV`, который является своего рода вариантом `GridSearchCV` для баевской оптимизации. В качестве суррогатной модели используются гауссовые процессы (Gaussian Processes). Начнем с функции `gp.minimize()`. Она имеет общий вид:

```
gp_minimize(func,
            dimensions,
            base_estimator=None,
            n_calls=100,
            n_random_starts=10,
            acq_func='gp_hedge',
            random_state=None,
            n_jobs=1)
```

Параметр	Предназначение
<code>func</code>	Задает минимизируемую функцию. Она должна принимать список параметров и возвращать значение целевой функции. Если у вас есть пространство поиска, где все измерения имеют имена, вы можете использовать <code>skopt.utils.use_named_args</code> в качестве декоратора вашей целевой функции, чтобы вызывать ее напрямую с поименованными аргументами
<code>dimensions</code>	Задает список размерностей пространства поиска. Каждую размерность поиска можно определить как: кортеж (<code>lower_bound, upper_bound</code>) (для размерностей <code>Real</code> или <code>Integer</code>); кортеж (<code>lower_bound, upper_bound, "prior"</code>) (для размерностей <code>Real</code>); список категорий (для размерностей <code>Categorical</code>); экземпляр объекта <code>Dimension</code> (<code>Real</code> , <code>Integer</code> или <code>Categorical</code>). Обратите внимание, что для размерностей <code>Integer</code> верхняя и нижняя границы включаются
<code>base_estimator</code>	Задает гауссовые процессы. По умолчанию используется ядро Матерна
<code>n_calls</code>	Задает количество вызовов <code>func</code> (по умолчанию 100)
<code>n_initial_points (ранее -n_random_starts)</code>	Задает количество оценок функции по случайным точкам перед ее аппроксимацией с помощью <code>base_estimator</code> (по умолчанию 10)

Параметр	Предназначение
acq_func	Задает функцию отбора: 'LCB' – для нижней доверительной границы; 'EI' – для отрицательного ожидаемого улучшения; 'PI' – для отрицательной вероятности улучшения; 'gp_hedge' – на основе вероятностей отбирает одну из вышеперечисленных функций на каждой итерации; 'EIps' – для отрицательного ожидаемого улучшения в секунду, чтобы учесть время вычисления функции (предполагается, что целевая функция возвращает два значения, первое из которых является значением целевой функции, а второе – затраченным временем в секундах); 'PIps' – для отрицательной вероятности улучшения в секунду, чтобы учесть время вычисления функции. Вид возвращаемых результатов аналогичен результатам 'EIps'
n_points	Количество точек, выбираемых для функции отбора (по умолчанию 10 000)
random_state	Задает стартовое значение генератора псевдослучайных чисел
n_jobs	Задает количество ядер процессора для распараллеливания

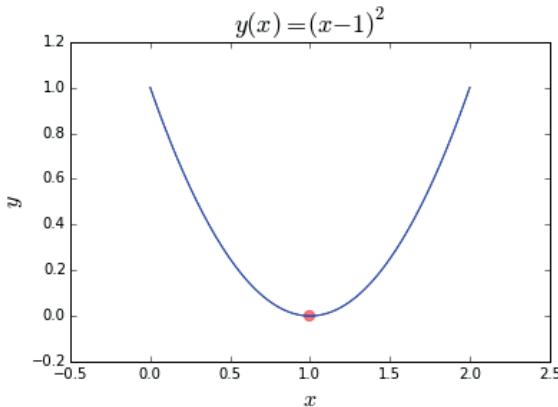
Давайте импортируем необходимые классы и функции.

```
# импортируем класс BayesSearchCV, функцию gp_minimize()
from skopt import BayesSearchCV, gp_minimize
from skopt.space import Real, Categorical, Integer
```

Теперь мы уже с помощью библиотеки skopt попробуем найти минимум квадратичной функции $y(x)=(x-1)^2$. Итак, изменим область поиска, чтобы включить предполагаемое оптимальное значение $x = 1$ плюс некоторые субоптимальные значения, лежащие по разным сторонам от него: $-2, 2$. Воспользуемся функцией `gp_minimize()`.

```
# найдем такое значение x, которое минимизирует
# квадратичную функцию y(x) = (x - 1)**2
best = gp_minimize(
    func=lambda x: (x[0] - 1) ** 2,
    dimensions=[Real(-2, 2)],
    n_calls=35,
    random_state=123)
# печатаем результат
print("наилучшее значение = %.4f" % best.fun)
print("x = %.4f" % best.x[0])

наилучшее значение = 0.0000
x = 0.9998
```



А теперь применим skopt к данным компании StateFarm. Сначала создаем пространство поиска.

```
# создаем пространство поиска
space_skopt = {'boost_learning_rate': Real(0.001, 0.1,
                                             prior='uniform'),
               'boost_max_depth': Integer(2, 8),
               'tr_num_imputer_strategy': Categorical(
                   ['mean', 'median', 'constant']),
               'tr_cat_imputer_strategy': Categorical(
                   ['most_frequent', 'constant'])}
```

Теперь клонируем исходный итоговый конвейер, создаем экземпляр класса BayesSearchCV и запускаем поиск. Класс удобен тем, что он полностью поддерживает интерфейс библиотеки scikit-learn (есть методы .fit(), .transform(), .score(), .predict(), .predict_proba() и атрибуты best_score_, best_params_).

```
from skopt import BayesSearchCV(estimator,
①
② Задает пространство поиска: словарь или список
    словарей
③ Задает количество итераций поиска (по сути количество
    отбираемых значений гиперпараметров)
④ Задает оптимизируемую метрику
⑤ Задает количество используемых ядер процессора
⑥ Задает количество значений гиперпараметров,
    отбираемых параллельно
⑦ заново обучает модель с наилучшими значениями
    гиперпараметров на всем обучающем наборе (по
    умолчанию задан)
⑧ Задает стратегию перекрестной проверки
⑨ Задает стартовое значение генератора псевдослучайных чисел
② search_spaces,
③ n_iter=50,
④ scoring=None,
⑤ n_jobs=1,
⑥ n_points=1,
⑦ refit=True,
⑧ cv=None,
⑨ random_state)
```

① Задает объект-модель машинного обучения, т.е. экземпляр класса, в котором реализован соответствующий метод машинного обучения

```
# клонируем исходный итоговый конвейер
pipe = clone(initial_pipe)
```

```
# создаем экземпляр класса BayesSearchCV
opt = BayesSearchCV(pipe,
                     space_skopt,
                     n_iter=10,
```

```
scoring='accuracy',
random_state=42,
cv=5)
# запускаем поиск
opt.fit(X_train, y_train);
```

Давайте взглянем на наилучшие значения гиперпараметров, наилучшее значение правильности (наибольшее значение правильности, усредненное по пяти проверочным блокам перекрестной проверки) и значение правильности на тестовой выборке.

```
# смотрим наилучшее значение
print("Наилучшая правильность cv: %.3f" % opt.best_score_)
# смотрим наилучшие гиперпараметры
print("Наилучшие гиперпараметры:",
      json.loads(json.dumps(opt.best_params_)))
# смотрим качество наилучшей модели на тестовой выборке
print("Правильность на тестовой выборке: %.3f" % opt.score(
    X_test, y_test))
```

Правильность cv: 0.924

Наилучшие гиперпараметры: {'boost_learning_rate': 0.07366877378057127, 'boost_max_depth': 8, 'tr_cat_imputer_strategy': 'constant', 'tr_num_imputer_strategy': 'constant'}

Правильность на тестовой выборке: 0.933

Видим, что класс BayesSearchCV позволяет нам быстро вычислить интересующую метрику на тестовой выборке.

Давайте сохраним результаты перекрестной проверки в датафрейм, избавимся от префикса param_ в столбцах, превратим датафрейм в сводную таблицу, отсортируем результаты по убыванию правильности, усредненной по проверочным блокам перекрестной проверки, и взглянем на итоговый результат.

```
# сохраняем результаты в датафрейм
results = pd.DataFrame(opt.cv_results_)
# избавляем от префикса param_
results.columns = results.columns.str.replace('param_', '')
# превращаем в сводную таблицу
tbl = results.pivot_table(values=['mean_test_score'],
                           index=['boost_learning_rate',
                                  'boost_max_depth',
                                  'tr_cat_imputer_strategy',
                                  'tr_num_imputer_strategy'])
# сортируем по убыванию правильности
tbl = tbl.sort_values(by='mean_test_score',
                      ascending=False)
tbl
```

				mean_test_score
boost_learning_rate	boost_max_depth	tr_cat_imputer_strategy	tr_num_imputer_strategy	
0.073669	8	constant	constant	0.923686
0.083901	7	constant	median	0.921102
0.054797	8	constant	median	0.919552
0.045038	8	constant	mean	0.918174
0.095592	6	most_frequent	mean	0.917485
0.062091	7	constant	median	0.916968
0.080156	5	most_frequent	mean	0.907838
0.041600	6	most_frequent	mean	0.906977
0.081427	3	most_frequent	median	0.901292
0.001359	7	most_frequent	mean	0.899742

Теперь выполним поиск с помощью функции `gp_minimize()`, а также воспользуемся декоратором `@use_named_args()`.

Мы импортируем функцию `use_named_args()` и задаем пространство поиска в немного другом формате.

```
# клонируем исходный итоговый конвейер
pipe = clone(initial_pipe)

# импортируем функцию use_named_args()
from skopt.utils import use_named_args

# задаем пространство поиска в ином формате
space_skopt2 = [
    Real(0.001, 0.1, name='boost_learning_rate',
         prior='uniform'),
    Integer(2, 8, name='boost_max_depth'),
    Categorical(['mean', 'median', 'constant'],
                name='tr_num_imputer_strategy'),
    Categorical(['most_frequent', 'constant'],
                name='tr_cat_imputer_strategy')
]
```

Пишем функцию `objective()`, возвращающую правильность, усредненную по итогам перекрестной проверки, и используем декоратор `@use_named_args`.

```
# пишем функцию, возвращающую правильность,
# усредненную по итогам перекрестной проверки,
# и используем декоратор @use_named_args
@use_named_args(space_skopt2)
def objective(**params):
    pipe.set_params(**params)
    return -np.mean(cross_val_score(pipe,
                                    X_train,
                                    y_train,
                                    cv=5,
                                    n_jobs=-1))
```

Выполняем поиск и смотрим результаты.

```
# осуществляем поиск
res = gp_minimize(objective,
                   space_skopt2,
                   n_initial_points=5,
                   n_calls=10,
                   random_state=42)

# смотрим наилучшее значение правильности
print("Наилучшее значение правильности cv: {:.4f} ".format(
    np.abs(res.fun)))
# смотрим наилучшие гиперпараметры
pd.DataFrame({'boost_learning_rate': pd.Series(res.x[0]),
               'boost_max_depth': pd.Series(res.x[1]),
               'tr_num_imputer_strategy': pd.Series(res.x[2]),
               'tr_cat_imputer_strategy': pd.Series(res.x[3])})
```

Наилучшее значение правильности: 0.9306

	boost_learning_rate	boost_max_depth	tr_num_imputer_strategy	tr_cat_imputer_strategy
0	0.099916	8	mean	constant

1.6. OPTUNA

Optuna – еще одна, относительно новая библиотека байесовской оптимизации гиперпараметров. Она поддерживает распределенные и параллельные вычисления в ходе оптимизации, отказ от бесперспективных итераций поиска – прунинг, многоцелевую (многокритериальную) оптимизацию, классы, в которых реализованы собственные схемы оптимизации для некоторых методов машинного обучения – тюнеры (например, класс `LightGBMTuner`). В отличие от `hyperopt`, библиотека optuna снабжена более четко структурированной документацией <https://optuna.readthedocs.io/en/stable/index.html>. Библиотеку можно установить с помощью команды `pip install optuna`. Давайте импортируем библиотеку optuna, из модуля `samplers` импортируем класс `TPEampler`, задающий способ семплинга гиперпараметров. Также из модуля `integration` импортируем класс `OptunaSearchCV` для выполнения более удобного поиска, из модуля `distributions` – классы `IntUniformDistribution`, `UniformDistribution`, `CategoricalDistribution`, задающие форму распределения значений гиперпараметров для поиска.

К примеру, нам надо оптимизировать квадратичную функцию $y = (x - 2)^2$. Сначала мы пишем соответствующую оптимизируемую функцию.

```
# задаем оптимизируемую функцию
def objective(trial):
    x = trial.suggest_uniform('x', -10, 10)
    return (x - 2) ** 2
```

Данная функция возвращает значение $(x - 2)^2$. Наша цель – найти такое значение x , которое минимизирует функцию `objective()`. В ходе оптимизации `optuna`, как и ранее рассмотренные библиотеки, итеративно вызывает и оценивает целевую функцию с различными значениями x . Объект `Trial` соответствует отдельному вызову целевой функции и создается каждый раз при каждом вызове функции.

API `suggest` (например, `suggest_uniform()`) вызываются внутри целевой функции для получения значений гиперпараметров на соответствующей итерации поиска. `suggest_uniform()` отбирает значения x равномерно в пределах заданного диапазона (в нашем примере в диапазоне от -10 до 10).

Затем создаем семплер – экземпляр класса `RandomSampler` для задания способа семплирования гиперпараметров.

```
# создаем экземпляр класса TPESampler
sampler = TPESampler(seed=10)
```

С помощью метода `.create_study()` создаем объект `study` – сессию оптимизации гиперпараметров, дословно «исследование». В метод `.create_study()` передаем семплер. С помощью параметра `direction` метода `.create_study()` мы можем задать минимизацию или максимизацию целевой функции. Это удобно при оптимизации метрики типа AUC-ROC или правильности, потому что нет необходимости менять знак функции перед обучением.

```
# создаем объект study (сессию оптимизации)
study = optuna.create_study(sampler=sampler)
```

Затем запускаем сессию, передав в метод `.optimize()` объекта `study` оптимизируемую функцию и количество испытаний.

```
# выполняем оптимизацию
study.optimize(objective, n_trials=100)
```

```
[I 2022-06-26 12:14:30,372] A new study created in memory with name: no-name-95bde23cfed2-45e0-8c27-225f2c052d3a
[I 2022-06-26 12:14:30,374] Trial 0 finished with value: 11.740305123732652 and
parameters: {'x': 5.426412865334919}. Best is trial 0 with value: 11.740305123732652.
[I 2022-06-26 12:14:30,374] Trial 1 finished with value: 134.21132166837336 and
parameters: {'x': -9.58496101281197}. Best is trial 0 with value: 11.740305123732652.
[I 2022-06-26 12:14:30,375] Trial 2 finished with value: 0.45288148546152784 and
parameters: {'x': 2.672964698525508}. Best is trial 2 with value: 0.45288148546152784.
[I 2022-06-26 12:14:30,376] Trial 3 finished with value: 8.857038183425992 and
parameters: {'x': 4.976077650772236}. Best is trial 2 with value: 0.45288148546152784.
. . . . .
```

```
[I 2022-06-26 12:14:30,396] Trial 29 finished with value: 0.00043176967072578425 and
parameters: {'x': 2.0207790680908886}. Best is trial 29 with value:
0.00043176967072578425.

[I 2022-06-26 12:14:30,446] Trial 94 finished with value: 27.10080266546033 and
parameters: {'x': -3.2058431272427264}. Best is trial 29 with value:
0.00043176967072578425.

[I 2022-06-26 12:14:30,446] Trial 95 finished with value: 4.405915951858926 and
parameters: {'x': -0.09902738235091313}. Best is trial 29 with value:
0.00043176967072578425.

[I 2022-06-26 12:14:30,447] Trial 96 finished with value: 56.87594954495065 and
parameters: {'x': 9.541614518453635}. Best is trial 29 with value:
0.00043176967072578425.

[I 2022-06-26 12:14:30,448] Trial 97 finished with value: 10.141189934705443 and
parameters: {'x': -1.1845235019866696}. Best is trial 29 with value:
0.00043176967072578425.

[I 2022-06-26 12:14:30,449] Trial 98 finished with value: 31.748084853080346 and
parameters: {'x': -3.634543890420976}. Best is trial 29 with value:
0.00043176967072578425.

[I 2022-06-26 12:14:30,449] Trial 99 finished with value: 2.5730093898695916 and
parameters: {'x': 0.39593971750760204}. Best is trial 29 with value:
0.00043176967072578425.
```

С помощью атрибутов `best_params`, `best_value`, `best_trial` выведем значение x , при котором было найдено наилучшее значение целевой функции, – 2.02, наилучшее значение целевой функции – 0.00043 и номер итерации, на которой было найдено наилучшее значение целевой функции, – 29.

```
# выведем значение x, при котором было найдено
# наилучшее значение целевой функции
print(study.best_params)
# выведем наилучшее значение целевой функции
print(study.best_value)
# выведем номер итерации, на которой было найдено
# наилучшее значение целевой функции
print(study.best_trial)

{'x': 2.0207790680908886}
0.00043176967072578425
FrozenTrial(number=29,      values=[0.00043176967072578425],      datetime_start=datetime.datetime(2022, 6, 26, 12, 14, 30, 396032),      datetime_complete=datetime.datetime(2022, 6, 26, 12, 14, 30, 396187),      params={'x': 2.0207790680908886},      distributions={'x': UniformDistribution(high=10.0, low=-10.0)},      user_attrs={},      system_attrs={},      intermediate_values={},      trial_id=29,      state=TrialState.COMPLETE,      value=None)
```

С помощью метода `.trials_dataframe()` можно вывести результаты в виде датафрейма pandas.

```
# выводим результаты в виде датафрейма pandas
results_df = study.trials_dataframe(
    attrs=('number', 'value', 'params'))
results_df
```

number		value	params_x
0	0	11.740305	5.426413
1	1	134.211322	-9.584961
2	2	0.452881	2.672965
3	3	8.857038	4.976078
4	4	4.120331	-0.029860
...
95	95	4.405916	-0.099027
96	96	56.875950	9.541615
97	97	10.141190	-1.184524
98	98	31.748085	-3.634544
99	99	2.573009	0.395940

100 rows × 3 columns

Мы можем задавать разные способы отбора значений для различных типов гиперпараметров. Здесь можно выделить формат пространства поиска для оптимизируемой функции, передаваемой в метод `.optimize()` объекта `study`, и формат пространства поиска, передаваемый в экземпляр класса `OptunaSearchCV`.

Формат пространства поиска для оптимизируемой функции, передаваемой

в метод `.optimize()` объекта `study`

```
# строковые значения гиперпараметра
grow_policy = trial.suggest_categorical('grow_policy', ['depthwise',
                                                       'lossguide'])

# целочисленные значения гиперпараметра
max_depth = trial.suggest_int('max_depth', 1, 9)

# непрерывные значения гиперпараметра: равномерный отбор
gamma = trial.suggest_uniform('gamma', 1e-2, 1.0)

# непрерывные значения гиперпараметра: логравномерный отбор
eta = trial.suggest_loguniform('eta', 1e-2, 1.0)
```

Формат пространства поиска, передаваемый в экземпляр класса `OptunaSearchCV`

```
# строковые значения гиперпараметра
'grow_policy': CategoricalDistribution(['depthwise',
                                         'lossguide'])

# целочисленные значения гиперпараметра
'max_depth': IntUniformDistribution(1, 9)

# непрерывные значения гиперпараметра: равномерный отбор
'gamma': UniformDistribution(1e-2, 1.0)

# непрерывные значения гиперпараметра: логравномерный отбор
'eta': LogUniformDistribution(1e-2, 1.0)
```

А теперь применим optuna к данным компании StateFarm.

Мы создадим экземпляр класса KFold, задающий стратегию перекрестной проверки, и напишем функцию, которую будем оптимизировать.

```
# создаем экземпляр класса KFold
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# пишем функцию, которую будем оптимизировать
def objective(trial):
    # задаем пространство поиска
    params = {
        'boost_learning_rate': trial.suggest_uniform(
            'boost_learning_rate', 0.001, 0.1),
        'boost_max_depth': trial.suggest_int(
            'boost_max_depth', 2, 8),
        'tr_num_imputer_strategy': trial.suggest_categorical(
            'tr_num_imputer_strategy', ['mean', 'median', 'constant']),
        'tr_cat_imputer_strategy': trial.suggest_categorical(
            'tr_cat_imputer_strategy', ['most_frequent', 'constant']))
    }
    pipe.set_params(**params)
    return np.mean(cross_val_score(pipe, X_train, y_train, cv=kfold))
```

С помощью метода .create_study() создаем объект study – сессию оптимизации гиперпараметров, дословно «исследование». В метод .create_study() передаем семплер. С помощью параметра direction метода .create_study() задаем максимизацию целевой функции. С помощью метода disable_default_handler() отключаем логгирование. Затем запускаем сессию, передав в метод .optimize() объекта study оптимизируемую функцию и количество испытаний.

```
# создаем задачу оптимизации
study = optuna.create_study(sampler=sampler,
                             direction='maximize')
# отключаем вывод результатов оптимизации
# в режиме реального времени
optuna.logging.disable_default_handler()
# выполняем оптимизацию
study.optimize(objective, n_trials=10)
```

Давайте напечатаем количество итераций, наилучшую правильность по итогам перекрестной проверки и наилучшие гиперпараметры.

```
# печатаем наилучшую правильность
best = study.best_trial
print("Наилучшая правильность cv: %.3f" % best.value)
print("Наилучшие гиперпараметры: ")
for key, value in best.params.items():
    print(" {}: {}".format(key, value))
```

```
Наилучшая правильность cv: 0.925
Наилучшие гиперпараметры:
boost_learning_rate: 0.06885062201841192
boost_max_depth: 8
tr_num_imputer_strategy: constant
tr_cat_imputer_strategy: constant
```

Теперь вычислим правильность для тестовой выборки с помощью нашей функции.

```
# вычислим правильность на тестовой выборке
accuracy_tst(X_train, y_train, X_test, y_test,
               best_params, space=None,
               optimize='optuna')
```

0.9344855305466238

Теперь для оптимизации воспользуемся классом OptunaSearchCV. Класс удобен тем, что он полностью поддерживает интерфейс библиотеки scikit-learn (есть методы .fit(), .transform(), .score(), .predict(), .predict_proba() и атрибуты best_score_, best_params_).

<pre>from optuna.integration import OptunaSearchCV(estimator, param_distributions, n_trials=50, scoring=None, n_jobs=1, subsample=1.0, refit=True, cv=None, timeout=None, enable_pruning=False, random_state)</pre>	<p>1 Задает объект-модель машинного обучения, т. е. экземпляр класса, в котором реализован соответствующий метод машинного обучения</p> <p>2 Задает пространство поиска: словарь или список словарей</p> <p>3 Задает количество итераций поиска (по сути, количество отбираемых значений гиперпараметров)</p> <p>4 Задает оптимизируемую метрику</p> <p>5 Задает количество используемых ядер процессора</p> <p>6 Задает подвыборку наблюдений, используемую для оптимизации. Если задано целое число, извлекает subsample наблюдений, если задано число с плавающей точкой, извлекает subsample * X.shape[0] наблюдений</p> <p>7 заново обучает модель с наилучшими значениями гиперпараметров на всем обучающем наборе (по умолчанию задан)</p> <p>8 Задает стратегию перекрестной проверки</p> <p>9 Задает время, затрачиваемое на оптимизацию</p> <p>10 Задает прунинг. Прунинг выполняется только в том случае, если модель машинного обучения поддерживает partial_fit</p> <p>11 Задает стартовое значение генератора псевдослучайных чисел</p>
---	--

Задаем пространство поиска в немного ином формате, создаем экземпляр класса OptunaSearchCV, выполняем оптимизацию и печатаем результаты.

```
# задаем пространство поиска в немного ином формате
param_distributions = {
    'boost_learning_rate': UniformDistribution(0.001, 0.1),
    'boost_max_depth': IntUniformDistribution(2, 8),
    'tr_num_imputer_strategy': CategoricalDistribution(
        ['mean', 'median', 'constant']),
    'tr_cat_imputer_strategy': CategoricalDistribution(
        ['most_frequent', 'constant'])
}

# клонируем исходный итоговый конвейер
pipe = clone(initial_pipe)

# создаем экземпляр класса OptunaSearchCV
optuna_search = OptunaSearchCV(
    pipe,
    param_distributions,
    scoring='accuracy',
    random_state=42,
    n_trials=10,
    cv=kfold)
```

```
# выполняем оптимизацию
optuna_search.fit(X_train, y_train)

# печатаем наилучшие значения гиперпараметров
print("Наилучшие гиперпараметры:\n",
      optuna_search.best_params_)

# печатаем наилучшее значение правильности
print("Наилучшая правильность cv: %.3f" % optuna_search.best_score_)

# печатаем правильность для тестовой выборки
test_score = optuna_search.score(X_test, y_test)
print("Правильность на тестовой выборке: %.3f" % test_score)

Наилучшие гиперпараметры:
{'boost_learning_rate': 0.0925232413252804,
 'boost_max_depth': 8,
 'tr_num_imputer_strategy': 'constant',
 'tr_cat_imputer_strategy': 'most_frequent'}
Наилучшая правильность cv: 0.931
Правильность на тестовой выборке: 0.940
```

А теперь модифицируем конвейер для работы с классом `LGBMClassifier` библиотеки `lightgbm`, пишем оптимизируемую функцию, выполняем оптимизацию и печатаем результаты.

```
# модифицируем конвейер для работы с классом LGBMClassifier
lgbm_pipe = Pipeline([
    ('tr', transformer),
    ('lgboost', LGBMClassifier(random_state=42, n_estimators=200))
])

# пишем оптимизируемую функцию
def lgbm_objective(trial):
    params = {
        'lgboost_learning_rate': trial.suggest_uniform(
            'lgboost_learning_rate', 0.001, 0.1),
        'lgboost_max_depth': trial.suggest_int(
            'lgboost_max_depth', 2, 8),
        'tr_num_imputer_strategy': trial.suggest_categorical(
            'tr_num_imputer_strategy', ['mean', 'median', 'constant']),
        'tr_cat_imputer_strategy': trial.suggest_categorical(
            'tr_cat_imputer_strategy', ['most_frequent', 'constant'])
    }
    lgbm_pipe.set_params(**params)
    return np.mean(cross_val_score(lgbm_pipe, X_train, y_train, cv=5))

# создаем объект study (сессию оптимизации)
study = optuna.create_study(sampler=sampler,
                             direction='maximize')

# выполняем оптимизацию
study.optimize(lgbm_objective, n_trials=10)

# печатаем результаты
print("Наилучшие гиперпараметры:\n", study.best_params_)
print("Наилучшая правильность cv: %.3f" % study.best_value)
```

Наилучшие гиперпараметры:

```
{'lgboost__learning_rate': 0.05264306808696978,
 'lgboost__max_depth': 8,
 'tr__num__imputer__strategy': 'mean',
 'tr__cat__imputer__strategy': 'constant'}
```

Наилучшая правильность cv: 0.920

Теперь посмотрим в работе класс-тюнер LightGBMTunerCV.

Нам нужно импортировать подмодуль `lightgbm` модуля `integration` и функцию обратного вызова `early_stopping()`.

```
# импортируем подмодуль lightgbm модуля integration,
# функцию early_stopping
import optuna.integration.lightgbm as lgb
from lightgbm import early_stopping
```

Функция `early_stopping()` активирует раннюю остановку. Модель будет обуваться до тех пор, пока метрика на проверочном наборе не улучшится как минимум на `min_delta`. Для продолжения обучения метрика на проверочном наборе должна улучшаться по крайней мере каждые `stopping_rounds`. Разберем, как работает перекрестная проверка. Допустим, мы обучаем градиентный бустинг из 1000 деревьев, т. е. у нас будет 1000 итераций. Задаем `stopping_rounds` равным 30.

```
[694]    valid_0's auc: 0.911122
[695]    valid_0's auc: 0.9111
[696]    valid_0's auc: 0.910962
[697]    valid_0's auc: 0.910948
[698]    valid_0's auc: 0.910935
[699]    valid_0's auc: 0.910833
[700]    valid_0's auc: 0.910822
[701]    valid_0's auc: 0.910684
[702]    valid_0's auc: 0.910806
[703]    valid_0's auc: 0.910747
[704]    valid_0's auc: 0.910675
[705]    valid_0's auc: 0.910675
[706]    valid_0's auc: 0.910731
[707]    valid_0's auc: 0.910722
[708]    valid_0's auc: 0.910869
[709]    valid_0's auc: 0.910776
[710]    valid_0's auc: 0.910785
[711]    valid_0's auc: 0.910727
[712]    valid_0's auc: 0.910618
[713]    valid_0's auc: 0.91072
[714]    valid_0's auc: 0.91086
[715]    valid_0's auc: 0.910883
[716]    valid_0's auc: 0.910871
[717]    valid_0's auc: 0.91074
[718]    valid_0's auc: 0.910765
[719]    valid_0's auc: 0.910815
[720]    valid_0's auc: 0.910821
[721]    valid_0's auc: 0.910826
[722]    valid_0's auc: 0.910839
[723]    valid_0's auc: 0.910908
[724]    valid_0's auc: 0.910944
Early stopping, best iteration is:
[694]    valid_0's auc: 0.911122
```

Рис. 4 Ранняя остановка

Видим, что хоть мы и задали 1000 итераций, обучение заняло 725 итераций (нумерация с 0). На итерации 695 оценка AUC-ROC перестала увеличиваться, стала ниже (0,9111) по сравнению с оценкой AUC-ROC на итерации 694 (0,911122). Тогда мы в течение 30 последующих итераций смотрим, не поднялась ли наша оценка AUC-ROC выше оценки AUC-ROC на итерации 694 – последней итерации, когда было достигнуто наибольшее значение AUC-ROC (0,911122). Если не поднялась, то останавливаемся и возвращаем в качестве наилучшей итерацию 694, на которой было достигнуто оптимальное значение метрики.

Категориальные признаки в обучающем массиве признаков заменяем на количественные. Для этого применяем Label Encoding, в ходе которого категориям переменной в лексикографическом порядке мы присваиваем целочисленные значения, начиная с 0 (например, категории 'кошка', 'мышь', 'собака' станут 0, 1, 2). Строковые метки зависимой переменной меняем на целочисленные. Наконец, на основе подготовленных массива признаков и массива меток мы формируем объект `Dataset`. `Dataset` представляет собой внутреннюю структуру данных `lightgbm` для хранения признаков и меток.

```
# выполняем Label Encoding
features = X_train.copy()
cols = list(features.columns[features.dtypes == 'object'])
for c in cols:
    features[c] = features[c].astype('category').cat.codes
# строковые метки зависимой переменной меняем на целочисленные
labels = np.where(y_train == 'Yes', 1, 0)
# формируем lgb.Dataset
dtrain = lgb.Dataset(data=features, label=labels)
```

Теперь создаем словарь параметров и гиперпараметров, создаем экземпляр класса `LightGBMTunerCV`, передав в него словарь параметров и гиперпараметров, объект `Dataset`, информацию о количестве итераций бустинга, стратегию перекрестной проверки, функцию ранней остановки в качестве callback-функции (используем 30 раундов для ранней остановки), и, наконец, запускаем оптимизацию.

```
# создаем словарь параметров и гиперпараметров
params = {
    'objective': 'binary',
    'metric': 'auc',
    'verbosity': -1,
    'boosting_type': 'gbdt'
}

# создаем экземпляр класса LightGBMTunerCV
tuner = lgb.LightGBMTunerCV(
    params,
    dtrain,
    num_boost_round=300,
    folds=KFold(n_splits=3, shuffle=True, random_state=42),
    callbacks=[early_stopping(30)],
)

# запускаем оптимизацию
tuner.run()
```

Теперь печатаем результаты оптимизации, выполненной с помощью класса LightGBMTunerCV. Класс LightGBMTunerCV пошагово оптимизирует гиперпараметры по заранее заданной схеме: lambda_l1, lambda_l2, num_leaves, feature_fraction, bagging_fraction, bagging_freq и min_child_samples.

```
# печатаем результаты
print("Наилучшее значение AUC-ROC cv:", tuner.best_score)
best_params = tuner.best_params
print("Наилучшие параметры:", best_params)
print(" Параметры: ")
for key, value in best_params.items():
    print("    {}: {}".format(key, value))

Наилучшее значение AUC-ROC cv: 0.903489630307949
Наилучшие параметры: {'objective': 'binary', 'metric': 'auc', 'verbosity': -1, 'boosting_type': 'gbdt', 'feature_pre_filter': False, 'lambda_l1': 1.1819270868465547e-05, 'lambda_l2': 1.0939299129330412e-08, 'num_leaves': 30, 'feature_fraction': 0.8999999999999999, 'bagging_fraction': 1.0, 'bagging_freq': 0, 'min_child_samples': 20}
Параметры:
    objective: binary
    metric: auc
    verbosity: -1
    boosting_type: gbdt
    feature_pre_filter: False
    lambda_l1: 1.1819270868465547e-05
    lambda_l2: 1.0939299129330412e-08
    num_leaves: 30
    feature_fraction: 0.8999999999999999
    bagging_fraction: 1.0
    bagging_freq: 0
    min_child_samples: 20
```

2. Docker

2.1. Введение

Docker – это платформа, которая предназначена для разработки, развертывания и запуска приложений в контейнерах. Упрощенно говоря, Docker – это виртуальная машина, на которой уже установлено окружение, которое потребуется для проекта.

Возникает вопрос: зачем нужен Docker в DS-проектах?

Скорость. Docker позволяет быстро создать среду разработки для data science проекта. Вам не потребуется отдельно устанавливать python, anaconda или библиотеки для data science. Все, что нужно, – найти подходящий образ Docker и запустить контейнер.

Повторяемость. При запуске Docker можно указать версии библиотек, используемых в проекте. Это гарантирует, что ваша модель покажет те же результаты на другом компьютере или спустя какое-то время.

Мобильность. Контейнер Docker можно использовать на локальном компьютере, на компьютере коллеги или на сервере поставщика облачных услуг.

В Docker существует два основных понятия – образ (image) и контейнер (container).

Образ (image) – это шаблон, который используется для создания контейнера. В образе контейнера Docker указывается операционная система и библиотеки для анализа данных. Образы можно собирать самостоятельно или использовать уже готовые, которые хранятся в Docker Hub.

Контейнер (container) – это рабочий экземпляр образа. Контейнеры состоят из слоев. Каждый слой, кроме последнего, устанавливает ПО или библиотеку.

Файл Dockerfile – это текстовый файл, который сообщает Docker о том, какие слои и в каком порядке нужно добавить в контейнер. Этот файл содержит описание базового образа и дополнительных слоев (например, библиотек NumPy, Pandas, Scikit-learn и пр.). Среди популярных официальных базовых образов для data science можно отметить:

- jupyter/base-notebook – образ содержит только Jupyter Notebook, нет никаких предустановленных библиотек;
- jupyter/scipy-notebook – образ уже содержит Jupyter Notebook, библиотеки pandas, matplotlib, scipy, seaborn, scikit-learn, statsmodel, beautifulsoup и пр.;
- jupyter/tensorflow-notebook – все из образа jupyter/scipy-notebook, а также дополнительно библиотеки tensorflow, keras.

Более подробно изучить образы контейнеров Docker можно тут: <https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html>.

Репозиторий контейнеров (Docker Hub). Это облачное хранилище, где хранятся образы, собранные разработчиками или пользователями Docker. Самым крупным хранилищем является репозиторий *Docker Hub*. Он используется при работе с Docker по умолчанию.

Платформа Docker (Docker Platform) – это программа, которая даёт возможность упаковывать приложения в контейнеры и запускать их на серверах.

Тома Docker (Docker Volumes) – хранилище информации, используемое в контейнерах.

Docker Compose – это инструмент, который упрощает развёртывание приложений, для работы которых требуется несколько контейнеров Docker. Docker Compose позволяет выполнять команды, описываемые в файле docker-compose.yml.

Docker управляет командной строкой в терминале. Все команды начинаются с ключевого слова docker, за которым идёт пробел, затем указание на то, на что именно будет направлена команда (`container`, `image`, `volume` и пр.), потом ещё один пробел, а далее следует сама команда. Например, именно так построена такая команда: `docker container stop container_id`.

Общая схема команд для управления контейнерами выглядит так: `docker my_command my_container`, где:

- `my_container` – имя или id контейнера;
- `my_command` – команда.

Перечислим основные команды для управления контейнерами:

- `create` – создание контейнера из образа;
- `start` – запуск существующего контейнера;
- `run` – создание контейнера и его запуск;
- `ls` – вывод списка работающих контейнеров;
- `inspect` – вывод подробной информации о контейнере;
- `logs` – вывод логов;
- `stop` – корректная остановка контейнера;
- `kill` – быстрая остановка контейнера;
- `rm` – удаление остановленного контейнера.

2.2. ЗАПУСК КОНТЕЙНЕРА DOCKER

Устанавливаем Docker Desktop <https://www.docker.com/products/docker-desktop> и запускаем.

Создаем папку с рабочим проектом локально на компьютере. Допустим, путь к папке выглядит так: `/Users/user/Documents/ds_project`. В моем случае путь к папке выглядит так: `/Users/artemgruzdev/Documents/ds_project`.

Папка `ds_project` содержит папку `Data`, в которой находятся исторический набор `contest_train.csv` и набор новых данных `contest_test.csv`, тетрадку `MegaFon Accelerator (2-е место).ipynb` с решением задачи MegaFon Accelerator с Kaggle, дающим 2-е место в приватном лидерборде <https://www.kaggle.com/c/mf-accelerator>, и текстовый файл `requirements.txt` с описанием версий необходимых библиотек. Файл `requirements.txt` содержит следующие записи:

```
catboost==1.0.6
pandas==1.4.3
numpy==1.21.5
scikit-learn==1.0.2
```

Открываем любой терминал.

После установки Docker запустите в терминале команду `docker`, тем самым проверяем, что Docker установлен правильно.

Выведем список созданных образов командой `docker image ls` и список запущенных контейнеров командой `docker ps -as`. Скорее всего, при первом запуске у вас не будет ни образов, ни контейнеров.

```
(base) MacBook-Pro-Artem:~ artemgruzdev$ docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
(base) MacBook-Pro-Artem:~ artemgruzdev$ docker ps -as
CONTAINER ID        IMAGE      COMMAND       CREATED        STATUS        PORTS     NAMES
(base) MacBook-Pro-Artem:~ artemgruzdev$
```

2.3. Создание контейнера Docker с помощью Dockerfile

Для создания контейнера используется файл Dockerfile (без расширения). Dockerfile сообщает Docker о том, как собирать образы, на основе которых создаются контейнеры.

1. В терминале делаем папку с рабочим проектом текущей с помощью команды `cd /Users/artemgruzdev/Documents/ds_project`.

```
(base) MacBook-Pro-Artem:~ artemgruzdev$ cd /Users/artemgruzdev/Documents/ds_project
(base) MacBook-Pro-Artem:ds_project artemgruzdev$
```

2. В рабочей папке создаем файл Dockerfile, в нем мы потом укажем инструкцию, по которой Docker подготовит окружение для проекта. Итак, в терминале запускаем команду `cat > Dockerfile` и нажимаем **Ctrl+D**.

```
(base) MacBook-Pro-Artem:ds_project artemgruzdev$ cat > Dockerfile
(base) MacBook-Pro-Artem:ds_project artemgruzdev$
```

Открываем файл Dockerfile (например, можно открыть с помощью программыTextEdit) и теперь для сборки образа контейнера для проекта вносим в Dockerfile следующий текст-инструкцию:

```
# задаем базовый образ, с которого начинаем сборку
FROM jupyter/scipy-notebook:latest

# задаем рабочую директорию для контейнера
WORKDIR /mnt/jupyter

# копируем все файлы из корня проекта
# в рабочую директорию
COPY . /mnt/jupyter

# выполняем команду - устанавливаем библиотеки,
# указанные в файле requirements.txt
RUN pip install -r requirements.txt

# определяем команду, которая запускается
# в начале работы образа
CMD [ "/bin/bash" ]
```

```

Dockerfile — Изменено
# задаем базовый образ, с которого начинаем сборку
FROM jupyter/scipy-notebook:latest

# задаем рабочую директорию для контейнера
WORKDIR /mnt/jupyter

# копируем все файлы из корня проекта
# в рабочую директорию
COPY . /mnt/jupyter/

# выполняем команду – устанавливаем библиотеки,
# указанные в файле requirements.txt
RUN pip install -r requirements.txt

# определяем команду, которая запускается
# в начале работы образа
CMD [ "/bin/bash" ]

```

Рис. 5 Заполнение Dockerfile

3. Если файл *requirements.txt* не создан (в нашем случае создан, шаг можно пропустить), создаем его в папке с рабочим проектом. В нем указываем нужные библиотеки и их версии. Это упростит работу:

- не потребуется устанавливать каждую библиотеку отдельно. При запуске контейнера Docker сразу вставит нужные библиотеки;
- перечисление версий используемых библиотек гарантирует, что модель покажет одинаковые результаты на другом компьютере или спустя какое-то время.

В нашем случае содержимое файла *requirements.txt* будет таким:

```

catboost==1.0.6
pandas==1.4.3
numpy==1.21.5
scikit-learn==1.0.2

```

4. Кроме того, в той же рабочей папке заведем файл *.dockerignore*, в котором можно перечислить файлы по маске, которые не нужно будет копировать из текущей папки в образ Docker. Например, такие как файлы *README**, *Dockerfile** и т. п. Для этого в терминале запускаем команду ***cat > .dockerignore***, вводим *Dockerfile**, *README** и нажимаем ***Ctrl+D***.

```

(base) MacBook-Pro-Artem:ds_project artemgruzdev$ cat > .dockerignore
Dockerfile*
README*
(base) MacBook-Pro-Artem:ds_project artemgruzdev$ █

```

Файл *.dockerignore* является скрытым и может не отображаться, для отображения можно воспользоваться комбинацией клавиш ***Command+Shift +.(точка)***.

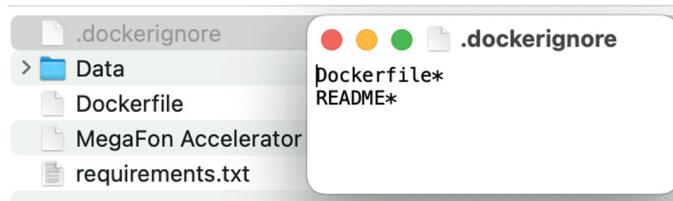


Рис. 6 Файл .dockerignore

5. После создания файла Dockerfile выполним команду `docker build -t megafon_2 ..`, где
- ◆ `-t megafon_2` – задает тег образа (в данном случае новый образ будет называться `megafon_2`);
 - ◆ `..` – указывает на путь, где лежит Dockerfile. В данном случае Dockerfile находится в текущей папке.

```
(base) MacBook-Pro-Artem:ds_project artemgruzdev$ docker build -t megafon_2 .
[+] Building 72.8s (9/9) FINISHED
[+] => [internal] load build definition from Dockerfile
[+] => => transferring dockerfile: 984B
[+] => [internal] load .dockerignore
[+] => [internal] load context: 107B
[+] => [internal] load metadata for docker.io/jupyter/scipy-notebook:latest
[+] => [internal] load build context
[+] => transferring context: 9.26kB
=> [1/4] FROM docker.io/jupyter/scipy-notebook:latest@sha256:2a4c0fe03442561b0d59298e2188dfac1abbbb00dab96a1f6e4e56c41c13444
=> CACHED [2/4] WORKDIR /mnt/jupyter
=> [3/4] COPY . /mnt/jupyter/
=> [4/4] RUN pip install -r requirements.txt
=> exporting to image
=> exporting layers
=> => writing image sha256:e06fd15939685cebc5f5808448629da21d073c73498b99d1fbf199fa258255bd
=> => naming to docker.io/library/megafon_2
(base) MacBook-Pro-Artem:ds_project artemgruzdev$
```

6. После выполнения команды создается образ `megafon_2`.

Чтобы проверить, что образ создался корректно, выполним команду `docker images`. В результате в списке образов видим образ `megafon_2`:

```
(base) MacBook-Pro-Artem:ds_project artemgruzdev$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
megafon_2        latest   346a294a3a96   About a minute ago   3.64GB
(base) MacBook-Pro-Artem:ds_project artemgruzdev$
```

Кроме того, убедиться в создании образа можно с помощью пункта **Dashboard** приложения Docker Desktop.

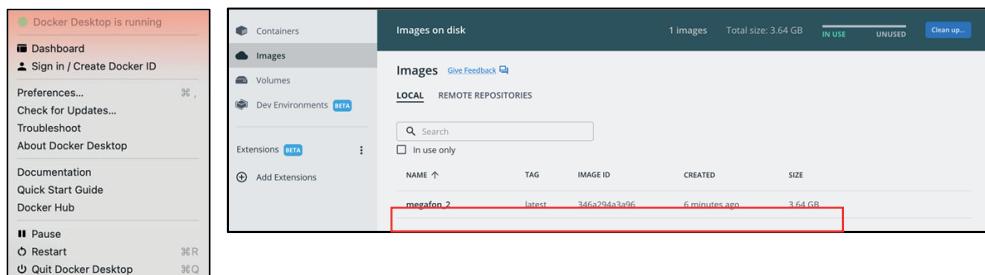


Рис. 7 Работа с приложением Docker Desktop: образ Docker

7. Теперь из созданного образа нужно запустить контейнер

```
docker run -it --name meg2 -p 8888:8888 megafon_2 /bin/bash -c "jupyter notebook --notebook-dir=/mnt/jupyter/ --ip='*' --port=8888 --no-browser",
```

где

- docker run – команда, создающая и запускающая контейнер из образа;
- -p 8888:8888 – указывает, какой порт будет открыт у контейнера;
- --name – задает имя контейнера (в данном примере контейнер называется meg2);
- Megafon_2 – образ, из которого запускается контейнер;
- /bin/bash -c "jupyter notebook --notebook-dir=/mnt/jupyter/ --ip='*' --port=8888 --no-browser" – команда, которая запускается внутри контейнера.

```
[root@Megafon-Pro-Artem:~/Projects/PycharmProjects/docker]# docker run -it --name meg2 -p 8888:8888 megafon_2 /bin/bash -c "jupyter notebook --notebook-dir=/mnt/jupyter/ --ip='*' --port=8888 --no-browser"
[1] 18:05:17.808 NotebookApp[1] Writing notebook server cookie secret to '/home/oivan/.local/share/jupyter/runtime/notebook_cookie_secret'
[1] 18:05:17.847 NotebookApp[1] WARNING: The notebook server is listening on all IP addresses and not using encryption. This is not recommended.
[1] 18:05:18.383 NotebookApp[1] JupyterLab extension loaded from /opt/conda/lib/python3.8/site-packages/jupyterlab
[1] 18:05:18.383 NotebookApp[1] JupyterLab application directory in /opt/conda/share/jupyter/lab
[1] 18:05:18.383 NotebookApp[1] JupyterLab static files directory: /mnt/jupyter
[1] 18:05:18.383 NotebookApp[1] Jupyter Notebook 6.1.4 is running at:
[1] 18:05:18.383 NotebookApp[1]   file:///home/oivan/.local/share/jupyter/runtime/nserver-6-open.html
[1] 18:05:18.383 NotebookApp[1]   http://c267c455718f:8888/?token=bd086334f7516bee8629efa384023844e3ecede7206ff832
[1] 18:05:18.383 NotebookApp[1]   or http://127.0.0.1:8888/?token=bd086334f7516bee8629efa384023844e3ecede7206ff832
[1] 18:05:18.383 NotebookApp[1] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[1] 18:05:18.399 NotebookApp[1]

To access the notebook, open this file in a browser:
  file:///home/oivan/.local/share/jupyter/runtime/nserver-6-open.html
Or copy and paste one of these URLs:
  http://c267c455718f:8888/?token=bd086334f7516bee8629efa384023844e3ecede7206ff832
  http://127.0.0.1:8888/?token=bd086334f7516bee8629efa384023844e3ecede7206ff832
```

8. После того как контейнер запустился копируем из консольной строки адрес доступа к jupyter notebook вместе с ТОКЕНОМ, например:

<http://127.0.0.1:8888/?token=bd086334f7516bee8629efa384023844e3ecede7206ff832>

Вставляем его в адресную строку браузера и запускаем.



Сгенерированный токен записываем отдельно, его можно будет в дальнейшем использовать как пароль доступа к jupyter notebook в данном контейнере.

Если при отработке решения в тетрадке возникают ошибки о нехватке ресурсов, попробуйте увеличить ресурсы, выбрав пункт **Preferences** приложения Docker Desktop и затем **Resources**.

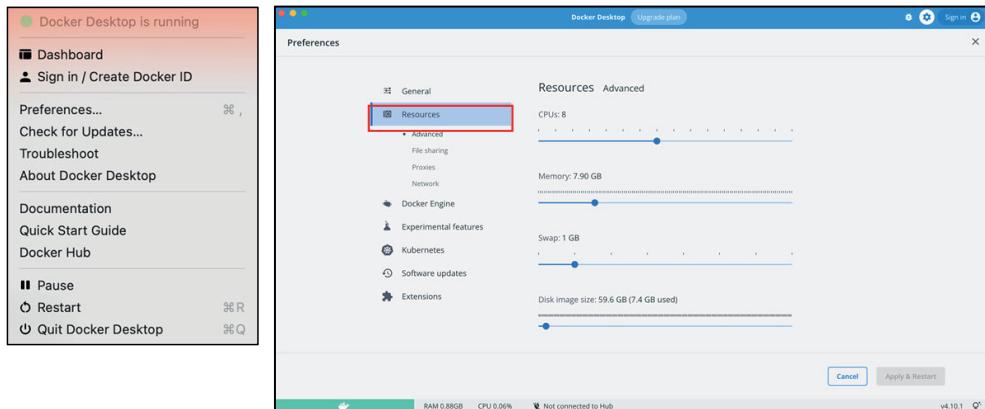


Рис. 8 Работа с приложением Docker Desktop: настройка ресурсов

3. Библиотека H2O

H2O – платформа с открытым исходным кодом, которая предназначена для быстрого, распределенного и масштабируемого машинного обучения. В США она является одной из самых популярных платформ, использующихся при работе с большими данными. Ценность платформы заключается в том, что в ней реализованы:

- развернутый вывод о модели, получаемый с помощью одной строки программного кода;
- широкое семейство обобщенных линейных моделей (линейная регрессия, бинарная логистическая регрессия, многоклассовая логистическая регрессия, гамма-регрессия, пуассоновская регрессия и прочие);
- возможность тонкой настройки регуляризации линейных моделей;
- случайный лес и градиентный бустинг, обрабатывающие категориальные признаки «как есть» и в виде количественных признаков (встроенные способы кодировки);
- изолирующий лес;
- стекинг;
- AutoML.

H2O предлагает пакет Python с одноименным названием h2o, который позволяет использовать в среде Python возможности платформы H2O.

H2O написана на Java, поэтому для работы пакета необходимо зайти на страницу загрузок Java <https://www.oracle.com/java/technologies/downloads/> и установить Java SE Development Kit (рекомендуется устанавливать версию на единицу меньше текущей).

3.1. Установка пакета h2o для Python

Убеждаемся, что инсталляция Java SE Development Kit выполнена.

В Python устанавливаем зависимости:

```
pip install requests
pip install tabulate
pip install "colorama>=0.3.8"
pip install future
```

Затем устанавливаем пакет h2o: `pip install h2o`.

Для удаления установленного пакета h2o можно воспользоваться командой `pip uninstall h2o`.

3.2. Запуск кластера H2O

Мы импортируем библиотеку h2o, с помощью функции `h2o.init()` запускаем кластер H2O. Параметр `nthreads` задает количество ядер процессора, используемое при вычислениях (по умолчанию используются все ядра процессора), параметр `max_mem_size` задает максимальный объем памяти для вычислений.

```
# импортируем необходимую библиотеку
import h2o

# запускаем кластер H2O
h2o.init(nthreads=-1, max_mem_size=8)
```

H2O_cluster_uptime:	01 secs
H2O_cluster_timezone:	Europe/Moscow
H2O_data_parsing_timezone:	UTC
H2O_cluster_version:	3.36.1.2
H2O_cluster_version_age:	1 month and 1 day
H2O_cluster_name:	H2O_from_python_artemgruzdev_23aan5
H2O_cluster_total_nodes:	1
H2O_cluster_free_memory:	8 Gb
H2O_cluster_total_cores:	16
H2O_cluster_allowed_cores:	16
H2O_cluster_status:	locked, healthy
H2O_connection_url:	http://127.0.0.1:54323
H2O_connection_proxy:	{"http": null, "https": null}
H2O_internal_security:	False
Python_version:	3.9.12 final

3.3. ПРЕОБРАЗОВАНИЕ ДАННЫХ ВО ФРЕЙМЫ H2O

Давайте прочитаем расширенные данные компании StateFarm в датафрейм. Если данные уже прочитаны как датафреймы pandas, их нужно преобразовать во фреймы H2O – специальную структуру данных, которая используется платформой H2O при вычислениях. Это можно сделать с помощью функции `h2o.H2OFrame()`. Обратите внимание, что всю предварительную подготовку данных лучше осуществить перед преобразованием во фреймы, используя возможности Python. Стандартизацию для линейных моделей можно не делать, т. к. при построении этих моделей в H2O стандартизация выполняется автоматически по умолчанию.

```
# импортируем библиотеки pandas и numpy
import pandas as pd
import numpy as np
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm_for_H2O.csv', sep=';')
data.head(3)
```

Если ранее использовалась функция `train_test_split()`, выполнялась предподготовка на обучающей и тестовой выборках, перед преобразованием во фреймы нужно сконкатенировать обучающий массив признаков с обучающим массивом меток, а тестовый массив признаков – с тестовым массивом меток, поскольку в H2O используется принцип: задаем фрейм и указываем, какая переменная – зависимая, а какие переменные – признаки.

```
# импортируем функцию train_test_split(), с помощью
# которой разделяем данные на обучающие и тестовые
from sklearn.model_selection import train_test_split
```

```

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)
. . . операции предварительной подготовки . . .
# конкatenируем обучающие массив признаков и массив меток
train = pd.concat([X_train, y_train], axis=1)
# конкatenируем тестовые массив признаков и массив меток
test = pd.concat([X_test, y_test], axis=1)

# преобразовываем датафреймы pandas во фреймы h2o
tr = h2o.H2OFrame(train)
tst = h2o.H2OFrame(test)

```

Ниже показан процесс получения фреймов H2O напрямую.

```

# загружаем данные в формате фрейма
data = h2o.upload_file(path='Data/StateFarm_for_H2O.csv')

# разбиваем фрейм на обучающий и тестовый
tr, tst = data.split_frame(ratios=[.7], seed=42)

```

3.4. ЗНАКОМСТВО С СОДЕРЖИМЫМ ФРЕЙМА

С помощью метода `.describe()` можно взглянуть на содержимое фрейма:

```
# смотрим содержимое фрейма
tr.describe()
```

Rows:5799 ← Информация о количестве строк (наблюдений) и
Cols:7 количестве столбцов (переменных)

	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
type	int	int	int	int	int	int	int
mins	0.0	61.0	0.0	0.0	0.0	1.0	0.0
mean	37865.85031902055	93.31401965856195	15.117261596827044	48.28694602517683	0.3821348508363507	2.990170719089499	0.09949991377823762
maxs	99961.0	298.0	35.0	99.0	5.0	9.0	1.0
sigma	30409.096303369428	34.75395002180724	10.092161225345725	27.74697967899083	0.9094184627653707	2.4063142665913926	0.29935787025903
zeros	1468	0	206	58	4605	0	5222
missing	0	0	0	0	0	0	0
0	65999.0	237.0	1.0	14.0	0.0	6.0	0.0
1	0.0	65.0	19.0	56.0	0.0	3.0	0.0
2	54500.0	63.0	28.0	17.0	0.0	6.0	0.0
3	37260.0	62.0	19.0	42.0	0.0	8.0	0.0
4	68987.0	71.0	11.0	40.0	0.0	6.0	0.0
5	42305.0	117.0	10.0	62.0	0.0	2.0	0.0
6	65706.0	91.0	19.0	86.0	0.0	8.0	0.0
7	0.0	90.0	7.0	79.0	0.0	1.0	0.0
8	53243.0	66.0	4.0	15.0	2.0	1.0	0.0
9	0.0	70.0	7.0	1.0	1.0	1.0	0.0

- `type` – информация о типах переменных;
- `mins` – минимальные значения переменных;
- `mean` – средние значения переменных;
- `maxs` – максимальные значения переменных;
- `sigma` – стандартные отклонения переменных;
- `zeros` – нулевые значения переменных;
- `missing` – пропущенные значения переменных;
- `0..9` – первые 10 наблюдений для переменных.

Поговорим немного о типах переменных в H2O.

В H2O количественным переменным соответствует тип `int`, когда значения представляют собой целые числа, и тип `real`, когда значения представляют собой числа с плавающей точкой. Категориальным переменным соответствует тип `enum`, когда значения представляют собой категории, и тип `string` – это текст, его обычно конвертируют в тип `enum`, потому что текстовые данные не могут напрямую использоваться при построении моделей. Порядковые переменные не поддерживаются.

Категориальную зависимую переменную, записанную как целочисленную, нужно обязательно преобразовать в категориальную, иначе будет решаться задача регрессии вместо классификации.

В тех случаях, когда вам нужно преобразовать целочисленную или строковую переменную в категориальную, можно воспользоваться методом `.asfactor()`. Если нужно преобразовать категориальную переменную в целочисленную, воспользуйтесь методом `.asnumeric()`.

Например, так:

```
data['tariff_id'] = data['tariff_id'].asnumeric()
```

У нас есть проблема. Категориальная зависимая переменная `Response` записана как целочисленная. Исправляем это.

```
# преобразовываем в категориальную переменную
tr['Response'] = tr['Response'].asfactor()
tst['Response'] = tst['Response'].asfactor()
```

Не забываем проверить, поменялся ли тип переменной.

```
# смотрим содержимое фрейма
tr.describe()
```

Rows: 5799
Cols: 7

	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
type	int	int	int	int	int	int	enum
mins	0.0	61.0	0.0	0.0	0.0	0.0	1.0
mean	37865.85031902055	93.31401965856195	15.117261596827044	48.28694602517683	0.3821348508363507	2.990170719089499	
maxs	99961.0	298.0	35.0	99.0	5.0	9.0	
sigma	30409.096303369428	34.75395002180724	10.092161225345725	27.74697967899083	0.9094184627653707	2.4063142665913926	
zeros	1468	0	206	58	4605	0	
missing	0	0	0	0	0	0	0
0	65999.0	237.0	1.0	14.0	0.0	6.0	0
1	0.0	65.0	19.0	56.0	0.0	3.0	0
2	54500.0	63.0	28.0	17.0	0.0	6.0	0
3	37260.0	62.0	19.0	42.0	0.0	8.0	0
4	68987.0	71.0	11.0	40.0	0.0	6.0	0
5	42305.0	117.0	10.0	62.0	0.0	2.0	0
6	65706.0	91.0	19.0	86.0	0.0	8.0	0
7	0.0	90.0	7.0	79.0	0.0	1.0	0
8	53243.0	66.0	4.0	15.0	2.0	1.0	0
9	0.0	70.0	7.0	1.0	1.0	1.0	0

3.5. ОПРЕДЕЛЕНИЕ ИМЕНИ ЗАВИСИМОЙ ПЕРЕМЕННОЙ И СПИСКА ИМЕН ПРИЗНАКОВ

Далее мы задаем название зависимой переменной и список названий признаков. Чуть позже с помощью них мы укажем модели машинного обучения, какая переменная в обучающем и тестовом фреймах будет зависимой переменной, а какие переменные будут признаками.

```
# задаем имя зависимой переменной
dependent = 'Response'
# задаем список имен признаков
predictors = list(tr.columns)
# удаляем имя зависимой переменной
# из списка имен признаков
predictors.remove(dependent)
```

3.6. ПОСТРОЕНИЕ МОДЕЛИ МАШИННОГО ОБУЧЕНИЯ

Построение моделей в H2O происходит так же, как в scikit-learn:

- импортируем из соответствующего модуля класс, в котором реализована интересующая модель машинного обучения;
- создаем экземпляр класса – объект-модель;
- обучаем модель, т. е. вычисляем параметры модели, с помощью которых будем получать прогнозы, – используем метод `.train()` объекта-модели.

В H2O нет деления классов на классификаторы и регрессоры, поведение класса определяется типом зависимой переменной. Например, у нас есть класс `H2ORandomForestEstimator`, строящий модель случайного леса. Если переменная является целочисленной (тип `int`) или вещественной (тип `real`), решается задача регрессии, т. е. строится ансамбль деревьев регрессии. Если переменная является кате-

гориальной (тип `enum`), решается задача классификации, т. е. строится ансамбль деревьев классификации. Если переменная является бинарной (например, не вернул – вернул кредит, не откликнулся – откликнулся), при этом закодирована целочисленными значениями и, таким образом, имеет тип `int`, будет ошибочно решаться задача регрессии, в этом случае ей нужно явно присвоить тип `enum`.

Давайте построим модель случайного леса. В H2O для этого метода используется название DRF – distributed random forest (распределенный случайный лес).

```
# импортируем класс H2ORandomForestEstimator
from h2o.estimators import H2ORandomForestEstimator
# создаем экземпляр класса H2ORandomForestEstimator
forest_model = H2ORandomForestEstimator(seed=42)
# обучаем модель
forest_model.train(x=predictors,
y=dependent,
training_frame=tr,
validation_frame=tst);
```

Если опустить точку с запятой после метода `.train()`, сразу после завершения обучения мы получим вывод модели.

3.7. ВЫВОД МОДЕЛИ

Выведем информацию о построенной модели.

```
# смотрим модель
forest_model
```

Первой приводится информация об общих характеристиках модели:

- `number_of_trees` – количество деревьев;
- `model_size_in_bytes` – размер модели;
- `min_depth` – минимальная глубина дерева по ансамблю;
- `mean_depth` – средняя глубина дерева по ансамблю;
- `max_depth` – максимальная глубина дерева по ансамблю;
- `min_leaves` – минимальное количество листьев по ансамблю;
- `mean_leaves` – среднее количество листьев по ансамблю;
- `max_leaves` – максимальное количество листьев по ансамблю.

```
drf Model Build progress: |██████████| (done) 100%
Model Details
=====
H2ORandomForestEstimator : Distributed Random Forest
Model Key: DRF_model_python_1656399751802_1
```

Model Summary:

	<code>number_of_trees</code>	<code>number_of_internal_trees</code>	<code>model_size_in_bytes</code>	<code>min_depth</code>	<code>max_depth</code>	<code>mean_depth</code>	<code>min_leaves</code>	<code>max_leaves</code>	<code>mean_leaves</code>
0	50.0	50.0	294713.0	17.0	20.0	19.92	414.0	506.0	463.58

Затем для обучающей выборки приводятся метрики качества модели, матрица ошибок, оптимальные пороги для метрик – пороговые значения спрогнозированной вероятности положительного класса, при которых достигаются максимальные значения метрик.

```
ModelMetricsBinomial: drf
** Reported on train data. **

MSE: 0.04619564763372571
RMSE: 0.21493172784334497
LogLoss: 0.3693878012757886
Mean Per-Class Error: 0.08679798904382008
AUC: 0.9457811472194362
AUCPR: 0.6697982301465149
Gini: 0.8915622944388724
```

Общие метрики качества**Матрица ошибок**

Строки – фактические классы зависимой переменной, столбцы – спрогнозированные классы зависимой переменной, для классификации на наблюдения отрицательного класса и наблюдения положительного класса используется пороговое значение вероятности

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.3176226465604006:

	0	1	Error	Rate
0	0	4949.0	273.0	0.0523 (273.0/5222.0)
1	1	70.0	507.0	0.1213 (70.0/577.0)
2	Total	5019.0	780.0	0.0591 (343.0/5799.0)

Maximum Metrics: Maximum metrics at their respective thresholds

	metric	threshold	value	idx
0	max f1	0.317623	0.747237	210.0
1	max f2	0.257955	0.830183	232.0
2	max f0point5	0.363177	0.695677	196.0
3	max accuracy	0.363177	0.942404	196.0
4	max precision	0.827996	0.750000	51.0
5	max recall	0.000000	1.000000	399.0
6	max specificity	1.000000	0.996170	0.0
7	max absolute_mcc	0.312368	0.725017	211.0
8	max min_per_class_accuracy	0.227179	0.924358	245.0
9	max mean_per_class_accuracy	0.192454	0.926699	261.0
10	max tns	1.000000	5202.000000	0.0
11	max fns	1.000000	529.000000	0.0
12	max fps	0.000000	5222.000000	399.0
13	max tps	0.000000	577.000000	399.0
14	max tnr	1.000000	0.996170	0.0
15	max fnr	1.000000	0.916811	0.0
16	max fpr	0.000000	1.000000	399.0
17	max tpr	0.000000	1.000000	399.0

Пороговые значения спрогнозированной вероятности события (вероятности положительного класса), при которых достигаются максимальные значения метрик

Пример интерпретации: при вероятности положительного класса 0,318 достигается максимальное значение F1-меры, равное 0,747

название метрики

пороговое значение вероятности события

значение метрики

индекс данного порогового значения в списке пороговых значений

Вслед за оптимальными порогами для метрик, вычисленными на обучающей выборке, приводится таблица выигрышей, вычисленная на обучающей выборке.

Gains/Lift Table: Avg response rate: 9,95 %, avg score: 13,38 %

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score
0	1	0.011726	1.000000	7.094301	7.094301	0.705882	1.000000	0.705882
1	2	0.020003	0.943232	6.490793	6.844574	0.645833	0.960290	0.681034
2	3	0.030005	0.900107	8.144176	7.277774	0.810345	0.924378	0.724138
3	4	0.040007	0.825845	8.317457	7.537695	0.827586	0.867226	0.750000
4	5	0.050009	0.777251	6.584653	7.347087	0.655172	0.801316	0.731034
5	6	0.100017	0.528105	6.446029	6.896558	0.641379	0.648841	0.686207
6	7	0.150026	0.254386	4.470633	6.087916	0.444828	0.375209	0.605747
7	8	0.200034	0.162500	0.727777	4.747881	0.072414	0.200446	0.472414
8	9	0.300224	0.091667	0.051895	3.180754	0.005164	0.120798	0.316485
9	10	0.403863	0.058824	0.066890	2.381680	0.006656	0.072412	0.236977
10	11	0.500776	0.038462	0.035766	1.927684	0.003559	0.047975	0.191804
11	12	0.599931	0.011793	0.034957	1.614859	0.003478	0.026072	0.160678
12	13	1.000000	0.000000	0.077976	1.000000	0.007759	0.000292	0.099500
								0.133802

capture_rate	cumulative_capture_rate	gain	cumulative_gain	kolmogorov_smirnov
0.083189	0.083189	609.430115	609.430115	0.079359
0.053726	0.136915	549.079289	584.457360	0.129830
0.081456	0.218371	714.417618	627.777446	0.209179
0.083189	0.301560	731.745652	653.769497	0.290453
0.065858	0.367418	558.465308	634.708660	0.352481
0.322357	0.6889775	544.602881	589.655770	0.654922
0.223570	0.913345	347.063288	508.791609	0.847661
0.036395	0.949740	-27.222255	374.788143	0.832544
0.005199	0.954939	-94.810537	218.075430	0.727057
0.006932	0.961872	-93.310975	138.167988	0.619666
0.003466	0.965338	-96.423395	92.768416	0.515893
0.003466	0.968804	-96.504257	61.485925	0.409631
0.031196	1.000000	-92.202385	0.000000	0.000000

Таблица выигрышей создается путем разбиения данных на группы по квантилям спрогнозированной вероятности положительного класса, взятым в качестве пороговых значений. Количество групп по умолчанию равно 20, однако если количество уникальных пороговых значений спрогнозированной вероятности меньше 20, то количество групп корректируется на количество этих значений.

Для каждой группы вычисляются накопленная доля данных (`cumulative_data_fraction`), нижнее пороговое значение спрогнозированной вероятности (`lower_threshold`), прирост (`lift`), накопленный прирост (`cumulative_lift`), процент достигнутого отклика (`response_rate`), накопленный процент достигнутого отклика (`cumulative_response_rate`), процент охвата (`capture_rate`), накопленный процент охвата (`cumulative_capture_rate`), выигрыш (`gain`), накопленный выигрыш (`cumulative_gain`) и максимальная абсолютная разница между кумулятивными функциями распределения наблюдений отрицательного класса и наблюдений положительного класса (`kolmogorov_smirnov`).

Прирост (`lift`) по n -й группе – это отношение фактической доли объектов положительного класса в n -й группе к общей фактической доле объектов положительного класса в наборе данных. Например, для вычисления прироста в шестой группе мы должны разделить фактическую долю объектов положительного класса в группе (берем ее из столбца `response_rate`) на общую фак-

тическую долю объектов положительного класса в наборе данных (берем ее из строки Gains/Lift Table: Avg response rate, предваряющей таблицу выигрышей): $0,641379 / 0,0995 = 6,44$.

Gains/Lift Table: Avg response rate: 9,95 %, avg score: 13,38 %

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate
0	1	0.011726	1.000000	7.094301	7.094301
1	2	0.020003	0.943232	6.490793	6.844574
2	3	0.030005	0.900107	8.144176	7.277774
3	4	0.040007	0.825845	8.317457	7.537695
4	5	0.050009	0.777251	6.584653	7.347087
5	6	0.100017	0.528105	6.446029	6.896558

Накопленный прирост (cumulative lift) по n -й группе – это отношение фактической доли объектов положительного класса, накопленной по первым n группам, к общей фактической доле объектов положительного класса в наборе данных. Попробуем вычислить накопленный прирост в шестой группе. Для вычисления накопленного прироста нужно разделить фактическую долю объектов положительного класса, накопленную по первым шести группам (берем ее из столбца cumulative_response_rate), на общую фактическую долю объектов положительного класса в наборе данных: $0,686207 / 0,0995 = 6,90$.

Gains/Lift Table: Avg response rate: 9,95 %, avg score: 13,38 %

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate
0	1	0.011726	1.000000	7.094301	7.094301	0.705882	0.705882
1	2	0.020003	0.943232	6.490793	6.844574	0.645833	0.681034
2	3	0.030005	0.900107	8.144176	7.277774	0.810345	0.924378
3	4	0.040007	0.825845	8.317457	7.537695	0.827586	0.867226
4	5	0.050009	0.777251	6.584653	7.347087	0.655172	0.801316
5	6	0.100017	0.528105	6.446029	6.896558	0.641379	0.648841

Процент отклика (response rate) по n -й группе – это доля объектов положительного класса (событий) в n -й группе от общего количества объектов в n -й группе.

Процент охвата (capture rate) по n -й группе – это отношение фактического числа объектов положительного класса в n -й группе к общему фактическому числу объектов положительного класса в наборе данных. Выигрыш (gain) для n -й группы вычисляется по формуле: $100 * (\text{Прирост по } n\text{-й группе} - 1)$.

Накопленный выигрыш (cumulative gain) для n -й группы вычисляется по формуле: $100 * (\text{Накопленный прирост по } n\text{-й группе} - 1)$.

Положительный выигрыш означает, что фактическая доля событий в группе больше общей фактической доли событий в наборе данных. Отрицательный выигрыш означает, что фактическая доля событий в группе меньше общей фактической доли событий в наборе данных.

После таблицы выигрышей, вычисленной на обучающей выборке, приводятся метрики качества модели, матрица ошибок, оптимальные пороги и таблица выигрышей, вычисленные на тестовой выборке (для экономии места опустим).

Затем приводится таблица с историей вычислений, где по каждой итерации приводится информация о номере итерации, временной метке итерации, продолжительности итерации, количестве построенных деревьев, значениях метрик (RMSE, логистической функции потерь, AUC-ROC, AUC-PR, прироста и ошибки классификации) на обучающей и тестовой выборках.

Scoring History:

	timestamp	duration	number_of_trees	training_rmse	training_logloss	training_auc	training_pr_auc	training_lift	training_classification_error
0	2022-06-28 10:02:36	0.041 sec	0.0	NaN	NaN	NaN	NaN	NaN	NaN
1	2022-06-28 10:02:36	0.203 sec	1.0	0.283558	2.030118	0.870049	0.454732	5.111275	0.098368
	validation_rmse	validation_logloss	validation_auc	validation_pr_auc	validation_lift	validation_classification_error			
	NaN	NaN	NaN	NaN	NaN	NaN			
	0.300895	2.364122	0.856338	0.431526	4.904858	0.107592			

Последний элемент вывода – таблица важностей признаков. При расчете важностей переменных H2O вычисляет квадратичную ошибку до и после разбиения, в котором использовалась интересующая переменная. Разница записывается как улучшение. Таким образом, получаем улучшение (уменьшение) квадратичной ошибки по каждому признаку. Затем суммируем важности признаков, чтобы получить итоговую важность, а потом нормируем, чтобы получить значения в диапазоне от 0 до 1. Обратите внимание, что в случае высококардинальных категориальных признаков с большой вероятностью именно они будут в начале списка.

3.8. ПОЛУЧЕНИЕ ПРОГНОЗОВ

Давайте получим вероятности и прогнозы для тестовой выборки. В качестве порога, определяющего разбиение на отрицательный и положительный классы, выступает значение вероятности положительного класса, при котором достигается максимальное значение F1-меры.

```
# получаем спрогнозированные значения и спрогнозированные
# вероятности классов зависимой переменной
predictions = forest_model.predict(tst)
predictions
```

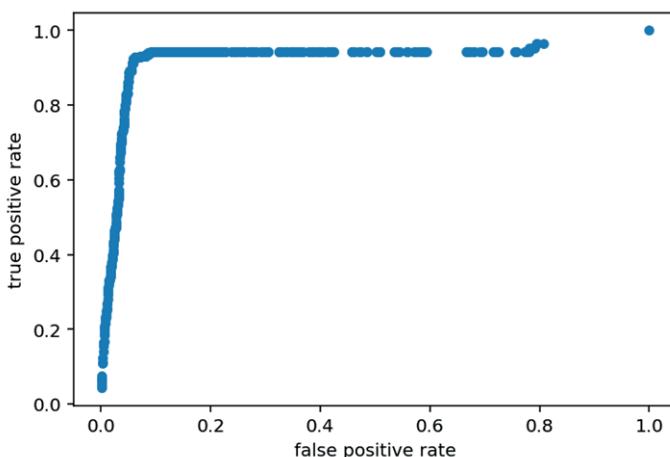
```
drf prediction progress: |██████████| (done) 100%
```

predict	p0	p1
1	0.542	0.458
0	0.928095	0.0719048
0	0.875667	0.124333
0	0.933	0.067
0	0.888	0.112
0	1	0
0	0.931667	0.0683333
0	0.96	0.04
0	0.808	0.192
0	1	0

3.9. ПОСТРОЕНИЕ ROC-КРИВОЙ И ВЫЧИСЛЕНИЕ AUC-ROC

С помощью метода `.roc()` можно построить ROC-кривую. Если для параметра `valid` задано значение `True`, будет построена ROC-кривая для проверочной/тестовой выборки.

```
# выводим ROC-кривую для тестовой выборки
import pandas as pd
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
tmp = model.roc(valid=True)
df = pd.DataFrame({'false positive rate': tmp[0],
                   'true positive rate': tmp[1]})
df.plot(kind='scatter',
        x='false positive rate',
        y='true positive rate');
```



С помощью метода `.auc()` можно вычислить оценку AUC-ROC. Если для параметра `valid` задано значение `True`, будет вычислена оценка AUC-ROC для проверочной/тестовой выборки.

```
# вычисляем AUC-ROC для тестовой выборки  
forest_model.auc(valid=True)
```

```
0.9276565747853592
```

3.10. ПОИСК ОПТИМАЛЬНЫХ ЗНАЧЕНИЙ ГИПЕРПАРАМЕТРОВ ПО СЕТКЕ

Обычный поиск по сетке в H2O осуществляется с помощью класса `H2OGridSearch`. Начиная с версии 3.28 в H2O появился распараллеливаемый поиск по сетке (`Parallel Grid Search`), с помощью параметра `parallelism` класса `H2OGridSearch` можно настраивать уровень распараллеливания, 1 – нет распараллеливания, последовательные вычисления, 0 – адаптивный режим, вычисляем, сколько моделей наилучшим образом можем построить за раз, > 1 – настраиваем уровень распараллеливания, имеет смысл применять, когда нужно перебрать большое количество небольших по размеру моделей (модели логистической регрессии) и когда выбрано большое количество моделей, сильно отличающихся по времени обучения.

В настоящий момент возможность подбора значений гиперпараметров методом комбинированной проверки через конвейер в H2O не реализована. Конвейеры в H2O реализованы, но работают нестабильно. Поэтому используется упрощенная схема: данные разбиваются на обучающую и тестовую выборки, выполняется предварительная подготовка данных (в нашем случае уже есть предварительно подготовленные обучающие и тестовые данные), на обучающей выборке запускается перекрестная проверка, на обучающих блоках учим только модели машинного обучения, на проверочных блоках настраиваем гиперпараметры модели машинного обучения, находим комбинацию оптимальных значений гиперпараметров, обучаем модель с этой комбинацией на обучающей выборке и проверяем на тестовой. Недостаток этой схемы заключается в том, что модели предварительной подготовки уже обучены, гиперпараметры моделей предварительной подготовки не подбираются, а также у нас возникают «утечки» данных при использовании моделей предварительной подготовки, использующих статистики: допустим, мы выполнили импутацию пропусков с помощью статистик на обучающей выборке, запускается перекрестная проверка, получается, что в каждой итерации перекрестной проверки пропуски в обучающих блоках импутированы с использованием информации, содержащейся в проверочном блоке. Поэтому старайтесь использовать операции предварительной подготовки, не предполагающие вычисление статистик. Для схем конструирования признаков, предполагающих вычисление статистик, пользуйтесь соответствующими гиперпараметрами классов.

Давайте воспользуемся k -блочной перекрестной проверкой. В ходе k -блочной перекрестной проверки для каждой комбинации значений гиперпа-

метров строится $k + 1$ моделей. В нашем случае мы применим 5-блочную перекрестную проверку, поэтому для каждой комбинации параметров будет построено шесть моделей: пять моделей перекрестной проверки и одна общая модель.

Первые пять моделей (их называют моделями перекрестной проверки) строятся на 4 обучающих блоках (80 % исходного обучающего набора) и проверяются на одном проверочном блоке (20 % исходного обучающего набора). Прогнозы, полученные по пяти проверочным блокам перекрестной проверки, объединяются, и вычисляются метрики качества на основе объединенных прогнозов (эти метрики будут приведены в отчете о построении модели в разделе `Reported on cross-validation data`). Именно эти метрики будут выведены в результатах поиска по сетке.

Кроме того, метрики качества, полученные по пяти проверочным блокам, усредняются, и мы получаем усредненные метрики перекрестной проверки (именно эти усредненные метрики наряду с метриками по каждому проверочному блоку будут приведены в отчете о построении модели в разделе `Cross-Validation Metrics Summary`).

Потом строится общая модель на 100 % обучающих данных, и мы получаем метрики качества на всей обучающей выборке (эти метрики будут приведены в отчете о построении модели в разделе `Reported on train data`).

Если задана тестовая выборка, то вычисляются метрики качества на тестовой выборке (данные метрики будут приведены в отчете о построении модели в разделе `Reported on validation data`).

Сейчас мы выполним поиск оптимальных значений гиперпараметра `max_depth`.

```
# импортируем класс H2OGridSearch для выполнения поиска по сетке
from h2o.grid.grid_search import H2OGridSearch
# задаем сетку значений гиперпараметров
hyper_parameters = {'max_depth': [15, 20, 25]}
# создаем экземпляр класса H2OGridSearch
gridsearch = H2OGridSearch(H2ORandomForestEstimator,
    grid_id='mygrid',
    hyper_params=hyper_parameters)
# выполняем поиск по сетке
gridsearch.train(predictors,
dependent,
ntrees=100,
training_frame=tr,
nfolds=5,
keep_cross_validation_predictions=True,
seed=100);
```

Теперь выведем результаты поиска. В выводе приводится информация по каждой модели, построенной в ходе поиска по сетке. По умолчанию модели приводятся в порядке увеличения логистической функции потерь, вычисленной по объединенным прогнозам в проверочных блоках.

```
# выводим результаты поиска по сетке
gridsearch
```

```
drf Grid Build progress: |██████████| (done) 100%
   max_depth      model_ids    logloss
0       15.0  mygridsearch_model_1  0.205550
1       20.0  mygridsearch_model_2  0.315175
2       25.0  mygridsearch_model_3  0.369030
```

Для удобства можно отсортировать полученные модели в порядке убывания AUC.

```
# сортируем результаты поиска по сортировке по убыванию AUC
gridperf = gridsearch.get_grid(sort_by='auc', decreasing=True)
gridperf

   max_depth      model_ids    auc
0       15.0  mygridsearch_model_1  0.891856
1       20.0  mygridsearch_model_2  0.885569
2       25.0  mygridsearch_model_3  0.884606
```

3.11. Извлечение наилучшей модели по итогам поиска по сетке

Извлекаем наилучшую модель и проверяем на тестовой выборке, которая не участвовала в построении моделей и настройке гиперпараметров.

```
# извлекаем наилучшую модель
best_model = gridperf.models[0]

# смотрим AUC наилучшей модели на тестовой выборке
bestmodel_perf = best_model.model_performance(tst)
print(bestmodel_perf.auc())
```

0.9287609579755988

Завершаем работу с H2O.

```
# завершаем работу с H2O
h2o.cluster().shutdown()

H2O session _sid_af76 closed.
```

3.12. Класс H2OAutoML

В последние годы спрос на экспертов по машинному обучению опережает предложение, несмотря на рост числа людей, начинающих работать в этой области. Чтобы восполнить данный пробел, были достигнуты большие успехи в разработке удобного программного обеспечения для машинного обучения, которым могли бы воспользоваться неспециалисты. Первые шаги, направленные на демократизацию машинного обучения, включали разработку простых унифицированных интерфейсов для различных алгоритмов машинного обучения.

Чтобы программное обеспечение для машинного обучения было действительно доступным для неспециалистов, в компании H2O разработали про-

стой в использовании интерфейс, который автоматизирует процесс обучения большого количества моделей-кандидатов. AutoML от H2O может быть полезным инструментом и для опытных пользователей. AutoML появился 6 июня 2017 года в версии H2O 3.12.0.1. Этот инструмент предлагает простую функцию-оболочку, которая выполняет большое количество задач, связанных с моделированием и обычно требующих большого количества строк кода. Он освобождает время специалистов, позволяя им сосредоточиться на других аспектах задачи конвейера обработки данных, таких как предварительная обработка данных, конструирование признаков и развертывание моделей.

AutoML от H2O может использоваться для автоматизации рабочего процесса машинного обучения, который включает автоматическое обучение и настройку многих моделей в течение указанного пользователем времени. Он реализован в специальном классе `H2OAutoML`, и сейчас мы проиллюстрируем его работу на простом примере.

Итак, давайте импортируем библиотеку `h2o`, необходимый класс `H2OAutoML` и запустим кластер H2O.

```
# импортируем необходимые библиотеки
import h2o
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
# импортируем класс H2OAutoML
from h2o.automl import H2OAutoML

# запускаем кластер H2O
h2o.init()
```

Загружаем обучающий и тестовый наборы для задачи бинарной классификации.

```
# загружаем обучающий и тестовый наборы
train = h2o.upload_file(path='Data/higgs_train_10k.csv')
test = h2o.upload_file(path='Data/higgs_test_5k.csv')
```

Задаем список признаков и зависимую переменную.

```
# задаем список признаков и зависимую переменную
x = train.columns
y = 'response'
x.remove(y)
```

Зависимой переменной (она у нас является целочисленной) явно задаем тип `enum` с помощью метода `.asfactor()`.

```
# зависимой переменной явно задаем тип enum
train[y] = train[y].asfactor()
test[y] = test[y].asfactor()
```

Запускаем AutoML. Необходимо указать одну из стратегий остановки (на основе времени или количества моделей). Если установлены оба параметра, выполнение AutoML остановится, как только будет удовлетворен один из параметров. Наилучшая модель выбирается:

- для бинарной классификации – по AUC-ROC;
- для многоклассовой классификации – средняя ошибка классификации (например, среднее 1/20, 1/50 и 2/30 будет равно 4,556);
- для регрессии – по среднему остаточному девиансу (для нормального распределения эквивалентен MSE).

Параметр `max_runtime_secs` задает максимальное время, в течение которого будет выполняться процесс AutoML, до обучения моделей стекинга. По умолчанию – 0 (без ограничений), но динамически устанавливается на 1 час, если ни один из параметров (`max_runtime_secs` и `max_models`) не указан пользователем.

Параметр `max_models` задает максимальное количество моделей для построения в рамках AutoML, исключая модели стекинга. По умолчанию `NULL/None`.

Параметр `preprocessing` задает список процедур предобработки. Пока доступно лишь `['target_encoding']` – схема кодирования категорий признака вероятностями на основе зависимой переменной.

Параметр `exclude_algos` задает список алгоритмов для исключения. Например, `exclude_algos = ['GLM', 'DeepLearning', 'DRF']` исключает из процесса автоматического построения моделей обобщенную линейную модель, нейронную сеть и случайный лес.

Параметр `include_algos` задает список алгоритмов для включения. Например, `include_algos = ['GLM', 'DeepLearning', 'DRF']` включает в процесс автоматического построения моделей обобщенную линейную модель, нейронную сеть и случайный лес.

```
# запускаем AutoML
aml = H20AutoML(max_models=20, seed=1)
aml.train(x=x, y=y, training_frame=train)
```

Перечислим основные параметры метода `.train()`:

- `validation_frame`: игнорируется, за исключением случаев, когда `nfolds=0`, используется для ранней остановки конкретных моделей и ранней остановки моделей поиска по сетке (за исключением случаев, когда параметры `max_models` или `max_runtime_secs` переопределяют раннюю остановку на основе выбранной метрики). По умолчанию `nfolds=5`, и поэтому для ранней остановки используются метрики перекрестной проверки и `validation_frame` игнорируется;
- `leaderboard_frame`: позволяет задать пользователю конкретный фрейм, использующийся для скоринга и ранжирования моделей на лидерборде. Этот фрейм применяется только для скоринга на лидерборде. Если `leaderboard_frame` не задан пользователем, то для оценки моделей на лидерборде будут использоваться метрики перекрестной проверки. Если же перекрестная проверка отключена (`nfolds=0`), то `leaderboard_frame` автоматически генерируется из обучающего фрейма;
- `nfolds`: задает количество блоков перекрестной проверки (по умолчанию 5);
- `sort_metric`: задает метрику, по которой сортируем модели на лидерборде. По умолчанию для бинарной классификации используется AUC, для многоклассовой классификации – `mean_per_class_error`, для регрессии – `deviance`.

Класс H2OAutoML обучает и проверяет методом перекрестной проверки следующие алгоритмы (в перечисленном порядке):

- три предварительно настроенные модели градиентного бустинга XGBoost;
- оптимальную модель GLM, найденную в ходе перебора значений гиперпараметров `lambda` и `alpha`;
- случайный лес с настройками по умолчанию (DRF);
- пять предварительно настроенных моделей градиентного бустинга GBM;
- глубокую нейронную сеть с почти стандартными настройками;
- полностью рандомизированные деревья (XRT);
- несколько моделей градиентного бустинга XGBoost, найденных в ходе случайногопоиска по сетке;
- несколько моделей градиентного бустинга GBM, найденных в ходе случайногопоиска по сетке;
- несколько моделей глубоких нейронных сетей, найденных в ходе случайногопоиска по сетке.

Несколько моделей, относящихся к одному и тому же методу машинного обучения, называют «семейством» (family).

В некоторых случаях на выполнение всех алгоритмов может не хватить времени, поэтому некоторые из них могут отсутствовать в лидерборде. Затем AutoML обучает две модели стекинга. Отдельные алгоритмы (или группы алгоритмов) можно отключить с помощью параметра `exclude_algos`. Это полезно, если у вас уже есть некоторое представление об алгоритмах, которые будут хорошо работать с вашим набором данных, хотя иногда это может привести к потере производительности, потому что большее разнообразие среди наборов моделей обычно увеличивает эффективность стекинга. Если вы работаете с высокоразмерными (более 10 тысяч столбцов) и/или разреженными данными, то можете пропустить древовидные алгоритмы (GBM, DRF, XGBoost).

Обе модели стекинга должны давать более высокое качество, чем отдельные модели, за редким исключением. Первая модель содержит все модели, а вторая модель – только наиболее эффективную модель из каждого класса/семейства алгоритмов. Стекинг «Лучшее из семейства» (Best of Family) оптимизирован для промышленного использования, поскольку он содержит всего шесть (или меньше) базовых моделей. Он должен быть относительно быстрым в использовании (для получения прогнозов на основе новых данных) без значительного снижения качества по сравнению со стекингом «Все модели» (All models).

Случайный поиск по сетке не используется для случайного леса, полностью рандомизированных деревьев и GLM.

В случае с GLM мы строим модель с включенным `lambda_search` (перебором значений гиперпараметра `lambda` – силы регуляризации) и списком значений `alpha` – соотношения L1- и L2-штрафов эластичной сети $\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. В итоге возвращаем модель с лучшей комбинацией значений гиперпараметров `lambda` и `alpha`.

Для XGBoost перебираются следующие гиперпараметры:

Гиперпараметр XGBoost	Перебираемые значения
booster	gbtree, dart
col_sample_rate	{0.6, 0.8, 1.0}
col_sample_rate_per_tree	{0.7, 0.8, 0.9, 1.0}
max_depth	{5, 10, 15, 20}
min_rows	{0.01, 0.1, 1.0, 3.0, 5.0, 10.0, 15.0, 20.0}
ntrees	Зафиксировано 10 000 (фактическое значение находится по итогам ранней остановки)
reg_alpha	{0.001, 0.01, 0.1, 1, 10, 100}
reg_lambda	{0.001, 0.01, 0.1, 0.5, 1}
sample_rate	{0.6, 0.8, 1.0}

Для GBM перебираются следующие гиперпараметры:

Гиперпараметр GBM	Перебираемые значения
col_sample_rate	{0.4, 0.7, 1.0}
col_sample_rate_per_tree	{0.4, 0.7, 1.0}
learn_rate	Зафиксировано 0.1
max_depth	{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
min_rows	{1, 5, 10, 15, 30, 100}
min_split_improvement	{1e-4, 1e-5}
ntrees	Зафиксировано 10 000 (фактическое значение находится по итогам ранней остановки)
sample_rate	{0.50, 0.60, 0.70, 0.80, 0.90, 1.00}

Для глубокой нейронной сети перебираются следующие гиперпараметры:

Гиперпараметр DeepLearning	Перебираемые значения
activation	Зафиксировано RectifierWithDropout
epochs	Зафиксировано 10 000 (фактическое значение находится по итогам ранней остановки)
epsilon	{1e-6, 1e-7, 1e-8, 1e-9}
hidden	Поиск по сетке 1: {50}, {200}, {500} Поиск по сетке 2: {50, 50}, {200, 200}, {500, 500} Поиск по сетке 3: {50, 50, 50}, {200, 200, 200}, {500, 500, 500}

Гиперпараметр DeepLearning	Перебираемые значения
hidden_dropout_ratios	Поиск по сетке 1: {0.1}, {0.2}, {0.3}, {0.4}, {0.5} Поиск по сетке 2: {0.1, 0.1}, {0.2, 0.2}, {0.3, 0.3}, {0.4, 0.4}, {0.5, 0.5} Поиск по сетке 3: {0.1, 0.1, 0.1}, {0.2, 0.2, 0.2} {0.3, 0.3, 0.3}, {0.4, 0.4, 0.4}, {0.5, 0.5, 0.5}
input_dropout_ratio	{0.0, 0.05, 0.1, 0.15, 0.2}
rho	{0.9, 0.95, 0.99}

Модели XGBoost могут отсутствовать среди моделей на лидерборде из-за ряда ограничений. XGBoost недоступен на компьютерах с Windows. XGBoost используется только в том случае, если он доступен глобально и не был явно отключен.

Итак, давайте взглянем на лидерборд моделей AutoML.

```
# смотрим лидерборд моделей AutoML
lb = aml.leaderboard
# печатаем все строки вместо 10 по умолчанию
lb.head(rows=lb.ngrows)
```

model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse	
StackedEnsemble_AllModels_1_AutoML_1_20220628_155558	0.789674	0.549929	0.804712		0.31228	0.431793	0.186446
StackedEnsemble_BestOffFamily_1_AutoML_1_20220628_155558	0.787926	0.551724	0.804767		0.322355	0.432595	0.187139
GBM_1_AutoML_1_20220628_155558	0.782194	0.557734	0.800444		0.328093	0.435367	0.189544
GBM_2_AutoML_1_20220628_155558	0.779728	0.560845	0.798186		0.328244	0.436651	0.190664
GBM_5_AutoML_1_20220628_155558	0.77877	0.561756	0.796697		0.333878	0.437108	0.191063
GBM_grid_1_AutoML_1_20220628_155558_model_2	0.777848	0.564655	0.795318		0.332602	0.438088	0.191921
GBM_3_AutoML_1_20220628_155558	0.77498	0.565286	0.794945		0.335329	0.438932	0.192661
GBM_4_AutoML_1_20220628_155558	0.769915	0.570673	0.787955		0.34518	0.441434	0.194864
XGBoost_3_AutoML_1_20220628_155558	0.769166	0.573105	0.783553		0.325881	0.442134	0.195482
XGBoost_grid_1_AutoML_1_20220628_155558_model_2	0.767801	0.581801	0.78593		0.33038	0.445473	0.198447
XGBoost_grid_1_AutoML_1_20220628_155558_model_3	0.767691	0.576985	0.788398		0.351533	0.443808	0.196966
XRT_1_AutoML_1_20220628_155558	0.764217	0.581439	0.782091		0.348096	0.445781	0.19872
DRF_1_AutoML_1_20220628_155558	0.7636	0.580678	0.781476		0.33548	0.445568	0.198531
XGBoost_grid_1_AutoML_1_20220628_155558_model_1	0.756654	0.601002	0.774611		0.370677	0.452522	0.204776
XGBoost_1_AutoML_1_20220628_155558	0.755093	0.604656	0.772562		0.356586	0.453599	0.205752
XGBoost_2_AutoML_1_20220628_155558	0.747688	0.620252	0.76154		0.351283	0.459233	0.210895
GBM_grid_1_AutoML_1_20220628_155558_model_1	0.747666	0.59151	0.763315		0.359667	0.451053	0.203448
DeepLearning_grid_3_AutoML_1_20220628_155558_model_1	0.742033	0.61313	0.744012		0.361878	0.457189	0.209022
DeepLearning_grid_2_AutoML_1_20220628_155558_model_1	0.741416	0.611374	0.748795		0.367854	0.456916	0.208772
DeepLearning_grid_1_AutoML_1_20220628_155558_model_1	0.737148	0.627246	0.746054		0.372826	0.460896	0.212425
DeepLearning_1_AutoML_1_20220628_155558	0.697137	0.636001	0.703676		0.408985	0.470511	0.221381
GLM_1_AutoML_1_20220628_155558	0.682628	0.63852	0.680698		0.397234	0.472683	0.223429

С помощью параметра `extra_columns` функции `h2o.automl.get_leaderboard()` в таблице моделей можно вывести дополнительные столбцы:

- `training_time_ms`: столбец, содержащий информацию о времени обучения каждой модели в миллисекундах (обратите внимание, что сюда не входит обучение моделей перекрестной проверки);
- `predict_time_per_row_ms`: столбец, содержащий информацию о среднем времени получения прогноза для одного наблюдения в миллисекундах.
- ALL: добавляет оба столбца `training_time_ms` и `predict_time_per_row_ms`.

выводим дополнительные столбцы `training_time_ms`

и `predict_time_per_row_ms`

```
lb = h2o.automl.get_leaderboard(aml, extra_columns='ALL') lb.head(rows=lb.nrows)
```

model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse	training_time_ms	predict_time_per_row_ms	algo
itoML_1_20220628_155558	0.789674	0.549929	0.807129		0.31228	0.431793	0.186446	4717	0.032771
itoML_1_20220628_155558	0.787926	0.551724	0.804767		0.322355	0.432595	0.187139	2114	0.019881
itoML_1_20220628_155558	0.782194	0.557734	0.800444		0.328093	0.435367	0.189544	641	0.00873
itoML_1_20220628_155558	0.779728	0.560845	0.798186		0.328244	0.436651	0.190664	380	0.007559
itoML_1_20220628_155558	0.777877	0.561756	0.796697		0.333878	0.437108	0.191063	409	0.00715
20220628_155558_model_2	0.777748	0.564655	0.795318		0.332602	0.438088	0.191921	332	0.007358
itoML_1_20220628_155558	0.77498	0.565286	0.794945		0.335329	0.438932	0.192661	370	0.007215
itoML_1_20220628_155558	0.769915	0.570673	0.787955		0.34518	0.441434	0.194864	451	0.007078
itoML_1_20220628_155558	0.769166	0.573105	0.783553		0.325881	0.442134	0.195482	673	0.002309
20220628_155558_model_2	0.767801	0.581801	0.78593		0.33038	0.445473	0.198447	1319	0.00407
20220628_155558_model_3	0.767691	0.576985	0.788398		0.351533	0.443808	0.196966	1239	0.002307
itoML_1_20220628_155558	0.764217	0.581439	0.782091		0.348096	0.445781	0.19872	1186	0.01249
itoML_1_20220628_155558	0.7636	0.580678	0.781476		0.33548	0.445568	0.198531	904	0.010536
20220628_155558_model_1	0.756654	0.601002	0.774611		0.370677	0.452522	0.204776	936	0.00328
itoML_1_20220628_155558	0.755093	0.604656	0.772562		0.356586	0.453599	0.205752	1045	0.003181
itoML_1_20220628_155558	0.747688	0.620252	0.76154		0.351283	0.459233	0.210895	931	0.003208
20220628_155558_model_1	0.747666	0.59151	0.763315		0.359667	0.451053	0.203448	570	0.009391
20220628_155558_model_1	0.742033	0.61313	0.744012		0.361878	0.457189	0.209022	37116	0.017961
20220628_155558_model_1	0.741416	0.611374	0.748795		0.367854	0.456916	0.208772	31979	0.011092
20220628_155558_model_1	0.737148	0.627246	0.746054		0.372826	0.460896	0.212425	32398	0.004347
itoML_1_20220628_155558	0.697137	0.636001	0.703676		0.408985	0.470511	0.221381	280	0.002246
itoML_1_20220628_155558	0.682628	0.63852	0.680698		0.397234	0.472683	0.223429	129	0.001168

С помощью модели стекинга All models получаем прогнозы для тестовой выборки.

получаем прогнозы

```
preds = aml.predict(test)
preds
```

```
stackedensemble prediction progress: |██████████| (done) 100%
```

predict	p0	p1
0	0.727538	0.272462
0	0.754832	0.245168
1	0.544422	0.455578
1	0.319891	0.680109
0	0.741932	0.258068
1	0.255611	0.744389
1	0.234863	0.765137
1	0.612528	0.387472
1	0.600721	0.399279
0	0.834065	0.165935

еще можно так

```
preds = aml.leader.predict(test)
preds
```

```
stackedensemble prediction progress: |██████████| (done) 100%
```

predict	p0	p1
0	0.727538	0.272462
0	0.754832	0.245168
1	0.544422	0.455578
1	0.319891	0.680109
0	0.741932	0.258068
1	0.255611	0.744389
1	0.234863	0.765137
1	0.612528	0.387472
1	0.600721	0.399279
0	0.834065	0.165935

Посмотрим качество этой модели на тестовой выборке.

смотрим качество модели стекинга All Models на тестовой выборке

```
bestmodel_perf = aml.leader.model_performance(test) print(bestmodel_perf.auc())
```

```
0.7939985762032892
```

Чтобы понять, как работает ансамбль, давайте заглянем внутрь модели стекинга All models. Модель стекинга All models – это ансамбль всех отдельных моделей, запущенных в ходе AutoML.

Давайте исследуем важности метамодели. Они покажут нам, какой вклад каждая базовая модель вносит. Стекинг AutoML использует по умолчанию в качестве метамодели GLM с неотрицательными весами, поэтому важности переменных метамодели фактически являются стандартизованными коэффициентами GLM.

Сейчас мы извлечем модель стекинга All Models и визуализируем стандартизованные коэффициенты – важности (вклады) базовых моделей.

получаем идентификаторы всех моделей в лидерборде

```
model_ids = list(aml.leaderboard['model_id'].as_data_frame().iloc[:, 0])
```

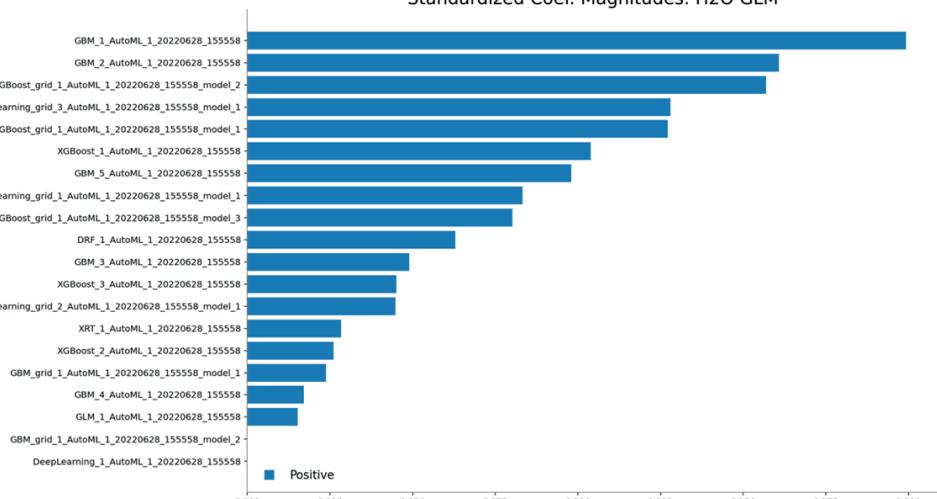
```
# извлекаем модель стекинга All Models
se = h2o.get_model(
    [mid for mid in model_ids if 'StackedEnsemble_AllModels' in mid][0])
# извлекаем метамодель
metalearner = h2o.get_model(se.metalearner()['name'])
# смотрим стандартизированные коэффициенты
metalearner.coef_norm()

# смотрим стандартизированные коэффициенты
metalearner.coef_norm()

{'Intercept': 0.15082902619212202,
 'GBM_1_AutoML_1_20220628_155558': 0.19926997641472557,
 'GBM_2_AutoML_1_20220628_155558': 0.16074907708265146,
 'GBM_5_AutoML_1_20220628_155558': 0.09807172962404992,
 'GBM_grid_1_AutoML_1_20220628_155558_model_2': 0.0,
 'GBM_3_AutoML_1_20220628_155558': 0.04903512218304459,
 'GBM_4_AutoML_1_20220628_155558': 0.017096238019499412,
 'XGBoost_3_AutoML_1_20220628_155558': 0.04516930371837921,
 'XGBoost_grid_1_AutoML_1_20220628_155558_model_2': 0.15693069217532737,
 'XGBoost_grid_1_AutoML_1_20220628_155558_model_3': 0.0802279157024242,
 'XRT_1_AutoML_1_20220628_155558': 0.02836522535795825,
 'DRF_1_AutoML_1_20220628_155558': 0.06288133736408479,
 'XGBoost_grid_1_AutoML_1_20220628_155558_model_1': 0.12715310209578262,
 'XGBoost_1_AutoML_1_20220628_155558': 0.10393346260205943,
 'XGBoost_2_AutoML_1_20220628_155558': 0.02614812412347114,
 'GBM_grid_1_AutoML_1_20220628_155558_model_1': 0.023838671675957882,
 'DeepLearning_grid_3_AutoML_1_20220628_155558_model_1': 0.12798694780996023,
 'DeepLearning_grid_2_AutoML_1_20220628_155558_model_1': 0.04479296260960738,
 'DeepLearning_grid_1_AutoML_1_20220628_155558_model_1': 0.08330204091297139,
 'DeepLearning_1_AutoML_1_20220628_155558': 0.0,
 'GLM_1_AutoML_1_20220628_155558': 0.015245532454528362}

# визуализируем стандартизированные коэффициенты
metalearner.std_coef_plot();
```

Standardized Coef. Magnitudes: H2O GLM



Сохраняем модель стекинга All Models с помощью функции `h2o.save_model()`.

```
pth = ('/Users/artemgruzdev/Documents/Github/' +
       'Data Preprocessing in Python/best_automl_model')
h2o.save_model(aml.leader, path=pth)
```

3.13. ПРИМЕНЕНИЕ КЛАССА H2OAutoML В БИБЛИОТЕКЕ SCIKIT-LEARN

Начиная с версии 3.28.0.1 мы можем использовать класс `H2OAutoML` как обычный класс библиотеки `scikit-learn` и комбинировать его с другими классами библиотеки `scikit-learn`. В модуле `h2o.sklearn` реализованы две оболочки (класса) для класса `H2OAutoML`:

- `H2OAutoMLClassifier`;
- `H2OAutoMLRegressor`.

Эти оболочки предлагают стандартный API библиотеки `scikit-learn` (методы `.fit()`, `.predict()`, `.fit_predict()`, `.score()`, `.get_params()`, `.set_params()`) и принимают различные форматы в качестве входных данных (фрейм `H2O`, массив `NumPy`, датафрейм `pandas`). Их можно комбинировать с исходными классами библиотеки `scikit-learn` в пайплайны. Кроме того, можно получить доступ к исходному алгоритму `H2OAutoML` из оболочки благодаря его атрибуту `estimator`: это дает доступ ко всем методам и свойствам, объявленным в `H2OAutoML`, таким как `leaderboard`, `leader`, `event_log`, `training_info` и др.

Коллаборация `scikit-learn` и `H2O` полезна для тестирования эффективности вашей предварительной подготовки. Например, вы делаете сложную предварительную подготовку данных и хотите посмотреть, как различные модели машинного обучения (линейные модели, модели ансамблей деревьев) «отреагируют» на нее, какими гиперпараметрами обладают лучшие модели из семейств, даст ли улучшение стекинг (при условии что его разрешено использовать).

Давайте загрузим необходимые библиотеки, классы и функции, отключим предупреждения и запустим кластер `H2O`.

```
# импортируем необходимые библиотеки, классы и функции
import h2o
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split,
                                    GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from category_encoders import CountEncoder
from h2o.sklearn import (H2OAutoMLClassifier,
                         H2OAutoMLRegressor)
from sklearn.impute import SimpleImputer

# отключаем предупреждения
import warnings
warnings.filterwarnings('ignore')
```

Для задачи классификации загрузим расширенные данные StateFarm. В качестве метрики качества возьмем правильность.

```
# загружаем данные для задачи классификации
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')
data.head()
```

	Customer Lifetime Value	Coverage	Education	EmploymentStatus	Gender	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN	32.0	5.0	NaN	1.0	No
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN	13.0	42.0	NaN	NaN	No
2	NaN	NaN	NaN	Employed	F	48767.0	108.0	NaN	38.0	0.0	NaN	No
3	7645.861827	Basic	Bachelor		NaN	NaN	0.0	106.0	18.0	NaN	NaN	7.0
4	2813.692575	Basic	Bachelor		NaN	M	43836.0	73.0	12.0	NaN	NaN	1.0

Давайте создадим простой конвейер из классов SimpleImputer и H2OAutoMLClassifier и запустим AutoML. Как и у класса H2OAutoML, у класса H2OAutoMLClassifier есть параметр preprocessing, в который можно передать список процедур предобработки. Пока доступно лишь ['target_encoding'] – схема кодирования категорий признака вероятностями положительного класса зависимой переменной.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    random_state=42)

# превращаем датафреймы в массивы NumPy
X_train = X_train.values
X_test = X_test.values

y_train = y_train.values
y_test = y_test.values

# создаем конвейер
pipe = Pipeline([
    ('imp', SimpleImputer(strategy='constant')),
    ('classifier', H2OAutoMLClassifier(preprocessing=None,
                                       max_models=10, seed=2022))
])

# запускаем AutoML
pipe.fit(X_train, y_train);
```

```
Parse progress: |██████████| (done) 100%
Parse progress: |██████████| (done) 100%
AutoML progress: |██████████| (done) 100%
```

Вычислим правильность на тестовой выборке. Обратите внимание, что прогнозы вычисляются с помощью наилучшей модели (не всегда этой моделью будет модель стекинга).

```
# получаем прогнозы для тестовой выборки
predictions = pipe.predict(X_test)

# оценим правильность на тестовой выборке
acc = accuracy_score(y_test, predictions)
print("Правильность на тестовой выборке: %.3f" % acc)
```

```
Parse progress: |██████████| (done) 100%
gbm prediction progress: |██████████| (done) 100%
Правильность на тестовой выборке: 0.941
```

Давайте взглянем на лидерборд моделей AutoML.

```
# выводим результаты AutoML
automl = pipe.named_steps.classifier.estimator
automl.leaderboard
```

model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse
GBM_4_AutoML_1_20220629_182612	0.913029	0.17163	0.629657	0.0897083	0.216536	0.046888
StackedEnsemble_BestOfFamily_1_AutoML_1_20220629_182612	0.912744	0.169933	0.628493	0.0919027	0.217101	0.0471331
StackedEnsemble_AllModels_1_AutoML_1_20220629_182612	0.912073	0.169591	0.629174	0.0909478	0.216839	0.047019
GBM_3_AutoML_1_20220629_182612	0.910665	0.174784	0.630274	0.0927643	0.219404	0.0481382
GBM_2_AutoML_1_20220629_182612	0.909894	0.180035	0.618754	0.115197	0.222876	0.0496736
XGBoost_2_AutoML_1_20220629_182612	0.908942	0.192712	0.592121	0.164934	0.233423	0.0544862
GBM_1_AutoML_1_20220629_182612	0.906138	0.192051	0.592964	0.134694	0.233383	0.0544677
XGBoost_3_AutoML_1_20220629_182612	0.904502	0.193615	0.579166	0.117902	0.232535	0.0540726
XRT_1_AutoML_1_20220629_182612	0.901894	0.218027	0.624927	0.115005	0.226119	0.0511296
DRF_1_AutoML_1_20220629_182612	0.898166	0.30724	0.616114	0.10327	0.226227	0.0511785

Худшие результаты дает модель случайного леса, попробуем исключить ее, возможно, являясь самым «слабым звеном», она ухудшает ансамбль целом.

```
# создаем конвейер
pipe = Pipeline([
    ('imp', SimpleImputer(strategy='constant')),
    ('classifier', H2OAutoMLClassifier(preprocessing=None,
                                         exclude_algos=['DRF'],
                                         max_models=10, seed=2022))
])
```

```
# запускаем AutoML
pipe.fit(X_train, y_train);
```

```
Parse progress: |██████████| (done) 100%
Parse progress: |██████████| (done) 100%
AutoML progress: |██████████| (done) 100%
```

Вычислим правильность на тестовой выборке.

```
# получаем прогнозы для тестовой выборки
predictions = pipe.predict(X_test)
# оценим правильность на тестовой выборке
acc = accuracy_score(y_test, predictions)
print("Правильность на тестовой выборке: %.3f" % acc)
```

Parse progress: |██████████| (done) 100%
gbm prediction progress: |██████████| (done) 100%
Правильность на тестовой выборке: 0.942

Правильность на тестовой выборке практически не изменилась.

Опять взглянем на лидерборд моделей AutoML.

```
# выводим результаты AutoML
automl = pipe.named_steps.classifier.estimator
automl.leaderboard
```

	model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse
	GBM_3_AutoML_2_20220629_182857	0.91451	0.174031	0.628115		0.0897157	0.219013
StackedEnsemble_BestOfFamily_1_AutoML_2_20220629_182857		0.913688	0.173071	0.627176		0.0899072	0.219735
	GBM_4_AutoML_2_20220629_182857	0.9135	0.171677	0.635448		0.0928575	0.216805
StackedEnsemble_AllModels_1_AutoML_2_20220629_182857		0.913416	0.169593	0.632087		0.0929533	0.216986
	GBM_2_AutoML_2_20220629_182857	0.909623	0.177571	0.628567		0.0968757	0.221213
	XGBoost_2_AutoML_2_20220629_182857	0.908942	0.192712	0.592121		0.164934	0.233423
	GBM_5_AutoML_2_20220629_182857	0.907962	0.18445	0.615031		0.123695	0.225956
	XGBoost_3_AutoML_2_20220629_182857	0.907085	0.189672	0.600344		0.141068	0.228614
	GBM_1_AutoML_2_20220629_182857	0.906138	0.192051	0.592964		0.134694	0.233383
	XGBoost_1_AutoML_2_20220629_182857	0.892651	0.208908	0.536815		0.185764	0.24532

Теперь попробуем более сложную предварительную подготовку с использованием класса `ColumnTransformer`. Мы попробуем кодировать категориальные признаки абсолютными частотами с помощью класса `CountEncoder` пакета `category_encoders`. Например, категориальный признак *Животное* распределен так: 'кошка', 'кошка', 'кошка', 'собака'. Тогда категория 'кошка' получит частоту 3, а категория 'собака' получит частоту 1. В итоге категориальный признак становится количественным и распределен уже так: 3, 3, 3, 1. Частоты категорий для переменной в обучающем наборе запоминаем и применяем их к категориям в тестовом наборе.

```
# разбиваем данные на обучающие и тестовые: получаем
# обучающий массив признаков, тестовый массив признаков,
# обучающий массив меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)

# создаем списки категориальных
# и количественных столбцов
cat_columns = X_train.select_dtypes(
```

```

include='object').columns.tolist()
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()

# массивы меток (серии) превращаем в массивы NumPy
y_train = y_train.values
y_test = y_test.values

# создаем конвейер для количественных переменных
num_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median'))
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('count', CountEncoder(return_df=False))
])

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
                 ('cat', cat_pipe, cat_columns)]

# создаем конвейер
pipe = Pipeline([
    ('tr', ColumnTransformer(transformers=transformers)),
    ('classifier', H2OAutoMLClassifier(preprocessing=None,
                                         max_models=10, seed=2022))])

# запускаем AutoML
pipe.fit(X_train, y_train);

```

```

Parse progress: |██████████| (done) 100%
Parse progress: |██████████| (done) 100%
AutoML progress: |██████████| (done) 100%

```

Правильность на тестовой выборке уменьшилась, кодирование частотами категориальных признаков не оправдало себя. Здесь отметим, что превращение категориальных признаков в количественные будет наиболее эффективным для высококардинальных признаков, а в этом наборе категориальные признаки имеют небольшую кардинальность (меньше 10 категорий). Для признаков низкой кардинальности дамми-кодирование будет неплохим вариантом обработки.

```

# получаем прогнозы для тестовой выборки
predictions = pipe.predict(X_test)
# оценим правильность на тестовой выборке
acc = accuracy_score(y_test, predictions)
print("Правильность на тестовой выборке: %.3f" % acc)

```

```

Parse progress: |██████████| (done) 100%
stackedensemble prediction progress: |██████████| (done) 100%
Правильность на тестовой выборке: 0.938

```

Взглянем на лидерборд моделей AutoML.

```
# выводим результаты AutoML
automl = pipe.named_steps.classifier.estimator
automl.leaderboard
```

model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse
StackedEnsemble_BestOfFamily_1_AutoML_3_20220629_183108	0.913198	0.170277	0.628174		0.0936234	0.217896
StackedEnsemble_AllModels_1_AutoML_3_20220629_183108	0.912461	0.170575	0.625438		0.0912375	0.218012
GBM_4_AutoML_3_20220629_183108	0.912366	0.172513	0.627018		0.0901869	0.217613
XGBoost_2_AutoML_3_20220629_183108	0.910819	0.189462	0.601831		0.136795	0.231553
GBM_2_AutoML_3_20220629_183108	0.909529	0.179718	0.621232		0.0988836	0.222935
GBM_3_AutoML_3_20220629_183108	0.908902	0.176946	0.62083		0.0928649	0.220594
GBM_1_AutoML_3_20220629_183108	0.904372	0.195252	0.581743		0.141662	0.235526
DRF_1_AutoML_3_20220629_183108	0.902045	0.302361	0.61531		0.131507	0.225393
XGBoost_3_AutoML_3_20220629_183108	0.901969	0.194533	0.587154		0.128295	0.231816
XRT_1_AutoML_3_20220629_183108	0.901799	0.260677	0.624442		0.104895	0.225108

Теперь пересоздадим конвейер и продемонстрируем применение класса GridSearchCV.

```
# пересоздаем итоговый конвейер

# создаем конвейер для количественных переменных
num_pipe = Pipeline([
    ('imputer', SimpleImputer())
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imputer', SimpleImputer())
])

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
                 ('cat', cat_pipe, cat_columns)]

# создаем конвейер
pipe = Pipeline([
    ('tr', ColumnTransformer(transformers=transformers)),
    ('classifier', H2OAutoMLClassifier(preprocessing=None,
                                         max_models=10, seed=2022))])

# задаем сетку гиперпараметров
hyperparams_dct = {
    'tr_num_imputer_strategy': ['mean', 'median', 'constant'],
    'tr_cat_imputer_strategy': ['most_frequent', 'constant']
}

# запускаем поиск по сетке
grid = GridSearchCV(pipe, cv=3, param_grid=hyperparams_dct)
grid.fit(X_train, y_train);
```

Теперь с помощью наилучшей модели вычислим правильность на тестовой выборке.

```
# получаем прогнозы для тестовой выборки
predictions = grid.predict(X_test)
# оценим правильность на тестовой выборке
acc = accuracy_score(y_test, predictions)
print("Правильность на тестовой выборке: %.3f" % acc)
```

```
Parse progress: |██████████| (done) 100%
stackedensemble prediction progress: |██████████| (done) 100%
Правильность на тестовой выборке: 0.941
```

еще можно так

```
# извлекаем наилучшую модель
best = grid.best_estimator_
# получаем прогнозы для тестовой выборки
predictions = best.predict(X_test)
# оценим правильность на тестовой выборке
acc = accuracy_score(y_test, predictions)
print("Правильность на тестовой выборке: %.3f" % acc)
```

```
Parse progress: |██████████| (done) 100%
stackedensemble prediction progress: |██████████| (done) 100%
Правильность на тестовой выборке: 0.941
```

Выведем наилучшие гиперпараметры и результаты поиска по сетке, записанные в сводную таблицу.

```
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    grid.best_params_))

Наилучшие значения гиперпараметров:
{'tr_cat_imputer_strategy': 'constant',
 'tr_num_imputer_strategy': 'constant'}
```

```
# запишем результаты поиска в DataFrame
results = pd.DataFrame(grid.cv_results_)
# превращаем в сводную таблицу
table = results.pivot_table(
    values=['mean_test_score'],
    index=['param_tr_cat_imputer_strategy',
           'param_tr_num_imputer_strategy'])
table
```

		mean_test_score	
param_tr_cat_imputer_strategy param_tr_num_imputer_strategy			
constant	constant	constant	0.933678
	mean	mean	0.929716
	median	median	0.932300
most_frequent	constant	constant	0.931611
	mean	mean	0.932300
	median	median	0.932300

Наконец, взглянем на лидерборд моделей AutoML.

```
# выводим результаты AutoML
best = grid.best_estimator_['classifier']
automl = best.estimator
automl.leaderboard
```

	model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse
StackedEnsemble_BestOfFamily_1_AutoML_22_20220629_185309	0.912305	0.169704	0.632853		0.092009	0.217249	0.0471972
GBM_4_AutoML_22_20220629_185309	0.911861	0.172413	0.630323		0.0917137	0.21719	0.0471715
StackedEnsemble_AllModels_1_AutoML_22_20220629_185309	0.910989	0.170257	0.629147		0.0926685	0.217529	0.0473187
XGBoost_2_AutoML_22_20220629_185309	0.910851	0.185772	0.625292		0.122281	0.227859	0.0519196
GBM_3_AutoML_22_20220629_185309	0.910332	0.175707	0.629146		0.0921948	0.219893	0.0483529
GBM_1_AutoML_22_20220629_185309	0.909179	0.18993	0.597023		0.144038	0.23173	0.0536986
GBM_2_AutoML_22_20220629_185309	0.908555	0.179801	0.621628		0.115005	0.223249	0.0498399
XGBoost_3_AutoML_22_20220629_185309	0.902631	0.193709	0.587397		0.140597	0.23112	0.0534165
XRT_1_AutoML_22_20220629_185309	0.898481	0.23301	0.623064		0.105565	0.22602	0.051085
DRF_1_AutoML_22_20220629_185309	0.898156	0.387124	0.617498		0.116338	0.225769	0.0509716

Для задачи регрессии загрузим данные о сделках с недвижимостью.

```
# загружаем данные для задачи регрессии
data = pd.read_csv('Data/Flats_missing.csv',
                    sep=';', decimal=',')
data.head()
```

Rooms_Number	District	Stor	Storeys	Space_Total	Space_Living	Space_Kitchen	Balcon_Num	Lodgee_Num	lat	Long	Cost_KV
0	1	Заельцовский	13	17.0	54.1	18.0	21.2	0.0	1	55.0725	82.9069
1	1	Заельцовский	10	17.0	54.5	18.0	21.1	0.0	1	55.0725	82.9069
2	1	Центральный	8	17.0	37.0	0.0	0.0	0.0	0	55.0725	82.9068
3	1	Центральный	2	17.0	42.0	0.0	0.0	0.0	0	55.0725	82.9068
4	1	Центральный	13	17.0	28.0	0.0	0.0	0.0	0	55.0725	82.9068

Здесь мы создадим простой конвейер из классов SimpleImputer и H2OAutoMLRegressor и запустим AutoML.

```

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    random_state=42)

# превращаем датафреймы в массивы NumPy
X_train = X_train.values
X_test = X_test.values

y_train = y_train.values
y_test = y_test.values

# создаем конвейер
pipe = Pipeline([
    ('imp', SimpleImputer(strategy='constant')),
    ('regressor', H2OAutoMLRegressor(preprocessing=None,
                                      max_models=10, seed=2022))
])

```

```

# запускаем AutoML
pipe.fit(X_train, y_train);

Parse progress: |██████████| (done) 100%
Parse progress: |██████████| (done) 100%
AutoML progress: |██████████| (done) 100%

```

Вычислим RMSE на тестовой выборке.

```

Parse progress: |██████████| (done) 100%
stackedensemble prediction progress: |██████████| (done) 100%
RMSE на тестовой выборке: 9483.515

```

Давайте взглянем на лидерборд моделей AutoML.

```

# выводим результаты AutoML
automl = pipe.named_steps.regressor.estimator
automl.leaderboard

```

	model_id	rmse	mse	mae	rmsle	mean_residual_deviance
StackedEnsemble_AllModels_1_AutoML_23_20220629_185527	11109.4	1.23418e+08	6734.8	0.181944		1.23418e+08
StackedEnsemble_BestOfFamily_1_AutoML_23_20220629_185527	11160.9	1.24565e+08	6771.15	0.182826		1.24565e+08
GBM_4_AutoML_23_20220629_185527	11227.1	1.26049e+08	6851.11	0.184222		1.26049e+08
GBM_1_AutoML_23_20220629_185527	11304.2	1.27785e+08	6924	0.184864		1.27785e+08
GBM_3_AutoML_23_20220629_185527	11348.7	1.28793e+08	6961.28	0.186158		1.28793e+08
DRF_1_AutoML_23_20220629_185527	11358.3	1.2901e+08	6921.17	0.185917		1.2901e+08
XRT_1_AutoML_23_20220629_185527	11360.2	1.29053e+08	6908.06	0.185547		1.29053e+08
GBM_2_AutoML_23_20220629_185527	11395	1.29846e+08	7012.8	0.186889		1.29846e+08
XGBoost_3_AutoML_23_20220629_185527	11622.4	1.35079e+08	7163.02	0.190612		1.35079e+08
XGBoost_2_AutoML_23_20220629_185527	11628.4	1.3522e+08	7110.53	0.190115		1.3522e+08

Мы можем извлечь нужную нам модель с помощью идентификатора.

```
# извлекаем нужную нам модель
h2o.get_model('DRF_1_AutoML_23_20220629_185527')

Model Details
=====
H2ORandomForestEstimator : Distributed Random Forest
Model Key: DRF_1_AutoML_23_20220629_185527

Model Summary:
  number_of_trees  number_of_internal_trees  model_size_in_bytes  min_depth  max_depth  mean_depth  min_leaves  max_leaves  mean_leaves
0              47.0                      47.0            9672250.0        20.0       20.0      20.0     14165.0    19153.0   16379.915
```

Мы можем извлечь наилучшую модель из семейства GBM.

```
# извлекаем наилучшую модель из семейства GBM
gbm = automl.get_best_model(algorithm='gbm')
gbm

Model Details
=====
H2OGradientBoostingEstimator : Gradient Boosting Machine
Model Key: GBM_4_AutoML_23_20220629_185527

Model Summary:
  number_of_trees  number_of_internal_trees  model_size_in_bytes  min_depth  max_depth  mean_depth  min_leaves  max_leaves  mean_leaves
0              76.0                      76.0            247659.0        10.0       10.0      10.0      28.0      570.0    254.92105
```

Теперь посмотрим ее параметры и гиперпараметры.

```
# смотрим параметры и гиперпараметры извлеченной модели
gbm.params

{'auc_type': {'actual': 'AUTO', 'default': 'AUTO', 'input': 'AUTO'},
 'balance_classes': {'actual': False, 'default': False, 'input': False},
 'build_tree_one_node': {'actual': False, 'default': False, 'input': False},
 'calibrate_model': {'actual': False, 'default': False, 'input': False},
 'calibration_frame': {'actual': None, 'default': None, 'input': None},
 'categorical_encoding': {'actual': 'Enum',
 'default': 'AUTO',
 'input': 'AUTO'},
 'check_constant_response': {'actual': True, 'default': True, 'input': True},
 'checkpoint': {'actual': None, 'default': None, 'input': None},
 'class_sampling_factors': {'actual': None, 'default': None, 'input': None},
 'col_sample_rate': {'actual': 0.8, 'default': 1.0, 'input': 0.8},
 'col_sample_rate_change_per_level': {'actual': 1.0,
 'default': 1.0,
 'input': 1.0},
 'col_sample_rate_per_tree': {'actual': 0.8, 'default': 1.0, 'input': 0.8},
 'custom_distribution_func': {'actual': None, 'default': None, 'input': None},
 'custom_metric_func': {'actual': None, 'default': None, 'input': None},
 'distribution': {'actual': 'gaussian',
 'default': 'AUTO',
 'input': 'gaussian'},
 'export_checkpoints_dir': {'actual': None, 'default': None, 'input': None},
 'fold_assignment': {'actual': 'Modulo', 'default': 'AUTO', 'input': 'Modulo'},
```

```
'fold_column': {'actual': None, 'default': None, 'input': None},
'gainslift_bins': {'actual': -1, 'default': -1, 'input': -1},
'histogram_type': {'actual': 'UniformAdaptive',
'default': 'AUTO',
'input': 'AUTO'},
'huber_alpha': {'actual': 0.9, 'default': 0.9, 'input': 0.9},
'ignore_const_cols': {'actual': True, 'default': True, 'input': True},
'ignored_columns': {'actual': None, 'default': None, 'input': None},
'interaction_constraints': {'actual': None, 'default': None, 'input': None},
'keep_cross_validation_fold_assignment': {'actual': False,
'default': False,
'input': False},
'keep_cross_validation_models': {'actual': False,
'default': True,
'input': False},
'keep_cross_validation_predictions': {'actual': True,
'default': False,
'input': True},
'learn_rate': {'actual': 0.1, 'default': 0.1, 'input': 0.1},
'learn_rate_annealing': {'actual': 1.0, 'default': 1.0, 'input': 1.0},
'max_abs_leafnode_pred': {'actual': 1.7976931348623157e+308,
'default': 1.7976931348623157e+308,
'input': 1.7976931348623157e+308},
'max_after_balance_size': {'actual': 5.0, 'default': 5.0, 'input': 5.0},
'max_confusion_matrix_size': {'actual': 20, 'default': 20, 'input': 20},
'max_depth': {'actual': 10, 'default': 5, 'input': 10},
'max_runtime_secs': {'actual': 0.0, 'default': 0.0, 'input': 0.0},
'min_rows': {'actual': 10.0, 'default': 10.0, 'input': 10.0},
'min_split_improvement': {'actual': 1e-05, 'default': 1e-05, 'input': 1e-05},
'model_id': {'actual': {'URL': '/3/Models/GBM_4_AutoML_23_20220629_185527',
'_meta': {'schema_name': 'ModelKeyV3',
'schema_type': 'Key<Model>',
'schema_version': 3},
'name': 'GBM_4_AutoML_23_20220629_185527',
'type': 'Key<Model>'},
'default': None,
'input': None},
'monotone_constraints': {'actual': None, 'default': None, 'input': None},
'nbins': {'actual': 20, 'default': 20, 'input': 20},
'nbins_cats': {'actual': 1024, 'default': 1024, 'input': 1024},
'nbins_top_level': {'actual': 1024, 'default': 1024, 'input': 1024},
'nfolds': {'actual': 5, 'default': 0, 'input': 5},
'ntrees': {'actual': 76, 'default': 50, 'input': 10000},
'offset_column': {'actual': None, 'default': None, 'input': None},
'pred_noise_bandwidth': {'actual': 0.0, 'default': 0.0, 'input': 0.0},
'quantile_alpha': {'actual': 0.5, 'default': 0.5, 'input': 0.5},
'r2_stopping': {'actual': 1.7976931348623157e+308,
'default': 1.7976931348623157e+308,
'input': 1.7976931348623157e+308},
'response_column': {'actual': {'__meta': {'schema_name': 'ColSpecifierV3',
'schema_type': 'VecSpecifier',
'schema_version': 3},
'column_name': 'C12',
'is_member_of_frames': None,
'default': None,
'input': {'__meta': {'schema_name': 'ColSpecifierV3',
'schema_type': 'VecSpecifier',

```

```
'schema_version': 3},
'column_name': 'C12',
'is_member_of_frames': None}],
'sample_rate': {'actual': 0.8, 'default': 1.0, 'input': 0.8},
'sample_rate_per_class': {'actual': None, 'default': None, 'input': None},
'score_each_iteration': {'actual': False, 'default': False, 'input': False},
'score_tree_interval': {'actual': 5, 'default': 0, 'input': 5},
'seed': {'actual': 2029, 'default': -1, 'input': 2029},
'stopping_metric': {'actual': 'deviance',
'default': 'AUTO',
'input': 'deviance'},
'stopping_rounds': {'actual': 0, 'default': 0, 'input': 3},
'stopping_tolerance': {'actual': 0.004462107482531602,
'default': 0.001,
'input': 0.004462107482531602},
'training_frame': {'actual': {'URL': '/3/Frames/AutoML_23_20220629_185527_training_py_152_sid_8f55',
'_meta': {'schema_name': 'FrameKeyV3',
'schema_type': 'Key<Frame>',
'schema_version': 3},
'name': 'AutoML_23_20220629_185527_training_py_152_sid_8f55',
'type': 'Key<Frame>'},
'default': None,
'input': {'URL': '/3/Frames/AutoML_23_20220629_185527_training_py_152_sid_8f55',
'_meta': {'schema_name': 'FrameKeyV3',
'schema_type': 'Key<Frame>',
'schema_version': 3},
'name': 'AutoML_23_20220629_185527_training_py_152_sid_8f55',
'type': 'Key<Frame>'}},
'tweedie_power': {'actual': 1.5, 'default': 1.5, 'input': 1.5},
'validation_frame': {'actual': None, 'default': None, 'input': None},
'weights_column': {'actual': None, 'default': None, 'input': None}}
```

4. Библиотека Dask

4.1. Общее знакомство

По мере роста потребности в распараллеливании алгоритмов машинного обучения, связанной с ростом размеров данных, а также из-за увеличения размеров моделей машинного обучения было бы очень полезно, если бы у нас был инструмент, который мог бы помочь нам выполнить параллельную обработку датафреймов pandas, мог бы без затруднений распараллелить вычисления в Numpy и даже распараллелить алгоритмы машинного обучения (возможно, алгоритмы из библиотек sklearn и tensorflow).

И такой инструмент существует, он называется Dask. Dask – это библиотека для параллельных вычислений, которая не только помогает распараллелить существующие высокоуровневые библиотеки машинного обучения (pandas, Numpy и scikit-learn), но также помогает распараллеливать низкоуровневые задачи/функции и может обрабатывать сложные взаимодействия между этими функциями, создавая граф задач с помощью низкоуровневых планировщиков. Она похожа на модули Python для многопоточных или многопроцессорных вычислений. Библиотеку можно установить с помощью команды `pip install dask`.

Существует также отдельная библиотека машинного обучения dask-ml, которая интегрируется с существующими библиотеками, такими как sklearn, xgboost и tensorflow.

Dask распараллеливает поставленные перед ним задачи, создавая график взаимодействий между ними. Можно визуализировать выполняемые операции с помощью метода Dask `.visualize()`, который поддерживает много типов данных и сложные цепочки выполняемых задач. Этот метод выведет вычислительный график, и если у ваших задач существует много узлов на каждом уровне (то есть структура цепочки ваших задач имеет много независимых ветвлений на разных уровнях типа распараллеливания задачи на фрагменты или чанки данных), то Dask сможет распараллелить их.

Dask предлагает несколько пользовательских интерфейсов:

- **высокоуровневый:**
 - ◆ массив Dask (Dask Array) – распараллеленная версия массива NumPy;
 - ◆ мешок Dask (Dask Bag) – распараллеленная версия списка;
 - ◆ датафрейм Dask (Dask Dataframe) – распараллеленная версия датафрейма pandas;
 - ◆ библиотека dask-ml – распараллеленные алгоритмы машинного обучения;
- **низкоуровневый:**
 - ◆ Delayed – отсроченные параллельные вычисления;
 - ◆ Futures – параллельные вычисления в режиме реального времени.

Массив Dask, мешок Dask и датафрейм Dask являются основными типами данных в Dask. Их еще называют высокоуровневыми коллекциями (high-level collections). Размер этих объектов может быть больше, чем позволяет ваша память, и тогда Dask может распараллелить вычисления на ваших данных «в виде блоков». «В виде блоков» в том смысле, что большие вычисления делятся на

множество небольших блоков, и количество блоков – это общее количество фрагментов (чанков) данных.

4.1.1. Массив Dask (Dask Array)

Множество массивов NumPy в массиве Dask

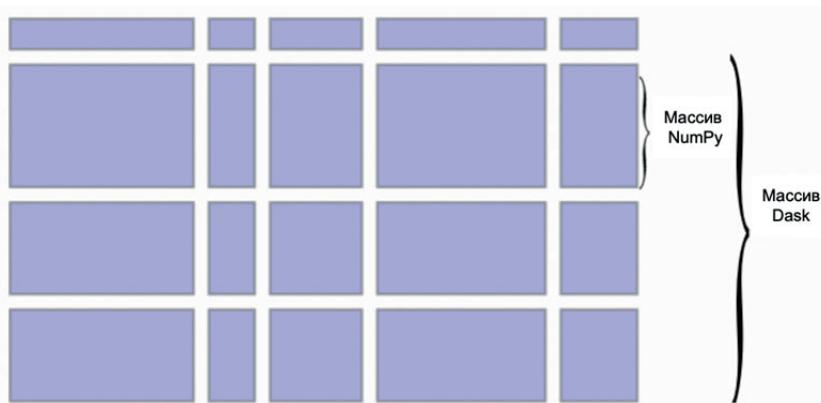


Рис. 9 Массив Dask

Массив Dask нужно использовать, когда ваш массив NumPy много весит (т. е. он не помещается в память) и NumPy ничего не сможет с этим поделать. Массив Dask (Dask Array) состоит из меньших по размеру массивов NumPy. Каждый из этих меньших по размеру объектов, называемый чанком (chunk), можно обрабатывать параллельно в кластере или поставить в очередь и обрабатывать по одной части за раз на локальной машине. К массиву Dask можно применить большую часть функций NumPy <https://docs.dask.org/en/latest/array-api.html>, которые вы можете использовать для ускорения. Однако некоторые функции не реализованы. Например, не реализовано большинство функций модуля `pr.linalg`. Массив Dask не поддерживает такие операции, как `tolist()`, которые были бы очень неэффективными для больших наборов данных. Аналогично очень неэффективно перебирать массив Dask с помощью циклов `for`.

Такую операцию, как сортировка, трудно распараллелить, и она имеет несколько меньшую ценность для очень больших данных (вам редко требуется полная сортировка). Можно воспользоваться более дружественным вариантом для распараллеливания – методом `.topk()`, который возвращает k наибольших элементов из массива по заданной оси и возвращает их в отсортированном виде от наибольшего к наименьшему.

Массив Dask с помощью метода `.from_аггай()` может считывать данные из любой структуры данных типа массива, при условии что она поддерживает формирование срезов, как в NumPy, и имеет свойство `.shape`. Кроме того, он также может прочитать данные из файлов `.пру` и `.загг`.

Давайте импортируем необходимые библиотеки, классы, функции и поработаем с массивом Dask.

```
# импортируем библиотеки, классы, функции
import numpy as np
import pandas as pd
import dask.array as da
import dask.dataframe as dd
import dask.bag as db
import dask.delayed
```

Сейчас создадим массив NumPy со значениями от 0 до 10.

```
# используем функцию np.arange() для создания
# массива NumPy со значениями от 0 до 10
np_arr = np.arange(11)
np_arr
```

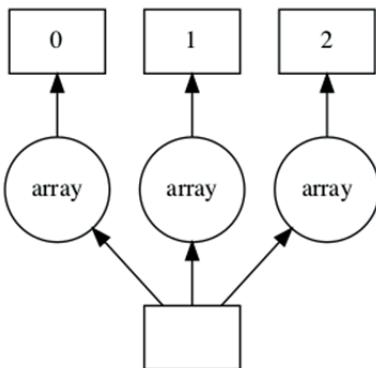
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Преобразовываем массив NumPy в массив Dask с помощью метода `.from_array()`. Задаем массив и размер чанка.

```
# преобразовываем массив NumPy в массив Dask
dask_arr = da.from_array(np_arr, chunks=5)
```

С помощью метода `.visualize()` можем вывести ориентированный ациклический граф вычислений (directed acyclic graph – DAG). Речь идет об ориентированном графе, в котором отсутствуют направленные циклы, но могут быть «параллельные» пути, выходящие из одного узла и разными путями приходящие в конечный узел. На нем мы видим, что в ходе создания массива Dask мы разбили наш массив NumPy на три более мелких массива. Граф состоит из четырех узлов (показаны в виде прямоугольников).

```
# визуализируем граф вычислений
dask_arr.visualize()
```



Теперь взглянем на характеристики массива Dask.

```
# взглянем на характеристики массива Dask
dask_arr
```

У нас приводится следующая информация:

- размер массива (Bytes/Aggay) и размер чанка (Bytes/Chunk), выводится визуализация массива (в нашем случае мы видим одномерный массив из 11 элементов, поделенный на три чанка);
- форма массива (Shape/Аггай) и форма чанка (Shape/Chunk);
- количество узлов соответствующего графа (Count/Аггай) и количество чанков (Count/Chunk);
- тип значений в массиве (Type/Аггай) и тип чанка (Type/Chunk).

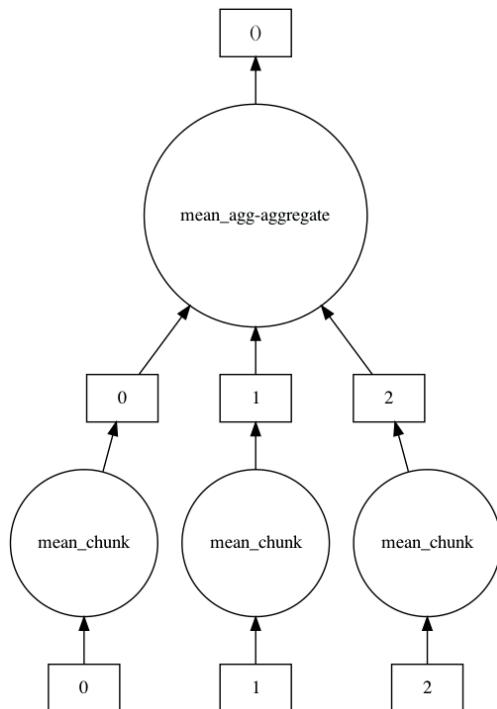
Обратите внимание: Dask выполняет ленивые или отложенные вычисления (lazy computations). Таким образом, чтобы взглянуть на значения массива Dask и получить фактические результаты (например, вычислить среднее значение по элементам массива Dask), необходимо использовать метод `.compute()`. Он вычислит результат параллельно и поблочно.

```
# взглянем на массив
dask_agg.compute()
aggay([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Теперь вычислим среднее и визуализируем график (справа).

```
# вычисляем среднее
dask_agg.mean().compute()
5.0

# визуализируем график вычислений
res = dask_agg.mean()
res.visualize()
```



Как происходят вычисления?
Например, нам нужно вычислить среднее в нашем массиве с 11 значениями. Он состоит из 3 чанков, в двух – 5 значений, в одном – одно значение, вычисляем сумму по каждому чанку (промежуточную сумму) и затем сумму промежуточных сумм.

Посмотрим размер каждого чанка с помощью свойства `.chunks`.

```
# смотрим размер каждого чанка
dask_arr.chunks

((5, 5, 1),)
```

Теперь напрямую создадим массив Dask с помощью функции `da.arange()`.

```
# напрямую создаем массив Dask
dask_arr2 = da.arange(11, chunks=5)
dask_arr2.compute()
```

Давайте сравним вычисление среднего в массиве NumPy (последовательные вычисления) и массиве Dask (параллельные вычисления).

Последовательные вычисления

```
# создаем массив NumPy
x = np.random.rand(50 * 365 * 24 * 60 * 60)
x.shape

(1576800000,)

%%time
```

```
x_sum = x.sum()
x_sum

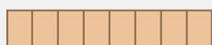
CPU times: user 1.99 s, sys: 1.7 s, total: 3.69 s
Wall time: 3.7 s
```

788404315.1567752

Параллельные вычисления

```
# создаем массив Dask
x_dask = da.from_array(x, chunks=len(x) // 8)
x_dask
```

	Array	Chunk	
Bytes	11.75 GiB	1.47 GiB	
Shape	(1576800000,)	(197100000,)	1576800000
Count	8 Tasks	8 Chunks	
Type	float64	numpy.ndarray	



```
%%time

x_dask_sum = x_dask.sum().compute()
x_dask_sum
```

CPU times: user 3.14 s, sys: 27.2 ms, total: 3.17 s

Wall time: 413 ms

788404315.1567822

Обратите внимание: в этом примере вычисления на массиве Dask заняли 413 миллисекунд, а вычисления на массиве NumPy заняли 3,7 секунды.

На рисунке ниже приводится сравнение скорости вычислений для массива NumPy и массива Dask в зависимости от размера массива.



Рис. 10 Сравнение скорости вычислений в массиве NumPy и массиве Dask

4.1.2. Датафрейм Dask (Dask DataFrame)

Пять датафреймов pandas, каждый из которых представляет ежемесячные данные (могут быть взяты из различных файлов) в одном датафрейме Dask

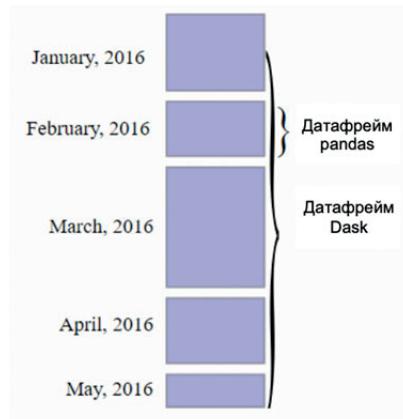


Рис. 11 Датафрейм Dask

Датафрейм Dask (Dask DataFrame) состоит из меньших по размеру датафреймов pandas. Как и в случае с массивом Dask, каждый из этих меньших по размеру объектов, называемый партицией (partition), можно обрабатывать параллельно в кластере, или поставить в очередь и обрабатывать по одной части за раз на локальной машине. Вы можете применить большинство методов библиотеки pandas <https://docs.dask.org/en/latest/dataframe-api.html>.

Тривиально распараллеливаемые операции (быстрые):

- элементарные операции: `df.x + df.y`, `df * df`;
- отбор строк: `df[df.x > 0]`;
- индексатор `loc`: `df.loc[4.0: 10.5]`;
- общие агрегирующие функции: `df.x.max()`, `df.max()`;
- работа со спископодобными объектами: `df[df.x.isin([1, 2, 3])]`.

Сложно распараллеливаемые операции (быстрые):

- `groupby-aggregate` с агрегирующими функциями: `df.groupby(df.x).y.max()`, `df.groupby('x').max()`;
- `groupby-apply` по индексу: `df.groupby(['idx', 'x']).apply(myfunc)`, где `idx` – название уровня индекса;
- вывод частот с помощью `.value_counts()`: `df.x.value_counts()`;
- удаление дублей: `df.x.drop_duplicates()`;
- объединение по индексу: `dd.merge(df1, df2, left_index=True, right_index=True)` или `dd.merge(df1, df2, on=['idx', 'x'])`, где `idx` – это название индекса для обоих `df1` и `df2`;
- поэлементные операции с разными партициями: `df1.x + df2.y`;
- вычисление скользящих средних: `df.rolling(...)`;
- коэффициент корреляции Пирсона: `df[['col1', 'col2']].corr()`.

Операции, требующие перемешивания данных (медленные, кроме операций по индексу):

- установка индекса: `df.set_index(df.x)`;
- `groupby-aggregate` не по индексу: `df.groupby(df.x).apply(myfunc)`;
- объединение не по индексу: `dd.merge(df1, df2, on='name')`.

Однако следует помнить, что операции, которые выполнялись медленно в pandas, типа построчного итерирования, остаются медленными и в датафреймах Dask.

Давайте с помощью функции `dd.read_csv()` создадим на основе CSV-файла датафрейм Dask и с помощью метода `.head()` выведем первые два наблюдения. Здесь нам метод `.compute()` не потребуется, поскольку вывод результатов метода `.head()` не предполагает интенсивных вычислений.

```
# создаем датафрейм Dask на основе CSV-файла
dask_df = dd.read_csv('Data/StateFarm.csv', sep=';')
# выведем первые два наблюдения
dask_df.head(2)
```

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	18975.456110	65999	237	1	14	0	6	0
1	4715.321344	0	65	19	56	0	3	0

А с помощью функции `dd.from_pandas()` можно превратить датафрейм pandas в датафрейм Dask.

```
# создаем датафрейм pandas на основе CSV-файла
df = pd.read_csv('Data/StateFarm.csv', sep=';')
# превращаем датафрейм pandas в датафрейм Dask
dask_df2 = dd.from_pandas(df, npartitions=2)
```

4.1.3. Мешок Dask (Dask Bag)

Мешок (Bag) – это математическое название для неупорядоченной коллекции, допускающей включение одного и того же элемента в совокупность по несколько раз. По сути, это синоним мульти множества. В отличие от множества, мульти множество допускает повторяющиеся элементы:

- список (list): *упорядоченная коллекция с повторяющимися элементами*, [1, 2, 3, 2];
- множество (set): *неупорядоченная коллекция без повторяющихся элементов*, {1, 2, 3};
- мешок (bag): *неупорядоченная коллекция с повторяющимися элементами*, {1, 2, 2, 3}.

Мешки Dask выполняют параллельные вычисления со списками Python, как с объектами, содержащими элементы различных типов данных. Это полезно, когда вы пытаетесь обработать некоторые полуструктуренные данные типа JSON-файлов или логов. Мешок Dask позволяет применить к коллекциям объектов Python такие операции, как map, filter, fold и groupby. Это делается параллельно, используя небольшое количество памяти с помощью итераторов Python. Это похоже на распараллеленную или питоновскую версию PySpark RDD.

Давайте с помощью функции db.from_sequence() создадим на основе списка мешок Dask.

```
# создаем мешок Dask на основе списка
dask_bag = db.from_sequence([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], npartitions=2)
# берем три элемента
dask_bag.take(3)

(1, 2, 3)
```

4.1.4. Интерфейс Delayed

Иногда мы сталкиваемся с проблемами, которые можно распараллелить, но они не вписываются в абстракции высокого уровня типа массива Dask или датафрейма Dask. Рассмотрим следующий пример:

```
# создаем функции
def inc(x):
    return x + 1

def double(x):
    return x + 2

def add(x, y):
    return x + y
# создаем список
data = [1, 2, 3, 4, 5]
# создаем пустой список
output = []
```

```
# запускаем цикл
for x in data:
    a = inc(x)
    b = double(x)
    c = add(a, b)
    output.append(c)
# получаем результат
total = sum(output)
```

В этой задаче явно есть параллелизм (функции `inc()`, `double()` и `add()` можно вызвать независимо), но не ясно, как преобразовать результат в большой массив или большой датафрейм.

Этот код выполняется последовательно в одном потоке. Однако мы видим, что многое в нем можно распараллелить.

Функция `delayed()` библиотеки Dask модифицирует ваши функции так, чтобы они работали лениво. Вместо немедленного выполнения вашей функции она будет откладывать выполнение, помещая функции и ее аргументы в граф задач для последующих параллельных вычислений с помощью метода `.compute()`.

Мы немного модифицируем наш код, обернув функции в `delayed()`. Это вызовет отсрочку выполнения функций и сгенерирует график.

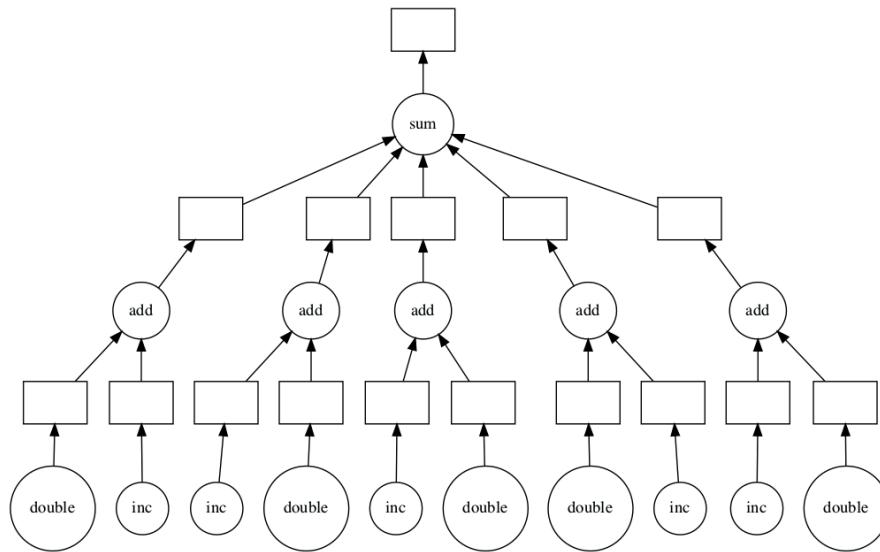
```
# модифицируем код, обернув в delayed()
output = []
for x in data:
    a = dask.delayed(inc)(x)
    b = dask.delayed(double)(x)
    c = dask.delayed(add)(a, b)
    output.append(c)

total = dask.delayed(sum)(output)
```

Мы использовали функцию `delayed()`, чтобы обернуть вызовы функций, которые мы хотим превратить в задачи. Ни одного вызова `inc()`, `double()`, `add()` или `sum()` еще не было. Вместо этого итоговое значение объекта представляет собой отсроченный результат, который представляет собой график вычислений. Глядя на график, мы видим четкие возможности для параллельных вычислений. Планировщики Dask будут использовать этот параллелизм, обычно улучшая производительность (хотя не в этом примере, потому что мы используем очень маленькие и быстрые функции).

Давайте визуализируем график вычислений.

```
# визуализируем график вычислений
total.visualize()
```



Теперь получаем результат, применяя параллельные вычисления.

получаем результат, применяя параллельные вычисления
total.compute()

45

Кроме того, можно часто увидеть функцию `delayed()` в качестве декоратора. Ниже наша исходная задача воспроизведена в виде параллельного кода:

```

# используем delayed() в качестве декоратора
@dask.delayed
def inc(x):
    return x + 1
@dask.delayed
def double(x):
    return x + 2
@dask.delayed
def add(x, y):
    return x + y
data = [1, 2, 3, 4, 5]
output = []
for x in data:
    a = inc(x)
    b = double(x)
    c = add(a, b)
    output.append(c)
total = dask.delayed(sum)(output)
  
```

4.2. МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ БИБЛИОТЕКИ DASK-ML

У Dask есть библиотека `dask-ml` (устанавливается командой `pip install dask-ml`), которая помогает распараллеливать популярные библиотеки машинного обучения типа `sklearn`, `tensorflow` и `xgboost`.

В машинном обучении вы можете столкнуться с различными задачами масштабирования. Стратегия масштабирования зависит от того, с какой проблемой вы столкнулись:

- 1) большие модели: данные помещаются в оперативную память, но обучение занимает слишком много времени. Множество комбинаций гиперпараметров, большой ансамбль из множества моделей и т. д.;
- 2) большие наборы данных: данные не помещаются в RAM, и создание выборки невозможно. Итак, вы можете:
 - ◆ для задач, которые помещаются в оперативную память, просто использовать Dask и вашу любимую модель scikit-learn или любимую библиотеку ML (например, LightGBM);
 - ◆ для больших моделей использовать `dask_ml.joblib` и вашу любимую модель scikit-learn;
 - ◆ для больших наборов данных использовать библиотеку `dask_ml`.

Давайте подробнее познакомимся с библиотекой `dask-ml`.

Модуль `dask_ml.preprocessing` содержит некоторые клоны классов библиотеки scikit-learn: `StandardScaler`, `MinMaxScaler`, `RobustScaler`, `LabelEncoder`, `OneHotEncoder`, `PolynomialFeatures` и т. д., а также некоторые собственные классы типа `Categorizer`, `DummyEncoder`, `OrdinalEncoder` и т. д. Вы можете использовать их точно так же, как при работе с датафреймами pandas.

Если в наборе данных есть категориальные переменные, а модель требует, чтобы все переменные были количественными (например, строим логистическую регрессию), этим переменным необходимо с помощью класса `Categorizer` присвоить тип `Categorical`, указав список переменных, а затем выполнить дамми-кодирование с помощью класса `DummyEncoder`, который работает аналогично функции `get_dummies()` библиотеки pandas (самостоятельно определяет, какие переменные имеют тип `Categorical`, и выполняет дамми-кодирование).

Модуль `dask_ml.impute` содержит класс `SimpleImputer` для импутации пропусков.

Модуль `dask_ml.model_selection` содержит функцию `train_test_split()` для разбиения набора на обучающую и тестовую выборки, классы `ShuffleSplit()` и `KFold()` для осуществления проверки, традиционные классы `GridSearchCV` и `RandomizedSearchCV` для оптимизации гиперпараметров в виде обычного поиска по сетке и случайного поиска по сетке соответственно, а также специальные классы `SuccessiveHalvingSearchCV`, `IncrementalSearchCV` и `HyperbandSearchCV` для оптимизации гиперпараметров на больших наборах данных.

Например, класс `SuccessiveHalvingSearchCV` работает следующим образом. Мы задаем порог – `n_initial_iter` итераций для обучения наших моделей. Когда порог достигнут, убираем 1/2 (т. е. половину) моделей, которые показали худшее качество, и удваиваем порог, повторяя до тех пор, пока не останется одна модель, либо задаем значение `max_iter`. Смысл такой стратегии заключается в том, что по мере исключения явно плохих моделей у нас остается больше ресурсов, чтобы исследовать потенциально хорошие модели.

Вышеприведенное значение 1/2 дано для ясности объяснения. На самом деле мы убираем процент худших моделей, вычисленный по формуле $1 - 1/\text{aggressiveness}$. По умолчанию значение `aggressiveness` равно 3, поэтому

му процент отклоняемых моделей будет составлять 0,66. Высокие значения `agressiveness` подразумевают более высокую уверенность в полученных оценках качества моделей (или то, что гиперпараметры влияют на оценку качества модели сильнее, чем данные).

Модуль `dask_ml.compose` содержит класс `ColumnTransformer` для обработки смешанных данных в конвейере. Обратите внимание, что для построения конвейеров в Dask используется класс `Pipeline` библиотеки scikit-learn.

Модуль `dask_ml.metrics` содержит функции `mean_absolute_error`, `mean_squared_error`, `r2_score`, `accuracy_score` и `log_loss`.

Модуль `dask_ml.linear_model` содержит классы `LinearRegression`, `LogisticRegression` и `PoissonRegression`. Модуль `dask_ml.xgboost` содержит классы `XGBClassifier` и `XGBRegressor`.

Давайте потренируемся использовать классы библиотеки scikit-learn и классы библиотеки dask-ml в Dask. Сначала импортируем необходимые библиотеки, классы и функции.

```
# загружаем необходимые библиотеки, классы, функции
import numpy as np
import pandas as pd
import dask.dataframe as dd
from sklearn.metrics import roc_auc_score, accuracy_score
from dask_ml.impute import SimpleImputer
from dask_ml.preprocessing import (StandardScaler,
                                  DummyEncoder,
                                  Categorizer)
from dask_ml.model_selection import train_test_split
from dask_ml.linear_model import LogisticRegression
```

Теперь прочитаем CSV-файл, содержащий расширенные данные компании StateFarm, в датафрейм Dask и взглянем на него.

```
# считываем CSV-файл в датафрейм Dask и смотрим
df_dask = dd.read_csv('Data/StateFarm_missing.csv', sep=';')
df_dask
```

Dask DataFrame Structure:

Customer Lifetime Value	Coverage	Education	EmploymentStatus	Gender	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
npartitions=1											
float64	object	object	object	object	float64	float64	float64	float64	float64	float64	object
...

Dask Name: from-delayed, 3 tasks

Убедимся в том, что работаем с датафреймом Dask.

```
# убедимся в том, что работаем с датафреймом Dask
type(df_dask)
```

```
dask.dataframe.core.DataFrame
```

Посмотрим количество наблюдений и количество переменных.

```
# посмотрим форму датафрейма Dask с помощью
# функции len() и метода .count()
len(df_dask), len(df_dask.count())
```

(8293, 12)

С помощью метода `.describe()` можно взглянуть на статистики количественных переменных, однако помним про ленивые вычисления и еще применяем метод `.compute()`.

```
# смотрим статистики, замечаем, что многие переменные
# имеют нормальное распределение (практически одинаковые
# значения средних и медиан)
df_dask.describe().compute()
```

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies
count	8289.000000	8291.000000	8282.000000	8288.000000	8285.000000	8287.000000	8288.000000
mean	7987.650889	37785.171994	93.198865	15.079875	48.138081	0.389767	2.964768
std	6841.535432	30396.251967	34.514287	10.093847	27.827103	0.915515	2.389635
min	1898.007675	0.000000	61.000000	0.000000	0.000000	0.000000	1.000000
25%	3982.180708	0.000000	68.000000	6.000000	24.000000	0.000000	1.000000
50%	5786.493980	34220.000000	83.000000	14.000000	48.000000	0.000000	2.000000
75%	8960.280213	62450.000000	109.000000	23.000000	71.000000	0.000000	4.000000
max	8325.381190	99981.000000	298.000000	35.000000	99.000000	5.000000	9.000000

С помощью свойства `.columns` и метода `.tolist()` можно взглянуть на список имён переменных.

```
# смотрим названия переменных
print(df_dask.columns.tolist())
```

```
['Customer Lifetime Value', 'Coverage', 'Education', 'EmploymentStatus', 'Gender', 'Income',
'Monthly Premium Auto', 'Months Since Last Claim', 'Months Since Policy Inception', 'Number
of Open Complaints', 'Number of Policies', 'Response']
```

Теперь посмотрим количество пропусков по каждой переменной.

```
# смотрим количество пропусков
# по каждой переменной
df_dask.isnull().compute().sum()
```

Customer Lifetime Value	4
Coverage	5
Education	3
EmploymentStatus	5
Gender	4
Income	2
Monthly Premium Auto	11
Months Since Last Claim	5
Months Since Policy Inception	8
Number of Open Complaints	6
Number of Policies	5
Response	0
dtype:	int64

С помощью свойства `.dtypes` посмотрим типы переменных.

```
# смотрим типы переменных  
df_dask.dtypes
```

```
Customer Lifetime Value      float64  
Coverage           object  
Education          object  
EmploymentStatus   object  
Gender             object  
Income              float64  
Monthly Premium Auto    float64  
Months Since Last Claim float64  
Months Since Policy Inception float64  
Number of Open Complaints float64  
Number of Policies     float64  
Response            object  
dtype: object
```

Создаем список категориальных переменных, исключив зависимую переменную *Response*.

```
# создаем список категориальных переменных  
cat_columns = df_dask.select_dtypes(  
    include='object').columns.difference(['Response']).tolist()  
# смотрим список  
cat_columns
```

```
['Coverage', 'Education', 'EmploymentStatus', 'Gender']
```

Посмотрим статистики по категориальным переменным.

```
# смотрим статистики по категориальным переменным  
df_dask[cat_columns].describe().compute()
```

	Coverage	Education	EmploymentStatus	Gender
unique	3	5	5	2
count	8288	8290	8288	8289
top	Basic	Bachelor	Employed	F
freq	5038	2496	5187	4250

С помощью класса `Categorizer` категориальным переменным присваиваем тип `Categorical`, используя ранее созданный список категориальных переменных.

```
# присваиваем переменным типа object тип Categorical  
cat = Categorizer(columns=cat_columns)  
df_dask = cat.fit_transform(df_dask)
```

Строковые значения *No* и *Yes* зависимой переменной *Response* преобразовываем в целочисленные значения 0 и 1.

```
# строковые значения No и Yes переводим
# в целочисленные 0 и 1
dct = {'No': 0, 'Yes': 1}
df_dask['Response'] = df_dask['Response'].replace(dct)
```

Теперь выведем частоты категорий по категориальным переменным.

```
# выведем частоты категорий по категориальным переменным
for col in cat_columns:
    print(df_dask[col].value_counts().compute())
```

```
Basic      5038
Extended   2501
Premium    749
Name: Coverage, dtype: int64
Bachelor    2496
College     2421
High School or Below  2397
Master      659
Doctor      317
Name: Education, dtype: int64
Employed    5187
Unemployed  2095
Medical Leave 392
Disabled    362
Retired     252
Name: EmploymentStatus, dtype: int64
F        4250
M        4039
Name: Gender, dtype: int64
```

Формируем массив признаков и массив меток.

```
# формируем массив признаков и массив меток
y_dask = df_dask.pop('Response')
```

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток. Для этого воспользуемся функцией `train_test_split()` библиотеки dask-ml.

```
# создаем обучающий массив признаков, обучающий массив меток,
# тестовый массив признаков, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    df_dask,
    y_dask,
    test_size=.3,
    shuffle=True,
    random_state=42)
```

Создаем список количественных переменных.

```
# создаем список количественных столбцов
num_columns = X_train.select_dtypes(include='number').columns.tolist() num_columns
```

```
['Customer Lifetime Value', 'Income',
 'Monthly Premium Auto', 'Months Since Last Claim',
 'Months Since Policy Inception', 'Number of Open Complaints', 'Number of Policies']
```

Выполним импутацию пропусков с помощью класса `SimpleImputer` библиотеки `dask-ml`.

```
# создаем экземпляр класса SimpleImputer
# для количественных переменных
num_imputer = SimpleImputer(strategy='median')
# обучаем модель
num_imputer.fit(X_train[num_columns])
# выполняем импутацию пропусков в количественных переменных
X_train[num_columns] = num_imputer.transform(X_train[num_columns])
X_test[num_columns] = num_imputer.transform(X_test[num_columns])

# создаем экземпляр класса SimpleImputer
# для категориальных переменных
cat_imputer = SimpleImputer(strategy='most_frequent')
# обучаем модель
cat_imputer.fit(X_train[cat_columns])
# выполняем импутацию пропусков в категориальных переменных
X_train[cat_columns] = cat_imputer.transform(X_train[cat_columns])
X_test[cat_columns] = cat_imputer.transform(X_test[cat_columns])
```

Проверим наличие пропусков.

```
# проверяем наличие пропусков в
# обучающей и тестовой выборках
print(X_train.isnull().compute().sum().sum())
print(X_test.isnull().compute().sum().sum())
```

```
0
0
```

Теперь выполним стандартизацию количественных переменных.

```
# создаем экземпляр класса StandardScaler
scaler = StandardScaler()
# обучаем модель
scaler.fit(X_train[num_columns])
# выполняем стандартизацию
X_train[num_columns] = scaler.transform(X_train[num_columns])
X_test[num_columns] = scaler.transform(X_test[num_columns])
```

Выполняем дамми-кодирование категориальных переменных.

```
# создаем экземпляр класса DummyEncoder
dum = DummyEncoder()
# обучаем модель
dum.fit(X_train)
# выполняем дамми-кодирование
X_train = dum.transform(X_train)
X_test = dum.transform(X_test)
```

Взглянем на результаты дамми-кодирования.

```
# взглянем на результаты
```

```
X_train.head()
```

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Inception	Number of Open Complaints	Number of Policies	Coverage_Basic	Coverage_Premium	Coverage_Extended
0	-0.768590	0.614122	-0.314656	1.679131	-1.552611	-0.429144	-0.821629	1	0	0
1	-0.317944	-1.235386	-0.314656	-0.205514	-0.223066	-0.429144	-0.402818	1	0	0
2	-0.317944	0.367396	0.457056	-0.106322	-0.366800	-0.429144	-0.402818	1	0	0
3	-0.033898	-1.235386	0.397694	0.290445	-0.007464	-0.429144	1.691241	1	0	0
10	-0.471322	-1.235386	-0.314656	-0.106322	-1.552611	-0.429144	0.015994	1	0	0

Теперь обучаем модель логистической регрессии с помощью класса `LogisticRegression` библиотеки `dask-ml` и оцениваем правильность на тестовой выборке с помощью метода `.score()`. Класс `LogisticRegression` работает только с массивами Dask (для преобразования датафрейма Dask в массив Dask можно воспользоваться методом `.to_dask_aggregate()`). Для массивов Dask с неизвестными размерами чанков строится модель без константы (`fit_intercept=False`).

```
# обучаем модель (для массива Dask с неизвестными размерами чанков
# строится модель без константы) и оцениваем правильность на
# тестовой выборке с помощью метода .score()
logreg = LogisticRegression(fit_intercept=False, n_jobs=-1)
logreg.fit(X_train.to_dask_array(lengths=True),
           y_train.to_dask_array(lengths=True))
logreg.score(X_test.to_dask_array(lengths=True),
             y_test.to_dask_array(lengths=True)).compute()
```

```
0.8986623429266315
```

Теперь построим модель с константой (`fit_intercept=True`).

```
# создаем массивы Dask с известными размерами чанков
X_train_with_known_chunks = X_train.to_dask_array(lengths=True) y_train_with_known_
chunks = y_train.to_dask_array(lengths=True) X_test_with_known_chunks = X_test.to_dask_
array(lengths=True) y_test_with_known_chunks = y_test.to_dask_array(lengths=True)

# обучаем модель и оцениваем правильность на тестовой выборке
# с помощью метода .score()
logreg2 = LogisticRegression(fit_intercept=True, n_jobs=-1) logreg2.fit(X_train_with_known_
chunks, y_train_with_known_chunks) logreg2.score(X_test_with_known_chunks, y_test_with_
known_chunks).compute()
```

```
0.8946528332003192
```

Теперь вычислим правильность и AUC с помощью функций `accuracy_score()` и `roc_auc_score()` библиотеки `scikit-learn`.

```
# оцениваем правильность с помощью функции accuracy_score
# библиотеки scikit-learn
pred = logreg.predict(X_test.to_dask_array(
    lengths=True)).compute()
```

```
accuracy_score(y_test, pred)  
0.8986623429266315  
  
# оцениваем AUC с помощью функции roc_auc_score  
# библиотеки scikit-learn  
proba = logreg.predict_proba(  
    X_test.to_dask_array(lengths=True)).compute()  
roc_auc_score(y_test, proba[:, 1])  
  
0.6010430736905298
```

4.3. ПОСТРОЕНИЕ КОНВЕЙЕРА В DASK

Теперь посмотрим, как в Dask можно построить конвейеры. Воспользуемся тем же самым набором. Импортируем необходимые библиотеки, классы и функции и прочитываем расширенные данные компании StateFarm в датафрейм Dask.

```
# импортируем необходимые библиотеки, классы и функции  
import numpy as np  
import pandas as pd  
import dask.dataframe as dd  
from dask_ml.impute import SimpleImputer  
from dask_ml.preprocessing import (StandardScaler,  
                                    DummyEncoder,  
                                    Categorizer)  
from pandas.api.types import CategoricalDtype  
from sklearn.compose import ColumnTransformer  
from sklearn.pipeline import Pipeline  
from dask_ml.linear_model import LogisticRegression  
from sklearn.metrics import accuracy_score  
from dask_ml.model_selection import train_test_split  
# загружаем данные  
df_dask = dd.read_csv('Data/StateFarm_missing.csv', sep=';')
```

Переименовываем метки зависимой переменной в целочисленные значения, создаем массив меток, разбиваем набор на обучающую и тестовую выборки.

```
# переименовываем метки зависимой переменной  
# в целочисленные значения  
df_dask['Response'] = df_dask['Response'].replace({'No': 0, 'Yes': 1})  
# создаем массив меток  
y_dask = df_dask.pop('Response')  
  
# разбиваем данные на обучающую и тестовую выборки  
X_train, X_test, y_train, y_test = train_test_split(  
    df_dask,  
    y_dask,  
    test_size=0.3,  
    shuffle=True,  
    random_state=42)
```

Теперь выполняем уже хорошо знакомые нам операции: создаем списки количественных и категориальных переменных, создаем трансформеры – отдельные конвейеры для переменных разного типа, создаем список трансфор-

меров, передаем этот список в `ColumnTransformer` и создаем итоговый конвейер. Однако есть важный момент. Если у нас есть категориальные признаки, мы должны присвоить им тип `Categorical`, указав для каждой переменной список категорий. Это можно сделать с помощью класса `Categorizer` библиотеки `dask-ml`. Для выполнения дамми-кодирования можно применить класс `DummyEncoder` библиотеки `dask-ml`. Для обработки пропусков в количественных признаках воспользуемся классом `SimpleImputer` библиотеки `dask-ml`. Пропуски в категориальных признаках запишем в отдельную категорию.

```
# создаем список количественных переменных
number = X_train.select_dtypes(include='number').columns.tolist()
number

['Customer Lifetime Value',
 'Income',
 'Monthly Premium Auto',
 'Months Since Last Claim',
 'Months Since Policy Inception',
 'Number of Open Complaints',
 'Number of Policies']

# создаем список категориальных переменных
categ = X_train.select_dtypes(include='object').columns.tolist()
categ

['Coverage', 'Education', 'EmploymentStatus', 'Gender']

# смотрим уникальные значения категориальных переменных
for col in categ:
    print(X_train[col].unique().compute())
    print("")

0      Basic
1    Extended
2    Premium
3      NaN
Name: Coverage, dtype: object

0          College
1        Bachelor
2  High School or Below
3          Master
4          NaN
5        Doctor
Name: Education, dtype: object

0   Medical Leave
1      Employed
2    Unemployed
3     Retired
4    Disabled
5      NaN
Name: EmploymentStatus, dtype: object

0      M
1      F
2      NaN
Name: Gender, dtype: object
```

```
# выделим пропуски в отдельную категорию
for col in categ:
    X_train[col] = X_train[col].astype(str)

# смотрим уникальные значения категориальных переменных
for col in categ:
    print(X_train[col].unique().compute())
    print("")

0      Basic
1    Extended
2   Premium
3       nan
Name: Coverage, dtype: object

0          College
1        Bachelor
2  High School or Below
3           Master
4           nan
5         Doctor
Name: Education, dtype: object

0   Medical Leave
1     Employed
2   Unemployed
3     Retired
4     Disabled
5           nan
Name: EmploymentStatus, dtype: object

0      M
1      F
2     nan
Name: Gender, dtype: object

# создаем списки категорий
coverage_lst = sorted(X_train['Coverage'].unique().compute().tolist())
educ_lst = sorted(X_train['Education'].unique().compute().tolist())
empl_lst = sorted(X_train['EmploymentStatus'].unique().compute().tolist())
gender_lst = sorted(X_train['Gender'].unique().compute().tolist())

# задаем для каждого категориального признака списки категорий
categories = {'Coverage': CategoricalDtype(coverage_lst, ordered=False),
              'Education': CategoricalDtype(educ_lst, ordered=False),
              'EmploymentStatus': CategoricalDtype(empl_lst, ordered=False),
              'Gender': CategoricalDtype(gender_lst, ordered=False)}

# создаем трансформеры
num_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('scaler', StandardScaler())
])

cat_pipe = Pipeline([
    ('categ', Categorizer(categories=categories)),
    ('dum', DummyEncoder())
])
```

```
# создаем список трансформеров
transformers = [('num', num_pipe, number),
                ('cat', cat_pipe, categ)]

# передаем список в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)

# создаем итоговый конвейер
ml_pipe = Pipeline([('tf', transformer),
                    ('logreg', LogisticRegression(
                        fit_intercept=False,
                        n_jobs=-1))])
```

Наконец, обучаем конвейер и вычисляем правильность на тестовой выборке.

```
# обучаем конвейер
ml_pipe.fit(X_train, y_train)
# получаем прогнозы для тестовой выборки
pred = ml_pipe.predict(X_test)
# смотрим правильность на тестовой выборке
accuracy_score(y_test, pred)
```

0.8986623429266315

5. Google Colab

5.1. Общее знакомство

Google Colab – это бесплатный облачный сервис на основе Jupyter Notebook. Google Colab предоставляет всё необходимое для машинного обучения прямо в браузере, даёт бесплатный доступ к быстрым GPU и TPU. Он поддерживает Python 3 из-под коробки (Python 2 больше не поддерживается).

С технической точки зрения Colab – это размещенная на хосте служба Jupyter для ноутбуков, которая не требует настройки для использования, но при этом предоставляет бесплатный доступ к вычислительным ресурсам, включая графические процессоры.

Colab позволяет использовать записные книжки Jupyter и делиться ими с другими без необходимости загружать, устанавливать или запускать что-либо.

Перечислим текущие основные ограничения использования Google Colab.

Через некоторое время вашу виртуальную машину Colab удалят (подробнее об этом на сайте [Google Colab](#)) и выполнение любого кода прервется. После этого вам выделят новую совершенно чистую виртуальную машину.

В Colab время работы блокнотов может составлять не более 12 часов и так же есть отключение по причине бездействия. Таким образом, максимальное время жизни экземпляра Colab составляет 12 часов.

Ноутбуки Google Colab так же имеют тайм-аут простоя 90 минут и абсолютный тайм-аут 12 часов. Это означает, что если пользователь не взаимодействует со своим ноутбуком Google Colab более 90 минут, экземпляр прекращает свою работу автоматически.

Кроме того, ресурсы, которые Google Colab предоставляет вам, не гарантированы и не безграничны, а лимиты использования иногда колеблются. Это необходимо, чтобы Colab мог бесплатно предоставлять ресурсы.

В Colab приоритетный доступ к ресурсам предоставляется тем, кто в последнее время использовал меньше ресурсов. Это позволяет предоставлять ограниченные ресурсы более широкому кругу пользователей. Чтобы максимально эффективно использовать Colab, закрывайте вкладки Colab, с которыми вы больше не работаете. Страйтесь также не использовать графические процессоры или дополнительную память, если они не нужны вам для текущей работы, все это учитывается. Так вы будете меньше сталкиваться с лимитами на использование в Colab.

5.2. Регистрация и создание папки проекта

Наберите в адресной строке браузера <https://colab.research.google.com/> – и вас перебросит на начальную страницу регистрации в Colab, где вы сможете авторизоваться под своим Google-аккаунтом.

После авторизации откроется стартовая страница.

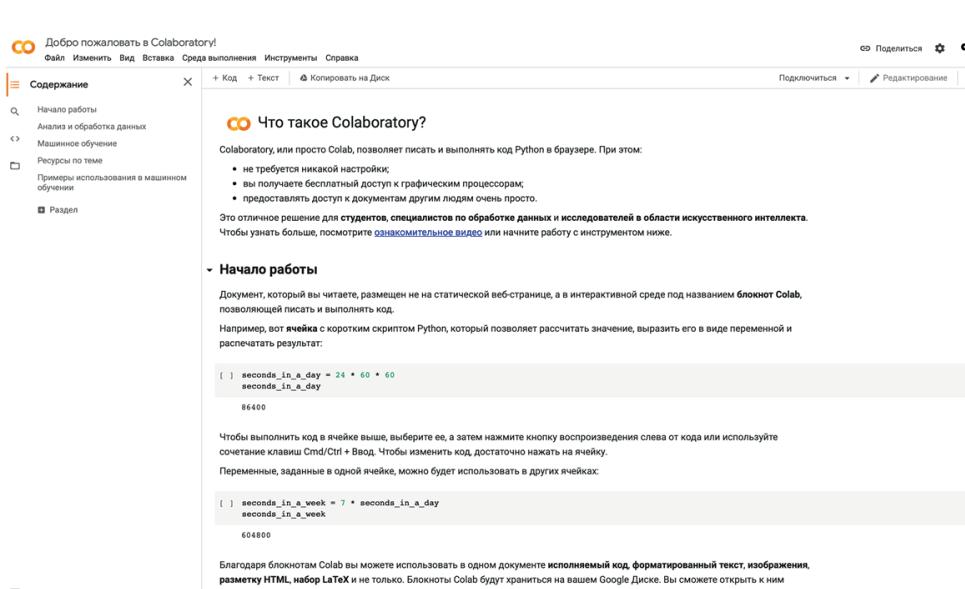
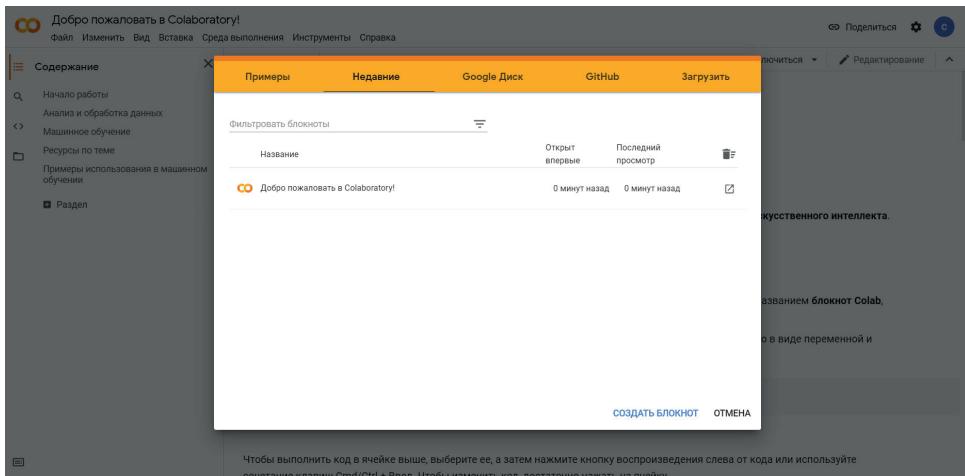
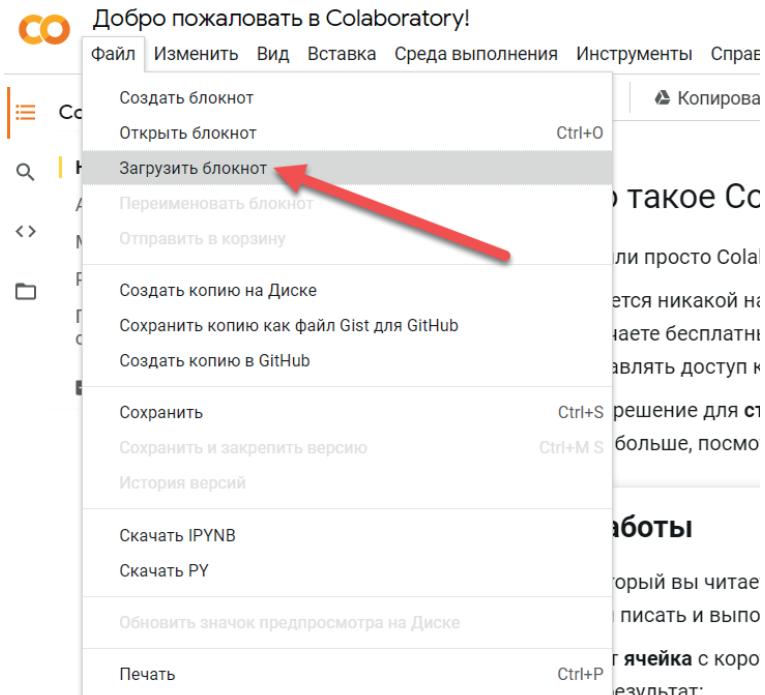


Рис. 12 Стартовая страница Google Colab

Здесь можно, например, создать блокнот (тетрадку) Jupyter с нуля или загрузить его туда готовый. Чтобы в Colab импортировать существующие блокноты Jupyter, выберите **Загрузить блокнот** в меню **Файл**.



В следующем окне вы можете загрузить свой ноутбук в Colab.

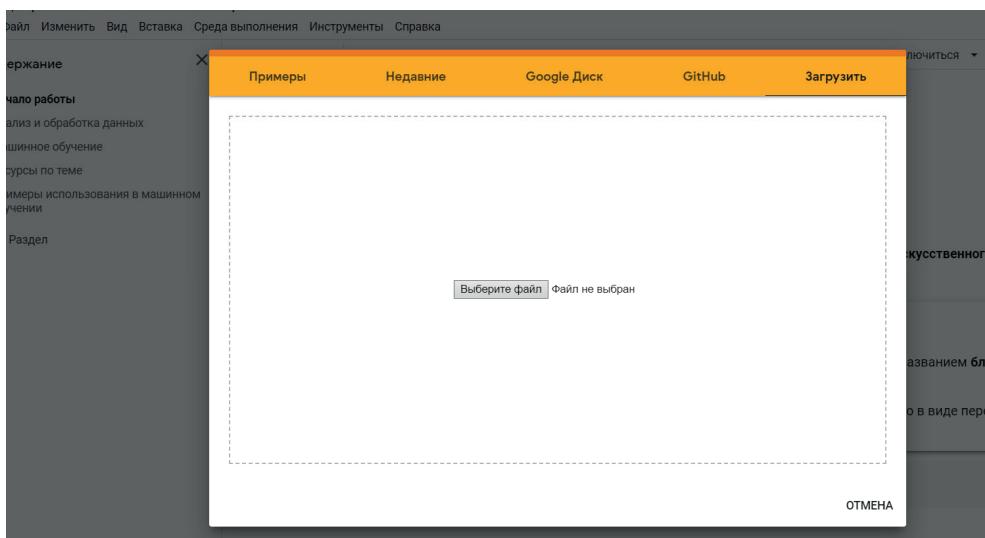


Рис. 13 Окно загрузки блокнота

Блокноты Colab хранятся на [Google Диске](#) или могут быть загружены с [GitHub](#). Ноутбуками Colab можно делиться так же, как Документами или Таблицами Google.

Таким образом, после загрузки в вашем Google-диске в папке Colab Notebooks будет виден ваш загруженный блокнот, например так:

Название ↑	Владелец	Последнее изменение	Размер файла
Собственный класс ClassicStacking.ipynb	я	12:20 я	25 КБ

В этой же папке Google-диска можно, как обычно, создавать различные подпапки и класть туда свои блокноты и данные для них. Поэтому можно начинать свою работу с Google-диска, создавая там нужную структуру папок и раскладывая по ним нужные ноутбуки и данные для них.

Сейчас мы на Google-диске создадим папку *Stacking*, в ней будет находиться модуль *classic_stacking.py*, блокнот *Собственный класс ClassicStacking.ipynb*, папка *Data* с наборами данных *ottogroup_train.csv* и *ottogroup_test.csv*. Все необходимые файлы находятся в подпапке *Для загрузки на Google-диск* папки *Часть 5_5.1.-5.3._Google Colab*.

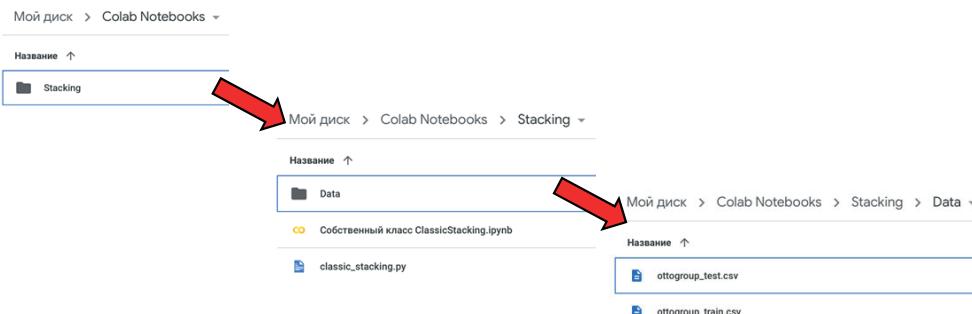


Рис. 14 Содержимое папки Stacking

Давайте откроем блокнот *Собственный класс ClassicStacking.ipynb* с Google-диска, щелкнув по нему два раза мышкой.

The screenshot shows the Google Colab interface with a notebook titled 'Собственный класс ClassicStacking.ipynb'. The menu bar includes 'Файл', 'Изменить', 'Вид', 'Вставка', 'Среда выполнения', 'Инструменты', 'Справка', and 'Изменения сохранены'. The code cell contains the following Python code:

```
[ ] # импортируем необходимую библиотеку pandas
import pandas as pd

[ ] # импортируем класс CatBoostClassifier из библиотеки catboost
from catboost import CatBoostClassifier

[ ] # импортируем самописный модуль classic_stacking
from classic_stacking import *

[ ] # загружаем исторический набор и набор новых данных
train = pd.read_csv('Data/ottogroup_train.csv')
test = pd.read_csv('Data/ottogroup_test.csv')
```

Рис. 15 Открытие ноутбука в Google Colab

Однако у нас возникнут ошибки при импорте классов из некоторых библиотек, если они не установлены (например, при импорте класса `CatBoostClassifier` библиотеки `catboost` возникнет ошибка `ModuleNotFoundError: No module named 'catboost'`).

The screenshot shows a code cell with an error message. The code is:

```
[2] # импортируем класс CatBoostClassifier из библиотеки catboost
from catboost import CatBoostClassifier
```

The error message is:

```
ModuleNotFoundError: Traceback (most recent call last)
<ipython-input-2-da0d4f45133a> in <module>
      1 # импортируем класс CatBoostClassifier из библиотеки catboost
      2 from catboost import CatBoostClassifier

ModuleNotFoundError: No module named 'catboost'
```

Below the error message, there is a note:

```
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.
```

And a link to examples:

```
To view examples of installing some common dependencies, click the
"Open Examples" button below.
```

At the bottom are two buttons: 'OPEN EXAMPLES' and 'SEARCH STACK OVERFLOW'.

Рис. 16 Ошибка при импорте классов из неустановленных библиотек

Мы не сможем импортировать самостоятельно написанный класс `ClassicStacking` из модуля `classic_stacking.py`, поскольку не прописан путь, откуда можно импортировать модули.

```
# импортируем самописный модуль classic_stacking
from classic_stacking import *

ModuleNotFoundError: No module named 'classic_stacking'

NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
```

[OPEN EXAMPLES](#) [SEARCH STACK OVERFLOW](#)

Рис. 17 Ошибка при импорте из модуля с неизвестным путем

Данный блокнот «не увидит» данные, которые вы положили в Google-диск.

```
 FileNotFoundError: [Errno 2] No such file or directory: 'Data/ottogroup_train.csv'

SEARCH STACK OVERFLOW
```

Рис. 18 Ошибка при загрузке данных

5.3. Подготовка блокнота Colab

Итак, наш блокнот «не увидит» данные, которые вы положили в Google-диск.

Это происходит потому, что блокнот запущен в выданной вам виртуальной машине, к которой не примонтирован ваш Google-диск с нужными данными. Речь идет о виртуальной машине с операционной системой Linux.

Эти данные можно загрузить в данную машину разными способами, но проще всего примонтировать к виртуальной машине ваш Google-диск и пользоваться всем, что там есть. Для этого в первой ячейке блокнота введем и запустим следующий код:

```
# примонтируем Google-диск
from google.colab import drive
drive.mount('/content/drive')
```

Вам необходимо разрешить ноутбуку доступ к вашим файлам на Google-диске.



Рис. 19 Запрос разрешения на доступ к файлам на Google-диске

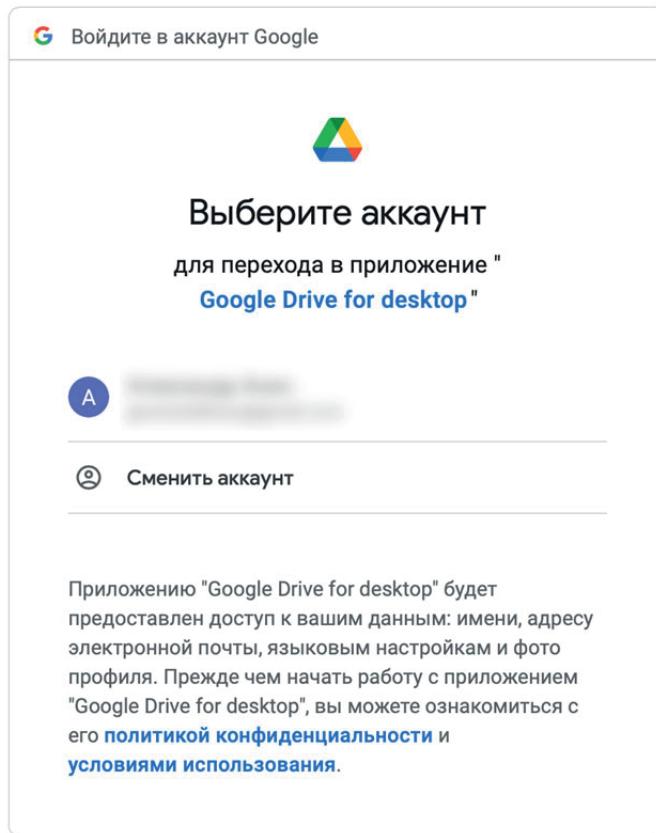


Рис. 20 Выбор аккаунта и запрос разрешения на доступ к Google-аккаунту со стороны приложения Google Drive for desktop

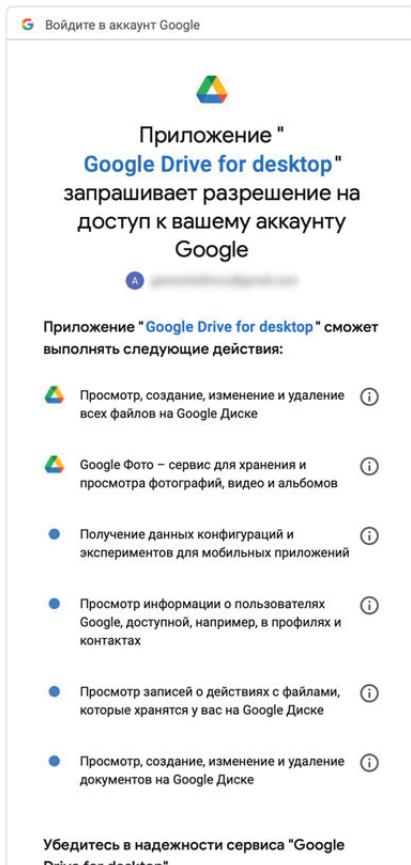


Рис. 20 Окончание

После подтверждения разрешения выйдет сообщение о том, что Google-диск примонтирован по пути `/content/drive`:

```
# примонтируем Google-диск
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

После этого монтирования полный путь в вашей виртуальной машине к папке Colab Notebooks на вашем Google-диске будет таким: `/content/drive/MyDrive/Colab Notebooks/`.

Мы можем выполнять на нашей машине любые обычные команды системы Linux, запуская их в ячейке блокнота, только добавляя к нему вначале знак восклицания! Таким образом, мы можем посмотреть содержимое этой папки, выполнив в следующей ячейке блокнота код:

```
!ls -al '/content/drive/MyDrive/Colab Notebooks/'
```

```
total 4
drwx----- 2 root root 4096 Aug 20 09:25 Stacking
```

В нашем примере в ней видна пока только одна папка – `Stacking`. Можно посмотреть содержимое папки `Stacking` с помощью следующей команды:

```
!ls -al '/content/drive/MyDrive/Colab Notebooks/Stacking/'
```

```
total 55
-rw----- 1 root root 4492 Mar 15 2021 classic_stacking.py
drwx----- 2 root root 4096 Aug 20 09:53 Data
-rw----- 1 root root 47265 Aug 20 11:30 'Собственный класс ClassicStacking.ipynb'
```

Можно просмотреть содержимое подпапки `Data`:

```
!ls -al '/content/drive/MyDrive/Colab Notebooks/Stacking/Data/'
```

```
total 39401
-rw----- 1 root root 27912710 Dec 10 2019 ottogroup_test.csv
-rw----- 1 root root 12433387 Dec 10 2019 ottogroup_train.csv
```

Теперь мы знаем полный путь к данным, который нужно указать при загрузке. Давайте укажем путь, и наши данные нормально загрузятся.

```
# загружаем исторический набор и набор новых данных
train = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/' +
                     'Stacking/Data/ottogroup_train.csv')
test = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/' +
                     'Stacking/Data/ottogroup_test.csv')
```

Однако в выданной нам виртуальной машине не установлена библиотека `catboost`. Установим ее так, как это обычно делается в системах Linux (не забывая добавлять в начале ячейки восклицательный знак !).

```
!pip install catboost
```

Мы увидим, как система скачает и установит нужную нам библиотеку.

```
!pip install catboost
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting catboost
  Downloading catboost-1.0.6-cp37-none-manylinux1_x86_64.whl (76.6 MB)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from catboost) (3.2.2)
Requirement already satisfied: plotly in /usr/local/lib/python3.7/dist-packages (from catboost) (5.5.0)
Requirement already satisfied: numpy==1.16.0 in /usr/local/lib/python3.7/dist-packages (from catboost) (1.21.6)
Requirement already satisfied: graphviz in /usr/local/lib/python3.7/dist-packages (from catboost) (0.10.1)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from catboost) (1.7.3)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from catboost) (1.16.0)
Requirement already satisfied: pandas<0.24.0,!=0.23.4 in /usr/local/lib/python3.7/dist-packages (from catboost) (1.3.5)
Requirement already satisfied: python-dateutil<2.7.3 in /usr/local/lib/python3.7/dist-packages (from pandas<0.24.0->catboost) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from pandas<0.24.0->catboost) (2022.2.1)
Requirement already satisfied: pyParsing!=2.0.4,!>=2.1.2,!>=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->catboost) (3.0.9)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->catboost) (1.4.4)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib->catboost) (0.11.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from kiwisolver>=1.0.1->matplotlib->catboost) (4.1.1)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.7/dist-packages (from plotly->catboost) (8.0.1)
Installing collected packages: catboost
Successfully installed catboost-1.0.6
```

После этого класс `CatBoostClassifier` из библиотеки `catboost` прекрасно импортируется.

```
# импортируем класс CatBoostClassifier из библиотеки catboost
from catboost import CatBoostClassifier
```

Хотя мы и положили рядом с файлом открытого блокнота файл *classic_stacking.py*, мы не можем его импортировать.

Это происходит в силу того, что путь */content/ddrive/MyDrive/Colab Notebooks/Stacking* не прописан в системе как путь, откуда могут импортироваться модули. Укажем системе этот путь как путь, откуда можно импортировать модули. Для этого выполним команду:

```
# импортируем модуль sys
import sys
# прописываем путь к модулю
sys.path.append('/content/drive/MyDrive/Colab Notebooks/Stacking/')
```

После чего мы можем легко импортировать все необходимое из нашего модуля *classic_stacking.py*. При этом лучшей практикой считается импорт не всего подряд с помощью символа * (так называемый «небрежный» импорт), а именно импорт нужных функций и классов.

```
# импортируем класс ClassicStacking из модуля classic_stacking
from classic_stacking import ClassicStacking
```

Затем можно импортировать остальные библиотеки. Устанавливать их не нужно, они уже установлены в системе.

```
# импортируем необходимые библиотеки и классы
import numpy as np
from xgboost import XGBClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

Кроме того, если мы сохраняем данные (в нашем случае – посылку для Kaggle), то можем их сохранить сразу на Google-диск. Для этого опять стоит указать полный путь к данным. В нашем случае нужно изменить код в самом конце блокнота:

```
submission.to_csv('own_cv_stacking_on_cl_probs_with_concat.csv', index=False)
```

на такой:

```
submission.to_csv('/content/drive/MyDrive/Colab Notebooks/Stacking/' +
'own_cv_stacking_on_cl_probs_with_concat.csv', index=False)
```

После чего мы можем запустить код в ноутбуке после строки

```
from sklearn.linear_model import LogisticRegression
```

и получить выходной файл в соответствующей папке Google-диска.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,
выслав открытку или письмо по почтовому адресу:

115487, г. Москва, пр. Андропова д. 38 оф. 10.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.galaktika-dmk.com.

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: books@aliants-kniga.ru.

Груздев Артём Владимирович

Предварительная подготовка данных в Python

Том 2: План, примеры и метрики качества

Главный редактор *Мовчан Д. А.*

dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Корректор *Синяева Г. И.*

Верстка *Луценко С. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать цифровая.

Усл. печ. л. 66,14. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com