Математика и Python для анализа данных

Оглавление

Введение **2**
1.1 Введение . . . 2

Математика для анализа данных **3**
2.1 Векторы и матрицы . . . 3
2.2 Определитель матрицы . . . 4
2.3 Операции с матрицами. Собственные числа матриц . . . 5

Библиотеки для работы с данными **7**
3.1 Знакомство с библиотекой NumPy . . . 7
3.2 Знакомство с библиотекой SciPy . . . 10

Подготовка данных **13**
4.1 Знакомство с библиотекой pandas . . . 13
4.2 Объект pandas.Series . . . 13
4.3 Объект pandas.DataFrame . . . 15
4.4 Группировка данных . . . 17
4.5 Работа с несколькими таблицами . . . 18
4.6 Преобразование признаков . . . 19

Введение

1.1 Введение

Анализ данных, машинное обучение, искусственный интеллект являются одними из самых горячих областей на сегодняшний день, а *Python* является одним из самых популярных языков для того, чтобы применять алгоритмы машинного обучения. С развитием интернета, появлением смартфонов и, в целом, с развитием технологий умных устройств мы с вами генерируем все больше и больше данных каждый день. **DataScience** - это наука о том, как анализировать эти данные, находить в них природную зависимость, делать выводы на основе этих данных.

Тяжело переоценить все возможности анализа данных. В этом курсе вы узнаете, как применять свои навыки программирования на языке *Python* для построения примитивных моделей, визуализации данных и работы с нейросетями. Для успешного освоения материалов курса достаточно уметь писать программы на языке *Python*. Однако, анализ данных - это не только программирование.

- Для понимания методов анализа данных, необходимо знание в математическом анализе, линейной алгебре и математической статистике. Звучит страшно? Не переживайте, если вы не знакомы с этими областями знаний или прошло очень много времени с тех пор как вы последний раз виделись, дополнить и освежить эти знания вы сможете в начале нашего курса.
- Затем мы с вами начнем говорить о, непосредственно, машинном обучении и познакомимся с алгоритмами **обучения с учителем** - когда вы тренируетесь на каких-то известных данных и делаете предсказания на новых данных.
- Далее мы с вами разберем алгоритмы **обучения без учителя** - когда вы пытаетесь найти какую-то природную структурную зависимость ваших данных.
- Потом мы с вами познакомимся с нейронными сетями, поговорим о том, что это такое, как они работают и в каких областях применяются и разберем несколько архитектур современных нейронных сетей.
- А в конце курса конечно же будет курсовой проект. Будьте готовы собрать все полученные знания и навыки чтобы применить их на задаче предсказания, с которой сталкиваются DataScientists в реальной жизни.

По завершению курса вы будете знать основные методы анализа данных и уметь пользоваться ими при помощи инструментов языка *Python*.

Математика для анализа данных

2.1 Векторы и матрицы

Давайте разберем основные объекты линейной алгебры, такие как: **вектор** и **матрица**, а также, что с ними можно делать.

Посмотрим на следующий пример. Представим, что у нас есть абонент сотовой связи, который звонит, пишет сообщение и сидит в интернете. Мы хотим этого абонента описывать какой-то единой структурой, в которой будет лежать наше значение по количеству минут, количеству сообщений и использованом трафике. Этой структурой может быть **вектор**. Например,

$\vec{a} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix}$, $\vec{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\vec{c} = \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix}$ Он представляет из себя совокупность этих элементов.

Теперь представим, что у нас есть не один абонент, а их несколько. Например, все абоненты города Москвы; и для того чтобы их описывать, нам нужна какая-то совокупность векторов.

Такая совокупность векторов будет называться **матрицей** $\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$

Давайте вернемся к нашему абоненту и рассмотрим следующую ситуацию. Представим, что абоненту предлагается некоторая акция, в которой ему предлагают увеличить количество звонков, количество сообщений и количество мегабайт на некоторые числа и мы хотим поэлементно, к уже имеющимся вектору добавить эти новые значения. Тут возникает операция поэлементного **сложения между векторами**.

$$\begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \dots \\ x_n + y_n \end{bmatrix}$$

А теперь представим, что нашему абоненту предлагается другая акция, в которой ему можно увеличить в несколько раз уже имеющееся количество минут, звонков и мегабайт. И здесь мы можем определить операцию поэлементного **умножения вектора на число**.

$$a \cdot \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} a \cdot x_1 \\ \dots \\ a \cdot x_n \end{bmatrix}$$

Все эти определения приводит нас к важному понятию как **векторное пространство**. **Векторное пространство** - это не пустое множество элементов на которых задана операция поэлементного сложения и умножения на число. Важно отметить, что **векторное пространство замкнуто относительно этих операций**. То есть результатом этих операций будет являться вектор лежащий в этом же пространстве.

Давайте рассмотрим следующих абонентов. Представим, что у нас есть абонент, который **только звонит**, который **только пишет сообщение** и абонент, который **пользуется мобильным интернетом**. Вектора этих абонентов такие

$$\vec{e_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \vec{e_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \vec{e_3} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

А теперь представим, что у нас есть очень активный абонент, который пользуется всем. Много звонит, пишет и сидит в интернете. Его вектор мы можем представить, как сумму этих трех векторов с некоторыми коэффициентами.

$$\begin{bmatrix} 15 \\ 3 \\ 100 \end{bmatrix} = 15 \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 3 \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 100 \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Можно заметить, что эти три вектора никаким образом нельзя выразить друг через друга, и они являются **линейно независимыми**. Но с помощью них можно описать любой другой вектор нашего пространства и так мы приходим к определению линейной зависимости. Вектор **линейно зависим**, если его можно выразить через другие вектора.

Как мы уже отметили эти три вектора позволяют описать любой другой вектор нашего пространства, и они образуют **базис** нашего векторного пространства. **Размерность векторного пространства** определяется максимальным числом линейно независимых векторов.

2.2 Определитель матрицы

Поговорим про то, как можно проверить систему векторов на линейную независимость. Один из подходов заключается в том, что мы составляем систему линейных уравнений и пытаемся ее решить, а другой заключается в том, что мы считаем величину, которая нам сразу говорит, является ли система векторов линейно независимой или нет.

Давайте рассмотрим двумерный случай и рассмотрим два вектора. Давайте составим матрицу из этих векторов. Существует такая величина, называемая **определителем** или **детерминантом** матрицы, который **равняется нулю в случае линейной зависимости векторов и не равен нулю в случае линейной независимости**.

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

Важно отметить, что **эта величина считается для квадратных матриц**.

Давайте теперь посмотрим, какой геометрический смысл в себе несет детерминант. Давайте рассмотрим наши вектора на плоскости. Если вектора линейно независимы, то они образуют параллелограмм ненулевой площади и **площадь этого параллелограмма как раз равняется значению определителя**. Если же вектора линейно зависимы, то площадь параллелограмма будет равняться нулю. Если мы рассматриваем систему векторов размерности больше, чем два, то для вычисления детерминанта используется рекуррентная формула.

2.3 Операции с матрицами. Собственные числа матриц

Поговорим про матричные операции, типы матриц, познакомимся с линейным преобразованием, а также рассмотрим такое понятие, как собственные вектора и числа.

Давайте начнем с простого. Для матриц с одинаковым размером определена операция **сложения**

$$\begin{bmatrix} 5 & 6 \\ -1 & 0 \\ 2 & -3 \end{bmatrix} + \begin{bmatrix} -2 & 3 \\ 1 & -1 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 5+(-2) & 6+3 \\ -1+1 & 0+(-1) \\ 2+0 & -3+3 \end{bmatrix}$$

И также для матрицы можно определить операцию **умножения на число**

$$2 \cdot \begin{bmatrix} 5 & 6 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 2 \cdot 5 & 2 \cdot 6 \\ 2 \cdot (-1) & 2 \cdot 0 \end{bmatrix}$$

У матрицы можно выделить **главную диагональ** - это те элементы, у которых индекс по срокам равен индексу по столбцам.

И для матриц вводится операция **транспонирование**, то есть поворот матрицы относительно главной диагонали. В этом случае столбцы становятся на место строк.

$$\begin{bmatrix} 5 & 6 \\ -1 & 0 \\ 2 & -3 \end{bmatrix}^T = \begin{bmatrix} 5 & -1 & 2 \\ 6 & 0 & 3 \end{bmatrix}$$

Также у матрицы можно найти обратную матрицу. **Обратная матрица** - это такая матрица, которая при умножении на исходную матрицу дает единичную матрицу.

$$AA^{-1} = A^{-1}A = I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Операция обращение матрицы определена только для квадратных матриц.

Давайте теперь рассмотрим, как вводится **перемножение матриц**. Давайте рассмотрим две матрицы. Прежде, чем проводить операцию перемножения, нам необходимо проверить размерность матриц. Нам нужно, чтобы **число столбцов в первой матрице равнялось числу строк второй матрицы**. То есть, если первая матрица у нас размера $m \times n$, то вторая матрица должна быть размера $n \times k$, и тогда размер финальной матрицы будет $m \times k$.

Как выполняется переммножение? Для того, чтобы найти значение находящееся в **первой** строке **первого** столбца, нам нужно взять **первую** строчку из первой матрицы, взять **первый** столбец из второй матрицы, умножить поэлементно и затем сложить. Так мы получим нужное значение. Для остальных элементов аналогично.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 2 \cdot 3 & 1 \cdot 2 + 2 \cdot 4 \\ 3 \cdot 1 + 4 \cdot 3 & 3 \cdot 2 + 3 \cdot 4 \\ 5 \cdot 1 + 6 \cdot 3 & 5 \cdot 2 + 6 \cdot 4 \end{bmatrix}$$

Почему именно так задаётся перемножение матриц? Вообще говоря, **матрица задает некоторое линейное преобразование пространства**, то есть переход из одного пространства в другое.

Давайте теперь рассмотрим какие бывают типы матриц. Например, бывают

- **диагональные матрицы** - это такие матрицы, у которых значение вне главной диагонали равны нулю. Частным случаем являются единичная матрица
- **симметричные матрицы** - это такие матрицы, которые при транспонировании не меняются, то есть транспонированная матрица равна исходной. $A = A^T$
- **ортогональные матрицы** - это такие матрицы, у которых транспонированная матрица является обратной. $AA^T = A^T A = I$

Как мы уже отметили, матрица задает некоторое линейное преобразование. Давайте посмотрим на диагональную матрицу. **Диагональная матрица может задавать преобразование, такое, как растяжение и поворот**. Значения на главной диагонали будут показывать нам во сколько раз изменяется значения по осям, а также если значение отрицательно, они будут еще указывать на то, что был поворот или нет.

Стоит отметить, что при линейном преобразовании вектора могут вести себя по разному. Например, некоторые вектора могут менять свое направление а некоторые - нет, и вектора, которые не меняют своего направления при линейном преобразовании, называются **собственными**.

Как мы можем найти собственные вектора? Для этого нам необходимо решить уравнение $A\vec{x} = \lambda\vec{x}, \vec{x} \neq \vec{0}$, где λ - это **собственное значение**.

У матрицы размера n может быть **не больше** n собственных значений. Собственные вектора очень важны при задачах, где мы хотим уменьшить размерность пространства, но при этом сохранить максимум информации.

Библиотеки для работы с данными

3.1 Знакомство с библиотекой NumPy

Давайте разберёмся с библиотекой *NumPy*, как с ней работать и для чего она нужна в анализе данных. Напомним, что библиотека *NumPy* - это библиотека для различного рода математических вычислений и работой с многомерными массивами. Она довольно мощная, она обладает большим математическим потенциалом в плане того, что очень много функций там есть, и у нее очень гибкий и понятный интерфейс для того, чтобы начать с этим работать. Для начала давайте разберемся какова основная структура библиотеки *NumPy*. Основная структура - это **некоторый многомерный массив**, который создается на основе наших данных. Давайте попробуем сначала его создать и затем рассмотрим детально какими свойствами обладает многомерный массив в *NumPy*.

Для начала импортируем библиотеку. Для создания некоторого массива нам необходимо вызвать функцию *array*, которая принимает на вход как раз наши данные в виде некоторой последовательности.

```
import numpy as np
x = np.array([1,2,3,4])
```

С помощью атрибута *dtype* можно узнать какие данные хранятся в нашем массиве.

```
x.dtype
```

Тип данных может определяться автоматически на основе того, какие есть входные данные, а также можно его указывать при создании массива как дополнительный аргумент.

```
x = np.array([1,2,3,4], dtype=np.int64)
```

В данном случае мы создали некоторый одномерный массив и для того, чтобы посмотреть какая размерность, мы воспользуемся атрибутом *shape*.

```
x.shape
```

Атрибут *shape* возвращает набор размерности по всем осям, которые у нас есть в массиве, то есть если бы это был какой-то двумерный массив, у нас бы вернулась пара число строк и число столбцов, но в данном случае у нас получается одномерный массив и возвращается только одно значение.

Давайте рассмотрим, как создать двумерный массив либо матрицу, и какими свойствами она будет обладать.

```
m = np.array([[2,3,4],[5,6,7]])
```

NumPy массивы можно создавать несколькими способами. Мы рассмотрели как можно создать массив на основе *list*, также можно попробовать создать массивы на основе встроенной функции.

Допустим, если мы хотим получить массив заполненный только единицами, для этого можно воспользоваться функцией *ones*, которая в качестве аргумента принимает размер нашего массива, и в результате мы получаем уже массив, заполненный только единицами.

```
m = np.ones(5)
```

Таким же способом можно создать массив заполненный только нулями

```
m = np.zeros(2)
```

Либо же можно создать также массив заполненный случайными числами, либо какими-то константными значениями.

Давайте рассмотрим случай, когда нам необходимо создать некоторую единичную матрицу, где по диагонали у нас будет лежать единицы, а остальные значения будут нулевыми.

```
m = np.eye(6)
```

Для создания массива заполненной случайными значениями можно воспользоваться методом *random*

```
m = np.random.random((2,3))
```

После того, как мы создали массив, мы хотим как-то с ним взаимодействовать и посмотреть какие-то определенные значения, либо значения удовлетворяющие некоторому условию, и для такого рода работы нам нужно каким-то образом проитерироваться по массиву, как-то обратиться по индексу.

Как устроена индексация в *NumPy*? Если у нас какой-то одномерный массив, мы можем просто выбрав нужный нам индекс, вытащить значение, и соответственно, индексация начинается с нуля.

```
x[0]
```

Давайте рассмотрим двумерную матрицу и посмотрим как в данном случае у нас выглядит индексация. Если мы хотим вытащить какой-то элемент, находящийся в, допустим, первой строке и в третьем столбце, нам необходимо задать индексы сразу же по двум осям, то есть сначала мы первым аргументом в квадратных скобках указываем индекс по оси X, а как второй аргумент указываем индекс по оси Y

```
m[1,3]
```

Можно указывать также некоторый промежуток значения индекса, например, следующим образом: сейчас мы выбираем все значения по срокам и берем только все значения в первых трёх столбцах.

```
m[:,:3]
```

Также можно, используя некоторые условия, вытаскивать значения, которые удовлетворяют заданному порогу. Допустим у нас есть некоторый массив, мы хотим найти все значения, превышающие какого-либо заданного числа. Как происходит в данном случае индексация?

Для начала нам необходимо задать наши условия, в результате этого у нас возвращается некоторая маска, маска - это массив того же размера, что наш исходный, но в качестве значений указывается *true* или *false*, что означает, что данное число в массиве превышает, либо не превышает заданного порога, и затем, для того чтобы вытащить нужное нам значение, мы эту маску передаем как индекс в наш массив, и у нас, в результате этого, возвращается уже массив значений, удовлетворяющих условию.

```
x[x > 2]
```

При работе с многомерными массивами, у нас часто возникает ситуация, что нам необходимо как-то поменять форму, либо размерность данных, например, если у нас была какая-то многомерная структура, мы хотим ее развернуть в один массив, одномерный, и посмотреть какие-то значения. В *NumPy* это можно осуществить, используя следующие методы: *flatten* и *reshape*. Чтобы развернуть все значения, хранящиеся в нашей матрице, в одномерный массив, нужно написать

```
x = np.array([[1,2,3],[6,5,4]])
x.flatten()
```

Используя метод *reshape*, мы можем получить некоторый массив из этих элементов других размеров. Для этого в методе *reshape* необходимо передать новые значения по двум осям X и Y, которые мы хотим увидеть.

```
x.reshape((6, 1))
```

Важно отметить, что **размер новой формы не должен по количеству элементов превышать размер нашей исходной матрицы**, то есть если у нас в матрице было всего шесть элементов, мы никак не можем задать форму, в которой будет, например, 15 элементов, мы получим сразу же ошибку.

Также используя метод *resize*, можно поменять форму нашего исходного массива. Отличие *resize* от *reshape* заключается в том, что *resize* автоматически меняет исходный массив, в то время как *reshape* просто изменяет его форму.

```
x.resize((6, 1))
```

После того, как мы научились создавать структуры в *NumPy*, мы можем уже производить над ними некоторые математические операции, например, можно складывать вектора друг с другом, умножать вектора на матрицы и так далее. Давайте рассмотрим на примере двух векторов, какие операции доступны в библиотеке.

```
v = np.array([9,10])
w = np.array([11,12])
```

В *NumPy* сложение векторов можно осуществить двумя способами: используя встроенный метод в библиотеке, либо используя оператор +.

```
res = v + w
res = np.add(v, w)
```

Стоит отметить, что данный вид операции подразумевает под собой **поэлементное** сложение, то есть мы складываем каждый элементы друг с другом.
Таким же способом мы можем посмотреть разность двух векторов

```
res = v - w
res = np.subtract(v, w)
```

или умножение векторов

```
res = v * w
res = np.multiply(v, w)
```

Стоит отметить, что для того чтобы посчитать какое-либо скалярное произведение между векторами, либо умножить вектор на матрицу, нам необходимо воспользоваться другим методом. Он называется *dot* и на вход принимает два вектора, которые мы хотим перемножить. Допустим, давайте посмотрим скалярное произведение двух векторов, результатом которого уже будет не некоторый массив, а какое-то конкретное число.

```
res = np.dot(v, w)
```

Таким же способом, используя функцию *dot*, мы можем перемножать вектора с матрицами, либо делать ещё какие-то перемножения между ними.

3.2 Знакомство с библиотекой SciPy

Библиотека *SciPy* - очень удобная библиотека для различного рода вычислений. Она включает в себя методы оптимизации, методы линейной алгебры, обработки сигналов и изображений. Мы остановимся на модуле с линейной алгеброй и рассмотрим, как можно находить обратную матрицу, детерминант, собственные числа. А также посмотрим, как можно находить минимум функции. Давайте приступим к модулю с линейной алгеброй.

Для начала нам необходимо импортировать соответствующие модули из *SciPy*.

```
from scipy import linalg
from scipy import optimize
import numpy as np
import matplotlib.pyplot as plt
```

Давайте теперь рассмотрим следующую матрицу и попробуем найти детерминант этой матрицы.

```
A = np.array([[1,3,5],[2,5,1],[2,3,8]])
```

В библиотеке *SciPy* есть соответствующий метод *det*, который на вход принимает матрицу. И результатом этого метода будет число

```
linalg.det(A)
```

Важно отметить, что **детерминант вычисляется только для квадратных матриц**, поэтому если мы попробуем передать неквадратную матрицу методу *det*, то получим ошибку.

В данном случае мы видим, что детерминант нашей матрицы не равен 0. Соответственно, мы можем попробовать найти обратную к ней матрицу. Для этого используется метод *inv*.

```
linalg.inv(A)
```

Давайте теперь рассмотрим, как мы можем найти собственные числа и вектора. Для этого в библиотеке *SciPy* уже есть реализованный метод, который возвращает сразу же пару, где первым значением будут собственные числа, а вторым — собственные вектора.

```
eigenvalues, eigenvectors = linalg.eig(A)
```

В итоге возвращается массив, в котором лежат собственные числа, а также возвращается матрица, в которой лежат собственные вектора.

Давайте теперь перейдём к оптимизации и рассмотрим следующую функцию.

```
def f(x):
    return x**2 + 6*np.sin(x)
```

Если мы посмотрим на график этой функции, то мы увидим, что у неё есть локальный минимум и глобальный.

```
x = np.arange(-10, 10, 0.1)
plt.plot(x, f(x))
plt.show()
```

Глобальный минимум находится в районе -1. Давайте попробуем его вычислить. Для вычисления минимума функции в *SciPy* есть соответствующий метод *minimize*. Давайте его вызовем и передадим в качестве аргумента нашу функцию, а также начальное приближение, то есть ту область, с которой мы начнем поиск минимума. Давайте зададим его равным 0.

```
optimize.minimize(f, x_initial=0)
```

Результатом работы функции будет некоторый объект, в котором присутствует множество параметров. Многие из этих параметров относятся к выбранному методу оптимизации, а последний будет соответствовать минимуму функции. Видим, что метод вернул в качестве минимума значение -1, что соответствует глобальному минимуму.

Важно отметить, что метод *minimize* обладает бОльшим числом аргументов, чем мы сейчас указали. Можно также указывать метод оптимизации в зависимости от того, какая у нас функция, а также точность приближения, с которой мы хотим найти минимум.

Давайте ещё раз глянем на график и посмотрим, в каких случаях метод *minimize* может ошибаться. Так как у нас есть локальный минимум, то случайно вместо глобального минимума метод может вернуть локальный минимум, и этого легко добиться, изменив только один параметр, например, начальное приближение. Давайте зададим значение, неравное 0, например, возьмём значение, равное 3. И мы видим, что результат, который соответствует минимуму, в данном случае близится к значению 3, что похоже на один из локальных минимумов функции.

На самом деле, библиотека *SciPy* может намного больше. Со всеми её возможностями вы можете ознакомиться в документации. В ней можно найти множество хороших примеров и подробное описание всех реализованных методов.

Подготовка данных

4.1 Знакомство с библиотекой pandas

pandas - является одной из ключевых библиотек и позволяет произвести полноценный процесс анализа данных. Загрузка данных и их обработка, визуализация и представление финального результата. Немаловажно, что библиотека регулярно обновляется и расширяет свой функционал. Давайте разберем каждый из этих шагов.

- Начнем с загрузки данных. *pandas* **позволяет загружать данные различных форматов**. Библиотека поддерживает загрузку текстовых файлов, бинарных, а также можно подключаться к базам данных и работать с ними напрямую.
- В каком виде *pandas* представляет данные? *pandas* **оперирует двумя основными структурами данных**: *Series* и *DataFrame*. *Series* - индексированный массив некоторого типа: числовой, бинарный или категориальный. *DataFrame* - двумерная структура данных, совокупность *Series* - другими словами, это таблица.
- После загрузки и формирования *Series* или *DataFrame* можно проводить обработку данных, например, можно **фильтровать, индексировать, объединять** различные *DataFrame* друг с другом, можно **писать свои вычислительные методы** и применять их на *DataFrame*.
- Помимо обработки и загрузки данных с помощью библиотеки можно **строить графики**, например, по выбранному столбцу из *DataFrame* можно построить гистограмму значений, можно посмотреть как значение в этом столбце изменяется во времени. Визуализация возможна благодаря интеграции *pandas* с другой библиотекой - *Matplotlib*. Вообще говоря *pandas* **интегрируется и с другими библиотеками**, например, с *NumPy* или *SciPy*, - это расширяет возможности библиотеки.

4.2 Объект pandas.Series

Структура *Series* представляет из себя объект, похожий на одномерный массив, но отличительной его чертой является наличие меток, то есть индексов вдоль каждого элемента из списка. Давайте для начала импортируем *pandas* и создадим *Series* объект.

```
import pandas as pd
s = pd.Series([1,2,3,4], index=['a', 'b', 'c', 'd'])
```

Для создания *Series* необязательно указывать индексы, *pandas* автоматически проставит числовой индекс. Также *Series* **можно создавать, передавая не только лист с данными, но и словарь**, ключи которого станут индексами в созданном *Series*.

```
d = {'Moscow': 1000, 'London': 300, 'Barcelona': None}
cities = pd.Series(d)
```

Вытащить значение в *Series* можно по индексу

```
cities['Moscow']
```

Если же мы хотим как-то **отфильтровать или выбрать строки по некоторым индексам**, то нам достаточно передать список этих индексов, и нам вернутся соответствующие значения.

```
cities['Moscow','London']
```

Представим ситуацию, что нам надо из *Series* **вытащить некоторые строки, которые удовлетворяют некоторым условиям**, например, значения не превышают заданного предела. Для этого для начала нам необходимо создать маску, то есть написать то условие, которому должна соответствовать *Series*, а потом передать его в качестве индекса.

```
cities[cities < 1000]
```

Как мы можем **поменять значение** в *Series*? Для этого достаточно выбрать значение в нужном нам индексе и присвоить ему новое.

```
cities['Moscow'] = 100
```

Если же мы хотим поменять значения в *Series*, которые мы отфильтровали, например, выбрали по заданной маске, то мы можем сделать это аналогичным способом.

```
cities[cities < 1000] = 3
```

Если в *Series* хранятся значения числового типа, то можно простым способом изменить значения всего *Series*, например, применить к значениям арифметические операции.

```
cities*3
```

Также, как вы уже успели заметить, в нашем *Series* присутствуют значения *NaN*, которые соответствуют тому, что по индексу «Барселона» у нас нет значения. Как мы можем отфильтровать такие строки?

В *pandas* реализовано два метода — *isnull* и *notnull*. Они противоположны друг другу.

```
cities[cities.isnull()]
```

Метод *isnull* вернул нам только одну строчку, а если мы попробуем вызвать метод *notnull*, то нам вернутся все остальные строки, в которых есть значения.

4.3 Объект pandas.DataFrame

Давайте импортируем нужные нам библиотеки.

```
import pandas as pd
import numpy as np
```

Как работать с *DataFrame*, мы рассмотрим на примере датасета по поездкам велобайка в Нью-Йорке.

```
df = pd.read_csv('citibike.csv')
```

Давайте посмотрим, как устроен сам *DataFrame*. Нам зачастую не надо распечатывать весь *DataFrame*, когда мы хотим узнать его структуру, какие там содержатся столбцы. Допустим, нам интересно **посмотреть только на первые 3 строки**, и в *pandas* есть метод *head*, который по умолчанию возвращает первые 5 строк нашего *DataFrame*.

```
df.head(3)
```

Если нам нужно посмотреть не первые 3 строки *DataFrame*, а последние, то можно воспользоваться методом *tail*, который, аналогично, возвращает последние 5 строк по умолчанию.

```
df.tail(3)
```

Помимо этого *DataFrame* обладает следующими свойствами: можно **посмотреть его размеры**, сначала указывается число строк, и затем указывается число столбцов

```
df.shape
```

можно **отдельно распечатать** именно название столбцов

```
df.columns
```

и также можно **посмотреть, какие типы данных есть** в нашем *DataFrame*

```
df.dtypes
```

Также можно **вывести** не весь *DataFrame*, а **некоторые столбцы**. Для этого необходимо указать лист нужных нам столбцов и их передать, и мы тогда можем распечатать не весь *DataFrame*, а только выбранные столбцы.

```
df[['start time', 'start station name']].head()
```

Давайте теперь разберем, как **обращаться к элементам** *DataFrame*. В *pandas* реализованы такие методы, как *loc* и *iloc*. *iloc* позволяет по индексу обращаться к строкам и столбцам.

Допустим, если нам необходимо вытащить значения хранящиеся в самой последней строке, мы можем в *iloc* указать индекс -1, и по умолчанию вернется последняя строка.

```
df.iloc[-1]
```

Также с помощью метода *iloc* можно посмотреть какое значение будет лежать в *DataFrame* на пересечении столбца и строки.

```
df.iloc[-1, 4]
```

Метод отличается *loc* от *iloc* тем, что мы можем указывать значения, слайсы по лейблу.

```
df.iloc[1,['tripduration']]
```

Если мы хотим отфильтровать с помощью метода *iloc* указывая границы, то у нас не включаются крайние значения. Метод *loc* работает наоборот, он включает крайние значения. Сравните

```
df.loc[0:6,0:4]
df.iloc[0:6, 'tripduration':'start duration time']
```

Зачастую при работе с *DataFrame* мы сталкиваемся с тем, что нам нужно **выбрать какие-то строки удовлетворяющие некоторым условиям**.

Как это можно сделать? Нам нужно создать маску, которая будет являться нашим условием и передать её в качестве индекса нашему *DataFrame*. Также можно создавать несколько условий на фильтрацию.

```
df[(df['tripduration'] < 1000) & (df['usertype'] == 'Subscriber')]
```

Помимо фильтрации в *pandas* реализованы некоторые методы, которые позволяют посмотреть статистику по всему *DataFrame*. Например, с помощью метода *describe* можно **посмотреть среднее, максимальное, минимальное значение сразу по всем столбцам**.

```
df.describe()
```

Так как наш *DataFrame* хранит не только числовые, но и категориальные признаки, то с помощью метода *describe* по ним тоже можно посмотреть статистику, но нужно указать, какой тип данных мы хотим увидеть.

```
df.describe(include=[np.object])
```

Также, когда мы работаем, например, с категориальными признаками, мы хотим знать **соотношение некоторых значений**. И это можно сделать с помощью метода *value_ counts*.

```
df['usertype'].value_counts(normalize=True)
```

И также можно **посмотреть количество уникальных значений** в некоторых *Series*. Это можно сделать с помощью метода *unigue*.

```
df['gender'].unique()
```

Также можно **найти корреляцию между всеми столбцами датасета**, если они численные, используя метод *corr*.

```
df.corr()
```

Помимо этого, бывает нужно **сделать какой-то sample** из исходного *DataFrame* (если у нас, допустим, очень большой *DataFrame*, и мы хотим сохранить какую-то маленькую часть). Это можно сделать с помощью метода *sample*, который выберет какую-то долю из исходного *DataFrame*.

```
df.sample(frac=0.1)
```

После указанных преобразований мы можем **сохранить** *DataFrame* и использовать метод *to_ csv*. Для этого нам необходимо указать путь к файлу, а также можно указать какие-то другие параметры.

```
df.to_csv('path_to_file.csv')
```

4.4 Группировка данных

Давайте поговорим про то, как можно группировать данные в *pandas*, и какие можно делать манипуляции с получившими группами. Допустим, у нас есть некоторая переменная, она принимает 2 или может быть несколько значений, и мы хотим **выделить все строки, в которых встречается только первое значение, только второе и так далее**. Это можно осуществить с помощью метода *groupby*, затем, после того как у нас выделилась некоторая группа, мы можем уже с ней отдельно работать, смотреть распределение других признаков по данной группе, а также можно сравнивать эти группы между собой. Давайте посмотрим, как это можно сделать в *pandas*.

```
import pandas as pd
import numpy as np

df = pd.read_csv('citibike.csv')
df.groupby(['usertype'])
```

И чтобы **посмотреть, какие получились группы**, можно воспользоваться атрибутом *groups*

```
df.groupby(['usertype']).groups
```

В результате вернётся некоторый словарь, где в качестве ключа лежат значения с группируемой переменной, в данном случае их всего два, а в качестве значения лежат индексы строк, в которых встречается именно это значение.

А затем, мы можем помимо такого представления в виде индексов уже посмотреть, **какие конкретно значения и строки у нас встречаются в группе**. Для этого можно, например, посмотреть первые строки из каждой группы, вызвав метод *first*.

```
df.groupby(['usertype']).first()
```

Здесь уже возвращается некоторый DataFrame со всеми признаками, которые были изначально.

И теперь, когда у нас уже есть некоторая сгруппированная структура, мы можем **посчитать некоторое распределение значений**. Допустим, мы хотим посмотреть среднюю продолжительность поездок по каждой группе пользователей. Как это можно сделать?

Мы также осуществляем группировку данных, а затем указываем лист значений необходимых для агрегирования, и затем метод, который мы хотим, по которому мы собственно говоря агрегируем данные, например, средняя продолжительность поездок.

```
df.groupby(['usertype'])[['tripduration']].mean()
```

У нас возвращается некоторый *DataFrame*, а также, если мы укажем в качестве аргумента не список значений, а всего лишь одно значение, в данном случае продолжительность поездки, у нас уже вернется объект типа *Serious*.

Также *pandas* позволяет группировать данные не только по какому-то одному признаку, а сразу же по группе признаков. Для этого нам просто необходимо добавить еще дополнительный столбец в метод *groupby*.

После группировки данных *pandas* можно уже считать некоторое агрегированное значение по признакам для каждой группы, но если нас интересует какое-либо **распределение значения не только по одному признаку, а сразу же по нескольким признакам**, то это можно сделать, используя метод *agg*.

```
df.groupby(['usertype']).agg({'tripduration':sum, 'starttime':'first'})
```

В результате применения метода возвращается *DataFrame*, в котором индекс - это наш исходный признак группировки, и соответствующие два столбца с нужными значениями.

Также, если нам необходимо **посмотреть изменение значений или вообще какие-либо другие метрики по какому-то одному признаку**, мы можем в этом же словаре, для этого признака, в качестве значения указывать список методов.

```
df.groupby(['usertype']).agg({'tripduration':[sum, min], 'starttime':'first'})
```

Давайте еще рассмотрим тот вариант, когда мы хотим **посчитать какое-нибудь значение**, не используя встроенные функции, а **используя какую-то свою функцию**. Это можно сделать, например, используя лямбда функцию.

```
df.groupby(['usertype']).agg({'tripduration': lambda x: max(x) + 1, 'starttime':'first'})
```

4.5 Работа с несколькими таблицами

Давайте рассмотрим следующую ситуацию. Представьте, что есть некоторый датасет, который состоит не из одной таблицы, а сразу из нескольких. Например, это датасет квартир Airbnb, он состоит из нескольких таблиц. Одна таблица относится к характеристикам самих квартир, другая таблица содержит календарь бронирования, а третья таблица отвечает за отзывы, которые оставляют посетители этих квартир. И нам необходимо каким-то образом связать эти три сущности вместе. То есть, если мы хотим посмотреть связь между тем, какой тип квартиры, с тем, какие отзывы оставляли люди, нам нужно как-то **совместить** эти **два датасета**. Самый простой способ, которым это можно сделать, — это сделать *join*. Вообще, какие бывают виды *join*?

Представим, что у нас есть две таблицы. И у них есть какое-то общее поле, например, ID квартиры. Как мы можем совместить эти два датасета? Допустим, мы можем взять все строки из правой части и добавить их все к левой части. Можем сделать наоборот: все строки из левой

части добавить к правой, либо сделать какое-то пересечение двух таблиц, либо же взять какое-то их общее объединение.

В *pandas* существует два метода *join* таблиц. Один называется *merge*, другой — *join*. Они отличаются тем, что с помощью метода *merge* мы можем связывать две таблицы по каким-то признакам, а в случае *join* мы связываем две таблицы по какому-то общему индексу.

Давайте рассмотрим конкретные примеры. Вернёмся к нашим таблицам. Таблица *listings* содержит подробную информацию о самих квартирах, *reviews* — это некоторые отзывы, которые оставляли люди и *calendar* — когда было совершено бронирование квартиры. Давайте попробуем к таблице *listings* добавить отзывы людей, используя метод *merge*.

```
pd.merge(listings, reviews, left_on=['id'], right_on=['listing_id'])
```

Результатом является некоторая таблица.

По умолчанию метод *merge*, если не указывать, какой метод *join* мы хотим осуществить, **осуществляет** *LEFT JOIN*, то есть добавляет все значения из правой таблицы к левой.

Также мы можем это поменять, просто указав в аргументе нужный метод.

```
pd.merge(listings, reviews, left_on=['id'], right_on=['listing_id'], how='inner')
```

Также, если мы хотим понять, из какой таблицы куда добавились значения, можно поменять значение у параметра *indicator*, который покажет, из каких таблиц были взяты выбранные значения. То есть в *DataFrame* возвращается дополнительный столбец *merge*.

```
pd.merge(listings, reviews, left_on=['id'], right_on=['listing_id'],
        how='inner', indicator=True)
```

Давайте рассмотрим, как осуществляется *join* при применении другого метода. Для этого нам **необходимо у каждой из таблиц, которые мы хотим совместить, задать некоторый индекс**.

```
calendar.set_index('listing_id', inplace=True)
reviews.set_index('listing_id', inplace=True)
```

Теперь нам нужно выбрать левую таблицу, допустим, это будет *calendar*. И применив метод *join*, мы передаём таблицу, которую хотим присоединить.

```
calendar.join(reviews, lsuffix='listing_id', rsuffix='listing_id')
```

Мы видим, что у нас добавились значения к *calendar* из отзывов, и теперь у нас задан новый индекс, и, как мы видим, значений в нашей таблице стало сильно больше.

4.6 Преобразование признаков

Поговорим про преобразование признаков в *pandas*, а именно разберем такие методы, как *map* и *apply*. Давайте загрузим нужные нам библиотеки и загрузим наш сайт велобайка по Нью-Йорку. Напомним, что он представляет из себя информацию про поездки, значит информация о том, откуда выехал велобайк, куда приехал, продолжительность поездки и тип пользователя.

```
import pandas as pd
import numpy as np

df = pd.read_csv('citibike.csv')
df.head()
```

С помощью функции *map* можно **преобразовывать элементы столбца**. Для этого нам нужно создать словарь, где ключу (старое значение столбца) соответствует новое значение после преобразования.

Давайте попробуем изменить столбец *usertype*. Так как *usertype* содержит только два поля, то мы *customer*, соответственно, будем сопоставлять, например, значение 1, и *subscriber* можем сопоставить значение 2, и у нас появится такой словарь.

```
usertype = {'Customer':1, 'Subscriber':2}
```

Далее, давайте применим *map* к нашему столбцу

```
df['usertype'].map(usertype).head()
```

Давайте разберем следующий метод - *apply*, который можно применять ко всему датасету, **например, если мы хотим вытащить минимальное значение для каждого столбца**: для этого достаточно в качестве аргумента *apply* передать нужную нам функцию. Также с помощью *apply* можно **применять преобразования к отдельному столбцу**.

Давайте рассмотрим на примере продолжительности поездок. Столбец содержит время в секундах и с помощью метода *apply* мы можем его превратить в минуты. Здесь преобразование задается с помощью лямбда функции.

```
df['tripduration'].apply(lambda x: x / 60).head()
```

Здесь показано простое преобразование, но часто бывает, что мы пишем какие-то сложные преобразования, какие-то свои функции и с помощью метода *apply*, комбинируя с лямбда функцией можно очень компактно и удобно записывать преобразования.

Также *apply* **работает со строками**: для этого достаточно указать *axis* равному 1 - один из аргументов *apply*. И предыдущие преобразования можно записать в другом виде.

```
df.apply(lambda x: x['tripduration'] / 60, axis=1).head()
```