В. В. КОНКИНА, А. Б. БОРИСЕНКО, И. Л. КОРОБОВА

ВВЕДЕНИЕ В БОЛЬШИЕ ДАННЫЕ И АНАЛИЗ ИНФОРМАЦИИ

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тамбовский государственный технический университет»**

В. В. КОНКИНА, А. Б. БОРИСЕНКО, И. Л. КОРОБОВА

ВВЕДЕНИЕ В БОЛЬШИЕ ДАННЫЕ И АНАЛИЗ ИНФОРМАЦИИ

Утверждено Ученым советом университета в качестве учебного пособия для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплину «Разработка информационного обеспечения», а также для студентов направления подготовки 09.04.01 «Информатика и вычислительная техника», изучающих дисциплину «Введение в большие данные и анализ информации», очной и заочной форм обучения

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

УДК 004.04
ББК 16.23
К64

Рецензенты:

Кандидат технических наук, старший научный сотрудник ФГБНУ «ВНИИТиН»
Н. Ю. Пустоваров

Кандидат технических наук, доцент, доцент кафедры
«Информационные процессы и управление» ФГБОУ ВО «ТГТУ»
А. А. Третьяков

Конкина, В. В.

К64 Введение в большие данные и анализ информации [Электронный ресурс] : учебное пособие / В. В. Конкина, А. Б. Борисенко, И. Л. Коробова. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2024. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже Pentium IV ; CD-ROM-дисковод ; 1,32 Мб ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2749-8

Рассмотрены основные концепции анализа данных, язык программирования Python, работа с библиотекой Pandas, очистка данных от выбросов, пропусков и дубликатов. Приведены основы теории вероятностей и статистики, применение их для исследования основных свойств данных, поиска закономерностей, распределений и аномалий. Описана библиотека Matplotlib. Подробно проанализированы взаимосвязи в данных методами статистики. Даны определения статистической значимости и гипотезы. Рассмотрены основы структурированного языка запросов SQL и реляционной алгебры для работы с базами данных.

Предназначено для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплину «Разработка информационного обеспечения», а также для студентов направления подготовки 09.04.01 «Информатика и вычислительная техника», изучающих дисциплину «Введение в большие данные и анализ информации», очной и заочной форм обучения.

УДК 004.04
ББК 16.23

ISBN 978-5-8265-2749-8

Без данных вы просто еще один человек со своим мнением

Уильям Деминг[*]

ВВЕДЕНИЕ

Анализ данных в современном мире чем-то напоминает добычу золота во времена золотой лихорадки. Данных очень много, и на первый взгляд их трудно структурировать, а тем более извлечь из них что-то полезное. Благодаря развитию вычислительной техники есть возможность структурировать и исследовать данные, находить неожиданные зависимости в них и извлекать новые знания.

Четкого определения того, какие данные считать большими, не существует. Нет какого-то предела объема, после которого обычные данные превращаются в большие. Но обычно речь идет, как минимум, о сотнях гигабайт и сотнях тысяч строк в базах данных. Еще большие данные, как правило, регулярно пополняются, обновляются и изменяются, т.е. их не только хранят, но и активно собирают.

Собранные данные в хранилище – это просто набор информации, который не способен принести никакой пользы. Чтобы польза была, необходим анализ больших данных – их структурирование и обработка по специальным алгоритмам с целью сделать определенные выводы.

Данные исследуют в четыре стадии:

- получение данных и ознакомление с ними;
- предобработка данных;
- анализ данных;
- оформление результатов исследования.

[*] Уильям Деминг (14 октября 1900 – 20 декабря 1993) – американский ученый, статистик и консультант по менеджменту.

1. ПРЕДОБРАБОТКА ДАННЫХ

1.1. РАБОТА С ПРОПУСКАМИ

Предобработка – процесс подготовки данных для дальнейшего анализа. Его суть заключается в поиске и устранении возможных «проблем» в данных.

Таблица, удобная для анализа данных:

- в каждом столбце хранятся значения одной переменной;
- каждая строка содержит одно наблюдение, к которому привязаны значения разных переменных.

Названия столбцов:

- без пробелов в начале, середине и конце;
- несколько слов разделяются нижним подчеркиванием;
- на одном языке и в одном регистре;
- отражают в краткой форме, какого рода информация содержится в каждом столбце.

Пропущенные значения бывают разные:

- чаще всего это *None* или *NaN (*от англ. *Not-a-Number* – нечисло);
- плейсхолдеры (тексты-заполнители) какого-то общепринятого стандарта, которого придерживаются составители. Чаще всего это *n/a*, *na*, *NA*, и *N.N*. либо *NN*;
- произвольное значение, которое по договоренности между собой используют создатели исходной таблицы данных.

Пропущенные значения можно как удалять, так и заполнять на основе известных данных.

Плюс удаления данных в том, что это простой процесс. Также можно быть уверенным, что те данные, которые остались, хорошие и отвечают всем требованиям. Потенциальные минусы: потеря важной информации и снижение точности.

Заполнение позволяет сохранить наибольшее количество данных. Очевидный минус – могут получиться плохие результаты на основе уже существующих данных.

NaN и *None* – эти особые значения указывают, что никакого значения нет. *NaN* отвечает за отсутствующее в ячейке число. Его тип данных *float*, поэтому с *NaN* можно проводить математические операции. *None* принадлежит к нечисловому типу *NoneType*, и математические операции с ним неосуществимы. Значения *NaN* могут привести к некорректным результатам при группировке данных. Строки с этими значениями не всегда стоит удалять, часто пропуски можно восстановить.

*Я*зык *Python* – основной инструмент аналитика данных. Для обработки данных подойдет и *Excel*, но лучше использовать профильный инструмент – программную библиотеку *Pandas*. Из того, что есть в *Python*, для операций с таблицами чаще всего применяют *Pandas*. Название *Pandas* происходит от сокращения *panel-data* (англ. «панельные данные») – пришло из терминологии применяемого в экономике панельного анализа, который изучает изменение определенного признака у определенного объекта во времени (например, уровень заболеваемости холерой в Африке в начале XXI века). Библиотека *Pandas* оказалась таким универсальным инструментом, что годится для исследования любых данных, которые вообще можно собрать в таблицу.

Возможности *Pandas*:

- готовые методы для манипуляций с таблицами: добавления, удаления, преобразования, агрегирования данных;
- одновременная обработка данных из разных файлов;
- готовые методы для операций с пропущенными значениями, выявления и устранения проблемных данных;
- использование данных в самых разных форматах.

В *Pandas* метод *value_counts*() возвращает уникальные значения с их количеством. Методы *isna*() или *isnull*() возвращают булевский список, в котором *True* означает, что значение в колонке пропущено. Для замены пропусков на какое-то значение применяется метод *fillna*() с аргументом *value*.

Переменные бывают двух типов: категориальные и количественные. Категориальная переменная принимает одно значение из ограниченного набора, а количественная – любое числовое значение в диапазоне. Количественные переменные, в отличие от категориальных, обладают возможностью сравнения.

Также переменные могут быть логическими (булевыми, от англ. *Boolean*). Такие переменные указывают на истинность или ложность какого-либо события. Если событие истинно, то переменная принимает значение 1, соответствующее *True*, а если ложно – 0, соответствующий *False*.

Перед обработкой пропусков нужно ответить на вопрос, существует ли закономерность в появлении пропусков. Иными словами, не случайно ли их возникновение в наборе данных.

Пропуски бывают трех типов:

1) полностью случайные: если вероятность встретить пропуск не зависит ни от каких других значений. Такой пропуск легко восстановить;

2) случайные: если вероятность пропуска зависит от других значений в наборе данных, но не от значений собственного столбца. Пропущенное значение связано с тем, что, например, такой категории не существует;

3) неслучайные: если вероятность пропуска зависит от других значений, в том числе и от значений собственного столбца. Отсутствующее значение зависит от значения переменной в другом столбце.

Существует несколько вариантов замены пропусков категориальных значений. Например, замена значением по умолчанию. Такой вариант хорошо подойдет для заполнения случайных пропусков. В *Pandas* для этого применяется метод *fillna*(). Не все пустые значения можно заполнить методом *fillna*(). Например, к пропущенным значениям *None* его не применить – метод распознает только значения *NaN* в таблице. Для замены *None* вызывают метод *loc*. Логическая индексация позволит выделить все строки в необходимом столбце, которые содержат *None*, и заменить их на новое значение.

Для применения некоторых функций к определенным столбцам, применяется метод *agg*(). Название столбца и сами функции записываются в структуру данных – словарь. Словарь состоит из ключа и значения. Ключ – это назва-

ние столбца, к которому нужно применить функции, а значением выступает список с названиями функций.

```
{'column':['function1','function2']}
```

После применения метода *agg*(), названия столбцов становятся «двойными». Чтобы обратиться к результату применения функции [*'function*1*'*] к столбцу[*'column*'], надо указать их подряд:

```
data['column']['function1']
```

Пропуски в количественных переменных заполняют характерными значениями. Это значения, характеризующие состояние выборки – набора данных, выбранных для проведения исследования. Чтобы примерно оценить типичные значения выборки, годятся среднее арифметическое или медиана.

Среднее арифметическое – это сумма всех значений, поделенная на количество значений. Медиана – это такое число в выборке, что ровно половина элементов больше него, а другая половина – меньше. Для получения среднего арифметического применяется метод *mean*().

Его применяют ко всей таблице, к отдельному столбцу или к сгруппированным данным. Для нахождения медианы есть специальный метод *median*(), его можно применять к таблице, столбцу или сгруппированным данным.

Минимум и максимум – это наименьшее и набольшее числа в наборе. Показатель максимума или минимума обычно вычисляют по отдельному признаку.

Среднее и медиана используются для оценки значения в центре выборки. Если выборка равномерна и в ней нет значений, слишком отличающихся от остальных, – среднее подойдет. Но когда есть оторванные от основной массы значения, они сильно смещают среднее вверх. В таком случае используется медиана.

Для прочтения файлов *Excel* есть особый метод *read_excel*(). Он похож на *read_csv*(), но в отличие от него *read_excel*() нужно два аргумента: строка с именем самого файла или пути к нему, и имя листа *sheet_name*. Если аргумент *sheet_name* пропущен, то по умолчанию прочитается первый по счету лист.

```
import Pandas as pd
df = pd.read_excel('file.xlsx', sheet_name='List1')
```

Следует отметить, что для чтения данных с помощью метода *read_excel*() необходимо, чтобы был установлена библиотека *openpyxl*.

Для того, чтобы перевести строковые значения в числа, есть стандартный метод *Pandas* – *to_numeric*(). Он превращает значения столбца в числовые типы *float*64 (вещественное число) или *int*64 (целое число) в зависимости от исходного значения. У метода *to_numeric*() есть параметр *errors*. От значений, принимаемых *errors*, зависят действия *to_numeric* при встрече с некорректным значением:

- *errors='raise'* – дефолтное поведение (поведение по умолчанию): при встрече с некорректным значением выдается ошибка, операция перевода в числа прерывается;
- *errors='coerce'* – некорректные значения принудительно заменяются на *NaN*;
- *errors='ignore'* – некорректные значения игнорируются, но остаются.

Для того, чтобы перевести данные в нужный тип, применяется метод *astype*(). В качестве аргумента передается строка с названием типа.

Для работы с датой и временем в *Python* существует особый тип данных – *datetime*.

Чтобы перевести строку в дату и время используется метод *to_datetime*(). Параметрами метода являются столбец, содержащий строки, и формат даты в строке.

Формат даты задается с помощью специальной системы обозначений, где:

- %*d* – день месяца (от 01 до 31);
- %*m* – номер месяца (от 01 до 12);
- %*Y* – год с указанием столетия (например, 2019);
- %*H* – номер часа в 24-часовом формате;
- %*I* – номер часа в 12-часовом формате;

- %*M* – минуты (от 00 до 59);
- %*S* – секунды (от 00 до 59).

```
date['column']= pd.to_datetime(date['column'], for-
mat='%d.%m.%YZ%H:%M:%S')
```

Среди самых разнообразных способов представления даты и времени особое место занимает формат *unix time*. Его идея проста – это количество секунд, прошедших с 00:00:00 1 января 1970 года. *Unix*-время соответствует Всемирному координированному времени, или *UTC*. Метод *to_datetime*() работает и с форматом *unix time*. Первый аргумент – это столбец со временем в формате *unix time*, второй аргумент *unit* со значением 's' сообщит о том, что нужно перевести время в привычный формат нужно с точностью до секунды. Часто приходится исследовать статистику по месяцам, дням, годам. Чтобы осуществить такой расчет, нужно поместить время в класс *DatetimeIndex* и применить к нему атрибут *month, day, year*:

```
date['column'] = pd.DatetimeIndex(date['column']).month
```

Выгружая данные из разных систем, нужно быть готовым к следующим трудностям:

- некорректный формат приводит к невыполнению кода. Говорят, что «код падает с ошибкой»;
- ошибки в данных встречаются ближе к концу файла, и код на строках с неверными значениями не выполняется. Значит, пропадают расчеты для предыдущих, правильных строк;
- данные могут поменяться.

К сожалению, предсказать все потенциальные ошибки невозможно. Для работы с непредсказуемым поведением данных есть конструкция *try-except*. Принцип работы такой: исходный код помещают в блок *try*. Если при выполнении кода из блока *try* возникнет ошибка, воспроизведется код из блока *except*.

```
try:
    # код, где может быть ошибка
except:
    # действия если возникла ошибка
```

Данные хранятся в *excel*-таблице из нескольких листов. Для того, чтобы использовать данные на всех листах, нужно склеить таблицы. Объединить несколько таблиц в одну поможет метод *merge*().

Аргументы:

- *right* – имя *DataFrame* или *Series*, который нужно присоединить к исходной таблице;
- *on* – общее поле в двух таблицам, по которым происходит соединение;
- *how* – какие *id* включены в итоговую таблицу. Может принимать значения *left – id* из левой таблицы будут включены в итоговую таблицу или *right – id* из правой таблицы будут включены в итоговую таблицу.

Сводная таблица применяется для обобщения данных и их наглядного представления. В *Pandas* для подготовки сводных таблиц есть метод *pivot_table*().

Аргументы метода:

- *index* – столбец или столбцы, по которым происходит группировка данных;
- *columns* – столбец, по значениям которого будет происходить группировка;
- *values* – значения, по которым хотим увидеть сводную таблицу;
- *aggfunc* – функция, которая будет применяться к значениям.

1.2. ПОИСК ДУБЛИКАТОВ

Часто при анализе данных возникают дубликаты. Если не найти дубликаты, то анализ данных может привести к некорректным результатам.

Дубликаты (дублированные записи) могут быть следующего вида:

- две и более одинаковых строки с идентичной информацией. Большое количество повторов раздувает размер таблицы, а значит, увеличивает время обработки данных;
- одинаковые по смыслу категории с разными названиями, например, «Политика» и «Политическая ситуация». Замаскированные повторы – источник серьезных и с трудом обнаруживаемых ошибок в анализ.

Дубликаты можно искать двумя способами.

Способ 1. Методом *duplicated*(). В сочетании с методом *sum*() он возвращает количество дубликатов. Если выполнить метод *duplicated*() без суммирования на экране будут отображены все строки. Там, где есть дубликаты, будет значение *True*, где дубликата нет – *False*.

Способ 2. Методом *value_counts*(), возвращающим уникальные значения с их частотой. Его применяют к объекту *Series*. Результат работы метода – список пар «значение-частота», отсортированный по убыванию. Значит, интересующие нас дубликаты будут в начале списка.

Series – одномерная структура данных *Pandas*, ее элементы можно получить по индексу. Каждый индекс представляет собой номер отдельного наблюдения, и поэтому несколько различных *Series* вместе составляют *DataFrame*:

- в *Series* хранятся данные одного типа;
- у *Series* есть имя (*Name*), информация о количестве данных в столбце (*Length*) и тип данных, которые хранятся в ней (*dtype*);
- индексация в *Series* аналогична индексации элементов столбца в *DataFrame*.

DataFrame – это двумерная структура данных *Pandas*, где у каждого элемента есть два индекса: по строке и по столбцу.

Каждая строка – это одно наблюдение, запись об объекте исследования. А столбцы – признаки этого объекта.

DataFrame() – это конструктор библиотеки *Pandas*, который используется для создания *DataFrame*. Перед именем конструктора стоит обращение к переменной, в которой библиотека хранится: *pd.DataFrame*(). У *DataFrame* есть неотъемлемые свойства, значения которых можно запросить. Они называются атрибуты. Например, это размер таблицы *df.shape* или количество столбцов *df.columns*. К каждой ячейке с данными в *DataFrame* можно обратиться по ее индексу и названию столбца. Этот процесс называется индексация и для *DataFrame* его проводят разными способами.

Дубликаты в строковых данных требуют особого внимания, поскольку регистр имеет значение: заглавная '*A*' и строчная '*a*' с точки зрения *Python* – раз-

ные символы, но имеют одинаковое значение – буква *A*. Чтобы учесть такие дубликаты, все символы в строке приводят к нижнему регистру вызовом метода *lower*(). В *Pandas* символы приводят к нижнему регистру методом с похожим синтаксисом: *str.lower*().

При разделении строк по категориям, проверка вхождения определенных подстрок в строку не всегда может дать корректный результат. Один из приемлемых вариантов для решения такой задачи – стемминг (от англ. *stemming* – находить происхождение). Стемминг – это процесс нахождения основы заданного слова, называемой стемом. В *Python* для стемминга есть специальная библиотека *NLTK*. Он содержит стеммеры – специальные объекты, содержащие правила определения стемов. Метод *stem*() применяется для извлечения стема для всех слов в строке.

```
from nltk.stem import SnowballStemmer
russian_stemmer = SnowballStemmer('russian')
russian_stemmer.stem(text)
```

Стемминг – не единственный алгоритм для поиска слов, записанных в разных формах. Более продвинутый процесс – лемматизация, или приведение слова к его словарной форме (лемме).

В русском языке формы записи в словаре (леммы) следующие:

- для существительных – именительный падеж, единственное число;
- для прилагательных – именительный падеж, единственное число, мужской род;
- для глаголов, причастий, деепричастий – глагол в инфинитиве несовершенного вида.

Одна из библиотек с функцией лемматизации на русском языке – *pymystem*3. *NLTK.pymystem*3 по умолчанию выдает список лемматизированных слов (слов, сведенных к лемме), а *NLTK* – строку.

```
# pymystem3 импортируется так:
from pymystem3 import Mystem
m = Mystem()
lemmas = m.lemmatize(text)
```

Для подсчета встречаемости значений в списке используется специальный контейнер *Counter* из модуля *collections*.

```
from collections import Counter
print(Counter(lst))
```

1.3. КАТЕГОРИЗАЦИЯ ДАННЫХ

Кроме внутренних преобразований, важны и внешние. Необходимо привести данные в удобный и читаемый вид. Одна из подзадач в этом процессе – именование столбцов. Для этого применяется метод *set_axis*().

В датасетах могут встречаться категории в виде названий, их длина может быть различной.

Такой способ хранения приводит к следующим последствиям:

- усложняется визуальная работа с таблицей;
- увеличивается размер файла и время обработки данных;
- чтобы отфильтровать данные по типу обращения, приходится набирать его полное название (а в нем можно допустить ошибки);
- создание новых категорий и изменение старых отнимает много времени.

Для того, чтобы оптимально хранить информацию о категориях, создается отдельный файл-словарь, названию категории соответствует номер. Этот номер в дальнейшем используется вместо текстового наименования категории в таблице.

Часто объекты, имеющие определенное значение признака, присутствуют в наборе только единожды. Работать с такими отрывками и делать из них статистические выводы нельзя. Поэтому с такими данными проводится категоризация – объединение данных в категории.

Одним из вариантов категоризации является выделение возрастных групп, например: до 18, от 18 до 65, старше 65.

Подобные правила классификации в *Python* удобно представлять в виде функций, которые принимают параметр – значение признака, а возвращают соответствующую категорию.

Для того, чтобы получить столбец с группой на основе столбца с каким-то другим признаком, можно воспользоваться написанной нами функцией *group* и методом *apply*(), в который в качестве параметра передается эта функция.

```
data['column_group'] = data['column'].apply(group)
```

Когда для категоризации недостаточно значения в одном каком-то столбце, то в функцию можно передать и всю строку как *Series*. В теле этой функции можно также получать значения в каком-то определенном столбце.

Применение метода *apply*() в случае обработки строки имеет два отличия:

1) метод *apply*() вызывается не к столбцу *data* ['*age*'], а к датафрейму *data*;

2) по умолчанию *Pandas* передает в функцию *group*() столбец. Чтобы на вход в функцию отправлялись строки, нужно указать параметр *axis* = 1 метода *apply*().

2. ИССЛЕДОВАТЕЛЬСКИЙ АНАЛИЗ ДАННЫХ

2.1. БАЗОВАЯ ПРОВЕРКА ДАННЫХ

Данные в формате *csv* в качестве разделителя могут иметь не только запятые, но и точки с запятой, знаки табуляции или другие символы. Десятичные дроби, записанные с запятой, тоже могут внести путаницу. В параметрах функции *read_csv*() можно указать, какими символами разделять колонки и дроби. Разделитель колонок задают параметром *sep*, а дробей – параметром *decimal*:

```
file = pd.read_csv('file.csv', sep=';', decimal=',')
```

Занимаясь предобработкой данных, можно применять *pivot_table*() – метод для построения сводных таблиц. Если в значении *aggfunc* указывается *sum*, то элементы столбца складываются. Если параметр *aggfunc* не указывать, то по умолчанию метод *pivot_table*() рассчитает среднее арифметическое значений, указанных в параметре *values*.

В работе с данными почти всегда возникают неприятные сюрпризы:

- из-за непонимания задачи или случайно выгружаются не те или неполные данные;
- ошибки в алгоритмах, считающих нужное значение;
- не тот формат предоставляемых данных;
- упущен какой-нибудь существенный факт.

Словом, в данных может быть все, что угодно. Именно аналитик ручается за их реалистичность.

Гистограмма – это график, который показывает, как часто в наборе данных встречается то или иное значение. Гистограмма объединяет числовые значения по диапазонам, т.е. считает частоту значений в пределах каждого интервала. Ее построение, подобно работе метода *value_counts*(), подсчитывающего количество уникальных значений в списке, однако *value_counts*() группирует строго одинаковые величины и хорош для подсчета частоты в списках с категориальными переменными.

В *Pandas* гистограмму строит специальный метод *hist*(), применяемый к списку или к столбцу датафрейма. Метод *hist*() находит в наборе чисел минимальное и максимальное значения, а полученный диапазон делит на области, или корзины, затем считает, сколько значений попало в каждую корзину, и отображает это на графике. Параметр *bins* определяет, на сколько областей делить диапазон данных, по умолчанию таких *bins* = 10. По умолчанию гистограмма выводится для всех значений от минимального до максимального. Масштаб можно изменить вручную, указав параметр *range*:

```
range=(min_value, max_value)
```

Для создания графиков (в том числе гистограмм), импортируют библиотеку *matplotlib*. Метод *show*() отображает графики.

```
import Pandas as pd
import matplotlib.pyplot as plt
# импортируем библиотеку, стандартно используется имя plt
pd.Series(...).hist(bins=n_bins, range=(min_value, max_value)
plt.show() # даем команду отобразить гистограмму
```

Распределение – это все возможные значения переменной с частотой их появления. Распределения бывают нормальными (англ. *Normal distribution* (*Gaussian*)) и Пуассона (англ. *Poisson distribution*).

Нормальное: чаще всего встречается среднее значение и близкие к нему, а крайние значения встречаются довольно редко.

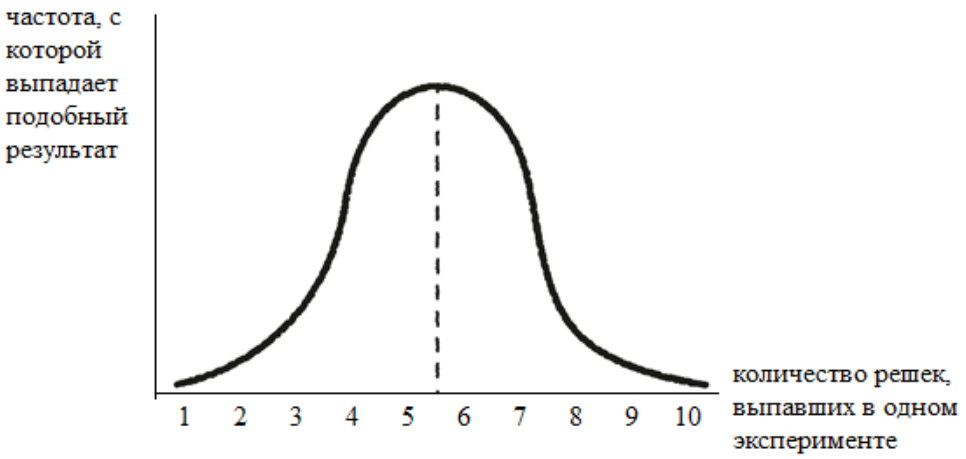

Рис. 1. Нормальное распределение

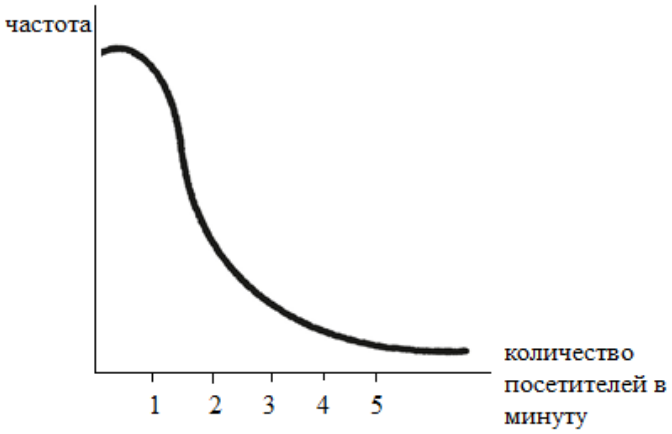

Рис. 2. Распределение Пуассона

Распределение Пуассона: число событий в единицу времени, если они в среднем происходят с измеренной частотой.

Характерный разброс – то, какие значения оказались вдали от среднего, и насколько их много.

Если считать характерным разбросом расстояние между минимальным и максимальным значением, то не всегда получится точное описание данных, на него могут повлиять выбросы. Поэтому в качестве характерного разброса применяют межквартильный размах.

Квартили (от латинского *quartus* – «четвертый») разбивают упорядоченный набор данных на четыре части: первый квартиль $Q1$ – число, отделяющее первую четверть выборки (25% элементов меньше, а 75% – больше него); медиана – второй квартиль $Q2$ (половина элементов больше и половина меньше нее); третий квартиль $Q3$ – это отсечка трех четвертей 75% элементов меньше и 25% элементов больше него).

Межквартильный размах – это расстояние между первым квартилем $Q1$ и третьим квартилем $Q3$.

Диаграмма размаха, или так называемый «ящик с усами» (англ. *Box and Whisker Plot* или *Box Plot*), позволяет отобразить все квартили для заданных данных.

«Ящик» ограничен первым и третьим квартилями. Внутри ящика обозначают медиану. «Усы» простираются влево и вправо от границ ящика на расстояние, равное 1,5 межквартильным размахам (*IQR*, от англ. *interquartile range*).

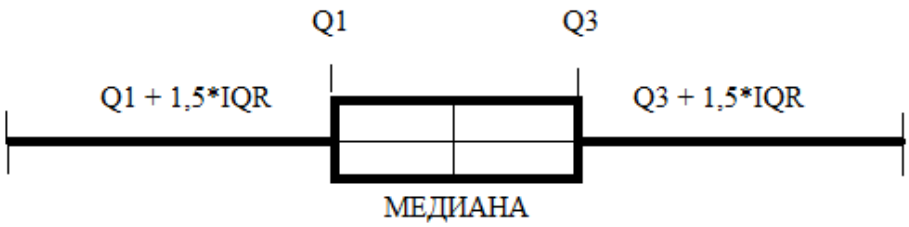

Рис. 3. Диаграмма размаха («ящик с усами»)

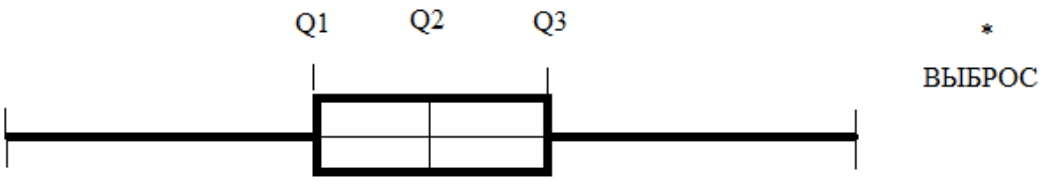

Рис. 4. Выброс на диаграмме размаха

В размах «усов» попадают нормальные значения, а за пределами находятся выбросы, изображенные точками. Если правый «ус» длиннее максимума, то он заканчивается максимумом. То же – для минимума и левого «уса».

В Python диаграмму размаха строят методом *boxplot*(), он позволяет визуально оценить характеристики распределения, не прибегая к гистограмме.

```
import matplotlib.pyplot as plt
data.boxplot()
plt.show()
```

Оси любого графика в *Pandas* можно изменять, для этого нужно применить метод *xlim*(*x_min, x_max*) для оси *X* и *plt.ylim*(*y_min, y_max*) для оси *Y*. Параметры в обоих случаях – минимальное и максимальное значение на графике.

```
import matplotlib.pyplot as plt
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

С помощью гистограмм и диаграмм размаха можно получить графическое описание любого набора данных. Однако не всегда по графикам можно определить такие характеристики, как среднее, медиану, число наблюдений в выборке и разброс их значений – числовое описание данных.

В *Pandas* для этого применяется метод *describe*().

Стандартное отклонение – числовая характеристика данных, входящая в числовое описание данных и характеризующая разброс величин, показывает, насколько значения в выборке отличаются от среднего арифметического. Оно позволяет понять природу распределения и определить, насколько значения однородны.

```
data['column'].describe()
```

2.2. СРЕЗЫ ДАННЫХ

В ходе работы над задачей часто нужны не все данные, а лишь их часть, или срез данных (англ. *Data Slice*). Срез данных – это часть данных из предоставленного набора, отобранная по определенным условиям.

Срез данных можно получить, применив фильтр. Этим фильтром может стать булев массив, состоящий из *True* и *False*. Так, значениями *True* отмечается то, что надо включить определенную строку в срез данных.

Формировать этот фильтр вручную – занятие утомительное, а на больших датасетах – практически невозможное, поэтому в *Pandas* применяются автоматические фильтры. Ниже приведен пример автоматического фильтра.

```
data['column'] == 'value'
```

Этот фильтр в дальнейшем служит как индекс датафрейма:

```
data[data['column'] == 'value']
```

Условием создания фильтра может выступать не только равенство, но и другие операции сравнения: !=, >=, <=, >, <. Значения в столбцах можно сравнивать и с заданными значениями, и между собой:

```
data['column1'] > data['column2']
```

В условиях допустимы арифметические операции:

```
data['column1'] / 2 > data['column2'] + 0.5
```

1. Логические операции

Логическая операция	Описание	Синтаксис
ИЛИ	Результат выполнения – *True*, если хотя бы одно из условий – *True*	`(df['a'] > 2) | (df['c'] != 'Y')`
И	Результат выполнения логической операции *True*, только если оба условия –*True*	`(df['a'] > 2) & (df['c'] != 'Y')`
НЕ	Результат выполнения –*True*, если условие – *False*	`~ ((df['a'] > 2) | (df['c'] != 'Y'))`

Чтобы проверить наличие конкретных значений в столбце, можно вызвать метод *isin*():

```
data['column'].isin(('value1', 'value2', 'value3'))
```

Иногда нужно получить срез, соответствующий сразу нескольким условиям – для этого существуют логические операции. Отдельные условия указывают в скобках.

Другим более простым методом получения срезов является метод *query*(). Необходимое условие для среза записывается в строке, которую передают как аргумент методу *query*(), а его применяют к датафрейму. В результате получаем нужный срез.

```
data.query('column == value')
```

Условия, указанные в параметре *query*():

- поддерживают разные операции сравнения: !=, >=, <=, >, <, также математические операции;
- допускают вызов методов к столбцам *column.mean*();
- проверяют, входят ли конкретные значения в список, конструкцией:

```
column in (''value1'', ''value2'', ''value3'')
```

Если нужно узнать, нет ли в списке определенных значений – пишут так:

```
a not in ("value1", "value2", "value3")
```

- работают с логическими операторами в привычном виде, где «или» – *or*, «и» – *and*, «не» – *not*. Указывать условия в скобках необязательно. Без скобок операции выполняются в следующем порядке: сначала *not*, потом *and* и, наконец, *or*.

Также, в методе *query*() можно получать значения внешних переменных:

```
variable = 2
data.query('column > @variable')
```

При переводе данных из строки в тип *datetime* применяется метод *pd.to_datetime*(). Зачастую, *Pandas* пытается самостоятельно угадать формат данных. Однако в таких случаях стоит указывать параметр *yearfirst=True*, который означает, что в переданной строке сначала идет год. Но все же рекомендуется прописывать формат вручную, чтобы избежать ошибок.

Для доступа к отдельным компонентам даты и времени используется атрибут.*dt* столбца с датой и временем:

```
data['datetime'].dt.year
data['datetime'].dt.weekday
```

Если нужно прибавить или убавить время, то используется *pd.Timedelta*(), которому можно передать желаемый сдвиг в днях, часах, минутах и т.д., передавая соответствующие параметры. Обратите внимание, что переход через 24 часа автоматически приводит к изменению даты.

```
data['shifted_datetime'] = data['datetime'] + pd.Timedelta(hours=10)
```

Для округления времени в *Pandas* есть метод.*dt.round*(). В качестве параметра передается шаг округления строкой вида '1*H*', что означает 1 час (*H – hour* – час). Не обязательно округлять с шагом в 1 час. Можно округлять и с шагом в несколько часов. Другие популярные единицы округления:

- '*D*' – *day* – день;
- '*H*' – *hour* – час;
- '*T*' или '*min*' – *minute* – минута;
- '*S*' – *second* – секунда.

```
data['datetime_round'] = df['datetime'].dt.round('3H')
```

Если нужно округлить в меньшую или большую сторону, вместо метода *round*() используются методы *floor*() и *ceil*() соответственно.

За построение графиков в *Pandas* отвечает метод *plot*(), который строит графики по значениям столбцов из датафрейма. На оси абсцисс (*x*) расположились индексы, а на оси ординат (*y*) – значения столбцов.

```
data.plot()
```

У метода *plot*() есть параметр *style*, который отвечает за стиль отображения точек:

- '*o*' – вместо непрерывной линии отображается каждая точка кружком;
- '*x*' – вместо непрерывной линии отображается каждая точка символом '*x*';
- '*o*-' – отображается непрерывная линия и точка.

Можно изменить оси: напрямую указать, какой столбец отвечает за какую ось:

```
data.plot(x='column_x', y='column_y')
```

Границы осей можно скорректировать параметрами *xlim* и *ylim*, как и в случае с ящиком с усами:

```
data.plot(xlim=(x_min, x_max), ylim=(y_min, y_max))
```

Для отображения сетки, указывается параметр *grid*, равный *True*:

```
data.plot(grid=True)
```

Размером графика управляют через параметр *figsize*. Ширину и высоту области построения в дюймах передают параметру в скобках.

```
data.plot(figsize = (x_size, y_size))
```

Когда данных много, точки на графиках сливаются, и из визуального представления мало что можно понять. Для решения этой проблемы применяется группировка данных. Пример ниже применяет группировку, благодаря которой график становится гораздо информативнее.

```
(data
     .query('column_id == "value"')
     .pivot_table(index='column1', values='column2')
     .plot(grid=True, figsize=(12, 5))
)
```

Стадии группировки можно описать формулой «*split-apply-combine*»:

- *split* (разделить) – разбиение на группы по определенному критерию;
- *apply* (применить) – применение какого-либо метода к каждой группе в отдельности, например, подсчет численности группы методом *count*() или суммирование вызовом *sum*();
- *combine* (объединить) – сведение результатов в новую структуру данных. В зависимости от условий разделения и выполнения метода это бывает *DataFrame* и *Series*.

С помощью графиков с группировкой можно обнаружить выбросы, которые до этого не были видны. Так, иногда из-за выбросов серьезно завышается среднее арифметическое. Как сделать так, чтобы аномально высокие значения не завышали среднее арифметическое? Решений этой задачи может быть два:

1. Убрать выбросы.
2. Вместо среднего арифметического считать медиану. Медиана устойчива к выбросам, но все же не безупречна: пики все еще могут отображаться.

В ходе работы можно обнаружить, что в данных, которые предоставили, содержится ошибка. В таком случае нужно сформулировать проблему, чтобы упростить поиск потенциальной ошибки в алгоритме выгрузки данных. Правильное сообщение об ошибке или баг-репорт должно четко объяснять, в чем именно ошибка и как ее найти. Коллеги, отвечающие за выгрузку, ничего о результатах исследования аналитика не знают. Поэтому нужно четко формулировать, где видится проблема.

2.3. РАБОТА С НЕСКОЛЬКИМИ ИСТОЧНИКАМИ ДАННЫХ

При работе со срезами можно использовать внешние переменные не только числового или строкового типа, но и листы, словари, серии и даже датафреймы. Обращаться к ним можно как к обычным внешним переменным. Если для среза используется лист, то проводится проверка, что значение в определенной колонке входит или не входит в лист:

```
our_list = [1, 2, 3]
print(data.query('column in @our_list'))
```

Аналогичная проверка осуществляется с помощью словаря, но проверяется, что значение в определенной колонке входит или не входит в ключи словаря:

```
our_dict = {0: 10, 1: 11, 2: 12}
print(data.query('column in @our_dict'))
```

Когда в переменной сохранен объект *Series*, конструкция *column in @our_series* проверит вхождение в список значений, а не индексов:

```
our_series = pd.Series([10,11,12])
print(data.query('column in @our_series'))
```

Если нужно проверить вхождение в индекс, это указывают явно, дописывая *index* через точку: *column in @our_series.index*

```
our_series = pd.Series([10,11,12])
print(data.query('column in @our_series.index'))
```

Когда имеют дело с объектом *DataFrame*, вхождение в индекс проверяют так же, как в *Series* – приписав *index* через точку к имени датафрейма:

```
our_dataframe = pd.DataFrame ({
'column1': [2, 4, 6],
'column2': [3, 2, 2],
'column3': ['A', 'B', 'C'],
})
print(data.query('column in @our_dataframe.index'))
```

Для проверки вхождения в какой-либо столбец передают его имя:

```
our_dataframe = pd.DataFrame ({
'column1': [2, 4, 6],
'column2': [3, 2, 2],
'column3': ['A', 'B', 'C'],
})
print(data.query('column in @our_dataframe.column2'))
```

Несколько гистограмм можно отобразить на одном графике. Для этого можно применять следующую конструкцию:

```
ax = data1.plot(kind='hist', y='column1', histtype='step',
     range=(0, 500), bins=25, linewidth=5, alpha=0.7, label='raw')
data2.plot(kind='hist', y='column1', histtype='step', range=(0, 500),
     bins=25, linewidth=5, alpha=0.7, label='filtered', ax=ax,
     grid=True, legend=True)
```

Обратите внимание, что вызвали не метод *hist*(), а *plot*() с параметром *kind*, которому установили значение *kind*='*hist*'. Это та же гистограмма, только поддерживающая параметры, которые нельзя задействовать методом *hist*():

- *histtype* – тип гистограммы, по умолчанию – это столбчатая (закрашенная). Значение '*step*' чертит только линию;
- *linewidth* – толщина линии графика в пикселях;
- *alpha* – густота закраски линии. 1 – это 100%-ная закраска; 0 – прозрачная линия;
- *label* – название линии;
- *ax*. Метод *plot*() возвращает оси, на которых был построен график. Чтобы обе гистограммы расположились на одном графике, надо сохранить оси первого графика в переменной *ax*, а затем передать ее значение параметру *ax* второго *plot*(). Так, сохранив оси одной гистограммы и построив вторую на осях первой, объединятся два графика;
- legend – выводит легенду: список условных обозначений на графике.

При добавлении столбцов в датафрейм нужно учесть несколько нюансов. Пусть у нас есть два датафрейма: *data*1 и *data*2. Добавим в *data*1 новую колонку, значения которой будут совпадать со значениями столбца *data*2['*column*']:

```
data1['new_column'] = data2['column']
```

Если бы столбец *new* уже был в *data*1, то все его элементы были бы удалены, а вместо них записаны новые.

Кажется, просто: *Pandas* копирует столбец из *data*2 и вставляет его в data1, однако все сложнее. Для каждой строки первого датафрейма *Pandas* ищет «пару» – строку с таким же индексом во втором датафрейме. Находит

и берет значение из этой строки. В нашем случае индексы в *data*1 и *data*2 совпадали, и все казалось простым копированием строк по порядку. Если же индексы не будут совпадать, например, в *data*2 не будет некоторых значений индекса *data*1, то при копировании столбца на их месте будет значение *NaN*.

Число строк в *data*2 не обязательно должно совпадать с числом строк *data*1. Если в *data*2 не хватит значений, то будет None. А будут лишние – просто не попадут в обновленный датафрейм. А вот повторяющиеся значения в индексе *data*2 приведут к ошибке. *Pandas* не поймет, какое из значений следует подставить в *data*1. Отдельный столбец можно создать и без датафрейма, в *Series* – это будет набор значений с индексами. При попытке присвоить объект с индексами, *Pandas* подберет соответствующие индексам строки. Если передавать столбцу список значений без индекса, такой как *list*, присвоение будет идти по порядку строк.

При работе над задачей нужно грамотно выбирать метод усреднения, так как он может повлиять на выводы. Где-то среднее арифметическое более точно опишет данные, а где-то может дать некорректный результат – тогда нужно вычислять медиану.

Метод *pivot_table* группирует данные, а что с ними делать, указывает значение параметра *aggfunc*. Среди таких значений есть '*first*' – первое значение из группы и '*last*' – последнее значение из группы.

В одном вызове *pivot_table* можно передать параметру *aggfunc* списком несколько функций – в результирующей таблице они будут в соседних столбцах.

Когда в индексе не одно значение, а целый список, то получаются двухэтажные названия столбцов – мультииндекс. Вообще индексы можно представить как еще один столбец в датафрейме. Выходит, мультииндекс – несколько столбцов. Это справедливо и для названий столбцов: они как дополнительные строки в датафрейме. В таком случае мультииндекс – это две и более строк с названиями.

Когда нужно к существующему датафрейму добавить несколько новых столбцов, существует более лаконичный способ сделать это, по сравнению добавлением каждого столбца по очереди. Такая процедура называется объединение (*join*) или слияние (*merge*).

Для слияния используем метод *merge*(), параметр *how* – метод объединения:

```
data1.merge(data2, on='column', how='inner')
```

Если некоторые значения в колонке слияния совпадают, то остаются только одни, такой тип объединения называется *inner*, т.е. внутренняя общая область, где есть и данные *data*1, и *data*2. Существует тип слияния *outer*, т.е. внешняя общая область – область, где есть данные хотя бы в одном из *df*1 или *df*2; режим объединения *left*, когда в результате слияния обязательно будут все строки из левого *DataFrame* (в нашем случае *data*1); и есть полностью аналогичный режим *right*, когда сохраняются все строки из *data*2. Если совпадут имена столбцов в *data*1 и *data*2, то *Pandas* переименует их, чтобы можно было разобраться, где какое значение. К названию столбца из *data*1 в таком случае припишется *_x*, а к столбцу из *data*2 –*_y*. Можно самостоятельно задать, что приписать к названию столбцов, используя параметр *suffixes*=('*_x*', '*_y*'), заменив ('*_x*', '*_y*') на нужные вам окончания имен столбцов. Отметим так же, что если столбец индекса именованный, то можно передать его имя в параметр *on*. Объединять можно не только по одному столбцу, но и по совпадению значений в нескольких столбцах – достаточно только передать список в *on*.

Так же в *Pandas* существует аналогичный метод *join*(), который ищет совпадения по индексам в *data*1 и *data*2, если не указать параметр *on*. А в случае указания параметра *on* – он будет искать соответствующий столбец в *data*1 и сравнивать его с индексом *data*2. Кроме того, если в *merge*() по умолчанию *how*='*inner*', то *join*()использует по умолчанию *how*='*left*'. Отличается и название параметра *suffixes* – они разделены на 2 независимых *lsuffix* и *rsuffix*. Еще *join* позволяет объединить сразу более двух *DataFrame* –их набор можно передать списком вместо одного *data*2.

2.4. ВЗАИМОСВЯЗЬ ДАННЫХ

График, где значения соединяются линиями, хорош, если иллюстрирует непрерывную связь. Если же в распоряжении данные, которые никак не связаны друг с другом, гораздо лучше обозначить их точками. Это возможно на особом типе графиков – диаграмме рассеяния (*scatter plot*):

```
data.plot(x='column_x', y='column_y', kind='scatter')
```

Очевидный недостаток диаграммы рассеяния в том, что местами может оказаться огромное количество точек, слившихся в единую массу. В облаке значений не разглядеть области более высокой плотности. Есть два способа сделать график нагляднее:

- сделать точки полупрозрачными, задав параметр *alpha* и подобрать его оптимальное значение;
- построить график, поделенный на шестиугольные области.

График делят на ячейки; пересчитывают точки в каждой ячейке. Затем ячейки заливают цветом: чем больше точек – тем цвет гуще. Такой график называется *hexbin* – графиком, разделенным на шестиугольные области. Число ячеек по горизонтальной оси задают параметром *gridsize*, аналогом *bins* для *hist*().

```
data.plot(x='column_x', y='column_y', kind='hexbin',
 gridsize=our_gridsize, sharex=False, grid=True)
```

Смысл этого графика, как и у гистограммы – отображение частотности. Но на гистограмме показана только одна величина, а здесь две. Повышенная частота определенных сочетаний указывает на закономерность. Часто цель анализа данных в том и состоит, чтобы показать связь двух величин.

Взаимозависимость двух величин называется **корреляция**.

График позволяет утверждать, что две величины явно взаимосвязаны, или коррелируют. В том случае, если существует прямая зависимость величин (чем больше одна, тем больше другая), то говорят, что корреляция положительная. В том случае, если существует обратная зависимость величин (чем больше одна, тем меньше другая), то говорят, что корреляция отрицательная.

Численно взаимосвязь оценивается с помощью коэффициента корреляции Пирсона. Он помогает определить, как сильно меняется одна величина при изменении другой; и принимает значения от –1 до 1. Если с ростом первой величины растет вторая, то коэффициент корреляции Пирсона положительный. Если при изменении одной величины другая остается прежней, то коэффициент равен 0. Если рост одной величины связан с уменьшением другой, коэффициент отрицательный. Чем ближе коэффициент корреляции Пирсона к крайним значениям: 1 или –1, тем сильнее взаимозависимость.

Если значение близко к нулю, значит связь слабая, либо отсутствует вовсе. Бывает, что коэффициент нулевой не от того, что связи между значениями нет, а потому, что у нее более сложный, нелинейный характер, коэффициент корреляции такую связь не берет.

Коэффициент Пирсона находят методом *corr*(). Метод применяют к столбцу с первой величиной, а столбец со второй передают в параметре.

```
print(data['column_1'].corr(data['column_2']))
print(data['column_2'].corr(data['column_1']))
```

Когда в задаче нужно найти попарные взаимосвязи величин, это можно сделать с помощью попарных диаграмм рассеяния. В *Pandas* такую задачу решают не *data.plot*(), а специальным методом: *pd.plotting.scatter_matrix*(*data*).

```
pd.plotting.scatter_matrix(data)
```

Помимо попарных диаграмм рассеяния, можно получить попарный коэффициент корреляции для всех величин. Это можно сделать с помощью матрицы корреляции:

```
data.corr()
```

2.5. ВАЛИДАЦИЯ РЕЗУЛЬТАТОВ

Если объект исследования является обособленным, никак не связанным с ближайшими соседями, то для анализа лучше подходит столбчатый график,

так как он подчеркивает изолированность данных. Столбчатый график строят методом *plot*(), параметром передают тип графика *kind='bar'*.

```
data.plot(y='column', kind='bar')
```

Однако, если объектов много и не все они обладают достаточно высокими целевыми показателями, можно провести их группировку для дальнейшей обработки. Группировку в таком случае можно провести как выборочное переименование.

Выборочно изменяют значения методом *where*(). Ему передают 2 параметра: булев массив и новые значения. Если в булевом массиве *True*, соответствующее ему значение не изменится; а если *False* – значение поменяется на второй параметр метода.

```
data['column'].where(s > control_value, default_value)
```

В некоторых случаях для визуализации данных уместно использовать круговую диаграмму, так как она хороша для показа доли каждого значения от 100% всех. Чтобы получить круговую диаграмму, достаточно в.*plot* задать *kind='pie'*. При этом нужно задать столбец с данным параметром *y*:

```
data.plot(y='column', kind='pie')
```

Работать с данными в зависимости от определенных значений в столбце по отдельности стандартными методами не всегда бывает удобно. Для того, чтобы работать с данными по группам, существует метод *groupby*(), который автоматически перебирает уникальные значения. Ему передается столбец, а он возвращает последовательность пар: значение столбца – срез данных с этим значением. И в дальнейшем можно обрабатывать эти срезы в соответствии с поставленными задачами.

```
for column_value, column_slice in data.groupby('column'):
# do somethin
```

3. СТАТИСТИЧЕСКИЙ АНАЛИЗ ДАННЫХ

3.1. ОПИСАТЕЛЬНАЯ СТАТИСТИКА

Категориальная (качественная) переменная принимает значение из ограниченного набора.

Количественная (численная) переменная принимает числовое значение в диапазоне. Количественные переменные подразделяются на:

- непрерывные, которые могут принимать любое численное значение;
- дискретные, которые могут принимать строго определенные значения.

Гистограмма хорошо подходит для работы с дискретными переменными. Для отображения частот непрерывных переменных нужно что-то другое.

Одним из подходов к визуализации значений непрерывных переменных является разделение множества значений на интервалы и подсчет количества значений, попадающих в каждый интервал.

В *Pandas* при построении гистограммы можно задавать не только количество интервалов – корзин, но и явно указывать их границы:

```
data.hist(bins=[value1, value2, value3, value4,..., valueN])
```

Однако этот подход не может дать полное представление о значениях переменной, так как полученная гистограмма сильно зависит от того, как разбили множество значений на интервалы.

Для того, чтобы решить недостаток разбиения на интервалы, применяется метод, отображающий частоту не высотой столбца в гистограмме, а его площадью. Площадь столбца находят как площадь прямоугольника: длину интервала умножают на высоту столбца.

Найденная площадь – частота непрерывной переменной, а высота столбца – плотность частоты. Гистограмма, использующая в качестве переменной – столбца плотность частоты, называется плотностной гистограммой.

Для того, чтобы оценить, сколько значений попало в любой интервал, не обязательно выбранный для построения, берут два значения и ищут площадь плотностной гистограммы между ними. Полученное число и будет оценкой количества значений, попавших в интервал.

Плотность частоты для непрерывных переменных можно задавать не только прямоугольниками, как на гистограммах, но и кривыми функциями. Работает тот же принцип: площадь между двумя значениями пропорциональна частоте попадания значений в интервал между ними.

Такие характерные значения выборки, как медиана и среднее значение, также называют метрики локации данных: по медиане и среднему можно судить, где примерно расположен набор данных на числовой оси.

Для расчета среднего значения берут все значения датасета – это наиболее полное использование информации при поиске метрики локации. Среднее значение называют алгебраическая метрика локации.

Медиана и квартили просто делят набор данных на части. Медиана – структурная метрика локации.

Для представления о данных недостаточно знать метрики локации, нужно еще понимать, как данные разбросаны вокруг них. У структурной метрики локации есть структурные метрики разброса – квартили.

Для подсчета разброса значений вокруг алгебраической метрики может применяться такой метод: вычисление среднего расстояние между средним значением и всеми остальными значениями переменной.

Ранее предложенный метод подсчета разброса значений вокруг алгебраической метрики имеет право на существование, но он не всегда дает полное представление о разбросе.

Улучшенная метрика разброса – не просто среднее расстояние между значениями датасета и средним, а средний – квадрат этого расстояния. Эта величина называется **дисперсией** (англ. *variance*), ее находят по формуле

$$\sigma^2 = \frac{\sum(\mu - x_i)^2}{n},$$

где μ – среднее арифметическое значение совокупности данных:

$$\mu = \frac{\sum(x_i)}{n}.$$

Библиотека *Numpy* в *Python* содержит большую библиотеку высокоуровневых математических функций. Импортируют ее так:

```
import numpy as np
```

Дисперсию рассчитывают методом *var*():

```
import numpy as np
variance = np.var(x)
```

У дисперсии есть один небольшой недостаток: единица ее измерения – это квадрат исходной величины. Чтобы вернуться к исходной единице измерения, из дисперсии извлекают квадратный корень. Получившаяся величина называется **стандартным отклонением** (standard deviation):

$$\sigma = \sqrt{\frac{\sum(\mu - x_i)^2}{n}}.$$

Стандартное отклонение находят методом *std*() из библиотеки *Numpy*:

```
import numpy as np standard_deviation = np.std(x)
```

Если дисперсия известна заранее, можно применить метод *sqrt*() из библиотеки *Numpy*. Корень из дисперсии будет равен стандартному отклонению:

```
import numpy as np
variance = 2.9166666666666665
standard_deviation = np.sqrt(variance)
```

Для большинства распределений верно правило трех стандартных отклонений, или правило трех сигм. Оно гласит – практически все значения (около 99%) находятся в промежутке:

$$(\mu - 3\sigma, \mu + 3\sigma).$$

Это правило позволяет не только находить интервал, в который наверняка попадут практически все значения интересующей нас переменной, но и искать значения вне этого интервала – часто их называют выбросами.

Многие данные «из жизни» распределены нормально, или симметрично. Однако датасеты могут быть ассиметричными, т.е. иметь скошенность в положительную или отрицательную сторону.

Определить скошенность легко по гистограмме. Для этого нужно представить асимметричную гистограмму как симметричную с «дополнительными» значениями.

Такая гистограмма с дополнительными значениями справа отображает частоту значений в скошенном вправо наборе данных. Его также называют датасетом с положительной скошенностью, ведь дополнительные значения находятся со стороны положительного направления оси.

Скошенный влево датасет получится, если добавить к симметричному набору данных значений слева. Если влево идет отрицательное направление оси, такой набор данных назовут датасетом с отрицательной скошенностью.

Скошенность данных также хорошо иллюстрирует диаграмма размаха.

Чтобы понять, в какую сторону скошен датасет, необязательно строить графики. Достаточно взглянуть на метрики локации: медиану и среднее. Помня о том, что медиана, в отличие от среднего, устойчива к выбросам, легко сделать вывод, что для скошенных вправо данных медиана будет меньше среднего, а для скошенных влево – больше.

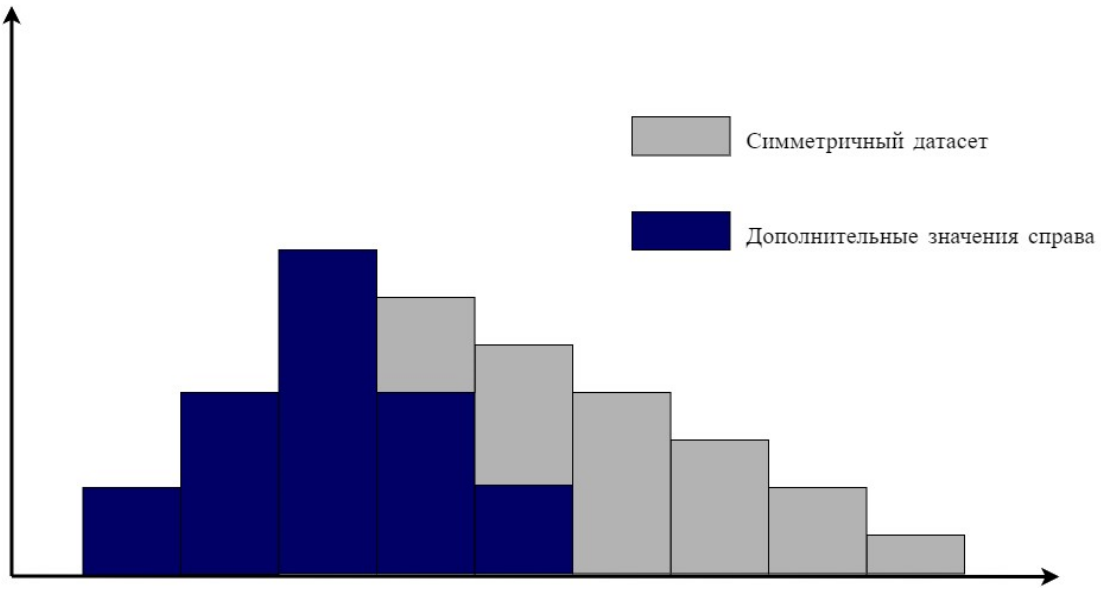

Рис. 5. Асимметричная гистограмма

3.2. ТЕОРИЯ ВЕРОЯТНОСТЕЙ

Эксперимент – это повторяемый опыт, который может окончиться разными исходами, или, как принято говорить, элементарными исходами: исход либо случился, либо не случился.

В простейшем случае эти исходы не отличаются: невозможно предпочесть один из них другому, – и значит вероятность каждого из них одинакова. Такие исходы называются равновероятными. В честном эксперименте (эксперимент с равновероятными исходами) с n элементарными исходами вероятность каждого исхода одинакова и равна $1/n$.

Множество всех элементарных исходов эксперимента принято называть вероятностным пространством. Из него можно выделить подмножества, содержащие в себе некоторое количество элементарных исходов – события.

Невозможное событие – событие, которое не произойдет никогда, вероятность его появления равна 0. Достоверное событие – событие, которое точно произойдет, вероятность его которого равна 1. Вероятность появления других событий находится в промежутке от 0 до 1.

При сохранении условия равновероятности всех элементарных исходов вероятность события – количество исходов, входящих в это событие, деленное на общее количество исходов, т.е. на размер вероятностного пространства.

Закон больших чисел: чем больше раз повторяется эксперимент, тем ближе частота заданного на этом эксперименте события будет к его вероятности.

Это правило можно использовать и в обратную сторону: если не знаем вероятность какого-то события, но можем много раз повторить эксперимент, по частоте выпадания исходов, входящих в это событие, можно судить о его вероятности.

Для отображения пересечения между событиями, используется диаграмма Эйлера–Венна.

Рис. 6. Диаграмма Эйлера–Венна

События А и Б пересекаются, значит существуют элементарные исходы, входящие и в А, и в Б.

Взаимоисключающими называют события, которые не могут произойти одновременно при проведении эксперимента – на диаграмме Эйлера–Венна они не пересекаются.

Вероятность взаимоисключающих событий равна нулю.

События называются независимыми, если наступление одного из них не влияет на вероятность другого. Если события независимы, то вероятность их пересечения равна произведению их вероятностей. Это правило работает и в обратную сторону.

Если взаимоисключающие события охватывают все вероятностное пространство, сумма их вероятностей равна единице.

Рис. 7. Взаимоисключающие события

Взаимоисключаемость событий видна на диаграмме Эйлера–Венна. А вот независимость так просто не обнаружишь, нужно проверять условие равенства произведения вероятностей событий вероятности их пересечения.

Случайная величина – это переменная, которая принимает случайные значения – те значения, которые нельзя предсказать до проведения эксперимента. У эксперимента есть исходы, которые могут описываться как количественно, так и качественно. Случайная же величина определяется на этих исходах численно. Это способ спроецировать исходы эксперимента, как бы они ни определялись, на числовую ось.

Как и все количественные переменные, случайная величина может быть дискретной или непрерывной.

Распределением вероятности случайной величины называется таблица, содержащая всевозможные значения случайной величины и вероятности их появления.

Для хранения числовых таблиц используется тип данных *numpy array* из библиотеки *Numpy*

```
table = np.array([[2,3,4,5,6,7],
[3,4,5,6,7,8],
...
[7,8,9,10,11,12]])
```

Если при работе со словарем необходимо получить список всех ключей словаря, то это можно сделать с помощью метода *keys*(). А список всех значений словаря – с помощью метода *values*().

```
dictionary = {...}
print(dictionary.keys())
print(dictionary.values()
```

Для эксперимента можно задать случайную величину и найти численное значение, к которому она будет в среднем стремится при многократном повторе эксперимента. Это значение называется математическим ожиданием случайной величины.

Если эксперимент состоит из равновероятных элементарных исходов, заданных численно, математическое ожидание будет равно среднему возможных значений.

Математическое ожидание случайной величины – сумма всех значений случайной величины, помноженных на их вероятности:

$$E(X) = \sum p(x_i)\, x_i.$$

Математическое ожидание – аналог метрики локации, только не для датасета, а для случайной величины. Оно показывает, вокруг какого значения распределена случайная величина, и – по закону больших чисел – к какому значению она будет в среднем стремиться при повторе эксперимента.

Поскольку случайная величина распределена вокруг этой «метрики локации», можно найти и меру ее разброса. Для этого нужно найти математическое ожидание квадрата случайной величины – это несложно если учесть, что значения меняются, а их вероятности – нет.

Если знаем математическое ожидание самой случайной величины и ее квадрата, **дисперсию** находят по формуле

$$Var(x) = E(X^2) - (E(X))^2.$$

Эксперименты с двумя возможными исходами называются биномиальными экспериментами. Обычно один из результатов называют успехом, а второй, соответственно, неудачей. Если вероятность успеха равна P, то вероятность неудачи $(1 - P)$.

Количество способов выбрать k успехов из n повторений эксперимента находят по формуле

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

Для вычисления факториала используется библиотека *math* и ее метод *factorial*:

```
from math import factorial
x = factorial(5)
```

Рассмотрим задачу о биномиальном эксперименте (повторении эксперимента с двумя исходами n раз) в общем виде. Если вероятность успеха P и неуспеха $1 - P$, а эксперимент был повторен n раз, то вероятность любого количества успехов k из этих n экспериментов:

$$P(k \text{ успехов из } n \text{ попыток}) = C_n^k p^k (1-p)^{n-k}.$$

Условия, при которых можно утверждать, что случайная величина распределена биномиально:

- проводится конечное фиксированное число попыток n;
- каждая попытка – простой биномиальный эксперимент ровно с двумя исходами;
- попытки независимы между собой;
- вероятность успеха p одинаковая для всех n попыток.

Ключевая теорема в статистике – **центральная предельная теорема**. Если немного упростить, она гласит: «Много независимых случайных величин, сложенных вместе, дают нормальное распределение».

Нормальное распределение описывает множество реальных непрерывных величин. Нормальное распределение определяют два параметра – среднее и дисперсия:

$$X \sim N(\mu, \sigma^2).$$

Эта запись читается так: переменная X распределена нормально со средним μ и дисперсией σ^2, т.е. стандартным отклонением σ.

Для того, чтобы по известным параметрам распределения найти вероятность попадания в те или иные интервалы, надо вызвать два метода из пакета *scipy.stats*: *norm.ppf* и *norm.cdf*:

- *ppf* – функция процентных значений;
- *cdf* – кумулятивная функция распределения.

Обе работают с нормальным распределением, заданным своими средним и стандартным отклонением:

- функция *norm.ppf* выдает значение переменной для известной вероятности интервала слева от этого значения;
- функция *norm.cdf*, наоборот, выдает для известного значения вероятность интервала слева от этого значения.

Чтобы задать нормальное распределение, используется метод *norm*() из пакета *scipy.stats* с двумя аргументами: математическим ожиданием и стандартным отклонением.

Вероятность получить некоторое значение *x*:

```
from scipy import stats as st
# задаем нормальное распределение
distr = st.norm(1000, 100)
x = 1000
result = distr.cdf(x) # считаем вероятность получить значение x
```

С помощью функции *norm.cdf* можно посчитать вероятность получить значение в промежутке от *x*1 до *x*2:

```
from scipy import stats as st
# задаем нормальное распределение
distr = st.norm(1000, 100)
x1 = 900
x2 = 1100
result = distr.cdf(x2) – distr.cdf(x1)
# считаем вероятность получить значение между x1 и x2
```

Для того, чтобы по вероятности получить значение, можно использовать метод *norm.ppf*:

```
from scipy import stats as st
# задаем нормальное распределение
distr = st.norm(1000, 100)
p1 = 0.841344746
result = distr.ppf(p1)
```

При большом количестве повторений биномиального эксперимента биномиальное распределение приближается к нормальному.

Для дискретного биномиального распределения, заданного числом повторов эксперимента n и вероятностью успеха p, математическое ожидание равно $n \cdot p$, а дисперсия $n \cdot p \cdot (1 - p)$.

Если n больше 50, эти параметры биномиального распределения можно взять как среднее и дисперсию для нормального распределения, которое будет достаточно близко описывать биномиальное. Максимально близкое к биномиальному, нормальное распределение задается его математическим ожиданием $n \cdot p$ в качестве среднего значения и дисперсией $n \cdot p \cdot (1 - p)$.

3.3. ПРОВЕРКА ГИПОТЕЗ

Логика проведения статистической проверки гипотез немного другая, по сравнению с механизмами в теории вероятностей. Прежде всего, будем судить о большом объеме данных, генеральной совокупности, по части – выборке.

Для анализа необязательно загружать все данные, достаточно взять небольшую, но репрезентативную, представляющую всю генеральную совокупность, часть данных. Самый простой способ добиться репрезентативности – взять случайную выборку. Из всего датасета генератором случайных чисел отбирают случайные элементы. По ним будут судить обо всей генеральной совокупности.

Она может состоять из нескольких неравных по размеру частей, сильно отличающихся по исследуемому параметру. Тогда есть смысл взять пропорциональные случайные выборки из этих частей, и потом соединить между собой. Получается стратифицированная выборка, более репрезентативная, чем просто случайная. Она так называется, потому что генеральную совокупность разбили на **страты** – группы, объединенные общим признаком. Случайные выборки получают уже из них.

По выборке судят о генеральной совокупности – точнее, об ее статистических параметрах. Обычно достаточно оценить среднее и дисперсию, чтобы сделать выводы о равенстве или неравенстве средних значений исследуемых совокупностей. Нас будет интересовать именно такая постановка задачи.

Что можно сказать о среднем и дисперсии генеральной совокупности по среднему и дисперсии, посчитанным на выборке, или выборочному среднему и выборочной дисперсии? Почти все, при условии, что выборка достаточно велика.

Одна из формулировок центральной предельной теоремы звучит так: если в выборке достаточно наблюдений, выборочное распределение выборочного среднего из любой генеральной совокупности распределено нормально вокруг среднего этой генеральной совокупности. «Любая генеральная совокупность» означает, что сама генеральная совокупность может быть распределена как угодно. Датасет из средних значений выборок все равно будет нормально распределен вокруг среднего всей генеральной совокупности.

Стандартное отклонение выборочного среднего от настоящего среднего генеральной совокупности называется **стандартной ошибкой** и находится по формуле

$$E.S.E. = \frac{S}{\sqrt{n}},$$

где *E.S.E.* – оцененная стандартная ошибка. «Оцененная» – имея только выборку и не зная точную ошибку, она оценивается исходя из имеющихся данных; S – оценка стандартного отклонения генеральной совокупности; n – размер выборки. Раз корень из n стоит в знаменателе, стандартная ошибка уменьшается с увеличением размера выборки.

Никакие экспериментально полученные данные никогда не подтвердят какую-либо гипотезу. Это наше фундаментальное ограничение. Данные могут лишь не противоречить ей или, наоборот, показывать крайне маловероятные результаты (при условии, что гипотеза верна). Но и в том, и в другом случае нет оснований утверждать, что выдвинутая гипотеза доказана.

Допустим, данные гипотезе не противоречат, тогда ее не отвергаем. Если же приходим к выводу, что получить такие данные в рамках этой гипотезы вряд ли возможно, появляется основание отбросить эту гипотезу.

Типичные статистические гипотезы касаются средних значений генеральных совокупностей и звучат так:

- среднее генеральной совокупности равно конкретному значению;
- средние двух генеральных совокупностей равны между собой.

Алгоритм проверки статистических гипотез всегда начинается с формулирования гипотез. Сначала формулируется нулевая гипотеза H_0. Например, «среднее рассматриваемой генеральной совокупности равно A», где A – некоторое число. Исходя из H_0 формулируется альтернативная гипотеза H_1. Для этой H_0 она звучит как «среднее генеральной совокупности не равно A». H_0 всегда формулируется так, чтобы использовать знак равенства.

Построим распределение на предположении, что гипотеза H_0 верна. В нашем случае это будет нормальное распределение вокруг интересующего нас параметра – среднего. Дисперсия, или стандартное отклонение, оценивается по данным выборки.

Рис. 7. Распределение на предположении верности нулевой гипотезы

Для нормального распределения вероятность попасть в тот или иной интервал равна площади графика над этим интервалом. В районе среднего значения и в некотором диапазоне вокруг него будут значения, которые весьма вероятно получить случайно.

Как определить, где еще не отвергаем нулевую гипотезу, а где пора? Критические значения задаются выбранным уровнем значимости проверки гипотезы. Уровень значимости – это суммарная вероятность того, что измеренное эмпирически значение окажется далеко от среднего. На графике уровень значимости – 5%, по 2,5% с каждой стороны. Но аналитик может выбрать и уровень в 1% или даже 0,01%. Если наблюдаемое на выборке значение попадает в эту зону, вероятность такого события при верной нулевой гипотезе признается слишком малым, значит, у нас есть основание отвергнуть нулевую гипотезу. Когда значение попадает в зону «Не отвергаем H_0», то оснований отвергать нулевую гипотезу нет. Считаем, что эмпирически полученные данные не противоречат нулевой гипотезе.

В *Python* существует метод, который просто возвращает статистику разности между средним и тем значением, с которым его сравнивают. Главное – уровень значимости, на котором они находятся друг от друга – *p-value*.

Статистика разности – это количество стандартных отклонений между сравниваемыми значениями, если оба распределения привести к стандартному нормальному распределению со средним 0 и стандартным отклонением 1. По этой цифре сложно сориентироваться.

Есть смысл принимать решение о принятии или отвержении нулевой гипотезы по *p-value*. Это вероятность получить наблюдаемый результат при условии, что нулевая гипотеза верна. Если это значение больше 10%, то нулевую гипотезу точно не стоит отвергать. Меньше – возможно, есть основания отвергнуть нулевую гипотезу. Общепринятые пороговые значения – 5 и 1%. Если признаете 5%-ную вероятность слишком малой и на этом основании отвергаете нулевую гипотезу, то в среднем в одном исследовании из 20 значимый эффект будет обнаружен не потому, что она неверна, а за счет случайной

ошибки. Окончательное решение, какой порог считать достаточным, всегда остается за аналитиком.

Если эмпирических значений немного (не больше 20 – 30), то соответствующее нулевой гипотезе распределение будет почти нормальным, но немного шире. Больший разброс объясняется недостатком данных.

Такое распределение называется *t*-распределением Стьюдента. Его придумал Уильям Госсет, сотрудник ирландской компании *Guinness*, когда исследовал урожайность сортов ячменя. Данных было мало и приходилось делать статистические поправки. Компания запрещала сотрудникам публикацию научных результатов, поэтому Госсет напечатал статью под псевдонимом «Студент» (от англ. *student*, читается как «стьюдент»).

Количество наблюдений (за вычетом единицы) при построении распределения Стьюдента официально называется «количество степеней свободы». Когда оно растет, распределение стремится к нормальному распределению с тем же средним.

Распределение Стьюдента называется *t*-распределением, а статистический тест с его использованием – *t*-тестом. Аналитику можно не беспокоиться, достаточно ли близко *t*-распределение для его данных подобралось к нормальному: методы библиотеки *scipy* сделают это автоматически.

Для проверки гипотезы о равенстве среднего генеральной совокупности некоторому значению можно использовать метод *scipy.stats.ttest_1samp*. Параметры метода: *array* – массив, содержащий выборку, *popmean* (англ. *population mean*, «среднее значение генеральной совокупности», буквально «среднее в популяции») – предполагаемое среднее, на равенство которому делается тест. После выполнения метод вернет статистику разности между *popmean* и выборочным средним из *array*, а также уровень значимости:

```
from scipy import stats as st
interested_value = 120
results = st.ttest_1samp(
     array,
     interested_value)
print('p-значение: ', results.pvalue)
```

Когда генеральных совокупностей две, бывает нужно сопоставить их средние. Чтобы проверить гипотезу о равенстве среднего двух генеральных совокупностей по взятым из них выборкам, можно применить метод *scipy.stats.ttest_ind*(). Методу передают параметры: *array*1, *array*2 – массивы, содержащие выборки; *equal_var* – необязательный параметр, задающий считать ли равными дисперсии выборок. Если есть основание полагать, что выборки взяты из схожих по параметрам совокупностей, тогда укажите *equal_var = True*, и дисперсия каждой выборки будет оценена по объединенному датасету из двух выборок, а не для каждой по отдельности по значениям в ней самой. Это позволяет получить более точные результаты, но только в том случае, если считать примерно равными дисперсии генеральных совокупностей, из которых взяты выборки. Иначе нужно указать *equal_var = False*, по умолчанию он задан как *equal_var = True* (если вообще его не указывать).

```
from scipy import stats as st
sample_1 = [...]
sample_2 = [...]
results = st.ttest_ind(
      sample_1,
      sample_2)
print('p-значение: ', results.pvalue)
```

Когда генеральная совокупность одна, полезно понять, равно ли себе среднее этой совокупности до и после изменения. Парная выборка означает, что измеряем некоторую переменную для одних и тех же единиц. Чтобы проверить гипотезу о равенстве средних двух генеральных совокупностей для зависимых (парных) выборок в *Python* применяется функция *scipy.stats.ttest_rel*().

```
from scipy import stats as st before = [...]
after = [...]
results = st.ttest_rel(
 before,
after)
print('p-значение: ', results.pvalue)
```

4. СБОР И ХРАНЕНИЕ ДАННЫХ

4.1. ИЗВЛЕЧЕНИЕ ДАННЫХ ИЗ ВЕБ-РЕСУРСОВ

Когда предоставленных данных мало для решения задачи, аналитик добывает информацию самостоятельно. Такое обогащение данных позволяет не только учесть больше факторов, но и выявить новые закономерности, прийти к неожиданным выводам. Аналитики обогащают имеющуюся информацию в Интернете. Сперва находят ценные для исследования веб-ресурсы, а затем извлекают из них нужные данные. Этот процесс называют *Web Mining*, или парсинг.

Чтобы получить необходимую информацию из веб-страниц, нужно заполучить код страницы и контент внутри него. Чтобы получить данные с сервера, вам понадобится метод *get*(). Для отправления *HTTP*-запросов подключают библиотеку *Requests*:

```
import requests
```

Метод *get*() библиотеки *Requests* выступает в роли браузера. Он принимает ссылку на сайт в качестве аргумента. Метод отправит *get*-запрос на сервер, обработает полученный оттуда результат и вернет объект *response*.

Response – специальный объект, содержащий ответ сервера на *HTTP*-запрос:

```
req = requests.get(URL) # сохраняем объект Response в переменную req
```

Объект *Response* содержит ответ сервера: код состояния, содержание запроса и код самой *HTML*-страницы. Атрибуты объекта *Response* позволяют возвращать не все данные с сервера, а только нужные для анализа.

Например, объект *Response* с атрибутом *text* «отдаст» лишь текстовое содержание запроса:

```
print(req.text)
# название атрибута пишут после объекта Response, разделяя точкой
```

Атрибут *status_code* отвечает на вопрос: отправил сервер ответ или возникла какая-то ошибка:

```
print(req.status_code)
```

К сожалению, не все запросы возвращаются с данными. Иногда результатом запроса бывает ошибка: в зависимости от типа ее обозначают специальным кодом.

Для поиска строк в больших текстах используется мощный инструмент – регулярные выражения. Регулярное выражение – правило для поиска подстрок (фрагментов текста внутри строк). Регулярные выражения позволяют создавать сложные правила, так что одно выражение вернет несколько подстрок.

Для работы с регулярными выражениями в *Python* импортируют библиотеку *re*. Дальше поиск ведется в два этапа. Сначала создают шаблон регулярного выражения. Это алгоритм, по которому нужно искать строку в тексте. Затем готовый шаблон передают специальным методам библиотеки *re*, которые ищут, заменяют и удаляют нужные символы. Таким образом, шаблон определяет, что и как искать, а метод – что с этим потом делать.

В таблице 2 приведены простейшие шаблоны регулярных выражений. Сложные регулярные выражения состоят из их комбинаций.

Самые распространенные задачи аналитика:

- найти подстроку в строке;
- разбить строки на подстроки на основании шаблона;
- заменить части строки на другую строку.

Вот какие методы библиотеки *re* для этого понадобятся:

1) *search*(*pattern*, *string*) ищет шаблон pattern в строке *string*. Хотя *search*() ищет шаблон во всей строке, возвращает он только первую найденную подстроку:

```
import re
print(re.search(pattern, string))
```

2. Синтаксис регулярных выражений

Регулярное выражение	Описание	Пример	Пояснение
[]	Один из символов в скобках	[*a*-]	*a* или -
[^…]	Отрицание	[^*a*]	любой символ, кроме «*a*»
-	Интервал	[0-9]	интервал: любая цифра от 0 до 9
.	Один любой символ, кроме перевода строки	*a*.	*as*, *a*1, *a_*
d (аналог [0-9])	Любая цифра	*a**d* | *a* [0-9]	*a*1, *a*2, *a*3
w	Любая буква, цифра или _	*a**w*	*a_*, *a*1, *ab*
[*A*-*z*]	Любая латинская буква	*a*[*A*-*z*]	*ab*
[А-я]	Любая буква кириллицы	а[А-я]	ая
?	Ноль или одно вхождение	*a*?	*a* или ничего
+	Одно и более вхождений	*a*+	*a* или *aa*, или *aaa*
*	Ноль и более вхождений	*a**	ничего или *a*, или *aa*
^	Начало строки	^*a*	*a*1234, *abcd*
$	Конец строки	*a*$	1*a*, *ba*

Метод *search*() возвращает объект типа *match*. Параметр *span* указывает диапазон индексов, подходящих под шаблон. В параметре *match* указано само значение подстроки.

Если нам не нужны дополнительные сведения о диапазоне, выведем только найденную подстроку методом *group*():

```
import re
print(re.search(pattern, string).group())
```

2) *split*(*pattern*, *string*) разделяет строку *string* по границе шаблона pattern.

```
import re
print(re.split(pattern, string))
```

Строка разделена на несколько частей. Границы деления строки проходят там, где метод встретил указанный в аргументе шаблон. Количеством делений строки можно управлять. За это отвечает параметр *maxsplit* метода *split*() (по умолчанию равен 0).

```
import re
print(re.split(pattern, string, maxsplit = num_split))
```

3) *sub*(*pattern*, *repl*, *string*) ищет подстроку по шаблону *pattern* в строке *string* и заменяет ее на подстроку *repl*.

```
import re
print(re.sub(pattern, repl, string))
```

4) Метод *findall*(*pattern*, *string*) возвращает список всех подстрок в *string*, удовлетворяющих шаблону *pattern*. А не только первую подходящую подстроку, как *search*().

```
import re print(re.findall(pattern, string))
```

Метод *findall*() удобен тем, что можно сразу посчитать количество повторяющихся подстрок в строке функцией *len*():

```
import re print(len(re.findall(pattern, string)))
```

Достать данные из строки, которая содержит код страницы, вручную сложно. Чтобы решить проблему, обратимся к возможностям библиотеки *BeautifulSoup*. Методы библиотеки *BeautifulSoup* превращают *HTML*-файл в древовидную структуру. После этого нужный контент можно отыскать по тегам и атрибутам.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(req.text, 'lxml')
```

Первый аргумент – это данные, из которых будет собираться древовидная структура. Второй аргумент – синтаксический анализатор, или парсер. Он отвечает за то, как именно из кода веб-страницы получается «дерево».

Парсеров много, они создают разные структуры из одного и того же *HTML*-документа. За высокую скорость работы выбрали анализатор *lxml*. Есть и другие, например, *html.parser, xml* или *html*5*lib*.

После превращения кода страницы в дерево можно искать данные различными методами. Первый метод поиска называется *find*(). В *HTML*-документе он находит первый элемент, имя которого ему передали в качестве аргумента, и возвращает его весь, с тегами и контентом.

```
tag_content = soup.find(tag)
```

Чтобы посмотреть контент без тега, вызывают метод *text*. Результат возвращается в виде строки:

```
tag_content.text
```

Существует и другой метод поиска – *find_all*. Этот метод находит все вхождения определенного элемента в *HTML*-документе и возвращает список:

```
tag_content = soup.find_all(tag)
```

Методом *text* вычленим только контент из тегов:

```
for tag_content in soup.find_all(tag): print(tag_content.text)
```

У методов *find*() и *find_all*() есть дополнительный фильтр поиска элементов страницы – параметр *attrs*. Он используется для поиска по идентификаторам и классам. Их имена уточняют в панели разработчика.

Параметру *attrs* передают словарь с именами и значениями атрибутов:

```
soup.find(tag, attrs={"attr_name": "attr_value"})
```

Чтобы получать данные из внешних ресурсов, устроенных намного сложнее обычной *HTML*-страницы, аналитики отправляют *GET*-запросы к сторонним сервисам через специальный интерфейс передачи данных – *API*.

API позволяет разработчику взаимодействовать с системой, не беспокоясь о том, как именно она реализована. Для этого *API* предоставляет «инструкцию» – набор методов с определенными параметрами. Библиотека *requests* позволяет

передавать параметры в *URL*. Ключевому слову *params* в *GET*-запросе передают словарь с параметрами.

```
import requests
BASE_URL = "https://weather.data-analyst.praktikum-
services.ru/v1/forecast"
# URL метода
params = { # словарь с параметрами запроса
      "city" : "Moscow", # определяем город
      "hours" : True # указываем, что нужен почасовой прогноз
}
response = requests.get(BASE_URL, params=params) print(response.text)
```

Отвечая на запрос, сервер возвращает структурированные данные в одном из специальных форматов. Самый распространенный из них – *JSON* (*JavaScript Object Notation*). Вот как выглядят данные в формате *JSON*:

```
[
      {
      "name": "Генерал Слокам",
      "date": "15 июня 1904 года"
      },
      {
      "name": "Каморта",
      "date": "6 мая 1902 года"
      },
      {
      "name": "Норье",
      "date": "28 июня 1904 года"
      }
]
```

JSON начинается с фигурных скобок, если содержит один объект, или с квадратных – если список объектов. Каждый элемент *JSON* записан в фигурных скобках, внутри которых пары ключ : значение.

Значениями ключей могут быть строки, числа, булевы значения, *null*, массивы, объекты. *JSON* не может содержать функций, переменных или комментариев. Обратите внимание, что ключи взяты в двойные кавычки – в *JSON* это обязательно.

В *Python* есть встроенный модуль для работы с данными в формате *JSON*. Его метод *json.loads*() конвертирует строки в формате *JSON*:

```
import json
x = '[{"name": "Генерал Слокам", "date": "15 июня 1904 года"}, {"name":
"Каморта", "date": "6 мая 1902 года"}]'
y = json.loads(x)
     for i in y:
     print('Name : {0}, date : {1}'.format(i['name'], i['date']))
```

```
Name : Генерал Слокам, date : 15 июня 1904 года
Name : Каморта, date : 6 мая 1902 года
```

Метод *json.dumps*(), наоборот, конвертирует данные из *Python* в формат *JSON*. При работе с буквами кириллицы указывают аргумент *ensure_ascii=False* :

```
out = json.dumps(y, ensure_ascii=False)
print(out)
```

```
[{"name": "Генерал Слокам", "date": "15 июня 1904 года"}, {"name":
"Каморта", "date": "6 м ая 1902 года"}]
```

4.2. SQL КАК ИНСТРУМЕНТ РАБОТЫ С ДАННЫМИ

База данных – это хранилище структурированной информации. Сущности – группы объектов с общими характеристиками. Объект – отдельный представитель сущности. Будем рассматривать реляционные базы данных. В них сущности – это таблицы, а объекты – строки таблиц. Для работы с базами данных используется СУБД – система управления базами данных. Это комплекс про-

грамм, который позволяет создать базу данных, наполнить ее новыми таблицами, отобразить содержимое, редактировать существующие таблицы. Одна из самых популярных СУБД – *PostgreSQL*.

Таблица – это совокупность строк и столбцов. Столбцы таблицы называются полями. В них обозначают характеристики объекта. У каждого поля есть уникальное имя и характерный тип данных. Строки таблицы называют записями. Каждая строка – информация об одном объекте. Ячейка – место пересечения строки и столбца. Для того, чтобы записи в таблице базы данных определялись однозначно, используется особое поле – первичный ключ. Все значения в этом поле уникальны. Бывают в таблицах не только простые первичные ключи из одного поля, но и составные первичные ключи. Они состоят из нескольких полей.

SQL – язык, предназначенный для управления данными в реляционной базе. Основные особенности синтаксиса SQL:

1) начало однострочного комментария обозначают двумя дефисами: --
2) многострочный комментарий заключают в /* косые черты со звездочками */
3) команды пишут заглавными буквами
4) каждый запрос заканчивается точкой с запятой ; :

```
SELECT * FROM название_таблицы;
-- Запрос на выборку всех данных из таблицы заканчивается ";"
SELECT * FROM название_таблицы
WHERE название_столбца IN (1,7,9);
-- Запрос на выборку по условию тоже заканчивается ";"
```

Чтобы выбрать данные из таблиц, нужно написать запрос. Запрос – это сформулированное в соответствии с синтаксисом *SQL* требование. В запросе объявляют, какие данные выбрать, и как именно их обработать. Нужную выборку получают оператором *SELECT*. Синтаксис запроса с *SELECT* такой:

```
SELECT
      название_столбца_1,
      название_столбца_2,
      название_столбца_3...
FROM
      название_таблицы;
      --Выбрать один столбец из таблицы
```

В запросе два ключевых слова: *SELECT* и *FROM*. *SELECT* выбирает нужные столбцы из таблицы базы данных. Чтобы выбрать из таблицы все столбцы, добавьте к оператору *SELECT* символ * (звездочка). *FROM* указывает, из какой таблицы брать данные.

Начало условия, по которому отбираются данные, обозначают командой *WHERE*. Проверка на соответствие условию проходит в каждой строке таблицы. В условиях используются математические операторы сравнения:

```
SELECT
      название_столбца_1,
      название_столбца_2 --выбираем названия столбцов
FROM
      название_таблицы --указываем таблицу
WHERE
      условие; --определяем условие, по которому будем отбирать строки
```

В SQL, как и в Python, есть логические операторы: *AND, OR, NOT*.

Они позволяют делать выборку по нескольким условиям.

```
SELECT
      *
FROM
      название_таблицы
WHERE
      условие_1 AND условие_2;
--Выбираются строки, которые соответствуют сразу обоим условиям
SELECT
      *
```

```
FROM
    название_таблицы
WHERE
    условие_1 OR условие_2;
--Выбираются строки, которые соответствуют хотя бы одному из условий
SELECT
    *
FROM
    название_таблицы
WHERE
    условие_1 AND NOT условие_2;
/*Выбираются строки, которые соответствуют условию_1 и не соответствуют условию_2*/
```

Если нужно сделать выборку, условие которой – нахождение значения поля в определенном диапазоне, применяется конструкция *BETWEEN*. Границы диапазона в *BETWEEN* включены в результирующую выборку:

```
SELECT
    *
FROM
    название_таблицы
WHERE
    поле_1 BETWEEN значение_1 AND значение_2;
/*Выбираются строки, в которых значение в поле_1 находится между значение_1 и значение_2, включительно*/
```

Когда нужно сделать выборку, в которой все значения поля находятся в определенном списке, используется оператор *IN*. После *IN* указывают список значений, которые нужно включить в результат:

```
SELECT
    *
FROM
    название_таблицы
WHERE
    название_столбца IN ('значение_1','значение_2','значение_3');
```

Если в списке должны быть числа, их указывают через запятую: *IN* (3,7,9). Строки тоже через запятую, но в одинарных кавычках: *IN* ('значение_1','значение_2','значение_3'). Дату и время обозначают так: *IN* ('чч-мм-ГГГГ','чч-мм-ГГГГ').

Как и в *Python*, в *SQL* есть функции для подсчета общего количества строк, суммы, среднего значения, максимума и минимума. Такие функции называют агрегирующие. Они собирают, или агрегируют все объекты группы, чтобы уже по ним вычислить нужные значения. Пример формата запроса с агрегирующей функцией:

```
SELECT
    АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле) AS here_you_are
/*here_you_are – имя столбца, в котором сохранятся результаты работы функции*/
FROM
    TABLE
```

После вызова агрегирующей функции имя столбца выводится в неудобном виде. Чтобы этого избежать, применяют команду *AS*, а потом записывают новое имя столбца. Функция *COUNT* (англ. «подсчет») возвращает количество строк в таблице:

```
SELECT
    COUNT(*) AS cnt
FROM
    table
```

В зависимости от задачи количество строк считают по-разному:

- *COUNT*(*) возвращает общее количество строк в таблице;
- *COUNT*(*column*) возвращает число строк в столбце *column* ;

COUNT(*DISTINCT column*) (англ. *distinct*, «отдельный, особый») возвращает количество уникальных строк в столбце.

Функция *SUM*(*column*) возвращает сумму по столбцу *column*. При выполнении функции пропуски игнорируются.

AVG (*column*) возвращает среднее значение по столбцу column. Минимальное и максимальное значения находят функциями *MIN*() и *MAX*().

Некоторые агрегирующие функции работают только с числовыми типами данных. Данные в выборке выглядят как числа, но в базе данных хранятся как строки. В жизни такое встречается часто – обычно из-за ошибок проектирования базы данных. Изменим тип данных столбца в самом *SQL*-запросе. Типы преобразуют конструкцией *CAST*:

CAST (название_столбца *AS* тип_данных)

Название_столбца – это поле, тип данных которого нужно преобразовать. Тип_данных – тип, в который данные нужно перевести. Есть и другая форма записи:

название_столбца :: тип_данных

В *SQL* используются следующие типы данных.

1. Числовые типы данных:

- *integer* – целочисленный тип, аналогичный типу *int* в *Python*. В *PostgreSQL* диапазон целых чисел от 2147483648 до 2147483647;
- *real* – число с плавающей точкой, как *float* в *Python*. Точность числа типа *real* до 6 десятичных разрядов.

2. Строковые типы данных:

- 'Практика' – значение строкового типа, в *SQL* запросе его заключают в одинарные кавычки;
- *varchar*(*n*) – строка переменной длины, где *n* – ограничение. Этот тип данных похож на *string* в *Python*, но в отличие от него ограничен по длине: в поле можно занести любую строку короче, чем *n* символов;
- *text* – строка любой длины. Полный аналог *string* в *Python*.

3. Дата и время. Любые вводимые значения даты или времени заключают в одинарные кавычки:

- *timestamp* – дата и время. Этот тип аналогичен *datetime* в *Pandas*. В формате *timestamp* чаще всего хранят события, происходящие несколько раз за день. Например, логи пользователей сайта;

- *date* – дата;
- *boolean* – логический тип данных. В *PostgreSQL* есть три варианта значений *TRUE* – «истина», *FALSE* – «ложь» и *NULL* – «неизвестно».

Если данные нужно разделить на группы по значениям полей, применяют команду *GROUP BY*:

```
SELECT
      поле_1,
      поле_2,
      ...,
      поле_n,
      АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле) AS here_you_are
FROM
      таблица
WHERE -- если необходимо
      условие
GROUP BY
      поле_1,
      поле_2,
      ...,
      поле_n
```

После команды *GROUP BY* перечисляют все поля из блока *SELECT*. Саму агрегирующую функцию включать в блок *GROUP BY* не нужно – с ней запрос не выполнится. *GROUP BY* в *SQL* работает аналогично методу *groupby*() в *Pandas*.

Конструкция *GROUP BY* работает для всех агрегирующих функций: *COUNT*(), *AVG*(), *SUM*(), *MAX*(), *MIN*(). Можно вызывать несколько функций сразу.

Итоги анализа обычно представляют в определенном порядке. Чтобы сортировать данные по указанным полям, применяют команду *ORDER BY*:

```
SELECT
      поле_1,
      поле_2,
      ...,
      поле_n,
      АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле) AS here_you_are
FROM
      таблица
WHERE -- если нужно
      условие
GROUP BY
      поле_1,
      поле_2,
      ...,
      поле_n,
ORDER BY -- если необходимо, перечисляем только те поля,
--по которым хотим отсортировать таблицу
      поле_1,
      поле_2,
      ...,
      поле_n,
      here_you_are
```

В отличие от *GROUP BY*, в блоке с командой *ORDER BY* перечисляем только те поля, по которым хотим сортировать.

У команды *ORDER BY* два аргумента. Они отвечают за порядок сортировки в столбцах:

- *ASC* сортирует данные в порядке возрастания. Это аргумент ORDER BY по умолчанию.
- *DESC* сортирует данные по убыванию.

Аргументы команды *ORDER BY* указывают сразу после поля, по которому сортировали данные:

```
ORDER BY
	название_поля DESC -- сортируем данные по убыванию
ORDER BY
	название_поля ASC -- сортируем данные по возрастанию
```

Команда *LIMIT* ограничивает количество строк в выводе. Ее всегда указывают последней в запросе. После *LIMIT* указывают требуемое число строк – *n*.

```
SELECT
	поле_1,
	поле_2,
	...,
	поле_n,
	АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле) AS here_you_are
FROM
	таблица
WHERE -- если необходимо
	условие
GROUP BY
	поле_1,
	поле_2,
	...,
	поле_n,
ORDER BY -- если необходимо, перечисляем только те поля,
--по которых хотим отсортировать таблицу
	поле_1,
	поле_2,
	...,
	поле_n,
	here_you_are
LIMIT -- если необходимо
	n -- n-максимальное количество строк, которое вернет такой запрос
```

Для того, чтобы применить фильтр по строкам, используется конструкция *WHERE*. В этой конструкции в качестве параметров используются строки таб-

лицы. Если же в фильтре нужно использовать результаты применения агрегированных функций, используется аналог *WHERE* – конструкция *HAVING*:

```
SELECT
      поле_1,
      поле_2,
      ...,
      поле_n,
      АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле) AS here_you_are
FROM
      TABLE
WHERE -- если необходимо
      условие
GROUP BY
      поле_1,
      поле_2,
      ...,
      поле_n
HAVING
      АГРЕГИРУЮЩАЯ_ФУНКЦИЯ(поле_для_группировки) > n
ORDER BY -- если необходимо, перечисляем только те поля,
--по которых хотим отсортировать таблицу
      поле_1,
      поле_2,
      ..., поле_n,
      here_you_are
LIMIT -- если необходимо
      n
```

В результирующую выборку попадут только те строки, для которых результат агрегирующей функции соответствует условию блоков *HAVING* и *WHERE*. Если команды *HAVING* и *WHERE* так похожи, почему нельзя записать все условия в одной из них? Дело в том, что *WHERE* отрабатывает перед группировкой данных и расчетом агрегирующей функции. Потому задать фильтр на результат агрегирующей функции в *WHERE* нельзя. И здесь выручает *HAVING*.

Обратите внимание на порядок команд: *GROUP BY*; *HAVING*; *ORDER BY*. Указывайте команды строго в этой последовательности, иначе запрос не выполнится.

Две основные функции для работы со временем и датой – *EXTRACT* и *DATE_TRUNC*. Обе функции вызывают в блоке *SELECT*. Шаблон функции *EXTRACT*:

```
SELECT
    EXTRACT(часть_даты FROM столбец) AS новый_столбец_с_датой
    FROM
    Таблица_со_всеми_датами
```

Название функции определяет ее суть. *EXTRACT* извлекает из даты нужную часть: год, месяц, минуту. Что еще можно получить вызовом *EXTRACT*:

- *century* – век;
- *day* – день;
- *doy* – день года: от 1 до 365/366;
- *isodow* – день недели: понедельник – 1, воскресенье – 7;
- *hour* – час;
- *milliseconds* – миллисекунда;
- *minute* – минута;
- *second* – секунда;
- *month* – месяц;
- *quarter* – квартал;
- *week* – неделя в году;
- *year* – год.

DATE_TRUNC усекает дату до часа, дня или месяца. В отличие от *EXTRACT*, часть, до которой нужно усечь дату, записывают как строку. А столбец, откуда берут данные о времени, указывают через запятую:

```
SELECT
      DATE_TRUNC('часть_даты_до_которой_усекаем', столбец) AS
новый_столбец_с_датой
FROM
      Таблица_со_всеми_датами
```

Часть даты, до которой данные нужно «обнулить», указывают в аргументе функции *DATE_TRUNC*, например, *'microseconds'* – микросекунды.

Подзапрос – это запрос в запросе, также называемый внутренним запросом. Он используется для получения нужной информации для применения во внешнем запросе. Подзапросы могут выполняться в разных частях запроса. Если подзапрос записать в блоке *FROM*, то *SELECT* выберет данные из таблицы, полученной в результате работы подзапроса. Имя этой таблицы указывают во внутреннем запросе, к ее столбцам обращаются во внешнем. Подзапрос записывают в круглых скобках:

```
SELECT
ПОДЗАПРОС_1.название_столбца,
ПОДЗАПРОС_1.название_столбца_2
FROM
-- Для лучшей читабельности кода, переносите подзапрос на новую строку
-- отделяйте подзапросы отступами
      (SELECT
      название_столбца,
      название_столбца_2
FROM
      название_таблицы
WHERE
      название_столбца = значение) AS ПОДЗАПРОС_1;
-- не забывайте давать имя подзапросу в блоке FROM
```

Внутренние запросы могут понадобиться в разных блоках внешнего запроса. Например, устроим подзапрос в блоке *WHERE*. Тогда выберутся данные из столбца со значениями, сгенерированными в результате работы подзапроса:

```
SELECT
      название_столбца,
      название_столбца_1
FROM
      название_таблицы
WHERE
      название_столбца =
      (SELECT
      столбец_1
      FROM
      название_таблицы_2
      WHERE
      столбец_1 = значение)
```

Дополним шаблон конструкцией *IN*, чтобы собирать данные из нескольких столбцов:

```
SELECT
      название_столбца,
      название_столбца_1
FROM
      название_таблицы
WHERE
      название_столбца IN
      (SELECT
      столбец_1
FROM
      название_таблицы_2
WHERE
      столбец_1 = значение_1 OR столбец_1 = значение_2)
```

Когда столбец таблицы содержит в себе значения поля другой таблицы, его называют внешний ключ. Он отвечает за связь между таблицами. Существуют связи трех типов:

- «один к одному»;

- «один ко многим»;
- «многие ко многим».

Один к одному значит, что строка в первой таблице связана с одной единственной строкой во второй таблице. По сути такая связь – расщепление одной таблицы на две. Это редкий тип связи, применяется в основном из соображений безопасности.

Один ко многим – тип связи, когда каждая строка в одной таблице соответствует многим строкам в другой таблице.

Многие ко многим – тип связи, когда несколько строк одной таблицы соответствуют нескольким строкам другой таблицы. Такая связь реализуется за счет стыковой таблицы. Она соединяет первичные ключи обеих таблиц.

Устройство базы данных иллюстрируют ER-диаграммы. На них изображены таблицы и связи между ними.

Таблицы на *ER*-диаграммах изображают прямоугольником, разделенным на два. В верхней части указывают название таблицы.

В нижней части перечисляют поля таблицы с указанием, к каким ключам они относятся – первичным или внешним. Сами ключи обычно обозначают подписями *PK* (первичный ключ) и *FK* (внешний ключ).

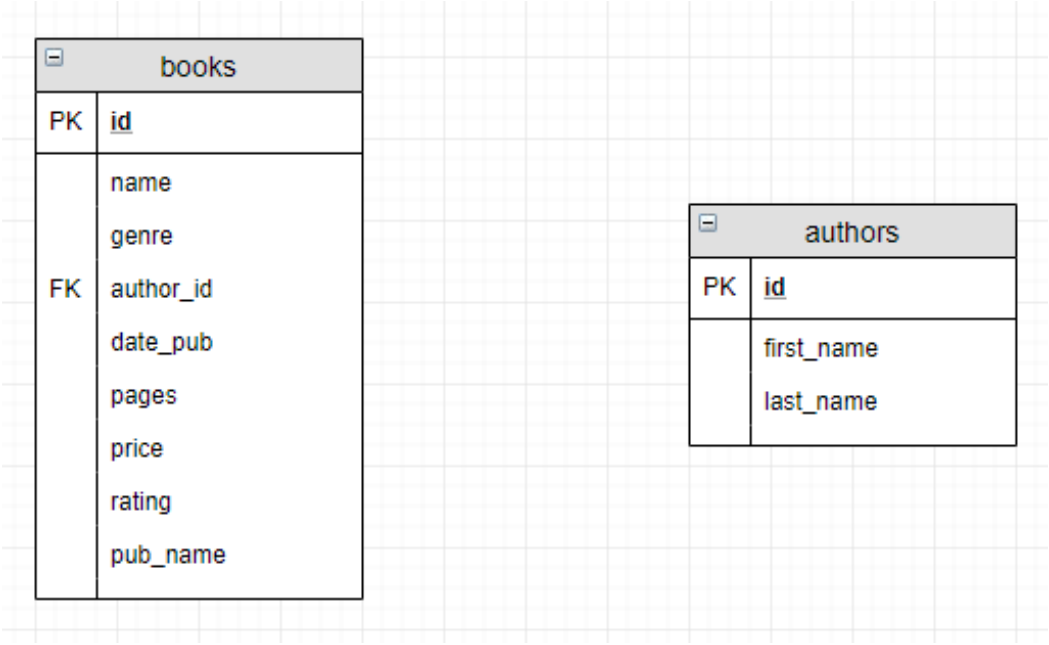

Рис. 8. *ER*-диаграмма

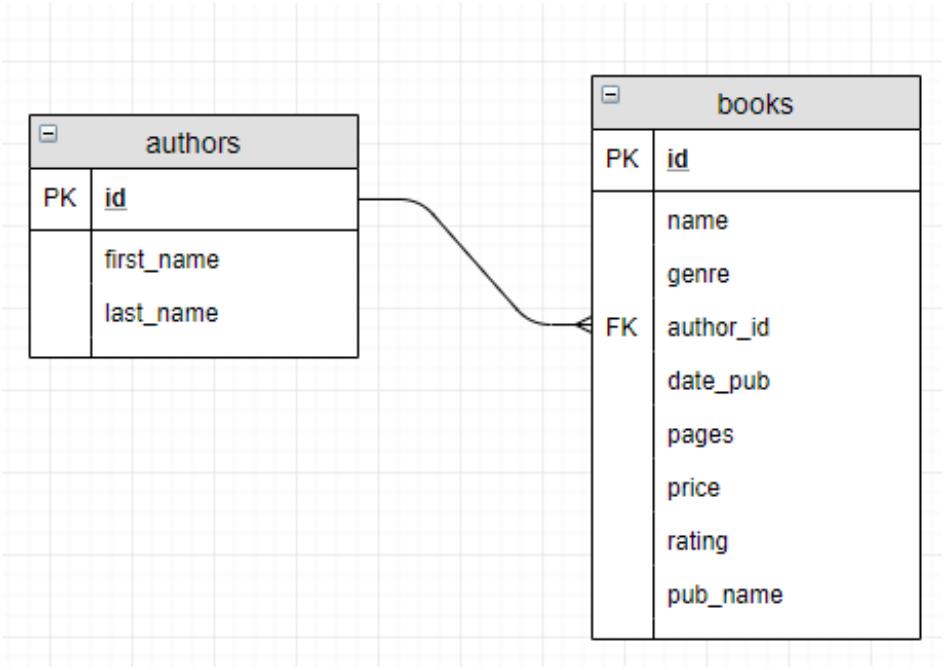

Рис. 9. Связь «один ко многим»

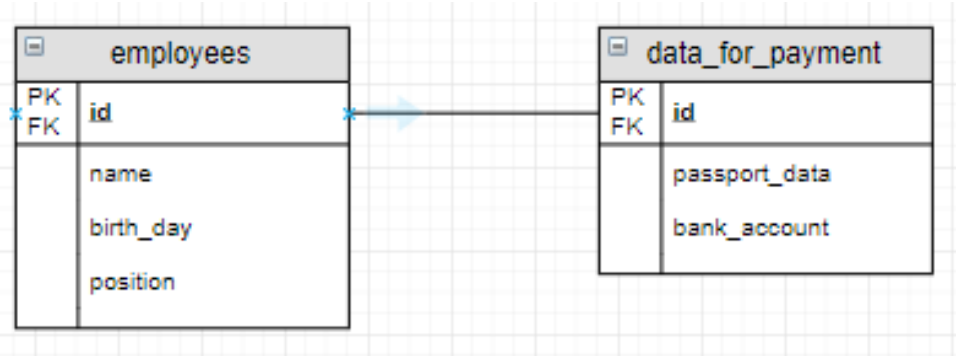

Рис. 10. Связь «один к одному»

Иногда вместо подписей ставят пиктограмму ключа, символ решетки # или другой – по договоренности. Связи тоже отображают на *ER*-диаграммах. Конец связующей линии показывает, одно или несколько значений одной таблицы соответствует значениям другой таблицы.

Продуктивную или основную базу данных компании одновременно используют разные сотрудники. Каждому нужна только часть данных для решения собственной задачи. Чтобы многочисленные пользователи не мешали друг другу, их совместную работу с базой данных нужно организовать. Этим занимаются администраторы баз данных. Такие специалисты управляют доступом пользователей, отслеживают нагрузку, обеспечивают безопасность

и создают резервные копии. База данных – живой организм. Она постоянно растет. «Здоровое» развитие базы обеспечивают архитекторы и разработчики баз данных. От результатов работы этих специалистов зависит структура данных, их целостность и полнота, а также возможность ее масштабировать: добавлять новые таблицы, связи и функции. Архитекторы и разработчики отвечают и за производительность базы данных. Данные в базу добавляет дата-инженер. Это специалист по *ETL*-процессам: выгрузке, трансформации и пополнению данных базы. Аналитик в этой схеме – простой пользователь. Он пишет запросы к базе и получает нужные данные, которые затем анализирует и применяет для проверки гипотез. Однако аналитик теснее всех работает с данными, а значит, первым сталкивается с нехваткой полей или таблиц.

В *SQL* пустые ячейки называют *NULL*. Их ищут конструкцией *IS NULL*:

```
SELECT
     *
FROM
     название_таблицы
WHERE
     название_столбца IS NULL;
--IS имеет значение. Такая проверка на NULL не пройдет:
SELECT
     *
FROM
     название_таблицы
WHERE
     название_столбца = NULL; -- этот код не сработает!
--Чтобы исключить из рассмотрения строки с NULL, применяют оператор NOT:
SELECT
     *
FROM
     название_таблицы
WHERE
     название_столбца IS NOT NULL;
```

Чтобы произвести действия в зависимости от некоторых условий, применяют конструкцию *CASE*. Она похожа на *if-elif-else* в *Python*:

```
CASE
      WHEN условие_1 THEN результат_1
      WHEN условие_2 THEN результат_2
      WHEN условие_3 THEN результат_3
      ELSE результат_4
END;
```

После оператора *WHEN* идет условие. Если строка таблицы подходит условию, то возвращается результат, указанный в *THEN* (англ. «тогда»). Если нет, строка проходит проверку по следующему условию. В конце концов, если строка не соответствует ни одному из условий, описанных в *WHEN*, возвращается значение после *ELSE,* и конструкция *CASE* закрывается оператором *END*.

Оператор *LIKE* находит схожие значения в таблице. Искать можно не только целое слово, но и его часть. Синтаксис *LIKE*:

```
название_столбца LIKE 'регулярное выражение'
```

Перед оператором *LIKE* указывают столбец, в котором нужно искать, а после *LIKE* – регулярное выражение. Регулярные выражения в *SQL* несколько отличаются от регулярных выражений в *Python*. Например, знак нижнего подчеркивания _ в регулярном выражении заменяет одно подстановочное значение, т.е. 1 символ. Знак процента % заменяет любое количество символов. Внутри квадратных скобок [] указывается диапазон или последовательность символов, которые должны содержаться в строке. Если же необходимо, чтобы символы из диапазона или последовательности отсутствовали, используется конструкция [^].

Диапазон или последовательность символов. Если нужно найти символ из регулярного выражения как подстроку, используется оператор *ESCAPE*. Ему передают символ, например, восклицательный знак. В регулярном выра-

жении этот восклицательный знак сообщает: «Символ, который идет после меня не относится к регулярному выражению. Это подстрока, которую нужно найти». Вот фрагмент кода, который найдет все строки, заканчивающиеся на знак процента % в таблице:

```
название_столбца LIKE '%!%' ESCAPE '!'
--найдет все строки заканчивающиеся на %
```

Все данные редко хранят в одной таблице. Обычно таблицы приходится соединять. Для этого есть оператор *JOIN*. «Джойнят» таблицы двумя способами: *INNER* и *OUTER*. Соединение *INNER* выдает строки строго на пересечении двух таблиц. *OUTER* получает полные данные первой таблицы и к ним добавляет данные на пересечении со второй таблицей:

INNER JOIN формирует выборку только из тех данных, условие для присоединения которых выполняется. От порядка присоединения таблиц результат не изменится. Пример формата запроса с *INNER JOIN*:

```
SELECT
--перечисляем только те поля, которые нужны
     TABLE_1.поле_1 AS поле_1,
     TABLE_1.поле_2 AS поле_2,
     ...
     TABLE_2.поле_n AS поле_n
FROM
     TABLE_1
INNER JOIN TABLE_2 ON TABLE_2.поле_1 = TABLE_1.поле_2
```

Разберем синтаксис подробнее:

- *INNER JOIN* – название способа соединения, после него указывают имя таблицы, с которой нужно соединить таблицу из блока *FROM*;
- *ON* открывает условие для присоединения: *TABLE*_2.поле_1 = = *TABLE*_1.поле_2. Следовательно, соединяют только те строки таблиц, которые соответствуют условию. В нашем случае, равенству значений полей.

Так как поля в разных таблицах могут быть названы одинаково, к ним обращаются с указанием имени таблицы. Сначала идет название таблицы, а потом имя ее поля: *TABLE*_1.поле_1.

OUTER JOIN, или внешнее соединение таблиц выполняют двумя способами:

- *LEFT OUTER JOIN*
- *RIGHT OUTER JOIN*

Способы объединения называются кратко: *LEFT JOIN* и *RIGHT JOIN*. *LEFT JOIN* возьмет все данные левой таблицы и строки на пересечении левой и правой, которые удовлетворяют условию присоединения. *RIGHT JOIN*, наоборот, возьмет всю правую таблицу, и строки на пересечении с левой, соответствующие условию.

Пример формата запроса с *LEFT JOIN*:

```
SELECT
      TABLE_1.поле_1 AS поле_1,
      TABLE_1.поле_2 AS поле_2,
      ...
      TABLE_2.поле_n AS поле_n
FROM
      TABLE_1
LEFT JOIN TABLE_2 ON TABLE_2.поле = TABLE_1.поле
```

Как и в запросах с *INNER JOIN*, название таблицы указывают для каждого поля. Обратите внимание, что в отличие от *INNER JOIN* перестановка таблиц меняет результат.

RIGHT JOIN – брат-близнец *LEFT JOIN*. Вот только в отличие от него, в результат своей работы берет всю правую таблицу, и строки на пересечении с левой, если они соответствуют условию.

Пример формата запроса с *RIGHT JOIN*:

```
SELECT
     TABLE_1.поле_1 AS поле_1,
     TABLE_1.поле_2 AS поле_2,
     ...
     TABLE_2.поле_n AS поле_n
FROM
     TABLE_1 RIGHT JOIN TABLE_2 ON TABLE_1.поле = TABLE_2.поле
```

Пример формата запроса с несколькими INNER JOIN:

```
SELECT --перечисляем только те поля, которые нужны
     TABLE_1.поле_1 AS поле_1,
     TABLE_1.поле_2 AS поле_2,
     ...
     TABLE_3.поле_n AS поле_n
FROM
     TABLE_1
INNER JOIN TABLE_2 ON TABLE_2.поле = TABLE_1.поле
INNER JOIN TABLE_3 ON TABLE_3.поле = TABLE_1.поле
```

К первой таблице присоединяем вторую, а потом третью.

Операторы *UNION* и *UNION ALL* «склеивают» данные из таблиц. Синтаксис операторов выглядит так:

```
SELECT
     название_столбца_1
FROM
     таблица_1
UNION --( или UNION ALL)
SELECT
     название_столбца_1
FROM
     таблица_2
```

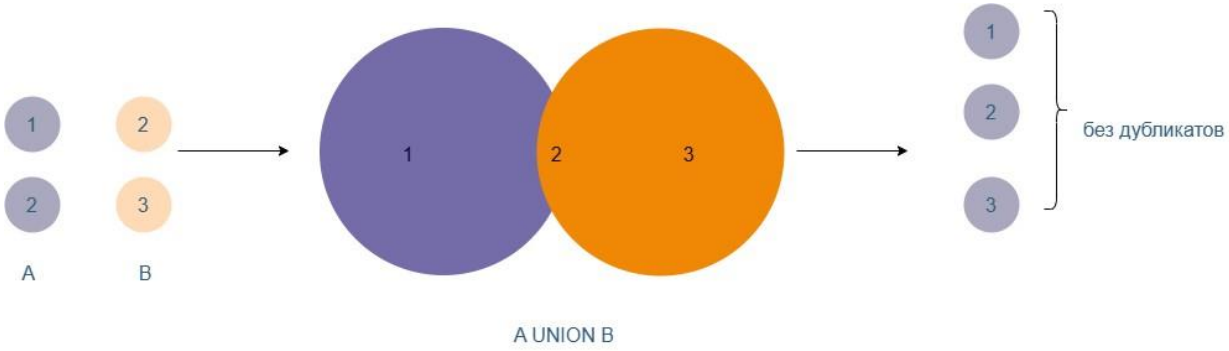

Рис. 11. Оператор *UNION*

Два запроса *SELECT-FROM* разделяет команда *UNION*. Условия, которые нужно соблюдать при «склеивании» данных: данные из первой таблицы должны совпадать с данными из второй таблицы по количеству выбранных столбцов и типу данных; поля первой таблицы нужно указывать в том же порядке, что и поля второй таблицы. *UNION* «склеивает» данные так, что дублирующиеся строки не попадают в результат.

ЗАКЛЮЧЕНИЕ

В XXI веке бизнесу и государству особенно важно уметь прогнозировать и предсказывать различные сценарии развития событий, которые повлияют на них. В экономике и других сферах одновременно взаимодействуют множество факторов – от точности прогнозов зависит благосостояние и выживание организаций и стран. Ключевой элемент точных прогнозов – грамотная работа с данными, источники и объем которых значительно увеличиваются ежедневно. Для это необходимы профессионалы, которые умеют исследовать проблему, формулировать и проверять с помощью инструментов анализа данных гипотезы, а также с помощью алгоритмов и моделей машинного обучения предсказывать следующий виток в развитии того или иного явления: будь это рыночный спрос или поворот самоуправляемой машины.

СПИСОК ИСТОЧНИКОВ

1. **Маккинни, У.** Python и анализ данных / У. Маккинни ; пер. с англ. А. А. Слинкина. – 2ое изд., испр. и доп. – М. : ДМК Пресс, 2020. – 540 с.

2. **Макшанов, А. В.** Большие данные. Big Data / А. В. Макшанов, А. Е. Журавлев, Л. Н. Тындыкарь. – 3-е изд., стер. – СПб. : Лань, 2023. – 188 с.

3. **Целых, А. Н.** Принятие решений на основе методов машинного обучения : учебное пособие / А. Н. Целых, Н. В. Драгныш, Э. М. Котов. – Ростов н/Д : ЮФУ, 2022. – 113 с.

4. **Курс** «Аналитик данных» [Электронный ресурс]. – URL : https://practicum.yandex.ru/data-analyst/

5. **Осипов, Д. Л.** Технологии проектирования баз данных / Д. Л. Осипов. – М. : ДМК Пресс, 2019. – 498 с.

6. **Юре,** Л. Анализ больших наборов данных / Л. Юре, Р. Ананд, Д. У. Джеффри ; пер. с англ. А. А. Слинкин. – М. : ДМК Пресс, 2016. – 498 с.

ГЛОССАРИЙ

1. CSV – формат файла (от англ. Comma-Separated Values, «значения, разделенные запятой»). Каждая строка представляет собой одну строку таблицы, где данные разделены запятыми. В первой строке собраны заголовки столбцов (если они есть).

2. GIGO – (от англ. garbage in – garbage out, буквально «мусор на входе – мусор на выходе») принцип, утверждающий, что при неверных входных данных даже правильный алгоритм анализа выдает неверные результаты.

3. p-value – уровень значимости, на котором находятся друг от друга среднее значение и то, с которым проводится сравнение.

4. User ID – уникальный номер, присвоенный посетителю, чтобы отличить его от остальных.

5. Алгебраическая метрика локации – среднее значение.

6. Альтернативная гипотеза – гипотеза, противоположная по смыслу нулевой гипотезе.

7. Баг-репорт (сообщение об ошибке) – сообщение, содержащее полную информацию об ошибке в программе, на сайте или в системе: суть ошибки и где, когда, при каких условиях она была обнаружена.

8. Библиотека – набор готовых методов для решения распространенных задач.

9. Биномиальные эксперименты – эксперименты с двумя возможными исходами, называемыми успехом и неудачей.

10. Вероятностное пространство – множество всех элементарных исходов эксперимента.

11. Взаимоисключающие события – события, которые не могут произойти одновременно при проведении эксперимента.

12. Второй квартиль Q2 (медиана) – половина элементов больше и половина меньше нее.

13. Выборка – часть генеральной совокупности.

14. Выборочная дисперсия – дисперсия, посчитанная на выборке.

15. Выборочное среднее – среднее значение, посчитанное на выборке.

16. Генеральная совокупность – большой объем данных для статистических исследований.

17. Гистограмма – это график, который показывает, как часто в наборе данных встречается то или иное значение.

18. Датасет с положительной скошенностью (скошенностью вправо) – датасет, гистограмма значений которого ассиметрична: дополнительные значения находятся справа, со стороны положительного направления оси.

19. Датасет с отрицательной скошенностью (скошенностью влево) – датасет, гистограмма значений которого ассиметрична: дополнительные значения находятся слева, со стороны отрицательного направления оси.

20. Диаграмма рассеяния (точечная диаграмма) – график, который каждый объект отображает как точку в заданных координатах.

21. Дискретная переменная – количественная переменная, которая может принимать строго определенные значения.

22. Дисперсия – среднее значение квадрата отклонения случайной величины от ее среднего значения. Мера разброса значений случайной величины.

23. Достоверное событие – событие, которое точно произойдет, вероятность его которого равна единице.

24. Закон больших чисел – чем больше раз повторяется эксперимент, тем ближе частота заданного на этом эксперименте события будет к его вероятности.

25. Категориальная (качественная) переменная – принимает значение из ограниченного набора.

26. Квартиль – число, разбивающее выборку на две части: значения в одной части меньше квартиля, а в другой – больше.

27. Количественная (численная) переменная – принимает числовое значение в диапазоне.

28. Коэффициент корреляции Пирсона – число от –1 до 1, которое показывает, как сильно будет изменяться одна величина при изменении другой.

29. Математическое ожидание случайной величины – численное значение, к которому случайная величина будет в среднем стремиться при многократном повторе эксперимента.

30. Межквартильный размах – это расстояние между первым Q1 и третьим Q3 квартилями.

31. Метрики локации данных – характерные значения выборки, по значениям которых можно судить, где примерно расположен набор данных на числовой оси.

32. Невозможное событие – событие, которое не произойдет никогда, вероятность его появления равна нулю.

33. Независимые события – события, наступление одного из них не влияет на вероятность другого. Если события независимы, то вероятность их пересечения равна произведению их вероятностей.

34. Непрерывная переменная – количественная переменная, которая может принимать любое численное значение.

35. Нормальное распределение – распределение, в котором чаще всего встречается среднее значение и близкие к нему, а крайние значения встречаются довольно редко.

36. Нулевая гипотеза – гипотеза, которая проверяется на выборке.

37. Парная выборка – выборка, используемая для измерения некоторой переменной для одних и тех же единиц.

38. Первый квартиль Q1 – 25% элементов меньше, а 75% – больше него.

39. Плотностная гистограмма – гистограмма, использующая в качества переменной столбца плотность частоты.

40. Плотность частоты – величина, равная высоте столбца гистограммы, площадь которого отображает частоту непрерывной переменной.

41. Равновероятные исходы – исходы, вероятность появления которых одинакова.

42. Распределение – это все возможные значения переменной с частотой их появления.

43. Распределение вероятности случайной величины – таблица, содержащая всевозможные значения случайной величины и вероятности их появления.

44. Распределение Пуассона – число событий в единицу времени, если они в среднем происходят с измеренной частотой.

45. Репрезентативная выборка – часть данных, представляющая всю генеральную совокупность.

46. Случайная выборка – часть данных из генеральной совокупности, выбираемая случайным образом.

47. Скошенность – ассиметричность датасета.

48. Случайная величина – это переменная, которая принимает случайные значения – те значения, которые нельзя предсказать до проведения эксперимента.

49. Событие – подмножество вероятностного пространства, содержащее в себе некоторое количество элементарных исходов.

50. Срез данных – часть данных из предоставленного набора, отобранная по определенным условия.

51. Стандартная ошибка – стандартное отклонение выборочного среднего от настоящего среднего генеральной совокупности.

52. Стандартное отклонение – числовая характеристика данных, которая показывает, насколько значения в выборке отличаются от среднего арифметического (квадратный корень из дисперсии).

53. Статистика разности – это количество стандартных отклонений между сравниваемыми значениями, если оба распределения привести к стандартному нормальному распределению со средним 0 и стандартным отклонением 1.

54. Страты – группы в генеральной совокупности, объединенные общим признаком.

55. Стратифицированная выборка – выборка, состоящая из пропорциональных выборок из разных страт.

56. Структурная метрика локации – медиана.

57. Третий квартиль Q3 – 75% элементов меньше и 25% элементов больше него.

58. Уровень значимости – это суммарная вероятность того, что измеренное эмпирически значение окажется далеко от среднего.

59. Характерный разброс – то, какие значения оказались вдали от среднего, и насколько их много.

60. Центральная предельная теорема – много независимых случайных величин, сложенных вместе, дают нормальное распределение.

61. Числовое описание данных – среднее, медиана, стандартное отклонение, количество наблюдений в выборке и разброс их значений.

62. Эксперимент – повторяемый опыт, который может окончиться разными исходами.

63. Элементарные исходы – исходы эксперимента, которые нельзя раздробить на более мелкие подисходы.

64. Ячеечная диаграмма – график, разделенный на ячейки, цвет которых отражает количество попавших в данную ячейку точек в заданных координатах.

ОГЛАВЛЕНИЕ

Учебное электронное издание

КОНКИНА Виктория Викторовна
БОРИСЕНКО Андрей Борисович
КОРОБОВА Ирина Львовна

ВВЕДЕНИЕ В БОЛЬШИЕ ДАННЫЕ И АНАЛИЗ ИНФОРМАЦИИ

Учебное пособие

Редактирование И. В. Калистратовой
Графический и мультимедийный дизайнер Т. Ю. Зотова
Обложка, упаковка, тиражирование И. В. Калистратовой

ISBN 978-5-8265-2749-8

Подписано к использованию 06.03.2024.
Тираж 50 шт. Заказ № 27

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106/5, пом. 2, к. 14
Телефон 8(4752)63-81-08.
E-mail: izdatelstvo@tstu.ru