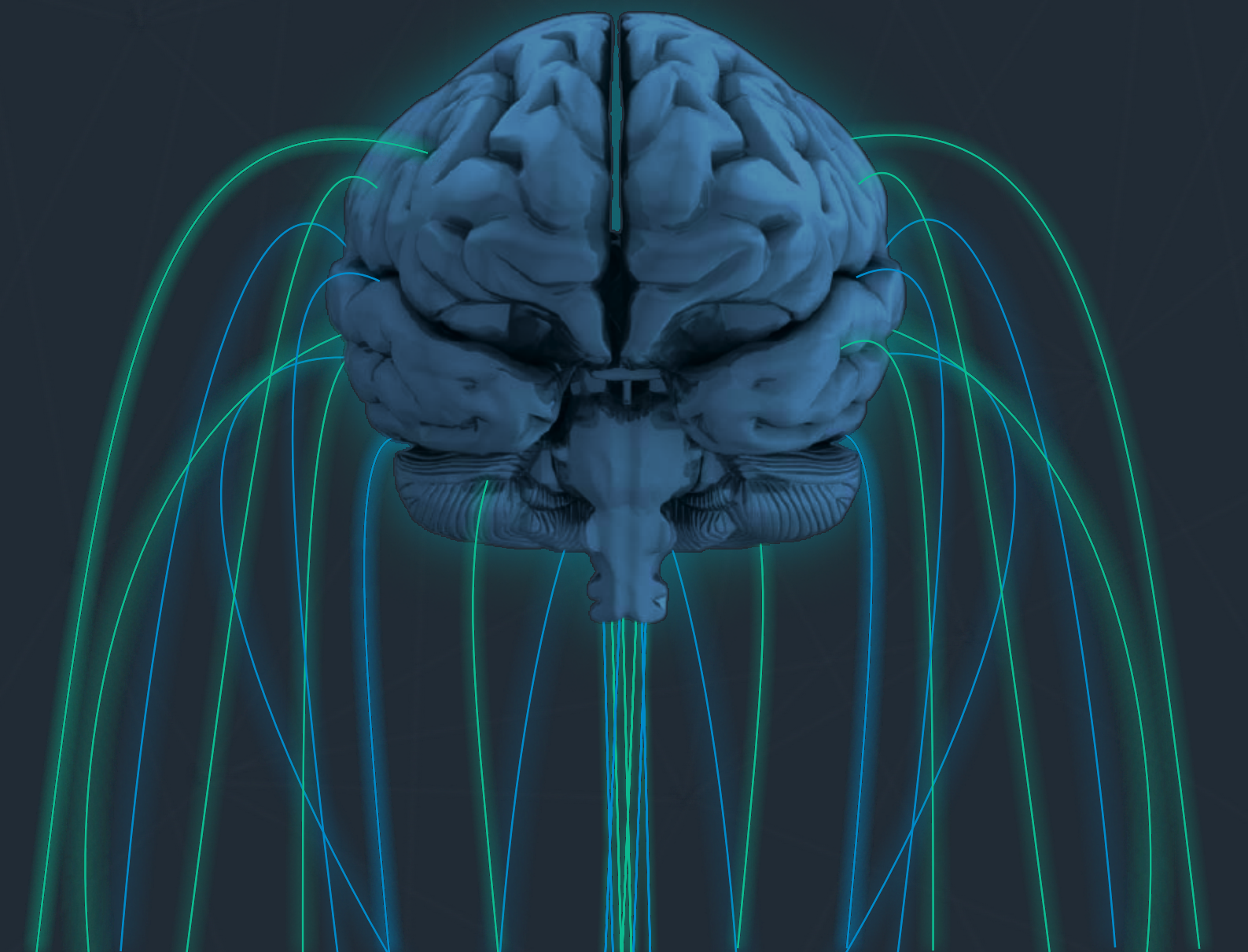
Sigmoid

Machine Learning Handbook



ML02

Păpăluță Vasile Vladimir Stojoc Răzvan Fișer
Marius Purici Covalevschi Andreea Smocvin Denis
Graur Elena Balamatiuc Eduard Andronovici Darinela
Clefos Alexandru

October 20, 2022

Contents

1 Preface. **7**
- 1.1 What is Machine Learning? . . . 7
- 1.2 What is this book about? . . . 7
- 1.3 Big thanks to the authors. . . . 8

2 Python Object Oriented Programming **9**
- 2.1 What are classes and objects? . . . 10
- 2.2 What are the key concepts of the OOP? . . . 13
  - 2.2.1 Encapsulation. . . . 13
  - 2.2.2 Inheritance. . . . 15
  - 2.2.3 Polymorphism. . . . 16
- 2.3 Conclusion. . . . 19

3 Statistical Analysis. Pandas. **20**
- 3.1 What is Pandas. . . . 20
  - 3.1.1 Series. . . . 20
  - 3.1.2 pandas Data Frame. . . . 22
- 3.2 Pandas on a real data set. . . . 25
- 3.3 Conclusion. . . . 30

4 Introduction into Linear Algebra. Numpy library. **31**
- 4.1 Operations on vectors and matrices. . . . 32
- 4.2 Numpy . . . 34
- 4.3 Conclusion . . . 48

5 Linear Regression **49**
- 5.1 The setup: . . . 51
  - 5.1.1 The Generalization of the problem: . . . 52
  - 5.1.2 The Cost Function: . . . 52
  - 5.1.3 Gradient Descent: . . . 53
- 5.2 Implementation: . . . 55
- 5.3 Conclusions: . . . 58

6 Logistic Regression **59**
- 6.1 The Logistic Function: . . . 59
- 6.2 The Cost Function: . . . 61
- 6.3 Implementation: . . . 62
- 6.4 Conclusions: . . . 65

7 Principal Component Analysis **66**
7.1 PCA algorithm: eigenvectors and eigenvalues: 67
7.2 PCA algorithm: singular value decomposition: 70
7.3 The implementation of PCA from scratch: 71
7.4 The implementation of PCA from sklearn: 73
7.5 Conclusion: 74

8 Probability. Naive Bayes **75**
8.1 A first definition of probability: 75
8.2 General Definition of probability: 75
8.3 Basic Theorems of Probability: 76
8.4 Conditional Probability: 76
8.5 Combinatorics: 77
8.6 Random Variables: 78
8.7 Discrete Random Variables and Distributions: 79
8.8 Continuous Random Variables and Distributions: 79
8.9 Mean and Variance of a distribution: 79
8.10 Naive Bayes: 80
8.11 Conclusion: 85

9 Random Forest **86**
9.1 How it works 87
9.2 Random Forest Classifier 88
9.2.1 Implementation 88
9.2.2 Finding Important Features in Scikit-learn 90
9.2.3 Generating the Model on Selected Features 92
9.3 Random Forest Regressor 93
9.4 Conclusions: 95

10 Support Vector Machine **96**
10.1 Finding the right hyperplane 97
10.2 Lagrange Multipliers 99
10.3 Implementation 99
10.4 Conclusion 103

11 Missing Values **104**
11.1 KNNI 113
11.2 MICE 114
11.3 Reparo 115
11.4 Conclusion: 116

12 Hyperparameter tuning. 117
12.1 How to tune my parameters? . . . 118
12.1.1 Grid search. . . . 118
12.1.2 Random Search. . . . 119
12.1.3 Underfitting and Overfitting. . . . 120
12.2 Conclusion. . . . 121

13 Gaussian Mixture Models. EM Algorithm 122
13.1 A probabilistic approach on clustering . . . 122
13.2 EM-algorithm. . . . 124
13.3 The scikit-learn implementation . . . 132
13.4 Conclusion . . . 137

14 KMeans 138
14.1 Silhouette Score . . . 144
14.2 Marketing Segmentation: . . . 148
14.3 Conclusion . . . 151

15 Data Visualisation 152
15.1 What is matplotlib? . . . 152
15.1.1 MATLAB-style interface . . . 152
15.1.2 Object oriented interface . . . 153
15.2 How to make simple line plots? . . . 154
15.3 How to make simple scatter plots? . . . 155
15.4 Error bars . . . 157
15.5 Contour plots . . . 158
15.6 Histograms . . . 159
15.7 Why you might also want to learn seaborn . . . 161
15.7.1 KDE plots . . . 161
15.7.2 jointplot . . . 163
15.7.3 boxplot . . . 164
15.7.4 Violinplot . . . 165
15.7.5 lmplot . . . 166

16 Outliers 167
16.1 Outlier Detection Methods. . . . 168
16.1.1 Isolation Forest. . . . 168
16.1.2 Elliptic Envelope. . . . 169
16.1.3 Local Outlier Factor. . . . 169
16.1.4 One-Class SVM. . . . 170

16.1.5 Box-Plot. 171
16.2 Conclusion: 172

17 Feature Engineering **173**
17.1 Classical Methods. 173
17.1.1 Feature Scaling. 173
17.1.2 Dummy Variable. 176
17.2 Advanced Imperio Methods: Numerical features. 178
17.2.1 Box-Cox Transformer. 179
17.2.2 ZCA Transformation. 181
17.3 Advanced Imperio Methods: Categorical features. 182
17.3.1 Frequency Imputation. 183
17.3.2 Target Imputation. 185
17.4 Conclusion. 187

18 Feature Selection **188**
18.1 Regression Feature Selection. 189
18.1.1 P-value-based feature selection. 189
18.1.2 LASSO Feature Selection. 191
18.2 Classification Feature Selection. 193
18.2.1 Correlation Feature Selection. 193
18.2.2 Point-Biserial Feature Selection. 194
18.3 Conclusion. 195

19 Class Balancing. **196**
19.1 Classical Methods. 198
19.1.1 Undersampling. 198
19.1.2 Class weights. 198
19.2 Oversampling. 199
19.2.1 SMOTE. 199
19.2.2 SMOTEENN. 202
19.2.3 ICOTE. 203
19.2.4 TKRKNN. 205
19.3 Conclusion. 206

20 Model interpretation. **207**
20.1 Classical Methods. 207
20.1.1 Linear Models interpretability. 208
20.1.2 CART interpretability. 209
20.2 Other ways to interpret a model. 211

20.2.1 Classification interpretability. 211
20.2.2 Regression interpretability. 213
20.2.3 Agnostic methods. 214
20.3 LIME interpretability. 215
20.3.1 SHAP interpretability. 217
20.4 Conclusion. 225

21 Streamlit **226**
21.1 What is streamlit? 226
21.2 The theory behind the product 226
21.3 How streamlit works? 227
21.4 Main commands 227
21.5 Conclusion 237

22 Memory optimization. Dask **238**
22.1 What is Dask? 238
22.2 Dask Array 239
22.3 Dask DataFrame 241
22.4 Conclusion 248

1 Preface.

1.1 What is Machine Learning?

Machine Learning is the subfield of Artificial Intelligence that tries to give computers the ability to learn from data. There is a Romanian quote:

> The man learns while he lives

We as an intelligent species learn from every experience we pass through. In the beginning, we learn to walk. Usually, babies walk on four legs. With time they learn to walk only on two legs. However, it never happens immediately; they fall, feel pain, stand up and continue to walk until they learn to walk like adults. This principle applies to everything: solving an equation at school, learning to program, cooking, or any other activity.

Machine Learning is trying to copy this principle of learning in computers using algorithms and principles from mathematics. However, for computers, the so-named experiences are actual data. They learn from tons of data to predict what a user would buy from an e-commerce website, what mood you have based on the comment that you wrote on Facebook or the image that you posted on Instagram, will you leave, or not the company that you are working on and so on. These are just a few use cases that Machine Learning tries to solve.

1.2 What is this book about?

This book was written by the Sigmoid team as an introduction to the Machine Learning field. It begins with the main tools that Machine Learning Engineers and Data scientists use in their day-to-day work, such as NumPy, Pandas, Matplotlib, Seaborn, and Sklearn. After it explains the mathematics behind Machine Learning Algorithms, followed by their implementation. Finally, it introduces the reader to such Machine Learning-related topics as Feature Engineer, Feature Selection, Class Balancing, Model Interpretation, and Hypermater Tunning. During this book, you will discover Sigmoid's python libraries created for research purposes by the technical department - Kydavra, Crucio, Imperio, and Reparo.

This book doesn't require advanced knowledge of python; however, the understanding of python data structures and python syntax will be a great plus.

Also, this book is just an introduction to this field. We highly encourage you to consult other sources while reading, especially after finishing this book. Also, you can check out our website www.sigmoidai.org here you can find links to our social media pages like Instagram, Youtube and so on!

1.3 Big thanks to the authors.

This book was written by four people from the technical department of Sigmoid: **Păpăluță Vasile**, **Stojoc Vladimir**, **Răzvan Fișer**, **Purici Marius**, **Covalevschi Andreea**, **Smocvin Denis**, **Graur Elena**, **Balamatiuc Eduard**, **Andronovici Darinela** and **Clefos Alexandru** who wrote the technical content of these topics. Also, the graphical design of the book was made by **Steț̦enco Iuliana** and **Sireț̦anu Andreia-Cristina**. Thank you for your effort during the development of this book; it would be impossible without your help and implication.

2 Python Object Oriented Programming

Object-Oriented Programming (OOP) is a higher-level programming paradigm that most programming projects rely on. OOP is based on the concept of 'objects', which can contain different types of valuable data, and 'classes' which are reusable pieces of code blueprints used to make these objects. There are a lot of Object-Oriented Programming languages, and Python is one of them.

Here is an example of why we need Object-Oriented programming. Suppose you work as a Python database administrator for a school. Your primary task is saving and storing information about all school students, including their names, surnames, grades, and teacher's comments.

If you know basic Python, you might understand what data types we would need for every piece of information enumerated earlier. For names, surnames and comments, we need strings. For grades, we need ints (if our grade system is from 1 to 10) or floats (average grades for a specific subject - 9.57) or even strings (A-F) depending on the grading system that the school uses. For our example, we will keep it simple and use ints, but we need a list of ints because we won't have only one grade. Also, because we don't have only one subject, we would need different lists for every school subject; thus, it would be better with dictionaries where we will store a list of grades for every subject.

This seems okay to me, but what's the problem with the basic Python, and why can't we store data in the way we just described? Here comes the generalization problem, we have to store information about hundreds or even thousands of students, and if we were to do it with basic Python, we would get this:

```
Student_1_name = 'James'
Student_1_surname = 'Rock'
Student_1_grades = {'Math':[6,7,8],'Science':[9,8,9]...}
Student_1_comments = 'Good student'
Student_2_name = 'Emma'
...
Student_1000_comments = 'Not bad student'
```

As you can imagine, there would be (Number of features * Number of students) rows, in this example 4 * 1000 rows, and that's not the biggest problem. This job is not automated, so we will have to do it all manually for each student; the result can be tough to read and easy to mess up. Here it comes, OOP - classes, objects, and relations between them.

2.1 What are classes and objects?

Classes are user-defined data types, blueprints for structures you want to describe.

Objects are instances of these classes.

Figure 1: Car example.

In the example above, we have a class (blueprint) named Car, and different objects, instances of the Car class. You can imagine this as Class Car being the concept of a real car and objects being different types of cars from different companies with different components.

The same thing we're doing with our problem. We will create a Student class representing the general concept of a student from our school, and then we will create instances for every student.

```
class Student:
    def __init__(self, name, surname, grades, comments):
        self.name = name
        self.surname = surname
        self.grades = grades
        self.comments = comments
```

Let's analyze it a bit. To define a class, in Python, we have the keyword 'class', after which we should specify the name of the class, which can be whatever you want. The only conditions are that it should be a sequence of letters without spaces between them, and as any variable, it can't start with a number. After that, we write ':' to show that we have begun to define our class.

Next, we have to define the constructor of our class. What is a constructor? The constructor is a reserved method in python classes that helps us initialize our objects. This example shows that our constructor `__init__` accepts five parameters. The first one is always self. Self represents the instance of the class. In Python, to access the attributes and methods of the class, you can use the "self" keyword. It binds the attributes with the given parameters.

Now, we initialize the self object with all the values given to the constructor. The parameter' name' of the object gets the parameter 'name' value that we sent as input to constructor, and so on for each parameter. The names of the variables don't matter. You should remember that the variables from the left are different from the right ones. The left variables are the parameters of the object we are creating, the parameters of the object we will access later with the help of the 'self'. The variables on the right are the ones we gave to the constructor. In this case, we set the student's name as the variable 'name' we gave to the constructor.

```
grades = {'Informatics':[7,9,8],'Math':[7,8,9]}
student1 = Student('Alex','Pierce', grades, 'Good student')
```

In that way, we can initialize our students without creating millions of variables but with just an object for every student. Also, as you may notice, we didn't send the self argument because the self will always be the object you are currently working on, and Python understands it. Now, we can access the information that is stored inside this object.

```
In:
print(student1.name)
print(student1.surname)
print(student1.grades)
print(student1.comments)

Out:
Alex
Pierce
{'Informatics': [7, 9, 8], 'Math': [7, 8, 9]}
Good student
```

Also, we can change the data of the objects that we create:

```
In:
grades = {'Informatics':[7,9,8],'Math':[7,8,9]}
student1 = Student('Alex','Pierce', grades, 'Good student')
student1.name = 'Richard'
print(student1.name)

Out:
Richard
```

There is not much meaning in data if we can't do something with it. Keeping with this example, usually, we would need a student's final

grade, and we have to calculate it, but instead of doing it by ourselves, we will give this task to our class. We will add a method that can be called from every student and calculate the average grade for the student from whom the method was called.

```
class Student:
    def __init__(self, name, surname, grades, comments):
        self.name = name
        self.surname = surname
        self.grades = grades
        self.comment = comments

    def average_grade(self):
        avg = 0
        for value in self.grades.values():
            avg+= sum(val)/len(val)
        return avg/len(self.grades.keys())

In:
grades = {'Informatics':[7,9,8],'Math':[7,8,9]}
student1 = Student('Alex','Pierce', grades, 'Good student')
print(student1.average_grade())

Out:
8.0
```

Here, we added a new method to our class and named it `'average_grade'`. As with every class method, it has the parameter self and nothing more. We don't have any other parameters because all that we need is already stored in the self object, so we need only it. Inside the method, we calculate the final grade for the student object from which we call the method. In the example above, we call `average_grade` from the student1 object. The average grade is also calculated using the grades of student1 initialized earlier with the constructor. Of course, there can be many different possibilities for how we can define this method. For instance, we could save the average grade in the new parameter of the object - `self.average_grade`, and then use it every time we need it without the need to recalculate it every time.

2.2 What are the key concepts of the OOP?

Three OOP concepts in Python are:

- Encapsulation.
- Inheritance.
- Polymorphism.

2.2.1 Encapsulation.

Encapsulation means containing all important information inside an object (like all the medicine inside a pill) and only exposing selected information to the outside world. **Encapsulation** hides the internal software code implementation inside a class and internal data inside objects.

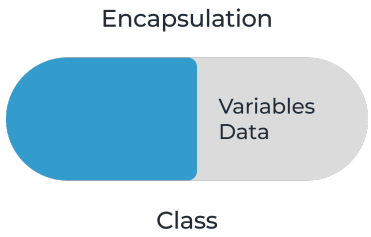


Figure 2: Encapsulation visually explained.

Encapsulation requires defining some fields as private and some as public.

- **Private**: methods and properties, accessible from other methods of the same class.
- **Public**: methods and properties, accessible from inside and outside the class.

Encapsulation adds security. Public methods and properties are used to access or update data. Attributes and methods can be private, so they can't be accessed outside the class. The protected level we will discuss in the Inheritance section.

In Python, we indicate the security level of our data and methods with the help of underscores:

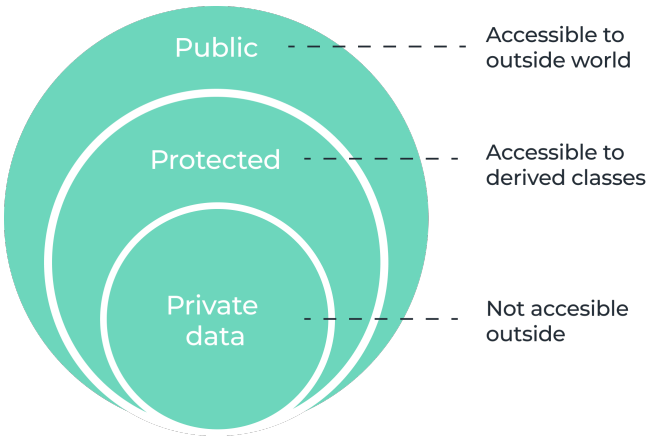


Figure 3: The access rights levels.

```
class Student:
    def __init__(self, name, surname, grades, comments):
        self.__name = name
        self.surname = surname
        self.grades = grades
        self.comment = comments

    def average_grade(self):
        avg = 0
        for value in self.grades.values():
            avg+= sum(val)/len(val)
        return avg/len(self.grades.keys())

    def print_name(self):
        print(self.__name)
```

`self.__name` - two underscores before parameter 'name' means that we set a student's name to be a private parameter, it cannot be accessed from the outside. We as well could make a private method. In that way we will not be able to call it from the outside but will be able to call it from every other method from the inside of the class.

```
In:
grades = {'Informatics':[7,9,8],'Math':[7,8,9]}
student1 = Student('Alex','Pierce', grades, 'Good student')
print(student1.name)

Out:
AttributeError: 'Student' object has no attribute 'name'
```

```
student1.print_name()

Output:
Alex
```

2.2.2 Inheritance.

Inheritance allows classes to inherit attributes and methods of other classes. Parent classes extend attributes and behaviors to child classes.

Basic attributes and behaviors are usually defined in a parent class. The child class can be created, extending the functionality of the parent class by adding additional attributes and behaviors.

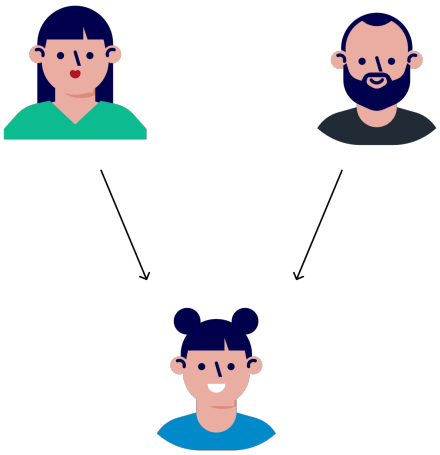


Figure 4: Inheritance example.

```
class Human:

    def __init__(self,name,surname):
        self.name = name
        self.surname = surname

    def sleeping(self):
        print("I'm sleeping")


class Student(Human):
    def __init__(self, name, surname, grades, comments):
```

```
        super().__init__(name,surname)
        self.grades = grades
        self.comment = comments

    def average_grade(self):
        avg = 0
        for value in self.grades.values():
            avg+= sum(val)/len(val)
        return avg/len(self.grades.keys())

In:
human1 = Human('Alex', 'Pierce')
human1.sleeping()

grades = {'Informatics':[7,9,8],'Math':[7,8,9]}
student1 = Student('Alex','Pierce', grades, 'Good student')
student1.sleeping()

Out:
I'm sleeping
I'm sleeping
```

In Python, we make the **Inheritance** of classes when defining the class name, class `Student(Inherited class)`.

In the example above, we initialized an instance of the Human class and one instance of the Student class. You may notice that the Human class has the function called 'sleeping' which prints "I'm sleeping", but because of **Inheritance** we as well can call it for instances of the child classes even if we didn't define 'sleeping' function for the Student class. In that way, we easily called the sleeping() function for the student1 object.

To complete the **Encapsulation** concept, private and public are not the only states that variables and methods can have. Protected is another security level that makes data private but also gives access to data to the derived (child) classes. In Python, it is done with a single underscore.

2.2.3 Polymorphism.

Polymorphism means designing objects to share behaviors. Using **Polymorphism**, objects can override shared parent behaviors with specific child behaviors. **Polymorphism** allows the same method to ex-

ecute different behaviors in two ways: method overriding and method overloading.

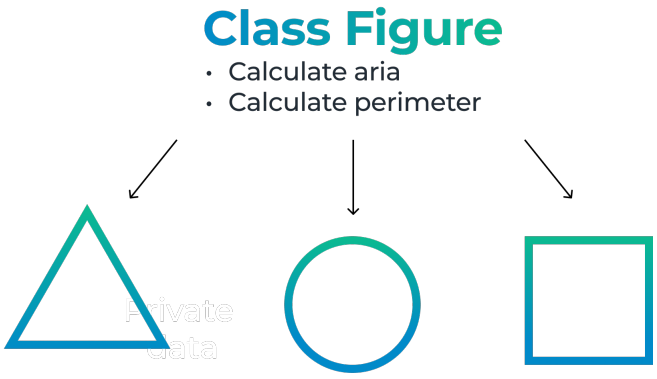


Figure 5: Polymorphism.

Method Overriding

```
class Human:
    def __init__(self,name,surname):
        self.name = name
        self.surname = surname

    def sleeping(self):
        print("I'm sleeping")

class Student(Human):
    def __init__(self, name, surname, grades, comments):
        super().__init__(name,surname)
        self.grades = grades
        self.comment = comments

    def average_grade(self):
        avg = 0
        for value in self.grades.values():
            avg+= sum(val)/len(val)
        return avg/len(self.grades.keys())

    def sleeping(self):
        print("Student is sleeping")
```

In this example, the Student class inherits the 'sleeping' function from parent class, but in this case, we want to change it, to override it, so we

just defined the same 'sleeping' function in the child class but with some changes. Now Student objects will print "Student is sleeping".

```
In:
human1 = Human('Tom', 'Cruise')
grades = {'Informatics':[7,9,8],'Math':[7,8,9]}
student1 = Student('Alex','Pierce', grades, 'Good student')

human1.sleeping()
student1.sleeping()

Out:
I'm sleeping
Student is sleeping  #sleeping() method was overridden
```

Method Overloading

Method overloading helps us call methods with the same name for different instances of different classes.

```
class Student:
    def __init__(self, name, surname, grades, comments):
        self.name = name
        self.surname = surname
        self.grades = grades
        self.comment = comments

    def average_grade(self):
        avg = 0
        for value in self.grades.values():
            avg+= sum(val)/len(val)
        return avg/len(self.grades.keys())

    def sleeping(self):
        print("Student is sleeping")

class Teacher:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def sleeping(self):
        print("Teacher is sleeping")
```

```
In:
student1 = Student('Alex','Pierce', grades, 'Good student')
teacher1 = Teacher('Tom', 'Cruise')

for obj in (student1,teacher1):
    obj.sleeping()

Out:
Student is sleeping
Teacher is sleeping
```

In this example, we have two different classes that both have 'sleeping' methods, and with the help of a 'for' cycle, we iterate through 2 objects and can call sleeping for both instances of different classes.

2.3 Conclusion.

Object Oriented programming requires thinking about the program's structure and planning at the beginning of coding and looking at how to break up the requirements into simple, reusable classes that can be used as blueprint instances of objects. OOP generally allows for better data structures, making the code reusable.

3 Statistical Analysis. Pandas.

Machine Learning, as we already know, is the field that allows machines to learn from data using different algorithms. Usually, the data that we feed to these algorithms are structured in various forms. The first type of data structures you will learn are series and tables in pandas named Series and Data Frames.

Also, an essential part of Machine Learning model development is to analyze your data. In most cases, it is a statistical analysis of the data. For this, we will use pandas.

3.1 What is Pandas.

pandas is a python library for manipulating tables (a.k.a. Data Frames) in python, reading data from multiple sources of file formats, and even some data visualization methods. First, before using it, let's install it. Open the terminal (on Linux of MaxOS) or cmd (on Windows) and type the following command:

```
pip install pandas
```

3.1.1 Series.

In mathematics, a series is an operation of adding an infinite amount of quantities, one after the other. During this topic, we will name Series, a special data type from pandas library that is a data structure very similar to python lists.

First, let's import the pandas library. In industry, the alias *pd* is a standard for the pandas library:

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

Also, we imported a library named numpy. We will need it to generate random series of numbers. To install it, just run in the command prompt the following command:

```
pip install numpy
```

Now, let's generate a random list of numbers:

```
list_data = [np.random.random() for _ in range(200)]
```

pd.Series allows you to easily create numerical series from different data structures like lists and NumPy arrays:

```
In:
series = pd.Series(list_data)
print(series)
Out:
0      0.463002
1      0.035916
2      0.964136
3      0.400559
4      0.229590
         ...
195    0.263317
196    0.344290
197    0.059696
198    0.428044
199    0.611425
Length: 200, dtype: float64
```

First thing that we can do with series is, of course, to iterate through them, which we can do in two ways:

Element-wise:

```
for element in series:
    print(element)
```

Index-wise:

```
for i in range(len(series)):
    print(series[i])
```

From the example above, you may already understand that we can access individual elements from a series using indexes:

```
series[0]
```

Pandas has a lot of functionalities for statistical analysis of the data. Let's get through a couple of them. The most used functionalities are finding a numerical series's minimal and maximal values. This allows you to understand the range of the values of the Series:

```
# Computing the minimal value of a series.
min_value = series.min()

# Computing the maximal value of a series.
max_value = series.max()
```

Mean is the sum of all values in a numerical series divided by the number of values in the series and is computed by the following formula.
$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x$
In pandas it is computed using **mean** function:

```
# Computing the mean value of a series.
mean = series.mean()
```

The median is the value from the series that splits the series into two subsequences of values. If the number of elements is odd, then the median is the value from the middle of the sorted series. If the number of elements is even, it is the mean value of the two values from the middle of the sorted series. In pandas it is computed using **median** function:

```
# Computing the median value of a series.
mean = series.median()
```

Sometimes, we may need to find the most frequent value (or values) from the numerical series. This value is named in statistics **mode**, and in pandas is computed using the **mode** function.

```
# Computing the mode value of a series.
mean = series.mode()
```

In some cases, we may need to know the location of the maximal and minimal value instead of the value itself. Here we can use **argmin** and **argmax** functions. They will return the index of the minimal and maximal value from the series:

```
# Finding the index of the minimal value from the series.
min_index = series.argmin()


# Findign the index of the maximal value from the series.
max_index = series.argmax()
```

However, as we said before, machine learning allows algorithms to learn from a lot of data, meaning multi-dimensional data. As you may probably understand, series are only unidimensional data. A special data structure named **Data Frame** exists for multi-dimensional data in pandas.

3.1.2 pandas Data Frame.

Data Frames are data tables with columns and rows that allow you to explore multi-dimensional data. Let's see how to create them.

We can create pandas Data Frames from different sources, starting with a (numpy) matrix.

```
# Generating a random matrix in numpy.
matrix = np.random.random((10, 5))

# Creating the pandas Data Frame.
df = pd.DataFrame(matrix)
```

Depending on which environment you are working with pandas, when you are going to print or display a Data Frame, it will look different but will have the following structure:

1	2	3	4	5
0.557176	0.592435	0.600017	0.718592	0.527595
0.113982	0.025779	0.757001	0.831159	0.942701
0.328168	0.579004	0.508260	0.489063	0.638944
0.076249	0.310056	0.990537	0.675540	0.738506
0.838247	0.450614	0.670663	0.391809	0.848397
0.360990	0.581482	0.087030	0.788210	0.732148
0.126620	0.240486	0.856519	0.473284	0.806297
0.603966	0.836226	0.383581	0.618920	0.545895
0.212569	0.913441	0.123466	0.611467	0.375844
0.919906	0.457875	0.086425	0.767061	0.837172

Also, we can give names to columns:

```
df = pd.DataFrame(matrix, columns=['col1', 'col2', 'col3',
   'col4', 'col5'])
```

col1	col2	col3	col4	col5
0.557176	0.592435	0.600017	0.718592	0.527595
0.113982	0.025779	0.757001	0.831159	0.942701
0.328168	0.579004	0.508260	0.489063	0.638944
0.076249	0.310056	0.990537	0.675540	0.738506
0.838247	0.450614	0.670663	0.391809	0.848397
0.360990	0.581482	0.087030	0.788210	0.732148
0.126620	0.240486	0.856519	0.473284	0.806297
0.603966	0.836226	0.383581	0.618920	0.545895
0.212569	0.913441	0.123466	0.611467	0.375844
0.919906	0.457875	0.086425	0.767061	0.837172

Now that we named our columns, we can access them individually by their names as we do with dictionaries in python:

```
In:
df['col1']
Out:
0    0.557176
1    0.113982
2    0.328168
3    0.076249
4    0.838247
5    0.360998
6    0.126620
7    0.603966
8    0.212569
9    0.919906
Name: col1, dtype: float64
```

When we access a column, we get a pd.Series object. We can also access more columns:

```
df[['col1', 'col2']]
```

And the result will be the following:

col1	col2
0.557176	0.592435
0.113982	0.025779
0.328168	0.579004
0.076249	0.310056
0.838247	0.450614
0.360990	0.581482
0.126620	0.240486
0.603966	0.836226
0.212569	0.913441
0.919906	0.457875

Sometimes, we may need to get a specific value from the table. For this purpose, we may use the indexation as we do in python matrices using the .iloc property:

```
In:
df.iloc[0, 1]
```

```
Out:
0.5924353226248812
```

We can use the indexation and slicing logic in pandas too. For example, in the following listing, I'm going to show all the rows from the columns starting from the column with index 1, meaning not showing the column with index 0 and only showing the last four columns.

```
df.iloc[:, 1:]
```

sklearn library that we will learn for the creation of the Machine Learning models uses numpy arrays, so we need a way to convert pandas Data Frames into numpy arrays. For this, we can use the values property of the Data Frames and Series:

```
df.values
```

3.2 Pandas on a real data set.

In the following subsection, we will show pandas in action on a real data set - Heart Disease UCI - that can be found on Kaggle. Please download the .csv file, and let's begin to explore it. To import a data set from a .csv file, we are going to use the read_csv function from pandas:

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

Let's begin by looking at the column names that we have in our data set:

```
In:
df.columns
Out:
Index(['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs',
'restecg', 'thalach', 'exang', 'oldpeak', 'slope', 'ca',
'thal', 'target'], dtype='object')
```

columns function returns a list-like data structure with the names of all columns in the Data Frame.

A useful function for data understanding in pandas is **info**. It displays the following information:

1. The class of the data structure
2. The index range - the number of rows and the starting and ending index.

3. The number of non-missing values in every column.

4. The data type of every column.

5. The needed memory size to store this Data Frame.

Also, we can explore the statistical parameters of every column as we did with the series.

```
In:
df.mean()
Out:
age          54.366337
sex           0.683168
cp            0.966997
trestbps    131.623762
chol        246.264026
fbs           0.148515
restecg       0.528053
thalach     149.646865
exang         0.326733
oldpeak       1.039604
slope         1.399340
ca            0.729373
thal          2.313531
target        0.544554
dtype: float64
```

We can get the main statistical parameters for all columns by using just one function - **describe**:

```
In:
df.describe()
Out:
                age         sex
count  303.000000  303.000000
mean    54.366337    0.683168
std      9.082101    0.466011
min     29.000000    0.000000
25%     47.500000    0.000000
50%     55.000000    1.000000
75%     61.000000    1.000000
max     77.000000    1.000000
```

Correlation measures how similar distributions of two numerical series are. There are some types of correlation - Pearson, Kendall, Spearman, etc. The most used is the Pearson correlation. Its formula is the following:

$p_{xy} = \frac{cov(x,y)}{\sigma_x \sigma_y}$

Here cov(x, y) is the covariance between two columns, and $\sigma(x)$ is the standard deviation of a feature x. We will explore how covariance and the standard deviation are computed in more detail on the topic of PCA. Right now, it is essential to understand that correlation is a value between -1 and 1. Suppose the absolute value of the correlation is near 1. In that case, these two columns are similar, meaning that if one value grows, the second one will grow too (if the correlation is positive) and vice versa (if the correlation is negative). If the correlation is near 0, these two columns aren't related at all. Usually, a correlation table is created - a n x n table with columns presented as columns and rows at the same time, and cells of this table are the correlation of the corresponding column and row.

	col1	col2	col3
col1	corr(col1, col1)	corr(col1, col2)	corr(col1, col3)
col2	corr(col2, col1)	corr(col2, col2)	corr(col2, col3)
col3	corr(col3, col1)	corr(col3, col2)	corr(col3, col3)

To compute this table in pandas, we can use the **corr** function of the Data Frame object.

```
df.corr()
```

As you probably already understood, all columns in the Heart Disease UCI Data Frame are numbers. However, this is only at first sight. If you will look closer then you will see that 'sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal' and 'target' have a limited range of values. Values with a limited range of values are named **categorical feature** because they usually encode some categories. For example, the 'target' column encodes that the person doesn't have a heart disease using '0' and '1' to encode the fact that the person is ill. When the feature has an unbounded range of values, it is named a **numerical feature**.

Usually, when we are working with categorical features, we are interested in their frequency - how often each category is found in the series. We can find the frequencies of a categorical column using **value_counts** function:

```
In:
df['sex'].value_counts()
Out:
1    207
0     96
Name: sex, dtype: int64
```

Suppose we would like to find the mean value of cholesterol ('chol' column) for every gender. In a pythonic way, we would write this code:

```
In:
for key in dict(df['sex'].value_counts()):
    print(f"{key} - {df[df['sex']==key]['chol'].mean()}")
Out:
1 - 239.28985507246378
0 - 261.3020833333333
```

However, especially for this case, pandas has a function named **groupby** that allows us to group samples by categories.

```
df.groupby(['sex']).mean()
```

This code gives us the mean values of every feature by gender. If I return to our initial task - get the mean level of cholesterol by gender:

```
In:
df.groupby(['sex'])['chol'].mean()
Out:
sex
0    261.302083
1    239.289855
Name: chol, dtype: float64
```

Another valuable function to work with categorical features is **map**. The map allows you to use a python dictionary to map values from a column with some other values. **NOTE: If a value from the column isn't presented in this dictionary, then it will be replaced with NaN - missing value.**

Let's make the 'sex' column more intuitive and replace numbers with gender names:

```
In:
df['sex'] = df['sex'].map({0 : 'female', 1 : 'male'})
print(df['sex'])
Out:
```

```
0        male
1        male
2      female
3        male
4      female
        ...
298    female
299      male
300      male
301      male
302    female
Name: sex, Length: 303, dtype: object
```

Now let's change it back:

```
df['sex'] = df['sex'].map({'female' : 0, 'male' : 1})
```

In some cases, we may need to remove one or more columns from the data frame or add one or more new columns. To remove columns, we may use the drop function. But before that, I will create a copy of this column to add it back later.

```
# Saving a copy of the column.
age_column = df['age'].values

# Removing the 'age' column.
df = df.drop(['age'], axis=1)
```

The addition of a new column is the same procedure as with python dictionaries:

```
df['age] = age_column
```

Order of samples in a data set can play a big role in the performance of the model built on this data set. We can shuffle the rows in a data set to solve this problem:

```
df = df.sample(frac=1)
```

Here, frac=1 means we are asking for the whole data set back. If we set frac = 0.5, we would get only half of the data set.

Sometimes, we may want to apply a function on a column that will change the column or will create a new one. For example, the normal cholesterol range is 125 - 200. We can create a new binary column that will determine whether the cholesterol is in the normal range. For this, we will use the **apply** function that takes as an argument a function that takes only on value. In our case will use the python lambda function:

```
In:
df['normal_cholesterol'] = df['chol'].apply(lambda x : x >
   125 and x < 200)
df['normal_cholesterol']
Out:
0      False
1      False
2      False
3      False
4      False
       ...
298    False
299    False
300     True
301     True
302    False
Name: normal_cholesterol, Length: 303, dtype: bool
```

3.3 Conclusion.

In this topic, we explored the most used pandas functionalities. Pandas is one of the most used tools during the data analysis and data preparation step of the Machine Learning Model development steps. We highly encourage you to explore this library further. 'Python Data Science Handbook' by **Jake VanderPlas** is a great book to continue the exploration of pandas.

4 Introduction into Linear Algebra. Numpy library.

Linear Algebra is a branch of mathematics that concerns itself with linear equations, vectors, matrices, etc. Since Machine Learning relies heavily on data and most data can be cleverly converted into numerical form, one way or another, one can immediately see how Linear Algebra is an essential tool in this field. Matrices and vectors are nothing but containers for our data! We can further apply operations on vectors and matrices to manipulate our data.

Python has an excellent library for manipulating matrices - NumPy. Moreover, NumPy is also necessary because it optimizes how lists work in python by using arrays. Python is generally considered to be a slow programming language. One cause of this is that one does not have to specify the data type of a variable or list, i.e., a list can contain variables of multiple data types. Numpy allows us to create arrays and matrices with specific data types like integers, floats, booleans, etc., and provides vectorized operations on these matrices that take less time to execute. Most of the code behind these operations is written in C to optimize these operations while still using familiar python syntaxis.

A matrix is just a two-dimensional array of numbers - containing rows and columns. An example of a matrix is this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is particular because it is square (the number of rows and columns are equal) and has 1s on the main diagonal and zeros everywhere else. This is called an **Identity Matrix**. We denote Identity Matrices by **In**, where n is the number of rows (or columns).

A vector is a matrix that has either only one column or only one row. A 1n matrix is a row vector, and a 1m matrix is a column vector. Most of the vectors we're going to encounter are going to be column vectors. Below you can see a column and row matrix, respectively.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$
$$\begin{bmatrix} d & e & f \end{bmatrix}$$

4.1 Operations on vectors and matrices.

We can perform the following operations with matrices and vectors:

- Addition of a scalar value.
- Multiplication by a scalar value.
- Transposition.
- Addition of a vector (or matrix).
- Multiplication by a vector (or matrix).

By adding a scalar value to a matrix or vector, we are adding this scalar to every value in that matrix (vector), as shown below:

$$a + 2 = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} + 2 = \begin{bmatrix} a_1 + 2 & a_2 + 2 & a_3 + 2 \end{bmatrix}$$
$$A + 2 = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{13} \end{bmatrix} + 2 = \begin{bmatrix} 2a_{11} + 2 & 2a_{12} + 2 & a_{13} + 2 \\ 2a_{21} + 2 & 2a_{22} + 2 & 2a_{13} + 2 \end{bmatrix}$$

We can perform multiplication by a constant on a matrix/vector by just multiplying each entry by the constant:

$$2a = 2 \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} = \begin{bmatrix} 2a_1 & 2a_2 & 2a_3 \end{bmatrix}$$
$$2A = 2 \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{13} \end{bmatrix} = \begin{bmatrix} 2a_{11} & 2a_{12} & a_{13} \\ 2a_{21} & 2a_{22} & 2a_{13} \end{bmatrix}$$

We can also transpose a matrix, which will swap rows and columns such that for any matrix A, each entry $a_{ij} \rightarrow a_{ji}$. We denote a transposed matrix byA^T. Below is an example of a transposition. Note that if we transpose a column vector, we get a row vector and vice versa.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}^T = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}^T = \begin{bmatrix}1\\2\\3\\4\end{bmatrix}$$

Next operation that we can do on matrices and vectors is the addition of matrices to matrices or vectors to vectors. However, there is a requirement that both operands should have the same dimensions. Below are shown the addition of the vectors and matrices:

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} + \begin{bmatrix}5 & 6 & 7 & 8\end{bmatrix} = \begin{bmatrix}6 & 8 & 10 & 12\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} + \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = \begin{bmatrix}6 & 8\\10 & 12\end{bmatrix}$$

You can also apply an operation between two vectors called the dot product. The dot product between two vectors a and b is denoted by ab or $a^T b$. So, the dot product is the transpose of the first vector multiplied by the second vector:

$$a \bullet b = a^T b = a_1 b1 + a_2 b_2 + ... + a_m b_m$$

But we haven't explained what it means to multiply two vectors/matrices!

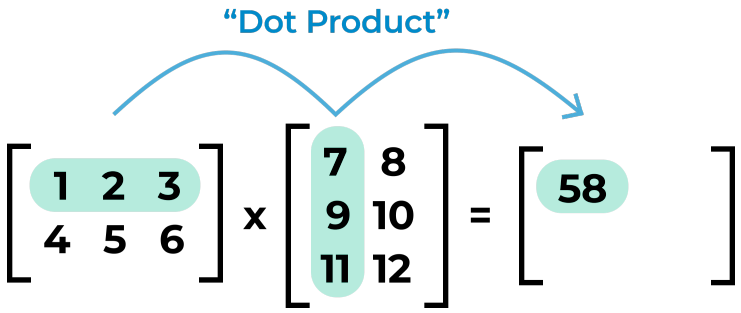


Figure 6: Dot product illustration.

So, if we multiply two matrices A and B, we obtain a third matrix C such that any entry $c_{ij} = a_i^T b_j$. Notice that for multiplication to be possible between two matrices, the first one has to have as many columns as the second one has rows. This means that if we multiply A and B, we obtain a third matrix C such that any entry $m \times n$ matrix with an $j \times k$ matrix, this implies $n = j$.

The inverse of a matrix A is a matrix B such that AB = I and BA

= I, respectively. This naturally implies that A and B must indeed be square matrices. However, not all square matrices are invertible. There are several ways to find an inverse of a square matrix if it has one. The most well-known is Gaussian Eliminations, which we encourage you to research online if you are not already familiar with it.

A way to test if a matrix is invertible is to compute its determinant, which is a scalar value and is denoted by det(A) or $|A|$. The following formulas show you how to compute the determinant for a 2x2 matrix:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = (ad - bc)$$

In the following listing is the formula for finding the determinant for a 3x3 matrix:

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} = a(ei - fh) - b(di - fg) + c(dh - eg) = aei + bfg + cdh - ceg + bdi - afh$$

A very important fact is that if $|A| = 0$ then the matrix is not invertible and it is called a **singular matrix**. It is also true that if a matrix is singular, its determinant will be equal to 0.

4.2 Numpy

We have mentioned previously that NumPy can do all the heavy lifting for us and automatically do all of these matrix operations for us and much more. Let's take a look now at what NumPy can do for us. Numpy is a python library written in C language for working and processing arrays. Usually, it calls C or CPython code for faster processing of arrays.

First, let's install the library, type in the cmd (on windows) or terminal (on Linux or MacOS) the following command:

```
pip install numpy
```

To start working with numpy, you must import it:

```
import numpy as np
```

Let's first generate three random arrays: a vector, a matrix, and a tensor.

```
# Setting up the seed
```

```
np.random.seed(42)

vector = np.random.randint(10, size=6)
matrix = np.random.randint(10, size = (3, 4))
tensor = np.random.randint(10, size = (3, 4 , 5))
```

randint function generates an array of integers. The first argument represents the maximal value of the generated number. If two values are passed, then the first is taken as the minimum value and the second one as the highest value. The size argument takes an integer or tuple representing the dimensionality of the generated array.

Now, we can print these newly generated tensors (tensors are a generalized term for vectors and matrices).

```
In:
matrix
Out:
array([[2, 6, 7, 4],
       [3, 7, 7, 2],
       [5, 4, 1, 7]])
```

Numpy arrays have some special functions that allow us to learn more about their dimensionality. The first one is ndim - which returns the number of dimensions of the array:

```
In:
print(f'vector - {vector.ndim}')
print(f'matrix - {matrix.ndim}')
print(f'tensor - {tensor.ndim}')
Out:
vector - 1
matrix - 2
tensor - 3
```

shape property returns the shape of the array.

```
In:
# shape - returns the shape of the array
print(f'vector - {vector.shape}')
print(f'matrix - {matrix.shape}')
print(f'tensor - {tensor.shape}')
Out:
vector - (6,)
matrix - (3, 4)
```

```
tensor - (3, 4, 5)
```

size property returns the number of elements in the array.

```
In:
# size = returns the number of elements in the array
print(f'vector - {vector.size}')
print(f'matrix - {matrix.size}')
print(f'tensor - {tensor.size}')
Out:
vector - 6
matrix - 12
tensor - 60
```

Another useful property of the numpy arrays is dtype, which returns the data type of the array:

```
In:
vector.dtype
Out:
dtype('int32')
```

To get an element from an array, we must use indexes. Indexes start at 0, like in the python lists.

```
In:
vector[0]
Out:
6
In:
matrix[0]
Out:
array([2, 6, 7, 4])
In:
tensor[0]
Out:
array([[5, 1, 4, 0, 9],
       [5, 8, 0, 9, 2],
       [6, 3, 8, 2, 4],
       [2, 6, 4, 8, 6]])
```

To get an element from an array with more dimensions than one, we must use more indexes, one for each dimension.

```
In:
```

```
matrix[0][1]
Out:
6
In:
tensor[0][1]
Out:
array([5, 8, 0, 9, 2])
In:
tensor[0][1][4]
Out:
2
```

We can use slices to get more than one element from a numpy array:

```
In:
# Get the first three elements of the array
vector[:3]
Out:
array([6, 3, 7])
In:
# Get the elements starting from element with index 3
vector[3:]
Out:
array([4, 6, 9])
In:
# Get every second element
vector[::2]
Out:
array([6, 7, 6])
In:
# In matrices and tensors, we can use slices to get the
   value of only some columns of rows
In:
# For example, let's get only the even column
matrix[:, ::2]
Out:
array([[2, 7],
       [3, 7],
       [5, 1]])
In:
# Or only the first half of columns
```

```
matrix[:, :3]
Out:
array([[2, 6, 7],
       [3, 7, 7],
       [5, 4, 1]])
```

To change a value in an array, you must do the following operation:

```
In:
vector[1] = 99
vector
Out:
array([ 6, 99,  7,  4,  6,  9])
```

Also, we can change more values using slices:

```
In:
# also, we can change more values using slices
matrix[:, 0] = np.zeros(3)
matrix
Out:
array([[0, 6, 7, 4],
       [0, 7, 7, 2],
       [0, 4, 1, 7]])
```

Now, every element of the first column is represented by 0.

np.zeros and np.ones allow to create arrays with zeros or ones, respectively:

```
In:
# Let's create a zero-filled vector.
np.zeros(3)
Out:
array([0., 0., 0.])
In:
# If we pass a tuple of values, the function will create
   an array with more dimensions
np.zeros((2, 4))
Out:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
In:
# By the same logic works and the np.ones functions.
np.ones(4)
```

```
Out:
array([1., 1., 1., 1.])
In:
# The same will take place if we will pass a tuple.
np.ones((2, 4))
Out:
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Creation of copies in python can be tricky, especially when we are talking data structures. Look at the example below to understand how to make copies to arrays:

```
In:
fake_copy = vector
# Let's verify their ids or memory addresses
print(id(fake_copy))
print(id(vector))
Out:
2040425261536
2040425261536
```

They are pointing to the same memory cell. So this is a fake copy. To create an actual copy, we should use the copy function.

```
In:
real_copy = vector.copy()
print(id(real_copy))
print(id(vector))
Out:
2040439290816
2040425261536
```

Now let's see the difference.

```
In:
print(vector)
print(fake_copy)
print(real_copy)
Out:
[0 8 6 8 7 0]
[0 8 6 8 7 0]
[0 8 6 8 7 0]
```

Let's change the vector and see what happens.

```
In:
vector[0] = 99
print(vector)
print(fake_copy)
print(real_copy)
Out:
[99  8  6  8  7  0]
[99  8  6  8  7  0]
[0 8 6 8 7 0]
```

As you can see the *fake_copy* array also changed itself. This happens because, in python, variables aren't boxes but stickers to boxes. You are just sticking another sticker to a box by creating a fake copy.

Concatenation is the process of sticking two or more arrays in one. Numpy has a special function for this - np.concatenate:

```
In:
x1 = np.array([1, 2, 3])
x2 = np.array([4, 5, 6])
np.concatenate([x1, x2])
Out:
array([1, 2, 3, 4, 5, 6])
```

As you can see in the listing above, the concatenate function just sticked the two vectors together. When we are working with matrices or multi-dimensional arrays, we can concatenate on different axes:

```
In:
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
np.concatenate([matrix, matrix])
Out:
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In the above example, we concatenate two matrices on rows, meaning that we are adding the second matrix row by row to the first matrix. To concatenate two matrices on columns, we should set the parameter axis = 1.

```
In:
np.concatenate([matrix, matrix], axis=1)
```

```
Out:
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

Axis parameter is saying on which axis to concatenate, and usually on which dimension on the array to apply the function:

- **axis = 1**: means the columns.
- **axis = 0**: means on the rows.

Sometimes, you can get stuck with using the axis parameters. If you want to get rid of using it, you can use the vstack and hstack functions, which are doing the concatenation, but they can be used to concatenate a matrix with a vector. vstack is used to stack arrays vertically:

```
In:
x = np.array([1, 2, 3])
grid = np.array([[4, 5, 6],
                 [7, 8, 9]])
np.vstack([x, grid])
Out:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

While hstack allows you to stack arrays horizontally:

```
In:
x = np.array([[99],
              [99]])
np.hstack([x, grid])
Out:
array([[99,  4,  5,  6],
       [99,  7,  8,  9]])
```

Sometimes, we may need to split an array into some parts. We may use the split, vsplit, and hsplit functions. np.split takes as arguments the array that should be split and a list with indexes on which to split:

```
In:
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
first, second = np.split(x, [5])
first
Out:
```

```
array([1, 2, 3, 4, 5])
In:
second
Out:
array([ 6,  7,  8,  9, 10])
```

vsplit allows us to split a multi-dimensional array on the vertical axis.

```
In:
matrix = np.arange(16).reshape((4, 4))
matrix
Out:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
In:
upper, lower = np.vsplit(matrix, [2])
upper
Out:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
In:
lower
Out:
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

hsplit allows us to split a multi-dimensional array on horizontal axis.

```
In:
matrix = np.arange(16).reshape((4, 4))
matrix
Out:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
In:
left, right = np.hsplit(matrix, [2])
left
Out:
array([[ 0,  1],
```

```
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
In:
right
Out:
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

Usually, python isn't the fastest programming language. Most of the time, this is because of the dynamic typization and loops because they take more time to be interpreted.

Let's see an example and compare pure python and the numpy one. Suppose we have an array of values and want to compute every element's inverse.

```
import numpy as np
np.random.seed(42)

def compute_inverse(values):
    # np.empty generates an empty array that can be filled
    after.
    out = np.empty(len(values))
    for i in range(len(values)):
        out[i] = 1.0 / values[i]
    return out

values = np.random.randint(1, 10, size=5)
compute_inverse(values)
```

Now let's create a big array, suppose with 1000000 values, and compute its inverse using the above function. For this, we will use a magic function from ipython - %%timeit. This function will allow us to measure how long it takes for the cell to run in a Jupiter notebook.

The creation of the array:

```
big_array = np.random.randint(1, 100, size=1_000_000)
```

Computing of the inverse:

The creation of the array:

```
In:
```

```
%%timeit -n 1 -r 1
compute_inverse(big_array)
Out:
2.36 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop
   each)
```

-n 1 -r 1 means that we will repeat this operation once and retry to run it only once. As we can see, it takes 2.3 seconds. Let's see how long it will take to numpy.

```
In:
%%timeit -n 1 -r 1
print(1.0 / big_array)
Out:
6.66 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop
    each)
```

As we can see, it took us only 6.66 ms to run this operation in numpy. Also, we can see that we divided 1 by an array. We can make the operations on vectors and matrices in numpy, which we talked about at the beginning of this chapter, as the following code snippet shows:

```
In:
# Creation on an ordered array.
x = np.arange(10)
In:
# Addition.
x + 2
Out:
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
In:
# Subtraction.
x - 2
Out:
array([-2, -1,  0,  1,  2,  3,  4,  5,  6,  7])
In:
# Multiplication.
x * 3
Out:
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
In:
# Division,
x / 4
```

```
Out:
# True Division.
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.
   , 2.25])
In:
# Floor Division.
x // 4
Out:
array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2], dtype=int32)
In:
x ** 2
Out:
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81], dtype=int32
   )
In:
x % 3
Out:
array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype=int32)
```

All these operations have functions implementer in numpy to:

```
In:
# Creation on an ordered array.
x = np.arange(10)
In:
# Addition.
np.add(x, 2)
Out:
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
In:
# Subtraction.
np.subtract(x, 2)
Out:
array([-2, -1,  0,  1,  2,  3,  4,  5,  6,  7])
In:
# Multiplication.
np.multiply(x, 2)
Out:
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
In:
# True Division,
```

```
np.divide(x, 2)
Out:
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.
   , 2.25])
In:
# Floor Division.
np.floor_divide(x, 2)
Out:
array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2], dtype=int32)
In:
p.power(x, 2)
Out:
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81], dtype=int32
   )
In:
np.mod(x, 2)
Out:
array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype=int32)
```

If you want to find the absolute values of a vector, there are three ways:

```
In:
x = np.array([-2, -1, 0, 1, 2])
abs(x)
Out:
array([2, 1, 0, 1, 2])
In:
np.absolute(x)
Out:
array([2, 1, 0, 1, 2])
In:
np.abs(x)
Out:
array([2, 1, 0, 1, 2])
```

In the previous topic about pandas, you learned about data aggregation functions. They are accessible in numpy too. Let's first generate a random vector of 100 values.

```
x = np.random.random(100)
```

To find the sum of a vector, we can use np.sum function:

```
In:
np.sum(x)
Out:
49.57394099300927
```

Also, we can find the minimal and the maximal value in an array:

```
In:
# The minimal value in the array.
np.min(x)
Out:
0.010434846331033976
In:
# The maximal value in the array.
np.max(x)
Out:
0.9899762168912337
```

There are also functions for finding the mean and the median for an array:

```
In:
# The mean value in the array.
np.mean(x)
Out:
0.49573940993009274
In:
# The median of an array.
np.median(x)
Out:
0.5118104887502954
```

Also, we can use aggregate functions on axes:

```
In:
x2 = np.random.random((5, 2))
x2
Out:
array([[0.410685  , 0.6906934 ],
       [0.11353877, 0.51628763],
       [0.58091506, 0.89581771],
       [0.87993057, 0.24566322],
       [0.74764971, 0.14486956]])
In:
```

```
# Finding the minimal value for every row
x2.min(axis=1)
Out:
array([0.410685  , 0.11353877, 0.58091506, 0.24566322,
   0.14486956])
In:
# Finding the minimal value for every column
x2.min(axis=0)
Out:
array([0.11353877, 0.14486956])
```

The same logic can be used for every aggregation function.

The last (but not the least) operation that we can apply in numpy on arrays is the dot product. To calculate the dot product between two vectors or matrices, you must use the np.dot function as follows:

```
In:
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
np.dot(a, b)
Out:
32
In:
a = np.array([1, 2, 3])
matrix = np.array([[1, 3, 4],
                   [4, 5, 6]])
np.dot(matrix, a)
Out:
array([19, 32])
```

4.3 Conclusion

In this topic, we learned the basics of Linear Algebra and how it can be used with the help of the NumPy library. Linear Algebra is essential in Machine Learning because we work with multi-dimensional data, and linear algebra has special data structures and operations. These can be applied to create powerful algorithms like Linear and Logistic Regression which you will find in the next chapters. We strongly recommend you look at other books to expand your knowledge about linear algebra.

5 Linear Regression

Now that you've acquired some experience with operations on matrices and got acquainted with the NumPy and Pandas libraries, it is time to dive head first into one of the most basic (but also one of the most utilized) machine learning models - **Linear Regression**.

Before explaining what the model does, we must first define a fundamental concept. **Target variables** or **dependent variables** are the values we try to predict after training a machine learning model on the whole data set. If we receive the data in tabular form, we expect that we can somehow predict the column of interest based on the other remaining columns. This column we try to predict is what we call the **target column**. For example, suppose you examine the **Heart Disease UCI** data set. In that case, you will notice that each row corresponds to an individual patient, and each column describes the various measurements that the doctors made on each patient. The last column (which, in our case, happens to be the target column) tells whether the patient does or doesn't suffer from heart disease. It is reasonable to think that we can predict whether a person is sick based on parameters such as **age**, **cholesterol levels**, **type of chest pain** etc.

It often happens that we can observe a linear relationship between the target values and the other columns in our data.

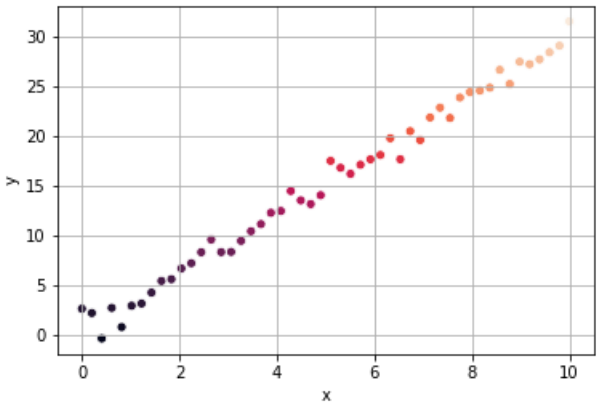


Figure 7: Relation between two features.

Above, you can see a plot of some data with a linear relationship. This could illustrate the relationship between time spent studying and the score obtained on the exam or the dependency between the area of a house and its market price. However, just like in real-life situations, the data is not perfect. The points do not lie perfectly on a line - some noise

is present. This noise prevents us from defining a line that simultaneously goes through all these points. However, we could still find a line that approximates the data well. If we add one more column to the data set, we can see that the data transforms into a 3D plane. Unfortunately, we cannot visualize linear dependency in datasets with dimensionality higher than 3; however, **Linear Regression** can still be applied to them.

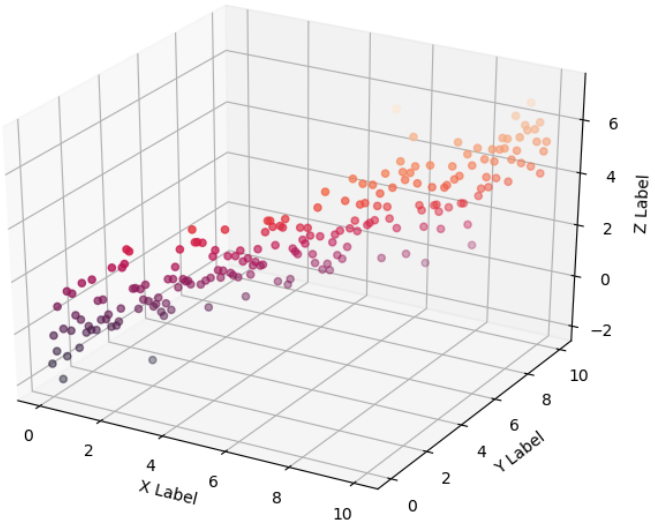


Figure 8: Relation between three features.

So we've established that even though we cannot fit a straight line (or n-dimensional plane) through all of our data points, we can still find a **best-fitting line** such that it approximates our data well. This might look like in the following figure on our previous examples:

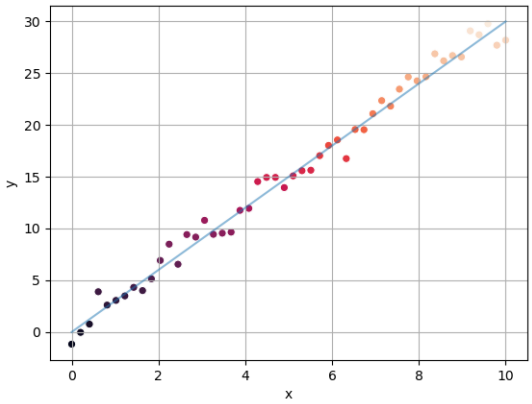


Figure 9: A line passing through the samples.

The "best-fitting line" or "plane" can be described by a vector of

parameters, also called weights. For instance, a straight line has a weight vector composed of its y-intercept and slope. This is all the information we need to construct the line. We then use the line to make the following predictions based on new data that the model has never seen before.

5.1 The setup:

Before actually implementing the Linear Regression model, we must first build a foundation so that we can turn our problem from words into a **mathematical model**. To understand the foundation, it is best to start with a concrete example on a small scale, which we can later generalize.

Let's say one would want to fit a "best-fitting" line through three non-colinear points: **(1, 1), (2, 2), (3, 2)**. You can see a picture of these points below:

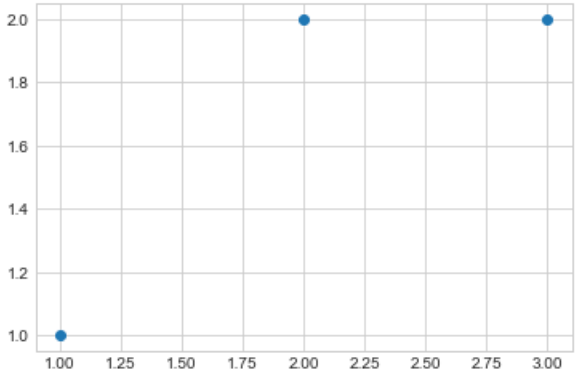


Figure 10: Three non-colinear points.

Let's imagine for a moment that we are trying to define a line y = ax + b that magically goes through all three points. We would define the following system, where we plug in the x and y coordinates from the points into the equation above:

$$\begin{cases} 1a + b = 1 \\ 2a + b = 2 \\ 3a + b = 2 \end{cases} \qquad (1)$$

Here, we are trying to solve for variables a and b. This system can be rewritten like this by the use of Linear Algebra:

$$a \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + b \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \qquad (2)$$

or

$$Ax = y \Leftrightarrow \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \quad (3)$$

Remember the rules by which we multiply two matrices to check that we can rewrite the system like this.

We can see from (1) and (2) that we are looking for a linear combination of columns of our matrix A that will output our target vector y. Also, notice how the second column is all made up of ones. This is because the weight corresponding to the column is, in fact, a constant - the b intercept of our line.

But remember! Our system is unsolvable, so instead of solving (3), we need to solve

$$A\hat{x} = p \quad (4)$$

where $\hat{x}$ is an "approximation" of x, and p is a vector that is as close as possible to y. x contains all the weights necessary to construct our intended line.

5.1.1 The Generalization of the problem:

Now, generally, we will always try to solve a system of the type

$A\hat{x} = p$

So, if our matrix A has n columns, then our vector $\hat{x}$, which we will now on notate by W will be of the form $W = [w_1, w_2..w_n]^T$, so that we define a line (or an n-dimensional plate rather) that is defined by $y = w_1x_1 + w_2x_2 + ... + w_nx_n$, where $x_1...x_n$ are variables corresponding to the different labels in our dataset. As mentioned above, our further predictions on new data are made by replacing $x_1...x_n$ with column vectors that represent our observations of our labels and y as the vector of predictions a.k.a. the target column.

5.1.2 The Cost Function:

We have mentioned above that we need our line to fit our dataset as well as possible. Now we must ask ourselves: What does "well" mean, and how do we measure how "well" the model fits our model?

To measure how well a model does on given data, we define a function that measures how far away our predictions are from the actual observations from the target column. For simplicity, from now on, we will refer to our data matrix as X, the weight vector of our model as w, the target column as y, and the column of predictions as $\hat{y}$. (Notice that in fact $\hat{y} = Xw$).

So, we define the cost function as follows:

$$f(w) = (Xw - y)^T(Xw - y) \quad (5)$$

Notice that $f(w)$ takes as input a vector w and returns a scalar value. f takes the difference between our predictions y and the actual observed y. Then, it computes the cross-product of this difference with itself. The output scalar value tells us how badly the model has done. This becomes obvious if we rewrite the function like this:

$$f = \sum_{i=0}^{n}(\hat{y}_i - y_i)^2 \quad (6)$$

It is apparent that this is a sum of all the squares of the individual errors we have made in predicting each observation. This function has the interesting property that if the errors are generally big (say bigger than 1), the output will become dramatically larger because of the square. Consequently, the output will become significantly smaller if the errors are small.

5.1.3 Gradient Descent:

Since we want our model to do as well as possible, we want to minimize the output of the cost function. For this purpose, we use an operation called **the gradient**, which is a derivative applied to a multivariate function, such as the one above. Generally, it returns another vector or matrix that represents the direction of growth of the function.

On the next page, you can see the graph of a multivariate function and its gradient field. The arrows represent the magnitude and direction of the gradient at each input. Notice how all the arrows point in the direction where the function is increasing. Intuitively, the negative of the gradient would point towards the places where the function is decreasing. We will later see how we can use this fact to minimize our error through an iterative method.

All we need now is the cost function (5) gradient. We will not show

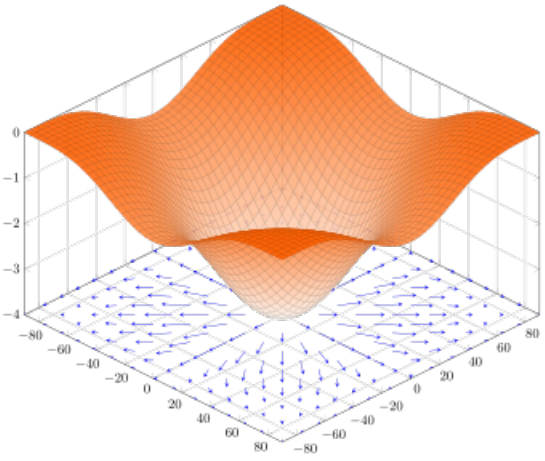


Figure 11: Gradient Descent.

how to calculate it ourselves because it involves some calculus outside this book's scope. So the gradient of (5) becomes:

$$\nabla f = 2X^T(Xw - y) \quad (6)$$

Now we can finally produce an iterative method, which we can later implement into code:

- Make an initial random guess w_0 for the weight vector W;
- Then, $w_i = w_{i-1} - \alpha X_T(Xw_{i-1} - y)$, where α is a scalar called the learning rate, which defines how far do we go in the direction of the gradient;
- Repeat step two until convergence;

This algorithm is intuitive because, at each step, we subtract the gradient from our previous estimate. Since we "add" the negative gradient to the estimate, we "push" it towards the value that minimizes the function. Thus, by applying this algorithm iteratively and going to the bottom of the graph, we obtain the vector w for which our cost function is as small as possible. You can imagine this process of subtracting gradient as rolling a ball on a 3D surface like the one above.

It is essential to examine the possibility that if we choose a learning rate α that is too big, we might "overshoot" and miss the minimum of our function. This can lead to infinite loops around the minimum or divergence. Moreover, if we choose a very small learning rate, the algorithm may become inefficient (i.e., the ball rolls very slowly down

the hill). Usually, we understand what learning rate to put when we try multiple variants. Sometimes, we see if it is learning too slowly or the learning rate is so big that it jumps over the minimum, which is also wrong. There are multiple ways to calculate and play with your learning rate to find the one that fits your data.

5.2 Implementation:

Now, the last thing we need to worry about is implementing the model itself. Below you can see an implementation of Linear Regression through Gradient Descent. This implementation divides the dataset into batches, although this is not entirely necessary.

```
import numpy as np

class LinearRegression():
    #Define a Linear Regression class to store our
  relevant functions in

    def __init__(self):
        '''
            Initializes the Linear Regression model.
            It also stores the self.params__ variable,
  which
        will be the weights that the model returns.
        '''
        self.params__ = None

    def gradientDescent(self, X, y, learning_rate=0.00001,
                        iterations=500, batch_size=16):
        '''
            This function applies the Gradient Descent
  model
            to the dataset
        :param X: numpy.ndarray
            The X matrix containing the independent
  variable columns.
        :param y: numpy.ndarray
            The target vector y.
        '''
```

```
        #Add a column of ones for the constant term
        X = np.concatenate([X, np.ones_like(y)], axis = 1)
        rows, cols = X.shape

        #Combine the X and y columns to more easily
shuffle it later
        X = np.append(X, y, axis=1)

        #Make the initial random guess for w
        w = np.random.random((cols, 1))

        #Go through all the iterations
        for i in range(iterations):
            #Shuffle the rows of the data
            np.random.shuffle(X)
            #Define X and y again
            y_it = X[:, -1].reshape((rows, 1))
            X_it = X[:, :-1]

            #Iterate through the batches
            for batch in range(math.ceil(rows / batch_size)
):
                batch_start = batch * batch_size

                #Cut a batch from the dataset
                x_batch = X_it[batch_start : min(
batch_start + batch_size, X.shape[0])]
                y_batch = y_it[batch_start : min(
batch_start + batch_size, X.shape[0])]

                #Subtract the gradient from our previous
estimation
                w -= learning_rate * np.matmul(x_batch.
transpose(), (np.matmul(x_batch, w) - y_batch))

        self.params__= w
        return self

    def predict(self, X):
```

```
        X = np.concatenate([X, np.ones(X.shape[0]).reshape
   ((X.shape[0], 1))], axis=1)
        return np.matmul(X, self.params__)
```

However, we usually want to use the Sklearn library that has a more advanced implementation of this model:

```
# Importing all needed libraries.
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,
   mean_absolute_error
from sklearn.linear_model import LinearRegression

# Define some mock-up data.
# Create a 10x2 matrix with random integers in the range
   0-100.
X = np.linspace(0, 100, 100).reshape(100, 1)
# Define a random noise vector to add onto the vector y.
e = np.random.uniform(-5, 5, (100, 1))
#Define the vector y = 3*x + e.
y = 3*X + e

# Split the data into train and test.
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.3, random_state=42)

# Create an instance of the LinearRegression class from
   sklearn.
lr = LinearRegression()

# Call the fit function, which applies the Linear
   Regression model
# onto the data.
lr.fit(X_train, y_train)

# Define a variable that stores the predictions of our
   model.
y_pred = lr.predict(X_test)

# Print the mean squared error and mean absolute error to
```

```
   get an idea of
# how well the model has done.
mean_squared_error(y_pred, y_test), mean_absolute_error(
   y_pred, y_test)
```

5.3 Conclusions:

In this topic, we have learned a couple of useful and omnipresent concepts like:

- Target and Input labels;
- The Idea of a Linear Model;
- What is a cost function;
- Cost function optimization through an optimization method such as gradient descent;

6 Logistic Regression

Notice that until now, we have talked about a model that outputs predictions of continuous values. What happens if we want to predict a set of discrete values? For example, it would be quite reasonable to think that we could predict whether a person is male or female (without physically looking at them) based on certain criteria like height, weight, amount of testosterone in their body, etc. Trying to achieve such a fit is called a classification problem. More specifically, since we are talking about two classes - male and female, it is called a binary **classification problem**. It is a frequent occurrence that we want to classify data into two classes: whether a patient is sick or healthy, the presence or absence of cancer cells in one's body, whether an employee will leave their job, etc. The machine learning standard for binary classification is the **Logistic Regression model**.

To understand the mechanics of Logistic Regression, we must follow the same recipe as for Linear Regression:

- Understand the mathematical model behind the method;
- Find an appropriate cost function to optimize, so the errors are as small as possible;

6.1 The Logistic Function:

In fact, the Logistic Regression Model is not too far off from Linear Regression - it still assumes that there is some linear relationship between the features of our data set. The only difference is that it also finds a way to turn this linear dependency into something that can be interpreted as a probability. The "tool" that we use to translate the linear model to something probability-like is called the **Logistic Function**:

$$f(x) = \frac{L}{1+e^{-k(x-x_0)}}$$

Where L is the maximum value of the curve, k is the steepness, and x0 is where the function's graph intercepts the y-axis. Since we expect an output within the range [0, 1], we usually resort to the **Standard Sigmoid Function**, which has L=1, k=1, and x0=0, so we can present it like this:

$f(x) = \frac{1}{1+e^{-y}}$

And the function is represented graphically in the following manner:

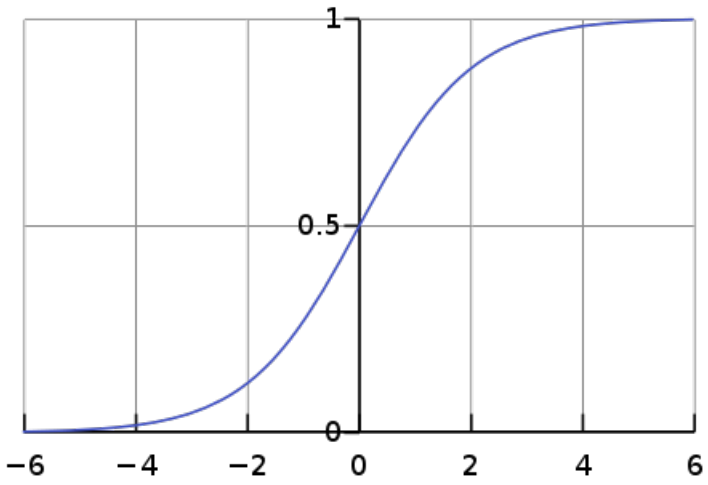


Figure 12: Sigmoid function.

We have mentioned above that Logistic Regression still expects some form of linearity between our features. This will be obvious once we state that y in the formula above is nothing but the output of the Linear Regression model, which we discussed above. So,

$$y = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n$$

where $w_0...w_n$ are the trained weights as a result of Linear Regression. So, we take the output from linear regression and then "squeeze" it into the range [0, 1] using the Sigmoid Function. Now we can interpret the output as probability and choose a threshold - for example, categorize any output $>= 0.5$ as 1 and any output < 0.5 as 0. We usually choose the threshold to be 0.5, but sometimes we might want to change that when we want to predict a class as well as possible. For example, when predicting whether a patient has or doesn't have cancer, we might want to be sure that we don't miss any ill patients. To do so, we might want to change the threshold in favor of the cancer class, even if that means that we might falsely categorize some patients as ill, even though they are not. A false positive is usually better than a false negative when it comes to medicine.

6.2 The Cost Function:

Since our final output is different, we must choose to optimize an appropriate cost function that reflects our output. The reason we can't use the Least Squared Sum (like we did previously on Linear Regression) is that, in combination with Logistic Regression, it returns a function that has more than one minimum, and we try to optimize it, the optimization method might get stuck in a local minimum that is not the global minimum (i.e., we do not find the optimal weights).

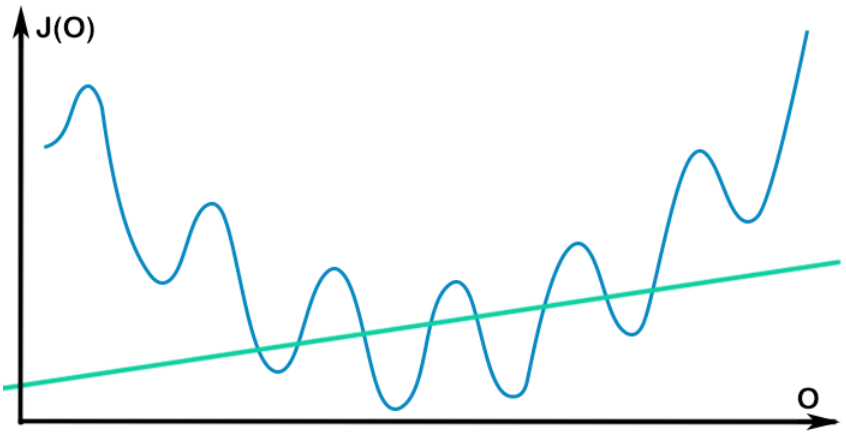


Figure 13: Local minimums.

For Logistic Regression, we usually use the appropriately-named Log Loss Function. The cost of an individual prediction on a data point is the following:

$$Cost(h_w(x), y) = \begin{cases} -log(h_w(x)) & y = 1 \\ -log(1 - h_w(x)) & y = 0 \end{cases}$$

where $h_w = \frac{1}{1+e^{-y}}$

of the output of the Sigmoid Function and θ is the vector of trained weights, and x is the vector of observations for that data point. This makes $w^T x$ the dot product between two vectors, which is a scalar and the output of a Linear Regression model. The cost function can be rewritten in a single line like this:

$$Cost(h_w(x), y) = -ylog(h_w(x)) - (1 - y)log(1 - h_w(x))$$

Notice how we cleverly use the fact that y can only possess 0 or 1 as a value to cancel the first or second term in this function completely. So, we take the mean of the cost function applied to each data point for the

whole data set.

$$J(w) = -\frac{1}{m}[y^{(i)}log(h_w(x^{(i)})) + (1 - y^{(i)})log(1 - h_w(x^{(i)}))]$$

We are trying to find the set of parameters θ that will minimize function J. For this, we can once again use the method of Gradient Descent, i.e., use the following recursive relation:

$$w_j := w_j - \alpha\frac{\delta}{\delta w_i}J(w)$$

where α is the learning rate we chose for our model. For the sake of simplicity, we will use a modified version of the gradient for our gradient descent:

$$w_j^{new} := w_{ij}^{old} - \alpha\frac{1}{N}x_j^T(\hat{y}_j - y_j), j\epsilon 1...N$$

where N is the number of features (number of columns of the matrix X) and w_j^{new} is a single entry in the vector of parameters which each iteration of the training process produces.

6.3 Implementation:

You can see a by-hand implementation of Logistic Regression below, although we recommend using the sklearn library instead. Coding your models by hand is a great way to test your knowledge, though.

```
import numpy as np
class LogisticRegression:
    def __init__(self, learning_rate : float = 0.05,
                 max_iter : int = 100000) -> None:
        '''
            The constructor of the Logistic Regression
  model.
                :param learning_rate: float, default=0.05
                    The learning rate of the model.
                :param max_iter: int, default = 100000
                    The number of iterations to go through
  .
        '''
        # Setting up the hyperparameters.
        self.__learning_rate = learning_rate
```

```
        self.__max_iter = max_iter

    def sigmoid(self, y : 'np.array') -> 'np.array':
        '''
            The sigmoid function.
                :param y: np.array
                    The predictions of the linear function
.
        '''
        return 1 / (1 + np.exp(-y))

    def fit(self, X : 'np.array', y : 'np.array') ->
LogisticRegression:
        '''
            The fit function of the model.
                :param X: 2-d np.array
                    The matrix with the features.
                :param y: 1-d np.array
                    The target vector.
        '''
        # Creatting the weights vector,
        self.coef_ = np.zeros(len(X[0])+1)

        # Adding the intercept column.
        X = np.hstack((X, np.ones((len(X), 1))))

        # The weights updating process.
        for i in range(self.__max_iter):
            # Prediction.
            pred = self.sigmoid(np.dot(X, self.coef_))

            # Computing the gradient.
            gradient = np.dot(X.T, (pred - y)) / y.size

            # Updating the weights.
            self.coef_ -= gradient * self.__learning_rate
        return self

    def predict_proba(self, X : 'np.array') -> 'np.array':
        '''
```

```
            This function returns the class probabilities.
                :param X: 2-d np.array
                    The features matrix.
                :return: 2-d, np.array
                    The array with the probabilities for
    every class
                    for every sample.
            '''
            # Adding the intercept column.
            X = np.hstack((X, np.ones((len(X), 1))))

            # Computing the probabilities.
            prob = self.sigmoid(np.dot(X, self.coef_))

            # Returning the probabilities.
            return np.hstack(((1 - prob).reshape(-1, 1),
                              prob.reshape(-1, 1)))

        def predict(self, X : 'np.array') -> 'np.array':
            '''
                This function returns the predictions of the
    model.
                :param X: 2-d np.array
                    The features matrix.
                :return: 2-d, np.array
                    The array with the probabilities for
    every class
                    for every sample.
            '''
            # Adding the intercept column.
            X = np.hstack((X, np.ones((len(X), 1))))
            return (self.sigmoid(np.dot(X, self.coef_)) > 0.5)
    * 1
```

Now using sklearn:

```
# Importing the laod_iris function and LogisticRegression
    model from sklearn.
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
```

```
# Getting the feature matrix and target vector from the
   iris data set.
X, y = load_iris(return_X_y=True)

# Creatting the model.
clf = LogisticRegression(random_state=0)

# Fitting the model.
clf.fit(X, y)

# Making predictions with the Logistic Regression model.
y_pred = clf.predict(X)
```

6.4 Conclusions:

To understand Logistic Regression, we've used the same tools and concept as for Linear Regression: Model, Cost function, and Optimization Function (Gradient Descent in this particular case). These concepts apply to most, if not all, ML models. Moreover, we have seen how similar Linear and Logistic Regression are: The latter is just a linear model that goes through a Sigmoid Activation Function.

7 Principal Component Analysis

Principal Component Analysis, or **PCA**, is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets to reduce the amount of data that has to be processed. PCA transforms large data sets into smaller ones containing the most useful information.

PCA gives simplicity, it reduces the data set sizes, but nothing comes without a price. Data loses a small part of the information because of the reduced columns. While using PCA, with the simplicity of data, comes a smaller accuracy, and the trade-off begins.

Because we reduce the number of variables in the data set, it is easier to explore and visualize analyzing data. It makes it much easier and faster for machine learning algorithms to work.

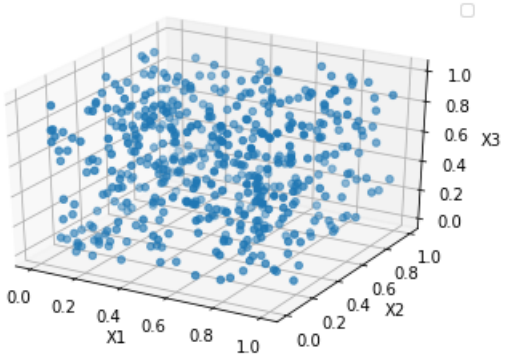


Figure 14: Before PCA.

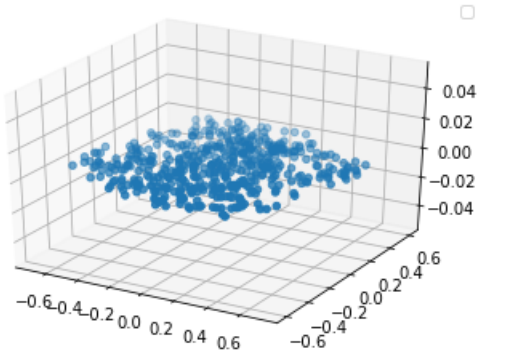


Figure 15: After PCA.

PCA can be implemented in two ways, using eigenvectors and their eigenvalues or using singular value decomposition. We will start with the first one.

7.1 PCA algorithm: eigenvectors and eigenvalues:

Step 1: Standardization.

The first step of this method is to standardize your data. Please consult the **Feature Engineering** topic to explore this process in more detail.

The main reason for this step is the standardization of the range of continuous values for features. Usually, different features exist in different ranges. For example, imagine having a data set about different hospital patients, and you have various features such as age or salary. In this example, age can take values from 0 to 150. Salary, for example, can take values from 0 to millions of dollars. In this case, we have to show that if the salary has bigger values, this doesn't mean that this feature is more important, and then, using standardization, we make their ranges equal. After this step, all ranges for all values will be between -3 and 3. The formula for Standard Deviation is shown below:

$$Z = \frac{x-\bar{x}}{\sigma}$$

In the formula above x is representing the values in the series, $\bar{x}$ - the mean value of the series and σ is the standard deviation of the series (it's formula is listed below).

Using StandardScaler, it's done by the following lines of code:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Step 2: Computation of Covariance matrix.

The covariance matrix is an f $\times$ f symmetric matrix (where f is the number of features) that has as entries the covariances associated with all possible pairs of the initial variables. For example, for a data set with three variables x, y, and z, the covariance matrix is a 3$\times$3 matrix of this form:

$$\begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}$$

Convariance matrix from a numpy matrix ca be computed using *np.cov* function from numpy:

In a few words, the covariance matrix shows the relationship between every pair of features, how correlated they are to each other. Correlation values can be negative numbers which indicate a negative correlation, positive numbers, which suggest a positive correlation, and also 0, which means no correlation at all. For example, imagine having a data set where you have to predict a person's salary based on some features. In a perfect world, the higher the academic degree, the higher the pay. In this example, we have a positive correlation because a slight increase in one feature leads to a rise in the value of the other.

An example of a negative correlation would be the probability of getting a cold based on the temperature outside. The colder outside (lower temperature), the bigger likelihood for us to get a cold, and vice versa. The warmer outside (higher temperature), the smaller chances of getting a cold. A slight increase in the temperature leads to a decrease in the value of the other feature.

Lastly, there is the zero correlation which means that a change in one variable won't cause any changes in the other one, for instance, the probability of having heart disease based on the patient's name. There are no known cases where a person's name can cause a bigger probability of having heart disease ;).

$$\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

The formula above is the one that we use to calculate the covariance between two features. Because we standardized our data, now the mean of every variable is 0, and we can reduce the formula to:

$$\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n} x_i y_i$$

And because we are summing up the multiplications of values from the same row, we can use the dot product for the matrix. In that case, we will use all matrices at once, but we have to transpose the Y matrix to get the same operation we had.

$$\sigma(x, y) = \frac{XY^T}{n-1}$$

Using np.cov, we can get our covariance matrix where X is the standardized data matrix:

```
import numpy as np

cov_mat = np.cov(X)
```

Step 3: Compute eigenvectors and eigenvalues.

Eigenvectors and eigenvalues are essential concepts in linear algebra, and we need them to get the Principal components of the data.

Right now, you don't have to understand how eigenvectors work. What is important is that eigenvectors show the directions of the axes where there is the most variance (most information) and that eigenvalues are simply the coefficients attached to eigenvectors, which show the amount of variance carried in each Principal Component.

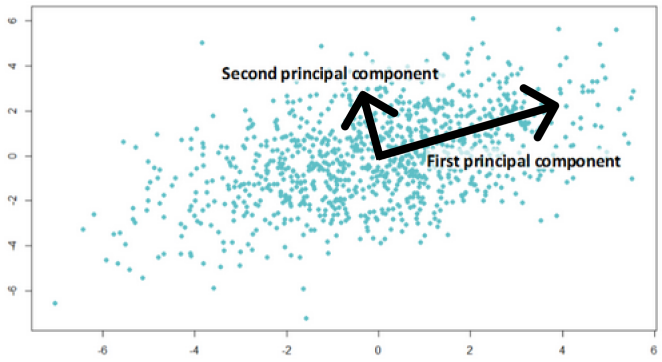


Figure 16: The principal components.

In our task, variance is a crucial thing. Imagine having a feature that is constant for every entry, like predicting heart diseases only on girls and having sex as a column. This column won't give us any information because this column will have only one unique value over all data set, which is 'female'. We just don't have any variance in that column, so we can confidently say that this column is useless. So, because eigenvalues attached to eigenvectors show the amount of variance, and variance means a lot of information, we can sort eigenvectors in descending order by their eigenvalues. We want to get eigenvectors with more information on the top, and eigenvectors with less or no information will be at the bottom of the list.

So this is how the PCA works: we get the eigenvectors, and by sorting all eigenvectors by their eigenvalues, we get our principal components.

Using NumPy and method eig, we can get eigenvectors for a given matrix, then we sort the eigenvectors we got by their eigenvalue:

```
# Sorting the list of tuples (eigenvalue, eigenvector).
eig_vals, eig_vecs = np.linalg.eig(corr_mat)
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:, i])
                for i in range(len(eig_vals))]
eig_pairs.sort(key=lambda x: x[0], reverse=True)
```

Step 4: Reduce dimensionality.

At the previous step, we got all principal components, so now it is time to select how much information we want to save. We choose how many principal components we want to keep. The other ones will vanish together with the information that they had. Because we sorted them, we can select the first N components (first n rows) from the matrix. That way, we can reduce the number of columns that our data set has, making data more compact and easier to visualize.

Using NumPy, we can select first n_components from our eigenvectors matrix:

```
# Creating the projection matrix
matrix_w = np.hstack((eig_pairs[i][1].reshape(np.size(X, 1)
   ,1) for i in range(n_components)))
```

Now, we can go to the second method of doing PCA, using singular value decomposition.

7.2 PCA algorithm: singular value decomposition:

The second way to reduce a matrix dimensionality, let's see how it works.

Step 1: Standardization.

Step 2: Compute SVD.

The singular value decomposition of a matrix A is the factorization of A into the product of three matrices $A = U \sum V$ where the columns of U and V are orthonormal. The matrix $\sum$ is diagonal with positive real entries.

U and V represent rotation matrices and the $\sum$ represents stretching matrix.

In that way, every matrix can be decomposed into three different matrices. In that way, we can get the eigenvectors of our covariance matrix, represented by the V matrix.

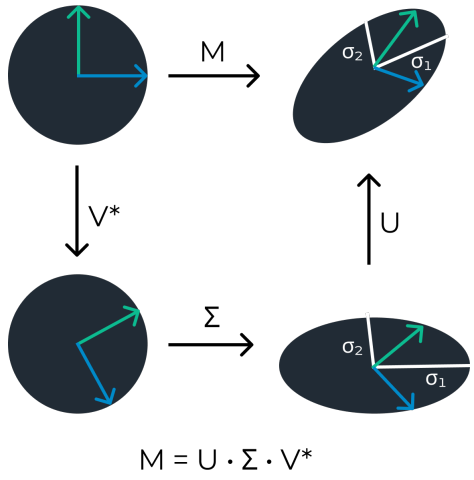


Figure 17: Singular Value Decomposition.

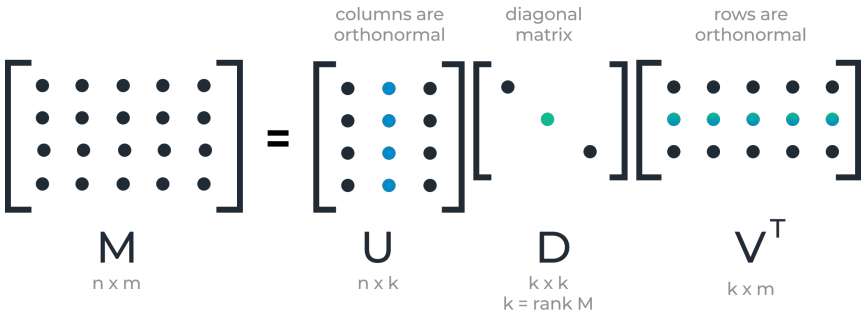


Figure 18: A matrix decomposed by SVD algorithm.

Using NumPy can do it with the help of svd method, here S is the $\sum$ matrix:

```
U, S, V = np.linalg.svd(X)
```

Step 3: Reduce dimensionality

Select the first n principal components from V that represent the first n eigenvectors of covariance matrix. Using NumPy just select first *n_components* rows and multiply it by the original matrix X.

```
V = V[:n_components, :]
new_X = np.dot(X, V.T)
```

7.3 The implementation of PCA from scratch:

```
class myPCA:
    def __init__(self, n_components : int = 2 , method :
   str = 'svd') -> None:
```

```
        '''
            The constructor of the PCA algorithm.
        :param n_compoents: int, default = 2
            The dimension to which the data will be
reduced.
        :param method: str, default = 'svd'
            The method used by PCA to reduce the
dimensionality of the data.
        '''
        self.__n_components = n_components
        if method in ['svd', 'eigen']:
            self.__method = method
        else:
            raise ValueError(f"'{method}' is not a method
implemented in this model")

    def fit(self, X : 'np.array'):
        '''
            The fitting method.
        :param X: np.array
            The data on which we want to fit the PCA
        '''
        if self.__method == 'svd':
            U, S, V = np.linalg.svd(X)
            self.__V = V[:self.__n_components, :]
        elif self.__method == 'eigen':
            corr_mat = np.corrcoef(X.T)

            # Getting the eigenvectors and eigenvalues
            self.eig_vals, self.eig_vecs = np.linalg.eig(
corr_mat)

            # Sorting the list of tuples (eigenvalue,
eigenvector)
            self.eig_pairs = [(np.abs(self.eig_vals[i]),
self.eig_vecs[:, i]) for i in range(len(self.eig_vals))]
            self.eig_pairs.sort(key=lambda x: x[0], reverse
=True)

            # Calculating the explained ration
```

```
            total = sum(self.eig_vals)
            self.explained_variance_ratio = [(i/total)* 100
    for i in sorted(self.eig_vals, reverse= True)]
            self.cumulative_variance_ratio = np.cumsum(
                self.explained_variance_ratio
            )

            # Creating the projection matrix
            self.matrix_w = np.hstack(
                (self.eig_pairs[i][1].reshape(np.size(X, 1)
   ,1) for i in range(self.__n_components))
            )
        return self

    def transform(self, X : 'np.array') -> 'np.array':
        '''
            The transform function.
        :param X: np.array
            The data that we must reduce.
        '''
        if self.__method == 'svd':
            return X.dot(self.__V.T)
        elif self.__method == 'eigen':
            return X.dot(self.matrix_w)
```

7.4 The implementation of PCA from sklearn:

```
import numpy as np

# Let's import the PCA algorithm from sklearn and reduce
   it to 2 dimensions
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)

# now, let's transform the data; pca_x represents the new
   reduced data
pca_X = pca.transform(X)
```

7.5 Conclusion:

In conclusion, the idea of PCA is simple — reduce the number of variables of a data set while keeping as much information as possible, helping the model learn faster.

8 Probability. Naive Bayes

Probability is a measure of uncertainty. Probability applies to machine learning because **in the real world, we need to make decisions with incomplete information**. Using probability, we can model elements of uncertainty such as risk in financial transactions and many other business processes. Probability provides us with a mechanism to quantify uncertainty. There are two ways of interpreting **probability**:

- **Frequentist probability** - considers the actual likelihood of an event;
- **Bayesian probability** - considers how strongly we believe that an event will occur.

8.1 A first definition of probability:

If the sample space S of an experiment consists of finitely many outcomes (points) that are equally likely, then the probability P(A) of an event A is:

$$P(A) = \frac{Number of points in A}{Number of points in S}$$

Although this fairly simple definition of probability is standard in many high school math books, it has a not so obvious limitation, the sample space S should consist of **finitely many** outcomes that are **equally likely**. This issue has been fixed in the following.

8.2 General Definition of probability:

Given a sample space S, with each event A of S (subset of S) there is associated a number P(A), called the **probability** of A, such that the following **axioms of probability** are satisfied.

1. For every A in S, $0 \leq P(A) \leq 1$
2. The entire sample space S has the probability: $P(S) = 1$
3. For **mutually exclusive** events A and B (if S is infinite, this axiom is extended for mutually exclusive events A1, A2, A3 . . .)

$P(A \cup B) = P(A) + P(B)$

These axioms of probability enable us to build up probability theory and its application to statistics

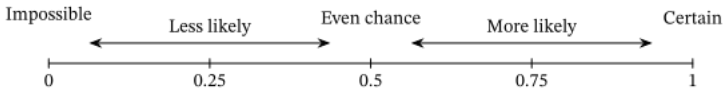


Figure 19: Probability margins.

8.3 Basic Theorems of Probability:

Theorem 1 - Complementation Rule

For an event A and its complement A^c in a sample space S,
$P(A^c) = 1 - P(A)$

Theorem 2 - Addition Rule fro Arbitrary Events

For events A and B is a sample space S,

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

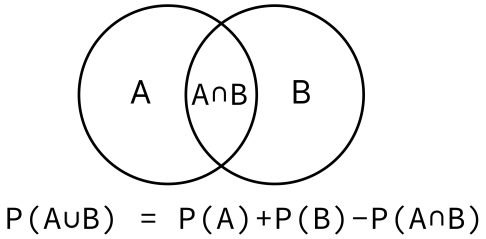


Figure 20: Addition Rule.

8.4 Conditional Probability:

Often it is required to find the probability of an event B under the condition that an event A occurs. This probability is called the **conditional probability** of B given A and is denoted by P(B—A).

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

Theorem 3 - Multiplication Rule

If A and B are events in a sample space S and $P(A) \neq 0, P(B) \neq 0$, then

$$P(A \cap B) = P(A)P(B|A) = P(B)P(A|B)$$

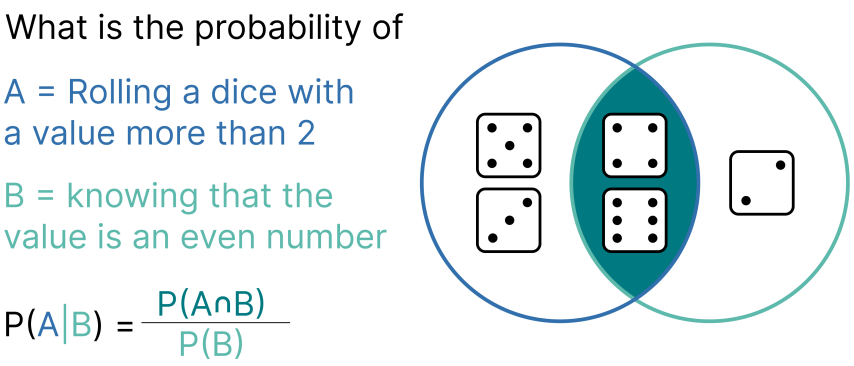


Figure 21: Conditional Probability.

8.5 Combinatorics:

Combinatorics is a key part for probability and statistics. **Combinatorics** is considered a branch of mathematics which is about counting.

A **permutation** of given things (elements or objects) is an arrangement of these things in a row in some order. For example, for three letters a, b, c there are 3! = 1 * 2 * 3 * 6 permutations: abc, acb, bac, bca, cab, cba.

1. **Different things**. The number of permutations of n different things taken all at a time is $n! = 1 \times 2 \times 3...n$

2. **Classes of equal things**. If n given things can be divided into c classes of alike things differing from class to class, then the number of permutations of these things taken all at a time is

$$\frac{n!}{n_1!n_2!...n_c!}$$

In a permutation, the order of the selected things is essential. In contrast, a combination of given things means any selection of one or more things without regard to order. There are two kinds of combinations, as follows.

The number of **combinations of n different things, taken k at a time, without repetitions** is the number of sets that can be made up from the n given things, each set containing k different things and no two sets containing exactly the same k things.

The number of **combinations of n different things, taken k at a time, with repetitions** is the number of sets that can be made up of k things chosen from the given n things, each being used as often as

desired.

The number of different combinations of n different things taken, k at a time, without repetitions, is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)...(n-k+1)}{1\times 2...k}$$

and the number of those combinations with repetitions is

$$\binom{n+k-1}{k}$$

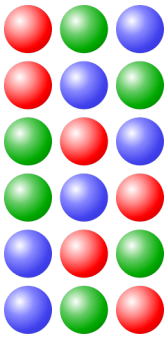


Figure 22: All ordering combinations of 3 different balls.

8.6 Random Variables:

A **random variable** X is a function defined on the sample space S of an experiment. Its values are real numbers. For every number a the probability $P(X = a)$

with which X assumes a is defined. Similarly, for any interval I the probability $P(X \in I)$

with which X assumes any value in I is defined.

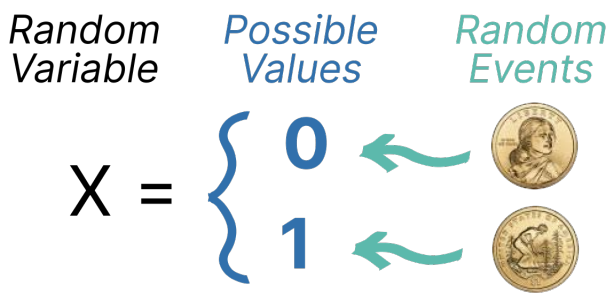


Figure 23: Random Variable.

8.7 Discrete Random Variables and Distributions:

By definition, a random variable X and its distribution are discrete if X assumes only finitely many or at most countably many values x1, x2, x3, . . ., called the possible values of X, with positive probabilities.

The discrete distribution of X is also determined by the probability function f(x) of X, defined by $f(x) = \begin{cases} p_j & x = x_j \\ 0 & otherwise \end{cases}$

From this we get the values of the distribution function F(x) by taking sums,
$F(x) = \sum_{x_j \leq x} f(x_j) = \sum_{x_j \leq x} p_j$

8.8 Continuous Random Variables and Distributions:

Discrete random variables appear in experiments in which we count. Continuous random variables appear in experiments in which we measure. By definition, a random variable X and its distribution are of continuous type or, briefly, continuous, if its distribution function F(x) can be given by an integral

$$F(x) = \int_{-\infty}^{\infty} f(x)dx$$

The integrand f(x) is called **density** of the distribution, is nonnegative, and is continuous perhaps except for finitely many x-values. Differentiation gives the relation of f to F as

$$f(x) = F'(x)$$

The formula for the continuous probability corresponding to an interval is:
$P(a < X \leq b) = F(b) - F(a) = \int_a^b f(x)dx$

8.9 Mean and Variance of a distribution:

In probability and statistics, the **mean**, or **expected value**, is a measure of the central tendency either of a probability distribution or of a random variable characterized by that distribution. The mean is defined by

Discrete distribution $\mu = \sum_j x_j f(x_j)$

Continous distribution $\mu = \int_{-\infty}^{\infty} xF(x)dx$

In probability theory and statistics, **variance** is the expectation of the squared deviation of a random variable from its mean. **Variance** is a measure of dispersion, meaning it is a measure of how far a set of numbers is spread out from their average value. The **variance** is defined by

Discrete distribution $\sigma^2 = \sum_j (x_j - \mu)^2 f(x_j)$

Continous distribution $\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x) dx$

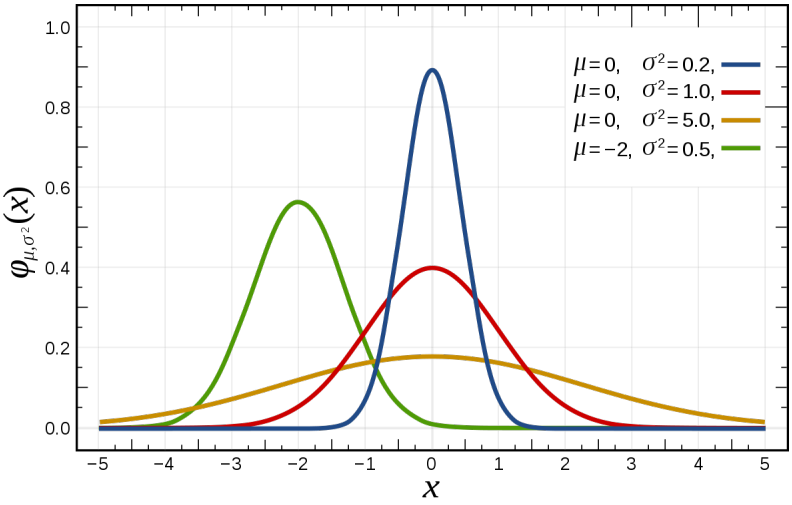


Figure 24: Mean and Variance of Normal Distribution.

8.10 Naive Bayes:

In statistics, naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong independence assumptions between the features. They are among the simplest Bayesian network models, but coupled with kernel density estimation, they can achieve higher accuracy levels.

Naive Bayes algorithm is one of the oldest forms of Machine Learning. The Bayes Theory (on which is based this algorithm) and the basics of statistics were developed in the 18th century. Since them until in 50' all the computations were done manually until appeared the first computer implementation of this algorithm.

Now let's see how everything that we learned before comes together in the Gaussian Naive Bayes algorithm:

First as we seen in the topics about Linear and Logistic Regression the Machine learning algorithms are taking into account the information from features to create a classification or regression example. In the same

way we should find a way to to the same in with probability. The answer to this problem is the Naive Bayes Theory. Shortly we can express it by the following formula:

$$P(c|x) = \frac{P(x|c) \times P(c)}{x)}$$

where,

- P(c—x) is the posterior probability of the class c given the feature vector x;
- P(c) is the prior probability of class c;
- P(x—c) is the likelihood which is the probability of feature vector given c;
- P(x) if the prior probability of the feature vector.

However we can simplify this formula because of the fact the P(x) remains the same in all cases, the new can express the formula above in the following form:

$$P(c|x) = P(x_1|c) \times P(x_2|c) \times ... \times P(x_n|c) \times P(c)$$

Now we can compute the probability of a sample belonging to class c. However this would work only for the categorical features, however the majority of data now a days is represented in an numerical way, so we would need a way to represent the probability of find a sample in a series, Here come the Normal (Gaussian) Distribution.

In probability theory a normal (or Gaussian) distribution is a type of continuous probability distribution for a real-valued random variable. The most important about it is that it allows us to find the probability of finding a sample in series knowing only the mean and standard deviation of the series. We can do this by using the following formula.

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

where:

- μ is the mean value of the series.
- σ is the standard deviation of the series.

so to have a working algorithm we need to store these values during the train phase of the algorithm and use them during the prediction phase:

```
from collections import OrderedDict
from functools import reduce
from operator import mul
class Gauss:

    def fit(self, X : np.array, y : np.array) -> Gauss:
        '''
            The fitting function of the gaussian naive
    bayes model.
        :param X: np.array
            The feature matrix.
        :param y: np.array
            The target vector.
        :return: Gauss
            This function return the object of the trained
     model.
        '''
        # Preparing the mean, standard deviation and
    priors vectors.
        self.mu = OrderedDict()
        self.std = OrderedDict()
        self.priors = OrderedDict()

        # Computing the mean, standard deviation and prior
     vectors
        for every class.
        for cls in np.unique(y):
            self.priors[cls] = len(y[y==cls]) / len(y)
            self.mu[cls] = np.mean(X[np.where(y==cls)],
    axis=0)
            self.std[cls] = np.std(X[np.where(y==cls)],
    axis=0)
        return self
```

In the listing above we are computing the priors to because we are going to use them during the predicting phase.

Also we need a function that implements the normal distribution

function:

```
 def normal_distribution(self, x : np.array, cls : str,
i : int) -> float:
      '''
          The normal distribution formula.
      :param x: np.array
          The sample for which we want to find the
probability.
      :param cls: str or int
          The class for which we want to compute the
probability.
      :param i: int
          The index of the feature.
      :return: float
          The probability of the sample for the normal
distribution.
      '''
      exponent = np.exp(-((x[i] - self.mu[cls][i]) ** 2
      exponent /= (2 * self.std[cls][i] ** 2)))
      return (1 / (np.sqrt(2 * np.pi) * self.std[cls][i
]**2)) * exponent
```

Now let's explore the predict_proba function. It is responsible for computing the probability of every sample to belong to every class using the formula from the second definition of Naive Bayes Theorem:

```
 def predict_proba(self, X : np.array) -> np.array:
      '''
          This function returns the probability for
          every sample in the data set.
      :param X: np.array
          The feature matrix passed to make predictions
on.
      :return: np.array
          An array with the probabilities for every
class for every sample.
      '''
      # Creating the empty list with probabilities.
      y_pred = []

      # Computing the probabilities for every class for
```

```
every sample.
        for x in X:
            y_pred.append([])

            # computing the probability for every class
for this sample.
            for cls in self.priors:
                prob = reduce(mul, [self.
normal_distribution(x, cls, i)
                                    for i in range(len(x))
]) * self.priors[cls]
                y_pred[-1].append(prob)

            y_pred[-1] = np.array(y_pred[-1])

            # Normalizing the vector.
            for i in range(len(y_pred[-1])):
                y_pred[-1] = y_pred[-1] / np.linalg.norm(
y_pred[-1])
        return np.array(y_pred)
```

This function at the end is normalizing every probability vector because the output of the Naive Bayes formula isn't between 0 and 1.

Now that we can compute the probabilities we can use it to predict the classes of every sample:

```
    def predict(self, X : np.array) -> np.array:
        '''
            This function returns the predicted class for
            every sample in the data set.
        :param X: np.array
            The feature matrix passed to make predictions
on.
        :return: np.array
            An array with the predicted classes for every
            class for every sample.
        '''
        # Creating the empty list for storing the
predicted classes.
        y = []
```

```
        # Getting the predicted probabilities for every
    passed sample.
        probas = self.predict_proba(X)

        # Getting the class with the highest probability
    for every sample.
        for pr in probas:
            y.append(
                list(self.priors.keys())[np.argmax(pr)]
            )
        return y
```

This implementation on Heart Disease UCI standardised data set data set gives us 0.855 accuracy, while the one implemented in sklearn - 0.881:

Let's take a look how to use the GaussianNB model from sklearn:

```
# Importing the model.
from sklearn.naive_bayes import GaussianNB

# Creating the model object.
gauss = GaussianNB()

# Fitting the model.
gauss.fit(X_train_scaled, y_train)

# Making predictions on new samples.
y_pred = gauss.predict(X_test_scaled)
```

8.11 Conclusion:

During this topic we learned the basics of probability theory and how it is used to build a strong classification model - Gaussian Naive Bayes. In the next topic we are going to see how the same principles can be used to create a clustering model. What is a clustering model? Let's see together.

9 Random Forest

Up to this point, you might have got familiar with Decision Tree, a Supervised learning technique with the structure of a tree. In this chapter, we will talk about Random Forest, a machine learning technique used to solve regression and classification problems.

Random Forest uses ensemble learning, a method that combines several classifiers to offer answers to challenging issues. As the name suggests, *"Random Forest uses many decision trees on different subsets of a given dataset and averages the results to increase the dataset's predicted accuracy."*. Instead of depending on a single decision tree, the random forest uses predictions from each tree to predict the ultimate result based on forecasts that received the majority of votes. A Random Forest eradicates the limitations of a decision tree algorithm reduces the overfitting of datasets and increases precision, making it more accurate than the decision tree algorithm.

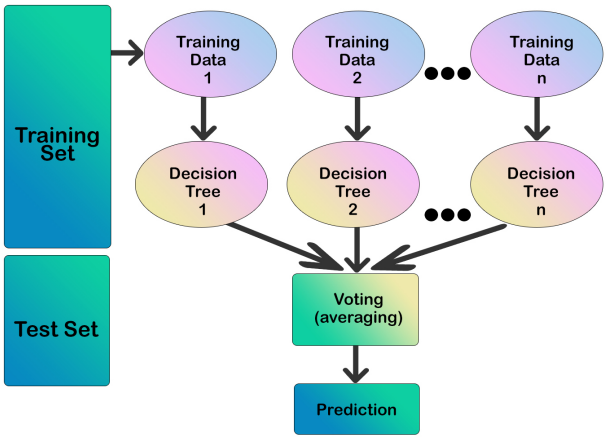


Figure 25: How Random Forest works.

Some decision trees may predict the proper output while others may not, but collectively, all the trees predict the correct outcome since the Random Forest mixes many trees to predict the class of the dataset. We select Random Forest because it runs quickly, requires less training time than other algorithms, predicts results with high accuracy even for massive datasets, and can maintain accuracy even when a significant amount of the data is missing.

Random Forest works in two-phase, the first is to create the random forest by combining N decision trees, and the second is to make predictions for each tree created in the first phase. Even if Random Forest can

be used for classification and regression tasks, it is not more suitable for Regression tasks.

9.1 How it works

Random Forest algorithms have three primary hyperparameters, which must be set before training. These include **node size**, the **number of trees**, and the **number of features** sampled. From there, the Random Forest can be used to solve regression or classification problems.

The Random Forest algorithm is made up of a collection of decision trees where each tree in the ensemble is comprised of a data sample drawn from a training set with a replacement called the bootstrap sample. Of that training sample, one-third of it is set aside as test data, known as the out-of-bag (oob) sample. Another instance of randomness is injected through feature bagging, adding more diversity to the dataset and reducing the correlation among decision trees. Depending on the problem type, the prediction's determination will vary. For a regression task, the individual decision trees will be averaged, and for a classification task, a majority vote—i.e. the most frequent categorical variable—will yield the predicted class. Finally, the oob sample is used for cross-validation, finalising that prediction.

A useful feature selection indicator is provided by Random Forest. The model includes an additional variable from Scikit-Learn that displays each feature's relative weight or contribution to the prediction. In the training phase, it automatically calculates the relevance score of each feature. The relevance is then scaled down until all scores add up to 1. For the purpose of building a model, this score will assist you in selecting the most crucial features and eliminating the less crucial ones.

The importance of each feature is determined by Random Forest using the gini importance or mean decrease in impurity (MDI). Gini importance, also referred to as the total decrease in node impurity indicates how much the model's fit or precision declines when a variable is removed. The significance of the variable increases with the size of the decrease. The mean decline is a crucial factor when choosing variables in this situation. The overall explanatory power of the variables can be expressed using the Gini index.

9.2 Random Forest Classifier

The Random Forest Classifier creates a set of decision trees from a randomly selected subset of the training set. A group of decision trees (DT) are created from a subset of the training data chosen at random, and the final prediction is made by adding up the votes from each DT.

9.2.1 Implementation

Consider the well-known classification dataset known as the iris flower dataset. It includes the sepal length and width, petal length and width, and type of flowers. This set has the following classes: Setosa, Versicolor, and Virginia. We will build a model to classify the type of flower. The dataset can be downloaded from the UCI Machine Learning Repository or found in the scikit-learn library. But sklearn's Random Forest does not automatically deal with the missing values. If the algorithm encounters any NaN or Null values in your data, it will return an error. Please do not disregard the EDA in general. Studying your data, normalising it, and handling the categorical features and missing values before you even start training is always better.

We will start by importing the datasets library from scikit-learn and load the iris dataset with *load_iris()*.

```
In:
#Import scikit-learn dataset library
from sklearn import datasets

#Load dataset
iris = datasets.load_iris()

# Creating a DataFrame of given iris dataset.
import pandas as pd
data=pd.DataFrame({
    'sepal length':iris.data[:,0],
    'sepal width':iris.data[:,1],
    'petal length':iris.data[:,2],
    'petal width':iris.data[:,3],
    'species':iris.target
})
```

First, we separate the columns into dependent and independent variables (features and labels). Then we split those variables into a training

and test set.

```
In:
# Import train_test_split function
from sklearn.model_selection import train_test_split

X=data[['sepal length','sepal width','petal length','petal
   width']]# Features
y=data['species']  # Labels

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.3)
```

After splitting, we will train the model on the training set and perform predictions on the test set.

```
In:
#Import Random Forest Model
from sklearn.ensemble import RandomForestClassifier

clf=RandomForestClassifier(n_estimators=100)

#Train the model using the training sets y_pred=clf.
   predict(X_test)
clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)
```

After training, check the accuracy using actual and predicted values.

```
In:
#Import scikit-learn metrics module for accuracy
   calculation
from sklearn import metrics
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

Out:
('Accuracy:', 0.93333333333333335)
```

We can also predict a single item; for example, sepal length is 3, sepal width is 5, petal length is 4, and petal width is 2. We can now predict what kind of flower it is.

```
In:
clf.predict([[3, 5, 4, 2]])

Out:
array([2])
```

Here, 2 indicates the flower type Virginica.

9.2.2 Finding Important Features in Scikit-learn

Here, we find essential features in the IRIS dataset. We can complete this task using scikit-learn by following these steps:

- First, we need to create a Random Forest model
- Second, we use the feature importance variable to see feature importance scores
- Third, visualise the scores using the seaborn library

```
In:
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=100)

#Train the model using the training sets y_pred=clf.
   predict(X_test)
clf.fit(X_train,y_train)

Out:
RandomForestClassifier(bootstrap=True, class_weight=None,
   criterion='gini',
            max_depth=None, max_features='auto',
   max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=
   None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100,
    n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

```
In:
import pandas as pd
feature_imp=pd.Series(clf.feature_importances_,
index=iris.feature_names).sort_values(ascending=False)
feature_imp

Out:
petal width (cm)     0.458607
petal length (cm)    0.413859
sepal length (cm)    0.103600
sepal width (cm)     0.023933
dtype: float64
```

We can also visualise the feature's importance. Visualisations are easy to understand and interpretable. For visualization, we can use a combination of **matplotlib** and **seaborn**. Because Seaborn is built on top of matplotlib, it offers many customised themes and additional plot types. Matplotlib is a superset of seaborn, and both are equally important for good visualisations.

```
In:
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
# Creating a bar plot
sns.barplot(x=feature_imp, y=feature_imp.index)
# Add labels to your graph
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.title("Visualising Important Features")
plt.legend()
plt.show()

Out:
Figure 31
```

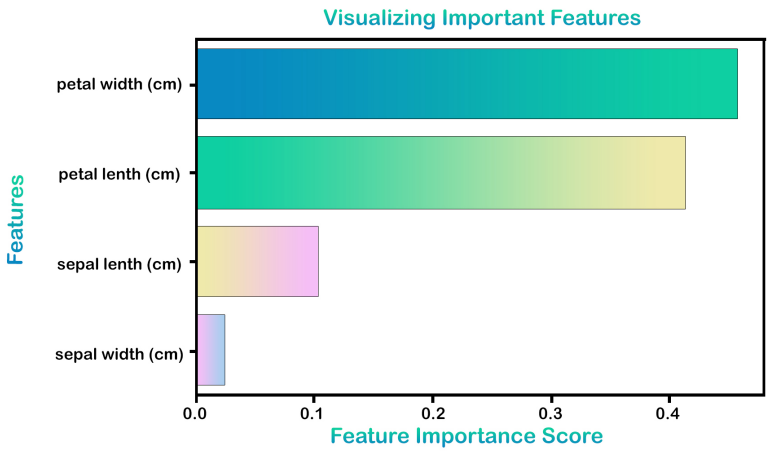


Figure 26: Feature Importancy.

9.2.3 Generating the Model on Selected Features

Here, we can remove the "sepal width" feature because it has very low importance and select the three remaining features.

In:

```
# Import train_test_split function
from sklearn.cross_validation import train_test_split
# Split dataset into features and labels
X=data[['petal length', 'petal width','sepal length']]
# Removed feature "sepal length"
y=data['species']
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.70, random_state=5) # 70% training and 30%
   test
```

After splitting, you will generate a model on the selected training set features, perform predictions on the selected test set features, and compare actual and predicted values.

In:

```
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=100)
```

```
#Train the model using the training sets y_pred=clf.
   predict(X_test)
clf.fit(X_train,y_train)

# prediction on test set
y_pred=clf.predict(X_test)

#Import scikit-learn metrics module for accuracy
   calculation
from sklearn import metrics
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

Out:
('Accuracy:', 0.95238095238095233)
```

After removing minor essential features (sepal length), we can see that the accuracy increased. It happened because we pulled misleading data and noise, increasing accuracy. A lesser amount of features also reduces the training time.

9.3 Random Forest Regressor

Each decision tree has a high variance, but when we combine them simultaneously, the resulting variance is low. This occurs because each decision tree is perfectly trained on the specific sample data, so the output depends on multiple decision trees rather than just one. The final outcome of a classification problem is taken using the majority voting classifier. In the case of a regression problem, the final result is the mean of all the outcomes.

Let us take a regression problem. link here you can find the dataset that was used. Fortunately, it is quite easy to use Random Forest, as the sklearn library has the algorithm implemented both for Regression and Classification tasks. In the previous section we used **RandomForestClassifer()**, but now we will use **RandomForestRegressor()**.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load dataset
```

```
dataset = pd.read_csv('Position_Salaries.csv')

X = dataset.iloc[:,1:2].values
y = dataset.iloc[:,2].values

# Split dataset into training set and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.3, random_state=0)
```

After splitting, we will train the model on the training set and perform predictions on the test set.

```
# Import Random Forest Model
from sklearn.ensemble import RandomForestRegressor

# for 20 trees
regressor = RandomForestRegressor(n_estimators = 20,
   random_state = 0)
regressor.fit(X_train, y_train)

#Train the model using the training sets
y_pred=regressor.predict(X_test)
```

The most important parameter of the **RandomForestRegressor** class is the **n_estimators** parameter. This parameter defines the number of trees in the random forest. We will start with **n_estimator=20** to see how our algorithm performs. The last and final step of solving a machine learning problem is evaluating the algorithm's performance. For regression problems, the metrics used to evaluate an algorithm are mean absolute error, mean squared error, and root mean squared error. Execute the following code to find these values:

```
from sklearn import metrics

print('Mean Absolute Error:', metrics.mean_absolute_error(
   y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(
   y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.
   mean_squared_error(y_test, y_pred)))
```

9.4 Conclusions:

To sum up everything mentioned above, you should remember that Random Forest Algorithm can handle binary, continuous, and categorical, but is more likely to use for classification tasks. Compared to decision trees, where decisions can be made by following the path of the tree, Random Forest is more complex; it requires more training time as each decision tree has to generate output for the given input data. Random Forest is an excellent choice if anyone wants to build the model fast and efficiently, as one of the strengths of Random Forest is it can handle missing values. Overall, random forest is a fast, simple, flexible, and robust model with some limitations.

10 Support Vector Machine

Support Vector Machine (SVM) is one of the most popular Supervised Learning algorithms, which is used for both Classification and Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This algorithm aims to find an optimal hyperplane for linearly separable patterns, but can also be extended to non-linearly separable patterns by appplying kernel functions.

For example, by using Gaussian RBF Kernel we can shift the points from a 2D plane to a 3D plane by just shifting all the green points above the red ones by using a mapping function like gaussian RBF, which introduces a hyperplane in the given space, as shown in the image below.

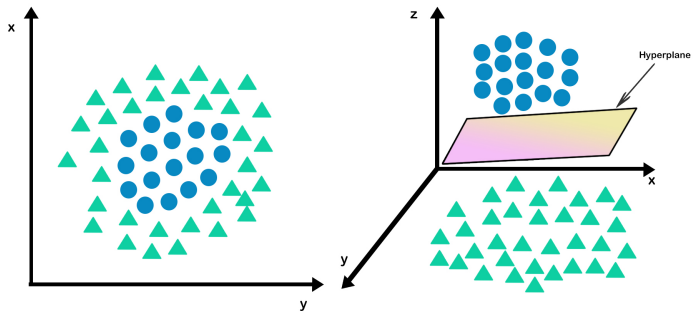


Figure 27: Gaussian RBF Kernel for non-linearly separable data.

The whole SVM revolves around its support vectors. Support vectors are the data points that lie closest to the decision surface (or hyperplane) and which help us build our SVM. They are the data points most difficult to classify. In the image below, they are represented as the outlined samples. Support vectors influence the position and orientation of the hyperplane. Using them, the margin of the classifier is maximized. Thus, deleting the support vectors would result in a different position of the hyperplane.

The resulting learning algorithm is an optimization algorithm rather than a greedy search. In contrast with Linear regression and Neural nets, whose optimality is influenced by all points, in the case of SVM the optimality is influenced only by "difficult points", which are the ones close

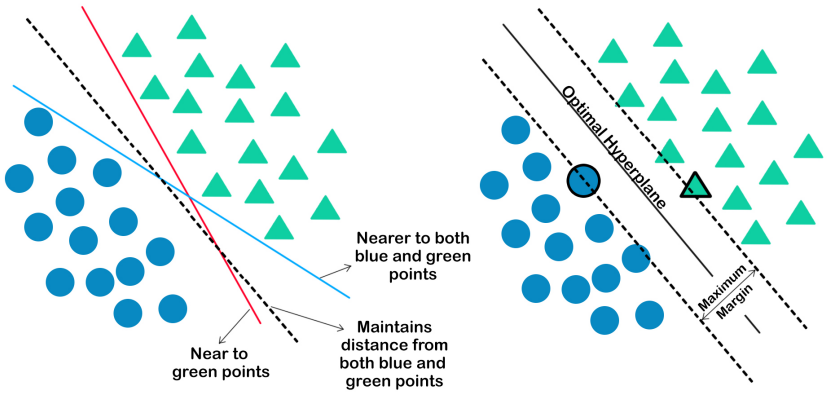


Figure 28: The optimal hyperplane.

to the decision boundary. SVM has a more balanced boundary between categories than Logistic Regression and is less sensitive to outliers.

10.1 Finding the right hyperplane

SVM uses the widest street approach. In other words, it tries to maximize the distance between its positive and negative class points while choosing its decision boundary. Wider margin ensures less overfitting of the model and makes the model more certain, by overcoming the problems which can appear when classifying the points nearer to the decision boundary. The decision boundary represents a hyperplane. Hyperplanes are planes in a lower dimensional space than the given input data, which separate the data into classes. If a space is 3-dimensional, then the hyperplane is a 2-dimensional plane. If the space is 2-dimensional (as in the image below), its hyperplane is a 1-dimensional line.

The decision surface H separating the classes is a hyperplane of the form:

$$wx + b = 0 \tag{1}$$

Where w is a weight vector, x - the input vector and b - the bias. Initially both w and b are unknowns. Optimal values of w and must be found.

In the same way, the planes H_1 and H_2 can be expressed as it follows: H_1: $wx_i + b = +1$ and H_2: $wx_i + b = -1$.

Following this logic, when a point x belongs to the positive class, the value of this expression would be:

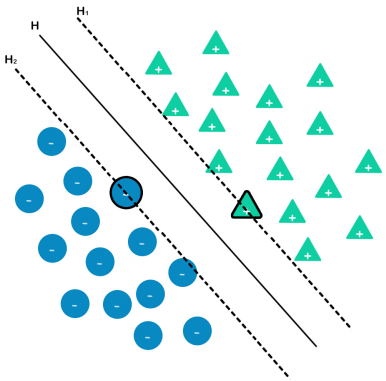


Figure 29: Negative and positive classes separated by planes.

$$wx_+ + b \geq +1 \tag{2}$$

and when x belong to the negative class:

$$wx_- + b \leq -1 \tag{3}$$

For mathematical convenience, we introduce a variable y, where $y_i = +1$ for positive samples and $y_i = -1$ for negative samples.
Multiplying (2) and (3) with the corresponding y_i, we obtain the same equation in both cases:

$$y_i(wx_i + b) \geq +1 \tag{4}$$

resulting in $y_i(wx_i + b) - 1 \geq 0$. In cases when the given sample is in the gutter (between H_1 and H_2), we obtain the following constraint:

$$y_i(wx_i + b) - 1 = 0 \tag{5}$$

In order to maximize the margins, we must first find a way to express them. If we do some clever math using vector properties, we discover that the distance between H_1 and H_2 can be expressed as $\frac{2}{\|\mathbf{w}\|}$, which needs to be maximized.

$$MAX\frac{2}{\|\mathbf{w}\|} = MAX\frac{1}{\|\mathbf{w}\|} = MIN\|\mathbf{w}\| = MIN\frac{1}{2}\|\mathbf{w}\|^2 \tag{6}$$

Now we need to find the extremum of the obtained expression.

10.2 Lagrange Multipliers

In mathematical optimization, the method of Lagrange multipliers is a strategy for finding the local maxima and minima of a function subject to equality constraints. After applying this method on (6) with respect to the constraint obtained in (5), we get the following formula (The primal form of the optimization problem):

$$\mathcal{L} = \frac{1}{2}\|\mathbf{w}\|^2 - \sum \alpha_i[y_i(wx_i + b) - 1] \tag{7}$$

where $\mathcal{L}$ is the Lagrangian and α_i - the Lagrange multiplier.
Applying the property that the derivatives at min are equal to 0, we get:

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum a_i y_i x_i = 0 \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum a_i y_i = 0 \tag{9}$$

which means that our final solution must satifly these two relationship:
1. $w = \sum a_i y_i x_i$
2. $\sum a_i y_i = 0$

This is the dual form of the optimization problem and the solution we find here will be the same as the solution to the original problem. The second expression can be solved for a, and by knowing the a_i we can find the weights w for the maximal margin separating hyperplane using the first relationship.
After training and finding the w by this method, given an unknown point u, measured on features of x_i, we can classify it by lookig at the sign of

$$f(x) = wu + b = (\sum a_i y_i x_i \cdot u) + b \tag{10}$$

If the sign of (10) is ≥ 1, then the point u belongs to the positive class, otherwise it belongs to the negative class.

10.3 Implementation

Below you can find a from scratch implementation of SVM which uses the Lagrange multipliers approach.

```
class SVM:
    # define a class to store our functions in
```

```
 def __init__(self, lr, n_iter = 1000):
     """
         The constructor of the SVM model.
             :param lr: float, default = 0.001
                 The learning rate of the model
             :param n_iter: int, default 1000
                 The number of iteration during which
the model is trained
     """
     # setting up the hyperparameters
     self.lr = lr
     self.n_iter = n_iter

 def get_cost_grads(self):
     """
         Function to calculate the cost for the given
normal vector w
         and find the gradient of the objective
function with respect to w
             :param lr: float, default = 0.001
                 The learning rate of the model
             :param n_iter: int, default 1000
                 The number of iteration during which
the model is trained
             :return:

     """
     # calculate the distance of the point
     distances = self.y * (np.dot(self.X, self.w)) - 1

     # if distance is more than 0, sample is not on the
 support vector
     # Lagrange multiplier will be 0
     distances[distances > 0] = 0

     # Get current cost using Lagrangian
     L = 1 / 2 * np.dot(self.w, self.w) - np.sum(
distances)

     # init an array filled with 0s - the gradient
```

```
        dw = np.zeros(len(self.w))

        # The dw updating process
        for ind, d in enumerate(distances):

            # if sample is not on the support vector
            if d == 0:
                di = self.w
            # if sample is on the support vector
            else:
                di = self.w - (self.y[ind] * self.X[ind])

            dw += di

        return L, dw / len(X)

    def fit(self, X, y):
        """
            The fit function of the model
                :param X: 2-d np.array
                    The matrix with features
                :param y: 1-d np.array
                    The target vector
        """
        self.X = X
        self.y = y

        # Add column vector of ones for computational
convenience
        self.X = np.column_stack((np.ones(len(X)), X))

        # initialize a vector filled with ones for storing
 the weights
        self.w = np.ones(len(self.X[0]))

        # the weights updating process
        for i in range(self.n_iter):
            L, dw = self.get_cost_grads()
            self.w = self.w - self.lr * dw
```

```
    def predict(self, X):
        """
            The predict function of the model
                :param X: 2-d np.array
                    The matrix with features for which the
    model must make the predictions
                :return: 1-d np.array
                    The predicted values
        """
        # Add column vector of ones
        X = np.column_stack((np.ones(len(X)), X))

        # get the sign (-1 or 1); -1 - class 0, 1 - class
   1
        mm = np.sign(X @ self.w)

        # getting the values for y_pred
        y_pred = np.where(mm <= 0, 0, 1)

        return y_pred
```

Testing the model on a linearly separable dataset:

```
In:
# creating a linearly separable set of data of 2000
   samples
X, y =  datasets.make_blobs(n_samples=2000, n_features=2,
        centers=2, cluster_std=1.05, random_state=40)

# separating the dataset into X_train and X_test
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size = 0.3)

# initializing and training the model
clf = SVM(lr=1e-3)
clf.fit(X_train, y_train)

# making the predictions
y_pred = clf.predict(X_test)

# obtaining the accuracy score
```

```
accuracy = accuracy_score(y_pred, y_test)
print(accuracy)

Out:
0.9983333333333333
```

Now let's try to use the sklearn model on the same dataset.

```
# importing the linear SVM model for classification from
   sklearn
from sklearn.svm import LinearSVC

# training the model
clf = LinearSVC()
clf.fit(X_train, y_train)

# making the predictions and obtaining the accuracy
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_pred, y_test)
```

For both models the obtained accuracy was 0.9983333333333333. However, you're encouraged to look at other SVM algorithms from sklearn and test them on your data, as well as to build your own version from scratch.

10.4 Conclusion

In this chapter we've learned about SVMs and the main idea behind them. It is one of the supervised algorithms which is mostly used for classification. It's main advantages are that it can be used on text, audio, images, etc., even on data that is not regularly distributed or have unknown distribution. SVMs don't generally suffer condition of overfitting and performs well when there is a clear indication of separation between classes. It is also less influenced by outliers. SVM can be extended to solve nonlinear classification tasks when the set of samples cannot be separated linearly. By applying kernel functions, the samples are mapped onto a high-dimensional feature space, in which the linear classification is possible.
But there are also some downsides of it: it is not suitable for very large data sets and it is not very efficient for data sets which have many outliers.

11 Missing Values

Usually when we work with datasets that were not cleaned for us beforehand, there are missing values. We note them as NaN (Not a Number). Also NaNs may represent undefined values, which we may obtain if an undefined behavior was performed (like division by 0). NaN values usually appear due to human error. In fact common sources of NaN values in a dataset are: missing data, join of two tables of data, wrong calculations, human factor (forgot to write, or wrote in the wrong format), failures in reading data from sensors. In some cases you may stumble upon None denoting missing or undefined values.

The difference between NaN and None is that NaN is a floating point number, while None is an object. You can perform mathematical operations with NaN, while mathematical expression with None will throw TypeErrors. Take a look at this behavior:

```
In:
import numpy as np
np.nan + 1
```

This outputs:

```
nan
```

If we do the same operation with None we'll get an error:

```
In:
None + 1
```

Out:

```
---------------------------------------------------------
TypeError Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_11468/1875925809.py
in <module> ----> 1 None + 1

TypeError: unsupported operand type(s) for +: 'NoneType'
and 'int'
```

Don't confuse NaN with 0 though, because 0 is a numerical value with a specific meaning, while a NaN value represents the missing of any value and, therefore, any meaning.

Why don't we like NaNs ?

Because some statistical values, such as mean or median, are not being assigned the correct value, because we were not able to calculate them

properly for every sample, since some values are missing. As a result we may run into wrong conclusions during Exploratory Data Analysis. Also, most algorithms implemented in sklearn (an not only from sklearn) cannot operate with NaN values. For example, we cannot train a model of Logistic Regression with NaN values present in our dataset. Try it ! Identifying NaN values in a DataFrame

Pandas comes in with the utilities to help identify and work with NaN values. More details can be found in pandas documentation.

Let's import a dataset to work on a specific example:

```
In:
df = pd.read_csv('../melb_data.csv')
df.info()
```

We can already see that there are NaN values:

```
Out:
<class 'pandas.core.frame.DataFrame'>\\
RangeIndex: 13580 entries, 0 to 13579
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Suburb         13580 non-null  object
 1   Address        13580 non-null  object
 2   Rooms          13580 non-null  int64
 3   Type           13580 non-null  object
 4   Price          13580 non-null  float64
 5   Method         13580 non-null  object
 6   SellerG        13580 non-null  object
 7   Date           13580 non-null  object
 8   Distance       13580 non-null  float64
 9   Postcode       13580 non-null  float64
 10  Bedroom2       13580 non-null  float64
 11  Bathroom       13580 non-null  float64
 12  Car            13518 non-null  float64
 13  Landsize       13580 non-null  float64
 14  BuildingArea   7130 non-null   float64
 15  YearBuilt      8205 non-null   float64
 16  CouncilArea    12211 non-null  object
 17  Lattitude      13580 non-null  float64
 18  Longtitude     13580 non-null  float64
 19  Regionname     13580 non-null  object
```

```
 20  Propertycount  13580 non-null  float64
dtypes: float64(12), int64(1), object(8)
memory usage: 2.2+ MB
```

Some columns have less non-null values than the others. This means we have NaNs in these columns.

In pandas there is the method .isna() that takes as an argument an array or a value and outputs True if we have NaNs present. It works on both Series and DataFrame.

```
In:
pd.isna(0)

Out:
False
```

Let's try a string:

```
In:
pd.isna('abcd')

Out:
False
```

Let's try with pandas' own version of NaN. It works not only with arithmetic operations as np.nan, but also with comparison operators.

```
In:
pd.isna(pd.NA)

Out:
True
```

With numpy's implementation we get the same result obviously:

```
In:
pd.isna(np.nan)

Out:
True
```

Let's try an array now:

```
In:
# in my case: array([0.25625595, 0.09247772, 0.58931643,
0.77828947, 0.75434765])
arr = np.random.random(5)
np.isna(arr)
```

```
Out:
array([False, False, False, False, False])
```

Add a NaN value and check again:

```
In:
# [nan 0.09247772 0.58931643 0.77828947 0.75434765]
arr[0] = np.nan
pd.isna(arr)

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

It returns a boolean mask which we can use to select NaN values from the array, or the non NaN values.

```
In:
mask = pd.isna(arr)
arr[mask]

Out:
array([nan])
```

It returns the only NaN value we have in the array. Let's do the opposite.

```
In:
arr[~mask]

Out:
array([0.09247772, 0.58931643, 0.77828947, 0.75434765])
```

We are using the ~(tilde) operator. It is an unary operator that inverses the bits of the given variable (note that a boolean mask is basically a binary array).

Pandas Series also have the .hasnans attribute that tells if a Series has NaN values. Let's look at it in action. Here is a function that tells the percentage of NaN values to total values by columns:

```
In:
def get_nans_percentage(X):
    columns_with_nan_values = {}

    for column in df:
        if X[column].hasnans:
            not_nans_count = X[column].count()
```

```
            X_len = len(X[column])

            nans_percentage = (X_len - not_nans_count) /
    X_len * 100
            nans_percentage = float("{:.2f}".format(
    nans_percentage))

            columns_with_nan_values[str(column)] =
    nans_percentage

     return columns_with_nan_values

get_nans_percentage(df)
```

```
Out:
{'Car': 0.46, 'BuildingArea': 47.5, 'YearBuilt': 39.58,
'CouncilArea': 10.08}
```

Dealing with NaN values

There are several methods to deal with NaN values. In general, we have 2 options:

1. Delete samples with NaN values. This is not the best idea most often, because we lose information. A model would train on less data making it less effective. It's bad especially when we have a small dataset.

2. Imputation. Here we replace NaN values with numerical values.

Let's try the first method. We have the following length of the DataFrame:

```
In:
len(df)
```

```
Out:
13580
```

Using the .dropna() method, that removes rows or columns with nan values, we get a lot less information:

```
In:
len(df.dropna())
```

```
Out:
6196
```

Note that we didn't modify the DataFrame yet. If we wanted to, we should have used it as follows:

```
In:
df.dropna(inplace=True)
len(df)

Out:
6196
```

This is similar to many other pandas methods. By default methods in pandas are not inplace. Let's apply the function we wrote above to check if he have nans:

```
In:
get_nans_percentage(df)

Out:
{}
```

Empty brackets mean we have no NaN values. Great! But try training a model with the DataFrame we obtained and compare it with the ones we will obtain shortly.

Consider imputation. There are many methods we could perform imputation. Perhaps the easiest one is to impute the mean of the column (or any other statistical value like median or mode). Also we can just impute 0. Let's create a function that will impute a specific value:

```
In:
def impute_value(X, val=0):
    for column in X:
        if X[column].hasnans:
            nan_mask = pd.isna(X[column])

            # df.loc[selection criterion, column]
            X.loc[nan_mask, column] = val
```

Now, apply it:

```
In:
get_nans_percentage(df)

Out:
{'Car': 0.46, 'BuildingArea': 47.5, 'YearBuilt': 39.58,
'CouncilArea': 10.08}
```

This was the percentage before applying the just defined function. And this is after:

```
In:
impute_value(df, 0)
get_nans_percentage(df)

Out:
{}
```

Print the dataframe to see the changes. Also try changing the function in such a way as to impute the mean of the column in which we found a NaN. Do the same for median and mode. You may consider using the .fillna() method if you find it easier.

At this point it's not clear which value to impute. It may be ok to impute 0 if it is the minimal value. But this is not always the case.

The mean is alright if he have a normal distribution. Though an ideal normal distribution is never found in real life data. So we lose information anyway. Also, if we have outliers forget about mean imputation, because outliers drastically change the mean. Consider being in a bar and asking some strangers their salary and computing the mean. What if Elon Musk was one of them ? Would the mean change a lot ?

On the other hand the median and mode aren't affected by outliers. There are cases when we can afford losing some information though. Sometimes we strive for a good precision and we must look into better imputation methods.

Let's actually look at an example. We'll generate some evenly spaced numbers and check their mean:

```
In:
# linspace (start, stop, num=50) - evenly spaced numbers
on a given range
x = np.linspace(500, 1000, num=10)
y = [0] * len(x)

plt.plot(x, y, marker='o')

mean = np.mean(x)
print(mean)
plt.plot(mean, 0, marker='o', c='r')

plt.legend(['Points', 'Mean'])

Out:
Figure 28
```

We obtain this graph:

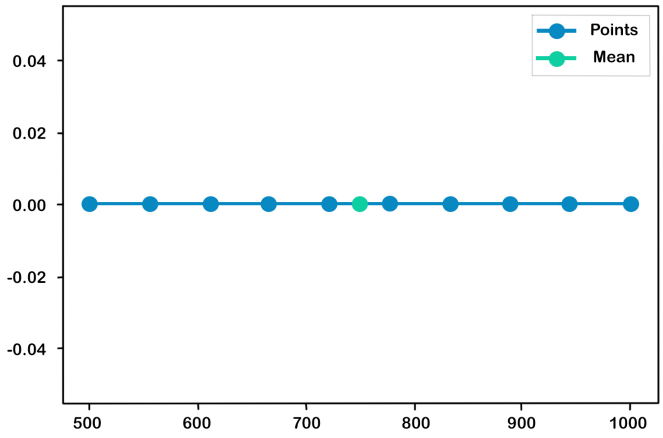


Figure 30: Mean 1.

Now, what if the first value was missing and we imputed zero ?

```
In:
x = np.linspace(500, 1000, num=10)
y = [0] * (len(x) - 1)

# missing value
x[0] = np.nan
print(x)

# imputation
x[0] = 0
print(x)

plt.plot(x[1:], y, marker='o')

mean = np.mean(x)
print(mean)
plt.plot(mean, 0, marker='o', c='r')
plt.plot(x[0], 0, marker='o', c='y')

plt.legend(['Points', 'Mean', 'Imputed value'])

Out:
Figure 29
```

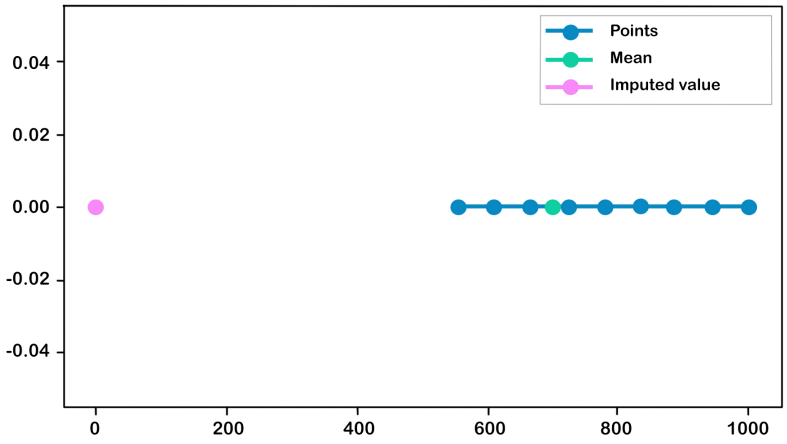


Figure 31: Mean 2.

This is the graph:

The mean is shifted. It is not at the center of the points. The imputed 0 is an outlier. What if we imputed the mean instead of 0 ?

```
In:
x = np.linspace(500, 1000, num=10)
y = [0] * (len(x) - 1)

# missing value
x[0] = np.nan
print(x)

# imputation
x[0] = np.mean(x[1:len(x) - 1])
print(x)

plt.plot(x[1:], y, marker='o')

mean = np.mean(x)
print(mean)
plt.plot(mean, 0, marker='o', c='r')
plt.plot(x[0], 0, marker='o', c='y')

plt.legend(['Points', 'Mean', 'Imputed value'])

Out:
Figure 30
```

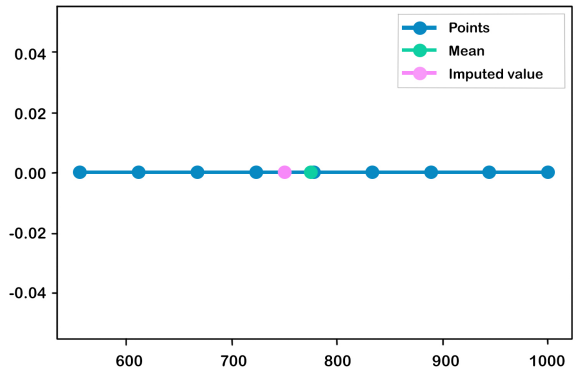


Figure 32: Mean 3.

The mean of the imputed data is closer to its true value.

Though, mean imputation is not always desirable, because some features may have the property that they must not have the mean as their values. For example, somebody with no education, with no initial income, who lives in an isolated island without Internet connection won't have an income as high as the mean of the population of a country with billionaires. He may not even have income. So imputing mean in the case of this citizen will be very wrong.

11.1 KNNI

K Nearest Neighbors Imputation is based on the KNN method of finding the k most similar (close) samples to a given sample in the dataset. sklearn has KNNI implemented. Let's look at an example:

```
In:
from sklearn.impute import KNNImputer

# remove non-numeric columns to not bother with
   preprocessing
df = df.select_dtypes('number')
imputed_df = KNNImputer().fit_transform(df)
df = pd.DataFrame(imputed_df, columns=df.columns)

get_nans_percentage(df)

Out:
{}
```

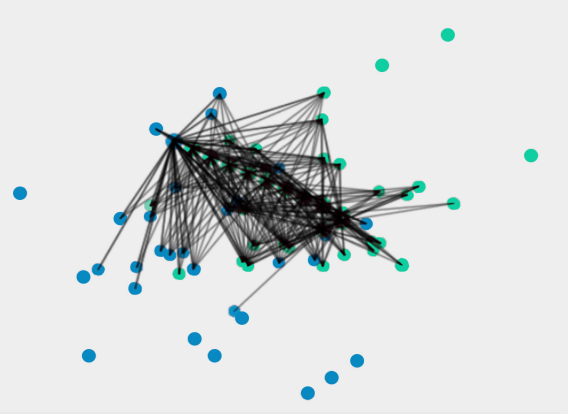


Figure 33: KNNI.

Here is an illustration of how KNNI works:
You can read more in the sklearn docs about KNNI.

11.2 MICE

Multiple Imputation by Chained Equations applies Linear Regression for every column with NaN values (using the present values) and predicts the values that should be imputed. MICE is implemented in the impyute library. Don't forget to install it: pip install impyute.

```
In:
from impyute.imputation.cs import mice as MICE

# remove non-numeric columns to not bother with
   preprocessing
df = df.select_dtypes('number')
imputed_df = MICE(df.values)
df = pd.DataFrame(imputed_df, columns=df.columns)

get_nans_percentage(df)

Out:
{}
```

Out of the methods described above MICE and KNN are the best for imputing continuous values since they use the whole dataset. Apart from these methods there are others like SICE, Cold-Deck imputation, Hot-Deck imputation, PMM etc.

11.3 Reparo

Sigmoid has its own library for imputation. Many methods mentioned above are already implemented. Let's take one to see how it works.

We'll take Cold Deck Imputation. It works by finding the nearest neighbors of a sample with nan values and replacing the missing values with the corresponding values in the selected neighbor. Then it does the same thing for other samples with NaNs.

Let's start by creating a toy DataFrame. (Don't forget to install reparo).

```
In:
import pandas as pd
import numpy as np

from reparo import CDI

df = pd.DataFrame({
    'a': [0, 1, 2, np.nan],
    'b': [np.nan, np.nan, 2, 3],
    'c': [0, 1, 2, np.nan],
    'd': [0, np.nan, 2, 3]
})

print(df)
```

We get the following DataFrame:

```
Out:
    a    b    c    d
0  0.0  NaN  0.0  0.0
1  1.0  NaN  1.0  NaN
2  2.0  2.0  2.0  2.0
3  NaN  3.0  NaN  3.0
```

We have a big percentage of NaN values. But we see that one row is free of NaNs, meaning the algorithm will be able to apply KNN. So proceed to the imputation. We'll need just two lines of code:

```
In:
imputer = CDI()
imputer.apply(df, df.columns)

print(df)
```

```
Out:
    a    b    c    d
0  0.0  2.0  0.0  0.0
1  1.0  2.0  1.0  0.0
2  2.0  2.0  2.0  2.0
3  2.0  3.0  2.0  3.0
```

As you can see all the missing values have been imputed.

Note that apply modifies the DataFrame in-place, meaning there is no turn back.

There is also the transform (and fit_transform) method that can impute a given numpy array.

11.4 Conclusion:

What should you know after reading this chapter is that the problem of NaN values comes naturally from the imperfection of humans and the systems we use to get data. You just have to know how to deal with these NaN values. This means knowing how they are represented, how can you work with them and how to get rid of them. This is usually one of the first steps when starting a machine learning project.

12 Hyperparameter tuning.

Machine Learning engineers are delighted when they get some results after working hard on data collection, data engineering, and model development, but usually, this is not the end. After having results, here comes the question - "How do I make my model better?". One of the answers to this question is Hyperparameter tuning.

A hyperparameter is a parameter whose value is used to control the learning process. Hyperparameter tuning represents tweaking these parameters to provoke little changes in an engineer's result from training a model. The goal is to get the optimal parameters, thus, improving the accuracy of the current model.

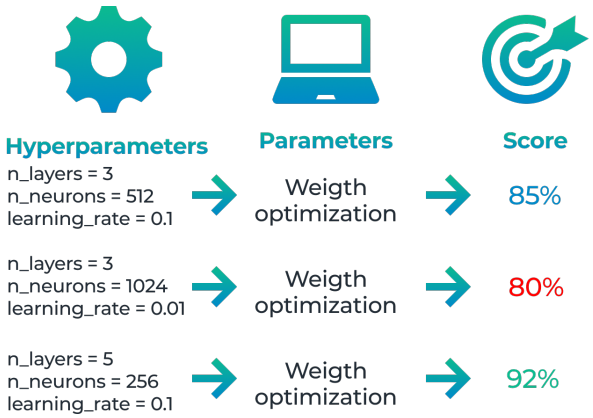


Figure 34: Hyperparameter tunning.

Here is some example of hyperparameters for different models that can be tunned:

- KNN: **n_neighbors** (K), which is the number of neighbors that will vote weights, a hyperparameter that depends on how these neighbors will be calculated.

- RandomForest: **n_estimators** meaning how many trees will be built, **max_depth** represents the maximal depth of a tree, **max_features** - the number of features to consider when looking for the best split.

- LogisticRegression: **penalty**, which is used to regularize the model (l1 or l2), **C**, inverse of regularization strength and **solver**, algorithm to use in the optimization problem.

- LinearRegression: **normalize** - if True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm.

- SVM: **C** Regularization parameter. The strength of the regularization is inversely proportional to C. **kernel** - specifies the kernel type to be used in the algorithm, can be 'linear', 'poly', 'rbf', 'sigmoid', or 'precomputed'. **gamma** - kernel coefficient for 'rbf', 'poly' and 'sigmoid'

Of course, these are not all the hyperparameters from these models that can be tuned. There are plenty of them. Every hyperparameter has a different meaning and will change the algorithm's logic in another way.

Different models have different hyperparameters, and different data require tuning different hyperparameters. Usually, complex models have many of them, and an inexperienced engineer would not know where to start.

But lucky of us, there are some methods to get this optimal combination of hyperparameters that will guarantee our win.

12.1 How to tune my parameters?

12.1.1 Grid search.

One of the most popular and easiest ways to tune your parameters is to try a lot of different combinations of them with the hope that you will find one that suits your data and gets you the best accuracy, and actually, this is what grid search does. Grid search tries different combinations of hyperparameters, but he does this in a more organized way.

Grid search takes a dictionary with the hyperparameters you want to tune as the keys and some values for these parameters that grid search will try.

An example of it will be:

```
from sklearn.model_selection import GridSearchCV
tuned_parameters = {
    'kernel': ['rbf'],
    'gamma': [1e-3, 1e-4],
    'C': [1, 10, 100, 1000]
}
```

```
clf = GridSearchCV(SVC(), tuned_parameters,scoring='recall'
   )
clf.fit(X_train, y_train)
print("Best parameters set found on development set:")
print(clf.best_params_)
```

As you might observe, GridSearch also has the parameter 'scoring', which controls which metric GridSearch will evaluate the most optimal combination of hyperparameters. In this example, our search will try to create all the combinations out of ['rbf'], [1e-3, 1e-4] and [1,10, 100, 1000], for instance, ('rbf', 1e-3, 10), ('rbf', 1e-4, 10), ('rbf', 1e-3, 1) and others. He trains the given algorithm with every combination, and as a result, you can get the best combination for the highest accuracy using the 'best_params_' property of the GridSearch instance.

12.1.2 Random Search.

Random search is very similar to GridSearch. The only difference is that it chooses all the values randomly, and luck plays its part since no intelligence is used to sample these combinations.

```
from sklearn.model_selection import RandomizedSearchCV
distributions = dict(C=uniform(loc=0, scale=4),penalty=['l2
   ', 'l1'])
clf = RandomizedSearchCV(logistic, distributions,
   random_state=0)
search = clf.fit(iris.data, iris.target)
print(search.best_params_)
```

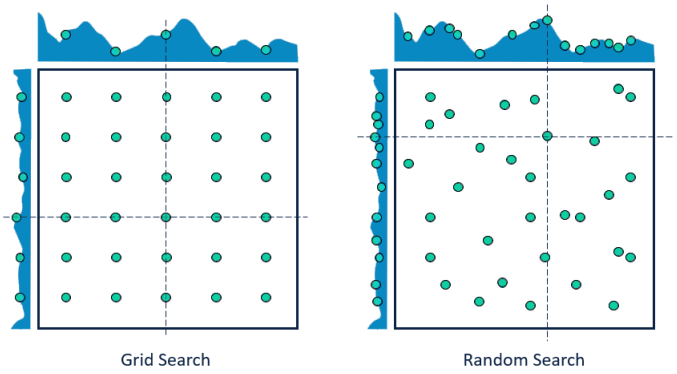


Figure 35: Grid Search vs Random Search.

12.1.3 Underfitting and Overfitting.

The GridSearch and RandomSearch don't require any logic in the parameters you choose to tune, which makes them easy to use. Now we will talk about a more intelligent way to adjust your parameters if you observe an overfitting or underfitting problem. Let's remember what overfitting and underfitting mean.

Underfitting is a scenario in machine learning where a data model cannot accurately capture the relationship between the input and output variables, generating a high error rate on both the training set and unseen data.

Overfitting is a concept in machine learning that occurs when a statistical model fits exactly the training data when the model memorizes the noise and matches too closely to the training set.

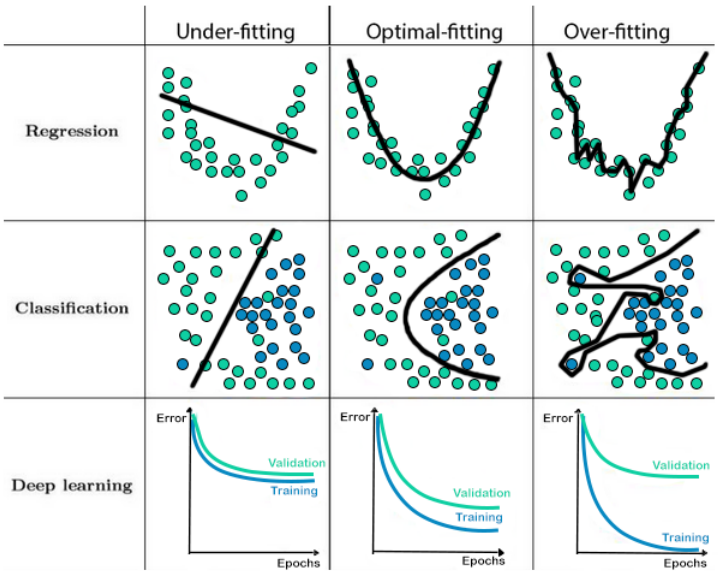


Figure 36: The representation of Under-fitting and Over-fitting in case of Regression, Classification, and Deep Learning.

Hyperparameters change different parts of the model. Some hyperparameters may make the model fit data more, others less.

Let's analyze some examples:

Random Forest:

- Increasing **max_depth** will reduce your bias, and this potentially can give you an overfitting problem because the bigger the depth, the more he tries to predict every example correctly in the dataset.

- And vice versa **n_estimators** or **min_samples_leaf** increasing it, will cause the model to generalise more, getting a bigger bias.

SVM:

- **C** - The bigger C, the bigger regularization and smaller the model bias, which directs the model to the underfitting direction.
- **Degree** - The bigger degree of the polynomial, the better it tries to predict every datapoint, so model bias gets smaller, and the bigger chance of overfitting.

12.2 Conclusion.

In this tutorial, you discovered hyperparameter optimization for machine learning in Python. While this is an essential step in modeling, it is by no means the only way to improve performance. Grid search is a handy technique that can save you time instead of tuning hyperparameters by yourself.

13 Gaussian Mixture Models. EM Algorithm

Till now you already learned some machine learning models. Linear Regression, Logistic Regression and Naive Bayes are learning from the features to predict the target column, This type of learning is named supervised learning. However labeled data is very expensive and the majority of data now a days isn't labeled. However the business need to understand data that they have and try to extract some insights from it, there come unsupervised learning. You already met the PCA algorithm, that is a dimensional reduction method. The second type of unsupervised learning is clustering.

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group than those in other group. In simple words, the aim is to segregate groups with similar traits and assign them into clusters.

It can be very useful for market segmentation. Suppose you are a store, and during your lifetime you had a lot of clients. However you would like to group them in specific groups to create specific targeted marketing campaigns. During this and the next topic we will learn about two clustering algorithms: GMM and K-Means and how they can be used to cluster a data set.

13.1 A probabilistic approach on clustering

From the rising of the Machine Learning and Artificial Intelligence fields Probability Theory was a powerful tool, that allowed us to handle uncertainty in a lot of applications, from classification to forecasting tasks. Today I would like to talk with you more about the use of Probability and Gaussian distribution in clustering problems, implementing on the way the GMM model. So let's get started.

What is GMM?

GMM (or Gaussian Mixture Models) is an algorithm that using the estimation of the density of the dataset to split the dataset in a preliminary defined number of clusters. For a better understandability, I will explain in parallel the theory and will show the code for implementing it.

For this implementation, I will use the EM (Expectation-Maximization)

algorithm.

Firstly let's import all needed libraries:

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

I highly recommend following the standards of scikit-learn library when implementing a model on your own. That's why we will implement GMM as a class. Let's also the __init__ function.

```
def __init__(self, n_components, max_iter = 100, comp_names
  =None):
    '''
        This functions initializes the model by seting the
   following
        paramenters:
        :param n_components: int
            The number of clusters in which the algorithm
  must split
            the data set
        :param max_iter: int, default = 100
            The number of iteration that the algorithm
  will
            go throw to find the clusters
        :param comp_names: list of strings, default=None
            In case it is setted as a list of string it
  will use to
            name the clusters
    '''
    self.n_componets = n_components
    self.max_iter = max_iter
    if comp_names == None:
        self.comp_names = [f"comp{index}" for index in
  range(self.n_componets)]
    else:
        self.comp_names = comp_names
    # pi list contains the fraction of the dataset for
  every cluster
    self.pi = [1/self.n_componets for comp in range(self.
  n_componets)]
```

Shortly saying, **n_components** is the number of cluster in which we want to split our data. **max_iter** represents the number of itera-

tions taken by the algorithm and **comp_names** is a list of string with **n_components** number of elements, that are interpreted as names of clusters.

The fit function:

So before we get to the EM-algorithm we must split our dataset. after that, we must initiate 2 lists. One list containing the mean vectors (each element of the vector is the mean of columns) for every subset. The second list is containing the covariance matrix of each subset.

```
def fit(self, X):
    '''
        The function for training the model
    :param X: 2-d numpy array
        The data must be passed to the algorithm as 2-d
array,
        where columns are the features and the rows are
the samples
    '''
    # Spliting the data in n_componets sub-sets
    new_X = np.array_split(X, self.n_componets)

    # Initial computation of the mean-vector and
covariance matrix
    self.mean_vector = [np.mean(x, axis=0) for x in new_X]
    self.covariance_matrixes = [np.cov(x.T) for x in new_X]

    # Deleting the new_X matrix because we will not need
it anymore
    del new_X
```

Now we can get to EM-algorithm.

13.2 EM-algorithm.

As the name says the EM-algorithm is divided in 2 steps — E and M.

E-step.

During the Estimation step, we calculate the r matrix. It is calculated using the formula below.

$$r_{nk} = \frac{\pi_k N(x_n|\mu_k,\sum_k)}{\sum_j \pi_j N(x_n|\mu_j,\sum_j)}$$

r matrix is also known as **'responsibilities'** and can be interpreted in the following way. Rows are the samples from the data set, while columns represent every cluster, the elements of this matrix are interpreted as follows r_{nk} is the probability of sample n to be part of cluster k. When the algorithm will converge we will use this matrix to predict the points cluster.

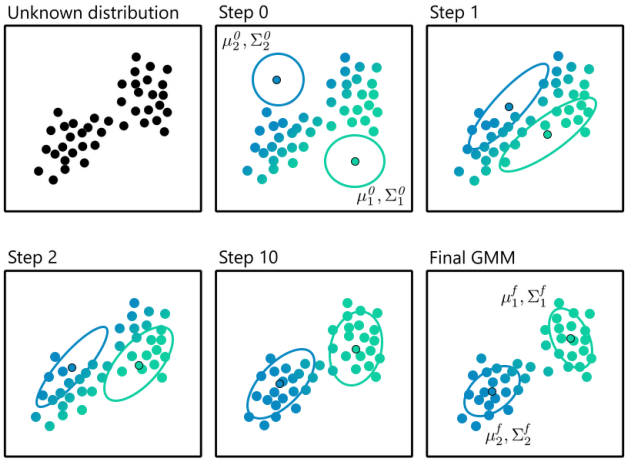


Figure 37: GMM algorithm.

Also, we calculate the N list, in which each element is basically the sum of the correspondent column in the r matrix. The following code is doing that.

```
 for iteration in range(self.max_iter):
     ''' --------------------   E - STEP
------------------- '''
     # Initiating the r matrix, evrey row contains the
probabilities
     # for every cluster for this row
     self.r = np.zeros((len(X), self.n_componets))
     # Calculating the r matrix
     for n in range(len(X)):
         for k in range(self.n_componets):
             self.r[n][k] = self.pi[k] * self.
multivariate_normal(X[n],
             self.mean_vector[k], self.
covariance_matrixes[k])
             self.r[n][k] /= sum([self.pi[j]*self.
multivariate_normal(X[n],
```

```
                self.mean_vector[j], self.
covariance_matrixes[j])
                for j in range(self.n_componets)])
        # Calculating the N
        N = np.sum(self.r, axis=0)
```

Point that the multivariate_normal is just the formula for normal distribution applied to vectors, it is used to calculate the probability for vectors in a normal distribution.

$$f(x;\mu,\sum) = \frac{1}{\sqrt{(2\pi)^d \times det(\sum)}} \times e^{-\frac{1}{2}\times((x-\mu)^T \times inv(\sum)\times(x-\mu))}$$

where d = dimension of the vector x, x = the input vector, μ = the mean vector, and $\sum$ is the covariance matrix.

and the code bellow implement it, taking the row vector, mean vector and the covariance matrix.

```
def multivariate_normal(self, X, mean_vector,
  covariance_matrix):
    '''
        This function implements the multivariat normal
  derivation formula,
        the normal distribution for vectors it requires
  the following parameters
        :param X: 1-d numpy array
            The row-vector for which we want to calculate
  the distribution
        :param mean_vector: 1-d numpy array
            The row-vector that contains the means for
  each column
        :param covariance_matrix: 2-d numpy array (matrix)
            The 2-d matrix that contain the covariances
  for the features
    '''
   return (2*np.pi)**(-len(X)/2)*np.linalg.det(
  covariance_matrix)**(-1/2)
   * np.exp(-np.dot(np.dot((X-mean_vector).T, np.linalg.
  inv(covariance_matrix)),
   (X-mean_vector))/2)
```

M-step:

During the Maximization-step we will step-by-step set the value fo the

mean vectors and covariance matrices to describe with them the clusters. To do that we will use the following formulas.

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} x_n$$
$$\sum_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} (x_n - \mu_k)(x_n - \mu_k)^T$$
$$\pi_k = \frac{N_k}{N}$$

In GMM the mean vectors are interpreted as centroids of clusters. An centroid is a point in space that is representing the cluster. In GMM a cluster is also represented by covariance matrix of the cluster. This allows us to have a point that is part of every cluster just in different proportion.

In the code it will look like that:

```
''' --------------------------   M - STEP
   -------------------------- '''
    # Initializing the mean vector as a zero vector
    self.mean_vector = np.zeros((self.n_componets, len(X
   [0])))

    # Updating the mean vector
    for k in range(self.n_componets):
        for n in range(len(X)):
            self.mean_vector[k] += self.r[n][k] * X[n]
    self.mean_vector = [1/N[k]*self.mean_vector[k]
    for k in range(self.n_componets)]

    # Initiating the list of the covariance matrixes
    self.covariance_matrixes = [np.zeros((len(X[0]), len(X
   [0])))
    for k in range(self.n_componets)]

    # Updating the covariance matrices
    for k in range(self.n_componets):
        self.covariance_matrixes[k] = np.cov(X.T, aweights
   =(self.r[:, k]), ddof=0)
    self.covariance_matrixes = [1/N[k]*self.
   covariance_matrixes[k]
    for k in range(self.n_componets)]
```

```
    # Updating the pi list
    self.pi = [N[k]/len(X) for k in range(self.n_componets)
  ]
```

And we are done with fit function. Iteratively applying EM-algorithm will make the GMM finally to converge.

The predict function:

The predict function is actually very simple we simply use the multivariate normal function using the optimal mean vectors and covariance matrices for each cluster, to find using which gives the biggest values.

```
def predict(self, X):
    '''
        The predicting function
        :param X: 2-d array numpy array
            The data on which we must predict the clusters
    '''
    probas = []
    for n in range(len(X)):
        probas.append([self.multivariate_normal(X[n], self.
  mean_vector[k], self.covariance_matrixes[k])
                           for k in range(self.n_componets)
  ])
    cluster = []
    for proba in probas:
        cluster.append(self.comp_names[proba.index(max(
  proba))])
    return cluster
```

The full code of the GMM implementation can be found here:

```
# Importing the libraries
import numpy as np
import pandas as pd
class GMM:
    '''
        This class is the implementation of the Gaussian
  Mixture Models
        inspired by scikit-learn implementation.
    '''
    def __init__(self, n_components, max_iter = 100,
  comp_names=None):
        '''
```

```
        This functions initializes the model by seting
 the following
        paramenters:
            :param n_components: int
                The number of clusters in which the
algorithm must split
                the data set
            :param max_iter: int, default = 100
                The number of iteration that the
algorithm will go
                throw to find the clusters
            :param comp_names: list of strings,
default=None
                In case it is setted as a list of
string it will use
                to name the clusters
    '''
    self.n_componets = n_components
    self.max_iter = max_iter
    if comp_names == None:
        self.comp_names = [f"comp{index}" for index in
        range(self.n_componets)]
    else:
        self.comp_names = comp_names
    # pi list contains the fraction of the dataset for
 every cluster
    self.pi = [1/self.n_componets for comp in range(
self.n_componets)]

 def multivariate_normal(self, X, mean_vector,
covariance_matrix):
    '''
        This function implements the multivariate
normal derivation
        formula, the normal distribution for vectors
it requires the
        following parameters
            :param X: 1-d numpy array
                The row-vector for which we want to
calculate the
```

```
                distribution
            :param mean_vector: 1-d numpy array
                The row-vector that contains the means
 for each
                column
            :param covariance_matrix: 2-d numpy array
(matrix)
                The 2-d matrix that contain the
covariances for the
                features
        '''
        return (2*np.pi)**(-len(X)/2)*np.linalg.det(
covariance_matrix)**(-1/2)
        * np.exp(-np.dot(np.dot((X-mean_vector).T,
        np.linalg.inv(covariance_matrix)),
        (X-mean_vector))/2)

    def fit(self, X):
        '''
            The function for training the model
                :param X: 2-d numpy array
                    The data must be passed to the
algorithm as 2-d array,
                    where columns are the features and the
 rows are the samples
        '''
        # Spliting the data in n_componets sub-sets
        new_X = np.array_split(X, self.n_componets)
        # Initial computation of the mean-vector and
covarience matrix
        self.mean_vector = [np.mean(x, axis=0) for x in
new_X]
        self.covariance_matrixes = [np.cov(x.T) for x in
new_X]
        # Deleting the new_X matrix because we will not
need it anymore
        del new_X
        for iteration in range(self.max_iter):
            ''' -------------------------   E - STEP
------------------------- '''
```

```
        # Initiating the r matrix, evrey row contains
the probabilities
        # for every cluster for this row
        self.r = np.zeros((len(X), self.n_componets))
        # Calculating the r matrix
        for n in range(len(X)):
            for k in range(self.n_componets):
                self.r[n][k] = self.pi[k] * self.
multivariate_normal(X[n],
                self.mean_vector[k], self.
covariance_matrixes[k])
                self.r[n][k] /= sum([self.pi[j]*
                self.multivariate_normal(X[n],
                self.mean_vector[j], self.
covariance_matrixes[j])
                for j in range(self.n_componets)])
        # Calculating the N
        N = np.sum(self.r, axis=0)
        ''' -------------------------   M - STEP
------------------------- '''
        # Initializing the mean vector as a zero
vector
        self.mean_vector = np.zeros((self.n_componets,
len(X[0])))
        # Updating the mean vector
        for k in range(self.n_componets):
            for n in range(len(X)):
                self.mean_vector[k] += self.r[n][k] * X
[n]
        self.mean_vector = [1/N[k]*self.mean_vector[k]
        for k in range(self.n_componets)]
        # Initiating the list of the covariance
matrixes
        self.covariance_matrixes = [np.zeros((len(X[0])
, len(X[0])))
        for k in range(self.n_componets)]
        # Updating the covariance matrices
        for k in range(self.n_componets):
            self.covariance_matrixes[k] = np.cov(X.T,
                                aweights=(self.r[:, k])
```

```
, ddof=0)
            self.covariance_matrixes = [1/N[k]*self.
covariance_matrixes[k]
            for k in range(self.n_componets)]
            # Updating the pi list
            self.pi = [N[k]/len(X) for k in range(self.
n_componets)]
 def predict(self, X):
     '''
         The predicting function
             :param X: 2-d array numpy array
                 The data on which we must predict the
clusters
     '''
     probas = []
     for n in range(len(X)):
         probas.append([self.multivariate_normal(X[n],
self.mean_vector[k], self.covariance_matrixes[k])
                        for k in range(self.n_componets)
])
     cluster = []
     for proba in probas:
         cluster.append(self.comp_names[proba.index(max(
proba))])
     return cluster
```

13.3 The scikit-learn implementation

GMM in sklearn library is implemented as **GaussianMixture** model. Firstly let's generate a toy data set, to see how to apply this module.

```
In:
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

X, y_true = make_blobs(
    n_samples=500, centers=5,
```

```
    cluster_std=0.9, random_state=17
)

kmeans= KMeans(5, random_state=420)
labels = kmeans.fit(X).predict(X)
plt.figure(dpi=175)
plt.scatter(X[ : , 0], X[ : , 1], c=labels, s=7, cmap='
   viridis')

Out:
Figure 26
```

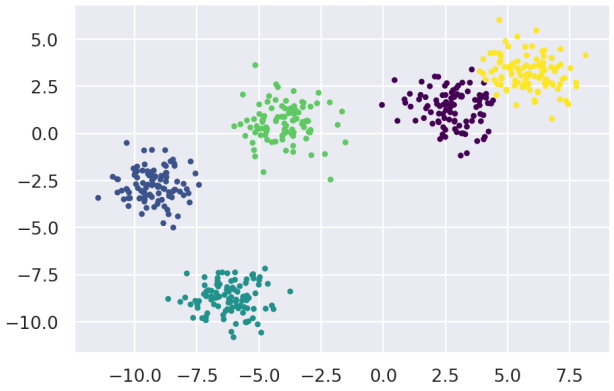


Figure 38: The generated data set.

We generated a random set with 500 sample and 5 centroids a.k.a. 5 groups of data samples.

Now we can apply the GMM algorithm on it using sklearn.

```
In:
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=5).fit(X)
labels = gmm.predict(X)
plt.figure(dpi=175)
plt.scatter(X[ : , 0], X[ : , 1], c=labels, s=7, cmap='
   viridis')
Out:
Figure 27
```

But because GMM contains a probabilistic model under the hood, it is also possible to find the probability that any point in the given dataset belongs to any given cluster.

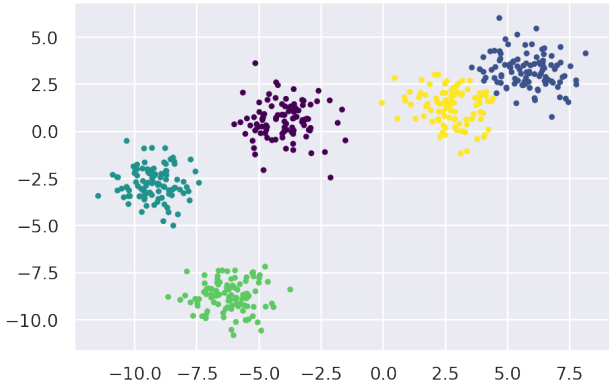


Figure 39: The data set divided in 5 clusters.

```
probabilities = gmm.predict_proba(X)
```

The **probabilities** variable is a numpy array, it has a shape of number of samples x number of clusters. Each row corresponds to a single data point and the jth column corresponds to the probability that the sample belongs to the jth cluster.

Because of the fact that GMM is using EM under the hood, each cluster is associated with a smooth Gaussian Mode. The following function will help us visualize the locations and shapes of the GMM clusters.

```
In:
from matplotlib.patches import Ellipse

def plot_gmm(gmm, X, label=True, ax=None):
    def draw_ellipse(position, covariance, ax=None, **
  kwargs):
        ax = ax or plt.gca()

        if covariance.shape == (2, 2):
            U, s, Vt = np.linalg.svd(covariance)
            angle = np.degrees(np.arctan2(U[1, 0], U[0, 0])
  )
            width, height = 2 * np.sqrt(s)
        else:
            angle = 0
            width, height = 2 * np.sqrt(covariance)

        for nsig in range(1, 4):
            ax.add_patch(Ellipse(
```

```
                position, nsig * width, nsig * height,
                angle, **kwargs
            ))

    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=7, cmap='
viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=7, zorder=2)

    ax.axis('equal')
    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covariances_,
gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

Now let's use the function to see how the clusters are shaped now on our data.

```
In:
gmm = GaussianMixture(n_components=5, random_state=420)
plt.figure(dpi=175)
plot_gmm(gmm, X)
Out:
Figure 28
```

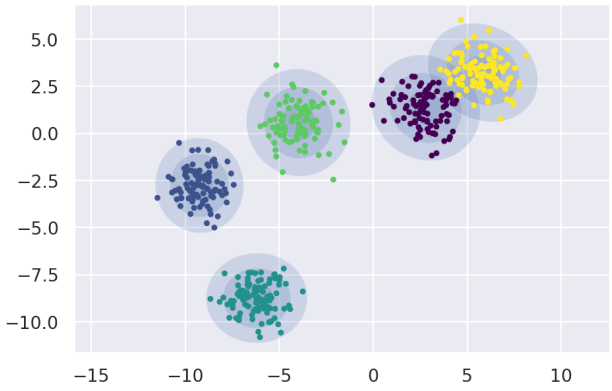


Figure 40: The shape of the clusters.

In order, to make this advantage of GMM a little more obvious we will create a new sample dataset that has a different shape.

```
In:
X, y_true = make_blobs(n_samples=400, centers=4,
   cluster_std=0.60, random_state=0)
X = X[:, ::-1] # flip axes for better plotting

rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2))
```

Now let's apply GMM algorithm and see what the resulting clusters look like.

```
In:
gmm = GaussianMixture(n_components=4,covariance_type='full'
   , random_state=42)
plt.figure(dpi=175)
plot_gmm(gmm, X_stretched)
Out:
Figure 29
```

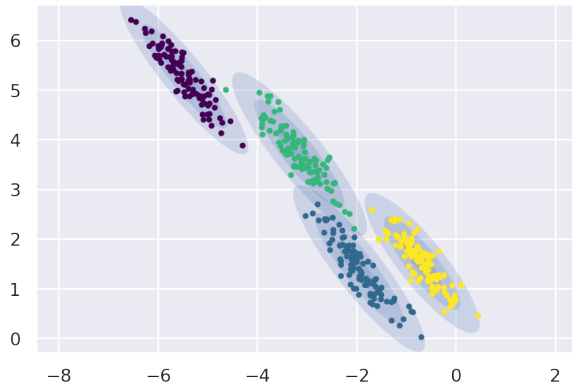


Figure 41: The shape of the stretched clusters.

Also, note that for the previous fit the hyperparameter covariance_type was set differently. This hyperparameter controls the degrees of freedom in the shape of each cluster.

- The default is **covariance_type="diag"**, which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes.
- A slightly simpler and faster model is **covariance_type="spherical"**, which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of KMeans, though it is not entirely equivalent.

- A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use **covariance_type="full"**, which allows each cluster to be modeled as an ellipse with arbitrary orientation.

13.4 Conclusion

In this topic we explored the application of probability in another type of machine learning task - clusterization. Even if GMM isn't a pure clusterization algorithm, it is a density estimation on, it is a good introduction into the clusterization algorithms. In the next topic we will explore K-Means and how to measure the performance of a clusterization algorithm.

14 KMeans

The KMeans algorithm searches for k number of clusters (the number k must be known in advance) within an unlabeled multidimensional data set. For the KMeans algorithm the optimal clustering has the following properties:

- The "cluster center" is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

KMeans is implemented in **sklearn.cluster.KMeans**, so let's generate a two dimensional sample data set and observe the k-means results.

```
In:
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=420, centers=3, cluster_std
   =0.40, random_state=0)
plt.scatter(X[ : , 0], X[ : , 1], s=15)

Out:
Figure 32
```

Figure 42: The generated data set.

Now, let's apply KMeans on this sample dataset. We will visualize the results by coloring each cluster using a different color. We will also plot the cluster centers.

```
In:
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
ids = kmeans.predict(X)

plt.scatter(X[ : , 0], X[ : , 1], c=ids, s=15, cmap='
   viridis')
centers = kmeans.cluster_centers_
plt.scatter(centers[ : , 0], centers[ : , 1], c='red', s
   =100, alpha=0.5)

Out:
Figure 33
```

Figure 43: The found clusters and their centroids.

As you can see the algorithm assigns the points to the clusters very similarly to how we might assign them by eye.

How does the algorithm work? KMeans uses an iterative approach known as expectation–maximization. In the context of the KMeans algorithm, the expectation–maximization consists of the following steps:

1. Randomly guess some cluster centers.
2. Repeat until converged.
   (a) E-Step: assign points to the nearest cluster center.
   (b) M-Step: set the cluster centers to the mean.

The KMeans algorithm is simple enough for us to write a really basic implementation of it in just a few lines of code.

```
In:
# This function has the following
# Signature: pairwise_distances_argmin(X, Y, axis=1,
   metric='euclidean',
# metric_kwargs=None)
# This function computes for each row in X, the index of
   the row of Y
# which
# is closest (according to the specified distance).
from sklearn.metrics import pairwise_distances_argmin
import numpy as np

def find_clusters(X, n_clusters, rseed=69):
    # Randomly guess some cluster centers
    rng = np.random.RandomState(rseed)
    i = rng.permutation(X.shape[0])[ : n_clusters]
    centers = X[i]

    while True:
        # E-Step: Assign labels based on closest center
        labels = pairwise_distances_argmin(X, centers)

        # M-Step: Find new centers from means of points
        new_centers = np.array([
            X[labels == i].mean(0) for i in range(
  n_clusters)
        ])

        # Check for convergence
        if np.all(centers == new_centers):
            break

        centers = new_centers

    return centers, labels
```

Well tested implementations, such as the one from scikit-learn, do a few more things under the hood, but this trivial implementation allows

us to view one caveat of expectation-maximization.

Let's test this implementation on our sample data set.

```
In:
centers, labels = find_clusters(X, 3)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=15, cmap='viridis
   ')
plt.scatter(centers[ : , 0], centers[ : , 1], c='red', s
   =100, alpha=0.5)
Out:
Figure 34
```

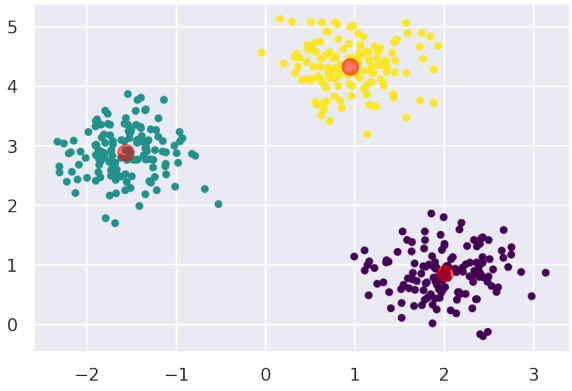


Figure 44: The found clusters and their centroids.

Everything seems to be ok, right? Now, let's try and change the random seed.

```
In:
centers, labels = find_clusters(X, 3)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=15, cmap='viridis
   ')
plt.scatter(centers[ : , 0], centers[ : , 1], c='red', s
   =100, alpha=0.5)
Out:
Figure 35
```

What happened??? Well, there are a few issues to be aware of when using the expectation-maximization approach. One of them is that the globally optimal result may not be achieved. Although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that this will lead to the globally optimal result. This is why, better implementations, such as the one from Scikit-Learn, by default

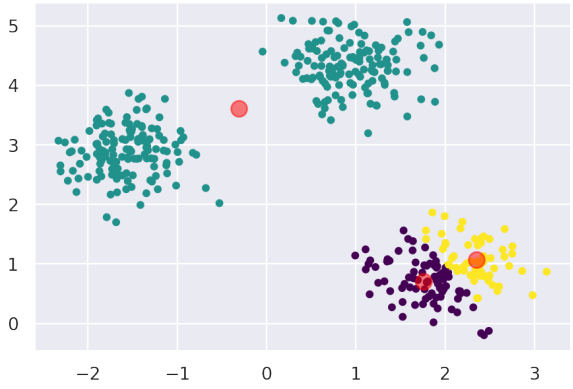


Figure 45: The found clusters and their non-optimal centroids.

run the algorithm for multiple starting guesses. (In Scikit-Learn this is controlled by the **n_init** parameter, which defaults to 10)

Another caveat of the KMeans algorithm is that it is limited to linear cluster boundaries. The fundamental assumption that points will be closer to their own cluster center than to others means that the algorithm will often be ineffective when dealing with clusters that have complicated geometries. Consider the following example, along with the results found by the typical KMeans approach.

```
In:
from sklearn.datasets import make_moons

X, _ = make_moons(500, noise=.06, random_state=69)
labels = KMeans(2, random_state=69).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=15, cmap='viridis
   ')
Out:
Figure 36
```

Fortunately, this issue is easily solved using one version of kernelized KMeans that is implemented in Scikit-Learn within the SpectralClustering estimator.

```
In:
from sklearn.cluster import SpectralClustering

model = SpectralClustering(n_clusters=2, affinity='
   nearest_neighbors',
assign_labels='kmeans')
labels = model.fit_predict(X)
```

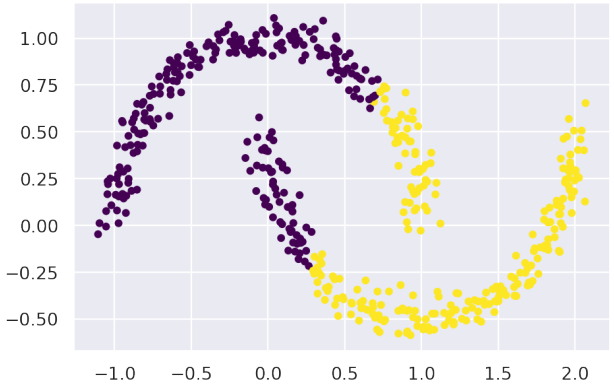


Figure 46: KMeans algorithm applied on Moons data set.

```
plt.scatter(X[:, 0], X[:, 1], c=labels, s=15, cmap='viridis
   ')
Out:
Figure 37
```

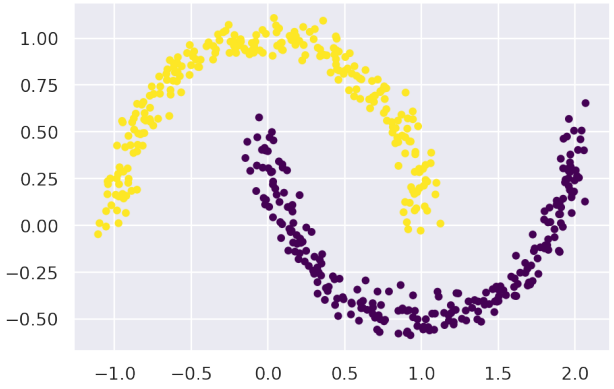


Figure 47: SpectralClustering algorithm applied on Moons data set.

Other problems that you might have to deal with when using the KMeans algorithm are

- KMeans might be slow for large number of samples. Because each iteration of the KMeans algorithm must access every point in the dataset, the algorithm might become relatively slow as the number of samples grows. Some sort of fix for this issue might be using just a subset of the data to update the cluster centers at each step.

- The number of clusters must be selected beforehand. This is a common challenge because KMeans cannot learn the number of clusters from the data. For example, we can ask the algorithm

to identify 4 clusters from our previous sample dataset and it will proceed without a second thought.

```
In:
labels = KMeans(4, random_state=420).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=15, cmap='viridis
   ')
Out:
Figure 38
```

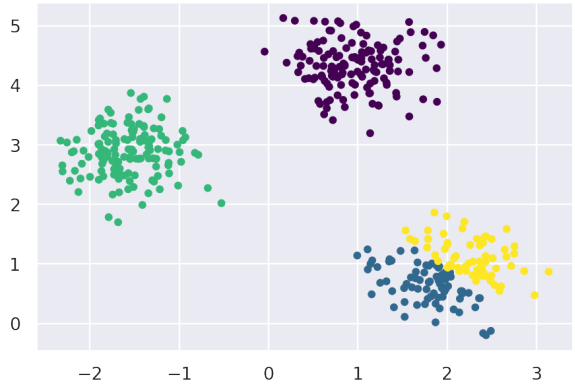


Figure 48: An example where the set number of clusters isn't the real one

Whether this is a meaningful result or not is a question that might depend on the context. Some well known approaches that might help when picking the right number of clusters are the elbow method (a heuristic approach that we won't discuss in this topic) and the **Silhouette Score** about which we will continue the discussion.

14.1 Silhouette Score

Silhouette analysis can be used to study the separation distance between the resulting clusters. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters and thus provides a way to assess parameters like number of clusters visually. This measure has a range of [-1, 1].

Silhouette coefficients (as these values are referred to as) near +1 indicate that the sample is far away from the neighboring clusters. A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters and negative values indicate that those samples might have been assigned to the wrong cluster.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (a) and the mean nearest-cluster distance (b) for each sample. The Silhouette Coefficient for a sample is (b - a) / max(a, b). To clarify, b is the distance between a sample and the nearest cluster that the sample is not a part of. Note that Silhouette Coefficient is only defined if number of labels is 2 <= n_labels <=n_samples - 1.

In scikit-learn there is **sklearn.metrics.silhouette_score** that is used to compute the mean **Silhouette Coefficient** of all samples and there also is **sklearn.metrics.silhouette_samples** that computes the **Silhouette Coefficient** for each sample.

Now let's put the newly acquired knowledge in practice and write a function that shows the Silhouette Plot for a certain number of clusters.

```
In:
from sklearn.metrics import silhouette_score,
   silhouette_samples
import matplotlib.cm as cm

def draw_silhouette_plot(X, n_clusters):
    clusterer = KMeans(n_clusters=n_clusters, random_state
   =69)
    cluster_labels = clusterer.fit_predict(X)

    silhouette_avg = silhouette_score(X, cluster_labels)
    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X,
   cluster_labels)

    # The Silhouette Score ranges from -1 to 1
    plt.xlim([-1, 1])
    # The (n_clusters + 1) * 10 is for inserting blank
   space between silhouette
    # plots of individual clusters, to demarcate them
   clearly.
    plt.ylim([0, len(X) + (n_clusters + 1) * 10])
    plt.yticks([]) # Clear the yaxis labels
    plt.xticks([-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4,
   0.6, 0.8, 1])

    y_lower = 10
    for i in range(n_clusters):
```

```
        ith_cluster_silhouette_values =
    sample_silhouette_values[cluster_labels == i]
        ith_cluster_silhouette_values.sort()
        size_cluster_i = ith_cluster_silhouette_values.
    shape[0]
        y_upper = y_lower + size_cluster_i

        color = cm.nipy_spectral(float(i) / n_clusters)
        plt.fill_betweenx(np.arange(y_lower, y_upper), 0,
    ith_cluster_silhouette_values, facecolor=color, edgecolor
    =color, alpha=0.7)

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10  # 10 for the 0 samples

     # The vertical line for the average silhouette score
    of all the values
     plt.axvline(x=silhouette_avg, color="red", linestyle="
    --")
     plt.title(f'The Silhouette Plot for n_clusters = {
    n_clusters}')
     plt.legend([ 'Silhouette Score' ] + [f'Cluster {i}' for
     i in range(n_clusters)])

draw_silhouette_plot(X, int(input('Enter the number of
    clusters: ')))
```

Out:
Figure 39-42

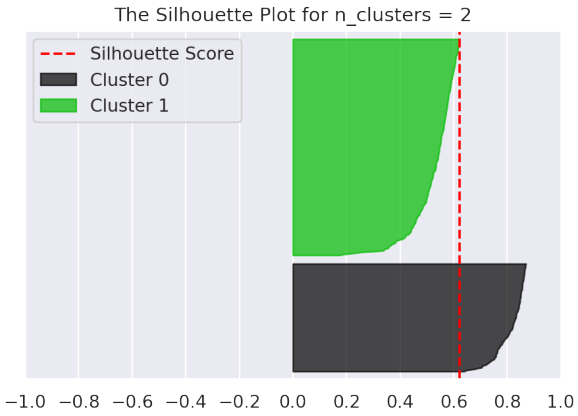


Figure 49: The Silhouette when number of clusters is equal with 2.

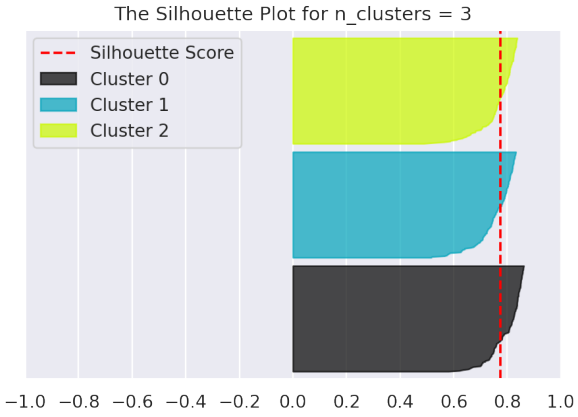


Figure 50: The Silhouette when number of clusters is equal with 3.

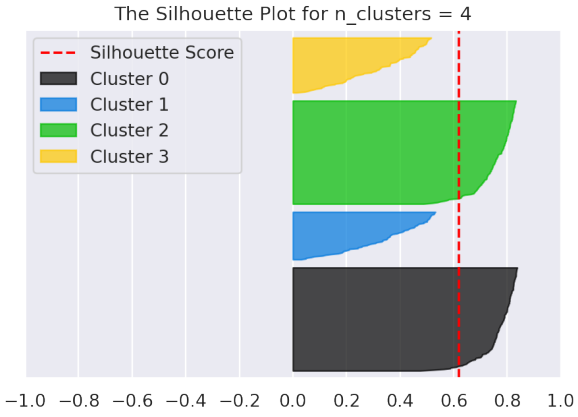


Figure 51: The Silhouette when number of clusters is equal with 4.

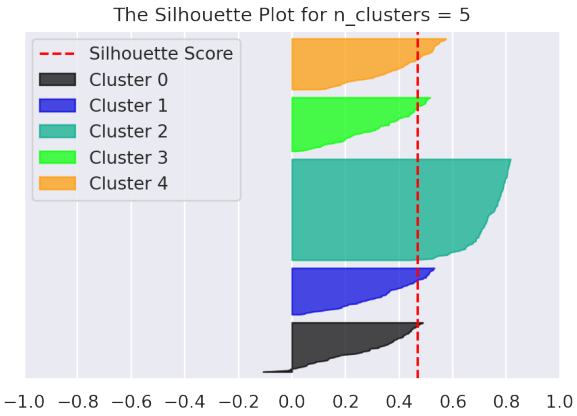


Figure 52: The Silhouette when number of clusters is equal with 5.

As you can see the best number of clusters judging by the average Silhouette Score is 3. Furthermore, these plots provide more valuable information than you might expect. They provide information about the size of each cluster relative to other clusters and they also provide information about the approximate Silhouette Coefficient for each sample in the cluster.

14.2 Marketing Segmentation:

In the context of Marketing Segmentation, also known as Customer Segmentation, Cluster Analysis involves the use of a mathematical model to discover groups of similar customers based on finding the smallest variations among customers within each group. These homogeneous groups are known as customer archetypes or personas.

For the following example, we will use some data from the kaggle website.

The CSV file can be downloaded from here or by using the kaggle CLI with the following command

```
kaggle datasets download --unzip -d vjchoudhary7/customer-
   segmentation-tutorial-in-python
```

First of all, let's import the data using pandas.read_csv and do a little preprocessing.

```
In:
import pandas as pd
from sklearn import preprocessing

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

```
df.drop('CustomerID', axis=1, inplace=True)

df.loc[df['Gender'] == 'Male', 'Gender'] = 0
df.loc[df['Gender'] == 'Female', 'Gender'] = 1

column_names = df.columns
# Standardize features by removing the mean and scaling to
    unit variance
scaler = preprocessing.StandardScaler()
X = scaler.fit_transform(df)
```

Note that we standardize the data before applying the clustering algorithm. This is a good practice and although sometimes this step is not required, it rarely hurts to do it.

Now, let's use the function that we defined previously in order to perform Silhouette Analysis and find the best number of clusters.

```
In:
from pylab import figure
for i, k in enumerate([2, 3, 4]):
    figure(i)
    draw_silhouette_plot(X, k)

Out:
Figure 43-45
```

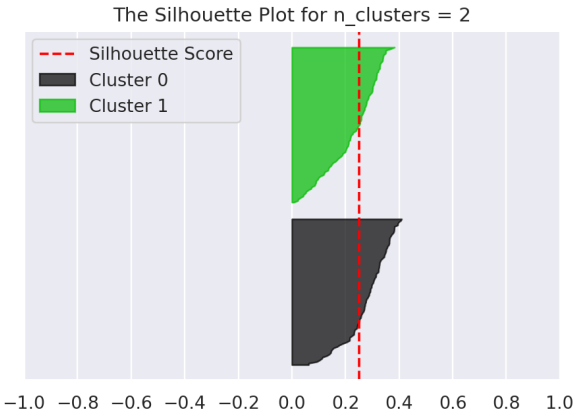


Figure 53: The Silhouette when number of clusters is equal with 2.

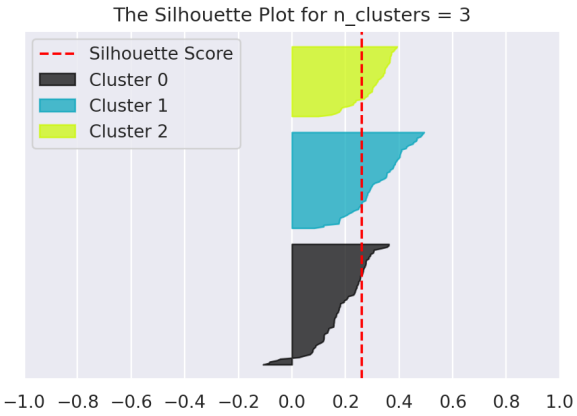


Figure 54: The Silhouette when number of clusters is equal with 3.

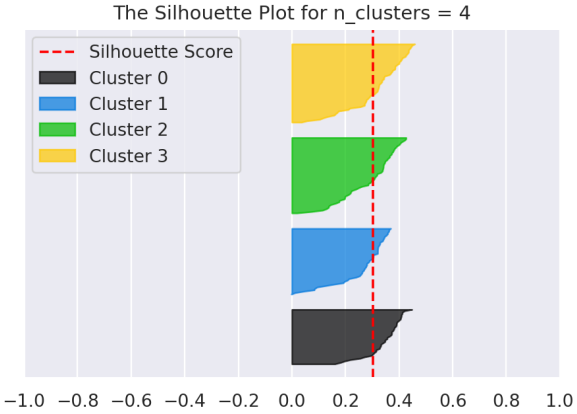


Figure 55: The Silhouette when number of clusters is equal with 4.

Based on these plots, the best number of clusters seems to be 4 because:

- it has the biggest Silhouette Score.
- the clusters are relatively of equal sizes
- it does not have samples with negative Silhouette Coefficients

Now, let's go a step further and see what the cluster centers look like for each of the clusters and see if we can divide the customers from this dataset into distinct customer archetypes.

In:

```
kmeans = KMeans(n_clusters=4, random_state=69).fit(X)

# Reverse the Standard Scaling we did earlier
```

```
centers = scaler.inverse_transform(kmeans.cluster_centers_)

# View the cluster centers as a pandas table
pd.DataFrame(
    data=centers,
    index=[ f'K{i}' for i in range(1, centers.shape[0] + 1)
    ],
    columns=column_names
)
```

Out:

	Gender	Age	Annual Income (k$)	Spending Score (1-100)
K1	0.0	28.250000	62.000000	71.675000
K2	0.0	49.437500	62.416667	29.208333
K3	1.0	48.109091	58.818182	34.781818
K4	1.0	28.438596	59.666667	67.684211

So, based on the cluster centers we can define 4 customer archetypes

- Younger Males with High Spending Score
- Older Males with Low Spending Score
- Younger Females with High Spending Score
- Older Females with Low Spending Score

14.3 Conclusion

In this topic we explored the KMeans clustering algorithm. Also we met the Silhouette Score a metrics which helps us to measure the performance of a clustering algorithm and select the best number of clusters. Also we found a real use-case where clustering algorithms can be applied in real industry.

15 Data Visualisation

In previous topics in this book you learned that date can take form of a vector, matrix and even a data frame. However this forms aren't always informative. To give your data a more informative look you can appeal to the data visualisation. Data visualisation may help you to extract insights from data and explore it easily than using pure statistics. In this topic we are going to explore matplotlib - a library for plotting graphs and some plots that are used most often.

15.1 What is matplotlib?

Matplotlib is a multiplatform data visualization library built on **NumPy** arrays, and designed to work with the broader **SciPy** stack. One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish.

Just as we use the np shorthand for NumPy and the pd shorthand for Pandas, these are some standard shorthands for Matplotlib imports:

```
In:
import matplotlib as mpl
import matplotlib.pyplot as plt
```

A potentially confusing feature of Matplotlib is its dual interfaces:

- the MATLAB-style state-based interface.
- the object oriented interface.

15.1.1 MATLAB-style interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface.

```
In:
plt.figure(dpi=150) # create a plot figure

x = np.linspace(0, 10)
```

```
# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x))

Out:
Figure 46
```

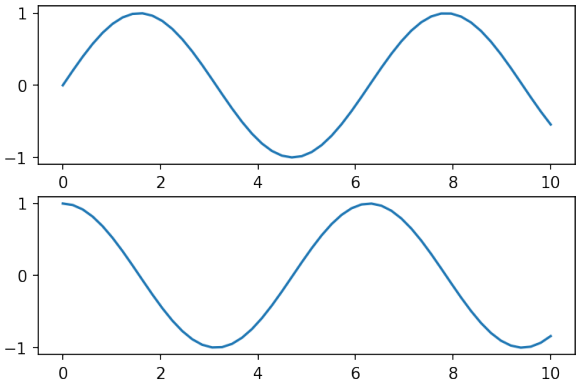


Figure 56: Line plot created with MATLAB-style interface.

15.1.2 Object oriented interface

The object-oriented interface is available for more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are methods of explicit Figure and Axes objects.

```
In:
fig, ax = plt.subplots(2, dpi=150)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))

Out:
Figure 47
```

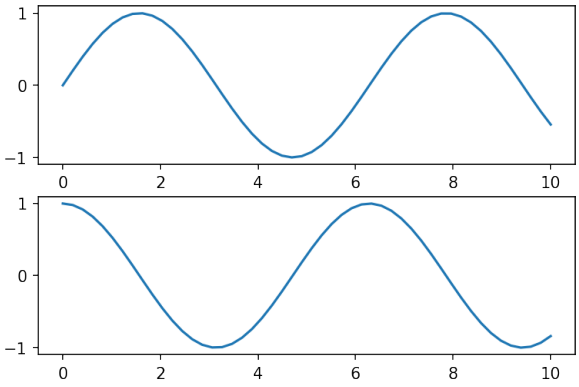


Figure 57: Line plot created with Object oriented interface interface.

For more simple plots, the choice of which style to use is largely a matter of preference, but the object-oriented approach can become a necessity as plots become more complicated.

15.2 How to make simple line plots?

Perhaps the simplest of all plots is the visualization of a single function y = f(x).

In the following example we demonstrate how a figure and axes can be created using **plt.figure** and **plt.axes** respectively. Once we have created an axes, we can use the ax.plot (or the **plt.plot**) function to plot some data. Since we want to create a single figure with multiple lines, we can simply call the **plt.plot** function multiple times. **plt.plot** function takes additional arguments (that are optional) that can be used to specify the **color**, **label**, **linestyle** and others. If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines. Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust axis limits is to use the **plt.xlim()** and **plt.ylim()** methods. We can also set the title and axis label using **plt.title**, **plt.xlabel**, **plt.ylabel**. And finally, when multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. It is easiest to specify the label of each line using the label keyword of the plot function and then call **plt.legend**.

```
In:
# We can use the plt.style directive to choose appropriate
# aesthetic styles for our figures
plt.style.use('seaborn-whitegrid')
```

```
fig = plt.figure(dpi=150)
ax = plt.axes()

plt.plot(x, np.sin(x), color='red', label='sin(x)')
plt.plot(x, np.cos(x), color='green', linestyle='dotted',
   label='cos(x)')

plt.xlim([-2, 12])
plt.ylim([-1.25, 1.25])

plt.title('The sine and the cosine function')
plt.xlabel('x')
plt.ylabel('y')

plt.legend()

Out:
Figure 48
```

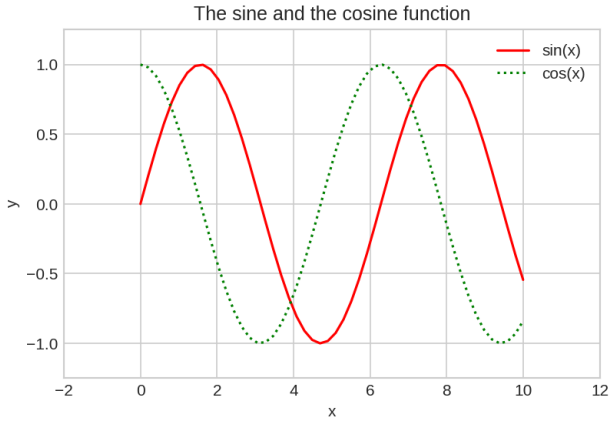


Figure 58: The sine and cosine function.

15.3 How to make simple scatter plots?

We can both use **plt.plot** and **plt.scatter** to create scatter plots.

The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

In the following example we create a scatter plot with points of many colors and sizes. In order to better see the results, we will also use the **alpha** argument to adjust the transparency level. The color argument is automatically mapped to a color scale shown by the **colorbar()** command, and the size argument is given in pixels. In this way, the color and size of points are used to convey information in the visualization, in order to illustrate multidimensional data.

```
In:
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.figure(dpi=150)
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='
   viridis')
plt.colorbar()

Out:
Figure 49
```

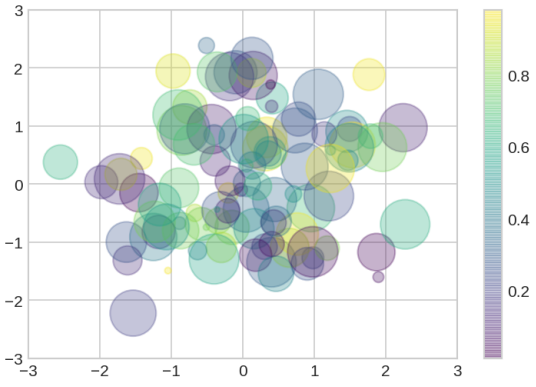


Figure 59: Scatterplot.

Aside from the different features available in **plt.plot** and **plt.scatter**, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as data sets get larger than a few thousand points, **plt.plot** can be noticeably more efficient than **plt.scatter**.

The reason is that **plt.scatter** has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In **plt.plot**, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large data sets, the difference between these two can lead to vastly different performance, and for this reason, **plt.plot** should be preferred over **plt.scatter** for large data sets.

15.4 Error bars

For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself.

In visualization of data and results, showing errors effectively can make a plot convey much more complete information.

The following example shows a basic **errorbar** plot. The **fmt** is a format code controlling the appearance of lines and points. It has the same syntax as the shorthand used in **plt.plot**.

```
In:
number_of_points = 70

x = np.linspace(0, 10, number_of_points)

dy=0.25
np.random.seed(17)
y = np.sin(x) + dy * np.random.randn(number_of_points)

plt.figure(dpi=150)
plt.errorbar(x, y, yerr=dy, fmt='.k')

Out:
Figure 50
```

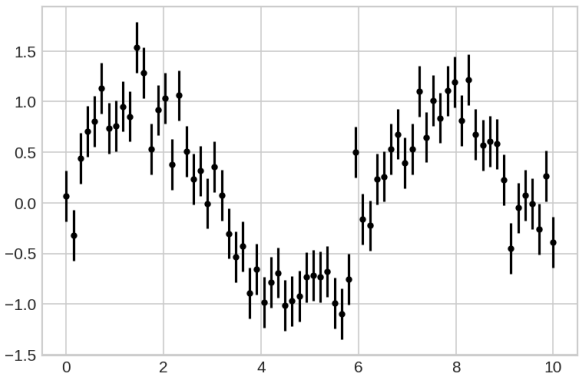


Figure 60: Error bars.

15.5 Contour plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task: **plt.contour** for contour plots, **plt.contourf** for filled contour plots, and **plt.imshow** for showing images.

In the following example we will use **plt.contour** to plot a function of two variables. **plt.contour** takes 3 arguments: a grid of x values, a grid of y values, and a grid of z values. The x and y values represent positions on the plot, and the z values will be represented by the contour levels. The most straightforward way to prepare such data is to use the **np.meshgrid** function, which builds two-dimensional grids from one-dimensional arrays. We will use the RdGy (short for Red-Gray) colormap.

```
In:
plt.style.use('seaborn-white')

def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)

x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X,Y)

plt.figure(dpi=150)
```

```
plt.contour(X, Y, Z, 20, cmap='RdGy')
plt.colorbar()

Out:
Figure 51
```

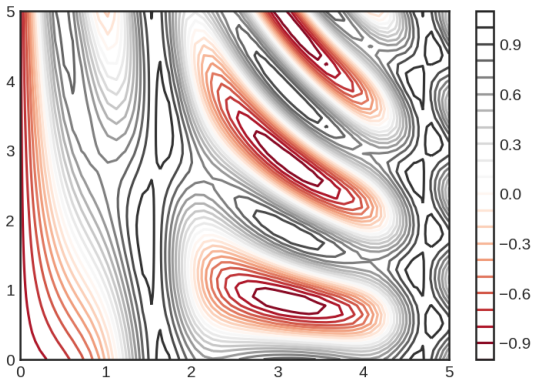


Figure 61: Contour plot.

15.6 Histograms

A simple histogram can be a great first step in understanding a dataset. We can use the **hist()** function to create a simple histogram in just one line. The **hist()** function has many options to tune both the calculation and the display.

In the following example we will plot multiple histograms.

```
In:
np.random.seed(0)
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(4, 1, 1000)

plt.figure(dpi=150)
plt.hist(x1, alpha=0.3, bins=40)
plt.hist(x2, alpha=0.3, bins=40)
plt.hist(x3, alpha=0.3, bins=40)

Out:
Figure 52
```

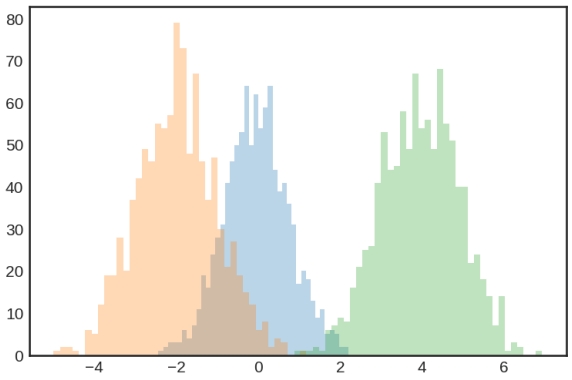


Figure 62: Histogram.

If the **bins** parameter is an integer, it defines the number of equal-width bins in the range.

Just as we create histograms in one dimension by dividing the number line into bins, we can also create histograms in two dimensions by dividing points among two-dimensional bins. This is illustrated in the following example.

```
In:
mean = [0, 0]
cov = [[1, 1], [1, 2]]

np.random.seed(0)
x, y = np.random.multivariate_normal(mean, cov, 10000).T

plt.figure(dpi=150)
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')

Out:
Figure 53
```

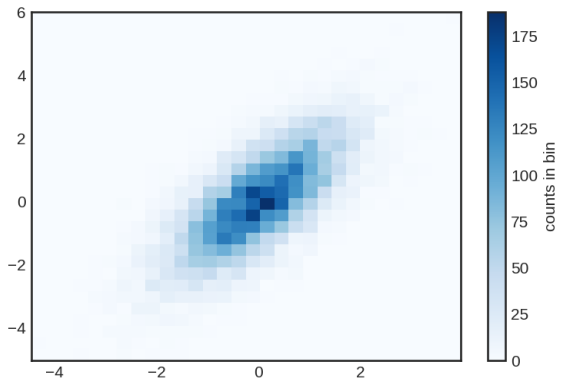


Figure 63: Bidimensional histogram.

15.7 Why you might also want to learn seaborn

Matplotlib has proven to be an incredibly useful and popular visualization tool, but even avid users will admit it often leaves much to be desired. An answer to this problem is **Seaborn**. **Seaborn** provides an API on top of **Matplotlib** that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.

Seaborn also provides straightforward ways to make violinplots, boxplots and others.

For the following examples we will be using the Heart Disease UCI dataset. This dataset can be downloaded using the following command

```
kaggle datasets download --unzip -d ronitf/heart-disease-
    uci
```

Note! Some of the following examples make use of the with statement. In python the with statement is a useful tool that is handy when you have two related operations which you would like to execute as a pair. In our examples, we use the with statement to change the sns style and then after the with block is executed the styles are reverted to what they were before (the with statement is sometimes referred to as a context manager).

15.7.1 KDE plots

A kernel density estimate (KDE) plot is a method for visualizing the distribution of observations in a dataset, analogous to a histogram. KDE

represents the data using a continuous probability density curve in one or more dimensions.

```
In:
import seaborn as sns
import pandas as pd

df = pd.read_csv('heart.csv')

with sns.axes_style('whitegrid'):
    plt.figure(dpi=150)
    sns.kdeplot(df.loc[ df['target'] == 0, 'age' ],
    shade=True, label='No Heart Disease')
    sns.kdeplot(df.loc[ df['target'] == 1, 'age' ],
    shade=True, label='Heart Disease')
    plt.legend()

Out:
Figure 54
```

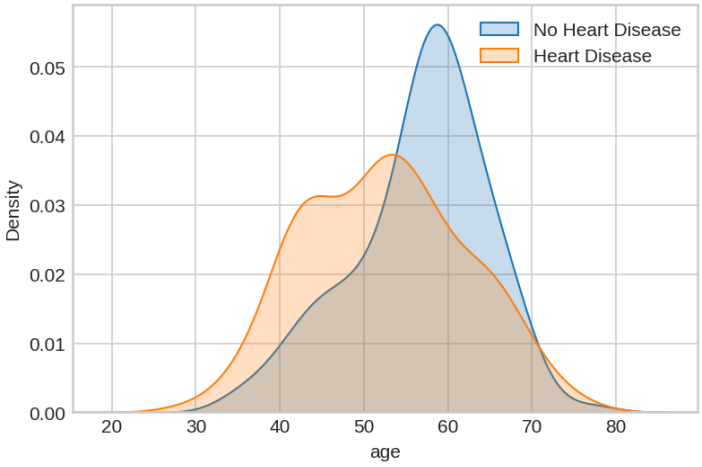


Figure 64: KDE plot.

We can also pass 2 columns to the kdeplot and we get a two-dimensional visualization of the data.

```
In:
with sns.axes_style('darkgrid'):
    plt.figure(dpi=150)
    sns.kdeplot(data=df, x='age', y='trestbps')
```

```
Out:
Figure 55
```

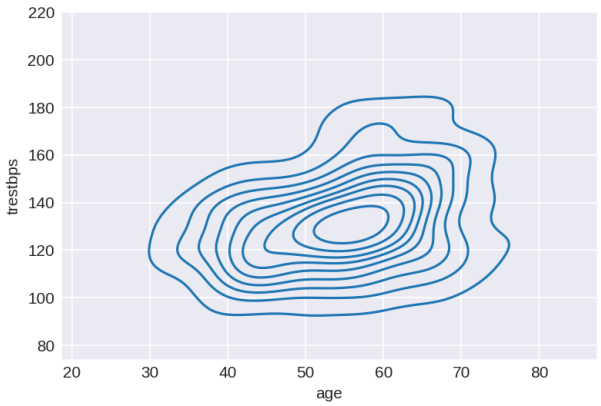


Figure 65: Bidimensional KDE plot.

15.7.2 jointplot

We can see the joint distribution and the marginal distributions together using **sns.jointplot**.

In the following example the first plot has the kind argument set to kde and the second plot has the kind argument set to **hex**.

```
In:
with sns.axes_style('darkgrid'):
    sns.jointplot(data=df, x='age', y='trestbps', kind='kde
   ', height=7)

with sns.axes_style('white'):
    sns.jointplot(data=df, x='age', y='trestbps', kind='hex
   ', height=7)

Out:
Figure 56-57
```

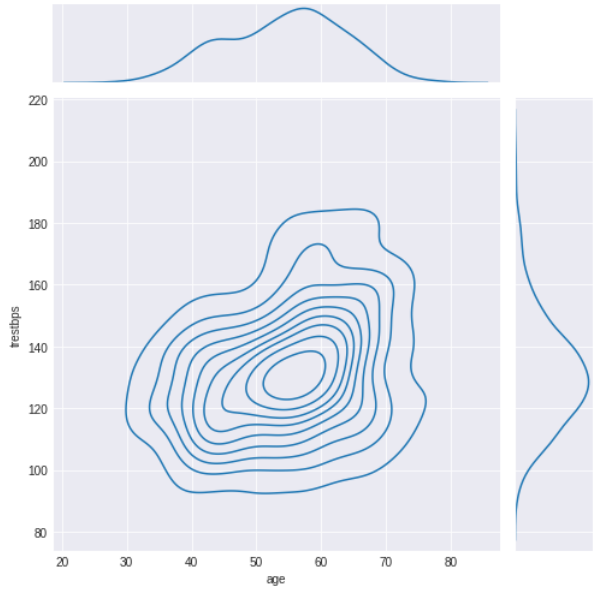


Figure 66: KDE jointplot.

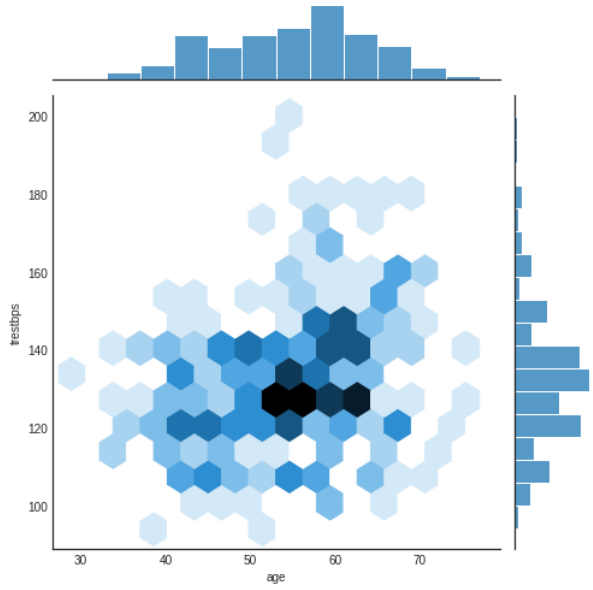


Figure 67: HEX jointplot.

15.7.3 boxplot

A box plot shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be "outliers" using a method that is a function of the inter-quartile range.

The following example draws a vertical box plot grouped by a categorical variable.

```
In:
plt.figure(dpi=150)
sns.boxplot(x='target', y='age', data=df)

Out:
Figure 58
```

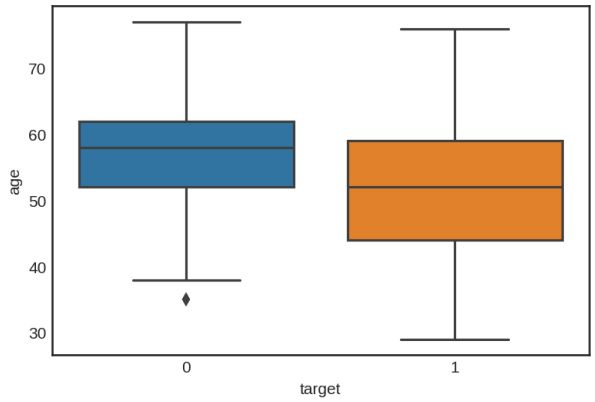


Figure 68: Boxplot.

15.7.4 Violinplot

Another nice way to compare distributions is by using a **violinplot**. It is a combination of boxplot and kernel density estimate. It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared. Unlike a box plot, in which all of the plot components correspond to actual data points, the violin plot features a kernel density estimation of the underlying distribution.

The same example as the above, the only difference is that we use **violinplot** instead of **boxplot**.

```
In:
plt.figure(dpi=150)
sns.violinplot(x='target', y='age', data=df)

Out:
Figure 59
```

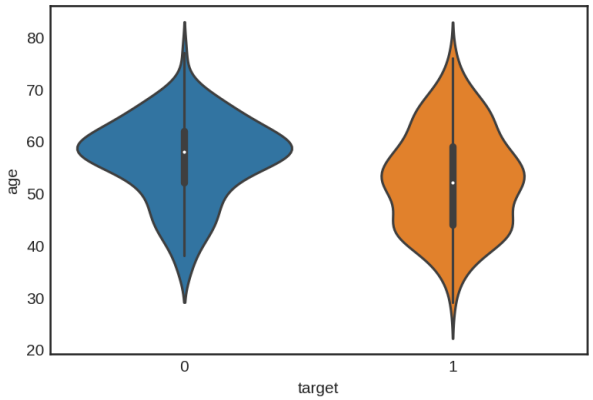


Figure 69: Violinplot.

15.7.5 lmplot

The last type of plot that we will show a example of is **lmplot**.

Although this plot might look a little bit scarry, lmplot just plots data and regression model fits across a FacetGrid.

```
In:
sns.lmplot(data=df, x='age', y='chol', hue='target', height
   =7, aspect=1)

Out:
Figure 60
```

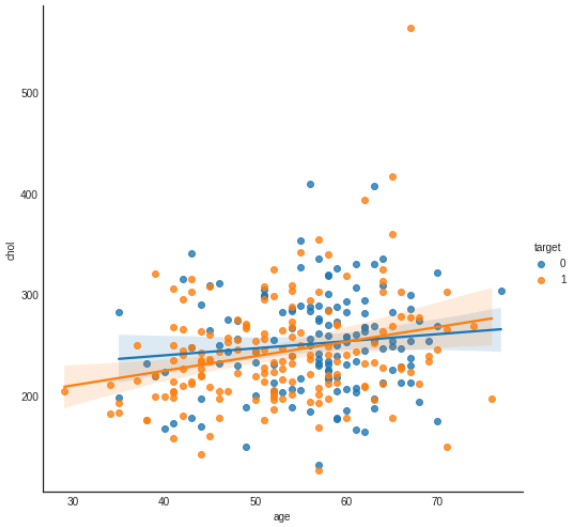


Figure 70: lmplot.

16 Outliers

Data is beautiful, and probably what makes it attractive is how different every set is and how many preprocessing steps you can apply to each of them. Usually, the data is not clean, but there is more than cleaning data from the wrong samples.

Let's take a simple example about height. Every day we meet a lot of people, and everyone has a different height, but for sure we usually see people with heights in the range of 150-190 cm (4-6.2 feet), and we do not really pay attention when we see someone with a height in this range. The interest comes in when we see someone very different, someone very short, or someone very tall. In this case, we treat this as an anomaly that is not similar to the normal, and guess what? This may happen with data. The data is not perfect because humans are not perfect, and there is always a chance to get some anomalies which in the ML field we call **Outliers**.

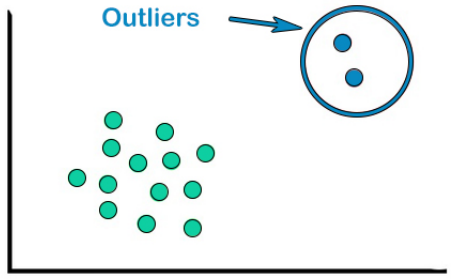


Figure 71: Outliers example.

The problem related to Outliers discussed in this chapter is the appearance of Outliers in training data. The problem is straightforward. Suppose you have to train a model to predict the best item to recommend to an average customer, given his age and other data. The data that you will give to your model is critical. It will be easy to learn if you have good data about what the average customer buys, but what if you have outliers? What if your data contains examples that are very weird and not usual at all? They will create an obstacle for you to train a good model because what interests you is the average, and you do not really care about rare examples of customers buying weird stuff. Because the model should focus only on good data, we will try to remove outliers,

but before that, we have to find them. This is why we need **Outlier Detection**.

16.1 Outlier Detection Methods.

Let's summarize, Outliers are just the values that look different from other values in the data.

16.1.1 Isolation Forest.

Isolation Forest is based on the idea of isolation of anomalies(outliers) that are both few and have attribute values that are very different from those of normal instances. The logic is simple, the more divisions that should do to isolate a sample, the less likely it is an outlier. You can check the image below, where the red dot represents the sample checked for being an outlier.

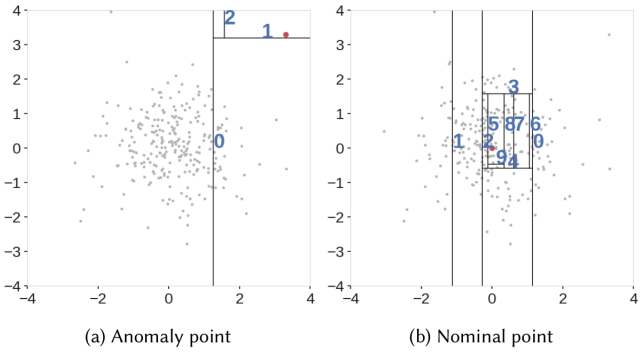


Figure 72: Visualization Isolation Forest.

The scikit-learn library provides an implementation of Isolation Forest

```
from sklearn.ensemble import IsolationForest
import pandas as pd

# Read data
df = pd.read_csv(r'path_to_data\data.csv')
# Create an instance of the algorithm
model=IsolationForest()
# Apply algorithm to data
df['outlier']=pd.Series(model.fit_predict(df.values))
# Remove rows where there is -1 in created column
df=df[df.outlier!=-1]
# Drop created column
```

```
df.drop(['outlier'], axis=1, inplace=True)
```

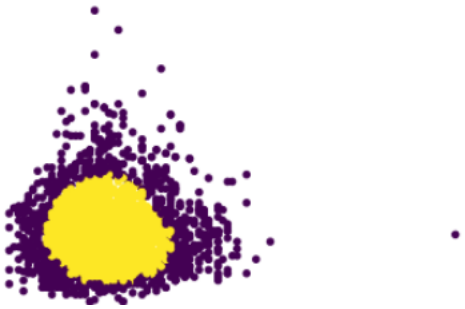


Figure 73: Visualization of results from Isolation Forest.

In the result, we get a new data set that does not contain outliers detected by Isolation Forest.

16.1.2 Elliptic Envelope.

Elliptic Envelope is an anomaly detection algorithm that creates an imaginary elliptical area around a given data set. Samples that fall inside the envelope are considered normal data, and anything outside the envelope is viewed as an outlier. The algorithm works best if the data has a Gaussian distribution.

The scikit-learn library provides access to this method via the EllipticEnvelope class.

```
from sklearn.covariance import EllipticEnvelope
import pandas as pd

df = pd.read_csv(r'path_to_data\data.csv')
model=EllipticEnvelope()
df['outlier']=pd.Series(model.fit_predict(df.values))
df=df[df.outlier!=-1]
df.drop(['outlier'], axis=1, inplace=True)
```

16.1.3 Local Outlier Factor.

The Local Outlier Factor is an outlier detection algorithm based on locating the examples far from other samples in the feature space. Each of the examples is assigned a scoring of how likely it is to be an anomaly based on the size of its local neighborhood - the bigger the score, the more it is probable that they are outliers. Although it might become less

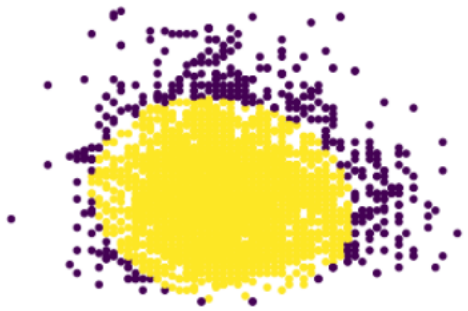


Figure 74: Visualization of results from Elliptic Envelope.

reliable as the number of features increases, referred to as the curse of dimensionality.

The scikit-learn library provides an implementation of this approach in the LocalOutlierFactor class.

```
from sklearn.neighbors import LocalOutlierFactor
import pandas as pd

df = pd.read_csv(r'path_to_data\data.csv')
model=LocalOutlierFactor()
df['outlier']=pd.Series(model.fit_predict(df.values))
df=df[df.outlier!=-1]
df.drop(['outlier'], axis=1, inplace=True)
```

Figure 75: Visualization of results from Local Outlier Factor.

16.1.4 One-Class SVM.

One Class SVM is an algorithm that uses the support vector machine (SVM), an algorithm developed initially for binary classification but can also be used for one-class classification, and it can be a good outlier detector as a result. It captures the density of the majority class and classifies examples on the extremes of the density function as outliers.

The scikit-learn library provides an implementation of one-class SVM in the OneClassSVM class. The class provides the "nu" argument that specifies the approximate ratio of outliers in the dataset, which defaults to 0.1.

```
from sklearn.svm import OneClassSVM
import pandas as pd

df = pd.read_csv(r'path_to_data\data.csv')
model=OneClassSVM(gamma='auto')
df['outlier']=pd.Series(model.fit_predict(df.values))
df=df[df.outlier!=-1]
df.drop(['outlier'], axis=1, inplace=True)
```

Figure 76: Visualization of results from One-Class SVM.

16.1.5 Box-Plot.

One trick to detect whether your data contains outliers is to use a Box plot. Look at the image below. Here, you can see dots outside the range limited by the vertical lines. By seeing these dots, you can assume that the data has outliers that are pretty far from the distribution of the average data samples.

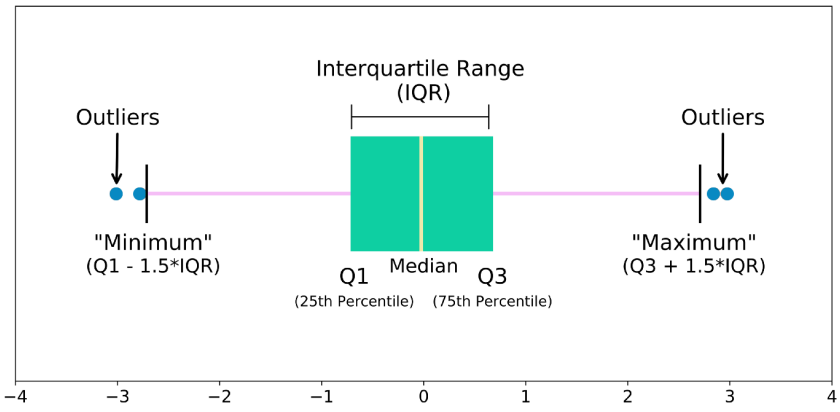


Figure 77: Box-plot.

16.2 Conclusion:

Now you should understand why you might still get errors even when you think your model is 100% accurate. There might be outliers in the new data you test on. Outlier detection is a significant step to making your data clean and beautiful and being aware of the data you must work with.

17 Feature Engineering

You already know that Machine Learning algorithms take numerical input representation of different events or objects shown to the algorithm during the training phase. However, not every numerical representation fits well in every Machine Learning Algorithm. For example, Models based on distance as KNN or SVM, are very sensitive to feature scale, while CARTs don't care about feature scale. Also, linear models, like Linear and Logistic Regression, are susceptible to the numerical representation of categorical features. However, dummy variables can easily solve this problem. These two cases are just two examples of feature adaptation under the Model's specific needs. However, features rarely come into a perfect numerical representation. We at Sigmoid created a python library for these specific needs - imperio. It has a bunch of methods for feature engineering - categorical and numerical at the same time.

During this chapter, we will meet some classical methods for feature engineering and more advanced techniques using the imperio library.

17.1 Classical Methods.

17.1.1 Feature Scaling.

To better understand the logic behind Feature Scaling and why we need it, let's dive deeper into some Machine Learning Algorithms formulas. Below you can see the formula for Gradient Descent used in the Logistic Regression Model to update the weights:

$$w_j = w_j - \alpha \frac{1}{m}(\hat{y}_i - y_i)x_j^{(i)}$$

Having features of different scales (for example - height measured in meters and weight in kilograms) makes the path to the global minimal have a zigzag shape. However, we can solve this problem by bringing all features to the same scale (-3 to +3 or 0 to 1). Below you can see a representation of this process:

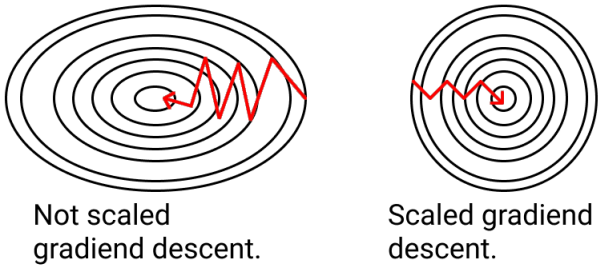


Figure 78: Gradient descend: not scaled data vs. scaled data.

As you can see in the image above, the gradient descent with the scaled features is going more smoothly and faster.

There are two main types of feature scaling: Normalization and Standardization.

Normalization (a.k.a. Min-Max Scaling) is the process of bringing all features values between 0 and 1. This can be achieved by subtracting the min vector (the vector with the smallest values in each column) from the feature vector. This result then is divided by the difference between the max vector (the vector with the biggest values in each column) and the min vector, or just by applying the following formula:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

In scikit-learn it can be applied using the **MinMaxScaler** from **sklearn.preprocessing**:

```
from sklearn.preprocessing import MinMaxScaler
min_max = MinMaxScaler()
X_train_scaled = min_max.fit_trainsform(X_train)
X_test_scaled = min_max.transform(X_test)
```

All transformers in scikit-learn (and imperio) follow a specific API, meaning that every transformer implements the following functions:

1. fit(X, y=None, **fit_params): This function fits up the transformer.

    - X - the feature numpy matrix.
    - y - the target numpy array, default = None.
    - fit_params - the fit parameters that control the fitting process.

2. transform(X, **fit_params): The transform function transforms the passed data.

    - X - the feature numpy matrix.
    - fit_params - the fit parameters that control the fitting process.

3. fit_transform(X, y=None, **fit_params): This function fits the transformer and transforms the data on which it fits.

    - X - the feature numpy matrix.
    - y - the target numpy array, default = None.
    - fit_params - the fit parameters that control the fitting process.

Standardization is the process of bringing the data between -3 and 3 or, saying in another way bringing the numerical series to a series with the mean equal to 0 and standard deviation equal to 1. This is done by applying the following formula:

$$x' = \frac{x - \mu}{\sigma}$$

During this transformation, the mean vector is subtracted from the feature vector, then divided by the vector of standard deviations. In scikit-learn it can be applied using the **StandardScaler** from **sklearn.preprocessing**:

```
from sklearn.preprocessing import StandardScaler
std = StandardScaler()
X_train_scaled = std.fit_trainsform(X_train)
X_test_scaled = std.transform(X_test)
```

An obvious question may appear: what is the practical difference between normalization and standardization? A geometric interpretation may help with this:

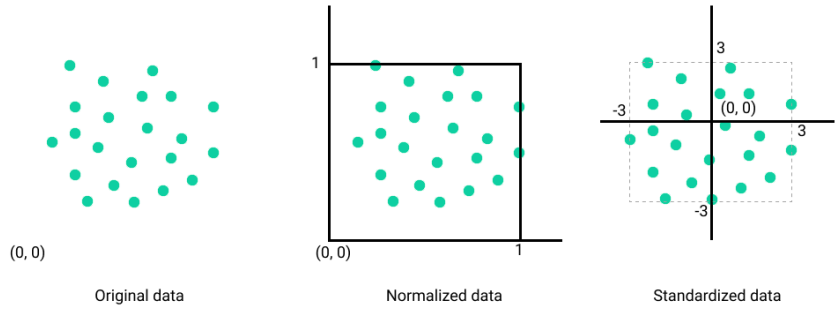


Figure 79: The difference between Normalization and Standardization.

As you can see, the normalization is just bringing the data between 0 and 1, while standardization is centering the data too, so it is more favorable for SVM models.

17.1.2 Dummy Variable.

We already know that for Machine Learning Algorithms, we must represent features as numbers. This means that categorical values will be represented as arbitrary numbers starting with 0, which, to be honest, isn't the best idea. In the plot below are four categories represented as numbers and the mean value of the target column:

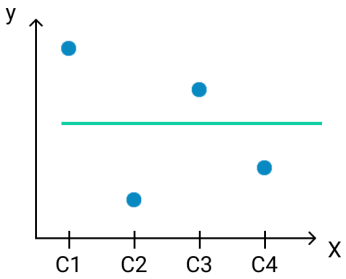


Figure 80: The representation of categories with arbitrary numbers.

As you can see, the red line represents a linear model that, well, doesn't learn anything. It happens because we had zero (in this case, usually, it is a random) correlation between X and y. So we need another way to represent categorical features, and here comes the dummy variable.

Dummy Variable (a.k.a. Indicator Variable) is an artificial variable created to represent an attribute with two or more categories. We will see an example that uses a categorical feature with only two categories,

but we can apply the same logic to a categorical feature with even more categories.

Suppose we want to create a model that will predict the wage of an employee depending on their gender:

$Salary = f(Gender)$

The Linear Regression models will mathematically look like this:

$Salary = w_0 + w_1 \times Gender$

Dummy Variable means creating a separate binary column for every category in the categorical feature. In these columns, we have 1 when the respective column name is equal to the respective categorical value, as shown below:

Gender	Gender_Male	Gender_Female
Male	1	0
Female	0	1

With these two columns added, the Linear Regression model looks like this:

$Salary = w_0 + w_1 \times Gender\_Male + w_2 \times Gender\_Female$

The table for the Linear Regression (with arbitrary data) will look like this:

Salary	Constant	Male	Female
Salary1	1	1	0
Salary2	1	0	1
Salary3	1	0	1
Salary4	1	1	0
Salary5	1	0	1

Now let's add a column that represents the sum of the "Male" and "Female" columns:

Salary	Constant	Male	Female	Calculated
Salary1	1	1	0	1
Salary2	1	0	1	1
Salary3	1	0	1	1
Salary4	1	1	0	1
Salary5	1	0	1	1

Now we can see that the "Calculated" and "Constant" columns are the same. This breaks the assumption of linear regression that observations should be independent of each other, and this is what we call a

dummy variable trap. By adding all the dummy variables in the data, we have compromised the accuracy of the regression model.

To avoid a dummy trap, we should always add (n-1) dummy variables from n categories in the category column because the nth dummy variable is redundant as it carries no new information. Now we will have the following formula after removing the Female column:

$Salary = w_0 + w_1 \times Gender\_Male$

And the following table:

Salary	Constant	Male
Salary1	1	1
Salary2	1	0
Salary3	1	0
Salary4	1	1
Salary5	1	0

This technique is usually used when we have more than two categories in a column, and we recommend you use it when your column has fewer than ten categories.

To apply dummy variables, you can use pandas function get_dummies:

```
import pandas as pd
df = pd.get_dummies(df, columns=['cat_column'], drop_first
   = True)
```

Usually, **get_dummies** requires the following arguments:

- **df** - the data frame on which to apply the dummy variable.
- **columns** - the list of names of columns to compute dummy variables from.
- **drop_first** - a boolean argument that controls whether to drop or not the first column from newly created dummy variables.

17.2 Advanced Imperio Methods: Numerical features.

Till now, we learned the classical methods of feature engineering. However, there are some more. At the moment of writing this chapter, Imperio has 11 data transformers integrated with scikit-learn API. We will go through some numerical and categorical ones.

17.2.1 Box-Cox Transformer.

One of the biggest problems with data in Data Science is its distribution. Almost every single time, it isn't normal. It happens because we cannot have all samples in the world in one data set. However, there exists a bunch of methods that can change that. Let's take a look at the Box-Cox transformation.

Box-Cox transformation is a transformation of a non-normal variable into a normal one. The normality of the data is a fundamental assumption in statistics. From the point of view of statistics, it allows you to run more statistics tests on the data, while from the Machine Learning point of view, it will enable algorithms easier to learn. Boc-Cox applies the following formula to the data:

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

And it gives the following effect:

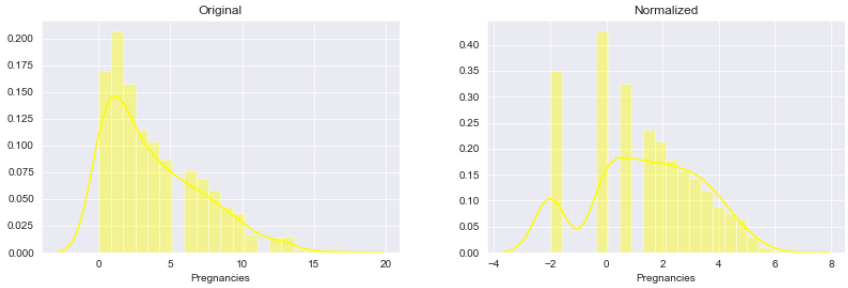


Figure 81: The effect of Box-Cox transformation on feature distribution.

To use Box-Cox transformation or any other transformer from imperio, you should first install imperio.

```
pip install imperio
```

All imperio transformers can be used in the same way as scikit ones:

```
from imperio import BoxCoxTransformer
boxcox = BoxCoxTransformer()
boxcox.fit(X_train, y_train)
X_transformed = boxcox.transform(X_test)
```

As said, it can be easily used in a scikit-learn pipeline:

```
from sklearn.pipeline import Pipeline
```

```
from imperio import BoxCoxTransformer
from sklearn.linear_model import LogisticRegression
pipe = Pipeline(
   [
    ('boxcox', BoxCoxTransformer()),
    ('model', LogisticRegression())
])
```

Besides the scikit-learn API, Imperio transformers have an additional function that allows the transformers to be applied on a pandas data frame:

```
new_df = boxcox.apply(df, target = 'target', columns=['col1
   '])
```

The BoxCoxTransformer constructor has the following arguments:

- **l** (float, default = 0.5): The lambda parameter used by Box-Cox Algorithm to choose the transformation applied to the data.
- **index** (list, default = None): The list of indexes of the columns to apply the transformer on. If set to None, it will be applied to all columns.

The apply function has the following arguments.

- **df** (pd.DataFrame): The pandas DataFrame on which the transformer should be applied.
- **target** (str): The name of the target column.
- **columns** (list, default = None): The list with the names of columns on which the transformer should be applied.

We highly recommend you apply this transformer only to the numerical columns. Applied to the Pima diabetes data set, we get a rise from 0.72 to 0.77 in accuracy. The confusion matrices can be viewed below:

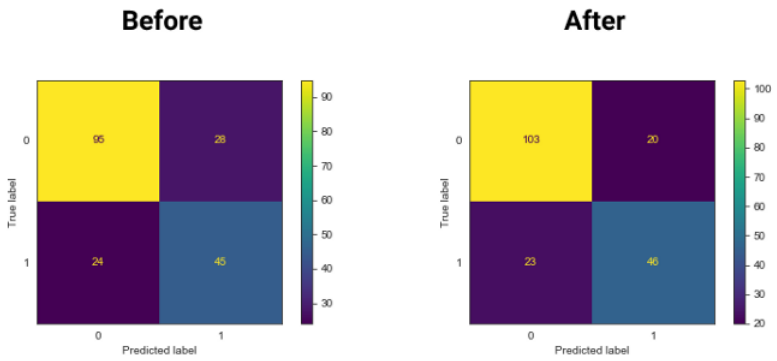


Figure 82: The confusion matrices before and after applying Box-Cox transformation

17.2.2 ZCA Transformation.

Multicollinearity is a known problem in Data Science. It happens that in the feature set, some features are intercorrelated. This is bad because it reduces the estimated coefficients' precision, which weakens your regression model's statistical power. That's why we would need a way to escape this situation. And here comes the ZCA transformation.

ZCA (or zero component analysis) transformation uses Eigenvalues and Eigenvectors to whiten the data, as shown below:

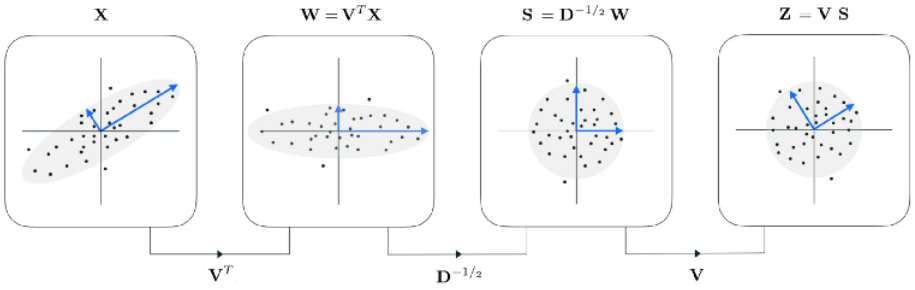


Figure 83: The representation of the ZCA Transformation.

To use the ZCATransformer from imperio, follow the code snippet below:

```
from imperio import ZCATransformer
zca = ZCATransformer()
zca.fit(X_train, y_train)
X_transformed = zca.transform(X_test)
```

Also, it can be integrated into a scikit-learn pipeline:

```
from sklearn.pipeline import Pipeline
from imperio import ZCATransformer
from sklearn.linear_model import LogisticRegression
pipe = Pipeline(
    [
        ('zca', ZCATransformer()),
        ('model', LogisticRegression())
])
```

The ZCATrasnformer constructor takes only one argument:

- **index** (list, default = None): The list of indexes of the columns to apply the transformer on. If set to None, it will be applied to all columns.

The apply function has the following arguments.

- **df** (pd.DataFrame): The pandas DataFrame on which the transformer should be applied.
- **target** (str): The name of the target column.
- **columns** (list, default = None): The list with the names of columns on which the transformer should be applied.

Applied on the Pima diabetes data set, it increases the accuracy from 0.77 to 0.83. The confusion matrices are shown below:

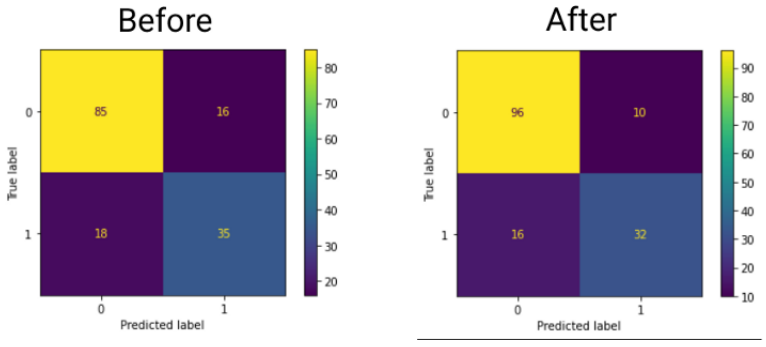


Figure 84: Confusion matrices before and after applying ZCA transformation.

17.3 Advanced Imperio Methods: Categorical features.

Imperio also has some transformers for categorical features. A question may arise in your mind: Why do we need another method if we have

dummy variables? The problem with the dummy variable approach is that the data can become very sparse when you have a column with many categories, which isn't the best data representation for a Machine Learning model. We propose two other methods to handle this: Frequency and Target imputation.

17.3.1 Frequency Imputation.

We talked before that a usual way to encode categorical values is by replacing them with random integers. However, it doesn't contain much information about the feature itself. Frequency Imputation tries to replace the category with the frequency of the category. Suppose half of your categorical column represents category A, 30% of it category B, and the rest is category C. The A category will be replaced with 0.5, B with 0.3, and C with 0.2, as shown below:

Before	After
CategoryA	0.5
CategoryA	0.5
CategoryB	0.2
CategoryC	0.3
CategoryB	0.2
CategoryA	0.5
CategoryC	0.3
CategoryA	0.5
CategoryC	0.3
CategoryA	0.5

this method assumes that if a category is more frequent, it is probably more important while making predictions. That's why it needs a higher encoding score. **However, this method fails when you have two or more categories with the same frequency, so be careful.**

To use the imperio Frequency Imputation Transformer from imperio, follow the code snippet below:

```
from imperio import FrequencyImputationTransformer
freq = FrequencyImputationTransformer(index = [2, 6, 8, 10,
    11, 12])
freq.fit(X_train, y_train)
X_transformed = freq.transform(X_test)
```

Don't forget to indicate the indexes of the categorical values!

And also it can be integrated with the scikit-learn API:

```
from sklearn.pipeline import Pipeline
from sklearn.linea_model import LogisticRegression
pipe = Pipeline(
   [
    ('freq', FrequencyImputationTransformer(index=[10, 11,
   12]),
    ('model', LogisticRegression())
   ]
)
```

The FrequencyImputationTransformer constructor has the following arguments:

- **index** (list, default = 'auto'): The list of indexes of the columns to apply the transformer on. If set to None, it will be applied to all columns.
- **min_int_freq** (int, default=5): The minimal number of categories a column must have to be transformed. Used only when the index is set to 'auto'.

The apply function has the following arguments.

- **df** (pd.DataFrame): The pandas DataFrame on which the transformer should be applied.
- **target** (str): The name of the target column.
- **columns** (list, default = None): The list with the names of columns on which the transformer should be applied.

Now let's apply it to the Heard Disease UCI data set. However we will apply it only on the non-binary columns: 'cp', 'restecg', 'exang', 'slope', 'ca', and 'thal'. Below are illustrated the confusion matrices before and after applying the transformation. We got 3% more accuracy.

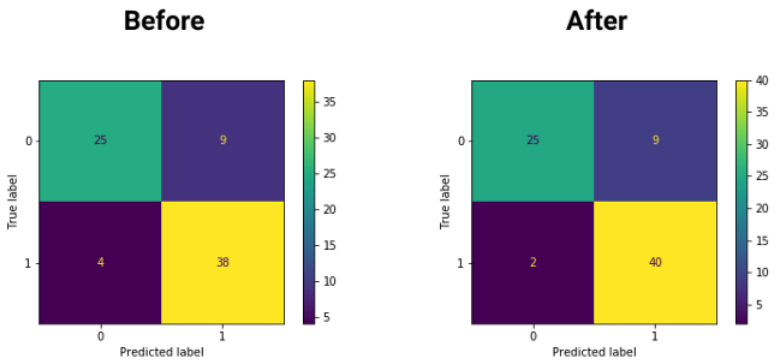


Figure 85: Confusion matrices before and after applying Frequency Imputation

17.3.2 Target Imputation.

As we just saw, Frequency Imputation solves the problem with the data sparsity created by the Dummy Variable Method. However, it brings another problem - **categorical collision** - when two or more categories have the same frequency in the data, they will be represented with the same value and seem unified in one category. Target Imputation comes here to solve this problem.

The idea behind Target Imputation is pretty simple too. In case of regression problems, we replace the category with the mean of the target column for this category, as shown below. In the case of classification problems, we replace categories with the frequency of the most common class for this category.

Before	Encoded column	Target column
CategoryA	154	100
CategoryA	154	240
CategoryB	550	500
CategoryC	30	30
CategoryB	550	600
CategoryA	154	130
CategoryC	30	20
CategoryA	154	200
CategoryC	30	40
CategoryA	154	100

This method assumes that in such a way, the new representation will present more information about the target column, and there are smaller

chances to meet two or more categories with the same mean value of the target column.

To use the imperio Frequency Imputation Transformer from imperio, follow the code snippet below:

```
from imperio import TargetImputationTransformer
target = TargetImputationTransformer(index = [2, 6, 8, 10,
   11, 12])
target.fit(X_train, y_train)
X_transformed = target.transform(X_test)
```

Don't forget to indicate the indexes of the categorical values!

And also it can be integrated with the scikit-learn API:

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
pipe = Pipeline(
    [
        ('target', TargetImputationTransformer(index = [10,
    11, 12]),
        ('model', LogisticRegression())
    ]
)
```

The TargetImputationTransformer constructor has the following arguments:

- **reg** (bool, default = True): A parameter that indicates wherever it is a classification or regression task. The apply function has the following arguments.
- **index** (list, default = 'auto'): The list of indexes of the columns to apply the transformer on. If set to None, it will be applied to all columns.
- **min_int_freq** (int, default=5): The minimal number of categories a column must have to be transformed. Used only when the index is set to 'auto'.

The apply function has the following arguments.

- **df** (pd.DataFrame): The pandas DataFrame on which the transformer should be applied.

- **target** (str): The name of the target column.
- **columns** (list, default = None): The list with the names of columns on which the transformer should be applied.

Now let's apply it to the Heard Disease UCI data set. However we will apply it only on the non-binary columns: 'cp', 'restecg', 'exang', 'slope', 'ca', and 'thal'. Below are illustrated the confusion matrices before and after applying the transformation. We got 6% more accuracy.

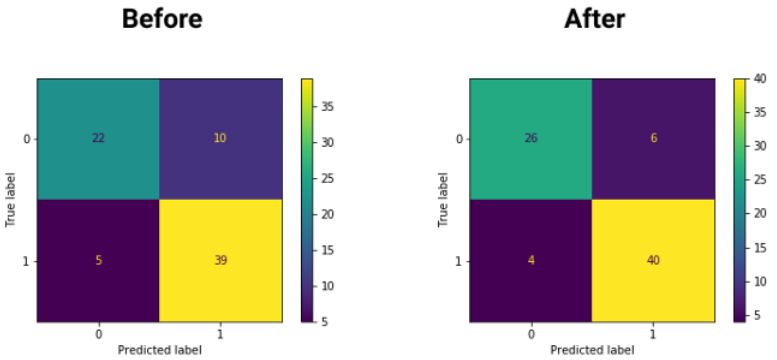


Figure 86: Confusion matrices before and after applying Target Imputation

17.4 Conclusion.

In this chapter, we learned what feature engineering is and why we need it. We went through some classical methods like feature scaling and Dummy Variables. Also, we looked at some more advanced and specialized techniques from imperio. We highly encourage you to learn more techniques, including the ones from imperio, which you can find on our website.

18 Feature Selection

Maybe you already understood from the topics learned previously that Machine Learning is all about feature representation. Every model tries to build a feature representation that may be useful in solving the given task. CARTs are trying to build a tree and find the needed threshold to separate the classes better. The Naive Bayes use the distribution of the features to model the probability of different outputs. The linear models represent the feature by their weights.

However, not all features may be helpful for the representation to solve the problem. Look at the Venn diagrams illustrated below. In the first diagram, we build a model using three features. In the second one, we only use two of them. The intersection area in the first image is much lower than in the second. This representation means that using just two features (right), you can better explain the value you are trying to explain than the one you get using all three values.

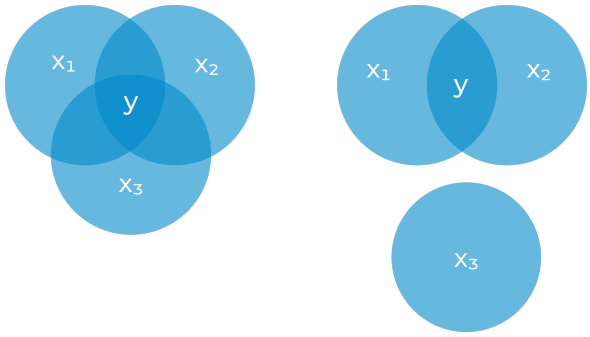


Figure 87: Feature selection.

Also, Machine Learning models are, at a glance, just algorithms that have their computation complexity. Below is the train time and space complexity of some models.

Model	Train time complexity	Space complexity
KNN	$k \times n \times d$	$n \times d$
Logistic Regression	$n \times d$	d
SVM	n^2	$s \times d$
CART	$n \times log(n) \times d$	
Random Forest	$n \times log(n) \times d \times t$	
Naive Bayes	$n \times d$	$c \times d$

In the table above **n** represents the number of rows in the data set, **d** the number of columns, **k** - number of neighbors, **s** - number of support

vectors and **t** - the number of trees. As you can see, the number of features every time influences the algorithm's complexity.

Usually, feature selection requires mathematical knowledge combined with a good understanding of algorithms, bringing all this usually to hundreds of lines of code. That's why here at Sigmoid, we work on one of our libraries - kydavra - a package created especially for feature selection. In this chapter, we will pass through the theoretical work mechanism for feature selection algorithms and their implementation in kydavra.

18.1 Regression Feature Selection.

18.1.1 P-value-based feature selection.

Before we dive deeper into p-values, we should first find out what the null hypothesis is. **The null hypothesis is a general statement that there is no relationship between two measured phenomenons (these phenomenons can also represent features).**

So, to find if a feature is related to the target label, we need to see if we can reject the null hypothesis. And here, we need p-values.

P-value - is the probability value for a given statistical model that, if the null hypothesis is true for a set of statistical observations, is greater than or equal in magnitude to the observed results.

This is a little hazy, I know. Let's reformulate it. The notion of p-values can be expressed more easily as **the probability of finding such observations out of our data set, considering that the feature isn't related to the label by the null hypothesis.** So, if the p-value is big, then there is little chance that using this feature in a production model will get good results.

The algorithm implemented in kydavra is illustrated in the following figure.

Everything starts with setting a significance level, usually noted as α equal with 0.05, sometimes with 0.1. The smaller the significance level, the fewer features will remain. After this, we must somehow compute the p-values for every feature in the data set. This task is taken in kydavra by the OLS model from the statsmodels library.

After getting the p-values for all features, we are searching for the feature with the highest p-value. If the p-value of the feature is lower than the significance level, then we are returning the set of features as the selected features. Otherwise, we are removing this feature from the

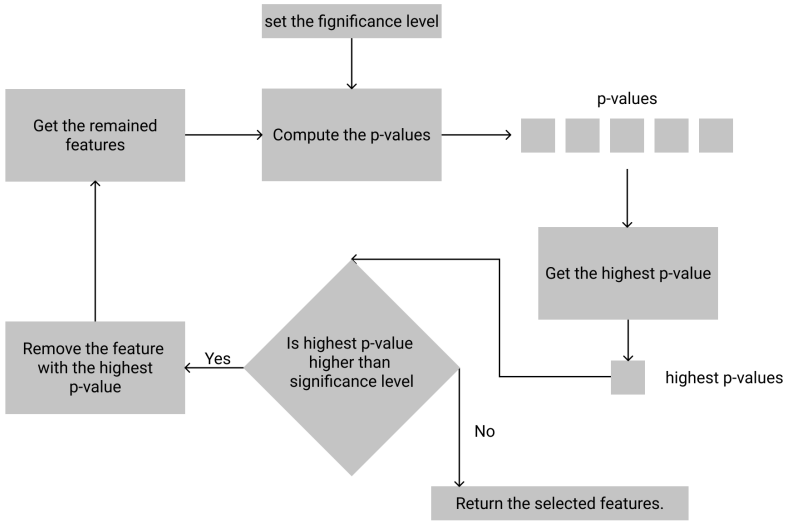


Figure 88: The algorithm of the PValueSelector from kydavra.

feature set and will pass through the whole cycle again with the new feature set. The algorithm will repeat this cycle until the highest p-value is lower than the significance level.

This logic is implemented in the kydavra PValueSelector. If you haven't installed till now kydavra you can do this just by running the following command in the terminal (for Linux or macOS) or cmd (for Windows):

```
pip install kydavra
```

To apply the selector on a data set, you should import it from kydavra, create an instance of it and apply the select function on the data frame passing the name of the target column too:

```
In:
from kydavra import PValueSelector
pvalue_selector = PValueSelector()
selected_columns = pvalue_selector.select(df, 'target')
```

In the code snippet above, the 'select' function takes two arguments: the data frame to apply the selection on and the name of the target column. It returns a list with the names of the selected columns.

After applying the feature selection with PValueSelector on the data frame, we can create a plot to see how the feature's p-value changes through iterations. This can be made by calling the plot_process function:

```
In:
```

```
pvalue_selector.plot_process()

Out:
Figure 79
```

Then you will get a plot like this:

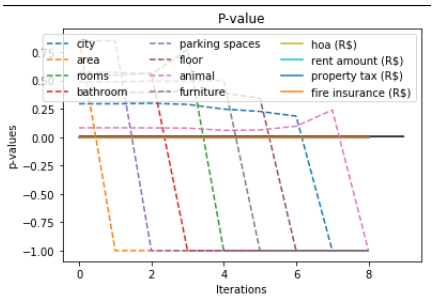


Figure 89: The plot generated by plot_process function of PValueSelector.

There is plotted the p-value of each feature through the iterations of the algorithm. The simple lines show the features kept in the model, and the dotted lines, show the features that the selector eliminates.

18.1.2 LASSO Feature Selection.

Another way to select the best set of features is by using regularization. Regularization is a technique used to reduce the error by fitting appropriately on the given training set and avoiding overfitting. If we are talking about linear models, it means adding a term to the error equation. We have two main types of regularization: L1 and L2. LASSO uses L1-regularization, and its cost function is:

$$E(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \alpha \sum_{j}^{f} |w_j|$$

Using this cost function instead of simple MSE, the weights of the linear function will tend to be 0. The alpha parameter is the parameter that controls the influence of regularization on the error function. **The main idea of this algorithm is to find the value of alpha that gives us the lowest possible error.** Knowing that we can get the weights of the features for the best-performing model (the model with the lowest error). The features with weights equal to or almost equal to 0 aren't important, and we can drop them.

This logic is fully implemented in the kydavra LassoSelector. Following the API of the kydavra selectors, it can be applied to a data frame

as the code snippet shows below:

```
In:
from kydavra import LassoSelector
lasso = LassoSelector()
selected_columns = lasso.select(df, 'target')
```

As usual, the select function takes a pandas data frame and a string that represents the name of the target column is the respective data frame.

The constructor of the LassoSelector takes the following arguments:

- **alpha_start** (float, default = 0) the starting value of alpha interval for searching.
- **alpha_finish** (float, default = 2) the final value of alpha interval for searching.
- **n_alphas** (int, default = 300) is the number of alphas that will be tested during the search.
- **extend_step** (int, default = 20) if the algorithm deduces that the most optimal value of alpha is alpha_start or alpha_finish it will extend the search range with extend_step, in such a way being sure that it will not stick and will find the optimal value finally.
- **power** (int, default = 2) used in 10^{-power} formula, define the maximal acceptable value to be taken as 0.

NOTE: We set the 10^{-power} limit because of the machine error. Most probably, we will never find the perfect solution. However, we can find an acceptable solution near the best one.

Also, you can plot the weights change around the best solution using the plot_process function after selecting the best features:

```
In:
lasso.plot_process()

Out:
Figure 80
```

Below is the plot that we got after applying the LassoSelector on the Avocado Prices data set:

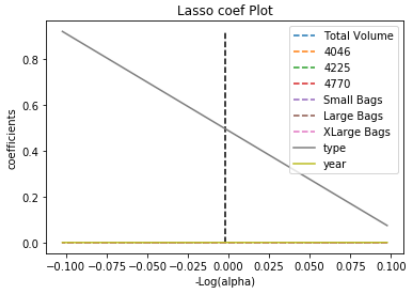


Figure 90: The plot generated by the plot_process of the LassoSelector

On the x-axis is plotted the minus logarithm of alpha and on the y-axis are the values of the weights. The simple lines represent the selected features, while the dotted ones are the features that were dropped out of the model.

18.2 Classification Feature Selection.

18.2.1 Correlation Feature Selection.

Maybe you have already heard about correlation while reading this book or by learning from other online sources. Simply saying **correlation is a measure that shows how similar the distribution of two numerical series is.** There are several types of correlations: Pearson, Spearman, and Kendall. All of them have different formulas and implementations in kydavra. I will deep dive only into the Pearson correlation. The formula of Pearson correlation between two numerical series is:

$$p_{xy} = \frac{cov(x,y)}{\sigma_x \sigma_y}$$

The Pearson correlation between two numerical series is computed by dividing the covariance between the series by the product of the standard deviations of the series. This value is always between -1 and +1. Usually, we are interested in the absolute value of the Pearson correlation to be between 0.5 and 0.8. However, it can change from case to case.

The Pearson Correlation selection is implemented in kydavra as **PearsonCorrelationSelector** (Kendall as **KendallCorrelationSelector** and Spearman as **SpearmanCorrelationSelector**) and as all correlation based selectors can be used as follows.

```
from kydavra import PearsonCorrelationSelector
pearson = PearsonCorrelationSelector()
selected_columns = pearson.select(df, 'target')
```

As always, the select function requires a data frame and the name of the target column. **However it also requires the target column to be represented numerically.**

Also the constructors of every selector have the following parameters:

- **min_corr** (float, between 0 and 1, default = 0.5) the minimal value of the correlation coefficient to be selected as an important feature.
- **max_corr** (float, between 0 and 1, default = 0.8) the maximal value of the correlation coefficient to be selected as an important feature.
- **erase_corr** (bool, default = False) if set to True, then the selector will erase columns that are correlated between them, keeping just one, if False, then it will keep all columns, even if they are correlated between them.

The last parameter was added to control the phenomena of multicollinearity - **when two or more features are correlated between them.** It isn't nice because you are giving the model the same information.

Each Selector is building inside the correlation matrix. Then it is just iterating through all cells to find the absolute value of correlations between **min_corr** and **max_corr**, firstly using the target column and after, the columns that are highly correlated with the target one.

18.2.2 Point-Biserial Feature Selection.

Another type of correlation that is very powerful for classification problems (unfortunately only for binary classification) is **Point Biserial Correlation**. It is the same as the Pearson correlation. The only difference is we are comparing dichotomous data to continuous data instead of continuous data to continuous data (the second drawback, it will ignore categorical features if they aren't represented as numerical ones). Suppose we have two groups we want to learn to classify (or also two classes). To compute the Point Biserial Correlation between a continuous feature and a binary one we will use the following formula:

$$r_{pb} = \frac{M_0 - M_1}{s_y}\sqrt{\frac{n_0}{n}\frac{n_1}{n}}$$

Here M_0 and M_1 is the mean of the continuous features for group 0, and 1 respectively. s_y is the standard deviation of the continuous data.

n_0, n_1 and n are the numbers of samples in the group 0, 1, and the total number of samples in the data set respectively.

However in kydavra, besides just looking if a correlation is between the **min_corr** and **max_corr**, we are also grouping features by their correlation into ranges of length 0.1. And are taking the features on the last two levels (by default).

In kydavra, this method is implemented as **PointBiserialCorrSelector** class and can be used like is showed in the following code snippet:

```
from kydavra import PointBiserialCorrSelector
point_biserial = PointBiserialCorrSelector()
selected_column = point_biserial.select(df, 'target')
```

The select function takes the same arguments as always, while the selector's constructor takes the following arguments:

- **min_corr** (float, between 0 and 1, default = 0.5) the minimal value of the correlation coefficient to be selected as an important feature.
- **max_corr** (float, between 0 and 1, default = 0.8) the maximal value of the correlation coefficient to be selected as an important feature.
- **last_level** (int, default = 2) the number of correlation levels that the selector will take into account, recommended to change only if the returned list of features doesn't satisfy your needs.

To add about Point Biserial Correlation is one of the harshest selectors in kydavra because it keeps very few features after the selection.

18.3 Conclusion.

In this chapter, you discovered the process of feature selection. Why do we need it? How can it help you? And finally, got to know some of the modules implemented in kydavra. Of course, kydavra has many more selectors, reducers, and filters, which we leave for self-study. Sigmoid has a bunch of articles on these topics on our website.

19 Class Balancing.

During this book, you probably already build many classification models. Maybe you already observed that, in most cases, the number of samples in every class isn't always 50/50. You will probably never get such a use-case in the industry, and it's something normal.

Let's see some examples. In the telco industry, companies rarely have problems with clients. Most of us have a phone which needs a SIM card that should be recharged once in a while. However, not all these clients are the same type. One example is the phenomenon of churn - a client that decides not to use the company's services anymore. And trust me, the number of these churns is much lower than the number of all clients. Also, not all telco clients have monthly subscriptions. The majority of them are just recharging their account when they need to. However, for a telco company, a client with a subscription plan is way better than one without because it's a secure income for a couple of years.

These two use-cases have two things in common. First, the company is interested in predicting the probability of conversion of a regular client in one of these two categories (churn and post-point) to make special offers to these clients special offers, to convert them, or to keep them. The second one is that usually, these clients represent a small fraction of the total number of clients.

Now, suppose you should create a model for churn prediction. And you get a data set with the following classes frequencies:

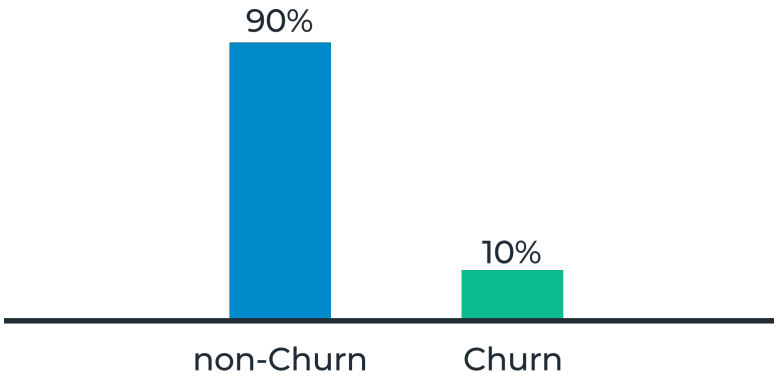


Figure 91: Imbalanced data.

Most probably, when you will train a model, you will get a 0.9 accuracy score. However, your model can be wrong as hell. Let me introduce another measure of a model's performance - the confusion matrix.

A confusion matrix is a $n \times n$ matrix, where n is the number of classes that a can be part of. You can see an example below:

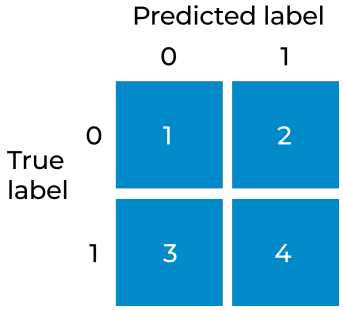


Figure 92: Confusion matrix.

We will dive deeper into this plot in the chapter about model evaluation and interpretation. Still, it will be enough to know that the values in each square represent the number of samples predicted for each class as the "Predicted label" and the number of samples that have that class from the "True label". For example, in the square number 1 will be the number of samples predicted as class 0 and having the true value 0. The main idea of this matrix is that the values from squares 1 and 4 should be as high as possible. We will show the code snippet to make this plot during this chapter. However, most probably, your confusion matrix will look like this:

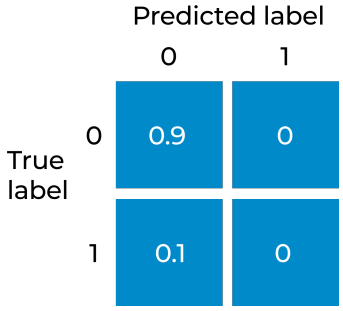


Figure 93: Confusion matrix of a model trained on unbalanced data.

This plot is a sign that your model is affected by class imbalance, meaning that it is usually more effective to predict all samples just as the most frequent class rather than to learn the difference between them. Because this problem is widespread in real industry use cases, we decided to add a chapter about it to this book. In this chapter, you will learn the classical way to solve this issue and some ways using **crucio** - a library with oversampling techniques.

19.1 Classical Methods.

Before we will go to advanced oversampling methods, let us first learn some classical ways to solve this issue. There are usually three ways to solve it: Undersampling, Oversampling, and using class weights.

19.1.1 Undersampling.

Undersampling is the most straightforward technique to balance an unbalanced data set. The idea is to reduce the number of samples in the majority class to the number equal to the number of samples in the minority class. Usually, it is done by randomly selecting m samples from the majority class, where m is the number of samples in the minority class.

19.1.2 Class weights.

In the examples above with the confusion matrix, we saw that sometimes for a model, it is more convenient to classify all samples as majority class samples. A way to escape this situation is to penalize harder errors by misclassifying minority class samples. In the base case, every error during classification is equal to 1. However, we could escape this situation by giving different weights to different misclassification cases. Below is a confusion matrix of prediction over an unbalanced data:

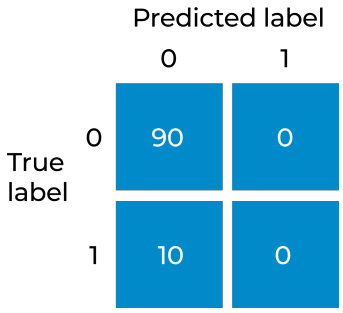


Figure 94: Confusion matrix of a model trained on unbalanced data.

In this case, the error would be equal to 10. However, if we would set for class 1 an error cost of 2, the error will become 20, which is larger, so the model will tend to escape this case. Most models in scikit-learn have an optional parameter class_weight which sets these weights. However, to use them in an unbalanced classification use-case, a best practice is to set it as a "balanced" value like in the example below.

```
logit = LogisticRegression(class_weight='balanced')
```

The models from scikit-learn that have this parameter are:

- LogisticRegression;
- RandomForestClassifier;
- DecisionTreeClassifier;

19.2 Oversampling.

Oversampling is a technique that works backward to undersampling. Instead of reducing the size of the majority class, we are generating new samples for the minority class using different algorithms. The biggest family of such algorithms is SMOTE. There are also other algorithms that make this process in different ways.

19.2.1 SMOTE.

SMOTE (or **Synthetic Minority Oversampling Technique**) is the simplest and the most popular oversampling algorithm. Let's see how it works on a simple example:

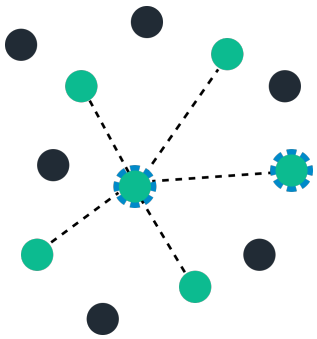


Figure 95: SMOTE: Searching the nearest neighbor.

First, we start by selecting at random a minority sample (in the image above - the green circles). In the picture, the selected sample has a dotted line around it. Then we must find the K nearest neighbors of this sample. From the nearest neighbors, we will choose one randomly (in the image below, represented as the second sample with a dotted line around). In the figure below, the new sample generated using these two samples is represented as a blue circle.

There *gap* is a random number in the interval (0, 1] and the *dist* represents the distance function between 2 points. Every sample is generated in such a way. Usually, it generates M - m samples, where M

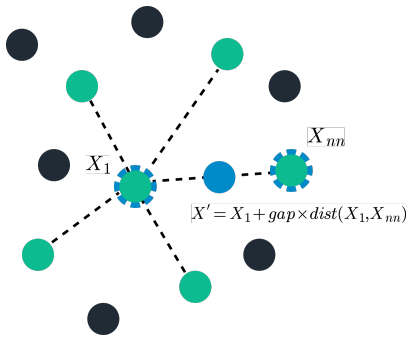


Figure 96: SMOTE: Generation of a synthetic sample.

represents the number of majority samples and m represents the number of minority samples.

To use this algorithm, we will use the crucio library. To install it, type the following command in the terminal or cmd:

```
pip install crucio
```

We will apply all crucio models on the pokemon data set. It stores the statistics about 721 pokemon, and the main task is to predict whether a specific pokemon is legendary or not. Here 8% of the data set represents the legendary type, and the rest are simple pokemon. We recommend you use the following framework for working with unbalanced data:

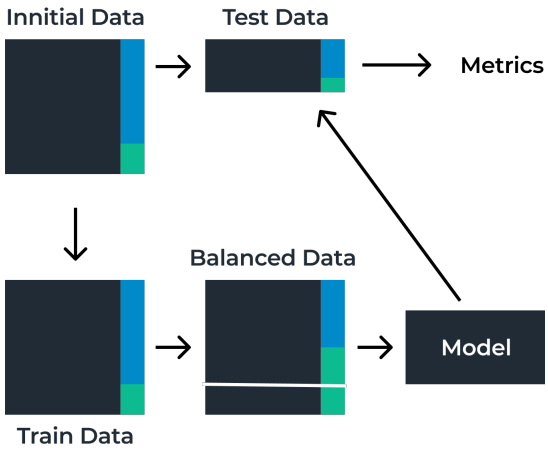


Figure 97: The framework of working with unbalanced data.

Before applying a Balancer to a dataset, you should first split it into a train and test set. We will balance only the train set, fit the model on it, and test using different metrics on the test set, like in the following code snippet:

```
from sklearn.model_selection import train_test_split
train_df, test_df = train_test_split(df, random_state=42)
```

To use the SMOTE model, you should follow the following instruction:

```
smote = SMOTE()
balanced_df = smote.balance(df, 'Legendary')
```

The smote constructor takes the following arguments:

- **k** (int, default = 5): The number of nearest neighbors from which SMOTE will sample data points. It should be higher than 0.
- **seed** (int, default = 42): The number used to initialize the random number generator.
- **binary_columns** (list, default = None): the list of binary columns from a data set, so sampled data can be approximated to the nearest binary value.

The balance function takes the data frame that should be balanced and the name of the target column. Below you can see the confusion matrices of the Random Forest model trained on unbalanced and SMOTE balanced data:

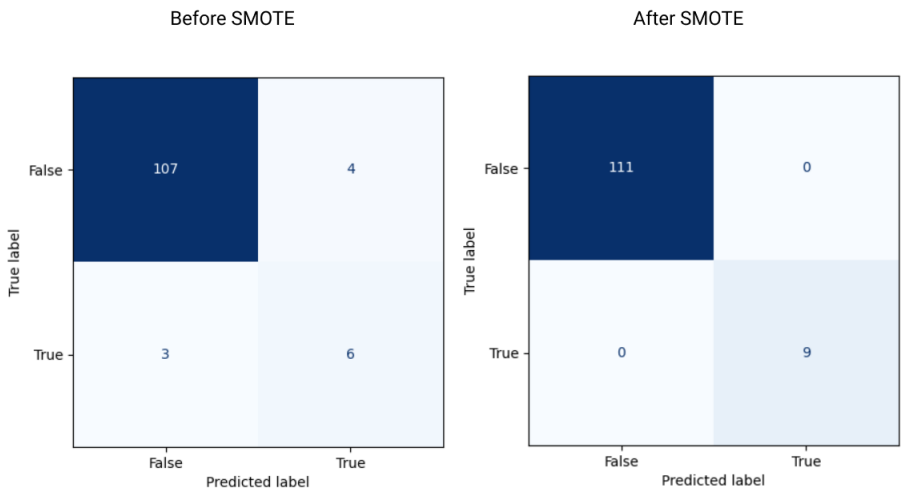


Figure 98: The confusion matrix before and after balancing the data with SMOTE.

We can see that applying SMOTE on the data allowed us to create a model with 100% accuracy. Below you can see how the data looked before and after SMOTE:

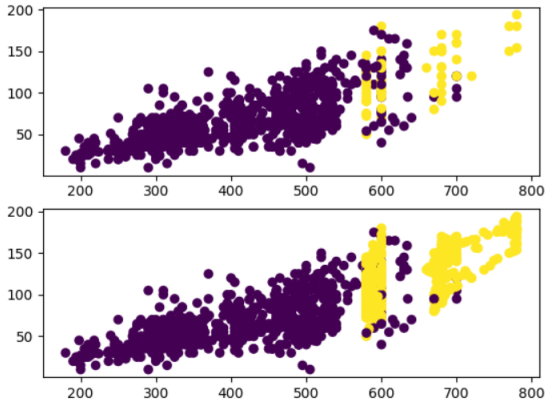


Figure 99: The scatter of the data set before and after applying SMOTE.

As the last note, SMOTE is the unique algorithm in crucio that can be applied to multi-class problems.

19.2.2 SMOTEENN.

One of the main drawbacks of SMOTE is that local outliers may influence it. Sometimes SMOTE can take a minority sample that has only majority samples around, being in the middle of the majority class zone. So taking its nearest minority samples neighbors will generate a minority sample in the middle of the majority class zone. That will directly influence the classifiers. So, we need a way to escape this situation.

SMOTE has a lot of extensions, but here we are going to talk about SMOTEENN. **SMOTEENN** is solving the problem of outliers using the EEN algorithm. ENN (Edited Nearest Neighbor) is an undersampling technique that looks for noise in data. It computes K-Nearest Neighbors for every minority class sample and depending on them; it decides if among those samples exists an imposter, if yes, then this example will be deleted. Take a look at Figure 90.

In the image above, we can see three groups of samples. The ENN algorithm will remove two green (minority samples) and one red (majority) sample because they are local outliers in this case. After the ENN algorithm is applied, a simple SMOTE algorithm is applied to oversample this new data. To use the SMOTEEN from crucio, follow the following code snippet:

```
from crucio import SMOTEENN
smoteenn = SMOTEENN()
balanced_df = smoteenn.balance(df, 'Legendary')
```

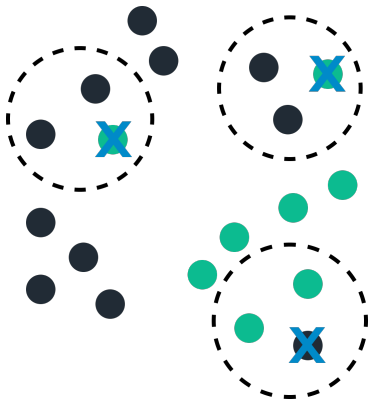


Figure 100: ENN Algorithm.

The parameters of the SMOTEENN constructors and 'balance' functions are the same as in the case of the SMOTE algorithm. Below you can see the performance Random Forest algorithm before and after applying SMOTEENN. We can see that after applying, we got 100% accuracy (Figure 91):

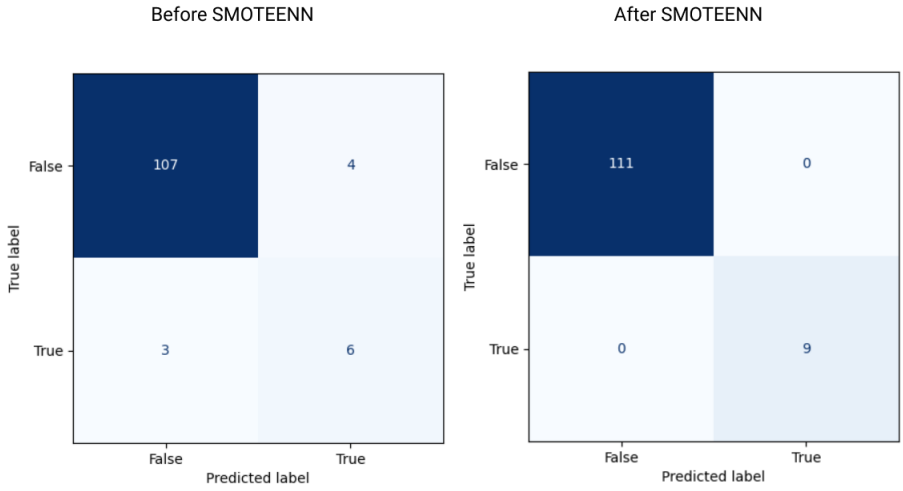


Figure 101: The confusion matrix before and after applying SMOTEENN.

19.2.3 ICOTE.

ICOTE (or **Immune centroids over-sampling method for multi-class classification**) is an oversampling method out of the SMOTE family that generates new minority samples by trying to replicate the principles of Immune systems. ICOTE algorithm can be separated into two phases:

1. Clone generation:

In ICOTE, minority samples are interpreted as white immune cells from the immune system. In an organism, when a "non-self" antigen enters the body, immune cells attack them. The majority class samples represent antigens in ICOTE. Usually, white immune cells create a lot of copies of themselves to attack the antigens. The same idea is used in ICOTE, generating clones of the minority class samples until the number of minority and majority samples is the same.

2. Mutants generation:

If every immune cell is ineffective versus the antigen, it would be a loss of resources to clone these cells. That's why immune cells sometimes mutate while cloning. In such a way, they make their chances of success higher. A similar mechanism is used in ICOTE. However, the mutation isn't random.

The distance to every majority class sample is calculated for every clone. Using this distance, the alpha parameter is calculated for every clone, as shown below (the inverse of the distance). This alpha is used to mutate the clone using the below formula.

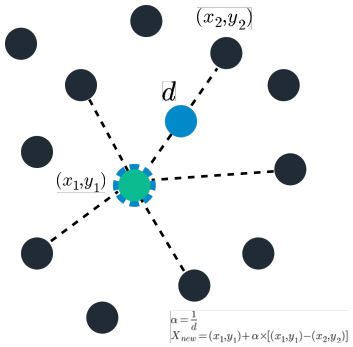


Figure 102: The generation of mutants in ICOTE.

Using this 2-step algorithm, we generate synthetic samples till the number of minority and majority is the same. To use ICOTE from crucio, you can follow the code snippet below:

```
from crucio import ICOTE
icote = ICOTE()
balanced_df = icote.balance(df, 'target')
```

ICOTE constructor has only two parameters:

- **seed** (int, default = 42): The number used to initialize the random number generator.

- **binary_columns** (list, default = None): the list of binary columns from a data set, so sampled data can be approximated to the nearest binary value.

The balance function takes, as always, the same arguments: a pandas Data Frame and the name of the target column.

19.2.4 TKRKNN.

TKRKNN (or **Top-K Reversed KNN**) has a very different way of generating new samples compared to the SMOTE family or ICOTE. The difference is that it generates new samples non-linearly - meaning that the new sample isn't somewhere on the line that links two samples. Let's see how it does that.

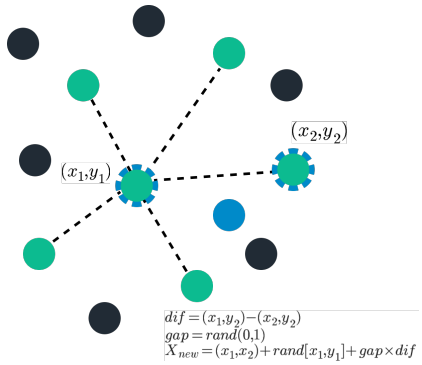


Figure 103: The generation of new samples in TKRKNN.

Firstly, we start by finding the K-nearest neighbours for all samples of the minority class. Then, TKRKNN finds the number of samples to generate equal to the number of minority samples subtracted from the number of majority samples.

Now, to generate every new sample, we are taking a random minority class sample and a random nearest neighbour. For every attribute of these vectors, we calculate the difference. This difference between these attributes is then multiplied by a random number - the gap, a scalar between 0 and 1, added to the corresponding attribute to the initially selected minority class samples. This modified minority sample is then added to the synthetic data generated. In such a way, the new sample has only some modified features.

To apply the TKRKNN algorithm from crucio, you should follow the next code snippet:

```
from crucio import TKRKNN
```

```
tkrknn = TKRKNN
balanced_df = tkrknn.balance(df, 'Legendary')
```

The parameters of the TKRKNN constructor and the balance function are the same as in the case of the SMOTE algorithm.

19.3 Conclusion.

In this chapter, we saw what unbalanced data is and how it can hurt your model's performance. Also, we saw some ways to solve this problem. The most studied ones were the oversampling techniques from crucio. We studied two members of the SMOTE family (SMOTE and SMOTEENN), followed by ICOTE and TKRKNN. However, crucio has a bigger arsenal of oversampling methods, especially the SMOTE family. Please go and check them on our website.

20 Model interpretation.

In previous chapters, you learned a bunch of Machine Learning Models and how they learn from data useful (and sometimes less useful) data representations. Also, you learned how to increase the model's performance by applying different preprocessing techniques such as Feature Engineering, Feature Selection, and more. Now suppose you have a Machine Learning model. How do you know it is good for the problem you are trying to solve?

Accuracy and prediction errors may be the first ideas that come up in your mind. From the previous chapter on Class Balancing, you learned that high accuracy doesn't always mean a good model, especially on unbalanced data.

Let me give you an example of that. Suppose you are asked to create a model that will diagnose lung cancer. In the US, only 16% survive five years after diagnosis. So, it is essential to have a highly performant model for this problem. Suppose we got a model with 99% on a test data (and even more, it was tested for unbalanced data effect on models). Would you trust this model?

I would say no! Why? In the fields where a wrong prediction has very high costs, be it financial or human lives, as in medicine, we cannot trust a model just by its accuracy. **"The problem is that a single metric, such as classification accuracy, is an incomplete description of most real-world tasks."** (Doshi-Velez and Kim 2017).

In such fields, we need something named interpretability. Usually, Machine Learning Models are taken as black boxes. You give input, and the model somehow processes it and returns the output. But we often need to figure out how this output was generated. Also, sometimes models can be unstable, and a small change in a feature can radically change the model's prediction. Also, there are cases when we are more interested in **why** this prediction than **what** prediction.

In this chapter, you will learn how to interpret a Machine Learning model, starting with some classical methods and finishing with so-named agnostic approaches which don't care about the model you are trying to explain - LIME and SHAP.

20.1 Classical Methods.

Some models are interpretable by their nature, while others aren't. Below you can see a graph showing a comparison of the model's interpretability

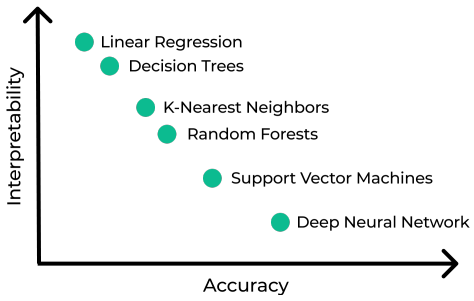


Figure 104: Model comparison: Accuracy vs. Interpretability.

compared with their performance.

As you can see from the plot, there is a negative correlation between the model's accuracy and interpretability. However, the most interpretable models remain the Linear Models and CARTs (Classification and Regression Trees). Let's see why it happens.

20.1.1 Linear Models interpretability.

As you may remember from previous chapters, Linear Models attempt to find a weight vector which, when multiplied using a dot product with the data vector returns prediction, has the smallest SME (Squared Mean Error). Knowing that a vector of values represents the model, we can assume two things:

- If the weight for a feature is equal or near zero, then this feature isn't important because its effect is erased by its weight.
- The higher the absolute value of the weight, the more effect this feature has on prediction, meaning that it is more important. Below is a bar plot showing the weights of a Linear Regression model found in the wine quality data set.

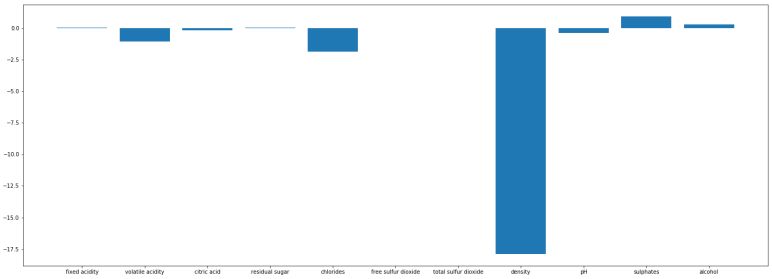


Figure 105: The weights of the linear model.

As we can see from the plot above, based on assumptions we made before, the most significant influence on the wine quality is the 'density' of the wine. Also, we can see that 'total sulfur dioxide' and 'free sulfur dioxide' don't influence the wine's quality. Besides that, the influence of 'fixed acidity' and 'residual sugar' is almost zero.

20.1.2 CART interpretability.

Classification and Regression Trees are another group of Machine Learning algorithms that are easy to interpret. As you probably already know, Decision Trees are built based on data, a tree of rules, where each node tries to separate the classes as much as possible. A good aspect of this approach is that we are getting the rules for splitting the data into pure classes, and we can better understand the data we are dealing with.

There are a couple of ways to plot a Decision Tree. However, the most beautiful one uses the **dtreeviz** library. To install it, just type:

```
pip install dtreeviz
```

After that, you can use the following script to export the tree as an SVG file:

```
In:
from dtreeviz.trees import *
from sklearn.datasets import load_iris
# Loading the data.
iris = load_iris()
X = iris.data
y = iris.target
# Creating the tree visualization.
viz = dtreeviz(model,
x_data=X,
y_data=y,
target_name='class',
feature_names=iris.feature_names,
class_names=list(iris.target_names),
title="Decision Tree - Iris data set")
```

Sometimes, it can generate an error like this:

```
Out:
ExecutableNotFound: failed to execute ['dot','-Tsvg',
'-o','C:\\Users\\ASUSVI~1\\AppData\\Local\\Temp\\
  DTreeViz_19292.svg',
```

```
'C:/Users/ASUSVI~1/AppData/Local/Temp\\DTreeViz_19292'],
make sure the Graphviz executables are on your systems'
   PATH
```

It can be easily solved by running the following command in the terminal or cmd in the case of windows.

```
dot -Tsvg -o C:\\Users\\ASUSVI~1\\AppData\\Local\\Temp\\
   DTreeViz_19292.svg
C:/Users/ASUSVI~1/AppData/Local/Temp\\DTreeViz_19292
```

The result is the following:

Figure 106: The created decision tree.

As we learned before, the nearer (the higher) the node is to the root, the more influential the feature is for splitting for the classification. As we see above, petal length is the most important because it separates the data set into one pure group (with only setosa samples) and one with two classes (Versicolor and Virginica). Also, we can see that, in general, the dimensions of petals are more informative about the species of the iris than the dimensions of the sepals because more nodes use petal information. Also, they are nearer to the root of the tree.

20.2 Other ways to interpret a model.

Linear Models and CARTs are easy to interpret. However, they aren't the most powerful Machine Learning algorithms. In the case of other algorithms, we have tools, metrics, and plots that can be applied to them.

20.2.1 Classification interpretability.

The most used tool for classification is the confusion matrix, illustrated below. We already met it in the chapter on balancing data sets. But let's recapitulate.

Labels		Predicted Label	
		N	P
True label	N	True Negative (TN)	False Positive (FP)
	P	False Negative (FN)	True Positive (TP)

True Negative (TN) is the number of samples representing the negative class and were predicted as negative ones. **False Positive (FP)** is the number of samples that represent the negative class but were predicted as positive ones. **False Negative (FN)** is the number of samples representing the positive class but were predicted as negative. And respectively, **True Positive (TP)** is the number of samples representing the positive class and which were predicted as positive ones.

From the table above, we can infer that a good model should have the TN and TP as high as possible. From this assumption appeared the formula for accuracy:

$$Accuracy = \frac{TP+TN}{TP+TN+FN+FP}$$

To plot the confusion matrix in python, you can use the following snippet code:

```
from sklearn.metrics import plot_confusion_matrix
plot_confusion_matrix(model, X_test, y_test)
```

Where the model is a Machine Learning model while X_test is the feature matrix and the y_test is the label vector.

Besides that, this matrix can be extended to more metrics, but I will tell you about three more. The full matrix can be found on the Wikipedia page.

Precision and Recall.

Sometimes, one class can be more important than another. For example, it is more critical for a lung cancer prediction system to spot as many positive cases as possible, even if it may classify some healthy patients as ill. **Please keep in mind that the cost of an additional control is much lower than a lost life.**

Another example is the churn prediction use case in the telco industry. The idea is that for a telco company, it is cheaper to retain a new customer that is paying monthly than to get new customers. They use models to identify customers with a high risk of churn and try different offers to retain them as much as possible. However, there is no free lunch, and the company can only call a part of its clients which probably will churn. That's why they need a model that will predict as much churn (TP) as possible from the real churn (P).

Similar problems, yeah? At first glance, it may be, but let's compare them. In medicine, you need to spot as many TP as possible from the population, while in telco, you need to have as many TP as possible from the predicted positive population. In medicine, **Recall** is more important, while in telco (churn prediction use case), the golden standard is the **Precision**.

$$Recall = \frac{TP}{TP+FN}$$
$$Precision = \frac{TP}{TP+FP}$$

To compute the precision and recall, you must import the precision_score and the recall_score from sklearn.metrics.

```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score

recall = recall_score(y_pred, y_test)
precision = precision_score(y_pred, y_test)
```

F1-score.

F1-score is the harmonic mean of Precision and Recall. The highest value that F1-score can achieve is 1.0, indicating perfect precision and recall, and the lowest possible value is 0 if either the precision or the recall is zero. This can be inferred from the formula below:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

To compute the F1-score, you must import the f1_score from sklearn.metrics.

```
from sklearn.metrics import f1_score
f1 = f1_score(y_pred, y_test)
```

20.2.2 Regression interpretability.

Regression models have different metrics. The most used ones are Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). Let's get through them.

Mean Squared Error measures the mean euclidean (or second norm) between the correct answers and the predicted ones. It is used to find the optimal weights for the Linear Regression algorithm. From here comes an interesting property of these metrics. Because Linear Regression searches for weights that minimize MSE, the MSE of Linear Regression is usually smaller than on other algorithms. The formula of MSE is the following:

$$MSE = \frac{1}{n} \times \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

To compute the MSE in sklearn, you should import it from sklearn.metrics.

```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_pred, y_test)
```

Mean Absolute Error measures the absolute distance between the correct and predicted values. It can be interpreted as the mean deviation of the model from the true values. Its formula is the following:

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i| = \frac{1}{n} \sum_{i=1}^{n} |e_i|$$

To compute the MAE in sklearn, you should import it from sklearn.metrics.

```
from sklearn.metrics import mean_absolute_error
mse = mean_absolute_error(y_pred, y_test)
```

Root Mean Square Error is basically the root of the MSE, as the formula below shows:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2}$$

To compute the RMSE in sklearn, you should import it from sklearn.metrics.

```
from sklearn.metrics import root_mean_square_error
mse = root_mean_square_error(y_pred, y_test)
```

Besides the metrics listed above, Regression Models can be interpreted using a graph. The idea is to create a seaborn lmplot where the actual values are on the x-axis and the predicted values on the y-axis. Then the lmplot will automatically create a linear function that fits these samples. If the line has a degree around 45°, it's almost a perfect fit. If the degree is higher, the model predicts higher values and vice versa. See below the plot and the code for it:

```
In:
plt.figure(figsize=(10, 10))
predictions = pd.DataFrame(y_pred,columns=['y_pred'])
predictions['y_test'] = y_test
sns.lmplot(data=predictions, x ='y_test',y='y_pred')

Out:
Figure 97
```

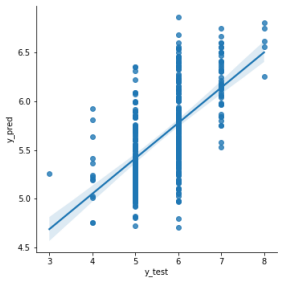


Figure 107: lmplot created for interpreting regression models.

20.2.3 Agnostic methods.

Previous methods present some model agnosticism because they can be applied to all methods. However, they all have a big drawback: they don't give an interpretable enough explanation of what happens under the hood. So it would be nice to have a method combining the explanation details given by linear models or decision trees and agnosticism. Such models are LIME and SHAP. Let's explore them.

20.3 LIME interpretability.

LIME (or Local Interpretability Model-Agnostic Explanations) tries to bring the performance of an interpretable model (linear regression or Decision Trees) to one that isn't as interpretable as SVMs. To understand how it works, we must get into some notations. Let G be the set of potential interpreted models. Then, $g \in G$ is the local explanation of the prediction. The explanation should not be too complex for humans. That's why we need a measure of complexity, noted as $\omega(g)$. In the case of linear models, it is the number of non-zero elements, while in the case of Decision Trees, it is the depth of the tree.

f is the model that predictions we try to explain. So we also need a measure of how g and f are close or a measure of how unfaithful g is approaching f, and it is noted as:

$$\zeta(f, q, \pi_x) = \sum_{z,z' \in Z} \pi_x(z)(f(z') - q(z'))^2$$

Here z' are samples drawn from non-zero elements of x uniformly at random. The x should be normalized (here, z' represents the normalized z). After that, all z's are recovered to regular dimensions and passed through the original model to get its labels. We are doing this to keep the local behavior of f and keep the explanation model agnostic. We use this newly created data set to fit the g. And the formula above is used as its loss. π here is a measure of distance, and it is computed using the following formula:

$$\pi_x(z) = exp(-D(x, z)^2/\sigma^2)$$

Here, D can be different for different cases, starting with euclidean distance for tabular data, cosine distance for text, and L2-norm for images.

Knowing all these, we now can express the explanation as to the following formula:

$$\varepsilon(x) = argmin_{g \in G} \zeta(f, g, \pi_x) + \Omega(g)$$

Now, the problem of searching for an explanation becomes an optimization problem, where we try to lower the sum above. The first term measures the local fidelity, while the second measures interpretability. In such a way, we are getting an interpretable model that locally is faithful

to the predictions.

Now, let's see how this process looks in code. To interpret models with LIME, we will need first to install a package - lime, using the following command:

```
pip install lime
```

Suppose we have an uninterpretable trained model on the Heart Disease UCI Data set, in our case, the GaussianNB. First, we must create an explainer:

```
In:
import lime
explainer = lime.lime_tabular.LimeTabularExplainer(X_train,
feature_names = list(df.iloc[:, :-1].columns),
class_names = ['target'],
categorical_features = ['sex', 'cp', 'fbs', 'restecg', '
   exang',
'slope', 'ca', 'thal'],
verbose=True, mode='regression')
```

Now we can create an explanation using a sample and a prediction function (in the case of sklearn models, it is model.predict):

```
In:
exp = explainer.explain_instance(X_test[2], gauss.predict)
```

Also, we can plot the explanation:

```
In:
exp.show_in_notebook(show_table=True, show_all=False)

Out:
Figure 98
```

You will get an output like this:

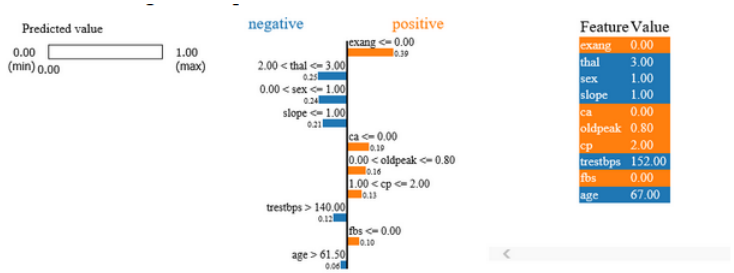


Figure 108: The model interpretation created by LIME.

The first column will show you the predicted value. The second one represents the influence of every feature on the output, especially which class it influenced the most. And the last column represents the feature values.

The middle column is the most informative. The plot above says that the 'exang' column significantly influences the prediction and brings the sample to the positive class.

One more interesting thing is even if the most considerable influence has the 'exang' feature, many minor effects bring the prediction to the negative (0) class. Now let's see what if we changed the 'thal' value to 2, what would happen.

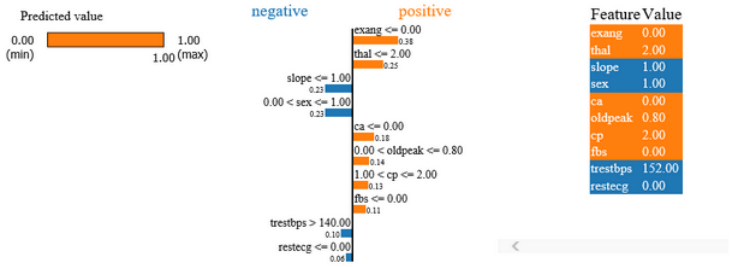


Figure 109: The model interpretation created by LIME after changing one feature value

It radically changed the prediction. The lesson that we can take from this example is that some models, even if they perform well, if even the most minor changes in data, can radically change the prediction. Also, if the prediction is based on some features with a smaller influence, and we will change one value, the result will radically change. This shows how unstable some models are and what features are better to erase from the model because they are too noisy.

Unfortunately, we can save the plot above only as an HTML file. This can be done using the following function:

```
exp.save_to_file('line.html')
```

20.3.1 SHAP interpretability.

SHAP (or **SHapley Additive exPlanations**) is another model agnostic technique for a model explanation. It also helps you to understand why a model made a specific decision.

SHAP values are based on Shapley values, a concept from game theory. A game typically needs two things: a game and some players. Connecting this to Machine Learning explainability:

- The "game" reproduces the outcome of the model.
- The "players" are the features included in the model.

Shapley quantifies each player's contribution to the game outcome, while SHAP quantifies each feature's contribution to the prediction made by the model. Let us think for a moment about this. What is the best way to see the influence of a feature on the model output? We can make two models, one with this feature present and one where it is absent (or set to 0). Then we can just compare these two outputs. However, what will we do when we have many more features and want to explain them?

Power set.

Suppose we have a set of features F that we want to explain. Using the intuition above, we need to find all possible combinations of features. Shapley values are based on the idea that the outcome of each possible combination (or coalition) of players should be considered to determine the importance of a single player. In math, this is called a "power set", and it can be represented as a graph:

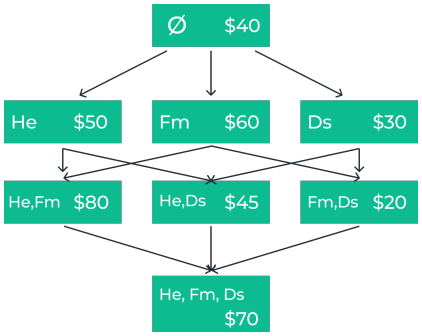


Figure 110: The graphical representation of a power set.

Suppose we have in our feature set three features - has or not a Higher education (He), has or not a Family (Fm), and has or not a Disease (Ds). We want to predict the person's income (figures are put randomly). The upper note represents the prediction of the node without using any feature - the mean of the income. The rest of the nodes represent a model with the predicted income using some features for a specific sample.

Edges on this graph represent the Marginal Contribution (MC) brought by a feature to a model. Let's compute the MC of the 'He' feature from

the upper node to the leftmost node from the second row:

$$MC_{H_e,\{H_e\}}(x_0) = Predict_{\{H_e\}}(x_0) - Predict_{\oslash}(x_0) = 50\$ - 40\$ = 10\$$$

Of course, to get the overall effect of the 'He' feature on the final model, we must compute the marginal distribution of 'He' on all models where 'He' is present. In our graph, it means to consider all edges connecting two nodes such that:

- the upper one does not contain 'He', and
- the bottom one contains

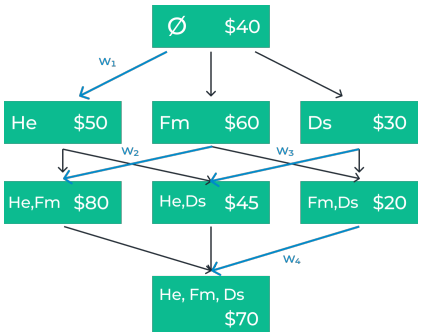


Figure 111: The graphical representation of the power set.

$SHAP_{He}(x_0) = w_1 \times MC_{He,\{He\}}(x_0) + w_2 \times MC_{He,\{He,Fm\}}(x_0) + w_3 \times MC_{He,\{He,Ds\}}(x_0) + w_4 \times MC_{He,\{He,Fm,Ds\}}(x_0)$

where $w_1 + w_2 + w_3 + w_4 = 1$

Now, there is a question - How are these weights computed? The idea is:

- The sum of all weights on the same row should be equal to the sum of all the weights on any other row, in our example: $w_1 = w_2 + w_3 = w_4$
- All the edges on the same row should have the same weights: $w_2 = w_3$ Keeping in mind that they all should sum up to 1, we get:
- $w_1 = \frac{1}{3}$
- $w_2 = \frac{1}{6}$
- $w_3 = \frac{1}{6}$

- $w_4 = \frac{1}{3}$

The weight of an edge is the inverse of the total number of edges in the same row. And even more, there is a formula to calculate it - and it uses the binomial coefficient. Let's remember how it is calculated, F is the total number of features, and f is the number of features chosen.

$$\binom{F}{f} = \frac{F!}{f! \times (F-f)!}$$

Now, we can compute the weight of the edge as follows:

$$f \times \binom{F}{f}$$

Now the formula for computing the SHAP value of the 'He' feature is:

$$SHAP_{He}(x_0) = [1 \times \binom{2}{1}]^{-1} \times MC_{He,\{He\}}(x_0) + [2 \times \binom{3}{2}]^{-1} \times MC_{He,\{He,Fm\}}(x_0) + [2 \times \binom{3}{2}]^{-1} \times MC_{He,\{He,Ds\}}(x_0) + [3 \times \binom{3}{3}]^{-1} \times MC_{He,\{He,DsFm\}}(x_0)$$

To generalize, we can use this general SHAP formula:

$$SHAP_{feature}(x) = \sum_{set:feature \in set} [|set| \times \binom{F}{|set|}]^{-1} \times [Predict_{set}(x) - Predict_{set/feature}(x)]$$

SHAP has a significant property: summing the SHAP values of each feature of a given observation yields the difference between the prediction of the model and the null model.

However, as you may already understand SHAP formula requires training 2^F models. This isn't always feasible. That's why the library we will look at uses some approximations and samplings, which I will let you do in your research.

Now, let's see how we can use SHAP values in code. To compute the SHAP values, we will need first to install a package - shap using the following command:

```
pip install shap
```

As in the case of LIME, suppose we have a trained model on the Heart Disease UCI Data Set, in our case, the Random Forest. First, we must import the shap library. Also, we have an additional step, some

plots in this library aren't created using python libraries but Javascript instead, so we will need to init the Javascript driver.

```
In:
import shap
shap.initjs()
```

Now we can create an explainer:

```
In:
explainer = shap.Explainer(forest.predict, X_train,
feature_names = list(df.columns)[:-1])
```

The explainer will take a prediction function (in the case of sklearn models, it is the model.predict), a 2-d NumPy array, the training set, and the feature names as a list.

```
In:
shap_values = explainer(X_test)
```

This may take a while, depending on how big your matrix is.

The first plot that we will see is the barplot. It plots the absolute value of the shap value of every feature, the code and the plot are listed below:

```
In:
shap.plots.bar(shap_values, max_display=14)

Out:
Figure 102
```

On the plot above, we can see that the highest SHAP value (and, respectively, the highest influence for the prediction) has the 'cp' feature. Also, we can see that the 'fbs' feature does not influence prediction. The following plot will show how every feature brings a certain prediction to one or another class. This type of plot is named the waterfall plot, and the next function can create it:

```
In:
shap.plots.waterfall(shap_values[0], max_display=14)

Out:
Figure 103-104
```

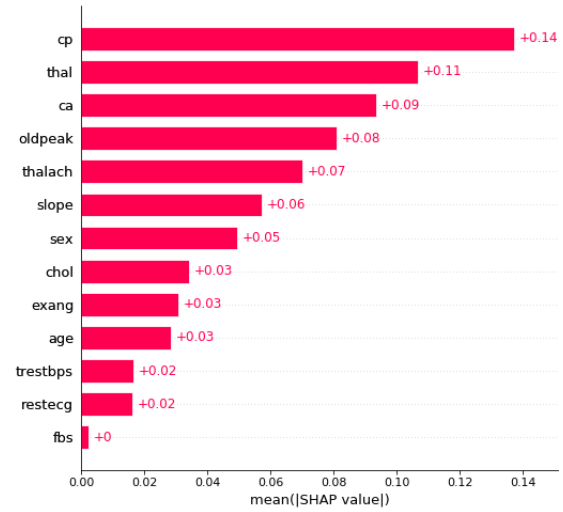


Figure 112: SHAP barplot.

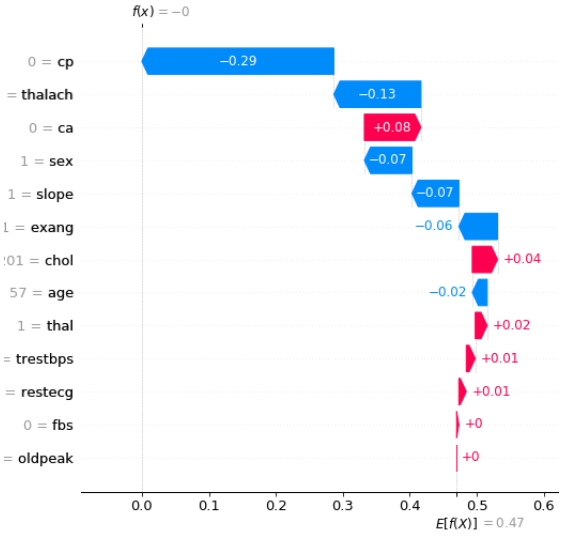


Figure 113: SHAP waterfall plot.

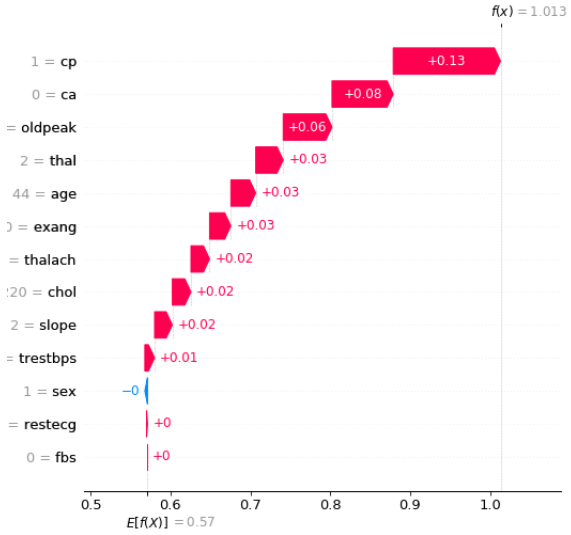


Figure 114: SHAP waterfall plot.

On the plots above, we can see how features bring the prediction from the base case to one or another prediction. This method requires only one sample. Also, to save a plot as an image with the plt.savefig function, we must set the parameter show=False in these functions.

Another interesting plot is the beeswarm plot. The following line of code can plot it:

```
In:
shap.plots.beeswarm(shap_values, max_display=14)

Out:
Figure 105
```

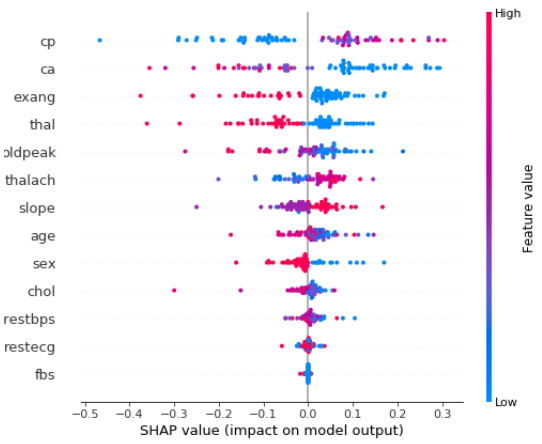


Figure 115: SHAP beeswarm plot.

In this plot, the features are ranked in descending order by their importance. Also, the legend from the right shows whether the value is associated with a higher or lower value of the feature value in this instance, represented in the color of the sample.

The last and most interesting plot is the force plot. To make it, you should use the following function:

```
In:
shap.force_plot(shap_values[0], X_test[0])

Out:
Figure 106
```

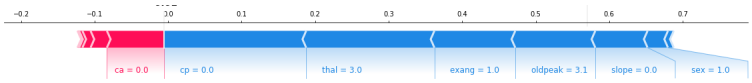


Figure 116: SHAP force plot.

To be able to save it, you should also set the matplotlib=True. However, the result isn't the best because it is created with javascript.

This plot shows how and how much each feature is moving the prediction apart from the base case. We can see that the most considerable influence on the prediction has the 'slope' feature.

20.4 Conclusion.

In this chapter, we saw why model interpretability is so important, how some models are self-explanatory, like linear and tree-based models, and how we can explain the decisions of other, more complex models. Besides that, we met two formidable technologies for model-agnostic interpretation - LIME and SHAP that can be used for model interpretability.

21 Streamlit

In machine learning, it is often hard to present the results of your model. There is always the terminal where we can get the output. However, this interface is not the best for daily users who have no idea what those numbers mean, and creating a website or an app from zero that will connect and work properly with your model could take ages, especially if you have no experience. Nevertheless, it is overwhelming if you have made some changes to your model and want to change the interface elements. However, do not worry; for this type of problem, people invented a fantastic solution known as streamlit.

21.1 What is streamlit?

Streamlit is an open source app framework in **Python** language. It helps us create web apps for data science and machine learning in a short time. It is compatible with major **Python** libraries such as **scikit-learn**, **Keras**, **PyTorch**, **SymPy**, **NumPy**, **pandas**, **Matplotlib** etc. With Streamlit, no callbacks are needed since widgets are treated as variables. Data caching simplifies and speeds up computation pipelines.

21.2 The theory behind the product

Now without further ado, let us dive deep into the code and the tools that we have available because that is what you are here for:

```
pip install streamlit
```

Great! Please run this command in your terminal to check if everything is fine and to make sure you can start working.

```
streamlit hello
```

A new tab should open in your default browser, and you should be able to see a welcoming message from streamlit.

Please create a new Python file in your directory and open it.

First import the library with:

```
import streamlit as st
```

Now let's proceed and write a title.

```
st.title("This is an example created by me")
```

Save your file, and open the command prompt in the directory in which you have the created file and run this command.

```
streamlit run the_name_of_your_file.py
```

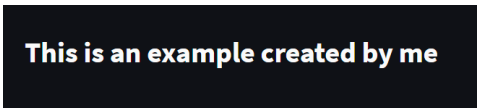


Figure 117: Title example.

21.3 How streamlit works?

So, to make things clear, here is how streamlit works in simple words: your code goes in the python file with all the elements and functions you have, and streamlit shows just the commands that you write using it. So, for example, you could train an entire model, make predictions, and print the accuracy on the platform. Remember, to run streamlit, you must run the command above in the cmd.

Now let us diversify your knowledge with some useful streamlit functions to know.

21.4 Main commands

There are a lot of possible combinations and functions that can be used with streamlit, but these are the main ones that might be useful in your work:

These are just a few of the multitude of particular functions of the streamlit library, so please feel free to take a look at their library docs. It is well-represented and has different examples for every particular function. This is the website: https://docs.streamlit.io/

To display almost anything you want in a fast way, you can use the st.write command:

```
"""st.write, as it says, writes arguments to the app,
starting from text and ending with whole dataframes.
It's similar to a print function."""
st.write("Hello **world**!")

"""First import a dataset of your own into the variable
dataframe to get some results, and then execute the
command below."""
st.write(dataframe)
```

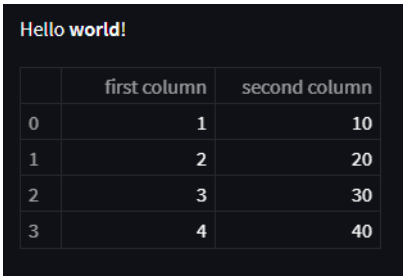


Figure 118: Hello world example.

Text elements are an essential part of the streamlit library, and there are many ways of printing or displaying text, so these are some handy methods that could help you do that quickly and intuitively:

```
"""To write text in the markdown language, the command
st.markdown can be used."""
st.markdown("#### Hey **there** ")

"""For writing titles on your pages, use the simple
st.title command"""
st.title("This is a title")

"""For creating a header to your page use st.header"""
st.header("This is a header")

"""For creating a code section on your page use
st.code command"""
st.code("a = 42")

"""And for writing latex style articles, just use
the simple st.latex command."""
st.latex("\int m y^2 \,dx")
```

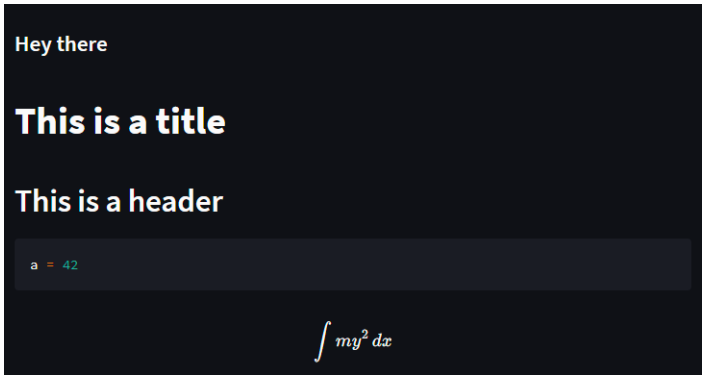


Figure 119: Texts typing examples.

Data display elements also have multiple ways of representation in the streamlit library. The ones below are just a few, but it is incredible how the same data frame can be represented in many ways.

```
"""To display your dataframes, just assign them to
the st.dataframe function. """
st.dataframe(your_data_frame)
```

	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7
0	0.9544	-2.0842	-0.8503	-1.4802	0.0613	0.9007	0.3716	1.2400
1	0.7611	2.9288	-0.4820	-0.0322	-0.0327	0.6974	-0.1142	1.0954
2	-1.4204	0.3314	1.0169	0.4932	1.7708	1.8495	1.2680	0.8942
3	1.4224	0.0414	0.5834	0.3946	-0.2511	-0.6247	1.9456	-0.8153
4	-0.8297	1.6550	0.3983	0.4556	0.1893	0.2308	-0.1911	1.0816
5	-1.4317	-0.2865	0.2217	-0.3220	-0.3296	-0.2431	1.0702	-1.1573
6	0.1130	1.0554	-0.9743	-0.7913	-0.9322	0.1008	0.3467	1.8034
7	1.3467	-1.1645	-2.7837	1.1014	-0.2096	0.5555	2.0224	1.4358
8	-0.6592	2.8996	1.1411	1.1570	-0.7486	0.8060	-0.7385	-0.5813
9	-0.9524	-0.8175	-0.2976	-1.0600	0.1752	-1.4467	0.3639	1.2420

Figure 120: Dataframe example.

```
"""To display some information in table format or
even dataframes, use st.table"""
st.table(your_table)
```

	col 0	col 1	col 2	col 3	col 4
0	-0.4444	-0.7433	0.9514	-1.4017	0.1734
1	2.2564	-0.1954	0.0525	-1.0796	0.5320
2	0.3095	1.0511	-2.1370	0.6377	0.7999
3	-0.2168	1.4075	-0.5302	-0.9794	-0.8787
4	0.1694	0.7843	-0.1644	0.0458	0.6214

Figure 121: Table example.

```
"""To provide some metric representation of your
data, you can try the st.metric command."""
st.metric("Some metrics", 42 , 10)
```

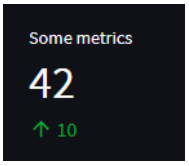


Figure 122: Metrics example.

```
"""To display json files, use the st.json command. """
st.json(your_json)
```

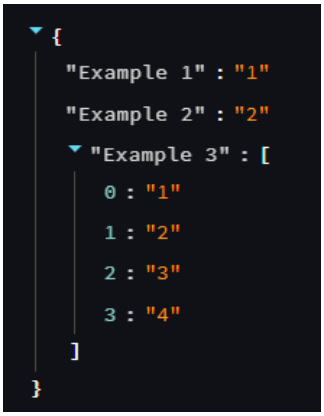


Figure 123: Json example.

Chart elements are a must-have for stunning projects because they are a tool for representing the diversity and beauty of the data. Here are some essential functions for the display of streamlit, but there are many more out there.

```
"""While being less customizable, this method of
displaying a line chart is easier to use for many
"just plot this" scenarios."""
st.line_chart(dataframe)
```

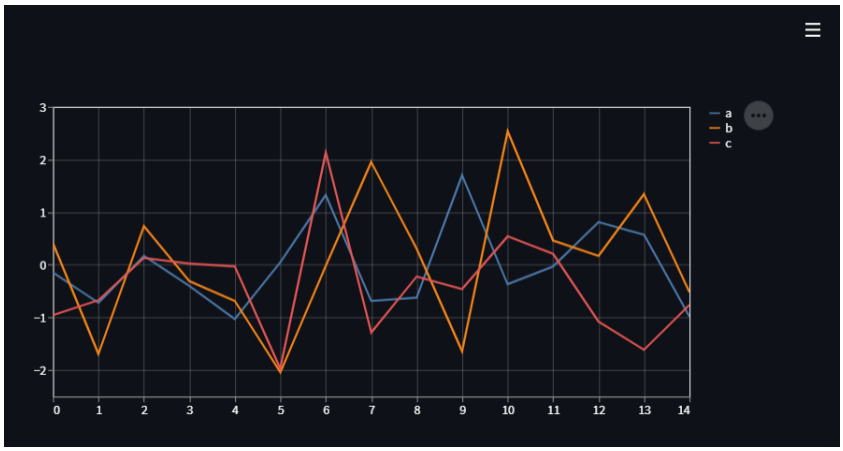


Figure 124: Line chart example.

```
"""To quickly create scatterplot charts on top of
a map, with auto-centering and auto-zoom, st.map
function can be really helpful."""
data = pd.DataFrame(
     np.random.randn(1000, 2) / [50, 50] + [37.76, -122.4],
     columns=['lat', 'lon'])

st.map(data)

"""To display a matplotlib.pyplot figure the
st.pyplot function can be used"""
arr = np.random.normal(1, 1, size=100)
fig, ax = plt.subplots()
ax.hist(arr, bins=20)

st.pyplot(fig)
```

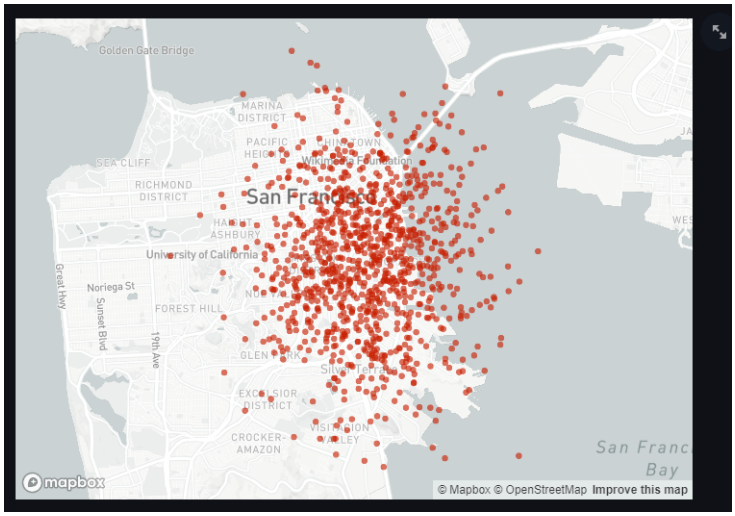


Figure 125: Map example.

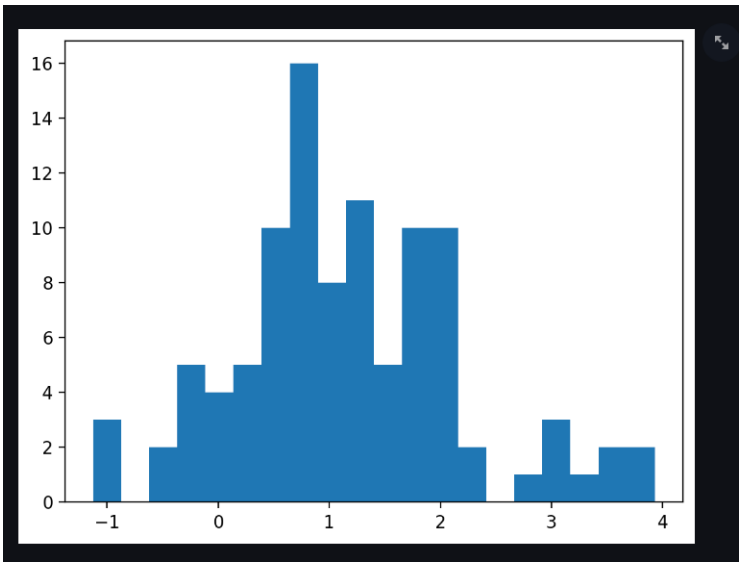


Figure 126: Pyplot example.

Communication between the user and the app is necessary for a good tool. In this case, input widgets can help you diversify your app and make real-time changes to your project.

```
"""To display a button widget, you can simply use
the st.button function that returns the value
True if it was pressed; otherwise, the
value False"""
clicked = st.button("This is a button")
```

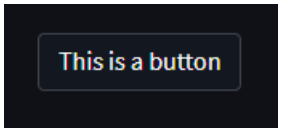


Figure 127: Button example.

```
"""In moments when you would like the user
to download some data or files directly from
your app, use the st.download_button"""
st.download_button("Download", file)
```

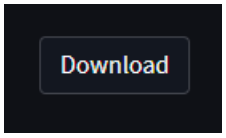


Figure 128: Download button example.

```
"""For simple checkboxes, use the st.checkboxfunction,
which returns True if it was checked
and False if it was not."""
selected = st.checkbox("Turn on live mode")
```

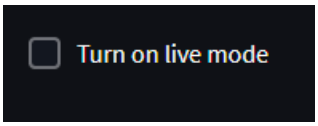


Figure 129: Checkbox example.

```
"""The st.radio displays a radio button widget
and returns the selected option."""
choice = st.radio("Pick something", ["Linear regression",
                  "Logistic Regression"])
```

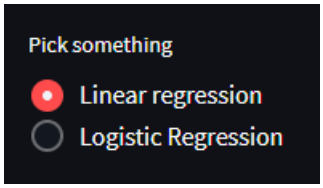


Figure 130: Radion button example.

```
"""The st.camera_input returns pictures from
the user's webcam."""
image = st.camera_input("Cheese")
```

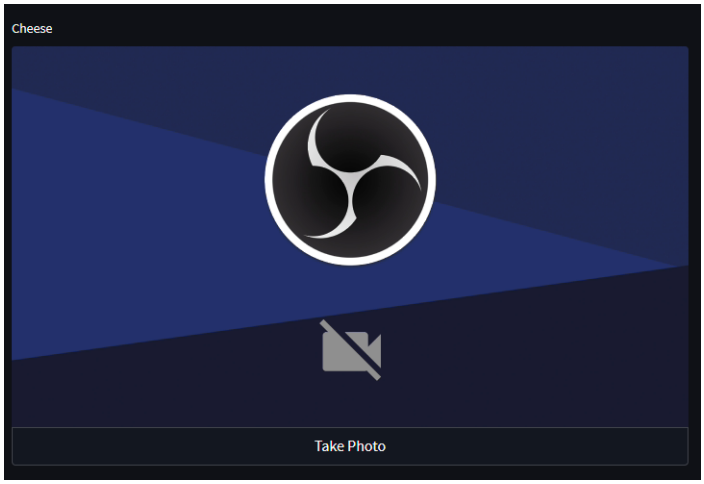


Figure 131: Camera example.

```
"""The st.file_uploader creates a widget
that allows you to assign any file from
your computer to your variable."""
data = st.file_uploader("Upload CSV")
```

Figure 132: Uploading files example.

Remember that the uploaded files are limited to 200 MB, but you can configure this using the server.maxUploadSize config option. For more details, feel free to check the documentation.

Presenting the resulting images to the user is necessary for computer vision projects. Therefore, the functions below allow you to do just that.

```
"""The st.image function can display an
image or a list of images."""
from PIL import Image
image = Image.open('sigmoid.png')
st.image(image, caption='Sigmoid logo')
```

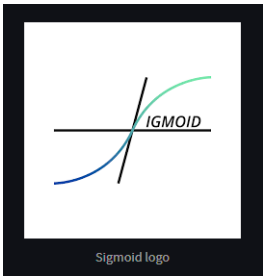


Figure 133: Displaying image example.

```
"""The st.audio component displays an audio player."""
audio_file = open('audio.mp3', 'rb')
audio_bytes = audio_file.read()
st.audio(audio_bytes, format='audio/mp3')
```

Figure 134: Audio files example.

```
"""And to display a video player, simply
use the st.video"""
st.video("https://www.youtube.com/watch?v=BXn-BURK920&
   ab_channel=Sigmoid")
```

Figure 135: Video files example.

Containers and columns are great tools for making your page look structured and clean. Here are some of the main functions to use in the process.

```
"""To allow your user to focus on the
content in your app, you can pass
different elements to st.sidebar
to pin them to the left."""
st.sidebar.write("This is some test on the sidebar")
```

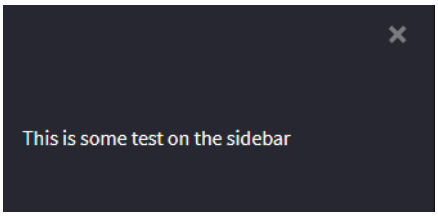


Figure 136: Sidebar example.

```
"""To insert containers laid out as
side-by-side columns, use the
st.columns method."""
col1, col2 = st.columns(2)
col1.write("This is the first column")
col2.write("This is the second column")
```

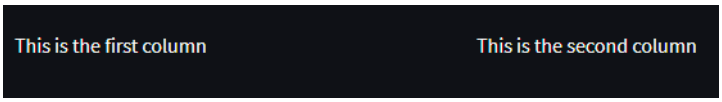


Figure 137: Columns example.

```
"""The st.expander function inserts a container into your
   app that can be used to hold multiple elements and can
   be expanded or collapsed by the user. When collapsed,
   all that is visible is the provided label."""
with st.expander("Check to see more"):
        st.write("Surprise")
```

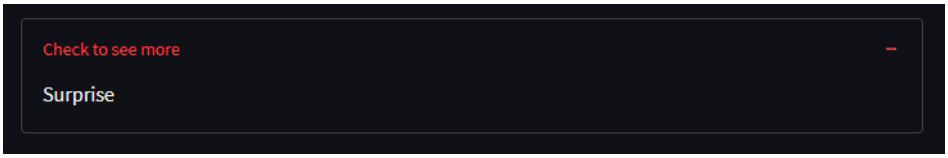


Figure 138: Expander example.

Last but not least, it can be helpful from time to time to get to the root of your app and make some fundamental changes or even to check some functions that you forgot about, for which you have the utility functions.

```
"""If you want to get to the root of your app, you should
   know
about the st.set_page_config function that allows you to
   configure the default
settings of the page."""
st.set_page_config( page_title="Sigmoid", page_icon=":
   turtle:" )
"""Keep in mind that this must be the first Streamlit
   command used
in your app, and must only be set once."""
```

Figure 139: Page settings configuration example.

```
"""To easily get access to the doc string of
different objects, use st.help"""
st.help(st.write)
st.help(pd.DataFrame)
```

21.5 Conclusion

After taking everything into account, streamlit is a great framework that could help you launch your artificial intelligence-related projects quickly by creating a user-friendly look for your app in just a few lines of code. There is much more to discover related to what streamlit can offer you, so remember that 6 hours of debugging can save you 15 minutes of reading the library from https://docs.streamlit.io/, where you can also find new features.

22 Memory optimization. Dask

It is quite obvious that building a good machine learning model presumes working with data. The previous chapters discussed several tools for operating with data, such as **Pandas** or **NumPy**; popular libraries among the data science community, with high performance and easy-to-use inbuilt data structures and functions. However, these libraries work well with the in-memory datasets (data that fits RAM). When it comes to handling large-size datasets or out-of-memory datasets, it is quite difficult to operate, as it might fail to cause memory issues.

There are several methods dealing with the mentioned issue. One of them is to reduce the used memory by modifying the data types of the columns in the analyzed dataset if possible. Another method, which will be discussed in the current chapter, is using the framework of the **Dask** framework.

22.1 What is Dask?

Dask is an open-source flexible library for parallel computing in **Python**. It is aimed to help people to improve code performance and memory usage without a lot of code modification, as it is built in coordination with libraries such as **NumPy**, **Scikit-learn**, **Pandas** and others.

So far, it may be unclear what parallel computing means. Parallel processing, in broad terms, refers to executing multiple tasks in parallel, simultaneously using multiple processors in the same machine. This system can carry out simultaneous data processing to achieve a faster execution time. As follows, Dasks splits the data by indices into clusters and keeps it on a single machine, each segment of data triggering many operations on the constituent data.

Dask itself is composed of two main parts: dynamic task scheduling and "Big Data" collections. The last one presents the main interest, especially parallel arrays, data frames, and lists. Commonly, Dask is necessary to be used in case of:

- manipulating large data, even when these do not fit in memory;
- accelerating long computations by using many cores;
- distributed computing on large data with standard operations.

To install Dask, it is sufficient to run the following command in terminal or cmd:

```
pip install dask
```

or

```
pip install dask[complete]
```

– in case you want to install all dependencies or specify the desired dependency in the square brackets.

22.2 Dask Array

Dask Array divides arrays into many small pieces, called **chunks**, each being small enough to fit into memory. Unlike NumPy, the operations in the Dask Array may be called "lazy". Operations queue up a series of tasks mapped over blocks, and no computation is performed until values are asked to be computed. Data is loaded into memory at that point, and computation proceeds in a streaming fashion, block-by-block.

Dask Array coordinates many NumPy or NumPy-like arrays (like CuPy, Sparse, etc.), arranging them into a grid, as shown in Figure 1. Dask Arrays are generally used to implement a subset of NumPy ndarray interface using blocked algorithms, cutting up large arrays into smaller subarrays, calling and grouping them in chunks. It allows computations on arrays larger than memory using all of the cores of the device by performing many smaller computations instead of one extensive computation.

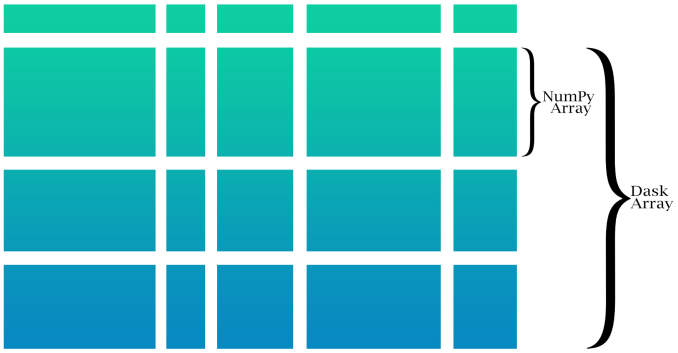


Figure 140: The structure of a Dask Array

A Dask Array may be created similar to a simple NumPy array. Let us see an example of generating a matrix using Dask:

In:

```
# Importing libraries
```

```
import dask.array as da
# Creating a Dask Array
dask_matrix = da.random.random((10000, 10000),
                               chunks=(1000, 1000))
dask_matrix
```

As can be observed in Figure 2, a matrix of the size 10000 x 10000 was created, where each chunk is of the size 1000 x 1000 and of the type NumPy array. Chunks can be made in any size that fits into the original matrix, even asymmetric ones. The only thing that is needed is to specify the desired dimensions of the chunks in the argument.

To access the desired values, it is necessary to add the compute() function after specifying the aimed data.

```
In:
# Accessing the first row of matrix
dask_matrix[0].compute()

Out:
Figure 134
```

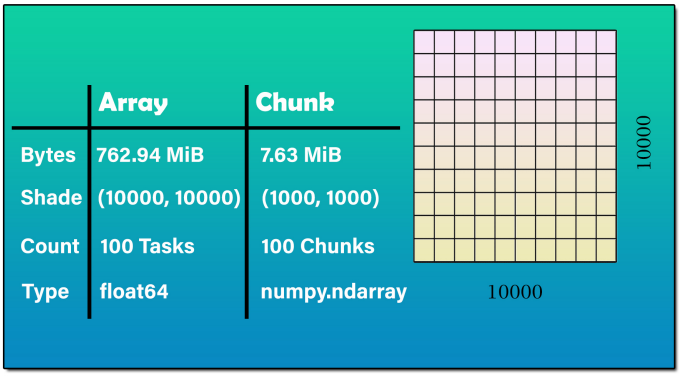


Figure 141: General structure of the generated Dask Matrix

Dask Arrays can be obtained from NumPy arrays, as well. In this case, it is also possible to specify the dimensions of the chunks. To do this, just run the code of the following structure:

```
# Importing the libraries
import numpy as np
# Setting the random seed
np.random.seed(42)
# Generating a vector of size 100
```

```
vector = np.random.randint(10000, size = 100)
tensor = np.random.randint(10000, size = (25, 25, 25))
# Creating a Dask Array with 25 chunks
vector_dask = da.from_array(vector, chunks = 25)
tensor_dask = da.from_array(tensor, chunks = 5)
```

Nonetheless, it is possible to perform NumPy calculations for the Dask Arrays, as well. In this case, Dask array objects are lazily evaluated. Operations like *mean()* or *sum()*, build up a graph of blocked tasks to execute. In the end, the *compute()* is called, which triggers all the necessary operations.

```
# Computing the mean of the vector
vector_dask.mean().compute()
```

22.3 Dask DataFrame

Although **Pandas** is a widely accepted framework, it has some limitations. Pandas can only handle datasets that fit into the device's main memory. However, there are cases when the dataset is too large to fit the memory, especially given the day-by-day growing real-life datasets. In these situations, Pandas is not the best choice for performing computations. In this case, **Dask Dataframe** is an optimal solution.

Dask Dataframes are big data frames that are internally composed of many Pandas data frames, as shown in Figure 3, similar to the relationship between NumPy arrays and Dask Arrays.

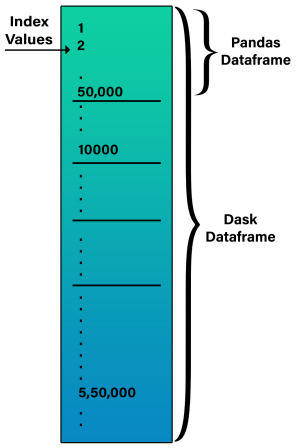


Figure 142: The structure of Dask Data Frame

As in the case of Dask Arrays, all operations performed on Dask Dataframes is lazy. Dask does not load data frames into the memory and keeps only reference and operations details on them. To trigger the calculation, it is necessary to call the *compute()* function. It will allow operations to perform on the desired data frame and execute in parallel clusters.

A Dask Dataframe application programming interface is a subset of the Pandas; it covers a well-used portion of the Pandas API. Therefore, a Dask Dataframe can be created similarly to a usual Pandas data frame. The Spanish Rail Ticket Pricing – Renfe dataset from Kaggle will be used for further computations.

```
# Importing the libraries
import dask.dataframe as dd
# Reading .csv as Dask Dataframe
ddf = dd.read_csv("data.csv")
```

Moreover, it is possible to set the size of each partition (blocksize) in the new created Dask Dataframe as follows:

```
# Creating data frame with blocks of 1000 MB
ddf = dd.read_csv("data.csv", blocksize = '1000MB')
```

On broad terms, the resulting output will be something like the table below.

```
Dask DataFrame Structure:
                id      company   ...    insert_date
npartitions=7
                int64   object    ...    object
                ...     ...       ...     ...
...             ...     ...       ...     ...
                ...     ...       ...     ...
                ...     ...       ...     ...

Dask Name: read-csv, 7 tasks
```

Another way of creating a Dask Dataframe is using an existing Pandas data frame. To make this possible, run the code of the following structure:

```
# Creating Dask dataframe from Pandas dataframe
ddf = dd.from_pandas(df, npartitions = 2)
```

where df is a defined Pandas data frame.

As it was specified previously, to trigger any computation on the Dask Dataframe, *compute()* function should be called. It may be applied on individual partitions, as well as on all of them.

```
# Computing heads for the data frame for each partition
for i in range(ddf.npartitions):
    print(ddf.partitions[i].compute().head())
```

The other functions specific to Pandas data frames can be applied in the same way, either for one specific partition or for a group. For example, to find out the description of each partition, it is sufficient to replace the compute() function from the previous sequence of code with the describe() function as follows:

```
# Getting the description for each partition in the
# Dask Dataframe
for i in range(ddf.npartitions):
    print(ddf.partitions[i].describe())
```

The previous lines of code gives an output of the following structure:

```
Dask DataFrame Structure:
                  id    duration    price    seats
npartitions=1
               float64  float64   float64  float64
                  ...      ...       ...      ...
Dask Name: describe-numeric, 43 tasks
Dask DataFrame Structure:
                  id    duration    price    seats
npartitions=1
               float64  float64   float64  float64
                  ...      ...       ...      ...
Dask Name: describe-numeric, 43 tasks
Dask DataFrame Structure:
                  id    duration    price    seats
...
Dask Name: describe-numeric, 43 tasks
Dask DataFrame Structure:
                  id    duration    price    seats
npartitions=1
               float64  float64   float64  float64
                  ...      ...       ...      ...
Dask Name: describe-numeric, 43 tasks
```

Despite being able to set the number of partitions for the created data frame or to specify the size of each partition, it is also possible to set the partitions to be distributed according to the indices of a specific column. For this purpose, the *set_index(column)* is used.

```
ddf _origin= ddf.set_index('origin')
```

As it may be observed, the line of code above gives the following outcome:

```
Dask DataFrame Structure:
                      id      ...     meta    insert_date
npartitions=27
ALBACETE              int64   ...     object  object
ALICANTE              ...     ...     ...     ...
...                   ...     ...     ...     ...
ZARAGOZA              ...     ...     ...     ...
ZARAGOZA              ...     ...     ...     ...
Dask Name: sort_index, 742 tasks
```

To see all the partitions of the data frame, it is sufficient to use the following construction:

```
ddf_origin.divisions
```

In general, *set_index(column)* method is a useful tool to sort the data by the chosen index column. It allows faster access, joins, group-apply operations, etc., on a specific data set. However, it is costly to do in parallel, so setting an index is a good practice when needed and not very frequently.

It should be remembered that, as stated previously, Dask DataFrame supports the majority of computations that can be done using Pandas. For example, it is possible to group elements by a value or even to specify new values for columns. Let us analyze a toy example done on a Pandas data frame. Consider the following line of codes:

```
In:
start = time.time()
df = pd.read_csv("data.csv")
# Grouping by origin and specify duration
df = df.groupby("origin", dropna=False,
                observed=True).agg({"duration": "mean"})
print('Computation time: ', time.time() - start)

Out:
```

```
Computation time:  131.4839825630188
```

It may be observed that the new data frame defines the average trip duration for each origin present in the dataset and follows the structure as in the table presented below.

```
origin  duration
ALBACETE        1.980307
ALICANTE        2.860042
BARCELONA       3.159051
CADIZ           4.534925
CASTELLO        3.345468
```

Also, notice that to have this computation done takes more than 2 minutes. Now, let us try to do the same thing using Dask Dataframe.

```
In:
start = time.time()
ddf = dd.read_csv("data.csv")
# Grouping by origin and specify duration
ddf = ddf.groupby("origin", dropna=False,
                  observed=True).agg({"duration": "mean"})

Out:
print('Computation time: ', time.time() - start)

Computation time:  0.04017996788024902
```

As it can be seen, the code remains almost the same. However, in this case, the output has the following form:

```
Dask DataFrame Structure:
                  duration
npartitions=1
                  float64
                  ...
Dask Name: aggregate-agg, 261 tasks
```

The created Dask Dataframe contains one partition, which has one specified column, "duration". Moreover, it might be noticed the fact that the operation computing itself will be done only after triggering the obtained data frame. Nonetheless, notice that the time needed to compute this dataset is much lesser than in the case of Pandas, as it could be said the device only keeps what should be done when needed, not do all operations immediately.

Further, let us see what kind of computations could also be performed. It may be performed for all manipulations used for Pandas data frames, such as selecting a specific column using loc or extracting specific values.

```
In:
# Computing the duration for the origin ALBACETE
ddf.loc['ALBACETE'].duration.compute()
# Extracting column which mean duration of trip is
# grater than 2.5 h
for i in range(ddf.npartitions):
        print(ddf.partitions[i][ddf.partitions[i].
              duration > 2.5].duration.compute().head())

Out:
origin
ALICANTE     2.860042
BARCELONA    3.159051
CADIZ        4.534925
CASTELLON    3.479095
GIRONA       4.228130
Name: duration, dtype: float64
```

Another interesting thing that should be mentioned is that Dask offers the opportunity of plotting the desired data. To have a graph representation, it is sufficient to add the *plot()* method after the description of the desired values to have a graph similar to the one in Figure 3.

```
In:
new_ddf = dd.from_pandas(df, npartitions = 7)
# Plotting the mean duration according to the origin
new_ddf.groupby(["origin"]).mean().compute().
                        plot(figsize=(10,6))

Out:
Figure 136
```

Nonetheless, it is possible to specify the type of graph to match the desired expectations, as it is shown in Figure 5.

```
In:
# Plotting the mean duration as bar diagram
# according to the origin
new_ddf.groupby(["origin"]).mean().compute().plot
```

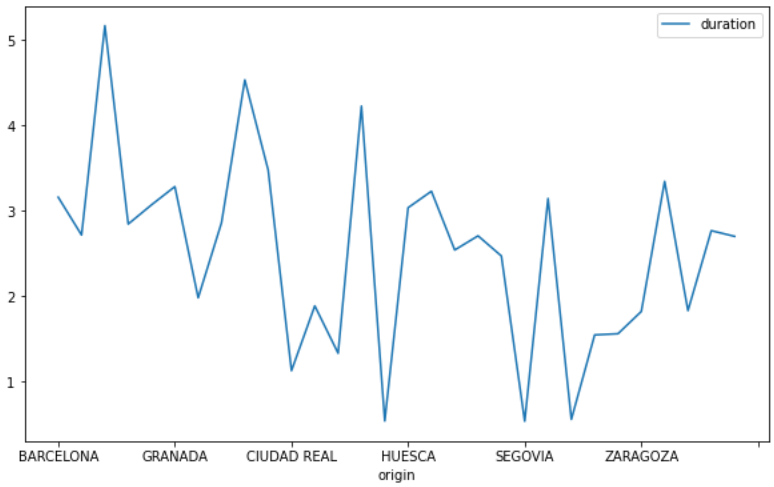


Figure 143: An example of a plot using Dask inbuilt function

```
(kind="bar",width=.8, figsize=(10,6))
```

```
Out:
Figure 137
```

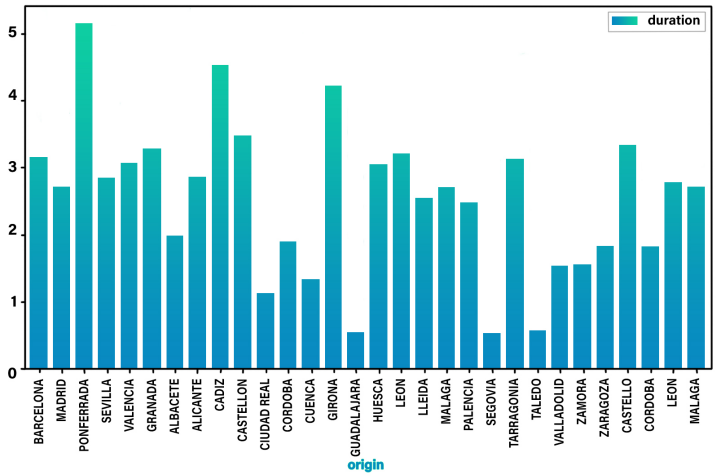


Figure 144: An example of plot bar diagram using Dask

It is necessary to mention that there exists a structure called Dask Bags, that implements operations like map, filter, groupby, and aggregations on collections of Python objects. It is used in the case of using different data structures for computations on unstructured or semi-structured data like text data, log, JSON records, or user-defined objects in Python. Although it is a helpful tool to know in specific cases, it is quite an advanced framework that will not be discussed during this chapter.

22.4 Conclusion

In this chapter, it was presented an introduction to a useful tool aimed to be a help in case of large datasets manipulation or big time cost computations, as it might appear in further data analyses. We hope that, during this chapter, you understand its usefulness and how it could be applied. However, it is necessary to mention that this tool is destined for working with large data. Therefore, in the case of datasets that fit the device memory, it is better to use already known libraries, such as Pandas or NumPy.