МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Белгородский государственный технологический университет
им. В.Г. Шухова

Кафедра программного обеспечения вычислительной техники
и автоматизированных систем

Утверждено
научно-методическим советом
университета

МЕТОДЫ АНАЛИЗА ДАННЫХ И МАШИННОГО ОБУЧЕНИЯ

Зуев С.В.

Учебное пособие
для укрупненных групп
09.00.00 и 10.00.00

Белгород
2023

УДК 004.622, 004.624, 004.67
ББК 32.972.13

Составитель: канд. физ.-мат. наук, С.В. Зуев

Рецензенты:

Методы анализа данных и машинного обучения: учебное пособие для укрупненных групп 09.00.00 и 10.00.00 / сост.: С.В. Зуев. .

В учебном пособии содержится теоретический материал по дисциплинам «Методы анализа данных», «Машинное обучение», «Интеллектуальный анализ данных» для студентов 2-3 курсов, обучающихся по направлениям подготовки бакалавриата или специальностям укрупненных групп 09.00.00 и 10.00.00. Кроме того, в пособии имеются задания для самостоятельной работы, примеры их решения, описания практических работ и указания к их выполнению. К пособию прилагается библиотека реализованных и используемых в тексте методов и функций.

Данное издание публикуется в авторской редакции.

УДК 004.622,
ББК 32.972.13

Оглавление

Введение

В начале XXI века в большинстве отраслей экономики во всем мире практически одновременно пришло осознание того, что накопленная информация самого разного характера является главным фактором в конкурентной борьбе. Может быть, большинству руководителей и раньше был ясен этот почти очевидный факт, но только поддержанный технологически он превратился в одно из главных направлений развития современной прикладной науки, а точнее – в новую прикладную науку, которую назвали *Data Science*, или просто *наука о данных*.

За пару десятилетий своего существования, наука о данных претерпела взрывной рост. Возникли новые направления, которые могут сами претендовать на звание наук: *интеллектуальный анализ данных* (*Data Mining*), *тематическое моделирование* (*Topic Modelling*) выросли непосредственно из науки о данных, а такие научные области как компьютерное зрение, искусственный интеллект, машинное обучение, кибернетика, существовали и в прошлом веке, но сейчас приобрели новый облик и оказывают значительно большее влияние на жизнь людей, чем раньше.

Спрос на аналитиков данных, и в целом специалистов в сфере искусственного интеллекта и машинного обучения, непрерывно растет. Любопытно, что на просторах интернета можно встретить оценки, говорящие о спросе на миллиарды аналитиков данных через 10 лет. Население планеты – около 8 миллиардов. Если речь идет о миллиардах, то это – минимум 2 миллиарда. То есть, как минимум каждый четвертый землянин в течение 10 лет должен стать таким аналитиком. С учетом того, что примерно 40% людей не находятся в работоспособном возрасте, приходим к тому, что аналитиком данных должны быть больше, чем 2 работника из 5. Поскольку в течение 10 лет перевести всю мировую экономику на искусственный интеллект и робототехнику не представляется возможным, то такое соотношение сил в пользу аналитиков данных однозначно невозможно, если люди всё еще будут нуждаться в чем-то ещё, кроме информации. Это рассуждение представляет собой простейшую логическую цепочку, которую можно построить, используя информацию о: количестве требующихся специалистов, численности населения Земли сейчас и в прогнозе на 10 лет, доле работоспособного населения, структуре занятости в разрезе IT и остальных отраслей сейчас и в прогнозе на 10 лет с учетом развития робототехники и искусственного интеллекта. Вся эта информация, сформулированная на понятном для машины языке, называется

данными и, как видно из определения, состав данных зависит от предмета изучения и поставленной задачи.

Чуть более точно науку о данных можно определить как пересечение следующих укрупненных областей знания:

- Алгоритмизация и программирование;
- Теория вероятностей, статистика, линейная алгебра и, немного, математический анализ, одним словом – математика;
- Знания в предметной области анализа данных.

Для специалиста – аналитика данных – любого уровня, обладание знаниями во всех трех приведенных областях критически важно (без этого нет специалиста). Поэтому, к сожалению (или к счастью), невозможно выпустить аналитика данных из учебного заведения. Потому что область знаний последнего пункта должна быть живой, из сегодняшней индустрии, и специалист должен в этой индустрии поработать какое-то время. Знания по первым двум пунктам можно получить в вузе, причем именно в виде, подготовленном для восприятия знаний предметной области. Именно этому и посвящено учебное пособие и курсы, читаемые его автором.

Если для читателя материал пособия сложен для восприятия, но это преодолимо, то такая ситуация – нормальна, поскольку сам по себе предмет не является простым. Если же трудности в освоении непреодолимы, то следует изучить пока школьную программу. Возможна обратная ситуация: некоторые вопросы, изложенные здесь, покажутся читателю очевидными и простыми. Помня об этом, пособие написано так, что имеется довольно мало ссылок между главами: считается, что для изучения текущей главы, читатель должен знать и понимать смысл предыдущих глав, но не обязательно именно в таком изложении. Так что если вы не желаете, например, читать про линейную алгебру, которая вам и так уже известна, то можете сразу идти к следующей главе, и так далее.

Если посмотреть, например, на учебный план по специальности «Информационная безопасность автоматизированных систем», то можно найти и алгоритмизацию (Математическая логика и теория алгоритмов), и программирование (множество дисциплин), и теорию вероятностей со статистикой, и алгебру, а также множество дисциплин в предметной области. Зачем же тогда отдельно изучать методы анализа данных, если все составляющие и так уже преподаются? Оказывается, все перечисленное необходимо для того, чтобы понять методы, изучаемые здесь. Ни в одной из указанных дисциплин нет изложения тех методов, что необходимы для анализа данных. Вы можете прекрасно это всё знать и даже иметь замечательный практический опыт в каждой

из тех дисциплин, но не сможете решить ни одного практического задания по анализу данных. Просто потому, что не знаете методов. Поэтому они и вынесены в названия пособия. Успех в решении проблемы чаще всего зависит от того, как именно будут выбраны и применены имеющиеся методы и средства, а это зависит не от подготовки в применении каждого средства, а от подготовки в совместном применении этих средств для достижения заданной цели. На это тоже обращено внимание: взаимосвязь методов раскрывается, где это возможно.

Дисциплина «Методы анализа данных и машинного обучения» тесно связана с теорией вероятности и математической статистикой. Иногда в учебных планах они стоят в одном семестре. Это не должно мешать – можно параллельно изучать несколько тем, и они будут рассмотрены с разных сторон. Основное применение теории вероятностей и математической статистики в деятельности специалиста по защите информации или разработчика программного обеспечения связано именно с анализом данных и процессов, что требует применять знания, полученные при изучении в одной дисциплине, сразу в применении к задачам другой.

Так как на втором курсе студенты еще не являются знатоками многих языков программирования, этот курс будет использовать один из самых просто читаемых и удобных для начального изучения науки о данных языков – Python. Предлагаемые лабораторные работы сделаны для дистрибутива Anaconda, включающего как научную среду Spyder, так и многофункциональную среду Jupyter Notebook.

Пособие содержит теоретический материал, примеры решения задач, упражнения и указания для их решения. В тексте приведена большая часть программных кодов, остальные собраны в библиотеку и доступны для скачивания. В Приложении – описания практических работ и методические указания по их выполнению.

В результате изучения курса «Методы анализа данных и машинного обучения», студент второго курса должен ориентироваться во всём материале, изложенном в настоящем пособии, иметь представление о методах подготовки, извлечения, обработки и визуализации данных, о математических и программных средствах, используемых для этих целей. Кроме того, уметь писать простые скрипты на Python, предназначенные для решения типовых задач Data Science, а также обладать навыками применения стандартных библиотек Python для решения задач анализа данных.

Курс состоит из Введения, пяти глав, Заключения и Приложения.

При подготовке настоящего пособия мною были бессовестно использованы разработки Ивана Притчина (с его согласия), легкие ремарки Петра Кабалянца и большое количество вопросов студентов, некоторые из которых ставили меня в тупик. Всем этим людям я безмерно благодарен и, конечно, без них эта работа не была бы начата и уж тем более закончена.

Глава I. Python для анализа и визуализации данных

1. Общие положения и дзен языка Python

Происхождение и источник. Язык программирования Python разработан Гвидо ван Россумом и увидел свет в 1991 году. Ясно, что создавался он тогда не для анализа данных, а просто как язык, позволяющий писать программы, понятные не только их разработчику. До сих пор наличие большого количества возможностей и инструментов разработки в сочетании с простой читаемостью кода делает Python весьма элегантным инструментом в создании скриптов, которые можно использовать для анализа или запутывания данных, управления или дезорганизации систем, информирования или дезинформирования пользователей и систем искусственного интеллекта. Огромное количество обучающих материалов по Python всех версий имеется в открытом доступе в интернете и в печатных изданиях. Также, как и курсов по его изучению.

Имеется некая философия, которой, как считается, должны придерживаться разработчики на языке Python. Это, так называемый дзен языка Python, который выдается по команде import this. Из всего того, что там можно прочесть, заслуживают особого внимания два утверждения:

- Встретив двусмысленность, отбрось искушение угадать;
- Если реализацию сложно объяснить, то идея плоха.

Для аналитика данных эти два положения являются значимыми, поскольку первое прямо адресует к тому, с чем обычно борется аналитик данных, а второе чаще всего просто экономит его время. Остальные положения можно принимать или не принимать, но считается, что разработчики на Python (по-русски просто питонисты), в основном, придерживаются этой философии.

Официальный сайт python.org предоставляет всё, что нужно для работы в Python, в основном, бесплатно и с нужными обучающими ресурсами, правда, на английском языке. Современная версия языка – Python 3.* доступна с 2018 года на официальном сайте на основе публичной лицензии, как и более ранние версии. Кроме того, ее можно получить в уже установленном и сконфигурированном виде в составе среды Anaconda (anaconda.com), которая также является свободно распространяемой и работает на всех распространенных операционных системах.

Для решения разного рода прикладных задач, Python использует различные библиотеки, которых очень много. В среде Anaconda многие библиотеки уже включены и при написании кода их нужно просто импортировать командой import.

Построение и синтаксис. Для разграничения блоков кода в Python используются отступы. Пробелы и отступы внутри круглых и квадратных скобок игнорируются, что помогает писать удобно читаемый код. Например, без особых объяснений понятно, что данный код

```
# список списков
a_matrix = [[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]]
```

представляет собой матрицу. В конце строки можно ничего не ставить, но если вы хотите в этой же строке разместить еще одну команду, то надо поставить точку с запятой. Если строка является управляющей инструкцией, то она обязана заканчиваться двоеточием.

Пустая строка интерпретируется как окончание блока кода независимо от того какой отступ следует дальше. Поэтому при копировании кода с текстового файла следует проверять все отступы и пустые строки.

Модули. Помимо основного набора инструкций языка Python, которые загружаются сразу после запуска среды программирования, существует значительно большее множество инструментов – функций, библиотек, которые загружаются по мере необходимости и по решению пользователя. Библиотеки, которые содержат наборы функций и методов по определенной теме, называются *модулями*. Чтобы воспользоваться инструментами модуля, его надо *импортировать*, то есть воспользоваться командой

```
import numpy
```

где `numpy` есть имя модуля. В данном случае это модуль для работы с многомерными массивами данных, а состав его можно узнать на многочисленных Интернет-ресурсах, просто введя в поиске название модуля. Каждую функцию или класс из модуля можно импортировать отдельно, если нет необходимости во всём модуле. При импорте можно переименовывать модули или их элементы для удобства. Например, такая инструкция импортирует функцию возведения в степень из модуля вещественных математических функций и называет её просто `pwr`:

```
from math import pow as pwr
```

и далее

```
pwr(3, q)
```

будет выдавать значение q^3 если, конечно, q определено к моменту вызова.

Перечень модулей языка Python можно пополнять самостоятельно – напишите скрипт и дайте ему расширение *.py* – после этого его можно импортировать, если разместить его в пути поиска интерпретатора. Наиболее употребительные модули для решения задач из стандартных областей уже размещены обычно в пути поиска интерпретатора, если установлена интегрированная среда разработчика.

При импорте модулей следует относиться внимательно к именам переменных и функций. Импортируя что-то, вы добавляете в *пространство имен*, то есть во множество всех имен, с которыми оперирует в данный момент ваша среда, имена из импортируемого модуля и, если встретятся совпадающие, то произойдет перезапись имён со всеми вытекающими плачевными последствиями.

Функции. Как и в любом языке программирования, *функция* в Python есть правило, которое принимает параметры (их количество может быть от нуля до 255) и выдаёт какой-то результат. Например, если необходимо выдать последний символ строки s произвольной длины, то это сможет сделать функция `lastc`, определенная так:

```
def lastc(s):
    """Эта функция возвращает последний символ строки s"""
    return s[-1]
```

Функции в Python могут выступать в качестве аргументов, быть присвоены переменным и переданы в другие функции. Такие объекты называются объектами *первого класса*.

Параметрам можно назначать аргументы (значения) по умолчанию. Нижеследующие коды вместе с определенной функцией `lastc` показывают, что имеется в виду.

```
def app_to_you(fun):
  """вызывает функцию fun с аргументом 'you' """
  return fun('you')

my_lastc = lastc             # другое имя для lastc
guess1 = app_to_you(my_lastc)      # = u

def flastc(s='f'):
  return lastc(s)

guess2 = app_to_you(flastc)     # = u
guess3 = flastc()                  # = f
```

Функции являются весьма важным инструментом и далее будут широко применяться.

Исключения. Ситуация, в которой программа ведет себя не так, как предполагал разработчик, называется *исключением* или (в информационной безопасности) *уязвимостью*. Наиболее частые источники исключений – деление на ноль, неподобающий формат выражения, а также специальный аргумент функции. Для обработки исключений в Python предусмотрена конструкция `try – except`, включающая также директивы `else` и `finally`. Ниже примеры их использования.

```
try:
    a = float(input("Введите число: "))
    print (100 / a)
except ValueError:
    print ("Это не число!")           # если введено не число
except ZeroDivisionError:
    print ("На ноль делить нельзя!")  # если введен 0
except:
    print ("Неожиданная ошибка.")     # не поделилось почему-то
else:
    print ("Код выполнился без ошибок")# всё получилось
finally:
    print ("Сюда дошел!")             # это в любом случае
```

Существенно, что блок except: без конкретного типа ошибки должен быть последним из всех блоков except.

Нежелательно слишком сильно увлекаться, помещая в блок `try`: большие участки кода. При этом можно получить работающий код, который будет делать вовсе не то, что вы хотели.

Упражнения

1.1.1. Установите на свой компьютер среду Anaconda, запустите Jupyter Notebook и напишите, не используя арифметические операции, код из двух-трех строк, выдающий ответ «да», если введенное 10-значное натуральное число делится на 8.

1.1.2. Напишите модуль для нахождения НОД числа 8 и заданного 10-значного числа, не используя арифметические операции, но с использованием функций пользователя.

1.1.3. Допишите модуль из упражнения 1.1.2, добавив исключения.

2. Структуры и типы данных

Имеются следующие встроенные типы данных:

- `none` – неопределенное значение переменной;
- `boolean` – одно из `true` или `false`
- `int` – целое число
- `float` – число с плавающей точкой
- `complex` – комплексное число
- `list` – список
- `tuple` – кортеж
- `range` – диапазон
- `str` – строки
- `bytes` – байты
- `bytearray` – массивы байтов
- `memoryview` – специальные объекты для доступа к внутренним данным объекта через `protocol buffer`
- `set` – множество
- `frozenset` – неизменяемое множество
- `dict` – словарь

Что касается первых четырех, то их использование очевидно. Остальные рассмотрим подробнее.

Комплексное число задается в виде пары вещественных чисел и имеет вид, например, $1 + 2j$, $3j$, $1 + 0j$ и т.п. Если нужно преобразовать число в комплексное из другого формата, то имеется функция `complex()`, которая с легкостью это сделает. Все арифметические операции с комплексными числами аналогичны тем, что есть для `float`. Но сравнения, естественно, не работают. Когда необходимы более сложные вычисления, на помощь приходит модуль `cmath`, в котором есть все (или почти все) нужные математические операции над комплексными числами и предусмотрен вызов трансцендентных констант в комплексном формате.

Тип данных `list` представляет собой упорядоченный изменяемый набор данных любого типа, задаваемый, например, так

```
v = 3/2
z = -1
lst = [‘string’,int(v),complex(z)]
```

где `lst` – имя списка, а в квадратных скобках через запятую перечислены элементы списка, которые в данном случае являются строкой, целым числом и комплексным числом, соответственно.

В отличие от списка, кортеж является упорядоченным неизменяемым множеством. Задается так же, как список, только используются круглые скобки. И, разумеется, не может быть изменен.

Диапазоны `range` используются очень часто при формировании циклов и заполнении списков, да и во многих других случаях. Формат задания диапазона такой:

```
range(start=0, stop, step=1)
```

где параметры start и step являются необязательными. Например,

```
range(1, 17, 2)      # 1, 3, 5, 7, 9, 11, 13, 15
```

конец диапазона сам в этот диапазон не входит.

Строки в Python есть последовательности символов, которые задаются между кавычками. Кавычки могут быть как одинарными, так и двойными, но с обеих сторон должны быть одинаковыми. Специальные символы внутри строки можно поставить, используя обратную косую черту:

```
st = '\t'     # tabular
len(st)    # =1
```

Если необходима сама обратная косая черта, то делают неформатированную строку с помощью префикса `r`:

```
st = r'\t'    # \t
len(st)    # =2
```

Тройные кавычки заключают в себе многострочный текст (с переносами на другую строку). Для строк работают операции “+” и “*”: первая осуществляет конкатенацию строк, а вторая – повторяет строку заданное число раз. Еще строки можно разделять на части и собирать методами `split` и `join`. Первый работает так:

```
sentence = "Можно составить список из слов этого текста"
words_list = sentence.split() # ['Можно','составить',…]
substring = sentence.split('и') # ['Можно состав','ть сп',…]
```

а второй

```
sentence2 = " Но лучше не надо"
text = '.'.join([sentence, sentence2])
""" Можно составить список из слов этого текста. Но лучше не
надо """
```

Тип данных `set`, или множество, есть совокупность неупорядоченных элементов без повторов.

```
s = set()     # пустое множество
s.add(1)         # s={1}
s.add(2)         # s={1,2}
s.add(2)         # s={1,2}
y = 2 in s    # true
z = 3 in s    # false
```

Множества применяются для быстрых проверок больших совокупностей элементов на принадлежность некоторой последовательности:

```
# a set of the stop-words
stopwords_list = [“a”, ”an”, ”at”, “yet” , ”you”]
stopwords_set = set(stopwords_list)
“zip” in stopwords_set # false
```

Другой способ применения – получение уникальных элементов в наборе данных (так как во множестве нет повторяющихся элементов):

```
item_list = [1,2,3,1,2,3]
num_items = len(item_list)   #6
item_set = set(item_list)    # {1,2,3}
num_distinct_items = len(item_set) #3
distinct_item_list = list(item_set) #[1,2,3]
```

Словарь (dict) или, иначе говоря, ассоциативный список, есть структура данных, в которой *значения* (5, 4, 3) связаны с *ключами* (“Ivan”, “Irina”, “Petr”):

```
empty_dict = {}  # пустой словарь
grades = {“Ivan” : 5, “Irina” : 4, “Petr” : 3}
```

Доступ к значению осуществляется по ключу:

```
Ivans_grade = grades[“Ivan”]    # 5
```

Проверка наличия ключа в словаре делается с помощью оператора in.

Словари имеют метод get():

```
Ivans_grade = grades.get(“Ivan”, 0)   # 5
Kate_grade = grades.get(“Kate”, 0)    # 0
```

С помощью get можно не обрабатывать исключения, так как при отсутствии ключа в словаре выдается значение по умолчанию. Присваивание значения ключу:

```
grades[“Ivan”] = 2  # заменено 5 на 2
grades[“Kate”] = 3  # добавили оценку для Kate
```

Пара полезных словарей из библиотек: defaultdict и Counter. Они импортируются из библиотеки collections и работают так.

```
from collections import defaultdict as dd

dd_list = dd(list)  # значения dd_list – это списки
dd_list[2].append(1) # теперь dd.list содержит {2: [1]}

dd_dict = dd(dict)  # значения dd_dict – это словари

dd_dict[“Ivan”][“City”] = “Valuiki”
# dd_dict сейчас: {“Ivan”: { “City” : “Valuiki”}}
```

Counter трансформирует последовательность значений в словарь, в котором ключам поставлены в соответствие частотности (а именно, сколько раз встретился тот или иной элемент в наборе). Пример:

```
from collections import Counter

c = Counter([0, 1, 2, 0])
# результат: c = {0 : 2, 1 : 1, 2 : 1}
```

С его помощью можно легко считать, например, частотность слов:

```
word_counts = Counter(document_words_list)
```

У словаря Counter есть метод most_common, который полезен для вычисления наиболее часто встречающихся слов:

```
for word, count in word_counts.most_common(10):
  print(word, count)
```

Исполнение этого кода даст словарь из 10 наиболее часто встречающихся слов в списке document_words_list.

Упражнения

1.2.1. Создайте произвольный двухуровневый список из целых, вещественных и комплексных чисел. Выберите из него только комплексные числа и запишите их в кортеж.

1.2.2. Создайте множество из 100 первых элементов последовательности частичных сумм ряда Фибоначчи. Выведите число значащих цифр у элемента последовательности с заданным номером.

3. Управляющие конструкции и общие методы

Управляющие конструкции. Мы рассмотрим работу условного оператора if и циклов for и while. Формат управления ветвлением такой:

```
if x > 1:
  x = 1 / x
elif x < 1:
  x = 1 / (x + 1)
else:
  print("x = 1")
```

Есть также формат однострочного трехместного (тернарного) оператора

```
parity = "четное" if x % 2 == 0 else "нечетное"
```

Рассмотрим цикл `while`. Он работает пока не выполнится заданное условие:

```
x = 0
while x<10:
  print(x, "меньше 10")
  x += 1
```

Другой вид цикла – `for` – используется чаще:

```
for x in range (10):
  print(x, "меньше 10")
```

делает то же самое, что и код выше. Для управления внутри цикла используются операторы `continue` и `break`:

```
for x in range(10):
  if x == 3:
    continue
  if x == 5:
    break
  print(x)
```

При выполнении фрагмента будут напечатаны значения 0, 1, 2 и 4.

Работа с логическими значениями. В Python булевы литералы пишутся с заглавной буквы: `True`, `False`. Если ожидается логический тип `Boolean`, то в Python в этом месте может быть любое значение. Следующие элементы интерпретируются как `False`: собственно, `False`; `None`; пустые списки `[]`, словари `{}`, множества, строки `""`, `0`. Всё остальное интерпретируется как `True`. Это позволяет применять оператор `if` для проверок на наличие пустых структур данных. Пример

```
s = some_string_function()      # возвращает строку
if s:
  first_ch = s[0]         # возвращает первый символ в строке
else:
  first_ch = ""
```

Это можно сделать проще:

```
first_ch = s and s[0]
```

поскольку логический оператор `and` возвращает второе значение при истинном первом и первое, если оно ложное. Выражение

```
safe_x = x or 0
```

согласно такой же логике, будет всегда числом.

Встроенная функция `all` языка Python принимает список и возвращает `True` только когда каждый элемент в списке дает `True`. Встроенная функция `any` языка Python принимает список и возвращает `True`, когда хотя бы один элемент в списке дает `True`.

Сортировка. Осуществляется методом `sort()`, который упорядочивает элементы списка прямо внутри него, без выделения дополнительной памяти. Встроенная функция `sorted` упорядочивает и возвращает новый список.

```
x = [4,1,2,3]
y = sorted(x) # список y: [1,2,3,4]
w = ['Флинт','Пират','Корсар','Джонни Депп']
x.sort() # x теперь [1,2,3,4]
w.sort() # w стал ['Джонни Депп', 'Корсар', 'Пират', 'Флинт']
```

По умолчанию, сортировка производится по не-убыванию. Чтобы сортировать по не-возрастанию, нужно задать аргумент `reverse=True`. Кроме того, можно использовать параметр key, который является функцией от элемента сортируемой коллекции. Например, сортировка списка по абсолютному значению, в убывающем порядке:

```
x = sorted([-4,1,-2,3], key=abs, reverse=True)
# x = [-4,3,-2,1]
```

Сортировка слов и их частотностей по убыванию:

```
wc = sorted(word_counts.items(),
          key=lambda word, count: count,
          reverse=True)
```

Здесь использована `lambda`-функция, которая позволяет задать функцию, не давая ей имя. Она работает так: `lambda: arg1, arg2,… : value`. То есть, указывает как создать значение из заданных аргументов. Значение – одно.

Генераторы последовательностей. При необходимости преобразования списка в другой список выбором определенных элементов или изменяя имеющиеся элементы, следует использовать *генераторы последовательностей* или (другое название) *списковые включения*. Генератор последовательности можно построить так:

```
# четные числа меньше 5
even numbers = [x for x in range(5) if x%2 == 0]

# квадраты чисел от 0 до 4
squares = [x*x for x in range(5)]

# квадраты четных чисел
even_squares = [x*x for x in even_numbers]
```

Таким же образом можно создать и словари, и множества:

```
# словарь квадратов чисел
square_dict = {x : x*x for x in range(5)}

# множество квадратов (будет только 1)
square_set = {x*x for x in [-1,1]}
```

Если имя переменной не используется в дальнейшем коде, его можно заменить символом подчеркивания:

```
# список из 0 длиной как y even_numbers
zeros = [0 for _ in even_numbers]
```

Последняя конструкция показана в учебных целях и может быть заменена на:

```
# список длиной как y even_numbers
zeros = [0] * len(even_numbers)
```

В генераторе может быть несколько циклов и последующие могут использовать результаты предыдущих. Примеры:

```
# 100 пар (0,0) … (19,4)
pairs = [(x, y) for x in range(20) for y in range(5)]
# только x>y
incr_pairs = [(x, y) for y in range(5) for x in range(y+1,20)]
```

Упражнения

1.3.1. Создайте короткий скрипт, находящий все нули заданной алгебраической функции на заданном интервале (переменная формата float).

1.3.2. Напишите скрипт для выборки из заданного текста всех слов, начинающихся на «в», и сортировки их в алфавитном порядке.

1.3.3. Скачайте любую статью (например, с Habr.ru) и сформируйте из нее строку. Посчитайте частотности для всех слов, содержащихся в ней.

4. Инструменты функционального и объектно-ориентированного программирования

Функции-генераторы. *Генератор* – это объект, который можно перебрать последовательно, но его значения предоставляются по требованию (так называемое *ленивое вычисление*). Зададим функцию-генератор с использованием оператора `yield`:

```
def lazy_range(n):
  i=0
  while i<n:
     yield i
     i += 1
```

Это последовательность `range`, реализованная ленивыми вычислениями: она выдает члены последовательности по одному:

```
for i in lazy_range(10):
  print(10-i)
```

В Python 3 последовательность `range` уже ленивая, так что нет необходимости строить подобные конструкции. В частности, можно задавать бесконечную последовательность:

```
def natural_numbers():
  n = 1
  while True:
     yield n
     n += 1
```

но на практике такое мало применимо.

В отличие от списка, генератор можно просмотреть всего один раз. При многократном применении понадобиться каждый раз создавать новый генератор или воспользоваться списком.

Есть еще один способ создать генератор – использовать генератор последовательности с оператором `for`, обрамленный круглыми или квадратными скобками:

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i%2 == 0)
```

Заметим здесь, что словарь `dict` имеет метод `items()`, который возвращает список пар «ключ-значение». В Python 2 был метод `iteritems()`, который «лениво» предоставлял по одной паре по мере навигации по словарю, но в Python 3 метод `items()` уже делает это и поэтому `iteritems()` уже нет.

Модуль `random`. Импортируя этот модуль, можно генерировать псевдослучайные числа и последовательности. Например, три равномерно распределенные случайные величины:

```
import random
uniform_randoms = [random.random() for _ in range(3)]
```

Можно сгенерировать воспроизводимые результаты. Для этого нужно задать зерно-генерации методом `random.seed`:

```
random.seed(10)
r1 = random.random()
random.seed(10)
r2 = random.random()
```

в результате `r1` совпадет с `r2`. Не совпадет, если seed будет на разных числах (например, во втором будет 7 вместо 10).

Метод `random.randrange()` выбирает последовательность случайных чисел (ПСЧ) из последовательности range. Метод `random.shuffle()` перемешивает элементы в списке в случайном порядке (аргументом является список). Другой метод, работающий со списком – это `random.choice()`. Он выбирает случайный элемент из списка и список при этом остается прежним.

Важным инструментом являются методы выборки без возвращения и с возвращением. Выборка без возвращения делается методом random.sample():

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 3)
```

так выбираются три выигрышных номера из 60. Если нужно произвести выборку с возвращением, то просто применяется `random.choice()` нужное число раз.

Модуль re. Импортируя модуль `re`, мы подключаем библиотеку так называемых *регулярных выражений*. Их работа направлена на осуществление поиска в текстах по шаблону. Набор регулярных выражений довольно обширен и их применение описывается во многих отдельных руководствах. Для того чтобы дать понимание о том, как они работают, приведем такой пример:

```
import re

# будут выведены все слова, в которых содержится хотя бы одна
# буква, за которым следует “лёт”
string = "самолёт, вертолёт, автомобиль"
print(re.findall("\w+лёт", string))

# замена цифр тире
print(re.sub("[0-9]", "-", "АИ-95")) # АИ---

# разбитие строки по буквам а и е
re.split(“[ae]”, “Москва-Петушки”) # ['Москв', '-П', 'тушки']
```

Объектно-ориентированное программирование. Выше мы рассматривали структуру данных `set` – множества. Для понимания организации парадигмы объектно-ориентированного программирования в Python, представим, что этот тип данных отсутствует и его надо определить. Создадим для этого *класс* `Set`, который будет описывать общие свойства *экземпляров* класса, то есть, множеств. Как и множество, экземпляр класса Set должен позволять добавлять (`add`) и удалять (`remove`) элементы, а также проверять содержится ли заданный элемент в нем (`contains`). Все это мы можем задать с помощью компонентных функций (методов класса), доступ к которым обеспечивается через точку после имени класса. Класс определяется ключевым словом class, далее описываются методы, первый аргумент которых всегда self. Выглядит это, например, так:

```
class Set:

  def __init__(self, values=None):
```

```
    self.dict = {}
    if values is not None:
       for value in values: self.add(value)

  def __repr__(self):
    return "Set: " + str(self.dict.keys())

  def add(self, value):
    self.dict[value] = True

  def contains(self, value):
    return value in self.dict

  def remove(self, value):
    del self.dict[value]
```

Далее можно создавать экземпляр класса следующим образом:

```
s = Set([1,2,3])
s.add(4)
print(s.contains(4)) # True
s.remove(3)
print(s.contains(3)) # False
```

Инструменты функционального программирования. В качестве альтернатив генераторам последовательностей, можно использовать встроенные функции `map`, `reduce` и `filter`. Это функциональные способы позволяют использовать меньшее количество явных циклов и ветвлений Пример применения `map`:

```
def double(x):
  return 2*x

xs = [1,2,3,4]
# генератор последовательности
twice_xs1 = [double(x) for x in xs]

# использование map
twice_xs2 = list(map(double, xs))

# если функция нужна только здесь:
twice_xs2 = list(map(lambda x: 2*x, xs))
```

Кроме того, можно воспользоваться методом `partial` из модуля `functools`, который возвращает функцию с частичным приложением аргументов:

```
from functools import partial

doubler = partial(map, double)
twice_xs3 = list(doubler(xs))
```

В результате получим три тождественных списка `twice_xs1`, `twice_xs2`, `twice_xs3`, равных `[2,4,6,8]`.

Функцию `map` можно использовать и для других целей. Например, для перемножения списков чисел:

```
def multiply(x,y):
  return x*y

product_list = list(map(multiply, [1,2], [4,5]))
# результат: [4,10]
```

Вместо ветвления функционально используется `filter`, пример (продолжая предыдущий код):

```
def is_even(x):
  return x%2 == 0

x_evens1 = [x for x in xs if is_even(x)]
x_evens2 = filter(is_even, xs)
evener = partial(filter, is_even)
x_evens3 = list(evener(xs))
```

Будет сформировано три одинаковых списка `[2,4]`.

Наконец, метод `reduce` делает свертку списка по указанной функции. Вот пример:

```
from functools import reduce

x_product1 = reduce(multiply,xs)
p_product = partial(reduce, multiply)
x_product2 = p_product(xs)
```

Получаем две одинаковых свертки списка: `x_product1` и `x_product2`, равные `1*2*3*4 = 24`. При этом уже нет необходимости сообщать тип данных результату работы `partial`: если в результате его действия получается число, то это не обязательно.

Есть возможность итеративно обойти список, используя элементы и их индексы. Для этого подходит встроенная функция `enumerate`, которая создает кортежи в формате (индекс, элемент). Например, для некоторого списка документов (здесь – пустого)

```
documents = []
```

можно произвести действие `do_something` с i-м документом, используя такой короткий код:

```
for i, document in enumerate(documents):
  do_something(i, document)
```

вместо использования более громоздкого:

```
for i in range(len(documents)):
  do_something(i, documents[i])
```

Также может быть полезен приём с использованием безымянных аргументов – `args` и аргументов по ключу – `kwargs`. Это немного похоже на фокус:

```
def miracle(*args,**kwargs):
  print("безымянные аргументы:", args)
  print("аргументы по ключу:", kwargs)

miracle(1, 2, key="красный", key2="воробей")
```

Будет напечатано:

```
безымянные аргументы: (1,2)
аргументы по ключу: {'key2': 'воробей', 'key': 'красный'}
```

В данном примере:

```
def another_miracle(x, y, z):
  return x+y+z

x y list = [1,2]
z dict = {"z":3}
print(another_miracle(*x_y_list, **z_dict))
```

программа выдаст значение 6.

Создадим, для примера, функцию-удвоитель, которая принимает произвольное число входящих аргументов:

```
def f2(x, y):
  return x+y
def doubler(f):
  def g(*args, **kwargs):
     return 2*f(*args, **kwargs)
  return g

g = doubler(f2)
print(g(1, 2))   # 6
```

Упражнения

1.4.1. Создайте множество из случайных натуральных чисел размером в 10000 элементов. Осуществите из него случайную выборку 10 элементов без возвращения, а затем с возвращением, не используя методы sample **и** choice**. Зафиксируйте время выполнения. Сделайте то же самое с использованием указанных методов модуля** random **и сравните скорости выполнения.**

1.4.2. С помощью регулярных выражений определите сколько раз, с учетом словообразований, в Гражданском кодексе РФ содержится слово «право».

1.4.3. Создайте класс Frac, типичный экземпляр которого является обыкновенной дробью. Опишите методы обращения, сложения и умножения.

1.4.4*. Создайте класс Descr, типичный экземпляр которого является кортежем из четырех пар чисел – координат четырехугольника. Опишите методы этого класса, позволяющие установить равенство четырехугольников и их подобие.

1.4.5. Напишите функцию, которая вычисляет векторное произведение в 3-мерном пространстве с помощью map**.**

1.4.6*. Напишите скрипт для вычисления определителя матрицы с помощью методов функционального программирования.

5. Средства визуализации данных

Библиотека `Matplotlib`. Несмотря на то, что для Python существует великое множество различных модулей для визуализации данных, мы сосредоточимся на библиотеке `matplotlib`. Причина в том, что этот инструмент позволяет сравнительно просто строить почти любые виды графиков и диаграмм, что просто необходимо в анализе данных, так как наглядное представление упрощает восприятие информации.

Использовать будем модуль `matplotlib.pyplot`. Он позволяет шаг за шагом выстроить визуализацию и по ее завершении сохранить результат в файл методом `savefig()` или вывести на экран методом `show()`.

Начнем с простого линейчатого графика, построенного из списочных данных. Допустим, нам надо визуализировать два ряда данных – годы и средний возраст жителей России по данным Росстата (https://rosstat.gov.ru/storage/mediabank/demo14.xls). Далее мы научимся брать данные из MS Excel, но сейчас нам нужно лишь несколько списков – их можно просто забрать оттуда копированием. Разместим их коде, который здесь приведен частично (полностью – в файле в av_pop.ipynb в архиве scripts.zip).

```
from matplotlib import pyplot as plt

years = [1926,1939,1959,1970,1979,1989,2002,2004,2005,2006,
2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,
2019,2020,2021,2022]
age dict = {}
age dict['2'] =
[14114,13806,13353,9326,10523,12032,6399,6660,6916,7066,7234,7
433,7671,7968,8051,8380,8687,8899,9262,9512,9582,9347,9032,857
9,8080,7597]
```

```
…
age_dict['72'] =
[2212,2426,4303,5806,8200,9646,12469,12325,12242,12358,12605,1
3111,13554,14210,14219,14380,14099,13587,13377,13086,13230,135
06,13797,14361,14686,14793]
keys = list(age_dict.keys())
pops sum = [sum(alist[i] for alist in age dict.values()) for i
in range(26)]
av ages = [sum(int(a)*age dict[a][y] for a in
keys)/pops_sum[y] for y in range(26)]
plt.plot(years, av_ages, color='red', marker='o',
linestyle='solid')
plt.title("Средний возраст")
plt.ylabel("лет")
plt.show()
```

В результате получим график, показанный на рис. 1.

Рис. 1. График среднего возраста жителей России (из данных Росстата)

Столбчатые диаграммы. Если нужно показать изменения некоторой величины на дискретном множестве элементов, то используются столбчатые диаграммы. Их можно также использовать для построения гистограмм сгруппированных значений числового ряда, то есть для исследования распределения чисел в ряду. Рассмотрим соответствующие примеры.

```
cars = ["Audi","Renault","VW","Toyota","Lada"]
num_owners = [4, 3, 5, 7, 4]
```

```
xs = [i+0.1 for i in range(cars)]

plt.bar(xs, num owners)
plt.ylabel("Число владельцев")
plt.title("Популярные автомобильные марки")
plt.xticks([i+0.1 for i in range(cars)], cars)

plt.show()
```

Этот скрипт выдаст график, изображенный на рис. 2. Введение и обработка переменной `xs` нужна для того, чтобы сместить столбцы в центр интервалов: по умолчанию ширина столбца 0,8 интервала и добавление слева отступа 0,1 интервала даст положение ровно по центру. Метки с названиями автомобильных марок размещаются командой `plt.xticks` там, где нужно. В данном случае – от начала столбца.

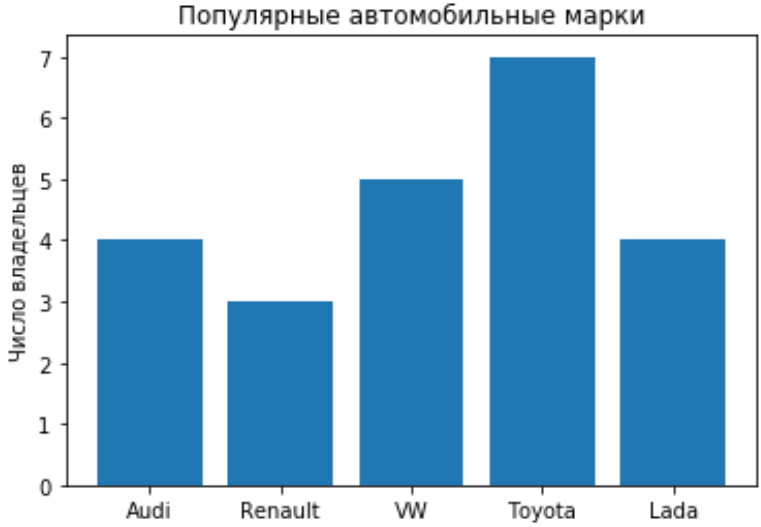


Рис. 2. Среднее число владельцев популярных марок автомобилей.

Следует отметить, что если бы мы начали отсчет по оси с 3, а не с 0, то Renault был бы вообще не популярен, а популярность Audi и Toyota визуально различалась бы в 4 раза. Это, разумеется, не соответствует действительности. Поэтому настоятельно не рекомендуется прибегать к такому приему, как смещение начала отсчета по вертикальной оси. Это значительно искажает восприятие данных.

Точечные диаграммы. Matplotlib предоставляет поистине неисчерпаемые возможности двумерной визуализации данных, но нельзя описать их все в одном маленьком параграфе. Просто знайте, что

возможно почти всё, а как сделать – читайте документацию соответствующих модулей. Для дальнейшего нам, помимо графиков и столбчатых диаграмм, понадобятся точечные диаграммы, так как они важны для построения диаграмм рассеяния (`scatterplot`) для визуализации кластеров и вообще связей между выборками данных. Поэтому опишем, вкратце, как построить такую диаграмму. Пусть имеется три списка одинаковой длины – дни первого заболевания (`days`), вес при рождении (`weights`), идентификаторы (`labels`), описывающие данные о заболеваемости поросят анемией при известном их весе при рождении. Напишем такой скрипт (matplotlib импортирован уже):

```
days = [16,14,15,26,23,27,13,14,14]
weights = [1.45,1.37,1.22,0.95,1.02,0.98,1.35,1.29,1.25]
labels = ['s12', 's16', 's35', 'o31', 'o47', 'o98', 'o11',
          'o14', 'o76']

plt.scatter(days, weights)

# назначаем метки
for label, days_count, weights_count in zip(labels, days, weights):
     plt.annotate(label,
                  xy = (days_count, weights_count),  #задаем метку
                  xytext = (5,-5),                   #смещаем метку
                  textcoords = "offset points')

plt.title("Зависимость заболеваемости от веса при рождении")
plt.xlabel("День заболевания")
plt.ylabel("Вес при рождении, кг")
plt.show()
```

Рис. 3. Диаграмма рассеяния для визуализации сроков заболеваемости анемией и веса при рождении.

Вот, собственно, и есть тот минимум средств визуализации, который необходим для дальнейшего изучения.

Упражнения

1.5.1. Постройте графики стандартного отклонения температур по месяцам года для Москвы и Анадыря, пользуясь данными Gismeteo (https://www.gismeteo.ru/diary).

1.5.2. Постройте три столбчатые диаграммы населения 5 стран мира (по вашему выбору) в 2000, 2010 и 2020 годах по данным ресурса worldometer.

1.5.3. Постройте диаграмму рассеяния для инцидентов информационной безопасности по следующим данным:

Месяц	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
К-во инцидентов	**2**	**14**	**11**	**3**	**1**	**4**	**3**	**21**	**15**	**9**	**4**	**3**
Ср. ущерб	**100**	**12**	**15**	**45**	**32**	**21**	**33**	**67**	**87**	**56**	**91**	**115**

Разделите инциденты по уровням ущерба и сопоставьте каждому уровню размер маркера, затем постройте диаграмму, отложив месяцы по горизонтальной оси.

Глава II. Математические основы анализа данных

1. Линейная алгебра

Определение вектора. *Вектор* – это объект, который можно представить списком чисел, являющийся в то же время элементом множества, которое называется *векторным пространством*. В этом множестве определены операции сложения, умножения на число, скалярного произведения, а также действие линейного оператора – линейное отображение векторного пространства в себя. Набор чисел будет вектором, если вы ***понимаете*** его вектором, то есть вы считаете его элементом множества таких же наборов и можете производить с ним указанные выше операции. То есть, `[2.0,3.0,4.0]` будет вектором,

если вы считаете его принадлежащим к множеству наборов из трех вещественных чисел и можете:

- сложить с другим таким набором и получить третий:

```
[2.0,3.0,4.0] + [3.0,-1.5,5.0] = [5.0,1.5,9.0];
```

- умножить на число:

```
2.2*[2.0,3.0,4.0] = [4.4,6.6,8.8];
```

- перемножить два набора скалярно:

```
[2.0,3.0,4.0]  [3.0,-1.5,5.0] = 18.5;
```

- подействовать на него оператором, представляемым, например, матрицей $M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$ и получить вектор `[2.0,-4.0,3.0]`.

Надо сказать, что строгое определение вектора в линейной алгебре, конечно, иное. Но в программировании и анализе данных под вектором понимается просто как определенного размера набор чисел. Числа в наборе называются *компонентами* вектора. От строгого математического определения остается еще требование, чтобы операции над векторами не выводили из пространства этих самых векторов. Его в программировании обычно строго не формулируют, но стараются придерживаться. Но список вот такого, например, вида: [3, 2.1, “London”, True, ‘0b001’], вектором не является даже в программировании и анализе данных. Потому что над ним невозможно произвести те операции, которые указаны выше.

Функции – операции над векторами в Python. Давайте определим функции в Python, которые осуществят описанные выше операции, если векторы представлены списками.

```
# сложение векторов
def vector_add(v,w):
  return [v_i + w_i for v_i,w_i in zip(v, w)]

# суммирование векторов из списка vectors
def vector_sum(vectors):
  return reduce(vector_add,vectors)

# умножение вектора v на число c
def scalar_multiply(c, v):
  return list(map(lambda x: c*x, v))

# скалярное произведение векторов
def dot(v, w):
  return sum(v_i * w_i for v_i, w_i in zip(v, w))
```

Введем еще несколько операций, которые нам будут полезны в дальнейшем.

```
# покомпонентное среднее
```

```
def vector_mean(vectors):
  return scalar_multiply(1/len(vectors), vector_sum(vectors))

# квадрат модуля вектора
def sum of squares(v):
  return dot(v, v)
```

Корень из этой величины даст нам длину вектора `v`. Для нее введем свою функцию:

```
from math import sqrt

# длина вектора
def magnitude(v):
  return sqrt(sum_of_squares(v))
```

Очень важно уметь находить евклидово расстояние между точками, которые определены заданными радиус-векторами – для этого создадим функции `squared_distance` и `distance`.

```
# квадрат расстояния
def squared distance(v, w):
  return sum of squares(vector add(v,  scalar multiply(-1, w)))

# расстояние
def distance(v, w):
  return math.sqrt(squared_distance(v, w))
```

Матрицы. Перейдем к матрицам. Без привлечения дополнительных модулей, в Python их можно реализовать как списки списков. Пример матрицы 2x3:

```
A = [[1,2,3],
     [4,5,6]]  # матрица из 2 строк и 3 столбцов
```

Заданная так матрица `A` имеет `len(A)` строк и `len(A[0])` столбцов. Определим функцию `shape`, которая будет выдавать форму матрицы.

```
# форма матрицы
def shape(A):
  n = len(A)
  m = len(A[0]) if A else 0
  return n, m
```

Кроме этого, далее будут использоваться функции для получения отдельных столбцов матриц, а также для формирования матриц.

получение i-го столбца матрицы

```
def get_column(A, i):
  return [a[i] for a in A]

def make_matrix(num_rows, num_cols, entry_fn):
  return [[entry fn(i, j)
           for j in range(num_cols)]
           for i in range(num_rows)]
```

Создадим, например, кососиагональную 3x3 матрицу.

```
def skew(i, j):
  return 1 if i == 2-j else 0

skew_matrix = make_matrix(3, 3, skew)
```

Упражнения

2.1.1. Создайте функцию `hprod`**, которая выдает эрмитово произведение комплексных векторов произвольной размерности. Эрмитово произведение – это сумма произведений компонент первого вектора на комплексно сопряженные компоненты второго вектора.**

2.1.2. Создайте функцию `det(M)` **для расчета определителя квадратной матрицы произвольной размерности.**

2.1.3. Создайте функцию `rank(M)` **для расчета ранга матрицы.**

2.1.4. Напишите функцию, принимающую квадратную матрицу и вектор как аргументы, а выдающую решение системы линейных алгебраических уравнений, найденное методом Гаусса. Обработайте случаи вырожденной матрицы и нулевого вектора.

2. Математическая статистика и вероятность

Случайные величины, выборки. Каждое рабочее утро я еду на работу. Туда меня может отвезти или автобус №20, или автобус №41. Никаких предпочтений у меня нет, и я уезжаю на первом, который придет, хотя проезд в первом стоит 20 рублей, а во втором – 25. Величина, значение которой является фактической суммой, уплаченной мною за проезд, есть случайная величина, принимающая значение в множестве {20,25}. По определению, *случайная величина* есть численное выражение неоднозначного исхода какого-то события, и она принимает значения в каком-то *множестве значений* этой случайной величины.

Допустим, я не знаю и не интересуюсь тем вопросом – есть ли какое-то расписание у автобусов, и вообще мне не важно, что влияет на то, какой из них подъедет, когда я выйду на остановку. Тогда появление любого из них для меня будет *случайным*. Если же я узнаю не только их расписание, но и те отклонения, которые они себе позволяют, то я смогу узнать заранее какой из них приедет в определенное время или, в крайнем случае, узнать это почти наверняка. В этом случае появление

автобуса определенного номера становится либо *детерминированным* (когда я знаю точно), либо (*статистически*) *предсказуемым* (когда я знаю, насколько вероятно то или иное значение). Как легко видеть, одно и то же событие для одного и того же наблюдателя может быть как случайным, так и статистически предсказуемым, в зависимости от *информированности* наблюдателя. В Data Science имеют дело именно со случайными величинами и поэтому считается, что наблюдатели изначально недостаточно информированы, то есть, рассматриваемые события для них либо случайны, либо статистически предсказуемы, но не детерминированы. Целью анализа и является проинформировать наблюдателя так, чтобы рассматриваемые события перешли из случайных в категорию предсказуемых, или стали предсказуемы с большими вероятностями. В анализе данных «проинформировать» означает предоставить информацию из уже зарегистрированного поведения в системе.

В рассмотренном примере, мне не обязательно знать расписания и возможные отклонения от них для перевода этих событий в разряд предсказуемых – может оказаться достаточным знаний о том, какие автобусы приходили во множестве случаев до сегодняшнего. То есть, если я хочу знать что-то о будущем, мне необходимо собрать информацию о прошлом. Эта информация о прошлом, или о зарегистрированном поведении в системе, задается математически в виде *статистической выборки*: набора значений случайной величины, заданных в виде последовательности, номера членов которой соответствуют каким-то известным мне моментам времени. Например, я устанавливаю моменты времени просто как номера дней, начиная с первого, когда я начал наблюдение. Тогда, если обозначить мою случайную величину через x, можно написать

$$x_1, \dots, x_n$$

после n дней записей. Список $[x_1, \dots, x_n]$ и есть статистическая выборка моей случайной величины

Выборка, которая охватывает наиболее информативное для данного наблюдателя множество испытаний, называется *генеральной совокупностью*. В примере с автобусами генеральная совокупность будет составлять выборку по всем моим поездкам на работу (как прошлым, так и будущим). Если удается доказать, что информация из выборки близка к данным генеральной совокупности, то выборка называется *репрезентативной*.

Статистики и распределения. *Статистками* называются численные характеристики статистических выборок. Из рассмотрения выше видна первая статистика – число точек выборки (n). Примеры

других статистик: наибольшее и наименьшее значение (для случайных величин с широким множеством значений), *среднее* значение, *дисперсия*, *среднеквадратичное отклонение* и бесконечное число так называемых *статистических моментов*. Определения этих статистик уже известны из курса теории вероятностей и математической статистики, но большинство из них приводятся далее.

Количество вхождений определенного значения случайной величины в выборку, деленное на число точек выборки, называется (относительной) *частотой* значения в выборке. Зависимость частоты значения от самого значения называется *статистическим распределением*.

Теперь перейдем к серьезным вещам. Сразу будем формулировать вводимые понятия в виде функций – мы же уже умеем читать скрипты Python. Выборка всегда будет представляться в виде списка $[x_1, \dots, x_n]$, а если нужна будет вторая случайная величина или вторая выборка, то $[y_1, \dots, y_m]$. При этом выборки не всегда являются векторами. Но мы, для простоты, будем пока рассматривать именно вектора. О том, как превратить произвольную выборку в векторную, поговорим позже.

```
# среднее значение
def mean(x):
  return sum(x) /l en(x)

# вектор отклонений от среднего
def de_mean(x):
  x_bar = mean(x)
  return [x_i - x_bar for x_i in x]
```

Функция для *медианы* распределения приведена далее. Ее можно написать короче, но так нагляднее для смысла. Для нечетной длины выборки это ближайшее к среднему значение, а для четной – среднее между двумя ближайшими к среднему значениями.

```
# медиана распределения
def median(x):
    def closest(y,av):
        return min(y, key = lambda t: abs(t-av))
    avx = sum(x)/len(x)
    cmx = closest(x,avx)
    if len(x)%2:
        return cmx
    else:
        x.remove(cmx)
        return (cmx+closest(x,avx))/2
```

Эта функция работает так, кто при четной длине выборки медианное значение из нее выбрасывается (из-за применения `remove`). Попробуйте написать код, который не будет страдать этим недостатком.

Медиану можно еще определить (это другое определение) как значение, меньше которого 50% отсчетов в выборке. Если вместо 50% взять другое значение, например, p, то получим *квантиль* – такой отсчет в выборке, меньше которого лежит доля в p отсчетов. Для $p = 0.5$ этот код можно превратить в медиану (попробуйте).

```
def quantile(x, p):
  p i = int(p * len(x))
  return sorted(x)[p_i]
```

Вариация, ковариация и корреляция. *Вариация* – это просто изменчивость данных в выборке. Она характеризуется несколькими величинами, среди которых *размах*, *дисперсия*, *стандартное* (*среднеквадратичное*) *отклонение*, *интерквантильный размах*. Определим их в виде функций.

```
# размах распределения
def data_range(x):
return max(x) - min(x)

# отклонения от среднего
def de mean(x):
return [x i - mean(x) for x i in x]

# дисперсия
def variance(x) :
    return mean(list(map(lambda t: t**2,x))) - mean(x)**2

# стандартное отклонение
def standard_deviation(x):
   return math.sqrt(variance(x))

# интерквантильный размах
def interquantile_range(x):
  return quantile(x, 0.75) - quantile(x, 0.25)
```

Последний показатель позволяет сразу исключить влияние небольшого числа *выбросов*, то есть отсчетов, портящих статистику.

Теперь определим показатели взаимного влияния двух выборок. Совместное отклонение двух переменных от своих средних измеряется *ковариацией*, которая пропорциональна скалярному произведению векторов отклонений от среднего:

```
# ковариация
def covariance(x, y):
  return dot(de_mean(x), de_mean(y)) / len(x)
```

Смысл ковариации в том, что она помогает установить зависимость между отсчетами распределений: если она близка к нулю, то никакой зависимости нет, а если она по модулю велика, то зависимость есть и знак ковариации определяет ее направление.

Считается (справедливо), что безразмерная характеристика какого-либо процесса всегда лучше характеристики с размерностью. Устраняя размерность из ковариации, получим *корреляцию* – это ковариация по отношению к стандартным отклонениям:

```
def correlation(x, y):    # корреляция
  if standard_deviation(x) > 0 and standard_deviation(y) > 0:
      return covariance(x,y)/standard deviation(x) /
             standard deviation(y)
  else:
     return 0
```

Модуль корреляции лежит между 0 (величины не коррелируют) и 1 (величины полностью коррелируют). Нужно отметить, что корреляция измеряет связь между величинами *при прочих равных условиях*: наличие дополнительного критерия в выборке может привести к изменению в статистике, основанной на корреляции – это суть так называемого *парадокса Симпсона*.

Связь, устанавливаемая корреляцией, линейна. То есть, выборки $[-1,0,1]$ и $[1,0,1]$ будут независимы по критерию корреляции. Однако между ними точно есть связь – модули соответствующих элементов равны. Но эта связь не является линейной и корреляцией не устанавливается.

Вероятность. Пусть мы знаем, как выглядит множество всех возможных значений случайной величины (в случае с моими автобусами – два значения 20 и 25, но, вообще говоря, это множество может содержать бесконечно много элементов). Множество всех подмножеств этого множества есть *вероятностное пространство*, а отношение мощности какого-то его элемента (он есть множество) к мощности исходного множества значений случайной величины, есть *вероятность*. В описании случайных величин использовать вероятность получается не всегда, так как вопрос о множестве значений часто все же остается открытым. Тем не менее, в таких простых ситуациях, как бросание кубика с шестью гранями, можно сделать естественное предположение о том, что исходов всего шесть и они равновероятны. Не все реальные системы так хороши.

Элемент вероятностного пространства называется *событием*. Выпадение в трех последовательных бросаниях 1, 2 и 3 – это событие, которое заключается именно в этом. А вероятностное пространство в этом случае будет иметь вид $[x_1, x_2, x_3]$ и, следовательно, будет состоять из 6^3 элементов. Наше событие соответствует всего одному из этих элементов и значит его вероятность равна 1/216. Представьте теперь, что вы не знаете сколько граней у кубика. Тогда установить вероятность будет гораздо сложней – потребуется все вычисления сделать с

переменным числом элементов, которое потом устремить к бесконечности. Далее события будем обозначать большими латинскими буквами (кроме P), а вероятность события будем обозначать через P.

События называются *зависимыми*, если наступление одного из них как-то влияет на наступление другого. В противном случае события называются *независимыми* и их вероятности подчиняются следующему правилу:

$$P(A,B) = P(A) \cdot P(B)$$

где $P(A,B)$ есть вероятность события, заключающегося в появлении и A, и B.

Если вероятность события $P(B)$ не равна нулю (то есть, событие возможно), то вероятность

$$P(A|B) = \frac{P(A,B)}{P(B)}$$

называется *условной вероятностью* события A при наступлении B. Очевидно, для независимых A и B, условная вероятность равна просто вероятности события A.

Определение условной вероятности симметрично по событиям A и B. Следовательно, можем записать

$$P(B|A) = \frac{P(A,B)}{P(A)}$$

а отсюда следует *формула Байеса*:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

По определению вероятности, сумма вероятностей события и отрицания этого события, равна 1. Поэтому всегда имеет место *формула полной вероятности*

$$P(B) = P(B,A) + P(B,\bar{A})$$

а значит

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|\bar{A})P(\bar{A})}$$

что составляет суть *теоремы Байеса*.

Теорема Байеса приводит к неожиданным следствиям, если рассматривать события с малыми вероятностями. Например, пусть вероятность выигрыша большой суммы (скажем, более миллиона рублей) в лотерее равна 10^{-7}. Допустим, существует человек, имеющий некоторую инсайдерскую информацию о выигрышных билетах, и эта информация о конкретном билете дает ответ «выигрыш» для выигрышного билета с вероятностью 0,99. Узнаем, насколько нам поможет в выигрыше эта инсайдерская информация.

$$P(A|B) = \frac{0{,}99 \cdot 10^{-7}}{0{,}99 \cdot 10^{-7} + 0{,}01 \cdot (1 - 10^{-7})}$$

То есть, пользуясь верной на 99% инсайдерской информацией, мы вместо 1 из 10 млн получаем возможность выигрыша в 1 из 100 тыс случаев, хотя, как могло показаться, можно было бы получить выигрыш с такой подсказкой почти наверняка. То есть, продать эту информацию человеку, не знакомому с теоремой Байеса, можно; тот же, кто знает теорему Байеса, не соблазнится на такое «заманчивое» предложение.

Случайные величины и распределения. Данное выше определение случайной величины позволяет понимать ее и как более сложный математический объект. В примере с теми автобусами можно сопоставить каждому значению случайной величины его вероятность (при этом мы по-прежнему не интересуемся «природой» появления того или иного автобуса на остановке). Например, автобус номер 20 приходит с вероятностью 0,6. Тогда для этой случайной величины получаем *распределение вероятностей* следующего вида:

Значение случайной величины	20	25
Вероятность	0,6	0,4

Теперь можно посчитать ту сумму, которую я в среднем трачу на проезд. Эта величина называется *математическим ожиданием* случайной величины и равна сумме произведений значений величины на соответствующую вероятность:

$$Mx = \sum x_i p_i$$

Указанное выше распределение является *дискретным* – в нем значения можно перечислять. В противном случае распределение называется *непрерывным*. Множество действительных чисел в интервале от 0 до 1 перечислять нельзя (в отличие от множества рациональных чисел – его элементы можно перечислять, но делать это придется бесконечное время). Для непрерывного распределения значения вероятностей определяют не для конкретного значения случайной величины, а для интервала ее значений. Чем меньше этот интервал, тем точнее будет распределение. Поэтому непрерывное распределение описывается непрерывной функцией $\rho(x)$, заданной на интервале изменения случайной величины, и принимающей значения в множестве $[0,1]$, которая в каждой точке x равна вероятности для случайной величины принять значение в интервале $(x, x + dx)$, где dx есть бесконечно малое приращение значения случайной величины. Эта функция называется *дифференциальной функцией распределения*, но чаще *плотностью вероятности* случайной величины.

Если все возможные значения непрерывной случайной величины равновероятны, то распределение называется равномерным и его плотность может быть задана следующим образом.

```
# равномерное распределение на интервале [a,b]
def rho_even(x, a, b):
  return 1 / (b-a) if x >= a and x < b else 0
```

Отсюда, в частности, легко увидеть, что

$$\int_a^b \rho(x)dx = 1$$

что справедливо для плотностей вероятностей любых распределений (при условии, что случайная величина определена на интервале $[a, b]$).

Функция $f(x)$, определенная соотношением

$$f(x) = \int_{-\infty}^{x} \rho(y)dy$$

называется (*интегральной*) *функцией распределения*. Она численно равна вероятности случайной величине принять значение, меньшее x.

Математики (в данном случае вместе с физиками) установили, что если имеется много независимых случайных величин с одинаковыми распределениями, то случайная величина, представляющая собой их среднее значение, всегда распределена с плотностью

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

где μ – ее математическое ожидание, а σ – стандартное отклонение. Это утверждение называется *центральной предельной теоремой* и очень помогает в анализе данных. Поэтому определим соответствующие функции в Python.

```
# нормальное распределение – плотность вероятности
def rho_norm(x, mu=0, s=1):
  return 1/sqrt(2*pi*s)*exp(-(x-mu)**2/2/s**2)

# нормальное распределение – функция распределения
def f_norm(x, mu=0, s=1):
  return (1+erf((x-mu)/sqrt(2)/s))/2
```

Здесь использована функция `erf` из библиотеки `math` – это так называемая *функция ошибок*. Функции `sqrt` и `exp`, а также постоянная `pi` тоже должны быть предварительно импортированы из `math`.

Для практических целей часто нужна обратная функция нормального распределения, то есть зависимость значения случайной величины от вероятности. Обратить функцию распределения можно дихотомией –

делением отрезка пополам до тех пор, пока не подберемся к нужному значению с заданной точностью. Соответствующий скрипт ниже.

```
# обратная функция нормального распределения в полосе значений
# -100…100
def inv_f_norm(p, mu, s, t=0.001): # t - точность
    # сначала перейдем к стандартному распределению
    if mu != 0 or s != 1:
        return mu + s * inv_f_norm(p,0,1,t)
    # ищем в полосе значений -100…100
    low_x, low_p = -100.0, 0
    hi_x, hi_p = 100.0, 1
    while hi_x - low_x > t:
        mid x = (low x + hi x)/2
        mid p = f norm(mid x)
        if mid_p < p:
            low_x, low_p = mid_x, mid_p
        elif mid p > p:
            hi x, hi p = mid x, mid p
        else:
            break
    return mid_x
```

Конечно, упомянутые эпизодические сведения из теории вероятностей, не претендуют на какую-нибудь полноту. Но наша цель – добраться до анализа данных, и мы уже совсем близко.

Упражнения

2.2.1. Сформируйте выборку курса биткоина к рублю, пользуясь табличными данными (https://cryptocharts.ru/bitcoin/). Найдите упомянутые в тексте выше статистики, пользуясь средствами Python и определенными функциями.

2.2.2. Рассчитайте корреляцию между изменениями курсов рубль / биткоин / евро по той же таблице.

2.2.3. Разделите таблицу из упражнения 2.2.1 на две части: 60% записей пометите в одну таблицу, а 40% в другую. В предположении, что курс рубля распределен нормально в первой выборке и константы распределения сохранятся на вторую выборку, найдите значение, выше которого курс не должен подниматься во второй выборке с вероятностью 0,95. Проверьте сбылся ли прогноз, пользуясь имеющимися значениями во второй таблице.

3. Проверка статистических гипотез

Статистическая гипотеза и ее проверка. Чаще всего, у аналитика данных сначала нет никаких данных. Есть только задача, которую надо решить. Про то, как добыть данные, мы поговорим позже. Допустим, что данные вы всё-таки откуда-то получили или нашли способ их получать прямо сейчас. И тогда возникает вопрос – что же с ними делать? В самих данных не написано ничего про распределение вероятностей – пока это просто набор чисел. Если аналитик по каким-то там своим «экспертным» причинам придумает случайной величине ее распределение вероятностей, то дальше возникнет задача проверить – прав ли аналитик в своей оценке. Назначенное аналитиком статистическое распределение или хотя бы предположение о каких-то соотношениях в нем, назовем *статистической гипотезой*. Например, при бросании монеты, утверждение о том, что она уравновешена, является статистической гипотезой. Утверждение о том, что мужчины, в среднем, тяжелее женщин – статистическая гипотеза. Утверждение, что ночью все кошки серые, не является статистической гипотезой, потому что не подразумевает другого исхода. Утверждение, что русские чаще попадают в рай, чем немцы, тоже не является статистической гипотезой, поскольку принципиально не проверяемо, то есть данные об этом отсутствуют.

Рассмотрим пример статистической гипотезы. Пусть у нас имеются некоторые ограниченные данные о тренировках биатлониста Сидорова. Сам Сидоров утверждает, что он промахивается только 1 раз из 10. Это – статистическая гипотеза. Тренеру следует её проверить чтобы понять – направить Сидорова на индивидуальную гонку или лучше всё-таки на спринт. Если биатлонист прав, то вероятность промахнуться для него всегда равна $p = 0{,}1$ и статистика его стрельб должна подчиняться нормальному распределению (из-за ключевого слова «всегда»). Параметры этого распределения легко вычислить, если заметить, что исходов может быть всего два – попал (0) и не попал (1), то есть, это схема Бернулли. Для схемы Бернулли вероятность получить m целевых исходов в серии из n попыток равна

$$P(m,n) = C_n^m p^m (1-p)^{n-m}$$

и, по определению среднего,

$$\mu = \sum_{m=0}^{n} mP(m,n) = \sum_{m=0}^{n} mC_n^m p^m (1-p)^{n-m}$$
$$= np^1(1-p)^{n-1} + 2\frac{n(n-1)}{2!}p^2(1-p)^{n-2}$$
$$+ 3\frac{n(n-1)(n-2)}{3!}p^3(1-p)^{n-3} + \cdots$$
$$+ (n-1)np^{n-1}(1-p)^1 + np^n$$
$$= np\left((1-p)^{n-1} + \frac{(n-1)}{1!}p^1(1-p)^{n-2}\right.$$
$$+ \frac{(n-1)(n-2)}{2!}p^2(1-p)^{n-3} + \cdots$$
$$\left.+ (n-1)p^{n-2}(1-p)^1 + p^{n-1}\right)$$
$$= np\sum_{m=0}^{n-1} C_{n-1}^m p^m (1-p)^{n-1-m}.$$

Вычислим момент второго порядка чтобы найти дисперсию:

$$\mu_2 = \sum_{m=0}^{n} m^2 P(m,n) =$$
$$= np^1(1-p)^{n-1} + 2^2\frac{n(n-1)}{2!}p^2(1-p)^{n-2}$$
$$+ 3^2\frac{n(n-1)(n-2)}{3!}p^3(1-p)^{n-3} + \cdots$$
$$+ (n-1)^2np^{n-1}(1-p)^1 + n^2p^n =$$
$$np\left((1-p)^{n-1} + 2\frac{(n-1)}{1!}p(1-p)^{n-2}\right.$$
$$+ 3\frac{(n-1)(n-2)}{2!}p^2(1-p)^{n-3} + \cdots$$
$$\left.+ (n-1)^2p^{n-2}(1-p)^1 + np^{n-1}\right)$$
$$= np\left(1 + \frac{(n-1)}{1!}p(1-p)^{n-2} + 2\frac{(n-1)(n-2)}{2!}p^2(1-p)^{n-3} + \cdots\right.$$
$$\left.+ (n-1)(n-2)p^{n-2}(1-p)^1 + (n-1)p^{n-1}\right) =$$

$$= np\left(1 + p(n-1)\sum_{m=0}^{n-2} C_{n-2}^{m} p^m (1-p)^{n-2-m}\right) = np\big(1 + p(n-1)\big)$$
$$= n^2p^2 + np - np^2$$

Дисперсию найдем как разность μ_2 и μ^2:

$$\sigma^2 = \mu_2 - \mu^2 = np - np^2 = npq$$

Таким образом, математическое ожидание и стандартное отклонение равны, соответственно

$$\mu = np, \sigma = \sqrt{npq} \approx 0{,}95\sqrt{n}.$$

Пусть имеется выборка из $n = 1000$ выстрелов. В случае верной гипотезы Сидорова случайная величина s, равная числу промахов, должна быть распределена нормально около среднего значения $\mu = 1000p = 100$ со стандартным отклонением $\sqrt{npq} = \sqrt{90} = 9.5$. В реальной выборке всё может оказаться не так и нам нужно быть готовыми к тому, что мы ошибемся в оценке гипотезы. Какие могут быть здесь варианты?

1. Гипотеза верна, мы ее приняли: верное положительное решение (++);
2. Гипотеза не верна, мы ее не приняли: верное отрицательное решение (--);
3. Гипотеза верна, мы ее не приняли: ложноположительное решение об отклонении (+-); это *ошибка 1-го рода*;
4. Гипотеза не верна, мы ее приняли: ложноотрицательное решение об отклонении (-+); это *ошибка 2-го рода*.

Совсем без ошибок оценить гипотезу вряд ли получится. Поэтому нам надо установить некоторые количественные критерии, которые будут характеризовать возможность ошибок. Этот критерий называется *уровнем значимости* и чаще всего выбирается равным 5% или 1%. Возьмем за основу значение уровня значимости в 5%.

Если гипотеза верна, то существует некоторый интервал значений случайной величины s, попадая в который своими промахами, Сидоров подтверждает гипотезу, а делая «неправильное» количество промахов – отклоняет, и при этом ошибка 1-го рода (отвергнуть честного Сидорова) возможна только с вероятностью 5%. Этот интервал можно посчитать с помощью функции распределения, так как она численно равна вероятности случайной величине принять значение, меньшее определенного. Таким образом, если

$$\int_{s_{low}}^{s_{high}} \rho(s)ds = 0.95$$

и наше среднее значение величины s по реальной выборке, лежит между s_{low} и s_{high}, то гипотезу можно принимать, в противном случае гипотеза должна быть отклонена. «Колокол» нормального распределения симметричен, и значит можно посчитать значения s_{low} и s_{high} по условию

$$s_{low} = \mu - \Delta s, \qquad s_{high} = \mu + \Delta s$$

и принять в нормальном распределении

$$F_{\mu,\sigma}(s_{low}) = 0.025,$$

так как «хвост» распределения площадью 2,5% слева имеет своего «близнеца» справа, что в целом даст 5%. Отсюда останется просто найти s_{low}.

Конкретно в приведенном примере, среднее число промахов должно быть в пределах от 82 до 118, чтобы обеспечить требуемый уровень значимости. Значение 82 выводится на экран командой

```
print(inv_f_norm(0.025,100,9.5))
```

а верхнее значение найдено как 100+100-82=118.

Биатлонисту Сидорову достаточно сделать от 82 до 118 промахов включительно чтобы подтвердить свою гипотезу по уровню значимости 5%, если принимать во внимание только ошибку 1-го рода.

Но есть еще возможность совершить ошибку 2-го рода, то есть, принять в команду лгуна Сидорова. Это произойдет, если реальная вероятность промаха не равна 10%, но мы приняли гипотезу. Но это условие бессмысленно, пока не установлено допустимого интервала вероятности (когда считать его лгуном). Поэтому предположим, что тренеру важно чтобы вероятность промаха не превышала 13% (для других значений можно посчитать в качестве упражнения). Если при значениях вероятности $p = 0.13$, Сидоров из 1000 выстрелов снова совершит число промахов, лежащее в интервале от 82 до 118, то будем считать, что мы допустили ошибку второго рода: реальное число промахов у Сидорова не 10% от совершенных выстрелов, а больше, чем 13%, а мы приняли его гипотезу. Величина, характеризующая возможность ошибки 2-го рода, называется *мощностью проверки* и численно равна вероятности не совершить такую ошибку. Мощность проверки можно посчитать так:

$$w = 1 - \int_{82}^{118} \rho_{\mu',\sigma'}(x)dx,$$

где $\rho_{\mu',\sigma'}(x)$ есть плотность нормального распределения с $\mu' = 130$ и $\sigma' = \sqrt{0.13 \cdot 0.87 \cdot 1000} = 10.63$. Тогда мощность проверки будет такой:

$$w_{0.13} = 1 - \int_{82}^{118} \rho_{130,10.63}\,(x)dx = 0.87$$

то есть, 87% что мы не совершаем ошибку второго рода.

Итак, если наш спортсмен из 1000 выстрелов сделал от 82 до 118 промахов включительно, то тренер может считать его гипотезу об 1 промахе из 10 верной по уровню значимости 5% с мощностью проверки 87%, если тренера устраивает уровень 13 промахов из 100.

Такие нужные *p*-значения. Существует терминология и метод проверки гипотез на основе вероятности ошибки первого рода, которая называется *p-значением*. Двустороннее p-значение для x вычисляется как удвоенный «хвост» нормального распределения от точки x в сторону «от среднего»:

```
def p_value(x, mu=0, s=1):
  if x >=mu:
     return 2*(1-f_norm(x, mu, s))
  else:
     return 2*f_norm(x, mu, s)
```

Если Сидоров промахнулся 111 раз из 1000, то:

```
p_value(111, 100, 9.5) # получаем 0.247
```

Вероятность ошибки первого рода больше 5% и гипотеза не отвергается. Легко проверить, что при вычислении с числом промахов 120 получается p-значение меньше 5% и это значит, что гипотезу надо отвергать.

Можно также вычислять одностороннее p-значение. Это просто «хвост» нормального распределения от заданной точки в сторону от среднего. Можем задать соответствующие функции, тем более что одна из них уже определена ранее.

```
def two_side_p_value(x, mu, s): # lower, upper
    return f_norm(x, mu, s), 1-f_norm(x, mu, s)
```

Доверительный интервал и подгонка *p*-значения. Если бы у нас имелось точное значение p для нашей выборки, то, в соответствии с центральной предельной теоремой, среднее значение должно быть распределено приближенно нормально с математическим ожиданием p и стандартным отклонением $\sqrt{p(p-1)/n}$

```
sqrt(p*(1-p)/1000)
```

Но значение p неизвестно, поэтому вместо него используется оценка, например, уровень, удовлетворяющий тренера, в нашем примере:

```
# тренера удовлетворяет уровень 13 промахов из 100
p_hat = 13 / 100
mu = p hat
s = m.sqrt(p_hat*(1-p_hat)/1000)  # 0.106
```

Поступая так же, как и раньше, найдем интервал, для которого параметр p является истинным с точностью до 95%:

```
p_min = inv_f_norm(0.025,p_hat,s)  # 0.1092
p_max = 2*p_hat - p_min            # 0.1508
```

То есть, если тренера устраивает уровень доверия в 95%, то вероятность промаха должна находиться в интервале от 0.1092 до 0.1508. Если бы Сидоров стрелял лучше, то тренер нашел бы ему лучшее применение. Заявленное Сидоровым значение 0.1 не попадает в этот интервал. Поэтому тренер изначально предполагает, что Сидоров завысил свой результат. Если же при испытании биатлонист покажет лучший (или худший) результат, то тренер будет тестировать его по другому уровню.

Гипотезы нужно формулировать до обращения к данным. Очистка данных от выбросов должна производиться без принятия во внимание гипотез. Иначе возникает эффект, который называют *подгонкой p-значения* (еще говорят *взлом p-значения* или *p-hacking*). Эффект заключается в том, что если у вас имеются данные и некоторый набор гипотез, то одна из гипотез определенно покажется значимой, после чего, устраняя выбросы, можно добиться нужного p-значения. Допустим, в нашем примере, тренеру важно к какому уровню ближе биатлонист – 8 промахов из 100 или 13 из 100. У биатлониста гипотеза своя – 10 из 100. Тренер оценивает биатлониста по своим двум гипотезам и видит, что биатлонист сделал 105 промахов из 1000. Так как ему больше нужен меткий биатлонист, он списывает один из промахов на влияние бокового ветра, и получает, что скорее верна гипотеза «8 из 100». Но на соревнованиях биатлонист дает 110 промахов из 1000, что никак не вяжется с 8 из 100, а скорее ближе к 13 из 100. В результате команда проиграла, хотя могла бы выиграть, если бы тренер сделал вывод о справедливости гипотезы 13 из 100. Всё потому, что гипотеза 8 из 100 для тренера была предпочтительней еще до анализа данных и проверка на нее делалась вместе с проверкой на гипотезу 13 из 100.

Функциональные зависимости, аппроксимация как статистическая гипотеза. Пусть имеется набор некоторых значений величин x_i, измеренных соответственно в моменты времени t_i. Эти величины описывают какой-то процесс, который может носить как детерминированный, так и случайный характер. Мы не знаем какого вида этот процесс. Если бы у нас имелось описание этого процесса в виде аналитических зависимостей или скорости изменения величины от самой величины, или ускорения от скорости и самой величины, то мы могли бы проинтегрировать соответствующие дифференциальные уравнения и найти вид зависимости значения величины x от времени.

После этого легко можно предсказать дальнейшее поведение величины x. Но у нас нет этих зависимостей.

Можно воспользоваться центральной предельной теоремой: предположить, что есть ***какая-то*** функция $y(t)$, которая описывает исследуемый процесс. Согласно теореме, отклонения измеренных величин от тех, что даются этой функцией, распределены нормально. То есть, случайная величина

$$\xi_i = x_i - y(t_i)$$

распределена приближенно нормально с математическим ожиданием, равным 0. Так как вид функции $y(t)$ неизвестен, его обычно формулируют в виде гипотезы, которая в данном случае и есть статистическая гипотеза, и выбирают маску функции: экспоненциальную, полиномиальную, тригонометрическую и т.д. в зависимости от того, какие ожидания имеются от этого процесса.

Если в процессе величина всегда растет или убывает со временем, то выбирают полиномиальную или экспоненциальную маску функции. Если значения величины то увеличиваются, то уменьшаются и не прослеживается никакой периодичности, то на каком-то интервале можно воспользоваться полиномиальной маской, подобрав нужную степень и коэффициенты полинома. А если все же прослеживается что-то похожее на периодичность, то имеет смысл использовать маску функции в виде суммы тригонометрических функций от времени с разными постоянными множителями при аргументе и разными коэффициентами перед слагаемыми.

Пусть, например, имеются данные о смертности людей от всех возможных причин в городе Нью-Йорк в сентябре-октябре 1918 года:

$$x_i = [11{,}13{,}20{,}30{,}50{,}60]$$

представляющие собой среднюю смертность в день, посчитанную за каждую неделю с 1 сентября по 2 ноября. Известно, что 10.5 – это нормальный уровень. Вычислим по какому закону росла смертность. В данных очевидно имеется значительное (в разы) возрастание. Поэтому возьмем в качестве маски экспоненциальную функцию $y = Ae^{bt}$ Для определения постоянных A и b воспользуемся известным алгоритмом аппроксимации, который построим прямо по определению и для экспоненциальных, и для полиномиальных функций.

Пусть имеются некоторые экспериментальные данные

$$\hat{x}_0, \dots, \hat{x}_{n-1},$$

замеренные в моменты времени $t_0, \dots, t_n$, соответственно. И пусть решено применить к этой зависимости маску функции вида

$$x = Ae^{bt}$$

Поскольку наиболее легко аппроксимация строится для линейных функций, сначала переходят к переменным, между которыми зависимость линейная:

$$y = \ln x = bt + \ln A = bt + a$$

и для экспериментальных данных

$$\hat{y}_0 = \ln \hat{x}_0, \dots, \hat{y}_{n-1} = \ln \hat{x}_{n-1}.$$

Теперь задача решается методом наименьших квадратов (МНК), который приводит к системе линейных алгебраических уравнений.

Имеем

$$F(a,b) = (y(t_0) - \hat{y}_0)^2 + \cdots + (y(t_{n-1}) - \hat{y}_{n-1})^2$$

Эта функция должна иметь минимум в искомой точке (a,b):

$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial b} = 0,$$

откуда, так как $\frac{\partial y}{\partial a} = 1, \frac{\partial y}{\partial b} = t$ найдем

$$b(t_0 + \cdots + t_{n-1}) + an = \hat{y}_0 + \cdots + \hat{y}_{n-1}$$

$$b(t_0^2 + \cdots + t_{n-1}^2) + a(t_0 + \cdots + t_{n-1}) = \hat{y}_0 t_0 + \cdots + \hat{y}_{n-1} t_{n-1}$$

Эта система 2-го порядка, решение которой можно найти по правилу Крамера. Теперь искомая зависимость, приближающая исходные данные, примет вид

$$x = e^a e^{bt},$$

с конкретными значениями параметров.

Пусть теперь маска функции (гипотеза) имеет вид полинома r-го порядка:

$$x = \sum_{k=0}^{r} a_k t^k.$$

Тогда МНК приводит к соотношениям

$$F(a,b) = (x(t_0) - \hat{x}_0)^2 + \cdots + (x(t_{n-1}) - \hat{x}_{n-1})^2,$$

$$F(a,b) = \left(\sum_{k=0}^{r} a_k t_0^k - \hat{x}_0\right)^2 + \cdots + \left(\sum_{k=0}^{r} a_k t_{n-1}^k - \hat{x}_{n-1}\right)^2,$$

$$\frac{1}{2}\frac{\partial F}{\partial a_l} = \left(\sum_{k=0}^{r} a_k t_0^k - \hat{x}_0\right) t_0^l + \cdots + \left(\sum_{k=0}^{r} a_k t_{n-1}^k - \hat{x}_{n-1}\right) t_{n-1}^l = 0,$$

и в итоге – снова к системе линейных алгебраических уравнений

$$a_0(t_0^l + \cdots + t_{n-1}^l) + \cdots + a_r(t_0^{l+r} + \cdots + t_{n-1}^{l+r}) = \hat{x}_0 t_0^l + \cdots + \hat{x}_{n-1} t_{n-1}^l,$$

$$l = 0, \dots, r$$

которая теперь имеет порядок $r+1$. Из нее определяются значения коэффициентов $a_0, \dots, a_r$.

Для описанного выше примера с испанкой в Нью-Йорке подойдет следующая функция, осуществляющая экспоненциальную аппроксимацию.

```
def approx_exp(x,t): # в списке x только положительные числа
  n = len(x)
  y = list(map(log,x))
  sum t, sum y = sum(t), sum(y)
  sum_t2 = sum(ti**2 for ti in t)
  sum_yt = sum(ti*yi for ti,yi in zip(t,y))
  a = (sum yt*sum t – sum y*sum t2)/ (sum t**2 – sum t2*n)
  b = (sum y*sum t – sum yt*n)/ (sum t**2 – sum t2*n)
  return exp(a),b # выдаются параметры A и b
```

Теперь можем построить график аппроксимирующей функции совместно с точками данных

```
x_hat = [11,13,20,30,50,60]
t = range(len(x hat))
A, b = approx exp(x hat,t)
x = list(map(lambda z: A*exp(b*z), t))
print("A=", A," b=",b)

from matplotlib import pyplot as plt
plt.plot(t, x hat,'rx', label ='Fact')
plt.plot(t, x, 'b-', label ='Approx')
plt.legend()
plt.show()
```

Получим результат, изображенный на рис. 4: налицо экспоненциальный рост и функция имеет вид

$$y = 10.02\, e^{0.37t}$$

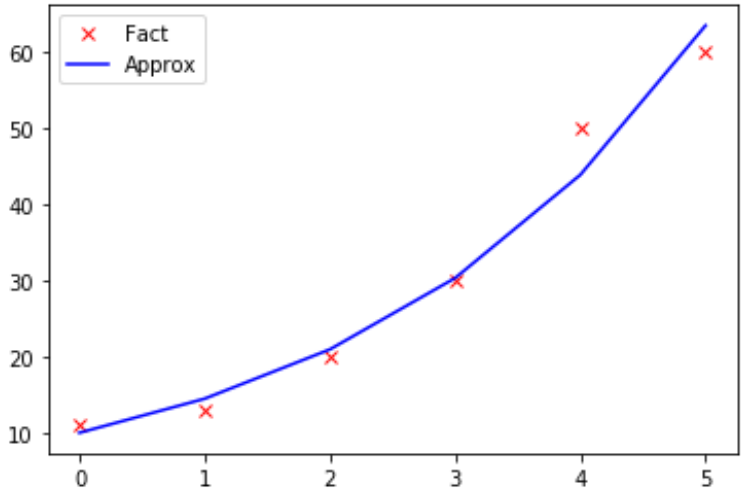


Рис. 4. «Испанка» в Нью-Йорке в сентябре-октябре 1918 года.

Величина $\xi_i = y_i - x_i$ распределена с выборочным средним и стандартным отклонением, которые, соответственно, равны

$$\mu = -0.13, \sigma = 2.99$$

Наибольшее по модулю отклонение от среднего равно -5.97, и соответствующее ему p-значение:

$$p_{max} = 0.05,$$

что говорит о том, что нулевую гипотезу можно принять.

Таким образом, вероятность того, что процесс экспоненциальный в рассматриваемый период времени, больше 95%. Это означает, что изменять нашу модель процесса имеет смысл только если новые данные приведут к тому, что экспоненциальное приближение перестанет работать с такой точностью.

Приведем также функцию для полиномиальной аппроксимации – она понадобится для выполнения упражнений и в целом полезна.

```
def approx_poly(x,t,r): # в списке x любые числа
    M = [[] for _ in range(r+1)]
    b = []
    for l in range(r+1):
        for q in range(r+1):
            M[l].append(sum(list(map(lambda z: z**(l+q), t))))
        b.append(sum(xi*ti**l for xi,ti in zip(x,t)))
    a = gauss(M,b)
    return a
```

Здесь использована функция `gauss`, решающая СЛАУ методом Гаусса. Ее можете написать сами, хотя она есть в приложенном наборе функций.

Байесовский статистический вывод. Чтобы как-то описать данные, аналитик может начинать сразу с какого-то распределения, а потом его уточнять. В этом случае принятое сначала распределение называется *априорным*, а уточненное в статистическом выводе – *апостериорным* распределением. Часто берут в качестве априорного следующее двухпараметрическое распределение

$$\rho(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)},$$

где

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

Гамма-функция Эйлера Γ уже зарезервирована в math, поэтому бета-распределение можно записать так:

```
from math import gamma

def beta_pdf(x, alpha, beta):
  B = gamma(alpha)*gamma(beta) / gamma(alpha+beta)
  if x<0 or x>1:
     return 0
  return x**(alpha-1)*(1-x)**(beta-1)/B
```

Это распределение концентрируется в точке

$$\frac{\alpha}{\alpha+\beta}$$

и чем больше максимум значений α и β, тем «плотнее» распределение – острее его вершина.

Пусть теперь имеется некоторая серия n испытаний с двумя исходами (например, условно «вверх» и «вниз»). При испытаниях считаем сколько раз выпал тот или другой исход и пусть выпало u «вверх» и d «вниз». Из теоремы Байеса можно вывести (неочевидное сразу) следствие, что апостериорное распределение опять будет бета-распределением, но с измененными параметрами. Если у априорного распределения были параметры α и β, то у апостериорного будут параметры

$$\alpha + u \text{ и } \beta + d.$$

Продолжая испытания дальше, можно пересчитывать параметры бета-распределения после каждой серии испытаний. Тогда с каждой серией мы будем всё дальше уходить от априорных параметров в сторону точных параметров бета-распределения для рассматриваемого процесса.

Упражнения

2.3.1. За январь-апрель 2022 года, согласно данным Gismeteo, средние значения температур по неделям в Белгороде были равны -1.7, -5.4, -4.0, -5.9, -1.6, 0.0, 0.6, 2.1, 0.1, -4.9, -3.5, 5.9, 8.5, 9.9, 13.3, 11.1, 14.4. Проверьте гипотезу о том, что этот рост описывается экспоненциальной функцией. Уровень значимости 5%. Сравните с гипотезой о полиномиальной функции 4-го порядка. Рассчитайте мощность проверки для лучшей функции по сравнению с худшей.

2.3.2. Отделите последние четыре значения от выборки в упражнении 2.3.1. К полученной выборке примените полиномиальную аппроксимацию 7 порядка. Посчитайте p-значение. Затем посчитайте то же самое на той же функции, но с полной выборкой. Почему p-значение резко уменьшилось?

2.3.3. Проводится эксперимент по рассеиванию частиц на двух отверстиях, расположенных на одинаковом расстоянии 1 по разные стороны от начала отсчета. В данных указаны координаты зарегистрированных попаданий рассеянных частиц на экран: [-0.3, -1.2, 3.0, 0.4, 0.9, 1.1, -1.8, -1.3, -0.7, 0.2, 1.4, 2.1, -2.2, 0.8, 0.5, 1.6, -1.5, -0.4, -2.0, 1.2] Найдите апостериорное распределение вероятностей рассеяния частиц, считая каждую пятерку отсчетов за пакет (всего 4 пакета и 4 шага к апостериорному распределению).

2.3.4*. Напишите функцию, которая принимает выборку отсчетов и времени их регистрации в виде двух списков, а выдает вид подходящей аппроксимирующей функции (exp или poly) и ее параметры.

4. Оптимизация градиентным спуском

Понятие градиентного спуска. Рассмотрим некоторую функцию F, которая принимает вещественный вектор $x_1, \dots, x_n$, а возвращает вещественное число. В физике и математическом анализе такая функция называется *скалярным полем*. Попробуем в общем виде найти экстремум этой функции. Из математического анализа известно, что для таких функций вектор их частных производных $\frac{\partial F}{\partial x_i}$, называемый *градиентом*, имеет направление самого большого возрастания функции.

Отсюда следует один из подходов к максимизации функции, то есть к нахождению ее максимума: следует взять какое-то значение аргумента $x = [x_1, \dots, x_n]$, вычислить градиент функции при таком аргументе и сделать небольшой шаг в направлении этого градиента, далее повторить итерацию. Если нужно минимизировать функцию, то шаги делают в направлении, противоположном градиенту. Этот алгоритм и называется *градиентным спуском*.

Если у функции F имеется всего один глобальный экстремум, то градиентный спуск (или подъем) его найдет. Но если у функции много локальных экстремумов (что очень часто встречается), то найти из них какой-то определенный таким способом может быть непросто.

Тем не менее, градиентный спуск – общепринятый метод, часто дающий необходимое решение в самых разнообразных задачах. Приведем функцию `gradient`, которая вычислит нам градиент заданной функции $f(x)$ при заданном значении ее аргумента с заданной точностью h.

```
def gradient(f, x, h=1e-05):
  grad = []
  for i,  in enumerate(x):
    xh = [x j+(h if j == i else 0) for j, x j in enumerate(x)]
      grad.append((f(xh)-f(x))/h)
  return grad
```

Вычисление градиента, например, для функции $F = x \cdot x$ в точке (1,0,0) даёт:

```
def f(x):
    return sum(x_i**2 for x_i in x)

print(gradient(f,[1,0,0]))
# [2.00001000001393, 1.000000082740371e-05,
1.000000082740371e-05]
```

что говорит о том, что для поиска минимума этой функции, нужно от точки x сдвинуться в сторону уменьшения первой координаты. Если так продолжать дальше, то придем в начало координат, которое и является в данном случае минимумом.

Использование градиентного спуска. Осуществим поиск минимума этой функции из указанной точки по алгоритму градиентного спуска. Для этого будем делать шаги итераций против направления градиента до тех пор, пока функция почти не претерпит изменений при шаге, то есть модуль ее градиента будет меньше, чем заданный шаг (похоже на равную нулю производную). Этот критерий можно сделать строже, но этим займемся позже.

```
x = [1,0,0]
stepsize = 1e-04
for s in range(int(2/stepsize)):
```

```
    G = m.sqrt(sum(g_i**2 for g_i in gradient(f,x))
    x = [x_i - stepsize*dr/G for x_i, dr in zip(x,gradient(f,x))]
    if abs(G) < stepsize:
        print(x)
        stepsize = 0
        break
if stepsize != 0:
    print("no success")
```

Конечно, вызывает вопрос установка количества петель цикла и почему использован цикл `for`, а не `while`, который здесь кажется уместнее? Ответ кроется в том, что для произвольной функции подобрать критерий остановки итераций можно только в случае, когда известны точные выражения для частных производных: в противном случае приращение будет всегда давать такую погрешность, которую невозможно оценить (ну, или эта оценка должна быть сделана отдельным модулем). Поэтому мы просто «спускаемся» на заданное число шагов и, если за время этого спуска найдем решение, то останавливаемся, а иначе – выдаем «безуспешно». Критерием найденного решения в данном случае является то, что модуль градиента стал меньше длины шага, хотя, возможны и другие варианты. Запуск этого скрипта дает нам

```
[-9.999997519874415e-05,
-5.0004749975025705e-06,
-5.0004749975025705e-06]
```

то есть, число, близкое к 0. Как известно, минимум у функции `F` как раз в точке 0. Если бы вместо `int(2/stepsize)` было установлено `int(1/stepsize)`, то в результате вышло бы `no success`. Потому что за это число шагов достичь критерия не удастся.

На практике чаще всего длину шага подбирают по ходу спуска, так как ясно, что если градиент большой, то экстремум не рядом. Но и универсальной зависимости тут нет: например, для функций $g = -e^{-x \cdot x}$ и $f = x \cdot x$ величины градиентов при одних и тех же шагах будут, конечно, сильно различны, и экстремум для первой функции наступает «внезапно» по сравнению со второй.

Функция, реализующая градиентный спуск, имеется в приложенной библиотеке и называется `gradient_descent`. Она имеет следующий вид:

```
def gradient_descent(f, x0, mu=0.05, h=1e-05, s=5000):
  x = copy(x0)
  for si in range(s):
      gf = gradient(f,x,h)
      mgf = scalar_multiply(mu*sqrt(sum(g**2 for g in gf)),gf)
      x = vector_add(x,  scalar_multiply(-1, mgf))
  return x
```

Стохастический градиентный спуск. Часто бывает так, что требуется «спуститься быстрее»: скорость обычного градиентного спуска (который еще называют *пакетным*) недостаточна. Тогда прибегают к алгоритму, который по своей идее похож на градиентный спуск, но в нем предусмотрена экономия на суммировании. Вот этот алгоритм:

- выбрать начальный вектор параметров a и скорость обучения μ;
- повторять до достижения приблизительного минимума:
  - случайным образом перемешать векторы в обучающей выборке;
  - сделать шаг в сторону градиента, вычисленного в первой попавшейся точке (определенной случайным вектором):

$$a \to a - \mu \cdot \operatorname{grad} F\,(x^j),$$

где x^j есть j-й вектор из выборки. Например, пусть

$$F(a_1, \dots, a_n) = \sum_{p=1}^{n} (f(x^p, a) - y^p)^2,$$

где y^p – это значение, соответствующее вектору x^p, а $f(x^p, a)$ – функция, моделирующая поведение y от x. Тогда

$$\operatorname{grad} F\,(x^j) = \frac{\partial F}{\partial x_i^j} = 2(f(x^j, a) - y^j)\frac{\partial f}{\partial x_i}(x^j, a)$$

безо всякого суммирования, в отличие от пакетного спуска, где суммирование приходится делать по всем векторам выборки. Добавим еще одну опцию в алгоритм: будем уменьшать длину шага на 10%, если не удается найти меньшее значение за данный шаг, и ограничим общее число таких шагов: будем считать, что минимум найден, если за это число шагов не удалось найти меньшее значение. Получим следующую функцию.

```
# Стохастический градиентный спуск
def minimize_stochastic(f, x, y, a_0, h_0 = 0.1, max_steps = 1000):
  a = a_0
  h = h_0
  min_a, min_F = None, float('inf')
  drunken_steps = 0
  while drunken_steps < max_steps:
    value = sum((f(xx,a)-yy)**2 for xx,yy in zip(x,y))
    if value < min_F:
      min_a, min_F = a, value
      drunken_steps = 0
      h = h_0
    else:
      drunken_steps += 1
       h *= 0.9
       n = random.randint(0,len(x)-1)
       grad = []
       for i,_ in enumerate(a):
         ah = a.copy()
```

```
            ah[i] += h
            # главное - в этом шаге (суммы нет!)
            grad.append(((f(x[n],ah)-y[n])**2-(f(x[n],a)-y[n])**2)/h)
        a = [a[i] - h*grad[i] for i,  in enumerate(a)]
    return min_a, value
```

Стохастическая максимизация, разумеется, тоже существует:

```
def maximize_stochastic(f, x, y, a_0, h_0=0.01):
  def negate(f):
    return lambda *args, **kwargs: -f(*args, **kwargs)
  return minimize_stochastic(negate(f), x, y, a_0, h_0)
```

В результате получим список a значений параметров функции f, которая теперь аппроксимирует нужную нам зависимость y от x.

Упражнения

2.4.1. Найдите минимум функции $f(x,y) = \sin x \cos y$, ближайший к точке (1,1)**.**

2.4.2. Методом стохастического градиентного спуска найдите параметры a функции $f(x,a) = \frac{1}{1+e^{-a\cdot x}}$ для заданных векторов x, их значений y и начального значения a_0 вектора параметров:

```
x = [[0,1,2,3],
     [-1,-2,-3,-4],
     [-12,3,4,5],
     [4,2,3,1],
     [6,-1,-2,5]]
y = [0.1,0.85,0.05,0.08,0.76]
a0 = [1,1,-1,1]
```

Затем напишите функцию, которая будет подбирать лучшую стартовую точку a_0.

Глава III. Сбор и обработка данных

1. Извлечение данных из веб-ресурсов

Получение информации из всемирной паутины. В первую очередь, следует понимать, что Python – это всего лишь язык программирования. Он не может «самостоятельно» извлекать информацию откуда бы то ни было. Ввиду того, что наиболее полезные вещи в Python делаются модулями, логично предположить, что должны быть модули и для извлечения данных, а именно, для обращения к

файлам и URL за данными. С точки зрения современных операционных систем, имеющих сетевой интерфейс, обращение к URL происходит легко – для этого имеются соответствующие системные утилиты. Например, в Linux и MacOS таковой утилитой является curl. Если ввести в командной строке

```
# curl https://www.bstu.ru
```

то содержимое заглавной страницы сайта БГТУ им. В.Г. Шухова вывалится прямо вам в терминал.

Язык программирования в состоянии упростить задачу запроса данных и обогатить возможности работы с ними. С его помощью можно не просто вытащить веб-страницу на просмотр ее кода, но и сохранить ее и/или сделать что-то с ней или информацией в ней, то есть произвести ее *парсинг*.

В частности, в Python имеется специальный модуль для создания и передачи http-запросов. Он называется `requests`. Это не единственный модуль, который умеет запрашивать информацию у гипертекстовых документов, но мы рассмотрим здесь его, так как на его примере сравнительно легко понять работу остальных.

Средствами языка можно организовать сбор информации с разных страниц – это уже будет *скрейпинг*. Для этого используется синтаксический анализ, но не самих данных, которые нужны будут далее, а их адресов и связей между адресами.

Для того чтобы понять, что и как запрашивать из сети, сначала нужно рассмотреть механизмы создания «кучи» информации и дальнейшего ее разбора. Примером, предоставляющим такую возможность, является библиотека BeautifulSoup.

Библиотека BeautifulSoup. Она уже входит в состав дистрибутива Anaconda. Чтобы воспользоваться ей, достаточно включить в скрипт строчку

```
from bs4 import BeautifulSoup
```

Для работы BeautifulSoup принимает документы XML или HTML. Это средство для синтаксического разбора и создания в памяти структур данных, соответствующих документу. То есть, BeautifulSoup, принимая строку с информацией, всяческим образом пытается привести ее в вид «хорошего», с точки зрения машины, документа. Например, коряво написанный HTML будет воспринят примерно так:

```
from bs4 import BeautifulSoup

html_string = "<html><p>Para 1<p>Para 2<blockquote>Quote
1<blockquote>Quote 2"
soup = BeautifulSoup(html_string)
print soup.prettify()
```

В результате будет выведено:

```
<html>
   <p>
      Para 1
   </p>
   <p>
      Para 2
      <blockquote>
         Quote 1
            <blockquote>
               Quote 2
            </blockquote>
      </blockquote>
   </p>
</html>
```

То есть, тексту придан читаемый формат и очевидные ошибки, вроде незакрытых тегов, исправлены.

Для работы с XML годится модуль библиотеки `bs4`, который называется `BeautifulStoneSoup`. Выглядит это так:

```
from bs4 import BeautifulStoneSoup

xml_string = "<doc><tag1>Contents 1<tag2>Contents
2<tag1>Contents 3"
soup = BeautifulStoneSoup(xml_string)
print soup.prettify()
```

Результат:

```
<doc>
   <tag1>
      Contents 1
        <tag2>
          Contents 2
        </tag2>
   </tag1>
   <tag1>
      Contents 3
   </tag1>
</doc>
```

В XML набор самозакрывающихся тегов заранее неизвестен. Поэтому для корректной работы их нужно описать с помощью аргумента `selfClosingTags`:

```
soup = BeautifulStoneSoup(xml_string,
                          selfClosingTags=['selfclosing'])
```

если в строке есть самозакрывающиеся теги `selfclosing`.

Следует упомянуть, что BeautifulSoup перекодирует входящую строку в Unicode. Кодировки проверяются в следующем порядке:

1. Кодировка, переданная конструктору супа в параметре `fromEncoding`.

2. Кодировка, обнаруженная в самом документе: например, в декларации XML или (для документов HTML) в атрибуте http-equiv тега META. Как только BeautifulSoup обнаружит подобное указание о кодировке документа, она повторно приступит к синтаксическому разбору документа с самого начала, но уже применяя найденную кодировку. Избежать этого можно только явным заданием кодировки, тогда любая найденная в документе кодировка будет игнорироваться.
3. Кодировка, вычисленная по нескольким первым байтам файла. Если кодировка определяется на этом этапе, то она будет либо UTF-*, либо EBCDIC, либо ASCII.
4. Кодировка, вычисленная библиотекой `chardet`, в случае если она была установлена.
5. UTF-8
6. Windows-1252

Для документов без декларации или в неизвестной кодировке не получится сделать каких-либо предположений. В таком случае – скорее всего, ошибочно, – будет использоваться кодировка Windows-1252.

Если обрабатывать с помощью BeautifulSoup документ в кодировке Windows-1252, то BeautifulSoup обнаружит и уничтожит изящные кавычки и другие символы, специфичные для Windows. Чтобы этого избежать, можно передать параметр `smartQuotesTo=None` в конструктор супа: тогда кавычки будут конвертироваться в Unicode, как и другие символы. Для изменения поведения BeautifulSoup и BeautifulStoneSoup можно передать в параметре `smartQuotesTo` значения "xml" или "html": `smartQuotesTo="html"`.

Упомянутый выше метод `prettify` добавляет значимые переводы строки и пробелы для придания структуре документа лучшей читабельности, а также удаляет текстовые узлы, состоящие только из пробелов, а это может изменить смысл XML документа. Функция `str` не удаляет такие узлы и не добавляет пробелы между узлами. Приведем пример.

```
from bs4 import BeautifulSoup

doc = "<html><h1>Heading</h1><p>Text"
soup = BeautifulSoup(doc)

# '<html><h1>Heading</h1><p>Text</p></html>'
str(soup)

# b'<html><h1>Heading</h1><p>Text</p></html>'
soup.renderContents()
```

```
# '<html><h1>Heading</h1><p>Text</p></html>'
soup.__str__()

# '<html>\n <h1>\n  Heading\n </h1>\n <p>\n  Text\n
</p>\n</html>'
soup.prettify()

print soup.prettify()
```

Будет напечатано:

```
<html>
  <body>
    <h1>
      Heading
    </h1>
    <p>
      Text
    </p>
  </body>
</html>
```

Для изъятия из супа содержимого определенного тега, можно использовать метод с названием тега:

```
heading = soup.h1
str(heading) # '<h1>Heading</h1>'
```

Функция `str` выдает всё, а функция `renderContents` – только содержимое внутри тега:

```
heading.renderContents() # 'Heading'
```

Хотя, в Anaconda выдача может быть в байтовом формате. Тогда вместо `renderContents()` используем `renderContents().decode ('UTF-8')`.

Будем считать, что мы разобрались с загрузкой документа. Теперь займемся собственно синтаксическим разбором: структурами данных BeautifulSoup, которые создаются по мере синтаксического разбора документа. Экземпляр класса BeautifulSoup или BeautifulStoneSoup (та самая «куча» информации) обладает большой глубиной вложенности связанных структур данных, соответствующих структуре документа XML или HTML. Он состоит из объектов двух других типов: объектов Tag, которые соответствуют тегам, к примеру, тегу `<title>` и тегу `<b>`; и объекты NavigableString, соответствующие таким строкам как `"Page title"` или `"This is paragraph"`.

Класс NavigableString имеет несколько подклассов (CData, Comment, Declaration и ProcessingInstruction), которые соответствуют специальным конструкциям в XML. Они работают также как

NavigableString, за исключением того, что, когда приходит время выводить их на экран, они содержат некоторые дополнительные данные. Для разъяснения рассмотрим такой скрипт:

```
from bs4 import BeautifulSoup

doc = ['<html><head><title>Page title</title></head>',
'<body><p id="firstpara" align="center">This is paragraph
<b>one</b>.','<p id="secondpara" align="blah">This is
paragraph <b>two</b>.','</html>']

soup = BeautifulSoup(''.join(doc))

print soup.prettify()
```

Результат:

```
<html>
  <head>
    <title>
    Page title
    </title>
  </head>
  <body>
    <p id="firstpara" align="center">
    This is paragraph
      <b>
      one
      </b>
    .
    </p>
    <p id="secondpara" align="blah">
    This is paragraph
      <b>
      two
      </b>
    .
    </p>
  </body>
</html>
```

Разберем этот случай. Теги имеют атрибуты: например, каждый тег `<p>` в приведенном выше примере HTML имеет атрибуты `"id"` и `"align"`. К атрибутам тегов можно обращаться таким же образом, как если бы объект `Tag` был словарем:

```
firstPTag, secondPTag = soup.findAll('p')
firstPTag['id']     # u'firstPara'
```

```
secondPTag['id']    # u'secondPara'
```

Все объекты Tag содержат элементы, перечисленные ниже (значение элемента может равняться None), а объекты NavigableString имеют все из них за исключением contents и string:

- parent: родительский тег, объект парсера или None;
- contents: является списком объектов Tag и NavigableString, содержащихся в элементе страницы (page element), его содержат только объект парсера и теги;
- string: сделан для удобства, когда в элементе всего один объект; в примере выше soup.b.string отобразит Unicode-строку "one", которая содержится в первом объекте с тегом <b> дерева синтаксического разбора;
- nextSibling, previousSibling: позволяют пропускать следующий или предыдущий элемент на этом же уровне дерева. В примере элемент nextSibling тега <head> равен тегу <body>, поскольку он является следующим вложенным элементом; в свою очередь, его элемент nextSibling равен None, поскольку вложенных по отношению к объекту <html> элементов больше нет;
- next, previous: позволяют передвигаться по элементам в том порядке, в котором они были обработаны парсером, а не в порядке появления в дереве; элемент next для объекта <head> равен объекту <title>, а не объекту <body>;
- имена тегов.

Над элементом contents объекта Tag (тега) можно производить итерации, рассматривая объект в качестве списка. Это полезное упрощение. Подобным образом можно узнать, сколько дочерних узлов имеет объект Tag, вызвав функцию len(tag) вместо len(tag.contents).

Теперь займемся поиском в дереве разбора. Для определения критериев отбора объектов BeautifulSoup есть несколько способов. Сначала посмотрим на наиболее общий из всех методов поиска – findAll. Он начинает с заданной точки и ищет все объекты Tag и NavigableString, соответствующие заданным критериям. Сигнатура метода findAll следующая:

```
findAll(name=None, attrs={}, recursive=True, text=None,
        limit=None, **kwargs)
```

Аргумент name ограничивает набор имен тегов. Имеется несколько способов ограничить имена и все они используются повсюду в

BeautifulSoup API. Самый простой способ – передать имя тега. Вот так можно найти все теги <b> в документе:

```
soup.findAll('b')     # [<b>one</b>, <b>two</b>]
```

Можно передать регулярное выражение. Код, который ищет все теги, имена которых начинаются на английскую букву "b":

```
import re

tagsStartingWithB = soup.findAll(re.compile('^b'))
[tag.name for tag in tagsStartingWithB]
# [u'body', u'b', u'b']
```

Можно передать список или словарь. Найдем, например, <title> и <p> через передачу списка:

```
soup.findAll(['title', 'p'])
# [<title>Page title</title>,
#  <p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
```

А теперь через передачу словаря (что работает быстрее)

```
soup.findAll({'title' : True, 'p' : True})
# [<title>Page title</title>,
#  <p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
```

Передается также специальное значение `True`, которому соответствуют все теги с любыми именами:

```
allTags = soup.findAll(True)
[tag.name for tag in allTags]
[u'html', u'head', u'title', u'body', u'p', u'b', u'p', u'b']
```

Наконец, можно передать вызываемый объект, который принимает `Tag` как единственный аргумент и возвращает логическое значение. Каждый `Tag`, который находит `findAll`, будет передан в этот объект и если его вызов возвращает `True`, то необходимый тег найден. Вот так можно найти теги с двумя и только двумя атрибутами:

```
soup.findAll(lambda tag: len(tag.attrs) == 2)
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
```

Аргументы ключевых слов (`kwargs`) налагают ограничения на атрибуты тега. Вот простой пример поиска всех тегов, атрибут `"align"` которых имеет значение `"center"`:

```
soup.findAll(align="center")
```

```
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>]
```

Значению `True` соответствует тег, заданный атрибут которого имеет любое значение, а `None` соответствует тег, у которого заданный атрибут не содержит значения:

```
soup.findAll(align=True)
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
[tag.name for tag in soup.findAll(align=None)]
# [u'html', u'head', u'title', u'body', u'b', u'b']
```

Специальный аргумент с именем `attrs` используется в ситуациях, когда имя тега совпадает с зарезервированным словом. Аргумент `attrs` есть словарь, который работает также как именованные аргументы. Сравните:

```
soup.findAll(id=re.compile("para$"))
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
soup.findAll(attrs={'id' : re.compile("para$")})
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
```

Теперь, на примере метода `findAll`, можно описать остальные методы.

Метод `find` почти в точности совпадает с `findAll`, за исключением того, что он ищет первое вхождение искомого объекта, а не все.

Методы `findNextSabling` и `findNext` можно просто проиллюстрировать. Для следующего супа

```
from bs4 import BeautifulSoup
soup = BeautifulSoup('''<ul>
 <li>An unrelated list</ul><h1>Heading</h1><p>This is <b>the
list you want</b>:</p> <ul><li>The data you want</ul>''')
```

пусть стоит задача вытащить элемент списка, стоящий первым после тега `<h1>`. Это можно сделать так:

```
soup.h1.findNextSibling('ul').li
# <li>The data you want</li>
```

Если же ввести такой код:

```
soup.find(text='Heading').findNext('ul').li
# <li>The data you want</li>
```

то будет произведен поиск тега <li> (с его содержимым), стоящим сразу после того тега <ul>, который идет после текста Heading.

Если вам понадобятся другие методы библиотеки bs4, то вы всегда их сможете найти в документации на русском языке по адресу: http://wiki.python.su/Документации/BeautifulSoup#A.2BBBEESwRBBEIEQARLBDk_.2BBEEEQgQwBEAEQg-

Библиотека requests. Теперь мы более-менее знаем, что делать с добытой из www «абракадаброй». Осталось только выяснить как ее добыть. В этом нам могут помочь http-запросы, которые можно делать прямо из Python с помощью средств библиотеки requests. Вызываем ее как обычно:

```
import requests
```

Пожалуй, основным http-методом для проведения исследования в интернет, является метод get. Он указывает на то, что вы хотите извлечь данные из определенного ресурса. Для того, чтобы выполнить запрос get, используется requests.get(). Если сделать вот такой запрос:

```
requests.get('https://api.github.com')
```

То в ответ «прилетит»

```
<Response [200]>
```

что буквально означает «запрос удался, всё отлично». Правда, если у вас что-то не так со связью, то вместо такого ответа будет выдана ошибка. Если же связь есть, но нет запрошенного URL, то будет выдано 404 вместо 200.

Чтобы не стучать в закрытые двери (или в отсутствующие), полезно знать способ обработки неудачных запросов в requests:

```
import requests
from requests.exceptions import HTTPError
for url in ['https://api.github.com',
'https://api.github.com/invalid']:
  try:
     response = requests.get(url)
        # если ответ успешен, исключения задействованы не будут
     response.raise_for_status()
  except HTTPError as http_err:
     print(f'HTTP error occurred: {http_err}')
  except Exception as err:
     print(f'Other error occurred: {err}')
  else:
     print('Success!')
```

Вытянем теперь информацию с веб-страницы get-запросом. Для этого мы должны знать ее точный адрес. Обычно адреса стареют – либо

информация удаляется или перемещается, либо сам адрес становится другим. Например, в 23:40 21 августа 2020 года существовал такой URL:

```
import requests
response =
requests.get('https://ratings.bstu.ru/direction/000000003/0000
00001/000000001/09.03.01/09.03.01%20Информатика%20и20%вычислит
ельная%20техника')
```

В 2023 году в ответ на этот запрос приходит 400, что означает, что ничего хорошего мы не получим. Интересующая нас информация теперь находится по другому адресу и нужно делать такой запрос

```
import requests
response =
requests.get('https://ratings.bstu.ru/rating/bakalavr/byudzhet
nye-mesta/ochnaya/09-03-01-informatika-i-vychislitelnaya-
tehnika')
# <Response [200]>
```

И что дальше? Как работать с результатом? Конечно, мы должны знать что именно мы ищем и чего хотим. Например, приведенный запрос идет к странице с таблицей рейтинга абитуриентов перед зачислением в БГТУ им. В.Г. Шухова, на направление 09.03.01 – информатика и вычислительная техника. Во время приемной кампании рейтинг может меняться ежедневно и поэтому нам интересны не столько данные с нее, сколько их динамика – мы будем делать регулярные запросы. А хотим мы, например, узнать каковы на текущий момент значения числа заявлений, согласий на зачисление и каков средний балл тех, кто оформил согласие, а также как менялись эти величины. На этом примере и покажем работу `requests`.

Ответ на наш запрос, содержащийся сейчас в переменной `response`, называется *пейлоад* (*payload*) и может быть конвертирован в различные форматы:

- `response.content` – даст байтовое представление;
- `response.text` – даст текстовое UTF-8 представление;
- `response.encoding = 'windows-1252'` перед `response.text` – даст текстовое windows-1252 представление.

Кроме того, не заглядывая в сам пейлоад, можно выгрузить заголовки, введя

```
response.headers
```

(иногда это нужно, но в нашем примере – нет). А еще, если вы видите, что пейлоад похож визуально на словарь, что часто бывает при обращении к API, то это говорит о том, что вам «подсунули» *объект JSON* (JavaScript Object Notation), визуально похожий на словарь. Чтобы

превратить его в настоящий словарь, используется незамысловатая конструкция:

```
response.json()
```

называющая вещи своим именем. Бывает, что люди даже не понимают, чем отличается JSON от словаря Python. Объяснение простое – тем же, чем слово «апельсин», написанное где и как угодно, отличается от слова «апельсин», написанного на реальном апельсине. JSON не предполагает размещения в памяти компьютера как структуры данных – это просто текст. А словарь – это именно структура данных, так как задать его можно вообще без фигурных скобок и двоеточий. Например, так:

```
my_crazy_dict = dict(zip('alien',range(5)))
# {'a': 0, 'l': 1, 'i': 2, 'e': 3, 'n': 4}
```

что вовсе не обязательно выводить на экран. Но всё это опять не к нашему примеру, хотя точно полезно.

Здесь разберем только ответ на второй вопрос примера. Остальные – пусть будут вашим развлечением. Итак, нужно узнать сколько сейчас есть согласий на зачисление. Для этого вспомним суп – он сейчас как раз нужен. После уже выше проведенной подготовки вводим

```
from bs4 import BeautifulSoup as BeSo
soup = BeSo(response.text)
P = soup.find(text='По общему
конкурсу').findNext('div').findAll(attrs={'class':
'agreement'})
counter = 0
for p in P:
    Yes = p.renderContents().decode('UTF-8')
    if Yes == 'Да':
        counter+=1
print('Имеется ',counter,' согласий')
# Имеется 45 согласий
```

Если вы разобрали как работает этот код – поздравляю, вы начинаете свою карьеру скрейпера, правда, пока специализируетесь на парсинге. Чтобы перейти именно к скрейпингу, нужно научиться делать то же самое, не заглядывая в структуру веб-страницы. Ведь при подготовке кода, который вы видите выше, нам пришлось посмотреть в код веб-страницы и выяснить, что текст «По общему конкурсу» находится в теге, который содержит всю интересующую нас таблицу данных. Затем мы увидели, что сведения о согласии на зачисление находятся в теге, у которого атрибут `class` имеет значение `agreement`. Собрали все такие теги (с помощью `findAll`), а потом взяли их содержимое и посчитали. Но это было бы невозможно, не заглянув в веб-страницу. Ну, или почти невозможно. Нет, возможно, только труднее. И в этом и состоит переход к скрейпингу.

Упражнения

3.1.1. Выберите с сайта bstu.ru всех преподавателей дисциплин, имеющих отношение к анализу данных. Перечень таких дисциплин разрешается задать вручную. Поиск производить в текущем расписании и в списке сотрудников. Результат вывести в виде словаря: ключ – название дисциплины, значение – список преподавателей.

3.1.2*. Потренируйтесь в скрейпинге. Напишите краулер – поисковый робот, который найдет и выдаст в виде списка строк все полные предложения со словами «данные, анализ, Белгородская» (с учетом словообразования), содержащиеся на веб-страницах, выдаваемых на первых двух страницах поиска Яндекса.

2. Работа с данными из файлов с помощью pandas

Задача представления и обработки данных. Чаще всего, информация или данные, с которыми вы хотите работать, уже существуют в каком-то виде. Если это табличные данные, то для них общепринятым средством работы является MS Excel. Это ПО является, пожалуй, наиболее распространенным в России из-за своей простоты и надежности – не требуется какой-либо специальной подготовки чтобы создать и заполнить таблицу в Excel, а файлы с расширением .xls и .xlsx читаются почти везде. Минусом является то, что обработка данных в Excel также проста. То есть, сложные обработки и анализ больших данных этим средством сделать довольно трудно.

Где-то в 2008 году Уэс Маккини озадачился вопросом – как сделать возможным многомерный и многоступенчатый анализ данных, приняв данные из Excel? Затем вопрос продолжился тезисом «и вернуть обратно в Excel». Вместе со своим коллегой Чан Шэ, он разработал библиотеку pandas и структуры данных DataFrame и Series, которые являются весьма удобными для хранения данных, а также обеспечивают работу в рамках среды Python не только для сбора и очистки данных, но для задач анализа и моделирования данных, без переключения на более специфичные языки, вроде R.

В настоящем пособии предполагается показать использование средств pandas для обработки данных, которые могут быть собраны в результате парсинга или скрейпинга из различных источников и

нуждаются в дальнейшей очистке. Очистка производится фильтрацией данных по определенным критериям – устраняются выбросы, некачественные данные, а также данные, не соответствующие цели исследования.

Структуры данных и методы pandas. В pandas имеется две основных структуры данных: Series и DataFrame. Для начала импортируем библиотеку

```
import pandas as pd
```

и теперь можем создать структуру данных типа Series:

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

Она похожа на обычный список, но, разумеется, таковым не является. Хотя бы потому, что если попросить выдать `s`, то получим

```
s
# 0     1.0
# 1     3.0
# 2     5.0
# 3     NaN
# 4     6.0
# 5     8.0
# dtype:   float64
```

Можно заменить заданный по умолчанию целочисленный индекс на любой другой. Например, на символы ‘d’, ‘b’, ‘a’, ‘c’. Для этого в конструкции Series надо применить параметр `index`:

```
s = pd.Series([1,2,3,4], index = ['d','b','a','c'])
s
# d     1
# b     2
# a     3
# c     4
# dtype:   int64
```

В Series можно назначить имя серии с помощью параметра `name`, можно «насильно» задать тип данных параметром `dtype`, а также многое чего еще – если интересно, смотрите документацию pandas: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html (на английском).

Представим такую ситуацию: надо создать серию, все значения которой равны друг другу. В качестве данных указываем это значение и создаем ровно такое количество индексов, сколько необходимо равных значений. Получим это:

```
s = pd.Series(10, index=range(3))
s
# 0     10
# 1     10
# 2     10
```

```
# dtype:   int64
```

Так мы генерируем серию из 3 значений, каждое из которых равняется 10. Серию также можно создавать из словаря:

```
d = {"Белгород": 400, "Москва": 12000, "Харьков": 1400}
pd.Series(d)
# Белгород    400
# Москва      12000
# Харьков     1400
# dtype:   int64
```

Видим, что ключи словаря вдруг стали индексами серии, а значения словаря – значениями серии. Вот еще пара полезных атрибутов, нужных для работы с сериями:

```
d.index
# Index(['Белгород','Москва','Харьков'],dtype='object')
d.dtype
# dtype('int64')
```

Сложение серии с числом приводит к увеличению на это число всех элементов серии. То же касается умножения на число. Разумеется, если тип данных позволяет.

При сложении серий складываются элементы с одинаковыми индексами (а не в соответствии с порядком их следования!). Для примера рассмотрим ситуацию с частично пересекающимся набором индексов у серии:

```
jan = pd.Series([10, 20, 30], index=['Иванов', 'Петров',
'Сидоров'])
feb = pd.Series([56, 10, 44], index=['Петров', 'Сидоров',
'Васечкин'])
jan
# Иванов      10
# Петров      20
# Сидоров  30
# dtype: int64
```

Не уточняя что произошло с Ивановым, зададимся вопросом – что будет, если сложить серии за январь и февраль? Увидим, что появился `NaN`:

```
jan + feb
# Васечкин NaN
# Иванов      NaN
# Петров      76.0
# Сидоров  40.0
# dtype: float64
```

но здесь мы его видеть совсем не хотели! Мы же хотели увидеть 10 у Иванова и 44 у Васечкина. Машина не может догадаться, что делать с Ивановым и Васечкиным – она просто будет пытаться сложить значения, у которых одинаковые индексы. А если нет значения с этим

индексом, то напишет NaN (Not a Number) и успокоится. Чтобы решить данный вопрос, можно вызвать метод add с атрибутом fill_value, значение которого и будет передаваться в сумму, если индекс найти не удалось:

```
pd.Series.add(jan, feb, fill_value = 10)
# Васечкин 54.0
# Иванов      20.0
# Петров      76.0
# Сидоров  40.0
# dtype: float64
```

Вместо add может быть вычитание sub. Кроме того, довольно любопытно работают сравнения и проверка расположения между заданных значений:

```
s1 = pd.Series([4, -1, 2, -3, 0])
s1 > 0
# 0     True
# 1     False
# 2     True
# 3     False
# 4     False
# dtype: bool
s1.between(-1, 3)
# 0     False
# 1     True
# 2     True
# 3     False
# 4     True
# dtype: bool
```

Проверка нахождения значения в серии isin работает так же, как и between.

Логические операции вызываются как &, |, ~, соответственно, “и”, “или”, “не”.

Доступ к элементам серии, удовлетворяющим определенным критериям, делается методом loc:

```
s1 = pd.Series([1, 4, 2, -5], index = list('abcd'))
s1.loc['a']
# 1
s1.loc[['b','d']]
# b     4
# d     -5
# dtype: int64
```

Метод iloc делает то же самое, но принимает не индекс, а номер позиции в серии.

Сравнения еще могут работать так:

```
s1[s1<0]
# d     -5
```

```
# dtype: int64
```

Это отличается от предыдущего тем, что выдаются только удовлетворяющие критерию значения.

Можно использовать срезы – точно так же, как в списках.

Если серии похожи на списки, то структура данных DataFrame – на двумерный массив. Это табличная структура, имеющая индексы и заголовки. Пример:

```
data = {'cities': ['Белгород', 'Воронеж', 'Москва', 'Курск'],
'years': [2003, 2010, 2014, 2000],
'k':[4, 2, 1, 3]
}
frame = pd.DataFrame(data)
frame
```

Out[1]:

	cities	years	k
0	Белгород	2003	4
1	Воронеж	2010	2
2	Москва	2014	1
3	Курск	2000	3

В этом примере DataFrame создан из словаря и это один из наиболее распространенных способов его создания. Каждый столбец DataFrame представляет собой Series и, следовательно, DataFrame можно сделать из Series, использовав их вместо списков.

Если нужно изменить тип данных, то надо воспользоваться соответствующим методом. Например, мы будем пользоваться численными данными, значит нам нужен метод `to_numeric`, который может работать непосредственно на один столбец

```
pd.to_numeric(frame['years'])
```

а может быть распространен на несколько столбцов

```
frame[["years", "k"]].apply(pd.to_numeric)
```

Теперь про фильтрацию. С точки зрения обработки данных, это одна из самых важных операций. На самом деле, всё просто: вы вырезаете из DataFrame нужный столбец и задаете условия на него в виде логического выражения:

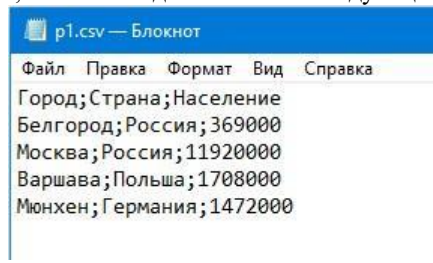
```
frame[(frame['years']>2005)&(frame['years']<2015)&(frame['k']>
1)&(frame['k']<4)]
```

в результате получите фрейм с единственной строчкой, содержащей Воронеж.

Преобразование данных в/из структуры pandas. Собственно, теперь имеет смысл перейти к использованию pandas, так как базовый экскурс в синтаксис и основные методы закончен. В частности, мы уже немножко умеем складывать, умножать и нарезать. Давайте научимся читать и писать.

Начнем с чтения из файла формата csv. Для чтения нужно вызвать функцию `read_csv`. Она имеет 49 параметров (их можно найти в документации к pandas, но хотелось бы взглянуть на того, кому они пригодятся все).

Предположим, что необходимо считать следующий файл:



Для чтения важно следующее:

- Местоположение файла. Файл имеет имя `p1.csv`, и лежит там же, где скрипт; это не всегда так и об этом стоит помнить.
- Символ-разделитель – в данном случае точка с запятой.
- Заголовок (название столбцов). Он в данном случае имеется (Город; Страна; Население). Но попробуйте распознать его наличие, не заглядывая в файл! Последнее важно для скрейпинга. Решение этой задачи, на самом деле, на поверхности – элементы заголовка не содержатся в множестве элементов данных и часто имеют другой тип данных.
- Кодировку лучше указать, если она известна, а если неизвестна – выяснить (как?! – творческое задание).

В данном случае читаем так:

```
df = pd.read_csv(
              filepath or buffer = 'p1.csv',
              sep = ';',
              encoding = 'cp1251'
              )
```

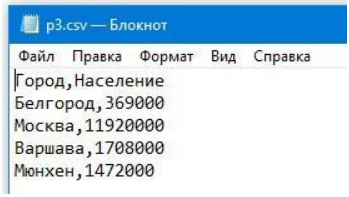
Результатом будет очевидный фрейм. Читать теперь умеем.

Писать надо с помощью метода `to_csv`. У него параметров еще больше, чем у прошлого метода, и искать их стоит там же. Возьмем имеющийся фрейм и сохраним его в файл `p3.csv`:

```
df.to_csv(
        path or buf  = 'p3.csv',
        sep = ',',
        header = True,
        index = False,
        columns = ['Город','Население'],
        encoding = 'cp1251'
```

```
        )
```

Причем данные о стране мы посчитали лишними и не стали оставлять эту колонку в файле, задав `columns=['Город','Население']` (хотя, конечно, могли бы и оставить, не задавая эту команду). Файл теперь выглядит так:



Теперь разберемся с форматом `.xlsx`. Читать из него можно следующим образом

```
dfsv = pd.read_excel('…/Audit_data.xlsx', sheet_name='Список студентов', header=None)
```

В принципе, основные атрибуты в этом примере есть – это название считываемого листа (`sheet_name`) и наличие или отсутствие заголовка (`header`). Другие атрибуты можно, как обычно, изучить в документации. Вместо точек, конечно, размещается путь к вашему файлу.

Запись в MS Excel делается с помощью специального конструктора `xlsxwriter`, который нужно установить, если у вас его нет. Это не составляет большой сложности и останавливаться на этом не будем. Когда установлен, запись производится так:

```
writer = pd.ExcelWriter(…/p3.xlsx', engine='xlsxwriter')
drl.to_excel(writer, 'p3')
writer.save()
```

где вместо … нужно поставить желаемый путь к файлу.

Можно было бы написать еще много слов о конвертации фреймов в `csv` и `xlsx` и обратно. Но нет ничего лучше, чем сделать это самостоятельно и увидеть всё, что можно написать на паре страниц текста. Таким образом точно всё запомнится.

Упражнения

3.2.1. Скачайте с ресурса kaggle.com датасет cwurData.csv, содержащий мировые рейтинги университетов. Создайте из этих данных DataFrame. Выберите в отдельный DataFrame все университеты Японии, Израиля, Германии, Франции и России. Выведите круговую диаграмму числа студентов в этих

университетах для каждой страны. Рядом постройте диаграмму рассеяния в осях «влияние - качество образования» (influence – quality of education), размер маркера пропорционален числу студентов, цвет – соответствует стране.

3.2.2. Написанный в упражнении 3.1.2 краулер примените для поиска полных предложений со словами и сочетаниями «most efficient, computation», и отдельно «cybersecurity, personal data», содержащиеся на страницах, выдаваемых на первых десяти страницах поиска Google. Создайте две структуры Series с этими предложениями: A и B. Создайте DataFrame со столбцами «A\B», «B\A», «A&B».

3. Исследование данных и управление ими

Этот параграф нуждается в небольшом вступлении. Дело в том, что дальше речь пойдет, в основном, о численных данных. То есть, о векторах. Если же ваши данные только похожи на численные, то есть, имеют вот такой вид: ‘12345’ и тому подобное, а уж тем более, ’12,34’ с намеком на то, что запятая разделяет целую и дробную части, то вам потребуется провести некоторую работу. Если в первом случае вам поможет банальный `int`, то во втором случае `float` вас не спасет, так как он ждет точку в качестве разделителя. И отдельного подхода требует ситуация, когда у вас в одном столбце встречаются и похожие на целые числа, и похожие на десятичные дроби, и пустые строки, и иногда NaN. Тут уж не обойтись без *предобработки* – это процедура, которая превращает ваши сырые данные в то, что может быть передано анализатору (вашей программе). Систематически о предобработке мы в этом курсе говорить почти не будем – только раздел Очистка и форматирование ниже, а она достойна большего – но упомянем лишь, что в тех примерах, которые приведены в данном пособии, предобработка или уже сделана, или может быть проведена с минимальными усилиями, на интуитивном уровне.

Исследование одномерных данных. Первым приемом, позволяющим упорядочить и визуализировать полученный ряд чисел – *одномерные данные* – является разбиение на дискретные интервалы (*бакеты*) с подсчетом точек, попадающих в каждый интервал. Это можно сделать с помощью математического округления.

```
def make_histogram(points, bucket_size):
  return Counter(bucket_size * math.floor(point / bucket_size)
for point in points)
```

После этого можно строить гистограмму

```
def plot_histogram(points, bucket_size, title = ''):
  histogram = make_histogram(points, bucket_size)
  plt.bar(histogram.keys(), histogram.values(), width =
bucket size)
  plt.title(title)
  plt.show()
```

Такие гистограммы помогают визуализировать одномерные данные чтобы понять различие выборок, у которых близкие статистики, но разное распределение.

Двумерные данные. Пусть имеется набор данных с парными значениями, например, следующий

```
def random_normal():
  return inv f norm(random.random(), 0, 1)

xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random normal() / 2 for x in xs]
ys2 = [ -x + random_normal() / 2 for x in xs]
```

При построении гистограмм на выборках `ys1` и `ys2` получатся очень похожие картины, так как распределение одно и то же. Однако, эти выборки значительно отличаются при рассмотрении совместного распределения по `xs`, что можно выяснить, построив диаграмму рассеяния.

```
plt.scatter(xs, ys1, marker = '.', color = 'black', label =
'ys1')
plt.scatter(xs, ys2, marker = '.', color = 'grey', label =
'ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title('Очень разное совместное распределение')
plt.show()
```

Результат изображен на рис. 5

Рис. 5. Совместное распределение двух нормально распределенных наборов данных.

Если вычислить корреляции между `xs` и `ys1`, а затем `xs` и `ys2`, то можно заметить, что они противоположны по знаку.

Многомерные данные. Непосредственно визуализировать их затруднительно. Поэтому анализируют соотношения между парными выборками. Выполняется анализ корреляционной матрицы, в которой в строке i и столбце j стоит корреляция между i-й и j-й выборкой.

```
def matrix_entry(i, j):
  return correlation(get_column(data, i), get_column(data, j))
def correlation_matrix(data):
  _, num_columns = shape(data)
  return make_matrix(num_columns, num_columns, matrix_entry)
```

Здесь считается, что функции `get_column` и `make_matrix` определены (это было сделано при рассмотрении линейной алгебры).

Если размерность данных не очень большая, то можно построить диаграммы рассеяния, подобные той, что на рис. 1, для всех пар разных i,j. Такой набор диаграмм называется *точечной матрицей*.

Очистка и форматирование. Чаще всего, полученные данные требуют предварительной обработки – содержат выбросы, разные типы данных и т.д. Лучше всего преобразовывать данные прямо во время парсинга. Например, если производится парсинг из `csv` файла не средствами pandas, а просто объектом `csv.reader`, то нужно создать функцию-обертку для этого читающего объекта. В нее передается список анализаторов, каждый из которых задает способ преобразования столбца, а `None` значит, что с этим столбцом ничего делать не надо.

```
def parse_row(input_row, parsers):
```

```
  return [parser(value) if parser is not None else value for
value, parser in zip(input_row, parsers)]
def parse rows with(reader, parsers):
  for row in reader:
     yield parse_row(row, parsers)
```

Если данные плохие, то всегда лучше получить `None`, чем ошибку в обработке. Следующая функция помогает разобрать данные или вернуть `None`.

```
def try_or_none(f):
  def f_or_none(x):
     try: return f(x)
     except: return None
  return f_or_none
```

После чего функция анализа строки переписывается так:

```
def parse_row(input_row, parsers):
  return [try_or_none(parser)(value) if parser is not None else
value for value, parser in zip(input_row, parsers)]
```

Далее останется только придумать что делать с плохими строками, которые теперь стали строками с `None`.

Аналогичные функции-помощники можно сделать для объекта `csv.DictReader`, работающего со словарями.

```
def try_parse_field(field_name, value, parser_dict):
  parser = parser_dict.get(field_name)
  if parser is not None:
     return try_or_none(parser)(value)
  else:
     return value
def parse_dict(input_dict, parser_dict):
return { field name: try parse field(field name, value,
parser_dict) for field_name, value in input_dict.iteritems() }
```

Далее следует проверка данных на выбросы. Это, скорее, творческая задача и ее решение будет индивидуальным в каждом конкретном случае. Чтобы выработать у себя навык работы над такими задачами, нужно просто практиковаться на различных наборах реальных данных.

Управление данными. Это общий подход к работе над данными. Проиллюстрируем его на примерах. Пусть имеются словари, содержащие курсы акций:

```
data = [ {'closing_price': 102.06, 'date':
datetime.datetime(2019, 8, 29, 0, 0), 'symbol': 'AAPL'} #,...
  ]
```

Представим, что это строки в электронной таблице. Предположим, нужно узнать самую высокую цену при закрытии торгов для акций Apple Inc. (AAPL). Шаги для решения будут такими:

1. Выбрать строки с AAPL
2. Извлечь цену закрытия closing_prise

3. Взять максимум max.

Все три шага можно «зашить» в один генератор последовательности:

```
max_aapl_price = max(row[“closing_price”] for row in data if
row[“symbol”] == “AAPL”)
```

Уже ясно, что в общем случае решения таких задач, надо выбирать определенного типа значение из словаря, а для этого нужны функции для получения поля из словаря и для извлечения этого поля из набора словарей:

```
def picker(field_name):
  return lambda row: row[field_name]
def pluck(field_name, rows):
  return map(picker(field_name), rows)
```

Создадим теперь функцию-группировщик строк по результату некоторой группирующей фукции `grouper`, применяя для каждой группы один из преобразователей значений `value_transform`:

```
def group_by(grouper, rows, value_transform = None):
  # ключ – результат вычисления grouper, значение – список
строк
  grouped = defaultdict(list)
  for row in rows:
     grouped[grouper(row)].append(row)
  if value_transform is None:
     return grouped
  else:
     return { key: value_transform(rows) for key, rows in
grouped.items() }
```

Теперь предыдущий пример переписывается в более универсальном виде:

```
mix_price_by_symbol = group_by(picker(“symbol”), data, lambda
rows: max(pluck(“closing_price”, rows)))
```

Это позволяет обрабатывать более сложные запросы.

Шкалирование. Довольно часто методы обработки данных чувствительны к шкале данных. Например, дан массив данных о росте и весе большой группы людей и нужно определить *кластеры* размеров тела. Под кластерами будем понимать наборы точек, расположенных рядом друг с другом. Слово «рядом» подразумевает использование расстояния, которое у нас определялось функцией `distance`. Это было расстояние в пространстве. В данном случае у нас пространство двух измерений – рост и вес. Предположим, что часть людей – из стран с дюймовыми единицами измерения длины и данные включают значения роста и в сантиметрах, и в дюймах. Рассмотрим часть этих данных (дюймы, сантиметры, килограммы).

```
list_of_h_w = [[63, 160, 68], [67, 170.2, 72.7], [70, 177.8,
78.7]]
```

```
list_length = len(list_of_h_w)
closest_inch = []
closest_cm = []
for i in range(list length):
    d_inch = float('inf')
    d_cm = float('inf')
    for j in range(list length):
        if j == i:
            continue
        d_i = distance(list_of_h_w[i][::2],
list_of_h_w[j][::2])
        d_c = distance(list_of_h_w[i][1:], list_of_h_w[j][1:])
        if d_i < d_inch:
            d_inch = d_i
            j_inch = j
        if d_c < d_cm:
            d_cm = d_c
            j_cm = j
    closest inch.append(j inch)
    closest_cm.append(j_cm)
print(closest_inch, closest_cm)          # [1, 0, 1] [1, 2, 1]
```

Это говорит нам о том, что из-за нелинейности меры distance возникает разница в определении ближайшего соседа при использовании разных единиц измерения длины: если измерять рост в дюймах, то второй элемент списка `list_of_h_w` будет иметь первого в качестве ближайшего соседа, а если в сантиметрах, то третьего. Чтобы избавиться от этого эффекта, произведем *шкалирование* данных: приведение их к безразмерному виду с центром распределения в нуле и стандартным отклонением 1.

```
# пишем функцию шкалирования данных в матрице matrix
def scale(matrix):
    rows, cols = shape(matrix)
    means = [mean(get column(matrix,i)) for i in range(cols)]
    stdevs = [mean(get column(matrix,i)) for i in range(cols)]
    def res(i,j):
        if stdevs[j] > 0:
            return (matrix[i][j] - means[j])/stdevs[j]
        else:
            return matrix[i][j]
    return make_matrix(rows,cols,res)
```

Теперь, запустив предыдущий скрипт во второй строке которого разместим `list_of_h_w = scale(list_of_h_w)`, получим

```
[1, 0, 1] [1, 0, 1]
```

то есть, результат теперь не зависит от единиц измерения.

Снижение размерности. Бывает так, что имеющиеся многомерные данные в действительности зависят от меньшего числа измерений, чем представлено. Так, например, для данных, выраженных черными

точками на рис. 5, очевидно, что они могут быть представлены в виде выборки, зависящей только от одной величины, которая не является ни `xs`, ни `ys`. Конечно, подобного рода задачи не решают, когда речь идет о размерности 2. Такая постановка вопроса имеет смысл только для задач с большой размерностью данных. Для них применяется метод факторного анализа, который называется *методом главных компонент* (*principal component analysis, PCA*).

В методе PCA данные сначала центрируются – в каждом столбце из элемента вычитается среднее по столбцу. Для вектора эта функция уже была сделана ранее, а для матрицы она имеет следующий вид:

```
def de_mean_matrix(A):
  r, c = shape(A)
  columns_means = [mean(get_column(A,i)) for i in range(c)]
  return make_matrix(r, c, lambda i, j: A[i][j] -
column_means[j])
```

Если в пространстве центрированных данных имеется направление, заданное единичным вектором $\boldsymbol{d}$, то каждый вектор $\boldsymbol{x}$, представляющий собой строку матрицы X, имеет проекцию $(\boldsymbol{a} \cdot \boldsymbol{d})$ на это направление. Это скалярное произведение и для него уже имеется функция `dot`. Направление может задать любой ненулевой вектор $\boldsymbol{w}$ – для этого его достаточно разделить на модуль (нормировать на 1), что делается следующей функцией

```
def direction(w):
  return [w_i/magnitude(w) for w_i in w]
```

При заданном векторе $\boldsymbol{w}$, можно вычислить дисперсию набора данных в направлении $\boldsymbol{w}$:

```
def dir_variance(X,w):
  return sum(dot(x_i, direction(w))**2 for x_i in X)
```

Далее, методом градиентного спуска максимизируем эту функцию среди всех направлений. Можно это сделать также методом стохастического градиентного спуска. Найденное направление $\boldsymbol{d}_1$ показывает геометрическую ось, вдоль которой варьируют данные и называется (*первой*) *главной компонентой*.

Дальнейшие действия зависят от того, что планирует аналитик. В нашем случае нам нужно снизить размерность данных, например, до m (в начале было n признаков и $n > m$). Тогда мы делаем так:

- находим главную компоненту;
- поворачиваем координатные оси в пространстве данных так, чтобы первая координатная ось совпала с найденной главной компонентой;
- записываем первые координаты всех векторов данных в новые данные (очередной координатой);

- отбрасываем первую координату в старых (только что повернутых) данных;
- повторяем эти шаги m раз.

В результате получатся m-мерные данные, сохранившие наибольшее возможное разнообразие.

Указанный алгоритм реализован в следующих функциях:

```
# Вычисление главной компоненты в данных data
def the_first_priciple_comp(data):
  n = len(data[0])
  init w = [random.random() for   in range(n)]
  w = gradient_descent(negate(partial(dir_variance,data)),
init_w)
  return direction(w)

# поворот данных так, что ось главной компоненты x становится
# первой координатной осью
def turn(data,x):
    T = []
    T.append(x)
    n = len(x)
    for i in range(1,n):
        denom = sqrt(1-sum(xi**2 for xi in x[i:]))
        T.append([])
        for j,xj in enumerate(x[:i-1]):
            T[-1].append(xj*x[i]/denom)
        T[-1].append(-denom)
    new_data = []
    for d in data:
        new_data.append(matrix_mul(T,d))
    return new_data

# снижение размерности данных методом главных компонент
# m - конечная размерность данных
def principal_components(data,m):
    res_d = [[] for _ in data]
    n = len(data[0])
    components = []
    projections = []
    for _ in range(m):
        components.append(the_first_priciple_comp(data))
        data = turn(data,components[-1])
        for j,dj in enumerate(data):
            res_d[j].append(dj[0])
        data = [di[1:] for di in data]
    return res_d
```

Пример снижения размерности дан ниже.

```
d = [[0,1,2,3,4,5],
    [-1,0,1,2,3,4],
```

```
    [3,3,3,3,3,3],
    [2,-2,-1,2,1,0],
    [1,-1,1,-1,1,1],
    [12,-1,0,0,1,0]]

principal_components(d,3)

# [[4.739518773128492, -1.8574312660574308, -
# 0.26769149833407935],
# [3.4066997420214724, -1.6215133909545965,
# 0.019428400868823192],
# [3.9984570933210586, -0.707753625308503, -
# 0.8613596976087065],
# [2.238834622608688, 2.586945880222264, -2.6495993629051413],
# [1.081863687769344, 0.26417459807706717,
# 0.10553454258944725],
# [5.033498181202137, 7.886674850078563, -5.976469716318785]]
```

Требуется соотносить снижение размерности с информативностью оставшейся после этого выборки – может случиться так, что после снижения размерности в ваших данных больше не будет нужной вам информации.

Упражнения

3.3.1. Найдите данные о среднегодовой температуре на планете за как можно более длительный период. Найдите также данные о динамике ущерба от лесных пожаров и наводнений. На общем временном интервале этих данных постройте диаграмму рассеяния и исследуйте – имеется ли корреляция между этими данными.

3.3.2. Проведите шкалирование и снижение размерности в датасете cwurData.csv из упражнения 3.2.1.

4. Машинное обучение

Модели и машинное обучение. Для описания взаимного поведения различных величин недостаточно просто сказать зависят они друг от друга или нет. Почти всегда требуется понимание того, как влияет заданное изменение одних величин на другие. Например, простейшая *бизнес-модель* может выдавать величину ежемесячной прибыли при известном количестве проданных единиц и известной марже на единицу: модель будет заключаться в том, что прибыль будет рассчитываться как произведение числа проданных единиц в месяц на

маржу на одной единице. Эта модель основана на простом и, главное, определенном математическом соотношении.

Однако, правила модели могут быть и более сложными. Например, модель может быть вероятностной и тогда математические соотношения будут для вероятностей, а измеряемые величины в модели будут случайными величинами. В общем случае *модель* – это некоторый свод количественных правил, выраженных формулами и имеющих отношение к измеряемым параметрам системы. Модели применяются для описания функционирования систем, а также для составления алгоритмов управления в системах.

Если имеется модель, то есть возможность составить алгоритм, а значит, произвести действия в системе, преследующие определенную цель. Достижение цели – это управленческая концепция и, как известно, добиваться цели можно по-разному. В общем случае считается, что субъект, добившийся цели эффективнее (эффективность может измеряться по-разному, это устанавливается в цели), лучше *обучен* достижению этой цели, а лучше сказать таких целей.

В начале имеются лишь данные и машина (вычислитель). Никакой модели пока нет. Не так давно (всего 2-3 десятка лет назад) прерогатива составления моделей на определенных данных принадлежала только людям. В наши дни люди уже не могут похвалиться тем, что они способны создать модель на любых выборках данных. Поэтому требуется чтобы модели могли быть составлены машинами. А поскольку составление модели есть главный шаг к достижению цели и после него всё уже можно алгоритмизировать, то процесс составления модели и ее применения на заданном наборе данных в исполнении машины назвали *машинным обучением*.

Типичным случаем машинного обучения являются модели для *предсказания* поведения систем в тех или иных ситуациях. То есть, вычисления характеристик системы в будущем, имея ее характеристики в прошлом и настоящем. Или, имея новое событие, вычислить его последствия (из списка или сформировать). Например, такими предсказаниями являются:

- решение о том, является ли сообщение спамом;
- решение о том, является ли транзакция мошеннической;
- на какие кликбайты из набора будет нажимать пользователь;
- какая команда выиграет хакатон.

Если имеется выборка с уже заданными правильными ответами (обучающая выборка), то такое машинное обучение называется обучением *с учителем*. Если правильные ответы не заданы, то обучение

ведется *без учителя*. Имеются и другие виды обучения, но они должны рассматриваться в специализированных курсах.

Алгоритмы машинного обучения обычно все-таки исходят из каких-либо семейств моделей, из которых уже выбирают ту, которая соответствует выборке. Например, семейство линейных моделей, когда известно, что функция, связывающая параметры модели, линейна, но неизвестно какие в ней коэффициенты. Есть полиномиальные, экспоненциальные модели и т.д. Вы встречались с ними в статистических гипотезах. С этой точки зрения говорят, что обучается именно модель.

Переобучение и недообучение. В теории машинного обучения известны две крайности, которые представляют собой определенного рода «ловушки» в создании моделей. Попав в них, машина не получит хорошую модель и останется на той её форме, которая не соответствует выборке.

Переобучение (*overfitting*) представляет собой ситуацию, когда машина не воспринимает новые данные: всё прекрасно работает на данных обучающей выборки, но на новых данных модель не дает правильных прогнозов. Это вызвано тем, что в обучающей выборке имелся *шум* в данных и модель обучилась ему, либо выборка тенденциозна и в ней различимы только конкретные входящие значения, на которых предсказание не составляется.

Другая сторона этой же ловушки – *недообучение* (*underfitting*), при котором прогнозная модель не работает даже на обучающей выборке. В этом случае считают, что выбранная модель недостаточно хороша и продолжают поиск.

Пожалуй, яркой иллюстрацией этой ловушки является полиномиальная аппроксимация функции: если закономерность на самом деле хорошо описывается линейной функцией, то полином нулевой степени (константа) не будет описывать даже обучающую выборку, а полином, например, 10-й степени будет прекрасно описывать всю обучающую выборку, но слева и справа от нее (и даже внутри, но между отсчетами) его ветви уйдут далеко от прямой и ошибка модели вне этой выборки будет весьма большой.

Слишком сложные модели приводят к переобучению. Чтобы не переобучить модель, имеющуюся выборку делят на обучающую и тестовую. Обычно, примерно 2/3 данных идут в обучающую выборку и остаток – в тестовую, на которой производят испытание обученной модели. Для произвольной доли `prob` данных, отправляемых в обучающую выборку, разделение может быть сделано так:

```
def split_data(data, prob):
```

```
  results = [], []
  for row in data:
     results[0 if random.random() < prob else 1].append(row)
  return results
```

После этого можно разделить на обучающую и тестовую выборки такие типичные данные, как матрица X из входящих переменных и вектор y исходящих переменных.

```
def train_test_split(x, y, test_pct):
  data = zip(x,y)
  train, test = split_data(data, 1 - test_pct)
  x_train, y_train = zip(*train)
  x_test, y_test = zip(*test)
  return x_train, x_test, y_train, y_test
```

Правильность предсказаний модели. В машинном обучении модель должна стремиться к повышению *точности* (*precision*) и *полноты* (*recall*) предсказаний и не ориентироваться на их *правильность* (*accuracy*). Разницу требуется пояснить. В параграфе, посвященном проверке статистических гипотез, уже было описано сопоставление истинности гипотез и выводов моделей об этом. Это иллюстрируется таблицей сопряженности:

Предположение о том, кто на фото	«Козёл»	«Не козёл»
Предсказано «козёл»	Истинноположительная	Ложноположительная
Предсказано «не козёл»	Ложноотрицательная	Истинноотрицательная

Пусть теперь имеются фото, которые были сделаны во множестве фермерских хозяйств. И часть из этих фермеров разводят кроликов. Доля кролиководов в выборке известна: всего 350 фото, 51 сделано у тех, кто разводит кроликов. На 15 фото изображены козлы. Имеем следующую таблицу сопряженности для этого примера:

	На фото козёл	На фото нет козла	Всего
Разводит кроликов	1	50	51
Не разводит кроликов	14	285	299
Всего	15	335	350

Пусть *правильность* модели определяется тем, что предсказания не противоречат реальности. Тогда модель, «Разводящие кроликов имеют козлов» выдает ошеломляющий результат: 286/350, то есть, около 82% по критерию правильности.

Однако, если вместо этого ввести критерий *точности*, который заключается в доле истинноположительных предсказаний среди положительных предсказаний, то модель покажет 1/51, то есть 2%. А если добавить критерий *полноты*, который соответствует доле положительных предсказаний, идентифицированных моделью, то показатель будет 1/15 = 7%. Ясно, что ни то, ни другое не является хорошим показателем для модели, что и подтверждает абсурдность соотнесения кроликов с козлами.

Полноту и точность можно объединить в один критерий, который называется *метрикой F1*. Для начала зададим полноту и точность в виде функций:

```
def precision(true_pos, false_pos, false_neg, true_neg):
  return true_pos/(true_pos + false_pos)
def recall(true pos, false pos, false neg, true neg):
  return true_pos/(true_pos + false_neg)
```

А теперь запишем функцию для метрики F1

```
# метрика F1 на значениях true_pos, false_pos, false_neg,
true_neg
def f1 score(true pos, false pos, false neg, true neg):
  p = precision(true pos, false pos, false neg, true neg)
  r = recall(true pos, false pos, false neg, true neg)
  return 2*p*r/(p+r)
```

Метрика F1 есть гармоническое среднее точности и полноты и всегда лежит между ними.

Смещение и дисперсия. Если модель делает много ошибок на практически любой обучающей выборке (как функция-константа при моделировании полиномиальной зависимости), то говорят, что она имеет высокое *смещение*. Если же любые две обучающие выборки дают примерно похожие модели, то это соответствует низкой *дисперсии* модели. Высокое смещение и низкая дисперсия, как правило, соответствуют недообучению. Если же в примере с полиномом выбрать, скажем 9-ю степень (когда реальная зависимость 1-го порядка), то такая модель будет иметь очень низкое смещение – практически не будет ошибок на обучающей выборке – зато у нее будет большая дисперсия, так как на другой обучающей выборке получатся совсем другие десять коэффициентов полинома модели. Такая ситуация (низкое смещение и высокая дисперсия) соответствует переобучению.

Эти два показателя помогают сбалансировать обучение моделей. Если модель имеет высокое смещение, то следует добавить больше признаков. На примере полинома это значит, что нужно перейти от одного параметра к двум (там сразу станет всё хорошо). Если модель имеет высокую дисперсию, то признак надо удалить или добыть больше данных (тут не всё так однозначно).

Следует только помнить, что дополнительные данные не улучшают смещение: если модель не содержит достаточно признаков для объяснения закономерностей, то привлечение дополнительных данных не улучшит ее.

Извлечение и отбор признаков. В моделях признаки, упомянутые ранее, обычно уже заданы. Но не всегда они заданы однозначно. Их определение составляет суть требований к подготовке аналитика данных в предметной области.

Например, рассмотрим задачу построения спам-фильтра. Модель не может просто так, на ровном месте, начать работать с сообщениями, которые есть просто текст. Нужны какие-то признаки. Например, такие:

- Есть ли в тексте сообщения слово «выигрыш»?
- Сколько раз в тексте встречается символ «?»
- Каков домен отправителя?

Признаки назначает аналитик данных. Сейчас главное, что указанные три признака различны принципиально – по типу данных, кодирующих признак: для первого это `boolean`, для второго – `int`, для третьего – строка, но содержащаяся в определенном конечном множестве (категорийные данные).

Оказывается, тип данных, кодирующих признак, накладывает ограничения на вид используемой модели. Так, в случае `boolean`, для указанной задачи подойдет наивный байесовский классификатор. А в случае `int` уже понадобится какая-нибудь регрессионная модель (регрессии будут рассмотрены далее). Если же признак представляет собой категорийные данные или наборы чисел, то потребуется что-то посложнее – деревья принятия решений или нейронная сеть (обо всём этом позже).

Иногда признаки надо не создавать, а удалять. Что-то подобное мы уже делали, когда снижали размерность данных. Есть и другие методы сокращения числа признаков, но все они не заменят того, что называется *опытом* и *знаниями* предметной области. Поэтому аналитики данных – это, в том числе, предметные специалисты, а не только программисты и математики. Ключевое – в одном человеке должно быть конъюнктивное сочетание всех трех этих признаков.

Упражнения

3.4.1. Найдите метрику F1 для примера с кроликами и козлами, приведенного в тексте.

3.4.2. Сформулируйте модель для предсказания того, что выбранный наугад пассажир в аэропорту Шереметьево является членом террористической организации. Для этого определите признаки-атрибуты, их возможные значения, а также создайте несколько наборов таких значений в качестве обучающей выборки. По построенной модели сделайте предсказатель. Оцените насколько хорошим он вышел.

5. Простые прикладные модели

Метод *k* ближайших соседей. Пусть имеется какой-то набор данных в виде векторов размерности n и меток, соответствующих каждому вектору. Метки могут принимать значения из заданного списка меток (вообще говоря, другой размерности). На векторах ранее введена была функция расстояния `distance`. Если стоит задача по такой имеющейся выборке предсказать значение метки на новом векторе, которого нет в выборке, то можно воспользоваться *методом k ближайших соседей*, который заключается в том, что метка пробной точки (списка) определяется метками k штук ближайших по `distance` точек к ней. Это очень простой метод и он, разумеется, не всегда работает: например, мой кот белый, а вокруг все коты только рыжие и серые, и таких «точек», как я, довольно много.

Вспомним класс `Counter`: он по аргументу – списку – выдает словарь вида «элемент списка: сколько раз встречается». Пусть метки у нас записаны в упорядоченный список `labels`. Тогда функция отбора по большинству голосов-меток будет выглядеть так:

```
def majority_vote(labels):
  # метки упорядочены от ближней к дальней
  vote_counts = Counter(labels)
  winner, winner_count = vote_counts.most_common(1)[0]
  num_winners = len([count for count in vote_counts.values() if
count == winner_count])
  if num_winners == 1:
     return winner
  else:
     return majority_vote(labels[:-1]) # пытаемся снова
```

Такой подход сработает, так как всё равно всё когда-то сведется к одной метке, которая и победит.

И вот как будет выглядеть классификатор по методу k ближайших соседей.

```
def knn_classify(k, labeled_points, new_point):
  # сначала сортируем
  by_distance = sorted(labeled_points, key = lambda point:
```

```
  distance(point[0], new_point))
  # k ближайших находим
  k nearest labels = [label for  ,label in by distance[:k]]
  # голосуем и пробуем снова
  return majority_vote(k_nearest_labels)
```

Теперь попробуем предсказать породу по дню заболевания для тех поросят, что были в первой главе. Их там, правда, было мало – надо добавить данных для полноты картины и записать немного по-другому.

```
piggies = [([16,1.45],'s12'),
  ([14,1.37],'s16'),
  ([15,1.22],'s35'),
  ([26,0.95],'o31'),
  ([23,1.02],'o47'),
  ([27,0.98],'o98'),
  ([13,1.35],'o11'),
  ([14,1.29],'o14'),
  ([14,1.25],'o76'),
  ([15,1.20],'s12'),
  ([15,1.19],'s16'),
  ([23,0.98],'o98'),
  ([26,0.92],'o31'),
  ([27,0.95],'o47'),
  ([16,1.46],'s35'),
  ([13,1.32],'o11'),
  ([14,1.30],'o14'),
  ([13,1.34],'o76')]
# ключ = порода, значение = пара (день, вес)
plots = {'s12': ([],[]),'s16': ([],[]),'s35': ([],[]),'o11':
([],[]),'o14': ([],[]),'o31': ([],[]),'o47': ([],[]),'o76':
([],[]),'o98': ([],[])}
# у каждой породы – свой цвет (насколько хватило)
colors = {'s12': 'b','s16': 'g','s35': 'r','o11': 'k',
  'o14': 'm','o31': 'y','o47': 'c','o76': 'g','o98': 'b'}
for (day,weight), breed in piggies:
     plots[breed][0].append(day)
     plots[breed][1].append(weight)
for breed, (x,y) in plots.items():
     plt.scatter(x,y,color = colors[breed], marker = 'o', label
= breed)
plt.legend(loc=0)
plt.title("Зависимость заболеваемости от веса при рождении")
plt.show
```

Теперь будет так как на рис. 6.

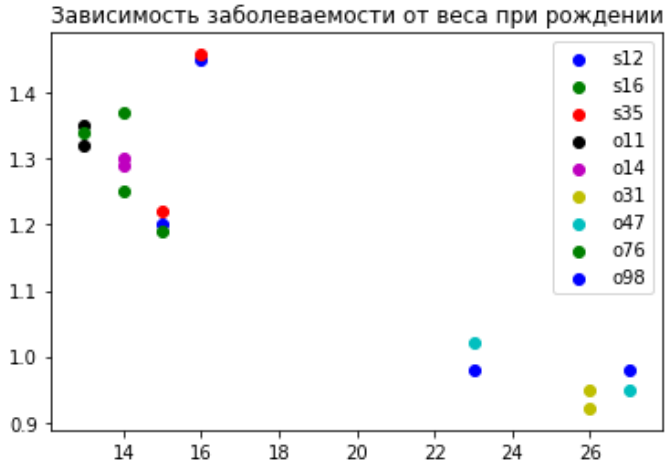


Рис. 6. Обогащенные данные по заболеваемости поросят.

Сколько соседей надо для хорошего предсказания? Выберем из набора от 1 до 6:

```
for k in range(1,7):
  n_correct = 0
  for pig in piggies:
     values, actual_br = pig
     other_pigs = [other_pig for other_pig in piggies if
other_pig != pig]
     predicted_br = knn_classify(k, other_pigs, values)
     if predicted_br == actual_br:
        n_correct +=1
     print(k, "соседей:",n_correct,"правильных из",
len(piggies))
```

Выход:

```
1 соседей: 4 правильных из 18
2 соседей: 4 правильных из 18
3 соседей: 4 правильных из 18
4 соседей: 2 правильных из 18
5 соседей: 0 правильных из 18
6 соседей: 0 правильных из 18
```

Получается, что из-за недостатка данных, у нас при любом выборе числа соседей предсказание «не очень». Но все-таки, для 1-3 соседей оно лучше, чем для большего их числа. Это связано еще и с тем, что пород довольно много для этой выборки данных. То есть, размерность

пространства выбора большая. В этом случае метод испытывает затруднения, которые получили название *проблема проклятия размерности*. Её смысл заключается в том, что пространства большой размерности вообще очень обширны – расстояния между точками в них «большие». Поэтому в том случае, если метод k ближайших соседей применяется к данным высокой размерности, то нужно сначала попытаться снизить эту размерность.

Наивный Байес. Если имеется тестовая выборка с сообщениями, часть из которых – спам (S), а остальные – хорошие сообщения ($\bar{S}$), то вероятность того, что сообщение со словом «выигрыш» (V) является спамом, равна

$$P(S|V) = \frac{P(V|S)P(S)}{P(V|S)P(S) + P(V|\bar{S})P(\bar{S})}$$

Этот способ неудовлетворителен, так как не принимает во внимание не только влияния слова на спамность сообщения, но и не учитывает множества слов в спам сообщении (крайне редко спам состоит из одного слова).

Предположим, что слов-индикаторов спам-сообщения – известное множество. Для создания фильтра под названием «*наивный Байес*» нужно предположить:

Наличие или отсутствие каждого конкретного слова в сообщении не зависит от наличия (отсутствия) других слов в сообщении.

В этом случае можно представить вероятность того, что спам-сообщение содержит выбранный набор слов $x = [x_1, \dots, x_n]$:

$$P(X_1 = x_1, \dots, X_n = x_n|S) = P(X_1 = x_1|S) \dots P(X_n = x_n|S)$$

Например, если половина спам-сообщений содержит слово «выигрыш», и половина спам-сообщений содержит слово «счёт», то наивная байесовская оценка вероятности того, что выбранное спам-сообщение содержит слова «выигрыш» и «счёт», будет равна 0.25 (произведению 0.5*0.5). Вероятность спамности сообщения в этом случае равна:

$$P(S|X = x) = \frac{P(X = x|S)}{P(X = x|S) + P(X = x|\bar{S})}.$$

Существует такое (существенное) препятствие для применения наивного Байеса. Пусть в обучающей выборке слово «информация» присутствует только в не-спамных сообщениях. Тогда оценка вероятности

$$P(I|S) = 0$$

То есть, наивный Байес будет выдавать вероятность того, что сообщение – спам, равную 0 для всех сообщений, содержащих слово «информация». Даже для тех сообщений, которые содержат и

«выигрыш», и «счёт». Для преодоления этой проблемы используют *сглаживание*. Определяют константу сглаживания k. Вероятность встретить слово x_i в спам-сообщении оценивают по формуле:

$$P(X_i|S) = \frac{k + N_s(x_i)}{2k + N_s},$$

где N_s – общее число спамных сообщений, а $N_s(x_i)$ – число спамных сообщений со словом x_i. Аналогично для $P(X_i|\bar{S})$.

Тогда если в 98 спам-сообщениях присутствует 0 слов «информация» и $k = 1$, то $P(I|S) = 0.01$.

Чтобы реализовать наивный байесовский фильтр, вычисляющий вероятность того, что заданное сообщение является спамом, нужно последовательно сделать несколько вещей. Сначала сформируем список слов из сообщения `message`, устранив повторы:

```
def tokenize(message):
  message = message.lower() # нам не нужны большие буквы
  # извлекаем слова
  all words = re.findall("[a-z0-9']+",message)
  return set(all_words) # возвращаем только уникальные
```

Теперь нужно подсчитать частотности слов в маркированной обучающей выборке `training_set` сообщений (она состоит из пар «сообщение – флаг спама»):

```
def count_words(training_set):
  counts = defaultdict(lambda: [0,0])
  for message, is_spam in training_set:
     for word in tokenize(message):
        counts[word][0 if is_spam else 1] +=1
  return counts
```

Применим сглаживание и получим оценки вероятностей в виде триплетов «слово, вероятность встретить в спаме, вероятность встретить в не-спаме»:

```
def word_probabilities(counts, total_spams, total_non_spams,
k=0.5):
  return [(w, (spam + k) / (total_spams + 2*k),
  (non_spam + k) / (total_non_spams + 2*k)
  ) for w, (spam, non_spam) in counts.iteritems()]
```

Теперь назначим вероятности сообщениям:

```
def spam_probability(word_probs, message):
  message_words = tokenize(message)
  lp_spam = lp_not_spam = 0.0
  for word, p_spam, p_not_spam in word_probs:
     if word in message_words:
        lp_spam += math.log(p_spam)
        lp_not_spam += math.log(p_not_spam)
     else:
        lp_spam += math.log(1 - p_spam)
```

```
        lp_not_spam += math.log(1 - p_not_spam)
  ps = math.exp(lp_spam)
  pns = math.exp(lp_not_spam)
  return ps/(ps+pns)
```

Задача обучения и практической реализации классификатора вынесена в лабораторную работу, как и реализация метода k ближайших соседей. Поэтому к этому параграфу упражнения отсутствуют.

Глава IV. Модели, методы и алгоритмы работы с данными

1. Регрессионный анализ

Простая линейная регрессия. Пусть имеются ряд значений независимой случайной величины $[x_1, \dots, x_n]$ и ряд значений зависимой случайной величины $[y_1, \dots, y_n]$. Требуется найти такие коэффициенты α и β, при которых y приближенно можно представлять как

$$\tilde{y} = \beta x + \alpha$$

Оценку, которую обозначили через $\tilde{y}$, можно назвать прогнозом. Для нее создадим функцию predict

```
def predict(alpha, beta, x):
  return beta*x + alpha
```

Конечно, это не точное значение. Будет ошибка

$$\varepsilon_i = y_i - \tilde{y}_i,$$

для которой сделаем функцию

```
def error(alpha, beta, x_i, y_i):
  return y_i - predict(alpha, beta, x_i)
```

Нужно найти такие α, β, при которых суммарная ошибка по всей обучающей выборке будет наименьшей. Под «суммарной ошибкой» подразумевается корень из суммы квадратов всех ошибок в выборке. Доказано (это легко), что при этом α, β определяются следующей функцией, что выражает собой *метод наименьших квадратов*:

```
def least_squares_fit(x,y):
  beta = (correlation(x,y)*standard_deviation(y) /
standard_deviation(x))
  alpha = mean(y) - beta*mean(x)
  return alpha, beta
```

Качество подгонки данных измеряет так называемый *коэффициент детерминации*, иначе называемый *R-квадрат*, который является долей суммарной вариации в зависимой переменной, предсказываемой моделью. Его можно построить следующими тремя функциями.

Первая вычислит полную сумму квадратов отклонений от среднего в зависимой выборке:

```
def total_sum_of_squares(y):
  return sum(ym_i**2 for ym_i in de_mean(y))
```

Вторая найдет сумму квадратов ошибок:

```
def sum_of_squared_errors(alpha, beta, x, y):
     return sum(error(alpha, beta, x_i, y_i) ** 2 for x_i, y_i
in zip(x,y))
```

Третья – собственно посчитает R-квадрат:

```
def r_squared(alpha, beta, x, y):
     return (1 - sum_of_squared_errors(alpha, beta, x, y) /
total_sum_of_squares(y))
```

После этого можно посчитать R-квадрат для полученных коэффициентов и данных выборок

```
r_squared(alpha, beta, missed_lectures, rejected_students)
```

Он покажет, насколько хорошо получилась *линейная регрессия*, то есть линейное приближение зависимости одного ряда чисел от другого.

Множественная регрессия. Пусть по-прежнему стоит задача найти приближенную линейную зависимость, описывающую некоторые имеющиеся данные и зависимые от них данные. Но только теперь независимые выборки состоят из наборов чисел (векторов) длины n. Обозначим $(x_0^0, \dots, x_{n-1}^0), \dots (x_0^{m-1}, \dots, x_{n-1}^{m-1})$ m штук независимых выборок одинаковой длины и $y_0, \dots, y_{n-1}$ – зависимую от них выборку. Условие регрессии:

$$\hat{y}_i = \sum_{k=0}^{m-1} \beta_k x_i^k + \alpha,$$

а функция ошибки

$$F(\alpha, \beta_0, \dots, \beta_{m-1}) = \sum_{i=0}^{n-1} (\hat{y}_i - y_i)^2$$

Условие ее минимума:

$$\frac{\partial F}{\partial \alpha} = \frac{\partial F}{\partial \beta_l} = 0,$$

из которого находим

$$\sum_{i=0}^{n-1} \left(\sum_{k=0}^{m-1} \beta_k x_i^k + \alpha - y_i \right) = 0$$

$$\sum_{i=0}^{n-1} \left(\sum_{k=0}^{m-1} \beta_k x_i^k + \alpha - y_i \right) x_i^l = 0, \qquad l = 0, \dots, m-1$$

Представим в виде системы линейных алгебраических уравнений:

$$\beta_0 \sum_{i=0}^{n-1} x_i^0 + \cdots + \beta_{m-1} \sum_{i=0}^{n-1} x_i^{m-1} + n\alpha = \sum_{i=0}^{n-1} y_i$$

$$\beta_0 \sum_{i=0}^{n-1} x_i^0 x_i^l + \cdots + \beta_{m-1} \sum_{i=0}^{n-1} x_i^{m-1} x_i^l + \alpha \sum_{i=0}^{n-1} x_i^l = \sum_{i=0}^{n-1} y_i x_i^l,$$

$$l = 0, \dots, m-1,$$

решая которую, найдем коэффициенты $\beta_0, \dots, \beta_{m-1}, \alpha$ и задача регрессии будет решена. Функцию, определяющую коэффициенты множественной регрессии, можно построить, глядя только на эту систему.

```
def regression(X,y): # X – это список m штук векторов
  m = len(y)
  M = []
  M.append([sum(x) for x in X]+[n])
  b.append(sum(y))
  for l,xl in enumerate(X):
     M.append([dot(x,xl) for x in X]+[sum(xl)])
     b.append(dot(y,xl))
  beta = gauss(M,b)
  return beta # свободный – в конце, т.е. beta[-1]есть alpha
```

Эта функция по заданным независимым векторам X и зависимому от них вектору y выдаст коэффициенты линейной регрессии α и β_k, которые обозначены в формуле условия регрессии.

Разумеется, не все данные можно успешно оценивать линейной регрессией. Во-первых, данные могут просто не иметь линейной зависимости и, кроме того, не приводиться к ней (например, x и $y = xe^x$, для чего крайне трудно найти обратную функцию). Во-вторых, даже если эта зависимость есть, то должны соблюдаться допущения модели линейной регрессии. Они следующие:

- Столбцы в матрице x должны быть линейно независимы
- Столбцы матрицы x не коррелированы со столбцом ошибок ε

Если не выполнится первое, то коэффициент β для зависимых столбцов не может быть найден. Если не выполнится второе, то оценки β будут выдавать систематически неправильный результат.

В связи с этим, следует иметь какую-то оценку качества множественной линейной регрессии. Для этого подойдет тот же критерий R-квадрат, что и раньше, но с приставкой `mult`. Создадим функцию для него. Сначала напишем вычисление приближения и ошибку:

```
def mult_predict(x_i, beta):
  return sum(xij*bj for xij,bj in zip(x_i,beta))+beta[-1]
```

```
def mult_error(x_i, y_i, beta):
  return mult_predict(x_i, beta) - y_i
```

Теперь R-квадрат:

```
def mult_r_squared(x, y, beta):
  sum_of_squared_errors = sum(mult_error(x_i, y_i, beta)**2
for x_i, y_i in zip(x,y))
  return 1.0 - sum_of_squared_errors / total_sum_of_squares(y)
```

Заметим, что добавление новых переменных, при прочих равных условиях, увеличивает R-квадрат.

Бутстрапирование данных. Если данных не очень много, а статистики по ним найти нужно, то есть два пути. Первый – просто знать о том, что полученные статистики найдены на дефицитных данных, и, следовательно не доверять им. Но единственная польза этого будет в том, что не будет потрачено время впустую – анализ все равно сделан не будет. Другой подход похож на некоторую магию: да, данных мало, но можно размножить их, предполагая, что в имеющихся данных уже есть информация об их распределении (может быть, не известная аналитику). Про информацию, имеющуюся в данных, можно написать отдельную книгу и построить на этом стратегии анализа и машинного обучения. Но есть очень простой подход для использования этой информации, которая в этом подходе интерпретируется как набор выборочных статистик и выборочное распределение. На основе построенного выборочного распределения, данные просто генерируются, а статистики сохраняются. Этот прием называется *бутстрапированием* (*bootstrapping*) и, в своей простейшей реализации, превращается в обыкновенный выбор с возвращением:

```
def bootstrap_sample(data):
  return [random.choice(data) for _ in data]
```

Далее можно построить нужную статистику

```
def bootstrap_statistic(data, stats_fn, num_samples):
  return [stats fn(bootstrap sample(data)) for   in
range(num_samples)]
```

Вот только выборки могут быть малой мощности, далекие от репрезентативности. Тогда эта простая реализация, которая апеллирует только к экземплярам выборки, не является состоятельной: нерепрезентативная выборка генерирует тенденциозные статистики при бутстрапировании, то есть, получается совсем не то, что выборка представляла в реальности. Есть метод и для этого случая, и он вытекает прямо из определения бутстрапирования. Обычно этот прием используют, когда имеется некоторый нерегулярный поток данных, но в перерывах между поступлениями данных требуется производить их анализ и выдавать предсказания. В этом случае нужно:

1. Записать непрерывную плотность вероятности $\rho(x)$, созданную из имеющейся выборки.
2. Сгенерировать нужное количество экземпляров выборки по этому распределению и отметить их как искусственные.
3. Построить нужные статистики.
4. При поступлении новых пакетов в выборку, удалить искусственные экземпляры, добавить вновь поступившие и повторить пункты 1-3.

В этом алгоритме остается немного туманным только вопрос о вычислении плотности вероятности $\rho(x)$. Здесь нужно руководствоваться экспертизой предметной области так же, как это было в параграфе, где рассматривалась аппроксимация. Например, если вы подозреваете нормальное распределение в выборке, то форма $\rho(x)$ вам уже известна, а варьируются только среднее значение и среднеквадратичное отклонение – они легко вычисляются по выборке и после этого вы можете просто использовать `random.gauss`. Если вы ожидаете равномерное распределение, то его функция у нас уже есть, а его параметры (границы) можно задать из выборки (если она достаточно большая), либо экспертным путем. Покажем бутстрапирование на этих примерах.

```
bootstrap(x,n,stats = 'normal'):
  if stats == 'normal':
    return [random.gauss(mean(x),standard deviation(x)) for
in range(n)]
  elif stats == 'even':
    return [min(x) + random.random()*(max(x) - min(x)) for _
in range(n)]
  else:
    return None
```

Можно добавлять нужные вам варианты статистик в дальнейшие `elif`. На примере нормального распределения функция работает так:

```
bootstrap([-1,-0.3,0.4,0.03,-1.3,1.1,2.5,4.2,3.4],5)
# [-1.7426508026421685, 2.600875887647808,0.9509521878199816,
# 0.03472330148751834,0.0726060985782323]
```

Видно, что новые экземпляры не содержатся в первоначальной выборке. С помощью бутстрапирования можно обогащать данные для регрессии – это обычное применение для него.

Регуляризация. Так называется метод, при котором к случайным ошибкам регрессии прибавляют штраф, увеличивающийся с ростом коэффициентов beta. Правила назначения такого штрафа могут быть разными. Если он пропорционален сумме квадратов коэффициентов `beta_i` (без beta_0), то такая регуляризация соответствует *гребневой регресии*:

```
def ridge_penalty(beta, base_penalty):
  return base_penalty * sum_of_squares(beta[:-1])
def ridge squared error(x i, y i, beta, base penalty):
  return mult_error(x_i, y_i, beta)**2+ridge_penalty(beta,
base_penalty)
```

Так получились «оштрафованные» ошибки. Дальше их можно использовать в поиске нужных β, модифицировать функцию `regression` предоставляется читателю.

Есть еще один довольно широко применяемый метод регуляризации – это *лассо-регрессия*. Штраф в ней пропорционален сумме модулей β_i.

Логистическая регрессия. Пусть стоит задача о принятии какого-то решения на основании данных. И модель должна нам выдать, например, число от 0 до 1, близость которого к каждой из альтернатив, дает нам возможность сделать соответствующий выбор. В качестве данных возьмем

```
r = [course, debts, missed, status]
```

где `course`, `debts`, `missed`, `status` есть, соответственно, курс, число несданных предметов, число пропусков и флаг состояния (подан на отчисление или нет) студента нашей кафедры. Сформируем эти данные в привычной для множественной регрессии форме

```
x = [[1] + r[:3] for r in data] # обстоятельства
y = [r[3] for r in data] # статус
```

Рассмотренная ранее линейная регрессия даст нам следующее

$$y_i = \alpha + \beta_1 c + \beta_2 d + \beta_3 m + \varepsilon$$

где c, d, m есть, очевидно, `course`, `debts` и `missed`. Это будут какие-то разные, в том числе, большие, значения. А нам надо чтобы было от 0 до 1. Известно, что логистическая функция удачно приводит числа откуда угодно в интервал от 0 до 1. Вот она:

$$f(x) = \frac{1}{1 + e^{-x}}$$

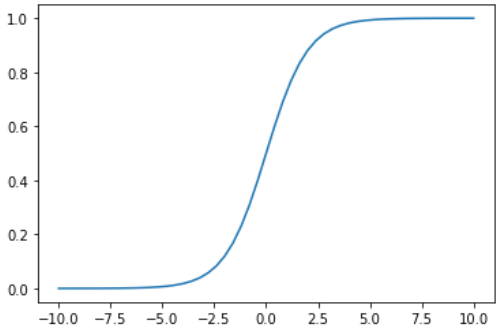


Рис. 7. Логистическая функция.

Нам понадобится не только сама функция, но и ее производная, которая оказывается равна

$$f'(x) = f(x)\big(1 - f(x)\big)$$

Зададим их

```
def sigmoid(x):
  return 1.0/(1+exp(-x))
def sigmoid d(x):
  return sigmoid(x)*(1-sigmoid(x))
```

Теперь построим такую модель

$$y_i = f(x_i\beta) + \varepsilon_i$$

и в качестве функции распределения возьмем

$$p(y_i|x_i,\beta) = f(x_i\beta)^{y_i}\big(1 - f(x_i\beta)\big)^{1-y_i},$$

логарифм этой функции (*правдоподобия*), равный

$$\ln p(y_i|x_i,\beta) = y_i \ln f(x_i\beta) + (1 - y_i)\ln\big(1 - f(x_i\beta)\big)$$

зададим и потом будем максимизировать.

```
def log_likelyhood_i(x_i,y_i,beta):
  if y_i == 1:
     return log(sigmoid(dot(x i,beta)))
  else:
     return log(1 - sigmoid(dot(x_i,beta)))
```

Теперь нужна сумма всех таких правдоподобий

```
def log_likelyhood(x,y,beta):
  return sum(log_likelyhood_i(x_i,y_i,beta) for x_i,y_i in
zip(x,y))
```

Потребуется вычислять градиент. Так что нужны еще такие функции:

```
def log_partial_ij(x_i,y_i,beta,j):
  return (y_i - sigmoid(dot(x_i,beta)))*x_i[j]
def log_grad_i(x_i,y_i,beta):
  return [log_partial_ij(x_i,y_i,beta,j) for j,_ in
enumerate(beta)]
def log_grad(x,y,beta):
  return reduce(vector_add,
         [log_grad_i(x_i,y_i,beta) for x_i,y_i in   zip(x,y)])
```

Данные различны, требуется шкалирование:

```
sc_x = scale(x)
```

Нужно разделить выборку на обучающую и тестовую:

```
random.seed(0)
x_train,x_test,y_train,y_test =train_test_split(sc_x,y,0.33)
```

Максимизируем функцию правдоподобия на обучающей выборке

```
fn = partial(log_likelyhood,x_train,y_train)
```

Устанавливаем отправную точку

```
beta_0 = [random.random() for _ in range(4)]
```

и максимизируем `fn` градиентным спуском

```
beta_hat = gradient_descent(negate(fn), beta_0)
```

В результате получаем четыре компоненты вектора β, который стоит в выражении $f(x_i\beta)$, и можем получать предсказания статусов для новых студентов, не содержащихся в обучающей выборке. По имеющемуся теперь вектору β можно сказать, что если проекция $x_i\beta$ на него уже достаточно велика, то ее дальнейший рост ничего не изменит в статусе, а если она близка к 0, то любое небольшое изменение может привести к изменению статуса и вызвать представление на отчисление.

Вот как можно работать на контрольной выборке

```
for x_i,y_i in zip(x_test, y_test):
  predict = sigmoid(dot(beta_hat, x_i))
  if y_i == 1 and predict >= 0.5:
     true_positives +=1
  elif y_i == 1:
     false negatives +=1
  elif predict >= 0.5:
     false_positives +=1
  else:
     true negatives +=1
precision = true positives/(true positives+false positives)
recall = true_positives/(true_positives+false_negatives)
```

О методе опорных векторов. Здесь остановимся на нем кратко, только понятие. Набор точек x_i таких, что $\left(\hat{\beta}\cdot x_i\right)=0$, представляет собой границу между классами (там статус у студента зыбкий). Эта граница в рассмотренном примере есть гиперплоскость в 4-мерном пространстве. Можно было сразу искать такую гиперплоскость и в этом заключается суть *метода опорных векторов* (*Support Vector Machine*). Критерием для поиска является следующий: гиперплоскость должна быть расположена так, что расстояния от нее до ближайшей точки в каждом классе максимальны. Метод *SVM* представляет собой решение задачи оптимизации.

Иногда плоскость для представленных данных может не существовать. Тогда переходят в высшую размерность, добавляя еще одно измерение в виде квадрата имеющегося измерения

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

после чего разделительная гиперплоскость обычно появляется.

Чтобы использовать SVM, нужно обратиться к соответствующему программному обеспечению (его много) или изучить методы оптимизации.

Упражнения

4.1.1. Постройте модель простой линейной регрессии для цены на бензин в России (открытые данные). Независимая переменная – курс USD/рубль. Посчитайте *R*-квадрат для неё.

4.1.2. Для данных о количестве заключенных браков в России (открытые данные) постройте модель множественной линейной регрессии в зависимости от трех факторов: среднедушевой доход, ВВП, количество жителей старше 18 лет. Используйте данные Росстат минимум за 10 лет. Вычислите *R*-квадрат.

4.1.3. Создайте рекомендательную систему для инвесторов в пару EUR/рубль: на имеющихся данных о стартовом соотношении, периоде инвестирования и успешности (да/нет) модель должна предсказывать успешность нового инвестирования методом логистической регрессии.

2. Деревья принятия решений

Понятие дерева принятия решений. «Принять решение» означает сделать выбор из каких-то альтернатив. Число альтернатив должно быть конечным, но может быть очень большим. И еще: не все они могут быть известны тому, кто принимает решение, то есть, альтернативы могут появляться по мере движения к принятию решения. Именно этот процесс – движение к принятию решения – и иллюстрирует дерево принятия решений. Пусть, например, вы хотите принять решение об отнесении какого-то живого существа к определенному биологическому виду. Тогда вы можете рассуждать примерно так, как на рис. 8.

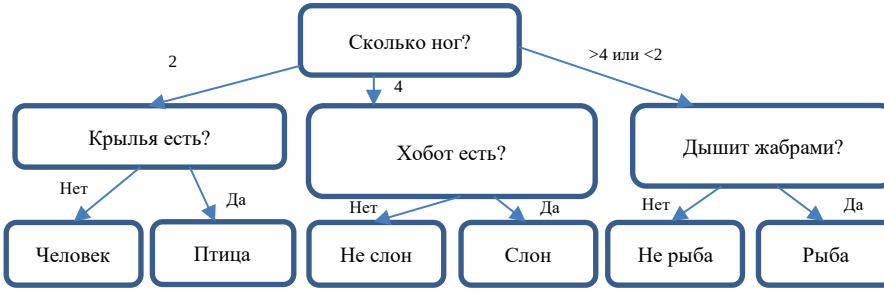


Рис. 8. Дерево принятия решения.

Конечно, это не полная классификация, и можно продолжать дерево дальше. Для этого надо превратить узлы типа «не слон» и «не рыба» в вопросы.

В общем случае, у аналитика есть какие-то данные, позволяющие давать ответы на вопросы, и принятие решения для него – это выбор конкретного экземпляра с данными, удовлетворяющими набору критериев. Делается этот выбор действиями, каждое из которых уменьшает число оставшихся для выбора экземпляров данных. Процесс заканчивается, когда остается только один экземпляр. Характеристики действий – иерархический набор признаков и связанных с ними выводов – и формирует *дерево принятия решений* (ДПР), графическая иллюстрация которого приведена на рис. 8. Проблема в том, что в действительно важных задачах эти ДПР значительно больше, чем приведенное на рис. 8, а проблема генерации признаков (то есть, построения ДПР) имеет большую вычислительную сложность.

Энтропия. Рассмотрим теперь один из вариантов построения ДПР. Нужно учесть то, как при этом изменяется оставшийся набор данных. Где-то раньше мы уже говорили, что информация уменьшает неопределенность. То есть, чем меньше данных останется, тем лучше был задан вопрос (то есть, мы получили больше информации). Для измерения количества информации есть специальная характеристика – *энтропия*. Пусть p_i – это доля экземпляров данных из множества S, принадлежащих классу c_i (классы не пересекаются и $\sum p_i = 1$). Тогда энтропия по определению равна

$$H(S) = -\sum_k p_k \log p_k,$$

логарифм берем по основанию 2 и считаем, что $0 \log 0 = 0$. Запишем в виде функции:

```
def entropy(class_probabilities):
  return sum(-p*log(p,2) for p in class_probabilities if p)
```

Чтобы вычислять значения p_k, понадобится функция

```
def class_probabilities(labels):
  total_count = len(labels)
  return [count/total_count for count in
Counter(labels).values()]
```

Если есть маркированные данные, то их энтропия будет равна

```
def data_entropy(labeled_data):
  labels = [label for _,label in labeled_data]
  probabilities = class_probabilities(labels)
  return entropy(probabilities)
```

Если множество данных S разделить на подмножества $S_1, \dots, S_m$, то энтропия всего множества может быть вычислена по формуле

$$H(S) = q_1 H(S_1) + \cdots + q_m H(S_m),$$

где q_j – пропорция данных в подмножестве S_j. И это запишем в виде функции

```
def partition_entropy(subsets):
  total count = sum(len(subset) for subset in subsets)
  return sum(data entropy(subset) * len(subset) / total count
for subset in subsets)
```

Алгоритм ID3. Построим ДПР одним из самых простых алгоритмов, который называется ID3. Рассмотрим задачу предсказания того, что студент продолжит обучение в следующем семестре на основе таких размеченных данных о студентах: направление подготовки (одно из трех возможных: 090301, 090304 или 100503), наличие задолженности за прошлую сессию (да/нет), наличие задолженности за более ранние сессии (да/нет), класс пропусков за истекший семестр (посетил менее 30%, от 30% до 50%, от 50% до 80%, более 80%), зарегистрирован в беседах ВК кафедры (да/нет), зарегистрирован в classroom.google по всем дисциплинам (да/нет), спортсмен (да/нет), активен в студенческих мероприятиях (да/нет), отзывы преподавателей в большинстве (плохо / хорошо / отлично). Меткой является «True (продолжает обучение) / False (не продолжает обучение)». Всё это задано в виде списка кортежей

```
inputs = [({'level':'Senior', 'lang':'Java', 'tweets':'no',
'phd':'no'}, False),
         ({'level':'Senior', 'lang':'Java', 'tweets':'no',
'phd':'yes'}, False),
         ({'level':'Mid', 'lang':'Python', 'tweets':'no',
'phd':'no'}, True),
         ({'level':'Junior', 'lang':'Python', 'tweets':'no',
'phd':'no'}, True),
 …
         ({'level':'Junior', 'lang':'Python', 'tweets':'no',
'phd':'yes'}, False)]
```

Нашей задачей является построение ДПР, которое для вновь вводимого набора решает – продолжит этот студент учебу или нет. ДПР будет состоять из *решающих узлов*, в которых задается вопрос и после ответа указывается нужное направление, а также *листовых узлов*, в которых озвучивается прогноз. Алгоритм ID3 имеет вид:

1. Если все данные имеют одинаковую метку, то создать листовой узел, предсказывающий эту метку, и остановиться.
2. Если список атрибутов пуст (нет вопросов), то создать листовой узел, предсказывающий наиболее общую метку, и остановиться.
3. Разбить данные по каждому атрибуту.

4. Выбрать разбиение (и соответствующий атрибут) с наименьшей энтропией.
5. Добавить решающий узел с выбранным атрибутом.
6. Повторить для каждого подмножества разбиения.

Это жадный алгоритм – он на каждом шаге оптимизирует свои действия, то есть, он локально оптимизирован. Глобально оптимизированный алгоритм значительно более вычислительно сложен.

Зададим функции, необходимые для реализации алгоритма. Сначала сделаем словарь, ключами в котором будут атрибуты, а значениями – кортежи из списка `input_data`.

```
def partition_by(inputs,attribute):
  groups = defaultdict(list)
  for inp in inputs:
    key = inp[0][attribute]
    groups[key].append(inp)
  return groups
```

Теперь вычислим энтропию разбиения по атрибуту – ключу созданного словаря.

```
def partition_entropy_by(inputs,attribute):
  partitions = partition_by(inputs,attribute)
  return partition_entropy(partitions.values())
```

Приведенный ниже скрипт реализует алгоритм ID3.

```
def build_tree_id3(inputs, split_candidates=None):
  if split_candidates is None:
    split_candidates = inputs[0][0].keys()
  num_inputs = len(inputs)
  num_trues = len([label for item, label in inputs if label])
  num_falses = num_inputs - num_trues
  if num_trues == 0: return False
  if num_falses == 0: return True
  if not split_candidates:
    return num_trues >= num_falses
  best_attribute = min(split_candidates,
                  key = partial(partition entropy by, inputs))
  partitions = partition by(inputs, best attribute)
  new candidates = [a for a in split candidates if a !=
best_attribute]
  subtrees = {attribute_value : build_tree_id3(subset,
new_candidates) for attribute_value,subset in
iter(partitions.items())}
  subtrees[None] = num_trues > num_falses
  return (best_attribute, subtrees)
```

Проведение классификации по созданному ДПР поручим следующей функции.

```
def classify(tree, inp):
  if tree in (True, False):
    return tree
```

```
  attribute, subtree_dict = tree
  subtree_key = inp.get(attribute)
  if subtree key not in subtree dict:
    subtree key = None
  subtree = subtree dict[subtree key]
  return classify (subtree, inp)
```

На приведенных выше данных о студентах получим следующее:

```
tree = build_tree_id3(inputs)
classify(tree, {"level":"Junior", "lang": "Python",
                "tweets": "no", "phd": "no"})
#True
```

Упражнения

4.2.1. Постройте дерево принятия решений, которое получится в результате полной проработки примера из текста (доделайте дерево дальше). Для этого создайте около 20 строк данных, максимально приближенных к реальности на ваш взгляд.

4.2.2*. Создайте данные и постройте ДПР для предсказания успешности прохождения собеседования кандидатом. Критерии и их значения можете выбрать самостоятельно.

3. Нейронные сети

Определение искусственной нейронной сети. В природе распознающей системой является нервная система. Она состоит из набора нейронов, у каждого из которых есть синапсы – по ним нейрон получает входящие сигналы – и аксоны (чаще всего, один) – по нему от нейрона отходит сигнал. Синапсы могут быть соединены как непосредственно с источником сигнала (например, клеткой глаза), так и с аксоном другого нейрона. Так получается нейронная сеть нервной системы.

В случае искусственной нейронной сети (ИНС) люди имитируют нейрон математическими объектами – *весами*, *функцией активации* и *смещением*, которые обрабатывают набор входящих сигналов. Получается, что ИНС представляет собой набор этих математических объектов. Если сигнал приходит в нейрон из другого нейрона, то вес того входа нейрона-получателя, который связан с нейроном-отправителем, будет отличен от нуля, а канал передачи данных между ними становится доступным. Набор доступных каналов называется *топологией ИНС*.

Один из наиболее распространенных python-пакетов для работы с ИНС называется tensor-flow, что в переводе означает «тензорный поток». Казалось бы, какое отношение имеют тензоры – объект линейной алгебры – к искусственным нейронным сетям? Оказывается, самое непосредственное. Для упрощения (у нас ведь упрощенный курс) будем понимать под тензором конечное множество чисел, упорядоченных с помощью индексов. Например, вектор – это тензор, упорядоченный одним индексом, а матрица – тензор, упорядоченный двумя индексами. Обобщая, можно сказать, что тензор может быть упорядочен любым конечным набором индексов.

С точки зрения ИНС, нас будут интересовать 4-индексные и 3-индексные тензоры. Технически вектор представляется списком, матрица – списком списков, 3-индексный тензор – списком списков списков, а 4-индексный, как легко догадаться, списком списков списков списков.

Искусственный нейрон, или далее просто *нейрон*, представляет собой тройку объектов: вектор (список) весов w, функцию активации f, и смещение b. Аргументом функции активации является произведение матрицы весов на вектор входящего сигнала, смещенное (то есть, сложенное с…) на смещение. Значением этой функции является величина выходного сигнала. Набор нейронов, сигналы которых никак не связаны (между ними нет ни прямого, ни опосредованного доступного канала), называется *слоем* ИНС. Слой представляется в виде списка нейронов. Если у всех нейронов функция активации одинаковая (чаще всего, это так), то слой – это список списков с указанием вида функции активации.

Рассмотрим сначала работу одного нейрона с J входами. Входной сигнал представлен векторами x_j^a, где a – номер пакета. Выход представлен набором чисел y^a, соответствующих каждому пакету и принимающих значения от 0 до 1. Закон преобразования входа в выход имеет вид

$$y^a = f\left(\sum_{j=0}^{J-1} w^j x_j^a + b\right)$$

где w^j – веса нейрона, b – его смещение, а f – функция активации. Далее в качестве функции активации мы рассмотрим сигмоиду, так как она уже определена и ее свойства мы описывали. Конечно же, могут быть и другие функции активации, но в кратком изложении мы их не рассматриваем.

Цель работы нейрона – научиться на некотором наборе пакетов выдавать определенные для этого пакета выходные сигналы. То есть, если есть размеченные данные (x_j^a, l^a), то, принимая на вход x_j^a, нейрон должен выдавать максимально близкое к l^a число. Это требование можно выразить так

$$\sum_{a=0}^{A-1}(y^a - l^a)^2 \to min$$

или, что то же самое, найти такие значения w^j и b, которые соответствуют минимуму функции

$$F(w^j, b) = \sum_{a=0}^{A-1}(y^a - l^a)^2.$$

Найдем этот минимум методом градиентного спуска.

$$\frac{\partial F}{\partial w^k} = 2\sum_{a=0}^{A-1}(y^a - l^a)\frac{\partial f\left(\sum_{j=0}^{J-1} w^j {x_j}^a + b\right)}{\partial w^k}$$

$$= 2\sum_{a=0}^{A-1}(y^a - l^a)y^a(1 - y^a){x_k}^a$$

$$\frac{\partial F}{\partial b} = 2\sum_{a=0}^{A-1}(y^a - l^a)\frac{\partial f\left(\sum_{j=0}^{J-1} w^j {x_j}^a + b\right)}{\partial b} = 2\sum_{a=0}^{A-1}(y^a - l^a)y^a(1 - y^a)$$

Согласно методу градиентного спуска,

$$w'^k = w^k - \mu\frac{\partial F}{\partial w^k}$$

или

$$\Delta w^k = -\mu\sum_{a=0}^{A-1}(y^a - l^a)y^a(1 - y^a)x_k^a, \Delta b = -\mu\sum_{a=0}^{A-1}(y^a - l^a)y^a(1 - y^a)$$

Это преобразование весов и смещений называется *обобщенным дельта-правилом*. А величина $(y^a - l^a)y^a(1 - y^a)$ обозначается δ^a

$$\delta^a = (y^a - l^a)y^a(1 - y^a).$$

Если в слое не один нейрон, а их K штук, то обобщенное дельта-правило будет выглядеть так (для каждого нейрона):

$$\Delta w_k^j = -\mu\sum_{a=0}^{A-1}(y_k^a - l_k^a)y^a(1 - y_k^a)x_j^a,$$

$$\Delta b_k = -\mu\sum_{a=0}^{A-1}(y_k^a - l_k^a)y^a(1 - y_k^a),$$

$$k = 0, \dots, K-1,$$

где y_k^a, l_k^a – выходной сигнал и метка k-го нейрона на a-м пакете.

Обучение нейрона есть процесс нахождения оптимальных для него весов и смещений. Обучать группу не связанных между собой нейронов (слой) можно с помощью последних указанных соотношений – они дают возможность находить следующее, более близкое к идеальному, приближение значений весов и смещений, зная предыдущие значения весов и смещений, и имея значения меток для каждого обучающего пакета.

Пусть теперь имеется двухслойная нейронная сеть с K нейронами во втором слое и с J нейронами в первом слое. Каждый из J нейронов имеет N синапсов, то есть, принимает сигнал в виде вектора длиной N. Обозначим выход нейронов первого слоя через z_j^a. Для обучения первого слоя у нас отсутствует метка h_j^a для каждого пакета на выходе первого слоя. Чтобы осуществить обучение многослойной (в данном случае – двухслойной) сети, полагают (предположение модели):

$$z_j{}^a - h_j{}^a = \sum_{k=0}^{K-1} \delta_k^a w_k^j$$

Как и раньше, можно записать дельта-правило для первого слоя:

$$\Delta v_j^n = -\mu \sum_{a=0}^{A-1} \left(z_j^a - h_j^a\right) z_j^a \left(1 - z_j^a\right) x_n^a,$$

$$\Delta c_j = -\mu \sum_{a=0}^{A-1} \left(z_j^a - h_j^a\right) z_j^a \left(1 - z_j^a\right)$$

Подставим наше предположение:

$$\Delta v_j^n = -\mu \sum_{a=0}^{A-1} \sum_{k=0}^{K-1} \delta_k^a w_k^j \, z_j^a \left(1 - z_j^a\right) x_n^a,$$

$$\Delta c_j = -\mu \sum_{a=0}^{A-1} \sum_{k=0}^{K-1} \delta_k^a w_k^j \, z_j^a \left(1 - z_j^a\right)$$

Это выражение позволяет обучить первый слой, если уже обучен второй слой. Алгоритм обучения выглядит так:

1. Провести прямой проход на данных весах и смещениях: получить v_j^n, c_j, w_k^j, b_k.
2. Обучить последний слой: получить новые веса $\widehat{w}_k^j$ и смещения $\hat{b}_k$ нейронов последнего слоя.

3. Вычислить разность $z_j^a - h_j^a$ сигнала и метки на выходе предыдущего слоя по предположению модели.
4. Обучить предыдущий слой: получить для него веса и смещения нейронов по только что приведенной формуле.
5. Повторять шаги 3 и 4, до входного слоя.

Этот метод обучения многослойных ИНС называется *метод обратного распространения ошибки*.

Если у весов ИНС нумеровать индексами не только нейрон и канал получения сигнала, но и слой, в котором нейрон находится, а у смещения – нейрон и слой, то вся ИНС будет представлять собой пару тензоров вида

$$W_{ad}^{lm}, B_a^l$$

где a – индекс (номер) слоя-акцептора сигнала, d – индекс (номер) слоя-донора сигнала, а индексы l и m указывают на номера нейронов, соответственно, в слоях a и d. Само значение величины W_{ad}^{lm} с конкретным значением индексов есть вес связи нейрона l слоя a с нейроном m слоя d, а величина B_a^l характеризует смещение в нейроне l в слое a.

ИНС с взаимодействием только между слоями, номер которых отличается на 1, называется *сетью прямого распространения*, дальше рассматривать будем только такие сети. Сеть прямого распространения с единой функцией активации f можно представить с помощью 3-индексного тензора весов, смещение характеризуется по-прежнему:

$$W_a^{lm}, B_a^l.$$

Таким образом, сеть прямого распространения есть два объекта: список списков списков и список списков, которые изначально можно задать случайным ненулевым выбором весов, а смещения положить равными 0.

Прямое распространение сигнала по ИНС. Очевидно, сети прямого распространения получили свое название не просто так: в них возможно распространение сигнала от некоторого входа (слоя с минимальным значением индекса) к выходу (слою с максимальным значением индекса). При этом выходной сигнал предыдущего слоя становится входным сигналом следующего слоя.

Построителем ИНС может быть, например, такая функция:

```
# инициируем веса и смещения в слоях с указанным числом
# нейронов
def art_nn(entry,layers):
  WW, W, B, b = [], [], [], []
    for li in range(layers[0]):
      W.append([random()-1/2 for _ in range(entry)])
```

```
      b.append(0)
    WW.append(W)
    B.append(b)
    if len(layers) == 1:
      return WW, B
    else:
      for s,ls in enumerate(layers[1:]):
        W, b = [], []
        for lj in range(ls):
          W.append([random()-1/2 for _ in range(layers[s])])
          b.append(0)
        WW.append(W)
        B.append(b)
      return WW, B
```

Она выдаст нам тензоры W и B. В обозначениях в коде используется мнемоническое правило: числа и списки обозначаются малыми буквами, списки списков – большими буквами, списки списков списков – двумя большими буквами. Собственно, выход этой функции и есть ИНС.

Зададим еще пару служебных функций:

```
# прямой проход сигналом x одного нейрона с весами w и
# смещением b
def fwd(x,w,b):
  return sigmoid(dot(w,x)+b)

# data есть список списков с меткой в конце каждого вектора
def ch_weights(w, b, data, mu=0.05):
  X, l = [di[:-1] for di in data], [di[-1] for di in data]
  y = [fwd(x,w,b) for x in X]
  new_w = []
  for k,_ in enumerate(X[0]):
    new_w.append(w[k] - mu*sum((y[a]-l[a])*y[a]*(1-
                 y[a])*X[a][k] for a,_ in enumerate(X)))
  b -= mu*sum((ya-la)*ya*(1-ya) for ya,la in zip(y,l))
  return new_w,b
```

Запишем функцию, реализующую прямое распространение сигнала в сети.

```
def forward(data, WW, B): # data не содержат метки
  layer_outputs = [[] for _ in B]
  for d in data:
    oi = []
    for i,bi in enumerate(B[0]):
      oi.append(fwd(d,WW[0][i],bi))
    layer_outputs[0].append(oi)
      if len(B) == 1:
        continue
      else:
        k = 1
        for W,b in zip(WW[1:],B[1:]):
```

```
          oo = []
          for i,bi in enumerate(b):
            oo.append(fwd(oi,W[i],bi))
          oi = copy(oo)
          layer_outputs[k].append(oo)
          k+=1
  return layer_outputs
```

Например, при вводе

```
WW, B = art_nn(4,[2,1])
data = [[1,2,3,0],[-1,0,1,1],[2,-1,-1,0],[2,1,-3,2],
        [4,-2,3,3],[-1,3,3,0],[5,4,3,1],[5,3,2,1],[4,4,3,1]]
labeled_data = [[1,2,3,0,0],[-1,0,1,1,1],[2,-1,-1,0,0],[2,1,-
3,2,1],[4,-2,3,3,1], [-1,3,3,0,0], [5,4,3,1,1], [5,3,2,1,1],
 [4,4,3,1,1]]
forward(data,WW,B)
```

будет получен результат прохода ИНС с двумя слоями (первый – два нейрона, второй – один нейрон), на вход которой подан пакет из девяти 4-мерных сигналов. Этот результат содержит наборы выходов из первого и из второго слоев:

```
[[[0.09629142343120865, 0.23997261454494073],
  [0.37734547262973617, 0.41913304420750547],
  [0.6495176006081035, 0.7057324166197992],
  [0.6032314904294377, 0.512955953959008],
  [0.18204103892216658, 0.7232492926398323],
  [0.08275358036454869, 0.12924170035486413],
  [0.026065448026246756, 0.1709144581683945],
  [0.05830937905141907, 0.27070721592804803],
  [0.029035141890123385, 0.1515530139844145]],
 [[0.5063297695185256],
  [0.5296569512994028],
  [0.5510764233257199],
  [0.5487823723249078],
  [0.509439547520545],
  [0.5061614260722612],
  [0.5007204742406287],
  [0.5026579753760051],
  [0.5011665216366817]]]
```

Но пока ИНС не обучена, эти выходы далеки от того, что требует разметка данных.

Сначала построим функцию, обучающую один слой.

```
# labeled_data содержит номер возбужденного нейрона в конце
# каждого вектора
def layer_train(labeled_data,W,b,mu=0.05):
  Y, L, Delta = [], [], []
  Y = forward([ld[:-1] for ld in labeled_data], [W], [b])[-1]
  for ld in labeled_data:
    L.append([1 if ld[-1] == i else 0 for i,_ in
              enumerate(b)])
    Delta.append([(Y[-1][i]-L[-1][i])*Y[-1][i]*(1-Y[-1][i])
```

```
                        for i,_ in enumerate(b)])
  new_W = [[wk - mu*sum(Delta[a][i]*labeled_data[a][k]
                          for a,  in enumerate(L))
            for k,wk in enumerate(w)]
          for i,w in enumerate(W)]
  new_b = [bi - mu*sum(Delta[a][i] for a,_ in enumerate(L))
             for i,bi in enumerate(b)]
  return new_W, new_b, Delta
```

Теперь вспомним предположение модели и распространим это обучение по всей ИНС:

```
def train_step(labeled_data,WW,B,mu=0.05):
  new_WW, new_B = [[] for _ in B], [[] for _ in B]
  data = [ld[:-1] for ld in labeled data]
  layer outputs = forward(data, WW, B)
  n = len(B) - 1
  Delta = 0
  if n==0:
    new WW[n], new B[n] = layer train(labeled data,WW[-1],B[-
1],mu)[:-1]
  else:
    new_WW[n], new_B[n], Delta = layer_train(layer_outputs[-
2],WW[-1],B[-1],mu)
    for i in range(1,n+1):
      Z = layer_outputs[n-i]
      H = [[Z[a][j] - sum(Delta[a][k]*WW[n-i+1][k][j] for k,
in enumerate(B[n-i+1]))
            for j,_ in enumerate(B[n-i])]
          for a,_ in enumerate(Z)]
      LD = [z+h for z,h in zip(Z,H)]
      new WW[n-i],new B[n-i], Delta = layer train(LD,WW[n-
i],B[n-i],mu)
  return new_WW, new_B
```

Реализованные в функциях алгоритмы проводят первый раунд обучения, который называется *эпохой*. В общем случае число эпох может быть довольно большим. Вот, например, как учится ИНС на данных, приведенных в примере.

```
for i in range(200):
    WW, B = train_step(labeled_data,WW,B,0.5)
    if not i%10:
        Y1 = forward(data, WW, B)[-1]
        L1 = [(ld[-1]+1)%2 for ld in labeled_data]
        EL = [round(abs(y1[0]-l1),1) for y1,l1 in zip(Y1,L1)]
        print(EL, sum(EL))
```

Здесь заложено 200 эпох, а результат отображается в каждой 10-й эпохе. На выходе будет следующее:

```
[0.7, 0.3, 0.7, 0.3, 0.3, 0.7, 0.3, 0.3, 0.3]
3.8999999999999995
[0.9, 0.1, 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.1]
3.3000000000000003
```

```
[0.9, 0.1, 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.1]
3.3000000000000003
[0.9, 0.1, 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.1]
3.3000000000000003
[0.9, 0.1, 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.1]
3.3000000000000003
[0.9, 0.0, 1.0, 0.1, 0.1, 1.0, 0.1, 0.1, 0.1]
3.4000000000000004
[1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0] 3.0
[1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0] 3.0
[1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0] 3.0
…
```

Как видим, уже после 60-й эпохи ИНС обучилась до того, что на обучающей выборке она выдает полностью точные результаты.

Для практического закрепления материала этого параграфа имеется лабораторная работа.

Глава V. Кластеризация и прикладные задачи

1. Модель и примеры кластеризации

Задача кластеризации. Проще всего понять, что такое кластеризация – по рисунку. Можем обратиться к тем же больным поросятам на рис. 3. Визуально на нем можно обнаружить два *кластера* – «восприимчивые толстые» и «сопротивляющиеся худые». Но есть еще обогащенные данные на рис. 6, где можно выделить кластер «сопротивляющихся супертолстых», состоящий, правда, всего из двух точек, но наиболее интересный для животноводов. Таким образом, как и раньше, получение результатов анализа данных (в данном случае – кластеров данных) зависит от установок аналитика данных. Не захочет он выделять «супертолстых» - не будет этого кластера.

Кластером будем называть такое подмножество в данных, элементы которого обособлены от остальных элементов множества данных, то есть, представленные в виде точек в пространстве, элементы кластера могут быть целиком размещены в пространственной области, не содержащей (иногда – почти не содержащей) других элементов данных.

Понятно, что разделить данные на кластеры можно не единственным образом. И вновь появившаяся точка – новый элемент множества данных – может попасть в разные кластеры в зависимости от установленных метрик. Если бы на рис. 6 появилась точка с

координатами (18, 1.1), то совсем не очевидно к какому кластеру ее отнести.

Далее будем считать, что наши данные представлены в виде d-мерных векторов и задача кластеризации сводится к определению групп данных и, если нужно, нахождению репрезентативного элемента в группе (он, как депутат, представляет всю группу перед аналитиком).

Метод k средних. В этом методе кластеризации заранее выбирается число кластеров k. В некотором роде, это может показаться абсурдным – мы не знаем, что за смысл будет в полученных группах данных, но нам хочется, чтобы этих групп было k штук. Тем не менее, иногда такая постановка вопроса полезна. Например, когда вы можете принять к анализу только 5 признаков объекта, а у вас их 1280. Тогда придется разделить всё это множество на 5 подмножеств и взять из каждого по одному «депутату» - признаку, представляющему группу.

Сделаем класс, который будет выполнять это. Его экземплярами будут «кластеризаторы по методу k средних».

```
class KMeans:
  def   init  (self, k):
     self.k = k
     self.means = None
  def classify(self, inp):
     return min(range(self, k), flag = lambda i:
squared_distance(inp, self.means[i]))
  def train(self, inps):
     self.means = random.sample(inps, self.k)
     assignments = None
     while True:
        new_assignments = map(self.classify, inps)
        if assignments == new assignments:
           return
        assignments = new_assignments
        for i in range(self.k):
           i points = [p for p,a in zip(inps, assignments) if a
== i]
           if i points:
              self.means[i] = vector_mean(i_points)
```

Для сравнения приведем также и функциональную реализацию:

```
def clustering(data,k):
    V = volume(data)
    C = rand_centers(V,k)
    G = group(C,data)
    new C = step centers(G)
    while compare(C,new C):
        C = dict(new_C)
        G = group(C,data)
        new C = step centers(G)
    return dict(new_C), dict(G)
```

Она реализует алгоритм, который имеет вид:

1. Устанавливаем меру в пространстве данных, т.е. определяем функцию, которая для любых двух векторов $x_j{}^a$ и $x_j{}^b$ вычислит число, являющееся расстоянием между точками, соответствующими этим векторам.
2. Определяем параллелепипед, в котором находятся наши данные $(min_a\ x_0^a, \ldots, min_a\ x_{J-1}^a) - (max_a\ x_0^a, \ldots, max_a\ x_{J-1}^a)$
3. Внутри этого параллелепипеда случайно выбираются k точек - центров кластеров.
4. Для каждой точки данных определяется ближайший к ней центр, данные группируются по центрам, то есть, образуются первичные кластеры.
5. В каждом кластере вычисляется точка – новый центр кластера – ее координаты являются соответствующими средними по координатам входящих точек данных.
6. Шаги 4 и 5 повторяются до тех пор, пока центр кластера сдвигается.

Приведенные шаги отражаются в следующих функциях, использующихся в кластеризаторе:

```
def volume(data):
    return [min([di[j] for di in data])
           for j,_ in enumerate(data[0])],
           [max([di[j] for di in data])
           for j,_ in enumerate(data[0])]
def rand_centers(V,k):
    centers = defaultdict(list)
    n = len(V[0])
    for i in range(k):
        for j in range(n):
            centers[i].append(random.randint(V[0][j],V[1][j]))
    return centers
def group(C,data):
    clasters = defaultdict(list)
    for d in data:
        d_inf = float('inf')
        for j in C.keys():
            if distance(d,C[j])<d inf:
                i = j
                d_inf = distance(d,C[j])
        clasters[i].append(d)
    return clasters
def step centers(G):
    new_C = defaultdict(list)
```

```
    for k in G.keys():
        n = len(G[k][0])
        for i in range(n):
            new_C[k].append(sum(g[i] for g in G[k])/n)
    return new_C

# m - это число знаков после запятой, принимаемых во внимание
def compare(C1,C2,m = 2):
    check_list = []
    for k in C1.keys():
        check_list.append(round(sum((c1-c2)**2 for c1,c2
                                in zip(C1[k],C2[k])),m))
    return any(check_list)
```

Далее рассмотрим пример применения кластеризации. Воспользуемся созданным классом. Допустим ваша задача – собрать отчеты торговых представителей, работающих в разных местах города. Представителей много и собирать их всех в центральном офисе неэффективно – затраты на поездку, плюс очередь из них в офисе. Гораздо лучше отправить за отчетами курьера, который остановится всего в k точках, к которым торговые представители подвезут отчеты – им это будет нетрудно, так как точки эти будут расположены близко к ним. На выборе значения k остановимся подробнее дальше, а пока будем считать, что оно задано. Тогда вашу задачу решит такой простой скрипт:

```
courier_route = KMeans(k)
courier_route.train(inps)
print(courier_route.means)
```

где `inps` это список списков текущих координат ваших торговых представителей (может формироваться в реальном времени). Если вам попался упрямый курьер, не желающий делать k остановок, а только, например, $k-2$, то ставите вместо k значение $k-2$ и всё – из класса будет вызван $k-2$ кластеризатор и будут выданы другие координаты.

Выбор числа k. Обычно всё-таки имеется свобода в выборе числа k. Как ею воспользоваться, то есть как выбрать самое лучшее k? Для этого можно использовать суммарное квадратичное отклонение: для каждого k находим сумму квадратов отклонений точек данных от выбранного среднего в их кластере. Будет что-то похожее на график на рис. 8. Функция, которая возвращает квадратичное отклонение, такова:

```
def squared_errors(inps, k):
  clasterbuilder = KMeans(k)
  clasterbuilder.train(inps)
  means = clasterbuilder.means
  inclaster = map(clasterbuilder.classify, inps)
```

```
  return sum(squared_distance(inp, means[claster]) for inp,
cluster in zip(inps, inclaster))
```

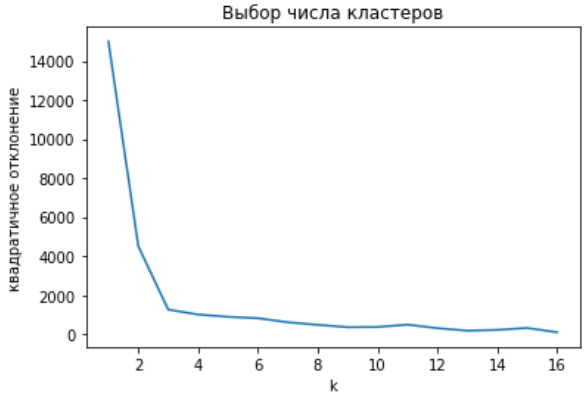


Рис. 8. Выбор числа кластеров по квадратичному отклонению.

Оптимальное число k будет находиться в месте излома кривой. На рис. 8 – это число 3. Аргументация проста: в случае меньшего числа кластеров, можно легко уменьшить кластеры – их размер сильно уменьшается при увеличении k, а в случае большего числа кластеров, усилия, связанные с увеличением значения k не приносят существенного результата – размеры кластеров остаются примерно теми же.

Иерархическая кластеризация. Восходящий метод. Рассмотрим теперь случаи, когда число кластеров заранее неизвестно. Будем строить кластеры, укрупняя их – это будет *восходящая кластеризация*. Изначально каждая точка данных будет кластером. Далее реализуем следующий алгоритм.

1. Создать $k = N$ кластеров, где N – число точек в данных.
2. Пока $k > 2$, найти два ближайших кластера и объединить.
3. Если полученная кластеризация не удовлетворяет требованиям аналитика, то отменить последний шаг 2.

Таким образом, мы дойдем сначала до двух больших кластеров, а затем будем все время увеличивать число кластеров, пока не удовлетворим наши требования.

Для параметризации используем кортежи и списки. В частности, изначально кластеры будут иметь такой вид

```
leaf1, leaf2 = ([13, 24],), ([11, -14],)
```

и так далее. Объединения кластеров будут выглядеть так:

```
merged1 = (1, [leaf1, leaf2])
```

Функция `is_leaf` будет возвращать нам ответ на вопрос – является ли кластер листовым (то есть, одноточечным)?

```
def is_leaf(cluster):
  return len(cluster) == 1
```

Функция `get_children` будет выдавать дочерние элементы кластера

```
def get_children(cluster):
  if is_leaf(cluster):
     raise TypeError('листовой не имеет дочерних')
  else:
     return cluster[1]
```

Для получения значений сделаем такую функцию

```
def get_values(cluster):
  if is_leaf(cluster):
     return cluster
  else:
     return [val for val in get_values(child) for child in
get_children(cluster)]
```

Расстояние между кластерами можно вычислять по-разному. Поэтому оставим свободу выбора целевой функции, задав ее по умолчанию как min для расстояния между ближайшими точками объединяющихся кластеров.

```
def cluster_distance(cluster1, cluster2, distance_agg = min):
  return distance_agg([distance(inp1, inp2) for inp1 in
get_values(cluster1) for inp2 in get_values(cluster2)])
```

Нужно будет еще уметь получать порядковый номер объединения. Это делается функцией

```
def get_merge_order(cluster):
  if is_leaf(cluster):
     return float('inf')
  else:
     return cluster[0]
```

Алгоритм восходящей кластеризации теперь можно собрать из созданных частей. Об эффективности пока не заботимся – важнее наглядность.

```
def bottom_up_cluster(inps, distance_agg = min):
  clusters = [(inp,) for inp in inps]
  while len(clusters) > 1:
     c1, c2 = min([(cluster1, cluster2) for i, cluster1 in
enumerate(clusters) for cluster2 in clusters[:i]], key =
lambda x,y: cluster_distance(x, y, distance_agg))
     clusters = [c for c in clusters if c != c1 and c != c2]
     merged_cluster = (len(clusters), [c1, c2])
     clusters.append(merged_cluster)
  return clusters[0]
```

Осталось написать функцию для разъединения кластеров.

```
def generate_clusters(base_cluster, num_clusters):
  clusters = [base_cluster]
  while len(clusters) < num clusters:
     next cluster = min(clusters, key = get merge order)
     clusters = [c for c in clusters if c != next_cluster]
     clusters.extend(get_children(next_cluster))
  return clusters
```

Дальше делаем так:

```
base_cluster = bottom_up_cluster(inps)
three_clusters = [get_values(cluster) for cluster in
generate_clusters(base_cluster, 3)]
```

и у нас готово разбиение на три кластера.

Для закрепления изученного материала, настоятельно рекомендуется выполнить упражнение к параграфу.

Упражнения

5.1.1. Скачайте датасет терактов https://www.kaggle.com/andrewmvd/terrorism-events. Напишите скрипт, который осуществляет кластеризацию по любому выбранному вами методу. Параметры кластеризации: время (дата), число жертв, страна. Страны параметризуйте порядковым номером, дату переведите в натуральное число.

5.1.2. Выберите любой датасет (можно на kaggle), параметризуйте его (переведите в векторы), затем найдите кластеры (в полном пространстве признаков) по методу k средних и восходящей кластеризацией так чтобы итоговое число кластеров было одинаковым. Найдите максимальное расстояние между центрами кластеров в первом и втором случае. Сравните методы по этому показателю.

2. Естественный язык

Смысл. Этот подзаголовок несет в себе два оттенка: смысл идеи данного параграфа и раскрытие понятия *смысл*. Разбирается с этим подраздел науки об обработке естественного языка – семантический анализ. В первую очередь, заметим, что если бы слово «смысл» стояло одно посреди чистого листа бумаги, то указанного контекста при прочтении не возникло бы. Значит, для определения смысла важно то, в окружении каких слов появилось данное.

Зачем вообще разбирать записи, сделанные на естественном языке? Есть, как минимум, три цели, которые преследуются этим:

1. Получить команду, которую можно использовать как директиву искусственного языка и, в результате, управлять машиной просто словами.
2. Получить все-таки смысл написанного – некоторый аналог лингвистической хеш-функции – короткую фразу на естественном языке (может быть, и не содержащуюся в исходном тексте), отражающую главную часть содержания текста.
3. Продолжить написанный текст дальше или сгенерировать новый текст на ту же тему.

Обратимся пока к последней задаче, так как в ней нам не нужно будет обращаться к смыслу – чтобы **продолжать** писать (говорить) о чем-то необязательно понимать смысл этого. Главное – это поддерживать этот смысл и составлять предложения, не лишенные логики. Чтобы это делать, может быть достаточно пользоваться теми же конструкциями, которые уже присутствуют в тексте. Конструкции из двух слов порождают *биграммные* модели, из трёх слов – *триграммные* модели, и так далее. Под следующим заголовком можно найти ответ на вопрос о том, почему не бывает (бессмысленны) стограммные модели.

***N*-граммные модели языка**. Допустим у вас есть 10 интернет-ресурсов, на которых вы хотите размещать одни и те же по смыслу посты, но пишет их вам один копирайтер. Он категорически отказывается писать один пост в 10 разных вариантах (за те же деньги), и вы решаете, что, имея один текст, все остальные сможет сгенерировать машина. Вроде бы это должно быть легко, но давайте посмотрим, как это можно сделать конкретно. Естественный язык (и русский тоже) состоит из предложений. В первую очередь, надо выделить их в исходном тексте. Спарсим какой-нибудь текст и разделим его на предложения в первом приближении.

```
from bs4 import BeautifulSoup as BS
import requests
url = 'https://habr.com/ru/post/352812/'
html = requests.get(url).text
soup = BS(html, 'html5lib')
content = soup.find('div', 'article-formatted-body article-
formatted-body_version-1')
regex = r"[\w']+|[\.]"
words = re.findall(regex, content.text)
```

Можно, конечно, пойти дальше и почистить данные хорошо – так чтобы остались только нормальные предложения, а не «предложения» типа «т.» и «е.». Но это не добавит понимания *N*-граммных моделей. Поэтому – сразу к делу.

Выберем первое слово – это будет слово, которое начинает весь наш текст, или просто следует за точкой. Найдем слова, которые следуют за этим выбранным словом в тексте. Выберем случайно из них одно и поставим после выбранного. Теперь повторим процесс с этим, только что поставленным словом. Так получается *биграммная модель* – конструкция, созданная на основе двух слов, участвующих в каждом цикле.

Триграммная модель строится аналогичным образом. Выбираем первое слово, затем находим к нему второе, как и раньше. А третье слово ищем по критерию того, что в тексте оно стоит после уже зафиксированных первых двух. Таким образом, мы всё время ищем тройки слов. Поэтому модель триграммная.

Как только построение N-граммной модели встречает точку после очередного слова, построение начинается снова. Поскольку в предложениях обычно не очень много слов, нет смысла делать N большой величиной – тогда процесс будет заканчиваться, так и не достигнув значения N. Поэтому нет стограммных моделей.

Реализуем для примера триграммную модель в виде функции, принимающей на вход URL, по которому расположен сайт с исходным текстом, тег, в котором надо искать текст, и строку задания атрибута этого тега. В качестве начального слова возьмем любое (поэтому разные запуски дадут разный результат).

```
def trigram_writer(url, tag, attr_key_value):
    html = requests.get(url).text
    soup = BS(html, 'html5lib')
    content = soup.find(tag, exec(attr_key_value))
    regex = r"[\w']+|[\.]"
    words = re.findall(regex, content.text)
    trigrams = zip(words, words[1:], words[2:])
    trans = defaultdict(list)
    init = []
    for first, second, third in trigrams:
        if first == '.':
            init.append(second)
        trans[(first, second)].append(third)
    second = random.choice(init)
    first = '.'
    result = [second]
    while True:
        candidates = trans[(first, second)]
        next_word = random.choice(candidates)
        first, second = second, next_word
        result.append(second)
        if second == '.':
            return " ".join(result)
```

В результате, например, с указанными аргументами, получим:

```
trigram_writer('https://habr.com/ru/post/352812/', 'div',
'xmlns="http://www.w3.org/1999/xhtml"')
```

```
“ Вопросы эффективного сведения практической зад
ачи к задаче анализа данных подбора списка фичей
или препроцессинга модели и ее параметров а такж
е зафиксировать необходимую терминологию в том ч
то большинство алгоритмов машинного обучения чем
моя .”
```

Как видим, фраза получилась довольно осмысленная, хотя и явно машинная. Этим обычно отличаются N-граммные модели: чтобы получилось что-то полезное, придется довольно продолжительно писать код и все время следить за результатом. То есть, автоматического решения задачи они не дают – только автоматизированное.

Упражнения

5.2.1. Сгенерируйте текст о Data Science, пользуясь любым источником достаточного объема и триграммной моделью.

5.2.2. Выполните предыдущее упражнение с использованием текста с длинными предложениями и 4-граммной модели. Оцените видимые изменения.

Заключение

На этом обзорное изучение методов Data Science можно считать законченным. Теперь термины вроде «бутстрапирование» не будут вызывать шока, а «наивный Байес» не будет прозвищем. В то же время, для того чтобы стать аналитиком данных, требуется еще много усилий. В частности, в триаде «математика – программирование – предметная область», мы продвинулись совсем немного в программировании на Python и в самых начальных математических концепциях. В предметной области вообще никуда не продвинулись. Для профессиональной деятельности изученного материала не то, чтобы мало, – он в таком виде вообще не поможет. Для решения реальных практических задач потребуется углубиться в предметную область и потом, основываясь на этих полученных обзорных знаниях, выбрать – что именно вам нужно детально изучить для решения прикладных задач. Вот для того, чтобы вы смогли выбрать, и был предназначен этот курс.

Если вы думаете, что вам не придется становиться аналитиком данных, то просто подумайте о том, что в начале XX века многие (почти все) думали, что им никогда не придется водить автомобиль. Тем не менее, к концу того века, водительское удостоверение с успехом заменяло паспорт в США, то есть, было распространено практически поголовно. То есть, не имеет значения в какой области вы собираетесь работать. Имеет значение только то, что если вы собираетесь применять свое высшее образование в принципе, то анализом данных вам все равно придется заниматься.

Приложение: Методические указания к лабораторным работам

Лабораторная работа №1. Градиентный спуск

Целью работы является выработка умений строить программную реализацию алгоритма градиентного спуска и стохастического градиентного спуска, полученных при изучении параграфа «Оптимизация градиентным спуском» главы «Математические основы анализа данных».

Для успешного выполнения работы необходимы функции `gradient`, `gradient_descent`, `minimize_stochastic`, определенные в тексте параграфа, созданные по заданию функции `f` и `F`, полностью импортированный модуль `dsmltf` а также знание математического анализа и умения, выработанные при решении задач.

Задание на работу:

1. Сгенерируйте ряд из 500 значений `x[i+2] = x[i+1]*(2+dt*L*(1-x[i]**2))-x[i]*(1+dt**2+dt*L*(1-x[i]**2))+dt**2*sin(omega*t)`, где `x[0] = 0, x[1] = (-1)**k*dt, L=k/100, omega = 1000/k, dt = 2*pi/1000, k` - ваш номер в журнале.
2. Методом градиентного спуска подберите номера (частоты) и коэффициенты разложения Фурье из двух гармоник (пять параметров), аппроксимирующего функцию `x[i]`.
3. Сделайте то же самое методом стохастического градиентного спуска.
4. Сравните времена работы алгоритмов для одной и той же заданной точности.
5. Сделайте выводы и оформите результаты работы в отчет.

Лабораторная работа №2. Парсинг

Целью работы является выработка умений практического применения теоретических знаний, полученных при изучении параграфа «Извлечение данных из веб-ресурсов» главы «Сбор и обработка данных».

Для успешного выполнения работы необходимы знания requests, BeautifulSoup, pandas.

Задание на работу:
1. Выберите на сайте объявлений интересующую вас рубрику самого нижнего уровня (например, Для дома и дачи – Мебель и интерьер – Кровати, диваны и кресла – Диваны).
2. Убедитесь в том, что сайт учел в выдаче ваше местоположение (если нет – укажите его).
3. Обратите внимание на адрес полученной вами страницы и на то, как он изменяется при переходе к следующей странице выдачи (где и как указывается номер страницы), зафиксируйте это.
4. Откройте код страницы выдачи (F12 или правой кнопкой мыши). Выберите ключевые элементы блока записи об одном объявлении так, чтобы вы могли выделить этот блок из всего текста: например, это может быть определенный тег, с определенным набором атрибутов.
5. Напишите скрипт, который перебирает все страницы выдачи, и собирает в список следующие параметры: id объявления, название, цена, срок публикации.
6. Проверьте полученный список на корректность – в нем не должно быть посторонних записей и выбросов (слишком малых, слишком больших и отсутствующих цен). При необходимости, устраните ненужные записи программно.
7. Отфильтруйте полученный список по заданному вами критерию: например, оставьте только цены между заданными вами значениями, или оставьте только объявления, находящие в просмотре не больше суток.
8. Сохраните результат вашей работы так, чтобы он был пригоден для машинного анализа.
9. Сделайте вывод о работе, указав встретившиеся сложности и то, как они были преодолены. Оформите отчет по шаблону.

Лабораторная работа №3. Метод *k* ближайших соседей

Целью работы является выработка умений практического применения теоретических знаний, полученных при изучении параграфа «Простые прикладные модели» главы «Сбор и обработка данных».

Для успешного выполнения работы необходимы функции `majority_vote` и `knn_classify`, определенные в тексте параграфа,

а также умения, выработанные в предыдущих работах и при решении задач.

Задание на работу:
1. Найдите в открытом доступе в интернет датасеты о происшедших землетрясениях.
2. Определите регион прогнозирования, ограничивая диапазоны широты и долготы.
3. Спарсите данные о землетрясениях в регионе в датасет.
4. Определите точку, в которой ожидается следующее землетрясение, просто указав ее координаты.
5. Округляя магнитуды до 1 знака после запятой, методом k ближайших соседей, найдите магнитуду землетрясения в выбранной точке.
6. Выполните п. 5, округляя магнитуду до целого числа.
7. Сравните полученные результаты и сделайте выводы. Оформите отчет о работе.

Лабораторная работа №4. Наивный байесовский классификатор

Целью работы является выработка умений практического применения теоретических знаний, полученных при изучении параграфа «Простые прикладные модели» главы «Сбор и обработка данных».

Для успешного выполнения работы необходимы функции `tokenize`, `count_words`, `word_probabilities` и `spam_probability`, определенные в тексте параграфа, а также умения, выработанные в предыдущих работах и при решении задач.

Задание на работу:
1. Спарсите спам из ваших почтовых ящиков в текстовые файлы.
2. Выберите наиболее часто встречающие слова в спаме. Длина слова - не менее 5 букв. Из полученного списка удалите прилагательные (используйте модуль `nltk`). Выберите из оставшихся 5-7 слов – они будут индикаторами.
3. Напишите классификатор спама на основе выбранных слов-индикаторов. Используйте наивный байесовский метод.
4. Создайте 5 сообщений (текстов) и проверьте их на спамность вашим классификатором.

5. Повторите пункты 3 и 4, применяя сглаживание в методе. При этом добавьте два слова, которые не встречались в спам-сообщениях, и в индикаторы, и в два тестовых сообщения.
6. Сравните результаты и сделайте выводы. Оформите работу по шаблону.

Лабораторная работа №5. Регрессионный анализ

Целью работы является выработка умений практического применения теоретических знаний, полученных при изучении параграфа «Регрессионный анализ» главы «Модели, методы и алгоритмы работы с данными».

Для успешного выполнения работы необходимы функции, рассмотренные в параграфе 1 главы 4 настоящего пособия, а также умения, выработанные в предыдущих работах и при решении задач.

Задание на работу:
1. Составьте датасет, признаки обстоятельств в котором: месяц, погода, вид рыбы, снасть; признак статуса: рыбалка удачна (1) или неудачна (0). Множества возможных значений признаков обстоятельств:
месяц - от 1 до 12
погода - солнце и штиль (0), солнце и ветер (1), облачно и штиль (2), облачно и ветер (3), дождь и штиль (4), дождь и ветер (5)
вид рыбы - окунь (1), карась (2), судак (3), плотва (4) и налим (5)
снасть - удочка (1), донка (2), спиннинг (3)
Включите не менее 30 записей в датасет (по примерному числу рыбалок в год). Если рыбалка не ваше хобби – создайте аналогичный по форме датасет на основе своих знаний в хорошо знакомой вам области.
2. Прошкалируйте данные
3. Составьте гипотезу зависимости удачной рыбалки (или успеха в вашей области) от указанных обстоятельств
4. Разбейте данные на обучающую и тестовую выборки
5. Методом логистической регрессии постройте предсказание успешного исхода для заданного набора обстоятельств.
6. Оцените точность предсказания и запишите вывод.

Лабораторная работа №6. Простая распознающая нейронная сеть

Целью работы является выработка умений практического применения теоретических знаний, полученных при изучении

параграфа «Нейронные сети» главы «Модели, методы и алгоритмы работы с данными».
Для успешного выполнения работы необходимы функции `art_nn, forward, layer_train, train_step,` определенные в тексте параграфа, а также умения, выработанные в предыдущих работах и при решении задач.

Задание на работу:
1. Разработайте и реализуйте ИНС для распознавания цифр, написанных в квадрате символов размером 5x5.
2. Достройте еще один слой в сеть. Расширьте поле ввода до 10x10, обучите на хорошо изображенных цифрах. В изображениях цифр могут быть вариации - поле уже достаточно большое.
3. Введите нестрого изображенные цифры (с дефектами, кривые, смещенные и т.п.). Выдавайте одну распознанную цифру (по максимальной вероятности). Длина выборки - 10.
4. Вычислите количество ошибочно распознанных и посчитайте относительную ошибку.
5. Увеличьте обучающую выборку и повторите п.4. Уменьшилась ли ошибка?
6. Интерпретируйте все результаты, сделайте выводы и оформите отчет о работе.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Грас Дж. Data Science. Наука о данных с нуля: Пер. с англ. - 2-е изд., перераб. и доп. - СПб.: БХВ-Петербург, 2021. - 416 с.: ил.
2. Маккинни Уэс. Python и анализ данных: Пер. с англ. - М.: ДМК-Пресс, 2015. - 484 с.: ил.
3. Лесковец, Ю. Анализ больших наборов данных / Ю. Лесковец, А. Раджараман. — М.: ДМК, 2016. — 498 с.
4. Шашков В.Б. Прикладной регрессионный анализ. Многофакторная регрессия: Учебное пособие.- Оренбург: ГОУ ВПО ОГУ, 2003. – 363 с.
5. Ростовцев В.С. Искусственные нейронные сети: учебник / В.С. Ростовцев. – Киров: Изд-во ВятГУ, 2014. – 208 с.
6. Гитис Л.Х. Кластерный анализ в задачах классификации, оптимизации и прогнозирования. / Л.Х. Гитис. – М: Изд-во МГГУ, 2001. – 104 с.
7. Воронцов К.В. Вероятностное тематическое моделирование: теория, модели, алгоритмы и проект BigARTM. / К.В. Воронцов. – М. Изд-во МФТИ, 2021. – 107 с.