ULTIMATE GUIDE TO MACHINE LEARNING WITH PYTHON

Nikola Živković

RUBIK'S CODE
BUILDING SMART APPS

For online information and ordering of these and other Rubik's Code books, please visit www.rubikscode.net.

Editor: Ivana Milićev

Cover design and book layout: Slavica Aj

Preface **8**
Why does this book exist? 8
The Structure 11
Prerequisites 12
Datasets 12
Tools 14
Python and Libraries 14
Jupyter Notebook IDE 15

1. The Basics of Machine Learning **18**
1.1 Definitions 18
1.2 Problems Machine Learning Solves 20
1.3 Types of learning 21
1.2 The Anatomy of Machine Learning Process 22
1.5 Introduction to Gradient Descent 24
1.6 Three datasets 26
1.7 Hyperparameters 26
1.8 Learning Rate 26
1.9 Performance Metrics 27
1.9.1 Classification Metrics 27
1.9.1.1 Confusion Matrix 27
1.9.1.2 Accuracy 29
1.9.1.3 Precision 30
1.9.1.4 Recall 31
1.9.1.5 Precision-Recall Curve 32
1.9.1.6 F1 Score 32
1.9.1.7 Receiver Operating Characteristic (ROC) curve & Area Under the curve (AUC) 33
1.9.1.8 Log Loss 34
1.9.2 Regression Metrics 35
1.9.2.1 Mean Absolute Error – MAE 35
1.9.2.2 Mean Squared Error – MSE 36
1.9.2.3 Root Mean Square Error – RMSE 36
1.9.2.4 Root Mean Squared Logarithmic Error – RMSLE 37
1.9.2.5 R Squared 38

2. Machine Learning Algorithms **39**
2.1 Regression Algorithms 39
2.1.1 Simple Linear Regression 39
2.1.1.1 Simple Linear Regression Python Implementation 42
2.1.2 Multiple Linear Regression 43
2.1.2.1 Multiple Linear Regression Python Implementation 45
2.1.2.2 Using SciKit Learn 47

2.2 Classification Algorithms 50
2.2.1 Logistic Regression 50
2.2.1.1 Preparing data for Logistic Regression 51
2.2.1.1 Logistic Regression Python Implementation 53
2.2.1.2 Using SciKit Learn 55
2.2.2 K-Nearest Neighbours (KNN) 56
2.2.2.1 Preparing Data for KNN 57
2.2.2.2 KNN Python Implementation 58
2.2.2.3 Using SciKit Learn 61
2.2.3 Naive Bayes 63
2.2.3.1 Preparing Data for Naive Bayes 64
2.2.3.2 Python Implementation 64
2.2.3.3 Using SciKit Learn 66
2.3 Regression and Classification Algorithms 67
2.3.1 SVM 67
2.3.1.1 Preparing Data for Classification 69
2.3.1.2 Python Implementation 71
2.3.1.3 Using SciKit Learn 73
2.3.1.3 Non-Linear Data 76
2.3.1.4 Preparing Data for Regression 80
2.3.1.5 Using SciKit Learn 81
2.3.2 Decision Trees 85
2.3.2.1 The Python Implementation 88
2.3.2.2 Using SciKit Learn 96
2.3.2.3 Decision Tree Regression 99
2.3.2.4 Using Sci-Kit Learn 100
2.4 Ensemble Learning and Random Forest 102
2.4.1 Prepare Data for Classification 102
2.4.3 Classification Python Implementation 103
2.4.4 Classification with Sci-Kit Learn 107
2.4.5 Prepare Data for Regression 108
2.4.6 Regression with Sci-Kit Learn 109
2.5 Clustering Algorithms 110
2.5.1 Preparing Data for Clustering 111
2.5.2 K-Means Clustering 112
2.5.2.1 Python Implementation 113
2.5.2.2 Using SciKit Learn 117
2.5.2.3 The Elbow Method 118
2.5.3 Agglomerative Clustering 121
2.5.3.1 Dendrogram 121
2.5.3.2 Using Sci-Kit Learn 123

2.5.4 DBSCAN 124
2.5.4.1 Using Sci-Kit Learn 125

3. Regularization 127
3.1 Preparing the Data 130
3.2 Ridge Regression 133
3.2.1 Using Sci-Kit Learn 134
3.3 Lasso Regression 136
3.3.1 Using Sci-Kit Learn 136
3.4 Elastic Net 138
3.4.1 Using Sci-Kit Learn 139
3.5 Early Stopping 140

4. Optimization 142
4.1 Cross-Entropy 143
4.2 KL Divergence 144
4.3 Optimization Algorithms 145
4.3.1 Gradient Descent and its Variations 145
4.3.1.1 Prepare Data 148
4.3.1.2 Python Implementation 149
4.3.2 Batch Gradient Descent 152
4.3.2.1 Python Implementation 153
4.3.3 Stochastic Gradient Descent 156
4.3.3.1 Python Implementation 156
4.3.3.2 Using Sci-Kit Learn 159
4.3.4 Momentum Optimizer 161
4.3.4.1 Python Implementation 161
4.3.5 Nesterov Accelerated Gradient 165
4.3.5.1 Python Implementation 166
4.3.6 AdaGrad 169
4.3.6.1 Python Implementation 170
4.3.7 RMSProp 174
4.3.7.1 Python Implementation 175
4.3.8 Adam 178
4.3.8.1 Python Implementation 179
4.4 Hyperparameter tuning 183
4.4.1 Preparing the Data 183
4.4.2 Grid Search 184
4.4.3 Random Search 187
4.4.4 Bayesian Optimization 190
4.4.5 Alternatives 193

5. Deploying Machine Learning Model 194

5.1 Rest API 195
5.2 Saving Model into a File 196
5.3 Deployment with Flask 197
5.3.1 Flask Basics 198
5.3.2 Putting it all together 199
5.4 Docker Basics 203
5.5 Deployment with Flask and Docker 205
5.6 Deployment with Fast API 208
5.6.1 FastAPI Basics 209
5.6.2 Putting it all together 211
5.6.2.1 Client Side 212
5.6.2.2 Model Trainer 214
5.6.2.2 Server Side 216
5.6.2.2.1 Data Contracts 216
5.6.2.2.2 Model Loader 216
5.6.2.2.3 REST API Module 218
5.6.2.2.4 Testing the server-side 220
5.6.3 Running it all together 222

6. Deep Learning 223
6.2 The biology behind Neural Networks 223
6.3 The Main Components and Concepts of Neural Networks 225
6.4 Activation Function 228
6.4.1 Perceptron 228
6.4.2 Sigmoid function 229
6.4.3 Hyperbolic Tangent Function 231
6.4.4 Rectifier Function 231
6.2 Intro to Pytorch 233
6.2.1 Installation 233
6.2.2 Tensors and Gradients 234
6.2.3 Creating and Initializing Tensors 234
6.2.4 Tensor Operations 237
6.2.5 Gradients 239
6.2.6 Linear Regression 240
6.3 Building Feed Forward Neural Networks 244
6.4 Convolutional Neural Networks 252
6.4.1 Convolutional Layer 254
6.4.2 Non-linearity 257
6.4.3 Pooling Layer 258
6.4.4 Flattening Layer 258
6.4.5 Fully-Connected Layer 259
6.4.6 The Architectures of Convolutional Neural Networks 259

6.5 Building the Convolution Neural Network 260
6.6 Deploying PyTorch Neural Networks 263
6.6.1 TorchServe Architecture 263
6.6.2 Installation 265
6.6.3 Saving Trained Model 265
6.6.4 Serving Model 266
6.6.5 TorchServe and Docker 267

7. Some Rules and Best Practices 269
7.1 Objective and Metrics Best Practices 269
7.1.1 Start with a Business Problem Statement and Objective 269
7.1.2 Gather Historical Data From Existing Systems 269
7.1.3 Use Simple Metric for First Objective 270
7.2 Infrastructure Best Practices 270
7.2.1 Infrastructure is Testable without Model 271
7.2.2. Deploy Model only After it Passes Sanity Checks 271
7.2.3 On-Premise or Cloud 271
7.2.4 Separate Services for Model Training and Model Serving 271
7.2.5 Use Containers and Kubernetes in Deployment 271
7.3 Data Best Practices 272
7.3.2 Data Quantity 272
7.3.1 Data Quality and Transformations 272
7.3.2 Document each Feature and Assign Owner 272
7.3.3 Plan to Launch and Iterate 272
7.4 Model Best Practices 273
7.4.1 Starting with an Interpretable Model 273
7.4.2. What to do if the model does not work? 273
7.4.2 Use Checkpoints 273
7.4.3 Performance over fancy metric 274
7.4.4 Production Data to Training Data 274
7.5 Code Best Practices 274
7.5.1 Write Clean Code 274
7.5.2 Write a lot of Tests 275

Final Words **276**

Preface

Are you interested in machine learning but get overwhelmed every time you start because there are so many things to cover? Do you feel there's loads to take in at once and it's just too much? If so, then you are in the right place. I myself was feeling the same way not too long ago. I wanted to learn all about this technology shaking up the industry. People were doing weird mixes of Picasso and Space Odyssey and generating screenplays with it, and all I wanted was to be a part of it. So badly! At that moment, I was just another software developer with several years of experience. Apart from that, I was younger and more arrogant and decided to learn everything about the topic in about a month. Just one more JavaScript framework - how hard can it be, right? Wrong. This was not just another language or programming principle. In fact, it took some time and a lot of effort before I grasped the basic concepts and really started to understand what was going on. Essentially, the problem was that my programming skills (although important) were insufficient to get me up to speed. The struggle was real.

As it turned out, my knowledge of Object-Oriented principles and functional programming was not enough to learn and understand everything. I dusted off old university books and re-utilized my knowledge of Math and Statistics. Apart from that, I had to learn a new programming language that was more suitable for this field. My main expertise back then was C# and JS, so I picked Python as my weapon of choice, as it has similarities with the languages I was comfortable with. In a nutshell, I had to fight multiple battles simultaneously just to kick off my journey, learn the concepts and finally be able to solve problems on my own. As I was studying, I took a substantial number of notes. They came in various forms. Sometimes they were confusing scribbles that ended up on my walls like weird modern-art drawings. Other times, they were mind maps, colorful and structured. Collages? You bet. However, there were times when the notebooks I used became small journals. Better ones became blog posts or conference talks. And sometimes, well, they became books. Like this one.

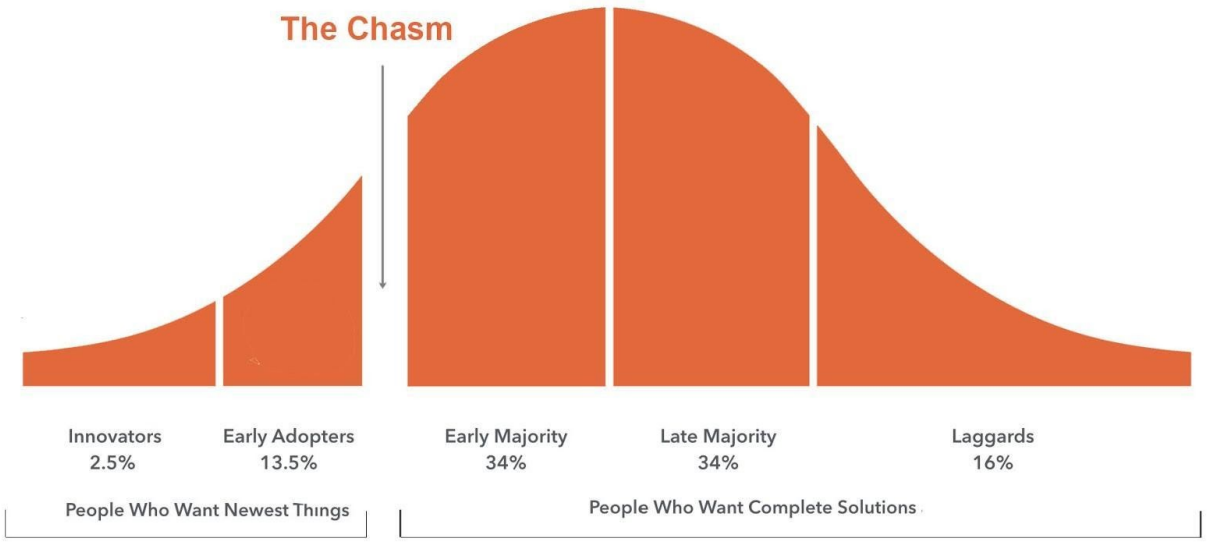
In this book, I documented my journey and created a systemized approach that can help you overcome obstacles while diving into the world of machine learning. The idea is to make this process as fun and understandable as possible.

Why does this book exist?

Machine learning, deep learning, and artificial intelligence are quite the buzzwords these days, aren't they? Wherever we go, we are bombarded with these terms. The mass media has even gone to the lengths of giving them "doomsday" scenarios. In fact, a number of people think that this pursuit of artificial intelligence will have rather hellish

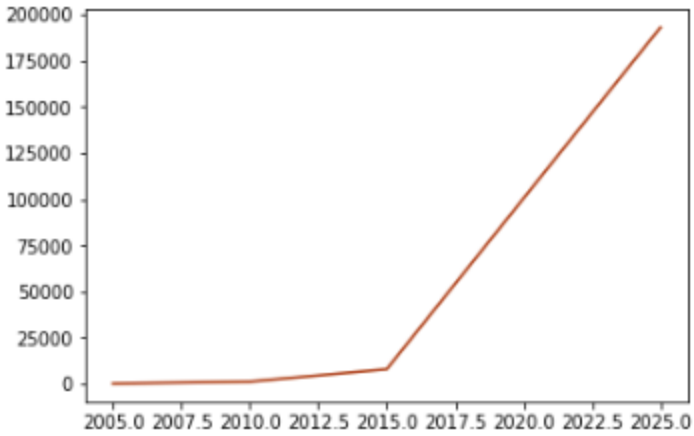
outcomes, but the experts in this niche beg to differ. We are here today to break the taboos and stigmas that cast a shadow over this field.

So, what is the motivation behind it all? The most convincing argument is that all these fields are crossing the chasm. Bridging the gap. We are out of the early adopters stage and more and more people find proper practical applications for these technologies. It is important to note that we are seeing an increasing number of practical use cases and applications for them. Also, we are trimming the areas in which, evidently, these solutions are not going to be useful. Keep in mind that the Early Majority occupies ~13% of the market, which is crucial. In short, these are no longer obscure sciences and you can create your machine learning models or neural networks on your home computer and use them to solve real-world scenarios.



The other reason why you should consider machine learning, deep learning, and AI, in general, is the fact that we are getting more data than ever. As humans, we cannot process all that data and make sense of it, but these solutions can. Statistics say that from the beginning of time up until 2005, humans produced 130 exabytes of data. Exabyte is a real word, by the way, I've checked. That is 10^18 bytes. Basically, if you scale up from terabyte, you get petabyte and when you scale from petabyte, you get - exabyte. To visualize that, if we cut down every tree in the Amazon forest (of course, I am not saying we should, I am just trying to prove a point… poorly?) and make paper out of it, and fill all those endless pages with some kind of information, we would create

one exabyte of data. This means that from the beginning of time up until 2005, humans created 130 Amazons of data. However, from that moment up until 2010, we had produced 1200 exabytes of data, and until 2015 we generated 7900. Predictions for the future are telling us that there will only be more data and that by 2025, we will have acquired 193 zetabytes (one magnitude over mentioned exabyte) of data. From my point of view, these are encouraging enough reasons to learn these skills, since we are constantly generating data. It is no wonder that we have more and more practical uses for it.



Another thing that I find intriguing is that the ideas behind machine learning, deep learning, and AI span way back into the past. You can observe the complete field as this weird steampunk science because techniques we use today are based on "ancient" knowledge. The idea of a learning machine can be traced back to the 1950s, to Turing's Learning Machine and Frank Rosenblatt's Perceptron. We will go into detail of the whole history of the field a bit later, but for now, let's say that we have acquired a lot of knowledge - 50 years of it, to be more exact.

Did you know that the first neural network was commercially used back in 1959 and that it is still in use today? Neither did I. That was the neural network MADELINE, still used to remove noise from landline conversations. With all these examples we may ask the question: "Why haven't these branches of engineering taken off earlier?" Well, as you will see, training machine learning models and neural networks take a lot of resources. Hardware back then simply didn't have enough processing power. Because of this, instead of taking the "artificial learning" approach, we took the computing approach.

However, things are different nowadays, with our nano processors and powerful GPUs and TPUs.

To sum up, we currently have a lot of knowledge that we can apply to data and we have the processing power to actually make it happen. It is crucial to understand that we are at the tipping point in history when we can make it or break it. That fact is scary and wonderful at the same time, but we must go onwards. To quote Stan Lee: "Excelsior!". Let's change the world!

The Structure

This book is divided into seven extensive chapters. Each chapter is dedicated to a certain sub-section of machine learning that will help you understand the process from building machine learning models to deploying them into production. The bonus material that goes with this book is extremely useful since it covers the basics of mathematics, Python programming language, and data analysis, heavily used in this book.

In the first chapter of this book, we cover some basic topics of machine learning. We define the problems machine learning is trying to solve, and we define the types of machine learning. Apart from that, we talk about the anatomy of machine learning algorithms and learn about performance metrics that we use throughout this book.

The second chapter of this book is reserved for machine learning algorithms you will face out there. We explore regression algorithms, classification algorithms, clustering algorithms, ensemble learning, and so on. Also, we implement these algorithms from scratch with Python and utilize existing solutions from Sci-Kit Learn Library. Personally, I found a lot of value in implementing machine learning from scratch because that proved to me that I understood how they work and how I can use them. That is why this is also included in this book.

The third part of this book covers regularization. We reveal what regularization is and how to use it. We learn about different options that we can use from Sci-Kit Learn Library.

The fourth chapter is all about optimization. We learn how the most popular optimization technique - stochastic gradient descent works, and we explore it's variations. Also, we learn about more advanced optimization techniques like AdaGrad and Adam. We implement those algorithms from scratch too and use existing implementations from Sci-Kit Learn. In this chapter, we learn how to optimize hyperparameters as well.

The fifth chapter of this book covers one burning topic - the deployment of machine learning algorithms. We learn about REST API and how to deploy machine learning algorithms with Flask and Docker. We learn how to do the same thing with more advanced technology - FastApi.

The sixth chapter covers deep learning and neural networks. In this chapter, we learn about motivations for this type of learning. We cover topics such as neurons, connections, activation functions, and convolutional neural networks. We learn how to use Pytorch for the implementation of these topics. We also learn how we can deploy neural networks with TorchServe.

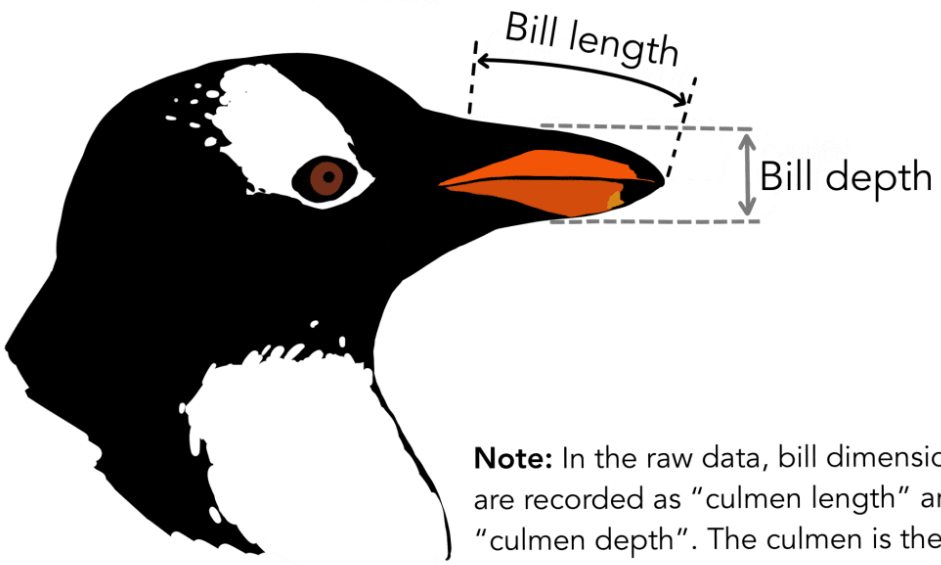
The seventh chapter is focused on some of the best practices when it comes to machine learning.

Prerequisites

Ok, before we proceed with this book, let's first focus on the datasets and libraries that are going to be used.

Datasets

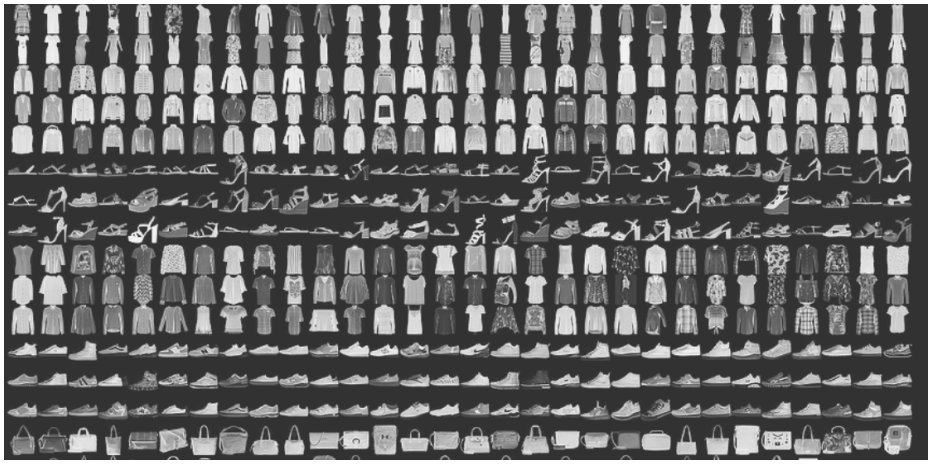
We use three datasets in this book for three different types of problems. Data that we use in this book for classification problems is from the **PalmerPenguins** Dataset. This dataset has been recently introduced as an alternative to the famous Iris dataset. It has been created by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER. You can obtain it via Kaggle, and it is available in the code that accompanies this book. The dataset is essentially composed of two datasets, each containing the data of 344 penguins. Just like in the Iris dataset, there are three different species of penguins coming from three islands in the Palmer Archipelago. Also, these datasets contain **culmen** dimensions for each species. The culmen is the upper ridge of a bird's bill. In the simplified penguin's data, culmen length and depth are renamed as variables *culmen_length_mm* and *culmen_depth_mm*.



Note: In the raw data, bill dimensions are recorded as "culmen length" and "culmen depth". The culmen is the dorsal ridge atop the bill.

Data that we use for the regression examples in this book is the famous Boston Housing Dataset. This dataset is composed of 14 features and contains information collected by the U.S Census Service concerning housing in the area of Boston, Massachusetts. It is a small dataset with only 506 samples.

Finally, the last dataset that we use in this book is the Fashion MNIST dataset. It is used in the final part of the book when we use neural networks for image classification. This dataset is a variation of the MNIST dataset, and it has the same structure as the MNIST dataset, i.e., it has a training set of 60,000 samples and a testing set of 10,000 clothes images. All images have been size-normalized and centered. The size of the images is also fixed to 28×28, so the preprocessing image data is minimized. Here is what that looks like:



Tools

Python and Libraries

While I was writing this book, it was predicted that *Python* will become the most popular programming language in 2021. If somebody had told me a couple of years ago that *JavaScript* would be taken off the throne by *Python*, I would hardly have believed it. One of the main reasons for *Python's popularity* is due to the rise of data science as a field in general. Since it provides a lot of useful packages, it has become a major language used in this field, along with a programming language called *R*. Here we use Python 3.7.6, installed using **Anaconda distribution**. The majority of the libraries in this book are already installed with this distribution, however, some are not. Make sure that you have these libraries installed:

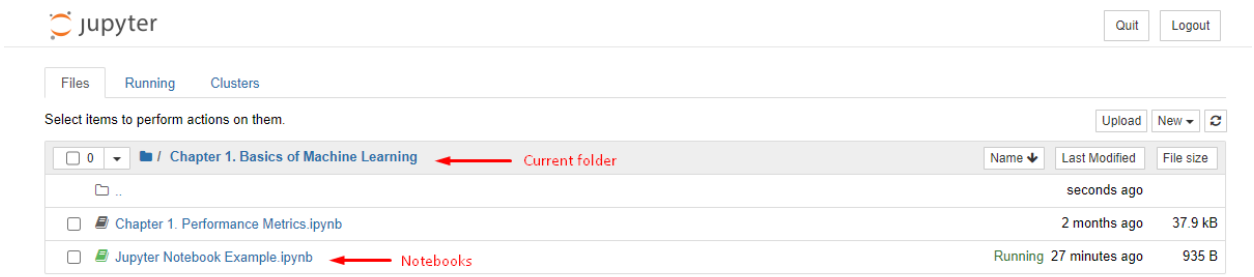
- **NumPy** – Follow **this guide** if you need help with installation.
- **Pandas** – Follow **this guide** if you need help with installation.
- **SciKit Learn** – Follow **this guide** if you need help with installation.

- **SciPy** – Follow **this guide** if you need help with installation.
- **Yellowbrick** – Follow **this guide** if you need help with installation.
- **Matplotlin** – Follow **this guide** if you need help with installation.
- **Sci-Kit Optimization** – Follow **this guide** if you need help with installation.
- **PyTorch** - Follow **this guide** if you need help with installation.

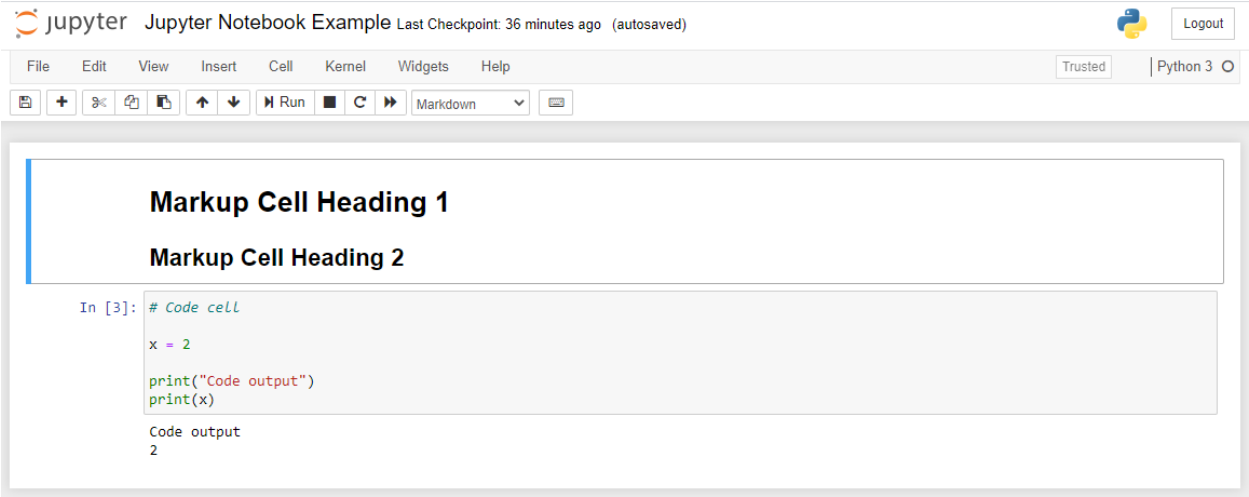
Jupyter Notebook IDE

Regarding the IDE, we use the Jupyter notebook in this book, a powerful tool for interactively developing and presenting data science projects. It has gained popularity among *Python* users, especially among data science folks, although it is popular among *R* users as well. Using Notebooks is now a major part of the data science workflow at companies across the globe. If your goal is to work with data, using a Notebook will speed up your workflow and make it easier to communicate and share your results.
The cool thing is that it is a part of the open-source Project Jupyter, so Jupyter Notebook is completely free. You can download the software on its own or as part of the Anaconda data science toolkit.

Evey Jupyter Notebook, with .ipybn extension, combines code with other rich media, such as narrative text, images, mathematical equations, and so on. In essence, it is a single document that can hold all the information you need, like a full explanation of your project, along with necessary images and equations, and with the interactive code that you can run and see the results immediately. To run Jupyter, you can use the shortcut Anaconda adds to your start menu. This will open a new tab in your default web browser, with the URL to *http://localhost:8888/.* It will look something like this:



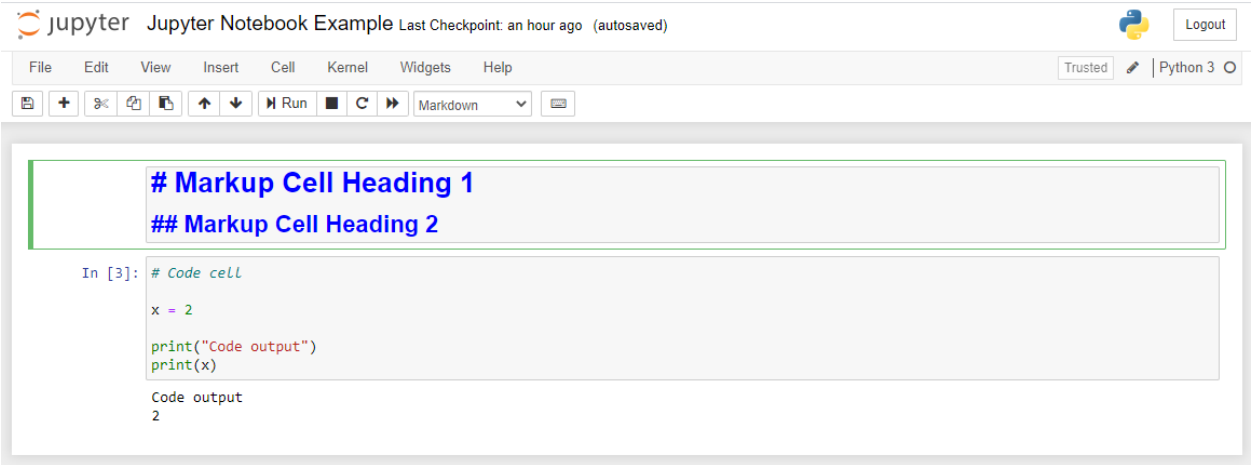
Here you can find a list of notebooks and folders. Then you can create a new notebook or open an existing one. Here is an example of one Jupyter Notebook:



The Notebook is composed of cells and kernels. There are two types of cells:

- A **markdown cell** contains text formatted using Markdown. These cells contain text, images and equations that we talked about. This is a "documentation part" of Jupyter Notebook.
- A **code cell** contains the code which is executed by the kernel. Once the code is run, the notebook displays the output below the code cell that generated it.

Cells are run by using the toolbox at the top or using the keyboard shortcut *Ctrl+Enter*. There are two modes of operating with cells, control mode, and edit mode. Once you are in control mode, the current cell has a blue stripe on the side, just like in the image above. This means that you can control and edit some properties of the cell. For example, you can convert a code cell into a markdown cell by pressing *M* on your keyboard. However, once you press enter, you will start the edit mode, and you can edit the content of the cell itself.

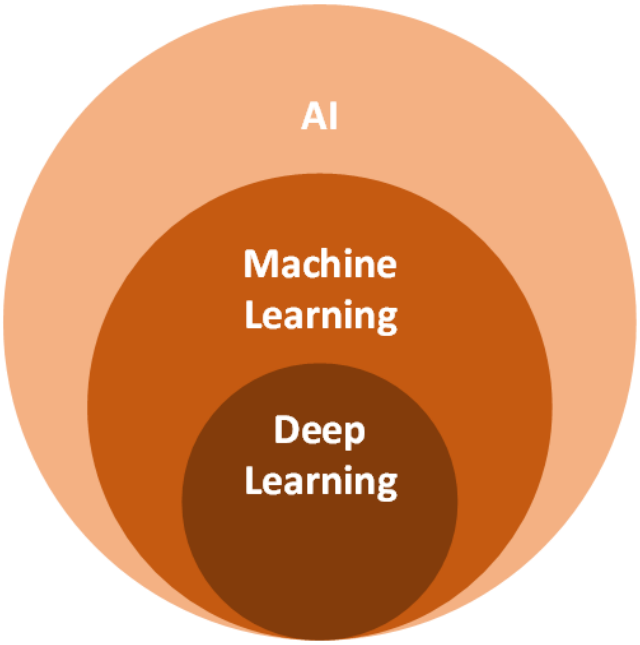


You can go back to command mode by pressing the *Esc* button or running the cell.

Kernel is a "computational engine" that executes the code contained in a notebook, and in our case, that is *Python*. It is important to know that the kernel's state is preserved over time, but between cells as well. This means that libraries and variables used in one notebook are available in another too.

1. The Basics of Machine Learning

Terms like Artificial Intelligence, Machine Learning and Deep Learning are omnipresent. It seems that you can not read an article that doesn't mention one of these terms. There is news about big breakthroughs in the world of machine learning every couple of days. Allegedly, almost every startup is using AI in some sphere of their work. It is time to define it and see what the similarities between those terms are and why they are so often put into the same context. Observe the image below:



Artificial Intelligence is considered to be this large field that includes machine learning. This means that the set of techniques used in AI is larger than the set of techniques used in the so-called classic Machine Learning. We can also see that Deep Learning, i.e., neural networks, is a subset of Machine Learning. One might say that Deep Learning is a special type of Machine Learning. This means that if we want to continue developing knowledge of Deep Learning, we need to define and delve into traditional Machine Learning a bit and see how ideas from this field have influenced the world of Deep Learning.

1.1 Definitions

Machine Learning is defined as a branch of science that uses statistical techniques in order to teach a computer how to perform certain tasks without explicitly programming it. Deep Learning uses the same principle, but it uses techniques from biology and nature instead of statistical ones in order to achieve the same objective.

One of the questions that immediately arises from the previous definition of Machine Learning is "What does learning mean in this context?". Mitchell (1997) offers a good solution to the question, defining learning in the following manner: "a computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*." This is quite a formal definition, so let's approach it from the intuitive side.

We could say that learning represents the means for attaining the ability to perform a task with a certain success rate through experience. However, in order to really get it, we need to explain the main terms that are mentioned here in more detail. First, we will see which kind of tasks we can solve by utilizing machine learning algorithms. Then, we will see how experience is increased; basically, we will talk about the types of learning. Finally, we will see how we can measure the performance of machine learning algorithms.

Generally speaking, Machine Learning tries to solve problems that are hardly solvable by humans or by a standard software development approach. As we mentioned, there is a lot of data out there, but sometimes it is hard to get concrete information out of it. Sometimes it is hard to see the forest for the trees. This is done by examining some samples of data, as we have already briefly mentioned. Samples are a collection of values we call features. Technically, we present a sample as a vector x, with values for each feature xi. For example, for every notebook, there is one row of values that describes how many papers it has, what the quality is, whether it is hardcover, etc. When we have several samples, we get a matrix, i.e., a table. This is why data is usually packed in a .csv file or SQL table.

	Feature 1	Feature 2	...	Feature N - 1	Feature N
Example 1	11	23	...	9	11
Example 2	3	3	...	6	9
...	...	...	...	...	...
Example M - 1	2	33	...	2	2
Example M	6	66	...	3	0

1.2 Problems Machine Learning Solves

How each machine learning algorithm processes these samples depends on the type of task and type of learning. There are several types of tasks for which machine learning is excellent. Here are some of them:

- *Classification* - In this type of task, machine learning has to determine which of *n* categories the provided sample belongs to. During the learning process, the algorithm learns how values of each feature affect the belonging to a certain category. The output is a discrete value.
- *Regression and Forecasting* - The output of solving this task is some kind of a continuous numerical value. For example, we use machine learning algorithms to predict the stock prices of a company. It is similar to the classification, but the output is different. Both of these problems are heavily used in Business Analytics and Fintech.
- *Sequence-to-Sequence* - Unlike the previous problems, the input for these types of problems is a sequence. For example, a machine translation problem is a sequence-to-sequence problem; in oder to translate a sequence of words in one language, a machine learning algorithm needs to generate a sequence of words in another language. Also, any other problems in the area of NLP - natural language processing, can also fall into this category. As you will have a chance to see throughout this book, deep learning is now the cutting edge technology for these types of problems.
- *Data synthesis* - we often want to generate information that is similar to the one that we have as input data. Machine learning algorithms can help us with that. This type of problem can be found in the media industry, for example, in speech and video synthesis.
- *Denoising* - You will often hear data scientists use the term "garbage in - garbage out", which refers to the situation in which you have polluted input data, and your machine learning algorithm is not performing well because of this. For cleaning up the data, you can sometimes use different machine learning algorithms. Apart from that, this type of algorithm can be useful to remove noise from an audio signal or image.
- *Anomaly detection* - Sometimes, you want to detect anomalies in the input data. For example, you want to detect credit card fraud based on the information from the list of purchases. This is possible with machine learning, as well.

There are many more types of problems that machine learning can address. However, these are the most common ones, and they are going to be addressed in this book in one way or another.

1.3 Types of learning

Depending on the type of problem we are trying to solve and how they are learning, we can differentiate different types of machine learning algorithms. There are three general types of learning:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

In **supervised learning**, a learning agent learns how to map certain inputs to an output. The agent learns how to do that because, during the learning process, it is provided with training inputs and labeled expected outputs for those inputs. Using this approach, we can solve many types of problems, mostly the ones that are classification and regression problems in nature. This is an important type of learning and it is the most commonly used commercial approach today, so we will examine it in more detail later.

Another type of learning is **unsupervised learning**. In this type of learning, the agent is provided only with input data, and it needs to make sense of it. The agent is basically trying to find patterns in otherwise unstructured data. This type of learning is usually used for classification or clusterization types of problems.

In **reinforcement learning**, the agent is trying to maximize the reward it gets, not to find hidden patterns. Reinforcement Learning is a type of learning in which a subject interacts with the environment. The subject performs actions and gets a response from the environment. If the response is positive (reward), the subject repeats those actions; otherwise (punishment), it stops doing them. Essentially, it is a type of learning which is inspired by this goal-directed learning from interaction.

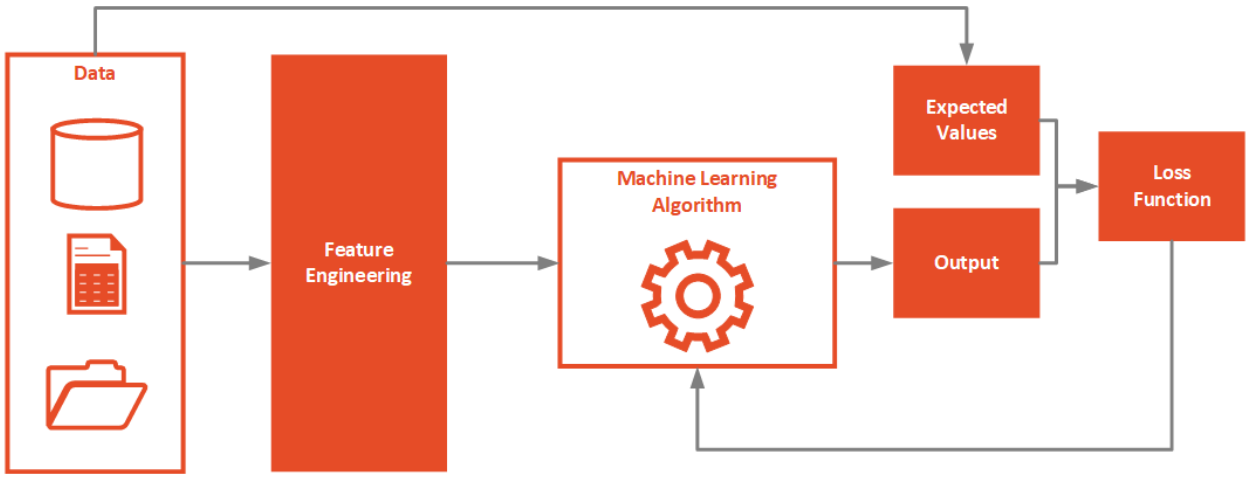
Now, when we know which types of tasks machine learning algorithms can solve and how they learn, i.e., gain experience, we need to observe their performance. Performance, in this case, represents a quantitative measure of how well the machine learning algorithm is performing. This is tightly connected with the type of the task, so measures can differ from case to case. For example, for classification and regression tasks, we usually use accuracy. Most commonly, clients will provide the data that they collected. In order to determine how well our machine learning algorithm is performing, we need to leave some data for evaluation purposes.

The reason behind it is that the goal of machine learning algorithms is ultimately to perform well on the data that they haven't seen before. That is why we split data sets into training and testing data. During the training process, we calculate the training error and try to minimize it, while during the testing period, we measure how well our machine learning algorithm is performing. We allocated different sections where we cover performance metrics and different datasets in more detail.

That would be basic information about machine learning. However, there are certain terms you'll come across if you start exploring this topic in more detail. So, let's cover some of them.

1.2 The Anatomy of Machine Learning Process

Let's take a moment to point out the main building blocks of a machine learning algorithm. The complete process of machine learning can be viewed as a pipeline. At the beginning of that pipeline is, of course, data. The data can come from different sources. It can be generated by other systems, or it can be created by humans. Sometimes we use web scrapers to get the data from the web, while other times, the data is available in a database. More often than not, this data is unstructured and sparse, which can be a problem for algorithms; that is why the second step of the pipeline is pre-processing of the data and feature engineering. This is the step necessary for standard machine learning algorithms, while more advanced approaches, like deep learning, don't need it.



After the data is preprocessed and prepared, we feed it to our machine learning algorithm. This is called the training process, during which the algorithm needs to learn. The machine learning algorithm itself contains different parameters within itself, which make this learning possible. What do we mean by this? Well, based on these input values, the algorithm creates predictions – output values. However, those predictions are not the only output that this algorithm provides.

We have the expected output in supervised learning, so we can use it to calculate how much predictions deviate from the expected results. For this, we can use different functions and measure the penalty in different ways. This function is called the loss function, and the goal of the algorithm is to minimize the penalty calculated by this function. In mathematics, the expression we minimize or maximize is called an objective function.

Then we use the calculated penalty to change the parameters of the machine learning algorithm to get better predictions next time. In general, this is done by minimizing the penalty we calculated. That is how the machine learning algorithm produces the output values during the training process and changes itself to improve its predictions. That is how the implicit output of a machine learning algorithm is the algorithm itself. The trained algorithm is usually called the model. This name comes from mathematics.

There is one very, very important concept that we have brought up several times – the training or learning process. This is a necessary process for every machine learning algorithm in which the neural network gets familiar with the problem it needs to solve. In practice, we usually have some collected data based on which we need to create our predictions, or classification, or any other processing. This data is called a training set. As we were able to see based on the behavior during the training and the nature of the training set, we have a few classes of learning.

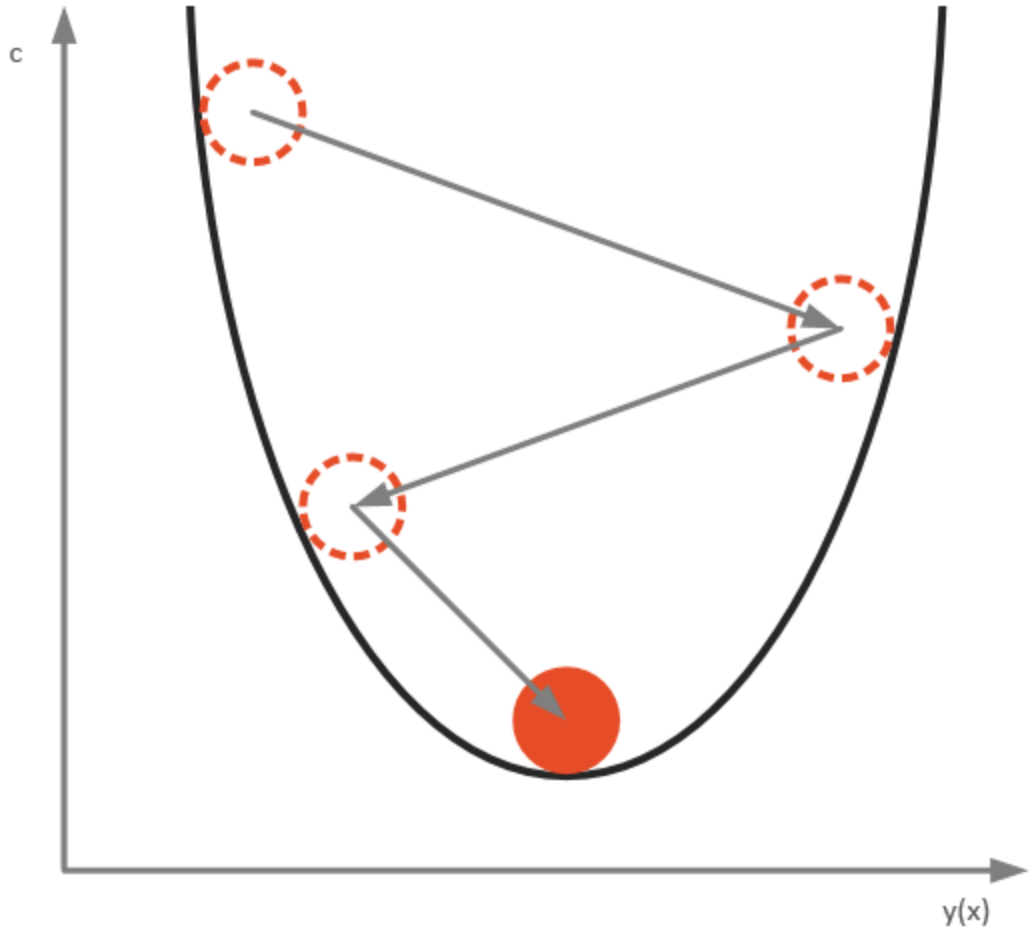
Supervised learning is most commonly used, so let's dig a little deeper into this topic. Basically, we get a training set that contains a vector of input values and a vector of desired output values. Once the machine learning algorithm calculates the output for one of the inputs, the cost function calculates the error vector. This error indicates how close our guess is to the desired output. One of the most-used cost functions is the mean squared error function:

$$C(w, b) = \frac{1}{2n} \sum ||y(x) - a||^2$$

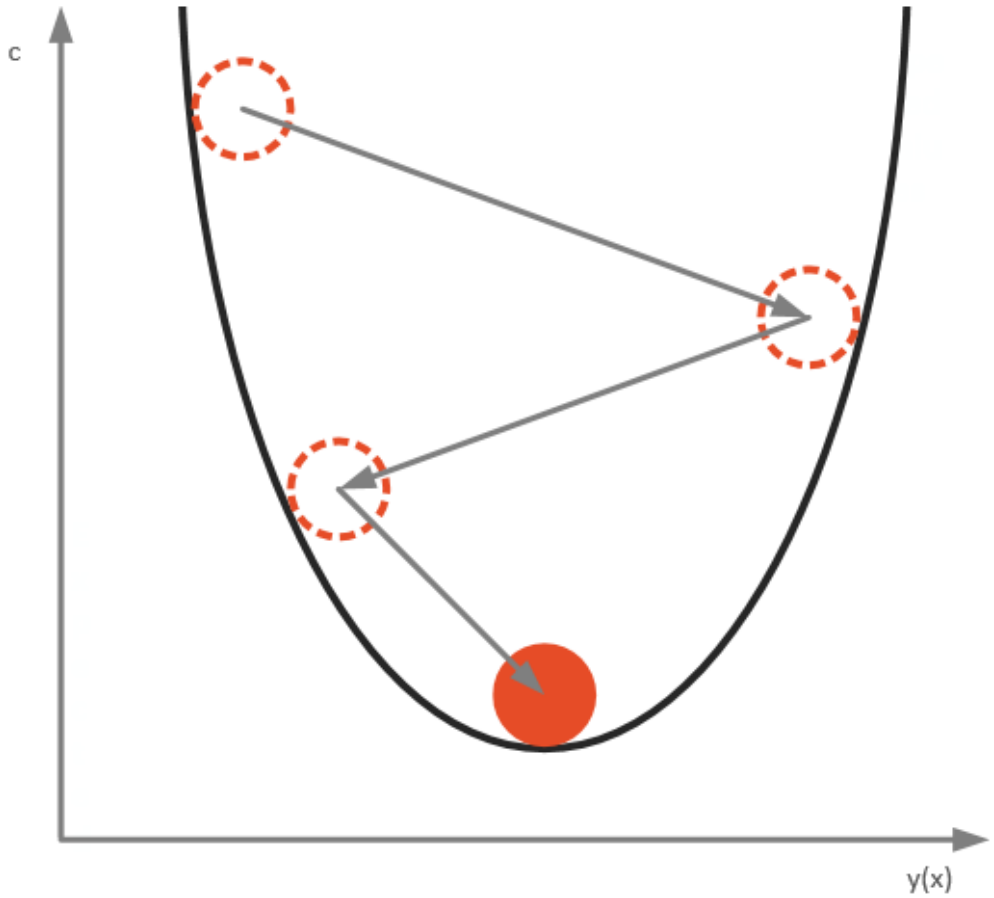
Here, *x* is the training input vector, *y(x)* is the generated output of the machine learning algorithm, and *a* is the desired output. Also, one can notice that this function is a function that depends on *w* and *b*. They represent the parameters of the machine learning algorithm. Now, this error is sent back to the machine learning algorithm. It is used to update the value of parameters.

1.5 Introduction to Gradient Descent

The Gradient Descent is explained in detail in *Chapter 4*, but since it is the essential optimization technique and one of the main ingredients of some algorithms, we are going to briefly explain it here. For more details, go to *Chapter 4.* This is the most popular optimization technique and the basis for other optimization techniques as well. There is one useful analogy that describes this process quite well. Imagine that you had a ball inside a rounded bowl like in the picture below. If you let the ball roll, it will go from one side of the bowl to the other until it goes still at the bottom.



Essentially, we can look at this behavior like the ball is optimizing its position from left to right, and eventually, it stops at the **bottom**, i.e., the lowest point of the bowl. The bottom, in this case, is the minimum of our cost function. This is what the gradient descent algorithm is doing. It starts from one position in which it gets the information about where "the ball" should roll by calculating derivatives and second derivatives of the cost function. Every time we calculate derivatives, we get information about the slope of the side of the function (i.e. bowl,) at its current position.



When the slope is negative (downward from left to right), the ball should move to the right; otherwise, it should move to the left. Be aware that the ball is just an analogy, and we are not trying to develop an accurate simulation of the laws of physics. We are trying to get to the minimum of the function using this alternative method since we realize that using calculus is not optimal.

In a nutshell, the training process of machine learning algorithm with Gradient Descent can be described like this:

1. Put the training set in the machine algorithm and get the output for each sample in it.
2. The output is compared with the desired output and error is calculated using a cost function.
3. Using the error value and the cost functions, the algorithm makes a decision on how to change the internal parameters. This way the error value is minimized.
4. The process is repeated until the error is minimal.

1.6 Three datasets

Data is precious. Many projects that want and need to use machine learning are stuck because they don't have enough data. However, why do we need that much data, and should we use all data for training purposes? In general, during the training process of machine learning algorithms, we split the complete dataset into three parts. These parts are, of course, created from one source, which we need to split in order to be able to perform all important steps in developing one machine learning application.

The first part is called - **training dataset**. This dataset is a subset of the complete dataset which we use for training machine learning algorithms. It is the largest part, and it usually contains 70-80% of the complete dataset.

The second part is called the **validation dataset**. This dataset is also used during the training process, but it is used to validate the training process. This means that we run model predictions on the validation dataset in each epoch and check how well it performs. This means that we don't want to leak more information to our dataset, but we want to see how well it performs on unknown data during the training process. It usually takes around 10% of the training dataset.

Finally, the third part is called a **test dataset**. This dataset is used during the testing phase. Once the training is done, we want to confirm that our model performs well. That is why we left some data on the side before training and used it later to evaluate the performance of the model. We use this data once the training process is completed and calculate the performance of the model on thus far unseen data. It usually takes around 20% of the complete dataset.

1.7 Hyperparameters

As we have learned from the previous chapters, every machine learning algorithm has a set of parameters that are modified during the training so the predictive power of the model is improved. However, there is a second set of parameters that are not changed by the machine learning algorithm but are manually set by machine learning engineers. These parameters are called **hyperparameters,** and they can be observed as a configuration for the training and for the machine learning algorithm itself.

1.8 Learning Rate

Of all hyperparameters, one of the most important ones is the learning rate. This hyperparameter dictates how fast our machine learning algorithm learns. In essence, this value is the factor that controls how much we modify parameters of the machine learning algorithm in response to the estimated error. Everything that is calculated with

the loss function is additionally multiplied by the learning rate. That is why it is often called step size. It is an old data science trick - adding additional factors that contribute to the final result.

Getting to the correct value of the learning rate parameter might be a challenging task. This is due to the fact that high learning rate values might cause unstable training and low values can cause that training lasts a very long time. Both options are, of course, unacceptable. Because the learning rate controls how quickly the model is adapted to the data, small values require more training epochs and might cause a model to be stuck, while larger values can cause the model to converge too quickly to the suboptimal solution. That is why selecting the correct learning rate is especially important. In the following chapters, we will learn how to optimize all hyperparameters of the machine learning algorithm, even the learning rate.

1.9 Performance Metrics

Before we dive into the machine learning algorithms and their specifics, let's first explore which **metrics** we can use to evaluate the performance of our machine learning models and how we do it in *Python*.

1.9.1 Classification Metrics

Since the problems in which we use machine learning fall within different **categories**, we have different metrics for different types of problems. First, let's explore metrics that are used for **classification problems**. In order to represent all these metrics, we use simple data:

```
actual_values = [1, 1, 0, 1, 0, 0, 1, 0, 0, 0]
predictions =   [1, 0, 1, 1, 1, 0, 1, 1, 0, 0]
```

So our dataset is composed of two classes – *Class 0* and *Class 1*. The model predicted the class of some samples well, while the others it misclassified. So let's explore how we define metrics that can tell us exactly how good our model is.

1.9.1.1 Confusion Matrix

There are some basic terms that we need to consider when it comes to the performance of classification models. These terms are best described and defined through the **confusion matrix**. The *Confusion matrix* or *Error Matrix* is one of the key concepts when we are talking about classification problems. This matrix is the *NxN* matrix, and it is a tabular representation of model predictions vs. actual values.

Confusion Matrix for the example data that we use is built like this:

		Actual Values	
		Class 1	Class 0
Predicted Values	Class 1	3	3
	Class 0	1	3

Each column and row is dedicated to one class. On one end, we have the actual values and on the other end, predicted values. What is the meaning of the values in the matrix? In the example above, from 4 values marked as *Class 0*, our model correctly classified 3 values and misclassified 1 value. This model, from the 6 values marked as *Class 1*, correctly labeled 3 and misclassified 3. If we refer to *Class 1* as positive and *Class 0* as a negative class, then 3 samples predicted as *Class 0* are called **true-negatives**, and the 1 sample predicted as *Class 1* is referred to as **false-negative**. The 3 samples correctly classified as *Class 1* are referred to as true-positives, and those 3 misclassified instances are called **false-positive.**

		Actual Values	
		Class 1	Class 0
Predicted Values	Class 1	True Positives	False Positives
	Class 0	False Negatives	True Negatives

Remember these terms, since they are important concepts in machine learning. It is one of the most asked questions in job interviews. Note that false-positives are also called **Type I** error and false-negatives are called **Type II** error. The good news is that we can use *Sci-Kit Learn* to get this matrix:

```
print(metrics.confusion_matrix(actual_values, predictions))
```

```
[[3 3]
 [1 3]]
```

The result is exactly like in the **table** above. This matrix is not only giving us details about how our prediction model works but on the **concepts** laid out in this matrix we are building some of the other **metrics**. Other classification metrics can be retrieved again by using *Sci-Kit Learn*, or to be more precise, by using **classification report**:

```
print(metrics.classification_report(actual_values, predictions))
```

```
              precision    recall  f1-score   support

           0       0.75      0.50      0.60         6
           1       0.50      0.75      0.60         4

    accuracy                           0.60        10
   macro avg       0.62      0.62      0.60        10
weighted avg       0.65      0.60      0.60        10
```

Let's go into more details for each of them. Note that the support column represents the number of samples of the true response that lie in that class.

1.9.1.2 Accuracy

We start off with the metric that is the easiest one to understand – **accuracy**. It is calculated as the number of correct predictions divided by the total number of predictions. When we multiply it by 100, we get accuracy in percentages.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ samples}$$

From the above output, we can see that the accuracy of our predictions is 0.60, meaning 60%. Let's check that. For *Class 0*, the model correctly predicted 3 out of 6, and for *Class 1,* the model correctly predicted 3 out of 4. If we put that into an accuracy formula we get (3 + 3) / (6 + 4) = 6 / 10 = 0.6. There is one more way to get this metric:

```
print(f'Accuracy Score is: {metrics.accuracy_score(actual_values,
predictions) * 100} % ')
```

```
Accuracy Score is: 60.0 %
```

Accuracy, however, is a tricky metric because it can give us **wrong impressions** about our model. This is especially the case in situations where the database is imbalanced, i.e., there are many samples of one class and not much of the other. If your model is performing well on the class that is dominant in the dataset, accuracy will be high even though the model might not perform well in other cases. Also, note that models that **overfit** have an accuracy of 100%.

1.9.1.3 Precision

Precision is a very useful metric and it carries more **information** than accuracy. Essentially, with precision, we answer the **question**: "What proportion of positive identifications was correct?" It is calculated for each class separately with the formula:

$$Precision = \frac{True\ Positives}{(True\ Positives + False\ Positives)}$$

Its value can go from 0 to 1. Applied to our example, the precision for *Class 0* can be calculated as samples correctly predicted as *Class 0* divided by the total number of samples predicted as *Class 0* – 3/4 = 0.75. For *Class 1* – 3/6 = 0.5. These values can be seen in the *classification report* above. We can also calculate the **precision** score like this:

```
print(f'Precision Score is: {metrics.precision_score(actual_values, predictions)}')
```

```
Precision Score is: 0.5
```

It goes without saying that we should aim for higher precision.

1.9.1.4 Recall

Recall can be described as the ability of the classifier to find all the positive samples. With this metric, we are trying to answer the **question**: "What proportion of actual positives was identified correctly?" It is defined as the fraction of samples from a specific class, correctly predicted by the model or mathematically:

$$Recall = \frac{True\ Positives}{(True\ Positives + False\ Negatives)}$$

This metric falls within the 0-1 range as well, with the 1 being the best value. Applied to our example, recall for *Class 0* can be calculated as samples correctly predicted as *Class 0* divided by the total number of samples predicted as *Class 0*– 3/6 = 0.5. For *Class 1* – 3/4 = 0.75. These values can be seen in the *classification report* above. We can also calculate recall score like this:

```
print(f'Recall Score is: {metrics.recall_score(actual_values, predictions)}')
```

```
Recall Score is: 0.75
```

Precision and recall are different and yet so similar. Precision is a measure of result **relevancy**, while recall is a measure of how many **truly relevant** results are returned. In the beginning, it might be hard to decipher the **difference** between these two. If you are confused as well, imagine this situation.

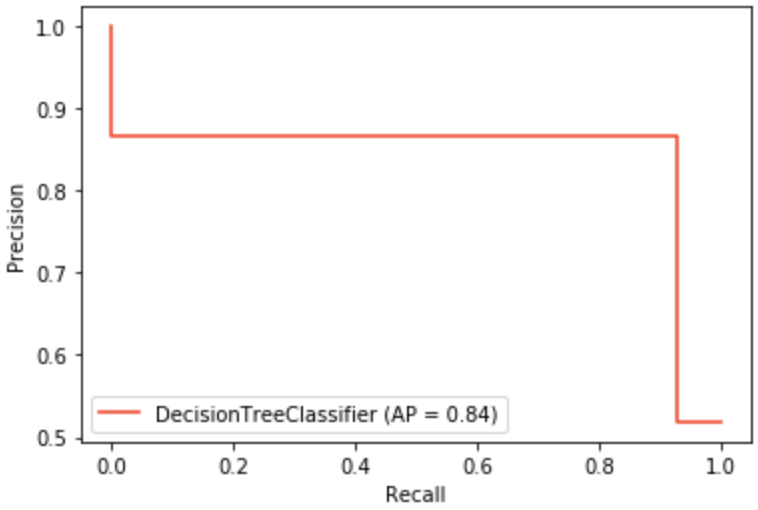
A client calls a bank in order to verify that her accounts are secure. The bank says that everything is properly secured; however, in order to double-check, they ask the client to

remember every time that she shared account details with someone. What is the **probability** that the client will remember every single time she did that? So, the client remembers 10 situations in which that happened, even though in reality there were 8 situations, i.e., she falsely identified two additional situations. This means that the client had a recall of 100%, meaning she did cover all the necessary options. However, the client's precision is 80% since two of those situations are falsely identified as such.

1.9.1.5 Precision-Recall Curve

In order to correctly evaluate a model, both metrics need to be taken into consideration. Unfortunately, improving precision typically **reduces** recall and vice versa. The precision-recall curve shows the **tradeoff** between precision and recall. To get a precision-recall curve, we use *Sci-Kit Learn* function *plot_precision_recall_curve* with the provided estimator:

```
disp = metrics.plot_precision_recall_curve(model, X_test, y_test,
color='orange')
```



The area under the curve represents both **high recall** and **high precision**. High scores for both show that the classifier is returning **accurate** results with a majority of positive results. Many other metrics are developed based on the definitions of both precision and recall. One of them is the **F1 Score.**

1.9.1.6 F1 Score

The **F1 Score** is probably the most popular metric that combines precision and recall. It represents the harmonic mean of them. For binary classification, we can define it with the formula:

$$F1\ Score = 2 * \frac{Precision * Recall}{(Precision + Recall))}$$

For our example, we can calculate **F1 score** for *Class 0* as – 2 * 0.5 * 0.75 / (0.5 + 0.75) = 0.6. For *Class 1* we get the same value – 0.6. These values can be found in the classification report or running *f1_score Sci-Kit Learn* Method:

```
print('F1 Score:', metrics.f1_score(actual_values, predictions))
```

```
F1 Score: 0.6
```

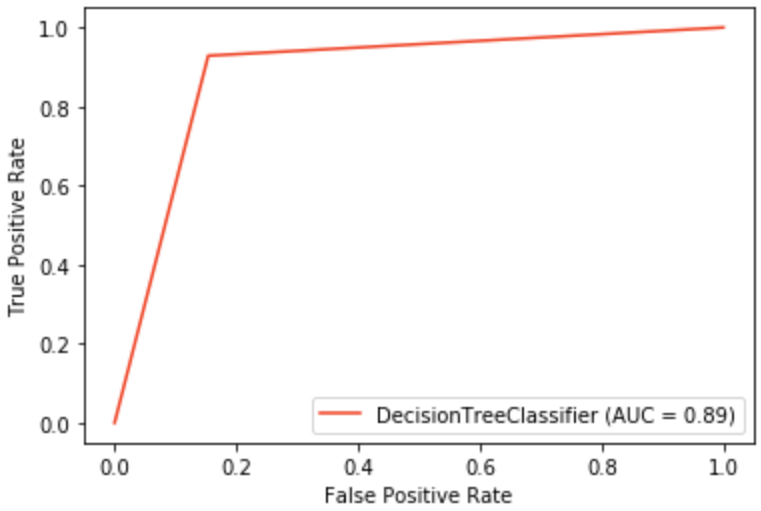
This formula can be generalized for classification problems with more classes as:

$$F1\ Score = \frac{(1 - \beta^2) * Precision * Recall}{\beta^2 * (Precision + Recall))}$$

where **β** is 1 in binary classification, i.e., it represents the number of possible classes reduced by 1. The best value for the **F1 score** is at 1 and the worst score at 0.

1.9.1.7 Receiver Operating Characteristic (ROC) curve & Area Under the curve (AUC)

Ok, those are all metrics from the *classification report*. However, there are several other **techniques** that we can apply in order to evaluate the performance of our model. When predicting the class of a sample, the machine learning algorithm first calculates the **probability** that the processed sample belongs to a certain class, and if that value is above a predefined **threshold,** it labels it as that class. For example, if for the first sample the algorithm predicts that there is a 0.7 (70%) chance that it is *Class 0* and the threshold is 0.6, the sample will be labeled as *Class 0*. This means that for different thresholds we can get different labels. This is where we can use the **ROC** (Receiver Operating Characteristic) curve. This curve shows the true positive rate against the false-positive rate for various thresholds.



However, this metric isn't helping us with model evaluation directly. What is especially interesting about the image above is the **area under the curve** or **AUC**. This metric is, in fact, used as a measure of performance. We can say that ROC is a probability curve and AUC measures the separability, i.e. the **AUC-ROC** combination tells us our model is capable of distinguishing **classes**. The higher this value, the better. We can calculate it fairly easily using *Sci-Kit Learn* function *roc_auc_score*:

```
print('AUC-ROC:', metrics.roc_auc_score(actual_values, predictions))
```

```
AUC-ROC: 0.625
```

1.9.1.8 Log Loss

What is really interesting about this metric is that it is one of the most used evaluation metrics in **Kaggle** competitions. Log Loss is a metric that quantifies the accuracy of a classifier by **penalizing** false classifications. Minimizing this function can be, in a way, observed as maximizing the accuracy of the classifier. Unlike the accuracy score, this metric's value represents the **amount** of **uncertainty** of our prediction based on how much it varies from the actual label. For this, we need **probability** estimates for each class in the dataset. Mathematically we can calculate it as:

$$Log\ Loss = -\frac{1}{N}\sum_{i=1}^{N}(y_i\ log(y_i') + (1 - y_i)log(1 - y_i'))$$

Where *N* is the number of samples in the dataset, *yi* is the actual value for the i-th sample, and *yi'* is a prediction for the i-th sample. In code, we can calculate it like this:

```
print('LOGLOSS:', metrics.log_loss(actual_values, predictions))
```

```
LOGLOSS: 13.815750437193334
```

This metric is also called logistic loss or cross-entropy loss in literature.

1.9.2 Regression Metrics

Since the goal differs when solving regression problems, we need to use different metrics. In general, the output of the regression machine learning model is always continuous, and thus metrics need to be aligned for that purpose. Just like for the classification, we use simple data that looks like this:

```
actual_values = [9, -3.3, 6, 11]
predictions =  [8.5, -2.9, 6, 9.2]
```

As you can see, some predictions deviate from the actual values. Let's see how we can calculate the **quality** of these predictions.

1.9.2.1 Mean Absolute Error – MAE

As the name suggests, this metric calculates the average absolute distance (error) between the predicted and target values. It is defined by the formula:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - y_i'|$$

Where *N* represents the number of samples in the dataset, *yi* is the actual value for the i-th sample, and *yi'* is the predicted value for the i-th sample. This metric is robust to outliers, which is really nice. In the code, we use the *mean_absolute_error* method:

```
print (f'MAE: {metrics.mean_absolute_error(actual_values,
predictions)}')
```

```
MAE: 0.6750000000000002
```

1.9.2.2 Mean Squared Error – MSE

This is probably the most popular metric of all regression metrics. It is quite simple; it finds the average squared **distance**(error) between the predicted and actual values. The formula that we use to calculate it is:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_i')^2$$

Where *N* represents the number of samples in the dataset, *yi* is the actual value for the i-th sample, and *yi'* is the predicted value for the ith sample. The result is a non-negative value and the goal is to get this value as close to **zero** as possible. This function is often used as a loss function of a machine learning model. In the code, we use the *mean_squared_error* method:

```
print (f'MSE: {metrics.mean_squared_error(actual_values,
predictions)}')
```

```
MSE: 0.9125000000000005
```

1.9.2.3 Root Mean Square Error – RMSE

Another very popular metric. It is a variation of the *MSE* metric. In general, it shows what the average **deviation** in predictions from the actual values is, and it follows an assumption that error is unbiased and follows a **normal** distribution. We calculate this value with the formula:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - y_i')^2}$$

Just like *MSE*, *RMSE* is a non-negative value, and the value 0 is the value we are trying

to achieve. A lower *RMSE* is better than a higher one. There is no out of the box code for this metric and we can calculate it quite easily using *Python*:

```
def rmse(actual_values, predictions):
          actual_values = np.asarray(actual_values)
          predictions = np.asarray(predictions)
          return np.sqrt(((predictions - actual_values)**2).mean())
               print(f'RMSE: {rmse(actual_values, predictions)}')
```

```
RMSE: 0.9552486587271403
```

Or we can use *mean_squared_error* again:

```
print(f'RMSE: {sqrt(metrics.mean_squared_error(actual_values,
predictions))}')
```

```
RMSE: 0.9552486587271403
```

RMSE punishes **large errors** and is the best metric for large numbers (actual value or prediction). Note that this metric is affected by **outliers**, so make sure that you remove them from the dataset beforehand.

1.9.2.4 Root Mean Squared Logarithmic Error – RMSLE

Another variation on *MSE* and *RMSE* is **RMSLE**. The formula is defined like this:

$$RMSLE = \sqrt{MSLE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(log(y_i' + 1) - log(y_i + 1))^2}$$

In the code, we can use the mean_squared_log_error function. Note that data must be scaled in 0-1 range:

```
actual_values_ranged = minmax_scale(actual_values,
feature_range=(0,1))
predictions_ranged = minmax_scale(predictions, feature_range=(0,1))

print(f'RMSLE:
{sqrt(metrics.mean_squared_log_error(actual_values_ranged,
predictions_ranged))}')
```

```
RMSLE: 0.0331452609154312 75
```

RMSLE is robust to the **outliers** and it penalizes the **underestimations** more than the overestimations. If the values of the samples are large numbers, consider using *RMSE*, because this metric is not the best fit for it.

1.9.2.5 R Squared

The metrics like *RMSE* and *RMSLE* are quite useful, but sometimes not intuitive. What we lack is some sort of benchmark for them. In cases where we need a more intuitive approach, we can use the **R-Squared** metric. The formula for this metric goes as follows:

$$R^2 = 1 - \frac{MSE_{model}}{MSE_{base}}$$

Where *MSEmodel* is the *MSE* of the predictions against real values, while *MSEbase* is the MSE of **mean prediction** against real values. This means that we use the mean of the predictions as a **benchmark**. Quite elegant, isn't it? To calculate it in the code we use *r2_score*:

```
print (f'R Squared: {metrics.r2_score(actual_values, predictions)}')
```

```
R Squared: 0.9696004330897203
```

2. Machine Learning Algorithms

2.1 Regression Algorithms

While we are solving regression problems, we create estimates of the relationships between a dependent variable (outcome) and one or more independent variables (feature). For example, we want to predict the price of the house based on the age of the tenants. Price is the outcome, a dependent variable, while the age of the tenant is the independent variable or feature. There are several approaches to solving this and the simplest one is Linear Regression.

2.1.1 Simple Linear Regression

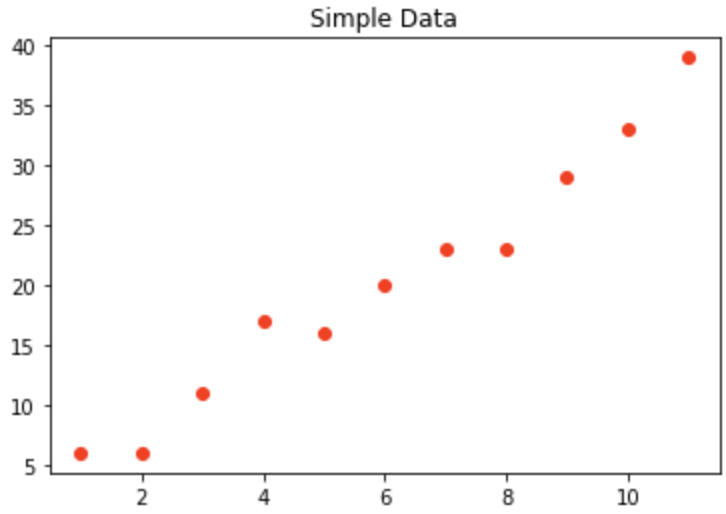
Sometimes the data that we have is quite **simple**. Sometimes, the output value of the dataset is just the **linear combination** of features in the input example. Let's simplify it even further and say that we have only **one** feature in the input data. A mathematical model that describes such a **relationship** can be presented with the formula:

$$f(x_i) = b_0 + b_1 * x_i$$

For example, let's say that this is our data:

X	1	2	3	4	5	6	7	8	9	10	11
Y	6	6	11	17	16	20	23	23	29	33	39

In this particular case, the mathematical model that we want to create is just a **linear function** of the input feature, where *b0* and *b1* are the model's **parameters**. These parameters should be learned during the training process. After that, the model should be able to give correct output **predictions** for new inputs. To sum up, during training, we need to learn *b0* and *b1* based on the values of *x* and *y*, so our *f(xi)* is able to return correct predictions for the **new** inputs. If we want to **generalize** even further, we can say that the model makes a prediction by adding a constant (bias term – *b0*) on the precomputed weighted sum (*b1)* of the input features. However, let's get back to our example and clear things up a little bit before we dive into generalization. Here is what the aforementioned data looks like on the **plot**:



By calculating optimal *b0* and *b1, o*ur linear regression model produces a line that will best fit this data. This line should be optimally **distanced** from all points in the graph. It is called the **regression line**. So, how does the algorithm calculate *b0* and *b1* values?

In the formula above, *f(xi)* represents the predicted output value for *i-th* example from the input, and *b0* and *b1* are regression coefficients that represent the **y-intercept** and **slope** of the regression line. We want that value to be as close as possible to the real value – *y*. Thus the model needs to learn the values regression coefficients *b0* and *b1*, based on which model will be able to predict the correct output. In order to make these estimates, the algorithm needs to know how bad the **current** estimations of these coefficients are. At the beginning of the training process, we feed samples into the algorithm, which calculates the output *f(xi)* of the current sample based on the **initial** values of regression coefficients. Then the error is **calculated** and coefficients are corrected. The error for each sample can be calculated like this:

$$e_i = y_i - f(x_i)$$

This means that we **subtract** the estimated output from the real output. Note that this is a training process, and we **know** the value of the output in the i-th sample. Because *ei* depends on coefficient values, it can be described by the **function**. If we want to minimize *ei* and for that, we need to define a function based on which we will do so. In this chapter, we use the **Least Squares Technique** and define the function that we want to minimize as:

$$J(b_0, b_1) = \frac{1}{2n}\sum_{i=1}^{n} e_i^2 = \frac{1}{2n}\sum_{i=1}^{n}(f(x_i) - f'(x_i))^2$$

The function that we want to minimize is called the **objective function** or **loss function**. In order to minimize *ei*, we need to find coefficients *b0* and *b1* for which *J* will hit the global minimum. Without going into mathematical details, here is how we can calculate values for *b0* and *b1:*

$$b_1 = \frac{SS_{xy}}{SS_{xx}}$$

$$b_0 = \bar{y} - \beta_1 \bar{x}$$

Here *SSxy* is the sum of cross-deviations of y and x:

$$SS_{xy} = \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) = \sum_{i=1}^{n} y_i x_i - n\bar{x}\bar{y}$$

while *SSxx* is the sum of squared deviations of x:

$$SS_{xx} = \sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n(\bar{x})^2$$

Ok, so much for theory, let's implement this algorithm using *Python*.

2.1.1.1 Simple Linear Regression Python Implementation

The algorithm we discussed previously is implemented within the *SimpleLinearRegression* class:

```
class LinearRegression():
    ''' Class that implemnets Simple Linear Regression '''
    def __init__(self):
        self.b0 = 0
        self.b1 = 0

    def fit(self, X, y):
        mean_x = np.mean(X)
        mean_y = np.mean(y)

        SSxy = np.sum(np.multiply(X, y)) - len(x) * mean_x * mean_y
        SSxx = np.sum(np.multiply(X, x)) - len(x) * mean_x * mean_x

        self.b1 = SSxy / SSxx
        self.b0 = mean_y - self.b1 * mean_x

    def predict(self, input_data):
        return self.b0 + self.b1 * input_data
```

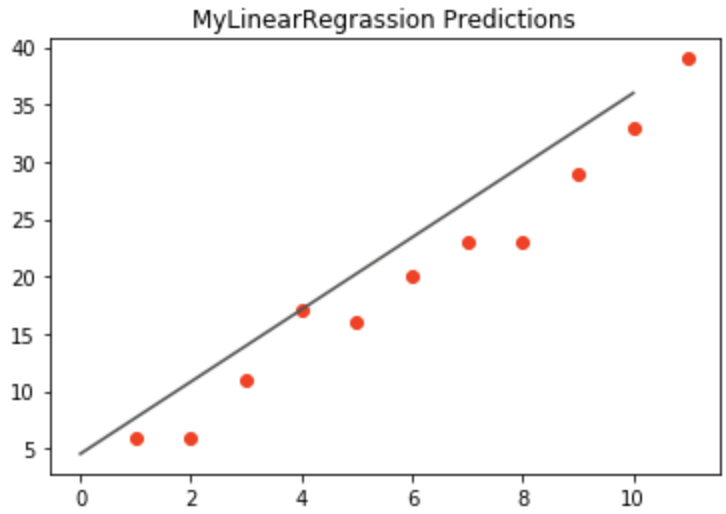
This class has two functions, out of which the *fit* method is extremely **interesting**. In this method, we first **calculate** the mean values of input data and the expected output. Then we calculate *SSxy* and *SSxx* and utilize those values to calculate *b0* and *b1.* Basically, based on the input data, we calculated the necessary values for *b0* and *b1*, i.e. this is how we created our **model**. We can consider the *fit* method as our **training** function. The *predict* method uses these values to calculate values for the new data. Here is how we can use this function on the data from the previous chapter:

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
y = np.array([6, 6, 11, 17, 16, 20, 23, 23, 29, 33, 39])

model = LinearRegression()
model.fit(x, y)
```

Then we use the **trained model** to predict new values and the plot regression line:

```
predictions = model.predict(x)
plt.scatter(x = x, y = y, color='orange')
```

```
plt.plot(predictions, color='orange')
plt.show()
```



2.1.2 Multiple Linear Regression

Ok, that was super simple. The usage of this example is very limited since we usually end up with **datasets** with more **features** in them. Let's take it up a notch and get a little more practical…and mathematical. We observe a set of labeled samples *{(xi, yi)} Ni=1*. The *N* is the size of the set, while *xi* is the *D*-dimensional feature vector, and *yi* is the output. Every feature x is the real number. One such dataset is the famous **Boston Housing Dataset**. Once we load it with *Pandas*, we get something like this:

```
data = pd.read_csv('./data/boston_housing.csv')
data.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

In this dataset, the **output** is the *medv* feature, while the rest of the features are **input** features. As you can see, there are several features *(x1 – crim, x2-zn,….)* for each

sample *i*. Now, we can **generalize** principles of linear regression and use them on a dataset with more features. We can present the model with a formula:

$$f(x) = b_0 + b_1 * x_1 + b_2 * x_2 + ... + b_n * x_n$$

Or to simplify it even further:

$$f(X) = b + w * X$$

We changed the notion there a little bit, but it is essentially the **same** as the previous formula; it is just **vectorized**. The bias *b0* became *b*. The *w* is now a *D*-dimensional vector (because we have a *D* number of features, remember?) of parameters. To predict the *y* for a given *x,* we use this model. Obviously, we want to find the optimal values for coefficients (*w, b*) for which the model will output accurate predictions. Unlike the simple linear regression that creates a line, the multiple linear regression creates a **hyperplane** since every feature represents one **dimension**. This hyperplane is chosen like that to be as **close** to all sample values as possible. To calculate the optimal coefficient, this time we want to minimize the **Mean Squared Error** function:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y - f(x_i))^2$$

To quickly find the values of w and b that minimize MSE, we use the so-called **Normal Equation**. This equation gives direct results for mentioned coefficients:

$$w, b = (X * X^T)^-1 * X^T * y$$

Ok, let's utilize this in the code.

2.1.2.1 Multiple Linear Regression Python Implementation

The algorithm we discussed previously is implemented within the *MultipleLinearRegression* class:

```
class MultipleLinearRegression():
    ''' Class that implements Multiple Linear Regression '''
    def __init__(self):
        self.b = 0
        self.w = []

    def fit(self, X, y):
        # If there is only one feature we need to reshape input.
        if len(X.shape) == 1:
            X.reshape(-1, 1)

        # Add 'ones' to model coefficient b in data.
        ones = np.ones(shape=X.shape[0]).reshape(-1, 1)
        X = np.concatenate((ones, X), 1)

        coeficients =
                  np.linalg.inv(X.transpose().dot(X)).dot(X.transpose()).dot(y)
        self.b = coefficients[0]
        self.w = coefficients[1:]

    def predict(self, X):
        predictions = []
        for x in X:
            prediction = self.b

            for xi, wi in zip(x, self.w):
                prediction += wi * xi

            predictions.append(prediction)

        return predictions
```

The '*blueprint*' is the same as for the previous implementation; we have two functions, *fit* and *predict*. The first one **creates** the model, while the other **utilizes** it. In the *fit* method, we first reshape input data if it has only one dimension. This way, we **incorporate** the implementation of Simple Linear Regression in this class as well. Then, we extend the input data with a vector of ones. This is done because the bias coefficient – *b* is not implicitly modeled in the data, i.e., this value is not multiplied with

any feature from the input data and we need to add it **manually**. So, we add an array of ones, so this value is calculated as well. After that, we utilize the **Normal Equation** and calculate all the coefficients. Here is how we can use this class on the Boston Housing Dataset:

```
X = data.drop('medv', axis=1).values
y = data['medv'].values

model = MultipleLinearRegression()
model.fit(X, y)

predictions = model.predict(X)
```

First, we **separate the** input data – X from the output data – y. Then we create an **object** of *MultipleLinearRegression* class and call the *fit* method with input data. Finally, we call the *predict* method to generate **predictions**. To compare predictions made by our implementation with the actual values, we can create *Pandas Dataframe*:

```
predictions = model.predict(X)
pd.DataFrame({
    'Actual Value': y,
    'Vanilla Model Prediction': predictions,
})
```

	Actual Value	Vanilla Model Prediction
0	24.0	30.003843
1	21.6	25.025562
2	34.7	30.567597
3	33.4	28.607036
4	36.2	27.943524
...	...	...
501	22.4	23.533341
502	20.6	22.375719
503	23.9	27.627426
504	22.0	26.127967
505	11.9	22.344212

From the results, we can see that overall results are **close** but not too close to the real results. Keep in mind we haven't performed any data **preprocessing,** and that we haven't scaled the data. Let's see how we can do that and use the *SciKit Learn* implementation of the same algorithm.

2.1.2.2 Using SciKit Learn

Our previous implementations are quite vanilla. If we try to use them on a large dataset, we may encounter some problems. For starters, it loads all the data, so we can only hope that we have enough memory for the whole dataset. Apart from that, this is really computing expensive ways to perform Linear Regression. The computational complexity of inverting a matrix of n features (which a normal equation does) is typically about O(n2.4) to O(n3). This means that if we have a larger dataset, which has twice as many features (2*n) and the computation time will be 5.3 to 8 times larger.

We haven't supported **gradient descent** or some other smart optimization technique. Lucky for us, the guys from *SciKit Learn* did. So let's first fix several things we overlooked in the previous implementation. We need to split our data into training and testing sets; this is mandatory. We can also compare the results of our implementation with the out-of-the-box solution that *SciKit Learn* provides.

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)

# Re-Train implementation
model.fit(X_train, y_train)

# Use and train SciKit Learn Linear Regression
sk_model = LinearRegression()
sk_model.fit(X_train, y_train)

# Make predictions
sk_predictions = sk_model.predict(X_test)
predictions = model.predict(X_test)

# Compare
pd.DataFrame({
    'Actual Value': y_test,
    'Vanilla Model Prediction': predictions,
    'SciKit Model Prediction': sk_predictions,
})
```

Here is the output:

	Actual Value	Vanilla Model Prediction	SciKit Model Prediction
0	20.5	20.535939	20.535939
1	5.6	11.400370	11.400370
2	13.4	13.024932	13.024932
3	12.6	18.448961	18.448961
4	21.2	22.634241	22.634241
...	...	...	...
97	25.0	25.216456	25.216456
98	19.5	16.798470	16.798470
99	19.9	20.075704	20.075704
100	15.4	14.467506	14.467506
101	21.7	20.719257	20.719257

We can see that we have the **same** results using both implementations. Also, we can see that the results are not that bad. Can we improve them? We can try using *SGDRegressor*. This is a Linear Regression Model that uses Stochastic Gradient Descent for **optimization**. If we want to use this algorithm, we must scale data. For that purpose, we utilize the *SciKit Learn's Pipeline,* the *StandardScaler and* the *SGDRegressior*:

```
sgd_pipeline = make_pipeline(StandardScaler(), SGDRegressor(max_iter=10000,
alpha=0.1))
sgd_pipeline.fit(X_train, y_train)

sgd_predictions = sgd_pipeline.predict(X_test)

pd.DataFrame({
    'Actual Value': y_test,
    'Vanilla Model Prediction': predictions,
    'SciKit Model Prediction': sk_predictions,
    'SGD Model Prediction': sgd_predictions,
})
```

First, we create a pipeline of the *StandardScalerI and* the *SGDRegressor.* This means that when we use the fit method of this pipeline, the data will first be **processed** by the scalar and then the regressor will be trained. Here is the comparison of the results:

	Actual Value	Vanilla Model Prediction	SciKit Model Prediction	SGD Model Prediction
0	20.5	20.535939	20.535939	19.214685
1	5.6	11.400370	11.400370	10.354499
2	13.4	13.024932	13.024932	17.140676
3	12.6	18.448961	18.448961	9.019113
4	21.2	22.634241	22.634241	29.144803
...	...	...	...	...
97	25.0	25.216456	25.216456	20.362897
98	19.5	16.798470	16.798470	22.423240
99	19.9	20.075704	20.075704	20.118347
100	15.4	14.467506	14.467506	-2.998486
101	21.7	20.719257	20.719257	24.714092

We can see that results are **mixed**. Sometimes we get better results using the standard approach, while other times we get better results using the Stochastic Gradient Descent.

2.2 Classification Algorithms

When we are solving classification problems, we want to predict the class label of the observed sample. For example, we want to predict a class of penguins based on their bill length and width. There are several approaches to solving this. As we will see, some of the solutions are based on calculating distances, while others are based on creating a probabilistic model. One way or another, the goal is to create function *y = f(X)* that minimizes the error of misclassification, where *X* is the set of observations and *y* is the output class label.

2.2.1 Logistic Regression

The first algorithm that we explore in this article is Logistic Regression. The name might be a bit confusing because it comes from statistics and it is due to the similar mathematical formulation for Linear Regression. Just to simplify things even more for this first algorithm, we explain it in the case of binary classification, meaning we have only two classes. As we mentioned, this algorithm has a similar formulation as linear regression. What we want to do is to model yi as a linear function of xi, but that is not as simple now when you can have only two values (two classes, remember?). So, the Logistic Regression model still computes a weighted sum of the input features and adds a bias term, but instead of outputting the result directly, it does some extra processing.

That is why we assign the value 0 to the first class (**negative class**) and the value 1 to the second class (**positive class**). That is how the problem is transformed into the problem of finding a **continuous** function whose codomain is (0, 1). This means that we want to estimate the **probability** that an observed sample belongs to a particular class. For that purpose, **sigmoid function** or standard logistic function is used:

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

So, with *Logistic Regression,* we still calculate *wX + b* (or to simplify it even further and put all parameters into the matrix – *θX)* value and put the result in the sigmoid function. If the **result** is greater than 0.5 (probability is larger than 50%), then the model predicts that the instance belongs to positive class(1), or else it predicts that it does not belong to it (negative class). Mathematically we can put it like this:

$$p = h_\theta(X) = \sigma(\theta^T X)$$

It is important to note that we need to modify the **loss function** as well in order for it to work on this type of data. For this purpose, we use the **log loss** function which is defined like this:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)log\left(1 - \hat{p}^{(i)}\right)\right]$$

Unlike the loss function that we used for the **Linear Regression**, this formula doesn't have its closed form. We can not use the *Normal Equation*, so we need to use the **gradient descent** to optimize it. For that purpose, we need to calculate partial derivatives of the cost function with regards to the jth model parameter *θj*:

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T \cdot \mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Ok, that would be the rough **theory** behind it. Let's move on to implementation.

2.2.1.1 Preparing data for Logistic Regression

Let's first load and prepare data for *Logistic Regression*. We perform **binary** classification, so we don't use all data from the *PalmerPenguins* dataset. Here is the complete data when we load it:

```
data = pd.read_csv('./data/penguins_size.csv')
data.head()
```

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

We remove all samples that are labeled with class '*Chinstrap*' and features that we don't want to use (we use only *culmen_length_mm* and *culmen_depth_mm*).

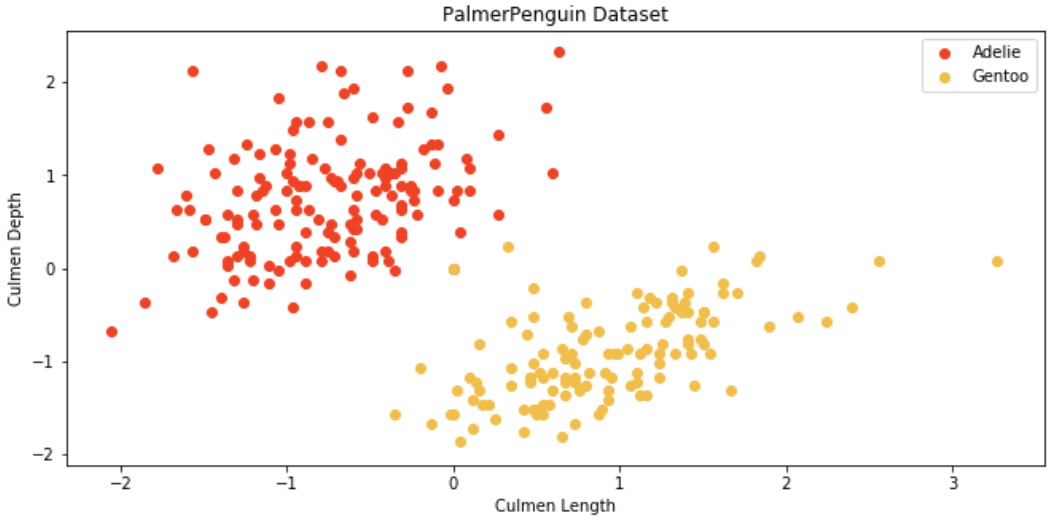
```
data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)
data = data[data['species'] != 'Chinstrap']
```

Ok, now the data is ready so we can define the input data, output data and **split** it into train and test sets.

```
X = data.drop(['species'], axis=1)
X = X.values
ss = StandardScaler()
X = ss.fit_transform(X)

y = data['species']
species = {'Adelie': 0, 'Gentoo': 1}
y = [species[item] for item in y]
y = np.array(y)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

Note that we **scale the** input data using the *StandardScaler*. For more transparency, here is how the input data and output data looks like when we **plot** it:

```
plt.figure(figsize=(11, 5))
plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='orange', label='Adelie')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='gray', label='Gentoo')
plt.legend();
```



2.2.1.1 Logistic Regression Python Implementation

Let's implement a class for *Logistic Regression* from scratch using *Python* and *NumPy*. The implementation is in the class *MyLogisticRegression*:

```
class MyLogisticRegression():
    '''Implements algorithm for Logistic Regression'''
    def __init__(self, learning_rate=0.1):
        self.learning_rate=learning_rate

    def __sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def __loss(self, p, y):
        return (-y * np.log(p) - (1 - y) * np.log(1 - p)).mean()

    def __extend_input(self, X):
        ones = np.ones((X.shape[0], 1))
        return np.concatenate((ones, X), axis=1)

    def __gradient(self, X, p, y):
        return np.dot(X.T, (p - y)) / y.size

    def fit(self, X, y, epochs):
        X = self.__extend_input(X)

        self.theta = np.zeros(X.shape[1])

        for i in range(epochs):
            z = np.dot(X, self.theta)
            h = self.__sigmoid(z)

            loss = self.__loss(h, y)

            if(i % 10000 == 0):
                print(f'Epoch {i} - Loss: {loss} \t')

            self.theta -= self.learning_rate * self.__gradient(X, h, y)

    def predict(self, X):
        X = self.__extend_input(X)
        probability = self.__sigmoid(np.dot(X, self.theta))

        return probability.round()
```

Since we use gradient descent for **optimization**, we need to initialize the learning rate through the **constructor**. Apart from that, there are **four** internal functions:

- **__sigmoid** – Performs sigmoid function on the input information.
- **__loss** – Calculates log loss
- **__gradient** – Calculates the gradient based on the input data, estimated values and actual class label
- **__extend_input** – Just like for Linear regression, we need to extend input data with ones so we can add it automatically as the bias term.

The two public methods, **fit** and **predict,** are used to train the model and to create new predictions for new samples. In the *fit* method, we first **extend** the input and initialize *theta* – the set of parameters. Then we **calculate** the *theta* based on the input data and use them along with the *sigmoid* function to calculate the predictions. After that, we can calculate the *loss*. Finally, we calculate the gradient and use the learning rate to adjust *theta,* i.e., we perform gradient descent. The *predict method* just extends input data and gets the **probability** based on the current parameters.

Finally, let's use the class on prepared data:

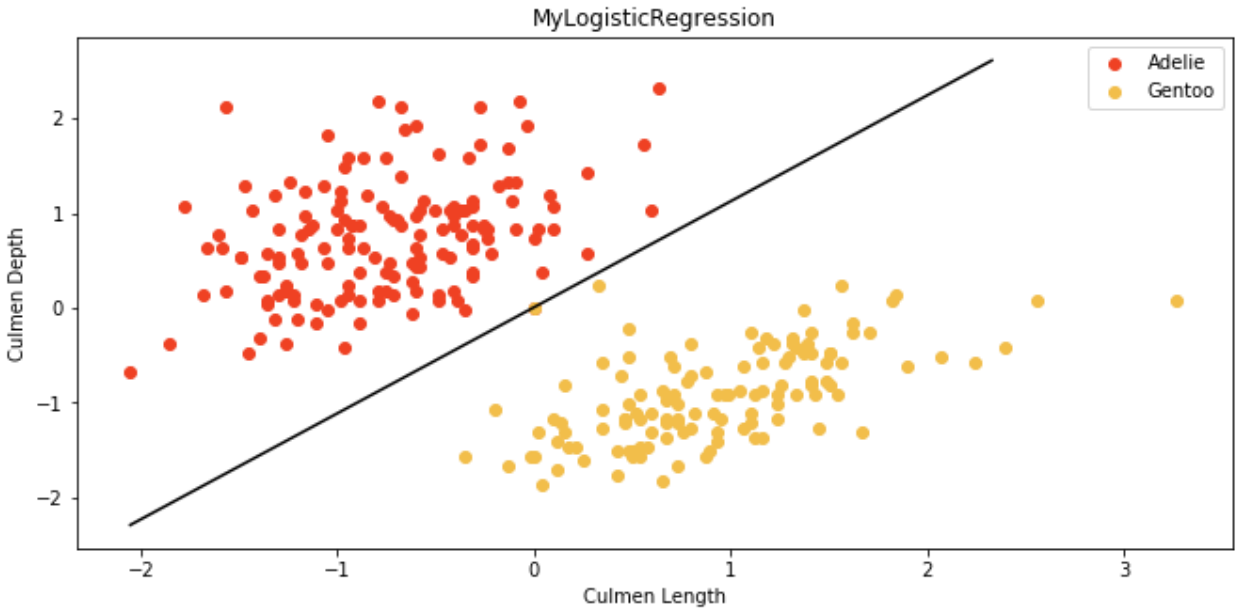
```
model = MyLogisticRegression()
model.fit(X_train, y_train, 200000)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        26
           1       1.00      1.00      1.00        29

    accuracy                           1.00        55
   macro avg       1.00      1.00      1.00        55
weighted avg       1.00      1.00      1.00        55
```

It's simple. First we create an **object** of the *MyLogisticRegression* class, then we call the *fit* method with the training data and call the *predict* method on the test data. We call

the *SciKit Learn* accuracy score, which just gets a percentage of the times prediction aligns with the real value. We had a 100% score, meaning we correctly predicted all values in the test set. Let's visualize that:

```
plt.figure(figsize=(11, 5))
plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='orange', label='Adelie')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='gray', label='Gentoo')
plt.legend()

x_values = [np.min(X[:, 0]), np.max(X[:, 1])]
y_values = - (model.theta[0] + np.dot(model.theta[1], x_values)) /
model.theta[2]

plt.plot(x_values, y_values, color='black');
```



We can even see our linear model presented as a line that separates two classes. This is exactly what we wanted.

2.2.1.2 Using SciKit Learn

Of course, we don't need to implement this algorithm on our own. *Logistic regression* is available as a module in the *SciKit Learn* library. Here is how you can use it:

```
model = LogisticRegression(C=1e20)
model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

```
print(metrics.classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        26
           1       1.00      1.00      1.00        29

    accuracy                           1.00        55
   macro avg       1.00      1.00      1.00        55
weighted avg       1.00      1.00      1.00        55
```

It's pretty much like our from-scratch implementation. The accuracy is again 100%. The principle we used here can be extended to more classes and features.

2.2.2 K-Nearest Neighbours (KNN)

Unlike *Logistic Regression*, this algorithm is not calculating probabilities but is based on **distances**. This effectively means that those are non-parametric models, but it also means that they keep all training data in memory after the training. In fact, **storing** training data is the training process. Basically, once a new previously unseen sample is passed into the algorithm, it calculates *k* training examples that are closest to *x* and returns the majority label (or average label, depending on the implementation). The distances can be calculated in different ways. The **Euclidean distance** or the **Cosine similarity** are often used in practice, but you can play around with the *Manhattan distance* or the *Chebychev distance.* In this chapter, we use the Euclidean distance which can be described with the formula:

$$d(x, y) = \sqrt{\sum_{i=1}^{k} y_i - x_i}$$

To sum up, this algorithm is simple and intuitive and it can be broken down into several steps:

- **Decide** the number of neighbors that the algorithm considers
- **Store** training data with the corresponding labels in memory
- Once a new input point comes in, **calculate** its distance from the training points based on the distance function of your choosing
- **Sort** the results and **pick** *k* points that are closest to the new input sample
- **Detect** the label of the majority of *k* points and assign this label to a new input sample

Let's implement KNN with *Python*.

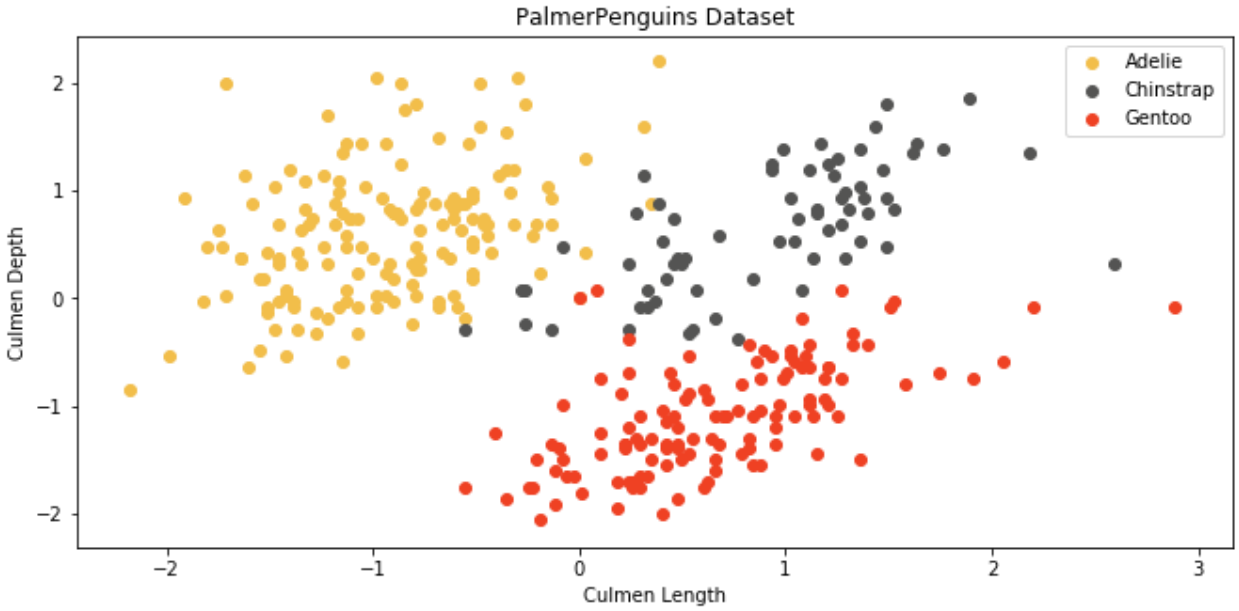
2.2.2.1 Preparing Data for KNN

For this algorithm, we don't consider binary classification but the full classification problem with **three** classes from *PalmerPenguins* dataset. Effectively, this only means that we don't remove the *Chinstrap* class from the dataset; the other preprocessing steps remain the same.

```
data = pd.read_csv('./data/penguins_size.csv')
ss = StandardScaler()

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)

# Prepare input
X = data.drop(['species'], axis=1)
columns = X.columns
X = X.values
X = ss.fit_transform(X)

# Prepare target
y = data['species']
species = {'Adelie': 0, 'Chinstrap': 1, 'Gentoo': 2}
y = [species[item] for item in y]
y = np.array(y)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

Here is what that looks like when we **plot** it:

```
plt.figure(figsize=(11, 5))
plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='orange', label='Adelie')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='black', label='Chinstrap')
plt.scatter(X[y == 2][:, 0], X[y == 2][:, 1], color='gray', label='Gentoo')
plt.legend();
```



2.2.2.2 KNN Python Implementation

The whole algorithm is **implemented** within the *MyKNearestNeighbors* class:

```
class MyKNearestNeighbors():
    '''Implements algorithm for K-Nearest Neighbors'''
    def __init__(self, num_neighbors=5, num_clasess=3):
        self.num_neighbors = num_neighbors
        self.num_clasess = num_clasess

    def __euclidian_distance(self, a, b):
        return np.sqrt(np.sum((a - b)**2, axis=1))

    def __get_neighbours(self, X_test, return_distance=False):
        neighbours = []

        test_train_distances = [self.__euclidian_distance(x_test, self.X_train)
                                \ for x_test in X_test]

        for row in test_train_distances:
            enumerated_neighbours = enumerate(row)
            sorted_neighbours = sorted(enumerated_neighbours,
                                       key=lambda x: x[1]) \ [:self.num_neighbors]

            index_list = [tup[0] for tup in sorted_neighbours]

            neighbours.append(index_list)
```

```
        return np.array(neighbours)

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        neighbors = self.__get_neighbours(X_test)
        predictions = np.array([
            np.argmax(np.bincount(self.y_train[neighbor]))
            for neighbor in neighbors
        ])

        return predictions
```

There are two private methods:

- **__euclidian_distance** – Calculates the Euclidean distance between two samples in the dataset
- **__get_neighbours** – Calculates the distance between all points in the train set and the input set, and returns k neighbors for each sample from the input set

The two public methods, *fit* and *predict,* are used as an interface to this class. The first one just **stores** training data in the class attribute. The *predict* method puts it all together. Basically, it calls the *__get_neighbours* method and utilizes that result to assign labels to every input set sample. The usage is the same as for previous algorithms:

```
model = MyKNearestNeighbors()
model.fit(X_train, y_train)

my_knn_predictions = model.predict(X_test)
print(metrics.classification_report(y_test, my_knn_predictions))
```

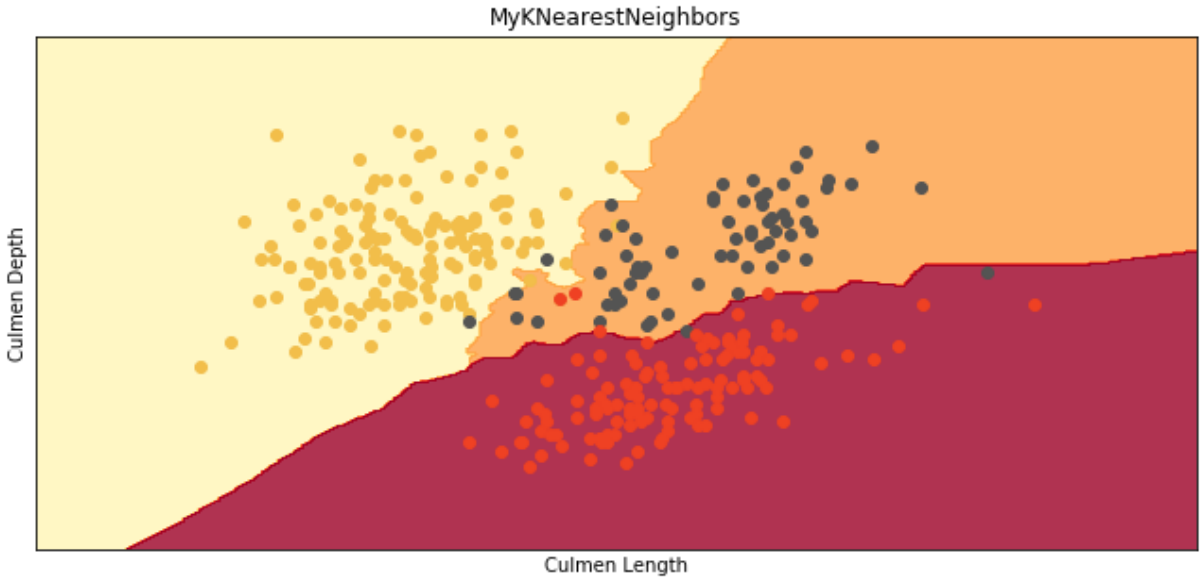
```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98        25
           1       0.83      1.00      0.91        10
           2       1.00      0.97      0.99        34

    accuracy                           0.97        69
   macro avg       0.94      0.98      0.96        69
weighted avg       0.98      0.97      0.97        69
```

The accuracy score is once again 100%. We can confirm it by creating the *pandas* data frame with both actual values and predictions:

```
pd.DataFrame({
    'Actual Value': y_test,
    'KNN Predictions': predictions,
})
```

	Actual Value	KNN Predictions
0	1	1
1	2	2
2	0	0
3	2	2
4	0	0
...	...	...
62	2	2
63	2	2
64	2	2
65	0	0
66	2	2

It is also interesting to see how this algorithm observes ths feature space, i.e. how it would classify all the points from the dataset range. So, we can predict the class for all the possible values of both features and plot it:



In this plot, we can see the "regions" for each class as seen by our algorithm. Also, we can see real values and how this algorithm made several bad predictions on some edge cases. This is really cool.

2.2.2.3 Using SciKit Learn

The *SciKit Learn*, of course, provides a class for KNN called *KNeighborsClassifier:*

```
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

print(model.score(X_test, y_test))
```

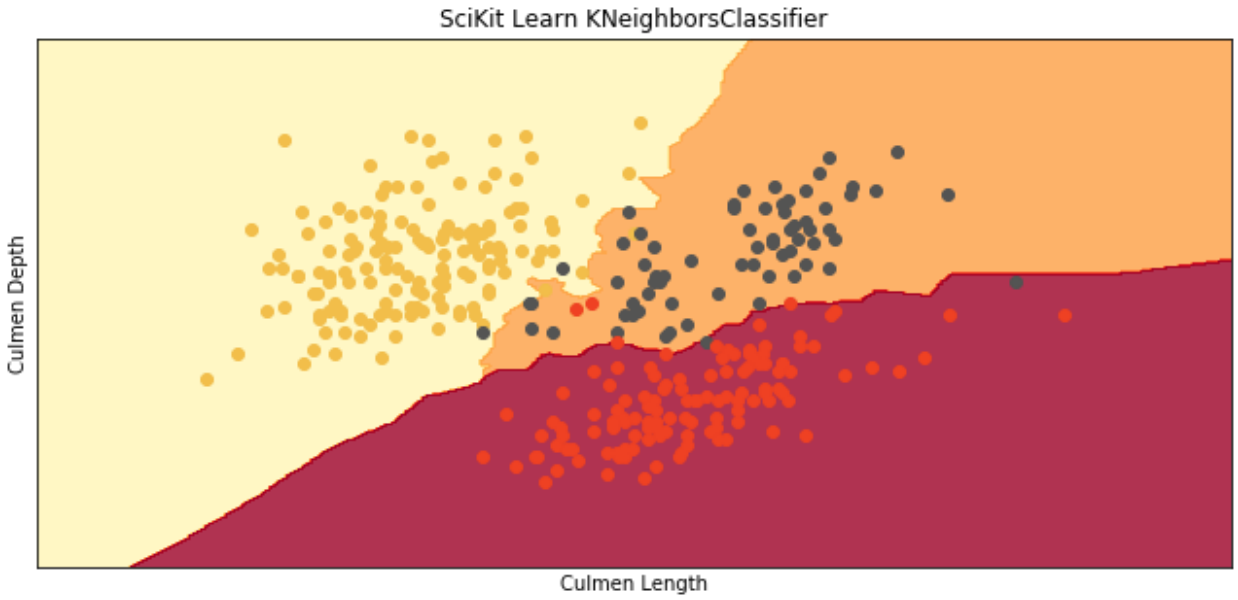
```
0.9710144927536232
```

```
sk_knn_predictions = model.predict(X_test)

print(metrics.classification_report(y_test, sk_knn_predictions))

pd.DataFrame({
    'Actual Value': y_test,
    'KNN Predictions': my_knn_predictions,
    'SciKit Learn KNN': sk_knn_predictions,
})
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98        25
           1       0.83      1.00      0.91        10
           2       1.00      0.97      0.99        34

    accuracy                           0.97        69
   macro avg       0.94      0.98      0.96        69
weighted avg       0.98      0.97      0.97        69
```

	Actual Value	KNN Predictions	SciKit Learn KNN
0	1	1	1
1	2	2	2
2	0	0	0
3	2	2	2
4	0	0	0
...	...	...	...
62	2	2	2
63	2	2	2
64	2	2	2
65	0	0	0
66	2	2	2

The accuracy in this approach is also 97%. This way, we have the same results as with our implementation, which is quite cool. If we plot it out, it looks quite like the first graph that we did with the predictions of our model.



2.2.3 Naive Bayes

The third and final classification algorithm we explore is the **Naive Bayes** algorithm. As we mentioned previously, classification problems can be solved by creating a predictive model. That is what we have done with *Logistic Regression*. Another way to create a predictive model would be to **estimate** the conditional probability of the class label, given the observation.This means we can calculate the conditional probability for each class label and then pick the label with the highest probability as the most likely label. In theory, Bayes Theorem can be used for this:

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

The main problem with this approach is that we need a really large dataset to calculate the **conditional probability** *P(x1, x2, …, xn | yi)*, because this formula assumes that each input variable is dependent upon all other variables. If the number of features is large, the size of the dataset becomes an even bigger problem. To simplify this problem, we **assume** that each input variable is independent of the other. This might sound weird…because it is. In reality, it is really rare that the input features don't **depend** on each other. However, this approach proved to do surprisingly well in the wild. That is why we can rewrite the formula from above as:

$$P(y_i|x_1, x_2, \ldots, x_n) = \frac{P(x_1, x_2, \ldots, x_n|y_i) * P(y_i)}{P(x_1, x_2, \ldots, x_n)}$$

To calculate *P(yi)* all we have to do is divide the frequency of class *yi* in the training dataset and divide it with the total number of samples in the training set *(P(yi) = # of samples with yi / total # of examples)*. The second part of the **equation**, the conditional probability, can be derived from the data as well. So, let's implement it.

2.2.3.1 Preparing Data for Naive Bayes

Before we dive into the algorithm implementation, let's prepare the *PalmerPenguins* dataset again. It is very similar for the preparation we have done for KNN:

```
data = pd.read_csv('./data/penguins_size.csv')
ss = StandardScaler()

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)

# Prepare input
X = data.drop(['species'], axis=1)
columns = X.columns
X = X.values
X = ss.fit_transform(X)

# Prepare target
y = data['species']
species = {'Adelie': 0, 'Chinstrap': 1, 'Gentoo': 2}
y = [species[item] for item in y]
y = np.array(y)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

The only difference is that we stored columns in the *columns* variable because we need them for the algorithm implementation.

2.2.3.2 Python Implementation

The implementation of this algorithm is simple and can be found in the class *MyNaiveBias*. Note that we utilized the *Pandas* library a bit more because it provides easy solutions for **grouping** data:

```
class MyNaiveBias():
    '''Implements algorithm for Naive Bias'''
    def __init__(self, input_columns):
        self.input_columns = input_columns

    def fit(self, X_train, y_train):
        X_train = pd.DataFrame(X_train, columns = self.input_columns)
```

```
        self.classes = np.unique(y_train)
        self.means = X_train.groupby(y_train).apply(np.mean)
        self.stds = X_train.groupby(y_train).apply(np.std)
        self.probabilities = X_train.groupby(y_train).apply(lambda x: len(x)) /
X_train.shape[0]

    def predict(self, X_test):
        X_test = pd.DataFrame(X_test, columns = self.input_columns)
        predictions = []

        for i in range(X_test.shape[0]):
            p = {}

            for c in self.classes:
                p[c] = self.probabilities[c]

                for index, row in enumerate(X_test.iloc[i]):
                    p[c] *= norm.pdf(row, self.means.iloc[c, index],
self.stds.iloc[c, index])

            predictions.append(pd.Series(p).values.argmax())

        return predictions
```

So, the fit method is calculating **statistics** for the training dataset. We store all classes, calculate means and standard deviations for the input training data grouped by each class. Finally, we calculate the prior probability for each class. The *predict method* is doing all the hard work here and utilizes those calculations. First, for each sample from the input set and for each class, we calculate the **conditional probability** and multiply it by the **prior probability** for that class. In the end, for each input sample, we pick the class with the **largest** probability. We use this class as the ones before:

```
model = MyNaiveBias(columns)

model.fit(X_train, y_train)
my_naive_predictions= model.predict(X_test)

print(metrics.classification_report(y_test, my_naive_predictions))

pd.DataFrame({
    'Actual Value': y_test,
    'Naive Bias Predictions': my_naive_predictions,
})
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        25
           1       0.83      1.00      0.91        10
           2       1.00      0.94      0.97        34

    accuracy                           0.97        69
   macro avg       0.94      0.98      0.96        69
weighted avg       0.98      0.97      0.97        69
```

	Actual Value	Naive Bias Predictions
0	1	1
1	2	2
2	0	0
3	2	2
4	0	0
...	...	...
62	2	2
63	2	2
64	2	2
65	0	0
66	2	2

2.2.3.3 Using SciKit Learn

We can use *SciKit Learn* implementation of Naive Bayes like this:

```
model = GaussianNB()
model.fit(X_train, y_train)
sk_nb_preditions = model.predict(X_test)

print(accuracy_score(sk_nb_preditions, y_test))

pd.DataFrame({
    'Actual Value': y_test,
    'Naive Bias Predictions': predictions,
    'SciKit Learn NB': sk_nb_preditions,
})
```

	Actual Value	Naive Bias Predictions	SciKit Learn NB
0	1	1	1
1	2	2	2
2	0	0	0
3	2	2	1
4	0	0	0
...	...	...	...
62	2	2	2
63	2	2	2
64	2	2	2
65	0	0	0
66	2	2	2

2.3 Regression and Classification Algorithms

In this chapter, we explore some powerful algorithms that can be used for both regression and classification. These include the Support Vector Machines and the Decision Trees. These are some of the most popular algorithms in the world of machine learning.

2.3.1 SVM

Let's first explore how this algorithm works for a simple binary classification in order to understand how it functions. This means that we will consider only two classes from the PalmerPenguins dataset. Like other machine learning algorithms, SVM observes every feature vector as a point in a high-dimensional space. In its core, SVM puts all feature vectors on an imaginary n-dimensional plot and draws an imaginary n-dimensional line (a hyperplane). This line separates examples with positive labels from examples with negative labels in the case of classification or collects as many samples as possible in case of regression. The hyperplane is defined by the function:

$$wx + b = 0$$

where *x* is the **feature vector**, *w* is a feature **weights vector** with the size same as *x*, and *b* is a bias term. This formula should be familiar from our journey through **Linear Regression** or **Logistic Regression**. In the case of binary classification, which we are considering at the moment, SVM **requires** that the positive label has a numeric value of

1, and the negative label has a value of -1. This means that the predicted label for a feature vector *x* can be calculated using the formula:

$$y = sign(wx - b)$$

The function *sign* returns value 1 if the input is positive and -1 if the input is a negative value. So, the SVM **model**, which during the training process should optimize *w* and *b*, can be described with the formula:

$$f(x) = sign(wx - b)$$

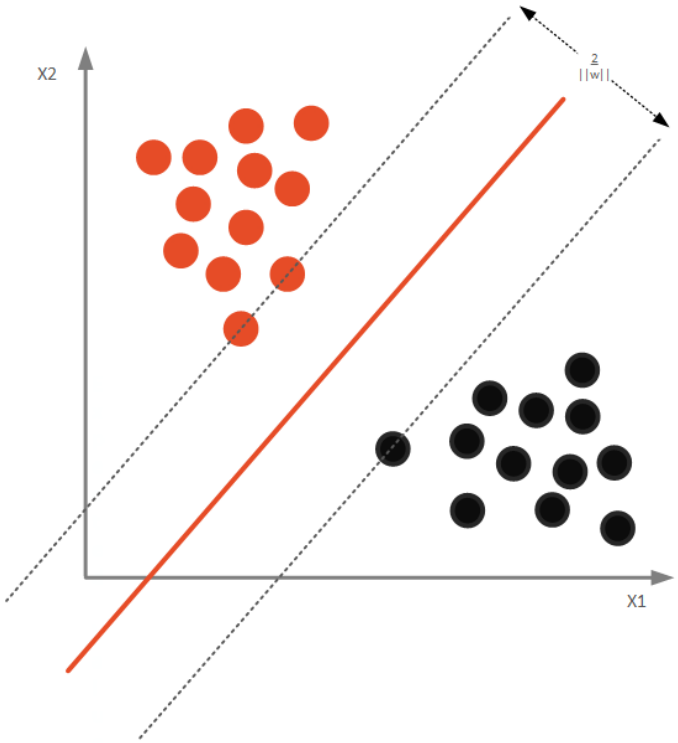
We can break this down and write it as:

$$f = sing(w_1 * x_1 + w_2 * x_2 + \ldots + w_n * x_n - b)$$

This all looks very similar to the Logistic Regression approach. To better understand what the difference from **Logistic Regression** is, let's consider **constraints** under which the algorithm operates:

$$w * xi - b >= 1 \;\; if \;\; yi = +1$$
$$w * xi - b < 1 \;\; if \;\; yi = -1$$

On a graph that looks like this:



This means that SVM doesn't only create a hyperplane but it also constructs **additional** vectors which are defining the **margin**. These vectors are called s**upport vectors**, and the distance between the closest examples of two classes is called the **margin**. The hyperplane with the support vectors is often referred to as the **street**. This means that SVM tries to **fit** the best street between the samples of different classes. Unlike the Logistic regression, which tries to fit the hyperplane as close as possible to these points, SVM tries to fit the hyperplane that is as **far** as possible from the samples but still separates classes successfully.

Note that a large margin leads to a better **generalization**, meaning the model will better classify new samples. However, notice that the margin is decided by the Euclidean norm of *w* (denoted by ||*w*|| in the image). This effectively means the smaller the weight vector *w*, the larger the margin and thus better generalization. To sum up, training the SVM algorithm for classification means finding the value of *w* and *b* that makes the margin as wide as possible while avoiding misclassification. Therefore, the objective is defined as a constrained optimization problem:

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

where *t(i)* is –1 for negative samples and *t(i)* = *1* for positive samples. To be more precise, this optimization problem is a **convex quadratic optimization** problem with linear constraints. Such problems are known as **Quadratic Programming**(QP) problems. The solution for this type of problem is outside of the scope of this book.

2.3.1.1 Preparing Data for Classification

As we mentioned, we use *PalmerPenguins* dataset for the classification examples. However, since we want to do binary classification, we need to do some **preparations**. First, we load the dataset, remove features that we don't use in this article and remove the one class because we perform binary classification:

```
data = pd.read_csv('./data/penguins_size.csv')
data.head()
```

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

We remove all samples that are labeled with the class '*Chinstrap*' and features that we don't want to use (we use only *culmen_length_mm* and *culmen_depth_mm*).

```
data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)
data = data[data['species'] != 'Chinstrap']
```

Then we separate the input data and scale it:

```
X = data.drop(['species'], axis=1)
X = X.values
ss = StandardScaler()
X = ss.fit_transform(X)
```

After that, we extract the output values and mark them with values -1 and 1, since the SVM algorithm requires that.

```
y = data['species']
species = {'Adelie': -1, 'Gentoo': 1}
y = [species[item] for item in y]
y = np.array(y)
```

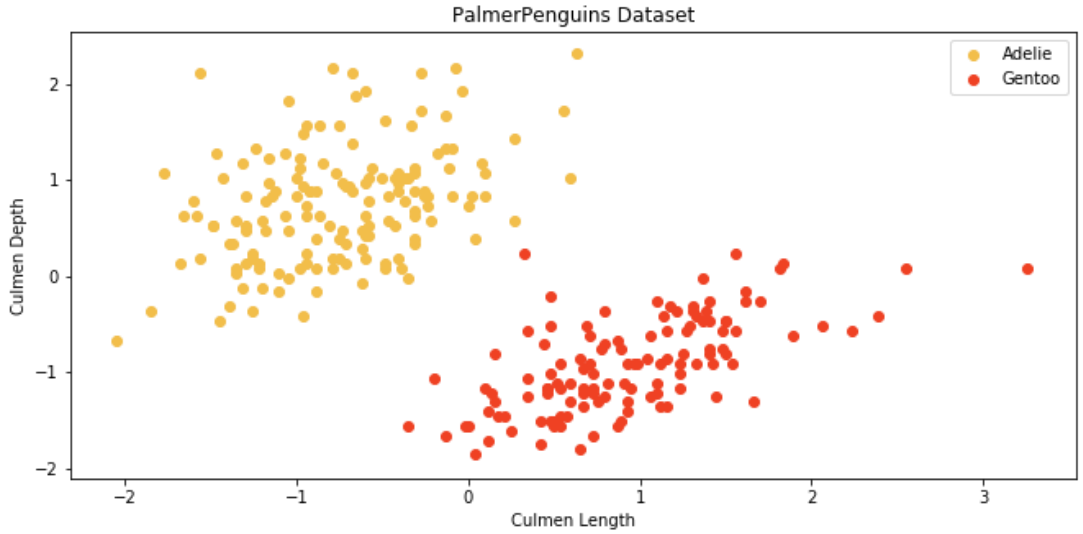
Another thing that we remove is the 182nd sample from the dataset. Why? Well, this sample was in between classes and it kinda messed with points that we were trying to make, so we remove it and split the data into training and test datasets:

```
# Remove sample that is too close
X = np.delete(X, 182, axis=0)
y = np.delete(y, 182, axis=0)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

That's it, our dataset is ready and here is how it looks when we **plot** it:

```
plt.figure(figsize=(11, 5))
plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='orange', label='Adelie')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='gray', label='Gentoo')
plt.legend();
```



2.3.1.2 Python Implementation

There are many methods to find the optimal *w* and *b* for the SVM. One of the most popular ones is the **Sequential Minimal Optimization** (SMO) which is used by the *SciKit Learn* as well. In its core, the *SMO* algorithm splits the quadratic programming optimization problem into smaller ones. However, this algorithm is rather complicated, so in this book, we implement a simpler one – **The Pegasos algorithm**. This algorithm uses stochastic gradient descent, and it is defined like this:

INPUT: S, λ, T
INITIALIZE: Set $\mathbf{w}_1 = 0$
FOR $t = 1, 2, \ldots, T$
  Choose $i_t \in \{1, \ldots, |S|\}$ uniformly at random.
  Set $\eta_t = \frac{1}{\lambda t}$
  If $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1$, then:
    Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \eta_t y_{i_t} \mathbf{x}_{i_t}$
  Else (if $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle \geq 1$):
    Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t$
  [Optional: $\mathbf{w}_{t+1} \leftarrow \min \left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$]
OUTPUT: $\mathbf{w}_{T+1}$

So let's implement this algorithm within the *MySVM* class:

```
class MySVM():
    def __init__(self, number_of_features, learning_rate=0.001):
        self.w = np.zeros(number_of_features + 1)
        self.learning_rate = learning_rate
        pass

    def __extend_input(self, X):
        ones = np.ones((X.shape[0], 1))
        return np.concatenate((ones, X), axis=1)

    def fit(self, X_train, y_train, epochs):
        X_train = self.__extend_input(X_train)

        for epoch in range(1, epochs):
            eta = 1/(self.learning_rate * epoch)
            fac = (1 - (eta * self.learning_rate)) * self.w

            for i in range(1, X_train.shape[0]):
                prediction = np.dot(X_train[i], self.w)

                if (y_train[i] * prediction) < 1 :
                    self.w = fac + eta * y_train[i] * X_train[i]
                else:
                    self.w = fac


    def predict(self, X_test):
        X_test = self.__extend_input(X_test)
        predictions = []

        for x in X_test:
            prediction = np.dot(self.w, x)
            prediction = 1 if (prediction > 0) else -1
            predictions.append(prediction)
        return np.array(predictions)
```

The implementation is fairly simple. In the constructor, we receive two **parameters**, the number of features and the learning rate. We need the number of features in order to initialize *w*. Note that we add 1 to that size because *b* is incorporated within *w*. That is why we need a private *__extend_input* method. This method is used to **extend** any input matrix with the column of ones. That is how *b* is implicitly modeled within the solution. Apart from that, this class contains two **public** methods, *fit* and *predict*, following the blueprint dictated by big libraries like *SciKit Learn*.

The fit method handles the training process. In it, we first extend the training set and loop for a defined number of **epochs**. There we perform necessary calculations defined by the *Pegasos* algorithm. Finally, we update *w* for **each** sample in the training set. The predict method just extends the test set and multiplies it with *w*. If the result is larger than 0 it appends label 1; otherwise, it appends label -1. As you may notice, unlike *Logistic Regression*, SVM doesn't output **probabilities** for each class. It is easy to use this class:

```
model = MySVM(X_train.shape[1])
model.fit(X_train, y_train, 10000)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

          -1       0.93      1.00      0.97        28
           1       1.00      0.93      0.96        27

    accuracy                           0.96        55
   macro avg       0.97      0.96      0.96        55
weighted avg       0.97      0.96      0.96        55
```

In the end, we get that accuracy of 96%.

2.3.1.3 Using SciKit Learn

Of course, we can use the *SciKit Learn* implementation of this classifier. Since our data is linearly separable, we can use *LinearSVC* class:

```
lsvc_model = LinearSVC(C=1, loss="hinge")

lsvc_model.fit(X_train, y_train)

lsvc_predictions = lsvc_model.predict(X_test)
print(metrics.classification_report(y_test, lsvc_predictions))
```

```
              precision    recall  f1-score   support

          -1       0.97      1.00      0.98        28
           1       1.00      0.96      0.98        27

    accuracy                           0.98        55
   macro avg       0.98      0.98      0.98        55
weighted avg       0.98      0.98      0.98        55
```

The hyperparameter *C* is used to control the wideness of the street. A smaller *C* value leads to a wider street; however, it leads to more **margin violations**. This means that samples could end up in the middle of the street or even on the wrong side. For the loss function, we used **hinge** loss. This loss is effective for SVM algorithms. It is defined with the formula: ***max(0, 1 – t)***. Its derivative is –1 if t < 1 and 0 if t > 1. Alternatively, we can do the same thing with the *SVC* class:

```
svc_model = SVC(kernel="linear", C=1)
svc_model.fit(X_train, y_train)

svc_predictions = svc_model.pict(X_test)
print(metrics.classification_report(y_test, svc_predictions))
```

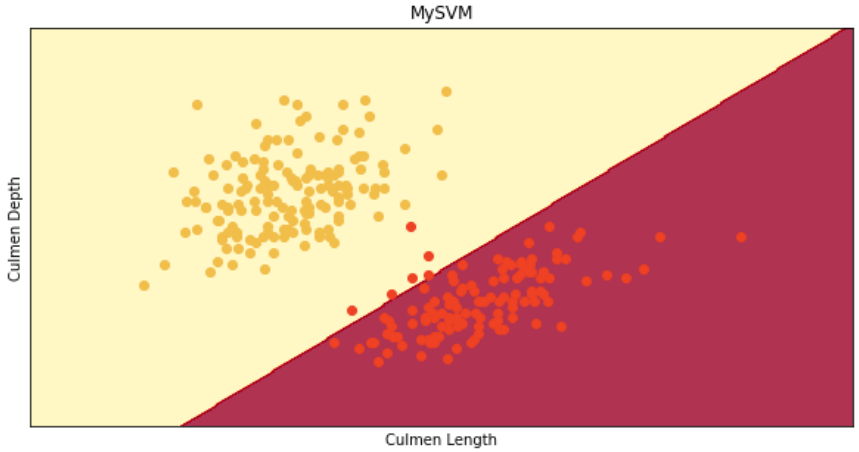
```
              precision    recall  f1-score   support

          -1       0.97      1.00      0.98        28
           1       1.00      0.96      0.98        27

    accuracy                           0.98        55
   macro avg       0.98      0.98      0.98        55
weighted avg       0.98      0.98      0.98        55
```

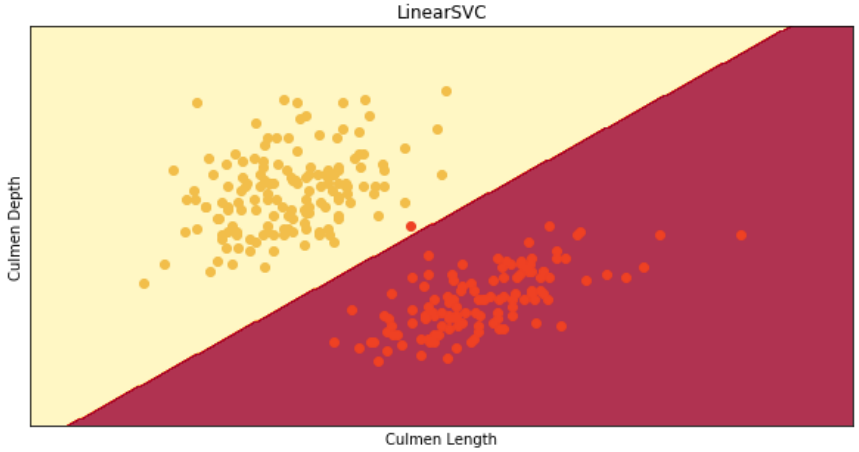
We got the result which is in terms of accuracy a little bit better, i.e., 98% accuracy. Notice the **kernel** hyperparameter, which is going to be explained further in the next chapter. Anyhow, here are the compared results of all three algorithms within pandas data frame:

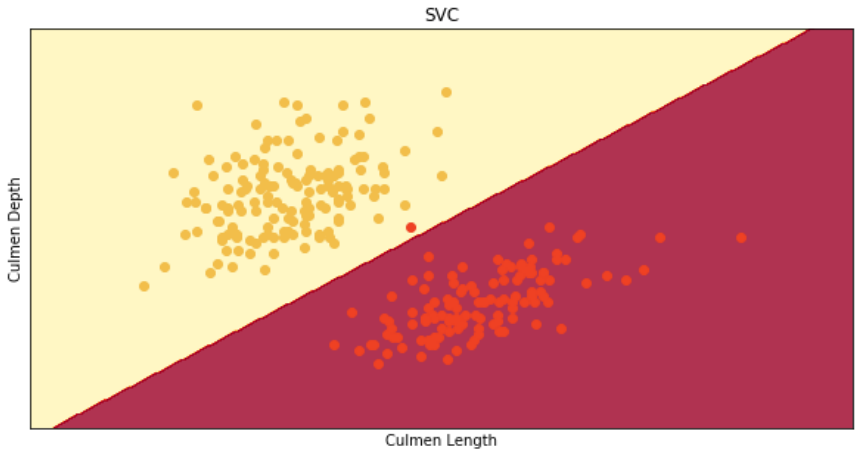
```
pd.DataFrame({
    'Actual Value': y_test,
    'My SVM predictions': predictions,
    'LinearSCV predictions': lsvc_predictions,
    'SVC predictions': svc_predictions,
})
```

	Actual Value	My SVM Predictions	LinearSCV Predictions	SVC Predictions
0	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	-1	-1	-1	-1
5	-1	-1	-1	-1
6	1	1	1	1
7	-1	-1	-1	-1
8	1	1	1	1
9	1	1	1	1
10	-1	-1	-1	-1

If we compare these models, here is what we get:





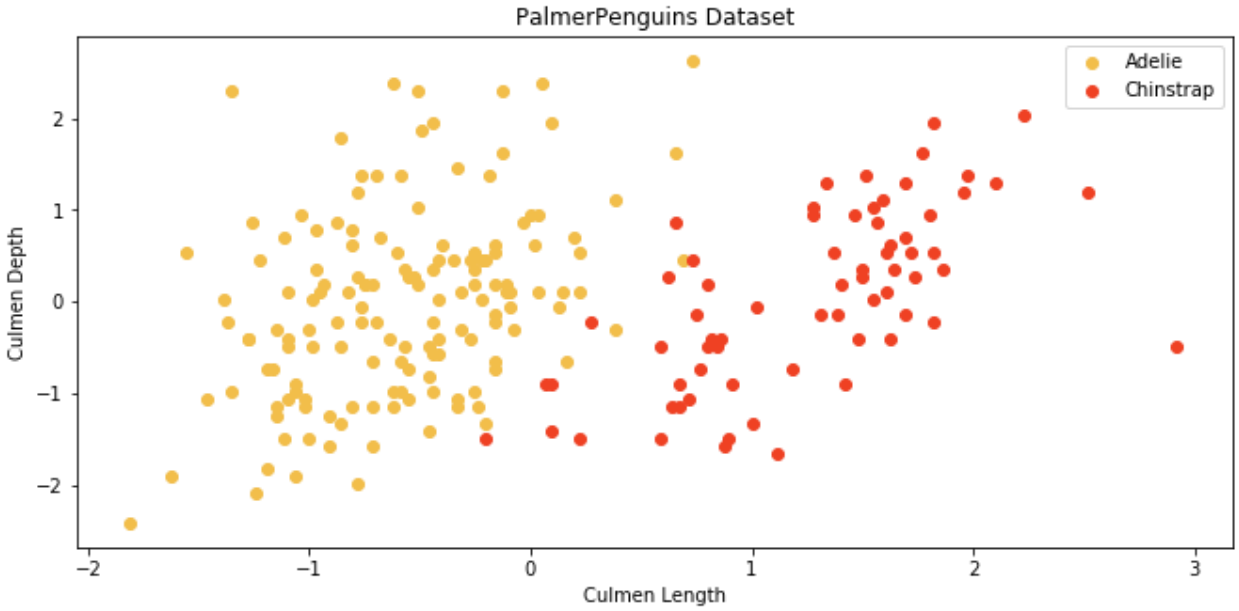


Notice the difference between a linear function that **separates** two classes between our and the *SciKit Learn* implementation. This is caused by different algorithms that implementations use.

2.3.1.3 Non-Linear Data

Thus far we observed a pretty nice example, data that is linearly separable. In reality, this is almost never the case. So, let's consider other classes from the *PalmerPenguins* dataset and load the *Adelie* and the *Chinstrap* classes.

```
data = pd.read_csv('./data/penguins_size.csv')
data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)
data = data[data['species'] != 'Gentoo']

X = data.drop(['species'], axis=1)

ss = StandardScaler()
X = ss.fit_transform(X)

y = data['species']
species = {'Adelie': -1, 'Chinstrap': 1}
y = [species[item] for item in y]
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)

plt.figure(figsize=(11, 5))

plt.title('PalmerPenguins Dataset')
plt.xlabel('Culmen Length')
plt.ylabel('Culmen Depth')

plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1], color=ORANGE, label='Adelie')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color=RED, label='Chinstrap')

plt.legend();
```



This time, it is not so easy to separate classes with just a straight line. The data is a bit scrambled, so what should we do in these situations when the data is not linear? Here we can apply probably the greatest SVM advantage – the **kernel trick**. This technique gives you the possibility to get the same result as if you were using polynomial features **without** actually having to add them. Kernels are just functions that **map** low-dimensional non-linearly-separable data into linearly-separable high-dimensional data. This means, in our case, that we map our 2D data which is not linearly separable into 3D data which is.

However, we don't know which **mapping** works for our data best, so if we would map all the vectors into a higher dimension and then apply SVM to it, that would be very inefficient. That is where the kernel trick comes into play. In essence, it uses kernels to work in **higher-dimensional** spaces without doing this transformation explicitly. So, let's use different kernels on our dataset. First, we use the **polynomial kernel** and here is the result that we get.

```
svc_model_poly = SVC(kernel="poly", C=0.6)

svc_model_poly.fit(X_train, y_train)

svc_poly_predictions = svc_model_poly.predict(X_test)
print(metrics.classification_report(y_test, svc_poly_predictions))
```

```
              precision    recall  f1-score   support

          -1       0.78      1.00      0.88        25
           1       1.00      0.61      0.76        18

    accuracy                           0.84        43
   macro avg       0.89      0.81      0.82        43
weighted avg       0.87      0.84      0.83        43
```

As you can see, the accuracy is around 84%. That is not bad, but can it get better? One of the most popular kernel functions is the Gaussian **RBF** kernel defined by the formula:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right)$$

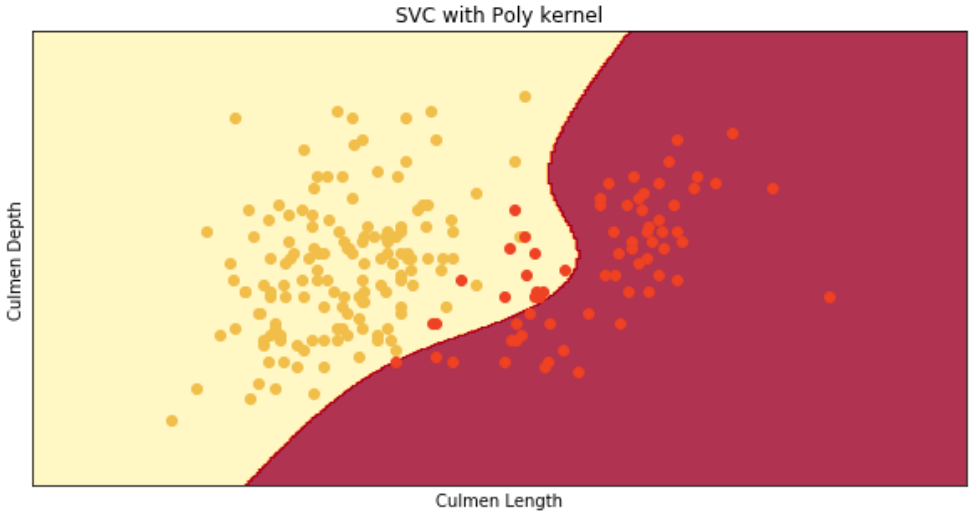
It is a bell-shaped function varying from 0 to 1 and it is often used for adding features using the **similarity features** method. Notice the parameter **gamma**. It defines how wide the bell-curve is and this is essentially the hyperparameter of the SVM. In essence, when we use this kernel, we create a gaussian bell-curve in 3-dimensional space, in this example (since our data is in 2-dimensional space), around the chosen landmark.

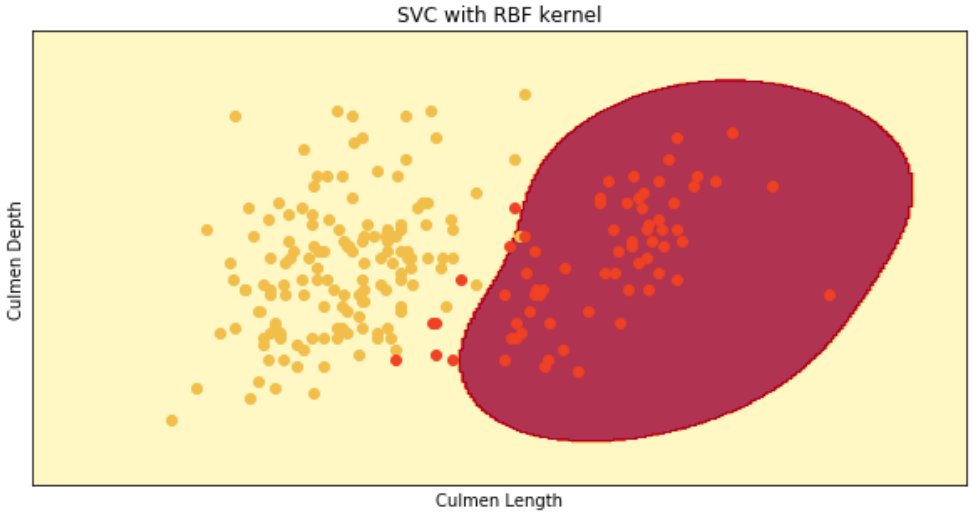
Then, all points are mapped from 2-dimensional space to 3-dimensional space, but every point is **mapped** to this curve. That is how we ensure that the data is linearly separable in 3-dimensional space. Then the SVM is applied. Of course, the kernel *trick* is applied, so we don't have to do all of these calculations, so the algorithm is pretty efficient. Let's use *RBF*:

```
svc_model_rbf = SVC(kernel="rbf", gamma=.5, C=0.1)

svc_model_rbf.fit(X_train, y_train)

svc_rbf_predictions = svc_model_rbf.predict(X_test)
print(metrics.classification_report(y_test, svc_rbf_predictions))
```

```
              precision    recall  f1-score   support

          -1       0.93      1.00      0.96        25
           1       1.00      0.89      0.94        18

    accuracy                           0.95        43
   macro avg       0.96      0.94      0.95        43
weighted avg       0.96      0.95      0.95        43
```

The accuracy is around 95%. That is much better than the polynomial kernel. If we visualize both approaches, we get something like this:





You can imagine a 3D Gaussian bell-curve coming out of your screen in the RBF example if you will.

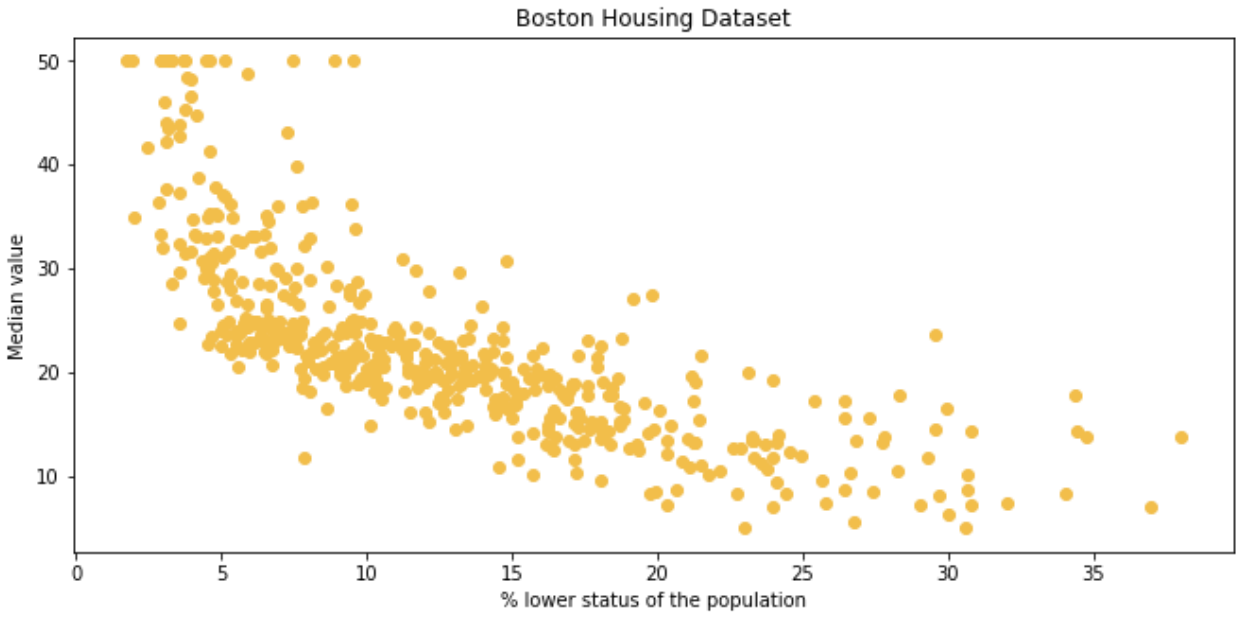
SVM for regression is not so different from the one for classification. In essence, all the **practices** that we learned for classification stand for regression as well, with one major difference. The classification SVM tried to fit the largest street **among** the samples of different classes without violating the margins. In regression, the SVM tries to fit as many samples as possible **on** the street while minimizing the number of samples of the street. The wideness of the street is controlled by the hyperparameter – **epsilon**. Let's check it out.

2.3.1.4 Preparing Data for Regression

For a regression example, we use the **Boston housing dataset**. This dataset contains 14 features, of which we use 2. Our goal in this example is to predict the **value** of the property based on the ***lstat*** feature. So, we remove all other features:

```
data = pd.read_csv('./data/boston_housing.csv')
data.head()

data = data.dropna()

X = data['lstat'].values
X = X.reshape(-1, 1)

y = data['medv'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

Here is what that looks like when we **plot** it:



As you can see, this is the case of **non-linear** regression. So, let's try to create a model that will best fit this data.

2.3.1.5 Using SciKit Learn

Before we proceed, let's implement a function that will print all relevant regression metrics:

```
def print_evaluation_metrics(actual_values, predictions):
    print (f'MAE: {metrics.mean_absolute_error(actual_values, predictions)}')
    print (f'MSE: {metrics.mean_squared_error(actual_values, predictions)}')
    print (f'RMSE: {sqrt(metrics.mean_squared_error(actual_values,
           predictions))}')
    print (f'R Squa: {metrics.r2_score(actual_values, predictions)}')
```

Now, we know that regression in this example is non-linear, but for educational purposes, we start with **linear** SVM regression and progress to more complicated kernels.

```
lsvr_model = LinearSVR(epsilon=1.5, max_iter=10000)
lsvr_model.fit(X_train, y_train)

lsvr_predictions = lsvr_model.predict(X_test)

print_evaluation_metrics(y_test, lsvr_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'LinearSVR prediction': lsvr_predictions,
})
```

	Actual Value	LinearSVR Prediction
0	20.5	16.755463
1	5.6	10.418651
2	13.4	12.913233
3	12.6	17.823546
4	21.2	22.762532
...	...	...
97	25.0	22.576155
98	19.5	16.232174
99	19.9	17.931071
100	15.4	14.232208
101	21.7	20.289455

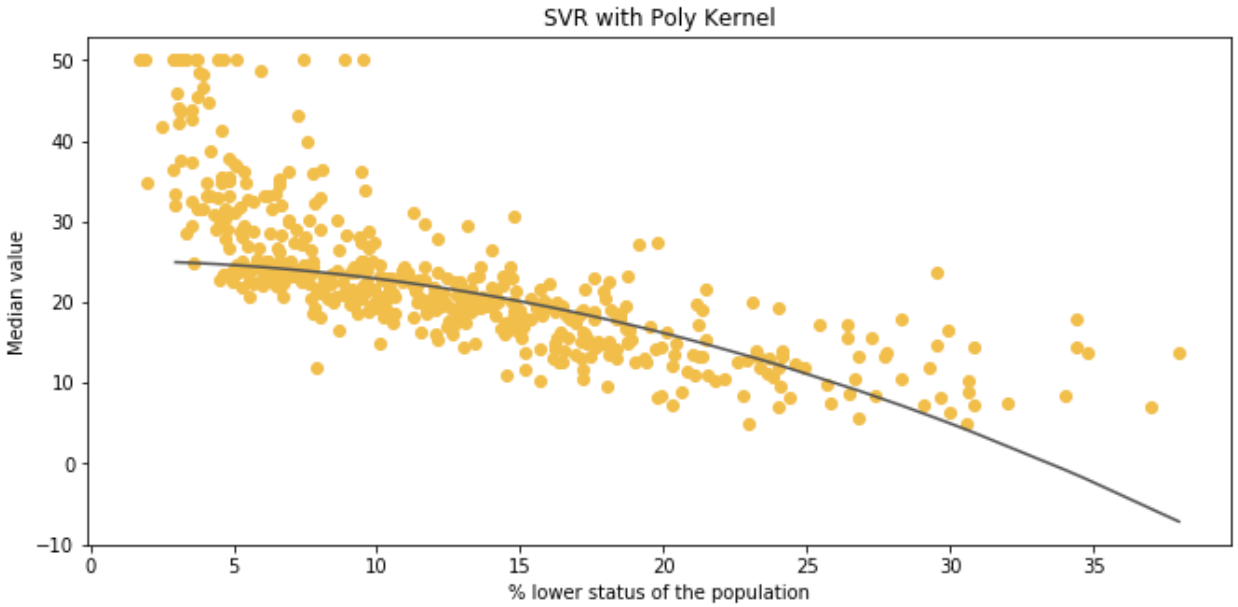
The results are, well, mixed. Sometimes they are not that bad, but sometimes they are far off. It looks like a random guess. When we plot it out, the results make more sense:

Interestingly, we can almost see how the SVM works and how it tried to fit as **many** points around that line. Of course, we can improve the results by using other kernels. Here is how we use polynomial kernel:

```
psvr_model = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
psvr_model.fit(X_train, y_train)

psvr_predictions = psvr_model.pict(X_test)

print_evaluation_metrics(y_test, psvr_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'LinearSVR Prediction': lsvr_predictions,
    'PolySVR Prediction': psvr_predictions
})
```

	Actual Value	LinearSVR Prediction	PolySVR Prediction
0	20.5	16.755463	17.943993
1	5.6	10.418651	9.078521
2	13.4	12.913233	12.987039
3	12.6	17.823546	19.092961
4	21.2	22.762532	23.110569
...	...	...	...
97	25.0	22.576155	22.997636
98	19.5	16.232174	17.344717
99	19.9	17.931071	19.203110
100	15.4	14.232208	14.833993
101	21.7	20.289455	21.365118

Compared to the linear model, the results are somewhat **better**. When we plot the model, here is what we get:

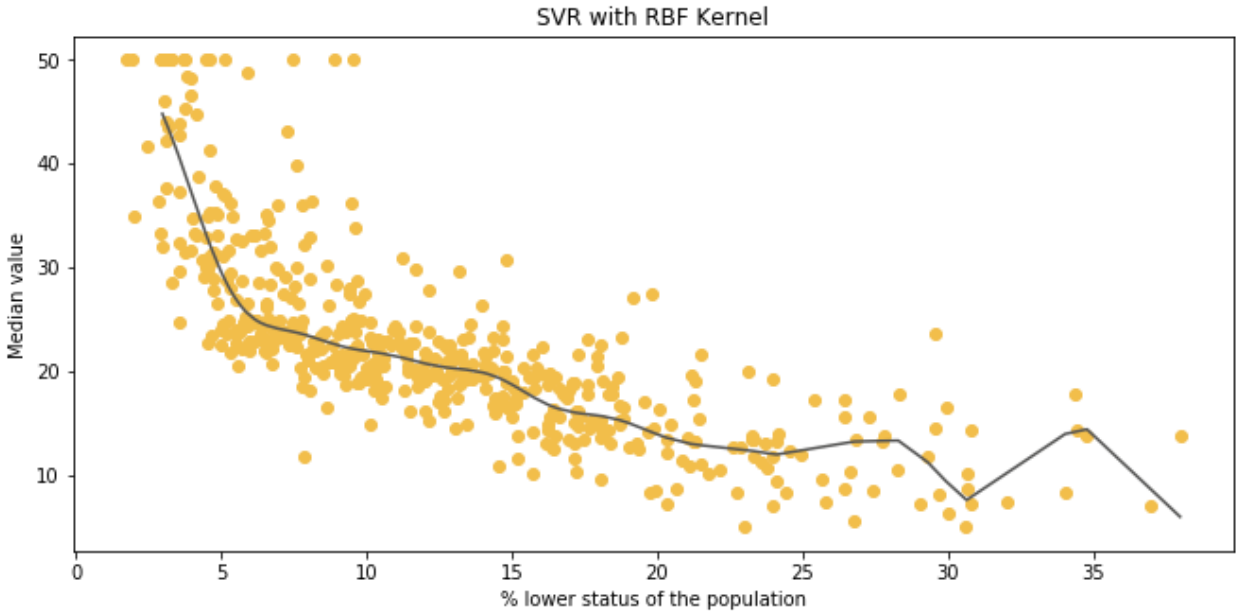


Notice the degree parameter in the **constructor** of the SVR class. Try changing the value of this parameter and the value of epsilon to get better results. Finally, let's use the **RBF** kernel:

```
svr_rbf_model = SVR(C=100, gamma=0.1, epsilon=.1)
svr_rbf_model.fit(X_train, y_train)

svr_rbf_predictions = svr_rbf_model.predict(X_test)

print_evaluation_metrics(y_test, svr_rbf_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'LinearSVR predictions': lsvr_predictions,
    'PolySVR predictions': psvr_predictions,
    'SVR RBF predictions': svr_rbf_predictions
})
```

	Actual Value	LinearSVR Predictions	PolySVR Predictions	SVR RBF Predictions
0	20.5	16.755463	17.943993	15.764753
1	5.6	10.418651	9.078521	13.254520
2	13.4	12.913233	12.987039	12.341407
3	12.6	17.823546	19.092961	16.573468
4	21.2	22.762532	23.110569	22.173334
...	...	...	...	...
97	25.0	22.576155	22.997636	22.046692
98	19.5	16.232174	17.344717	15.350709
99	19.9	17.931071	19.203110	16.741805
100	15.4	14.232208	14.833993	12.914028
101	21.7	20.289455	21.365118	20.303769

This model gets the best results. Let's plot it:

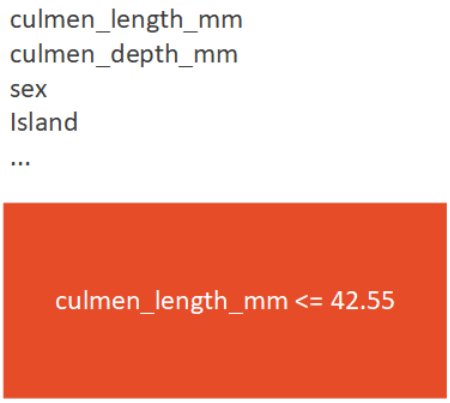


2.3.2 Decision Trees

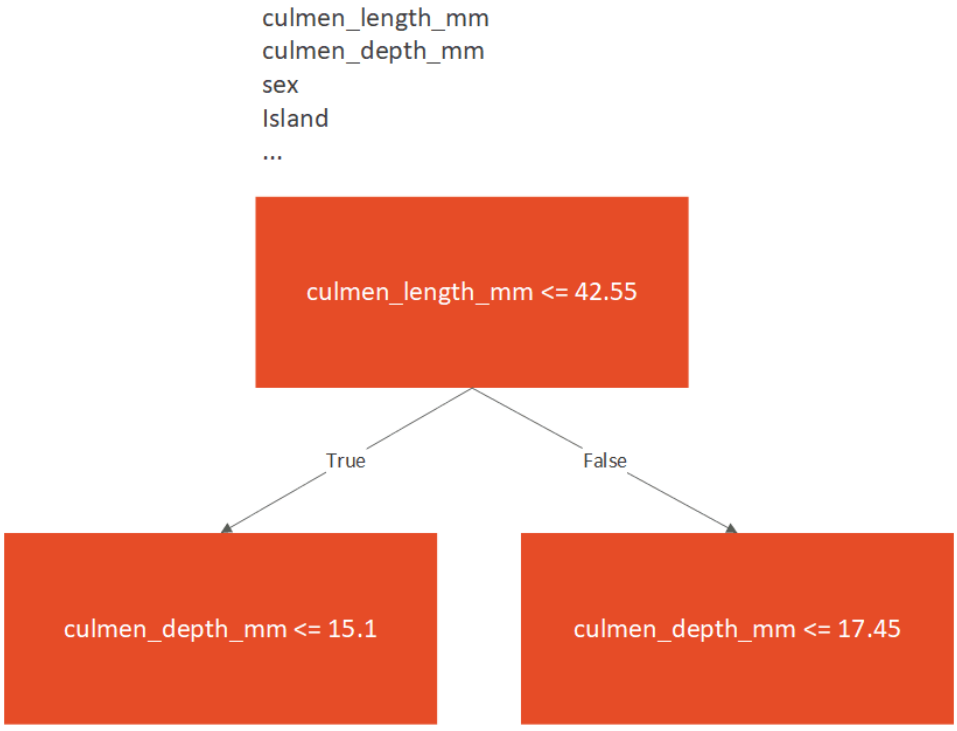
In essence, a *Decision Tree* is a set of algorithms because there are **multiple** ways to solve this problem. Some of the most famous ones are:

- CART
- ID3
- C4.5
- C5.0

In this book, we focus on the **CART** algorithm, which is the easiest and one of the most popular ones. Among others, the *Sci-Kit Learn* library uses this algorithm under the hood. This algorithm produces a **binary tree**, which might not be the case with other algorithms. This means that the node is either branching into two nodes or not branching at all (*terminal node* or *terminal leaf*). Here is the preview of how **CART** builds a *Decision Tree*.



In the beginning, it adds the **root node** of the three, and we push all data to it. In this first node, we examine the value of one of the features. In the example of *PalmerPenguins*, let's say that it examines the *culmen_length_mm* feature and compares it with the chosen **threshold**, in this case, 42.55. Thus, the data is **partitioned** into two sets. The first one for which this question is true and the other one for which it is not. Then two new **nodes** are created which examine some other feature and use some other thresholds:



The process is then **repeated**. The depth of the tree is controlled by the *max_depth* hyperparameter. How are the thresholds created? To understand that, we need to explain two important concepts: **impurity** and **information gain**. Impurity can be defined as a **chance** of being incorrect if you assign a label to an example by random. This means that a node is "pure" if all training instances it applies belong to the same

class; this means that when you assign a label to a random sample, you can not make a mistake. There are different ways for measuring impurity, such as the **Gini index** and **entropy**. In this chapter, we use the *Gini index*. To calculate the *Gini impurity index*, we use the formula:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

where *pi,k* is the ratio of class *k* instances among the training instances in the *i*-th node. For example, if we have 43 instances of the training set in the node, 13 belong to one class, while 30 instances belong to a second class. Given that we have only those two classes in the training dataset, we calculate the *Gini impurity* 1 – (13/43)2 – (30/43)2 ≈ 1 – 0.09 – 0.49 ≈ 0.42. When the node is "pure", its *Gini index* is 0.

On the other hand, the **information gain** lets us find the best **threshold** which will reduce this impurity the most. To calculate the information gain, we need to calculate the **average** impurity and then subtract that from the **starting** impurity. That is how we know the quality of thresholds that we used.

Based on these two concepts, we can define how the *CART* algorithm functions. In its essence, it is a **greedy** algorithm that repeats the process for each level (depth). First, it splits the training dataset into two subsets using a single feature *j* and a threshold *tj*. The feature and the threshold are picked like that so that they produce the purest subsets weighted by their size. The cost function that the CART minimizes can be defined as:

$$J(k, t_k) = \frac{m_l}{m} Gl + \frac{m_r}{m} Gr$$

where *ml* and *mr* represent the number of instances on the respective side (right, left); *m* is the total number of instances and *Gl* and *Gr* represent the *Gini impurity index* on the respective side. Once this is done, it does the same to each subset. The process is repeated **recursively** until the **maximum depth** is reached or a split that reduces impurity can not be found. This algorithm can be used for both regressions and classification. The only difference is that in one case, the resulting decision is the **class** of the sample, while in the other, it is the **value** of the sample. Also, instead of trying to

reduce the *Gini impurity index,* the following is used for regression tasks **MSE** (mean squared error):

$$J(k, t_k) = \frac{m_l}{m} MSE_l + \frac{m_r}{m} MSE_r$$

2.3.2.1 The Python Implementation

As we mentioned for the classification examples, we use *PalmerPenguins* dataset. However, since we want to do binary classification, we need to do some **preparations**. First, we load the dataset, remove features that we don't use in this article, and remove the one class because we perform binary classification:

```
data = pd.read_csv('./data/penguins_size.csv')
data.head()
```

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

We remove all samples that are labeled with the class '*Chinstrap*' and features that we don't want to use (we use only *culmen_length_mm* and *culmen_depth_mm*).

```
data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)
data = data[data['species'] != 'Chinstrap']
```

Then we separate the input data and scale it:

```
X = data.drop(['species'], axis=1)
X = X.values
ss = StandardScaler()
X = ss.fit_transform(X)
```

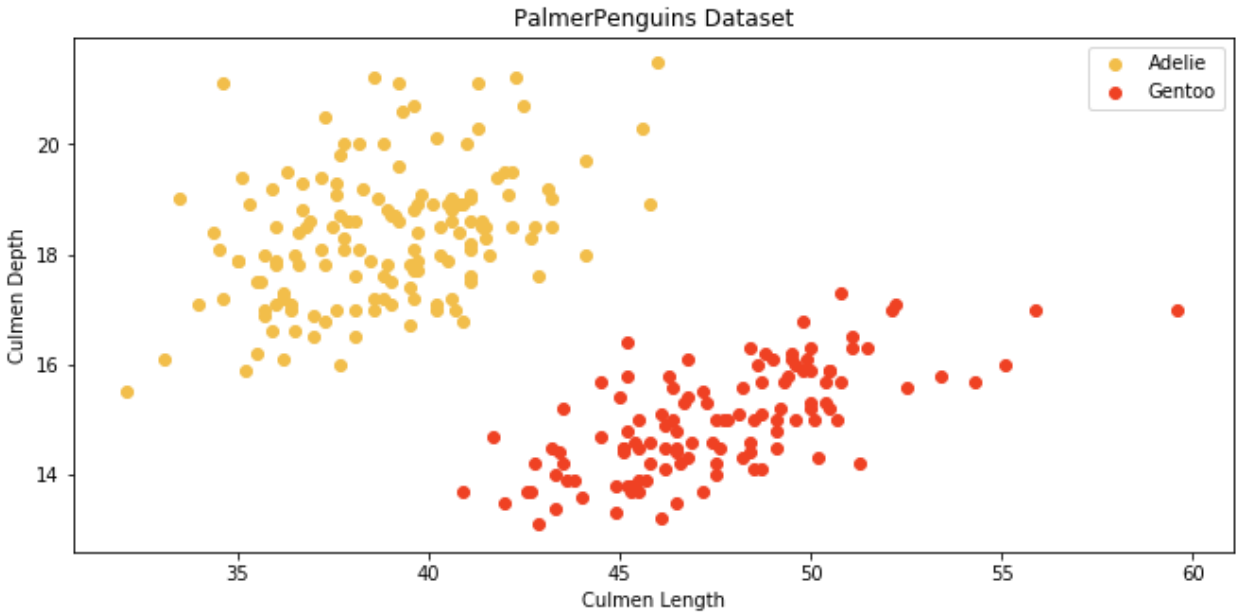
After that, we extract the output values and mark them with values -1 and 1, since the SVM algorithm requires that.

```
y = data['species']
species = {'Adelie': -1, 'Gentoo': 1}
y = [species[item] for item in y]
y = np.array(y)
```

Another thing that we remove is the 182nd sample from the dataset. Why? Well, this sample was in between classes and it kinda messed with points that we try to make, so we remove it and split the data into training and test datasets:

```
# Remove sample that is too close
X = np.delete(X, 182, axis=0)
y = np.delete(y, 182, axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

And that's it, our dataset is ready and here is what it looks like when we **plot** it:



Ok, once that is done, we can implement a helper class *MyGiniCalculator*. This class is used for the *Gini impurity index calculations* we mentioned in the previous chapter:

```
class MyGiniCalculator():

    def _single_class_gini(self, class_count, total_instances_count):
        return -(class_count/total_instances_count)

    def _group_gini(self, class1_count, class2_count):
        if class1_count == 0 or class2_count == 0:
            return 0

        total_count = class1_count + class2_count
        return self._single_class_gini(class1_count, total_count) +
self._single_class_gini(class2_count, total_count)

    def _leaf_gini(self, data):
        gini = 0

        num_istances = len(data)
        classes = set(data)

        for classs in classes:
            class_count = sum(data == classs)
            gini += class_count * 1.0/num_istances * self._group_gini(sum(data
== classs), sum(data != classs))

        return gini, num_istances

    def get_gini(self, is_left, y):
        if len(is_left) != len(y):
            print('Wrong length')
            return None

        num_istances = len(y)
        left_gini, left_count = self._leaf_gini(y[is_left])
        right_gini, right_count = self._leaf_gini(y[~is_left])

        total_gini = left_count*1.0/num_istances * left_gini +
right_count*1.0/num_istances * right_gini

        return total_gini
```

Note that this class has several methods, out of which only one is public:

- **_single_class_gini** – Calculates the *Gini impurity index* for a single class
- **_group_gini** – Since we build a binary tree, this method calculates the *Gini impurity index* for a group of two classes
- **_leaf_gini** – This method calculates the *Gini impurity index* for one leaf
- **get_gini** – This method calculates the *Gini impurity index*

Awesome! This class helps us a lot when it comes to the implementation of the ***MyDecissionTreeClassifier*** class, which contains the implementation of the Decision Tree:

```
class MyDecissionTreeClassifier():
    def __init__(self, feature_names, max_depth = 5):
        self.gini_calculator = MyGiniCalculator()
        self.feature_names = feature_names
        self.max_depth = max_depth
        self.depth = 0

    def _get_best_feature_split(self, feature, y):
        min_gini = 10

        for value in set(feature):
            is_left = feature < value
            gini = self.gini_calculator.get_gini(is_left, y)
            if gini < min_gini:
                min_gini = gini
            cutoff = value

        return min_gini, cutoff

    def _get_best_split(self, X, y):
        feature = None
        min_gini = 1
        cutoff = None

        for i, c in enumerate(X.T):
            gini, cur_cutoff = self._get_best_feature_split(c, y)
```

```
            if gini == 0:
                return i, cur_cutoff, gini
            elif gini <= min_gini:
                min_gini = gini
                feature = i
                cutoff = cur_cutoff

        return feature, cutoff, min_gini

    def _get_prediction(self, row):
        cur_layer = self.trees

        while cur_layer:
            prev_layer = cur_layer

            if row[cur_layer['index']] < cur_layer['cutoff']:
                cur_layer = cur_layer['left']
            else:
                cur_layer = cur_layer['right']
        else:
            return prev_layer.get('value')

    def fit(self, X, y, parent_leaf={}, depth=0):

        if parent_leaf is None or len(y) == 0 or depth >= self.max_depth:
            return None

        feature, cutoff, gini = self._get_best_split(X, y)
        y_left = y[X[:, feature] < cutoff]
        y_right = y[X[:, feature] >= cutoff]

        parent_leaf = {
            'feature': self.feature_names[feature],
            'index': feature,
            'cutoff': cutoff,
            'value': np.round(np.mean(y))}


        parent_leaf['left'] = self.fit(X[X[:, feature] < cutoff], y_left, {},
depth+1)
        parent_leaf['right'] = self.fit(X[X[:, feature] >= cutoff], y_right,
{}, depth+1)

        self.depth += 1
        self.trees = parent_leaf
```

```
        return parent_leaf

    def predict(self, X):
        results = np.array([0]*len(X))

        for i, row in enumerate(X):
            results[i] = self._get_prediction(row)

        return results
```

Ok, there are a lot of bits and pieces that need to be explained here. Let's split this class into smaller chunks and start form the constructor:

```
def __init__(self, feature_names, max_depth = 5):
        self.gini_calculator = MyGiniCalculator()
        self.feature_names = feature_names
        self.max_depth = max_depth
        self.depth = 0
```

Here we initialize several fields, of which the *max_depth* is the most important one. It is the **hyperparameter** that controls the '*greediness*' of our algorithm. Of course, we create instances of the *MyGiniCalculator*. The next method is *_get_best_feature_split*:

```
def _get_best_feature_split(self, feature, y):
      min_gini = 10

      for value in set(feature):
          is_left = feature < value
          gini = self.gini_calculator.get_gini(is_left, y)
          if gini < min_gini:
              min_gini = gini
          cutoff = value

      return min_gini, cutoff
```

This is a private method used to get the split of the feature. Basically based on the value of the *Gini impurity index* for that feature, we calculate the threshold or the **cutoff** value. This method is used in *_get_best_split*:

```
def _get_best_split(self, X, y):
      feature = None
      min_gini = 1
      cutoff = None

      for i, c in enumerate(X.T):
          gini, cur_cutoff = self._get_best_feature_split(c, y)

          if gini == 0:
              return i, cur_cutoff, gini
          elif gini <= min_gini:
              min_gini = gini
              feature = i
              cutoff = cur_cutoff

      return feature, cutoff, min_gini
```

This private method is used to iterate through all features and calculate the best **thresholds** and **features** on which the split should be done on each depth of the tree. Basically, this method calculates the *Gini impurity indexes* for all features and picks the one feature that **minimizes** the *Gini impurity index* the most. The final private method is *_get_prediction*:

```
def _get_prediction(self, row):
        cur_layer = self.trees

        while cur_layer:
            prev_layer = cur_layer

            if row[cur_layer['index']] < cur_layer['cutoff']:
                cur_layer = cur_layer['left']
            else:
                cur_layer = cur_layer['right']
        else:
            return prev_layer.get('value')
```

This method is used once predictions are made. The two public methods are very interesting. Let's start with the *fit* method:

```
def fit(self, X, y, parent_leaf={}, depth=0):

        if parent_leaf is None or len(y) == 0 or depth >= self.max_depth:
            return None

        feature, cutoff, gini = self._get_best_split(X, y)
        y_left = y[X[:, feature] < cutoff]
        y_right = y[X[:, feature] >= cutoff]

        parent_leaf = {
            'feature': self.feature_names[feature],
            'index': feature,
            'cutoff': cutoff,
            'value': np.round(np.mean(y))}


        parent_leaf['left'] = self.fit(X[X[:, feature] < cutoff], y_left, {},
                                       depth+1)
        parent_leaf['right'] = self.fit(X[X[:, feature] >= cutoff], y_right,
                                       {}, depth+1)

        self.depth += 1
        self.trees = parent_leaf

        return parent_leaf
```

This method performs the **training** process. Once the input and output values are passed into this method, it gets the **best split** and stores it into the *parent_leaf* dictionary. Every leaf is essentially abstracted by one **dictionary,** and every leaf has its **left** and **right** leaf (except for terminal ones). So, for the first level, we calculate the parent leaf split. Then, based on the feature and values on this level, we **recursively** create the *left* and *right* leaf based on these values. This is how we build a tree. Once the training is done, we can use the predict method:

```
def predict(self, X):
        results = np.array([0]*len(X))

        for i, row in enumerate(X):
            results[i] = self._get_prediction(row)
```

```
        return results
```

This method is fairly simple. It utilizes *_get_prediction*, which follows the path to the result. Here is how we can call this class on the prepared dataset:

```
model = MyDecissionTreeClassifier(feature_names=data.drop(['species'],
axis=1).columns)
tree = model.fit(X_train, y_train)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))

pd.DataFrame({
    'Actual Value': y_test,
    'MyDecissionTreeClassifier Predictions': predictions,
})
```

```
              precision    recall  f1-score   support

          -1       1.00      0.93      0.96        28
           0       0.00      0.00      0.00         0
           1       0.96      0.96      0.96        25

    accuracy                           0.94        53
   macro avg       0.65      0.63      0.64        53
weighted avg       0.98      0.94      0.96        5
```

The accuracy_score is ~0.94, which is not that bad.

2.3.2.2 Using SciKit Learn

Of course, the faster way is to use the *Sci-Kit Learn* library for this purpose and their *DecisionTreeClassifier*:

```
dt_model = DecisionTreeClassifier(max_depth=2)

dt_model.fit(X_train, y_train)

dt_predictions = dt_model.predict(X_test)
print(metrics.classification_report(y_test, dt_predictions))
```

```
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00        28
           1       1.00      1.00      1.00        25

    accuracy                           1.00        53
   macro avg       1.00      1.00      1.00        53
weighted avg       1.00      1.00      1.00        53
```

With this class we get 100% accuracy. If we compare the results:

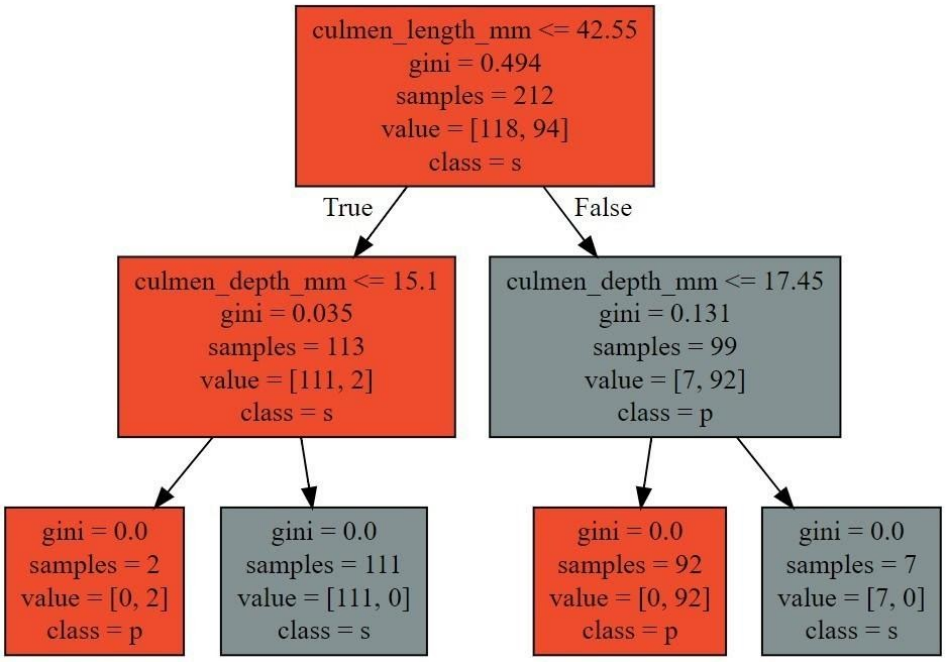
```
pd.DataFrame({
    'Actual Value': y_test,
    'MyDecissionTreeClassifier Predictions': predictions,
    'DecisionTreeClassifier Predictions': dt_predictions
})
```

	Actual Value	My Decision Tree Predictions	DecisionTreeClassifier Predictions
0	1	1	1
1	1	1	1
2	1	1	1
3	1	1	1
4	-1	-1	-1
5	-1	-1	-1
6	1	1	1
7	-1	-1	-1
8	1	1	1
9	1	1	1
10	-1	-1	-1
11	-1	-1	-1
12	1	1	1
13	1	1	1
14	1	1	1
15	-1	-1	-1
16	-1	-1	-1
17	-1	-1	-1
18	-1	-1	-1
19	1	1	1
20	-1	-1	-1
21	-1	-1	-1
22	-1	-1	-1
23	-1	-1	-1

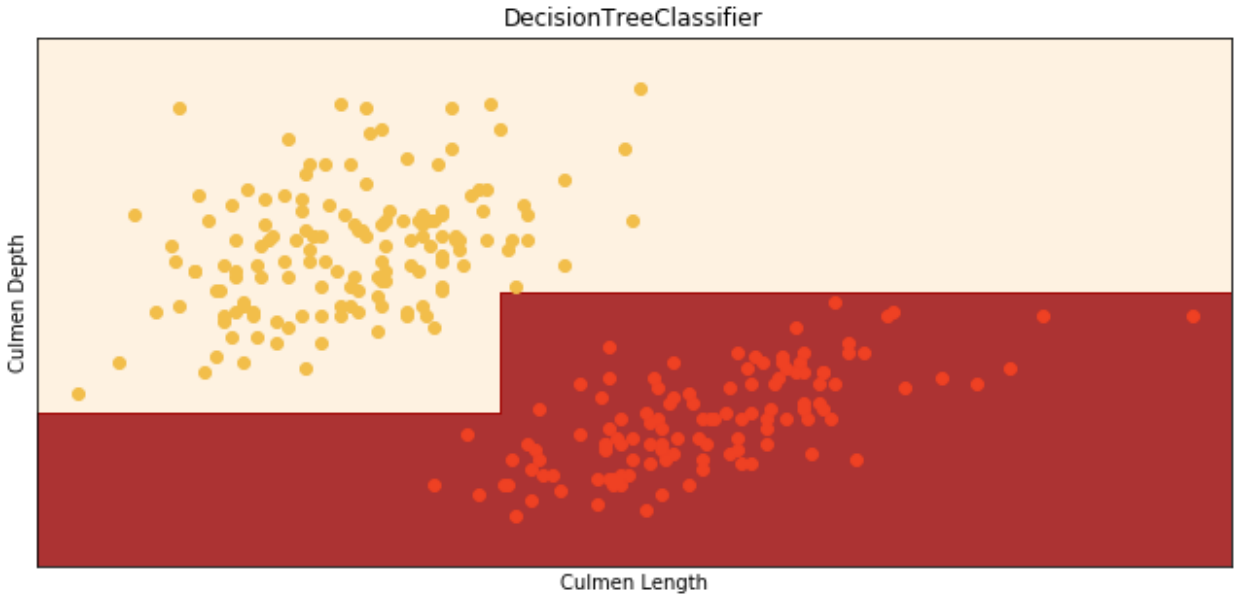
One more cool thing we can do with the *Sci-Kit Learn's* Decision Trees is that we can export them into **.dot** file using *export_graphviz*:

```
export_graphviz(
    dt_model,
    out_file="penguins.dot",
    feature_names=data.drop(['species'], axis=1).columns,
    class_names='species',
    rounded=False,
    filled=True
)
```

This file can be opened **here**. Here is the result:



It is a very cool visual representation of the created Decision Tree and its leaves. In this diagram, we can see every leaf of the Gini index, every feature with the threshold used and the number of samples. Another cool thing we can do is present the split on the data diagram:

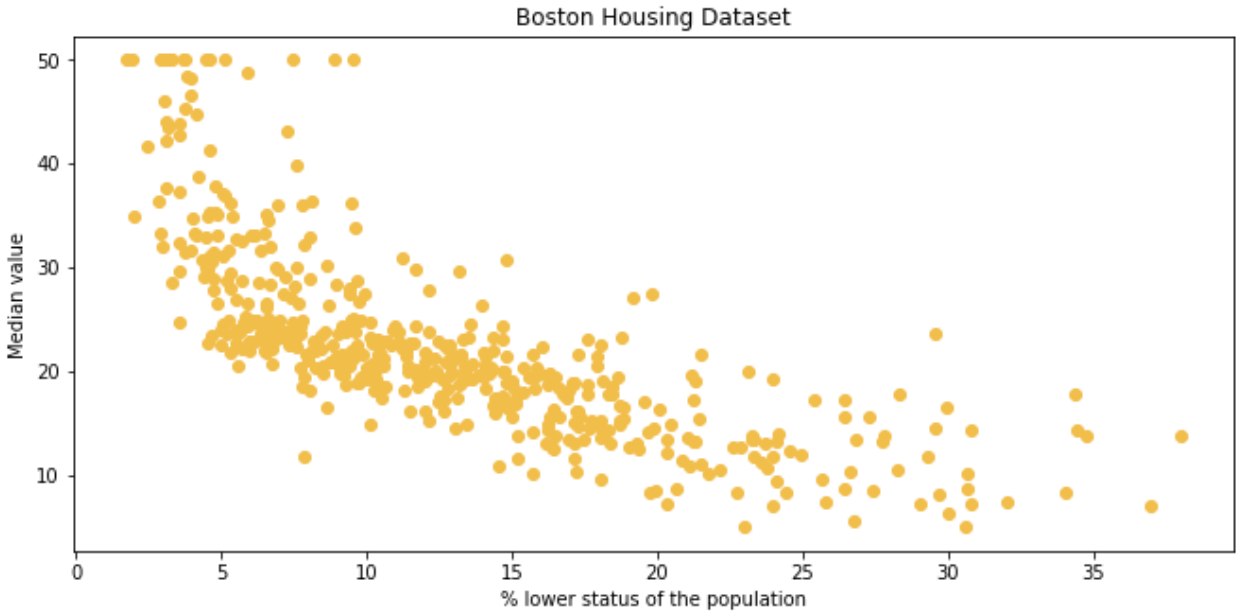


We can see how the Decision Trees make almost perfect splits on the data. This is probably one of the major downfalls of these algorithms - they are prone to overfitting.

2.3.2.3 Decision Tree Regression

The really cool thing about Decision Trees is that they can be used for regression too. Let's load the *Boston Housing* dataset and check out how that is done:

```
data = pd.read_csv('./data/boston_housing.csv')
data.head()

data = data.dropna()

X = data['lstat'].values
X = X.reshape(-1, 1)

y = data['medv'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)

plt.scatter(x = X, y = y, color='orange')
plt.show()
```



The data is fairly simple. This dataset contains 14 features, out of which we use 2. Our goal in this example is to predict the value of the property based on the age of the occupant. So, we remove all other features. As you can see, this is the case of non-linear regression. So, let's try to create a model that will best fit this data.

2.3.2.4 Using Sci-Kit Learn

Once again, the *Sci-Kit Learn* helps us do this:

```
dtr_model = DecisionTreeRegressor(max_depth=5)
dtr_model.fit(X_train, y_train)

dtr_predictions = dtr_model.predict(X_test)

print_evaluation_metrics(y_test, dtr_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'DecisionTreeRegressor Prediction': dtr_predictions,
})
```

	Actual Value	DecisionTreeRegressor Prediction
0	20.5	15.928571
1	5.6	12.692308
2	13.4	12.692308
3	12.6	15.928571
4	21.2	29.600000
...	...	...
97	25.0	21.637209
98	19.5	17.628571
99	19.9	15.928571
100	15.4	12.692308
101	21.7	20.160317

We can see that the results are not bad and it is really interesting once we **plot** it:

2.4 Ensemble Learning and Random Forest

The interesting occurrence in **machine learning** is that sometimes we tend to get better results by using multiple predictors and then averaging results rather than using one special algorithm. This technique in which we use multiple algorithms instead of one is called **Ensemble Learning**. Ensemble Learning is based on the law of large numbers, which means that even if the algorithms that are composing the ensemble are weak learners, the ensemble can be a strong learner. There are several ways these ensemble learners function. For example, in the technique called **hard voting**, several classifiers vote for the class, and the class that gets the majority of the votes is the output. This is a bit unintuitive, but if you build an ensemble containing 1,000 classifiers and each of them has an accuracy of 51% on its own, assemble based on hard-voting can have an accuracy up to 75%. There is also a **soft voting** technique. In this case, each algorithm outputs probability; the ensemble will predict the class with the highest class probability, **averaged** over all the individual classifiers.

One of the most popular ways to build ensembles is to use the same algorithm multiple times but on the different subsets of the training dataset. Techniques that are used for this are called **bagging** and **pasting**. The only difference in these techniques is that while building subsets, bagging allows training instances to be sampled several times for the same predictor, while pasting doesn't allow that. When all algorithms are trained, the ensemble makes a prediction by aggregating the predictions of all algorithms. In the classification case, that is usually the hard-voting process, while the average result is taken for the regression.

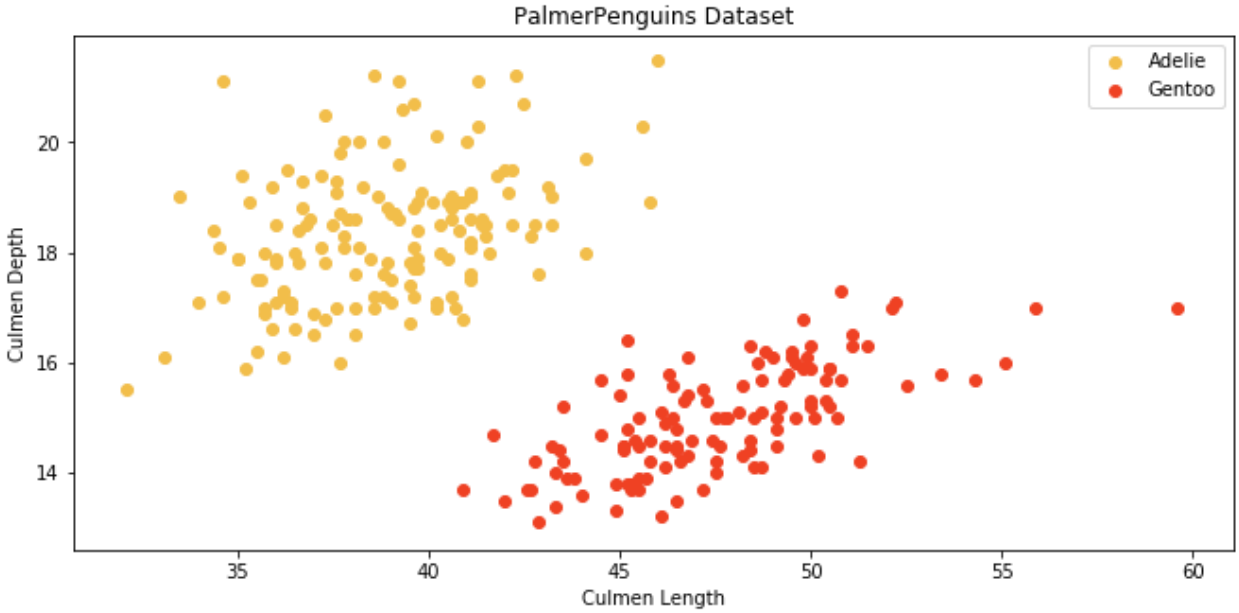
Random Forest is one of the most powerful algorithms in machine learning. It is an ensemble of Decision Trees. In most cases, we train Random Forest with bagging to get the best results. It introduces additional randomness when building trees as well, which leads to greater tree diversity. This is done by the procedure called **feature bagging**. This means that, during the training, each tree is trained on a different subset of features. In turn, this leads to lower **variance** of the complete model. This should be enough theory; let's implement *RandomForest* from scratch using *Python*.

2.4.1 Prepare Data for Classification

Just like we did in the previous chapter, first we prepare data for Random Forest. First, we explore how we can use Random Forest for classification and we preprocess the data in the same way as for decision trees:

```
data = pd.read_csv('./data/penguins_size.csv')

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)
data = data[data['species'] != 'Chinstrap']
```

```
X = data.drop(['species'], axis=1).values

y = data['species']
species = {'Adelie': 1, 'Gentoo': 2}
y = [species[item] for item in y]
y = np.array(y)

# Remove sample that is too close
X = np.delete(X, 182, axis=0)
y = np.delete(y, 182, axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```



2.4.3 Classification Python Implementation

The complete solution is implemented within **MyRandomForest** class, so let's check it out. Note that in this implementation we use *MyDecissionTreeClassifier* class we implemented in previous chapter.

```
class MyRandomForest():
    def __init__(self, num_trees, feature_names, max_depth=5):
        self.num_trees = num_trees
        self.max_depth = max_depth

        self.trees = []
```

```
        for _ in range(num_trees):
            self.trees.append(MyDecissionTreeClassifier(feature_names,
self.max_depth))

    def get_random_subsets(self, X, y, num_subsets):
        num_samples = np.shape(X)[0]

        X_y = np.concatenate((X, y.reshape((1, len(y)))).T), axis=1)
        np.random.shuffle(X_y)

        subsets = []
        subsample_size = int(num_samples // 2)

        for _ in range(num_subsets):
            index = np.random.choice(
                range(num_samples),
                size=np.shape(range(subsample_size)),
                replace=True)

            X = X_y[index][:, :-1]
            y = X_y[index][:, -1]

            subsets.append([X, y])

        return subsets

    def fit(self, X, y):
        num_features = np.shape(X)[1]
        subsets = self.get_random_subsets(X, y, self.num_trees)

        for i in range(self.num_trees):
            X_subset, y_subset = subsets[i]

            # Feature bagging
            idx = np.random.choice(range(num_features), size=num_features,
replace=True)
            self.trees[i].feature_indices = idx
            X_subset = X_subset[:, idx]
            self.trees[i].fit(X_subset, y_subset)


    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))

        for i, tree in enumerate(self.trees):
```

```
            idx = tree.feature_indices
            prediction = tree.predict(X[:, idx])
            y_preds[:, i] = prediction

        y_pred = []
        for sample_predictions in y_preds:

y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())

        return y_pred
```

This is a lot of code, so let's split it a little bit and explain each piece. In the constructor of the class, we create an array of *MyDecissionTreeClassifier* instances and define the *max_depth* for each of them. Apart from that, we set the *max_features* **hyperparameter** used for **feature bagging**.

```
def __init__(self, num_trees, feature_names, max_depth=5):
        self.num_trees = num_trees
        self.max_depth = max_depth

        self.trees = []
        for _ in range(num_trees):
            self.trees.append(MyDecissionTreeClassifier(feature_names,
                                                        self.max_depth))
```

In the *get_random_subsets* method, we create a defined number of subsets, which are later used to **train** each individual tree.

```
    def get_random_subsets(self, X, y, num_subsets):
        num_samples = np.shape(X)[0]

        X_y = np.concatenate((X, y.reshape((1, len(y)))).T), axis=1)
        np.random.shuffle(X_y)

        subsets = []
        subsample_size = int(num_samples // 2)

        for _ in range(num_subsets):
            index = np.random.choice(
                range(num_samples),
                size=np.shape(range(subsample_size)),
```

```
                replace=True)

            X = X_y[index][:, :-1]
            y = X_y[index][:, -1]

            subsets.append([X, y])

        return subsets
```

The *fit* method is fairly simple. For each tree, we create a subset of data, perform feature bagging and run the **training** process.

```
    def fit(self, X, y):
        num_features = np.shape(X)[1]
        subsets = self.get_random_subsets(X, y, self.num_trees)

        for i in range(self.num_trees):
            X_subset, y_subset = subsets[i]

            # Feature bagging
            idx = np.random.choice(range(num_features), size=num_features,
replace=True)
            self.trees[i].feature_indices = idx
            X_subset = X_subset[:, idx]
            self.trees[i].fit(X_subset, y_subset)
```

Finally, let's see what we get when we use this on the *PalmerPenguins* dataset:

```
model = MyRandomForest(feature_names=data.drop(['species'], axis=1).columns,
num_trees=11, max_features=2)
tree = model.fit(X_train, y_train)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        28
           2       1.00      1.00      1.00        25

    accuracy                           1.00        53
   macro avg       1.00      1.00      1.00        53
weighted avg       1.00      1.00      1.00        53
```

As you can see, we get accuracy improvement just from using *MyDecissionTreeClassifier*.

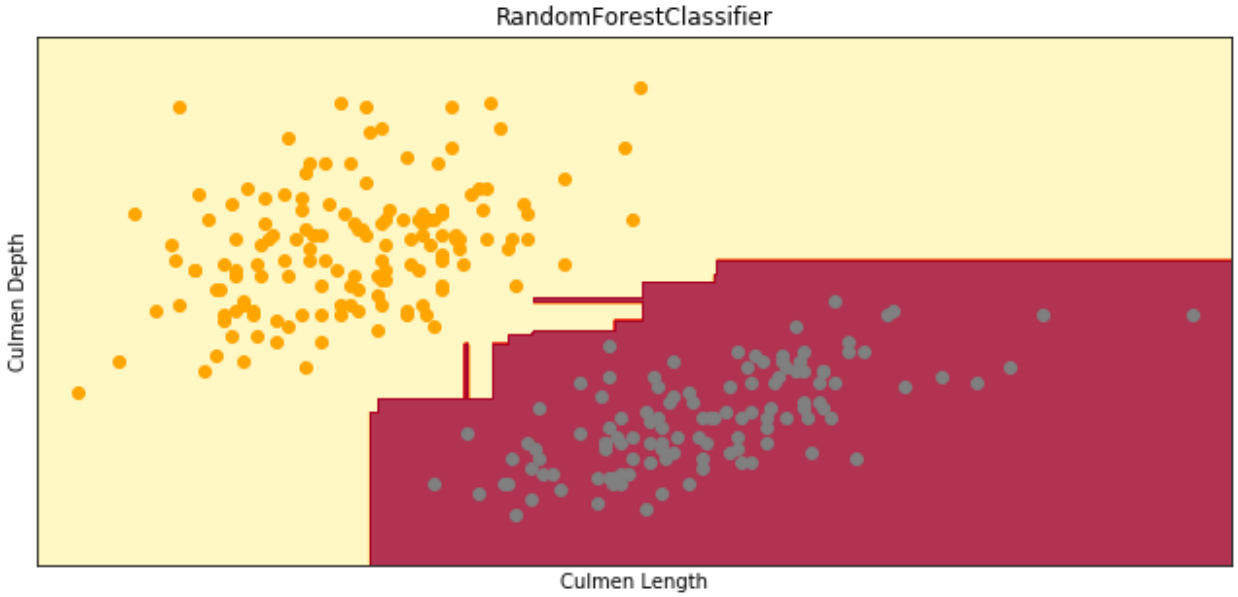
2.4.4 Classification with Sci-Kit Learn

The *Sci-Kit Learn* library provides a better implementation of this algorithm. Here is how we can use the *RandomForestClassifier*:

```
rf_classifier = RandomForestClassifier(n_estimators=11, max_leaf_nodes=16,
n_jobs=-1)
rf_classifier.fit(X_train, y_train)

rf_predictions = rf_classifier.predict(X_test)
print(metrics.classification_report(y_test, rf_predictions))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        28
           2       1.00      1.00      1.00        25

    accuracy                           1.00        53
   macro avg       1.00      1.00      1.00        53
weighted avg       1.00      1.00      1.00        53
```

Nice! We got awesome metrics. Or did we? Random Forest and Decision Tree are, in general, great algorithms, but they are prone to overfitting. If you see metrics like this, make sure that your implementation didn't overfit. Let's check out what the classification diagram for *Random Forest* looks like and what the differences from the Decision Trees one are:

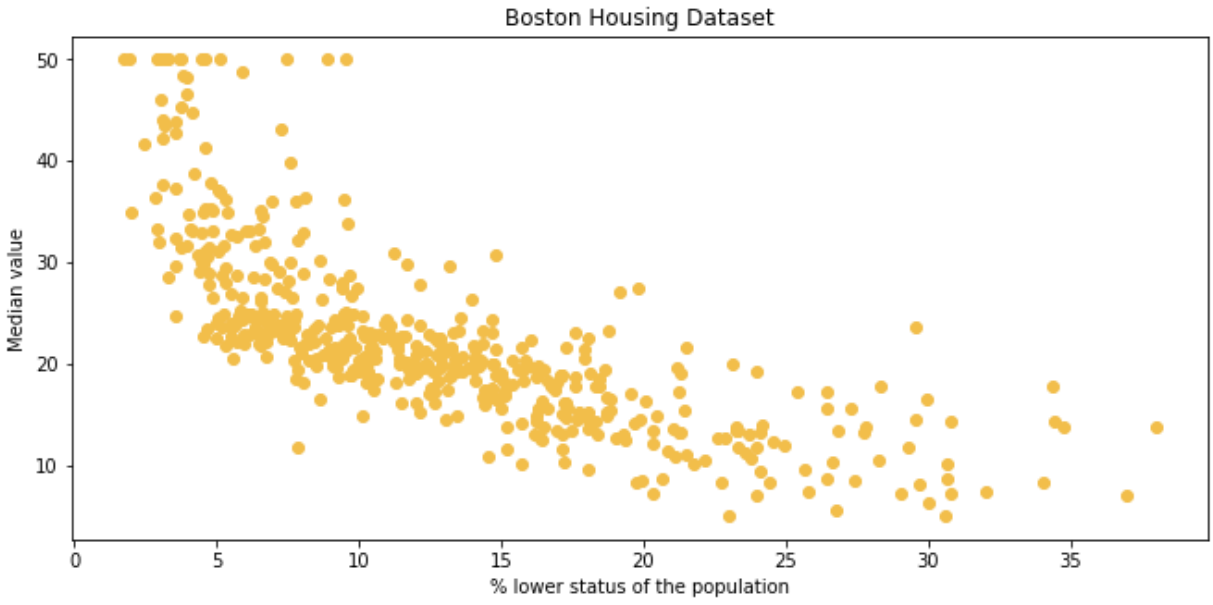


Ok, that looks much better. It seems that *Random Forest* got a better feel of where the border between classes is.

2.4.5 Prepare Data for Regression

Again, let's just prepare the regression data for Random Forest real quick. You have seen this several times in this book, but we need to make sure that we are on the same page. So here it is:

```
data = pd.read_csv('./data/boston_housing.csv')
data.head()

data = data.dropna()

X = data['lstat'].values
X = X.reshape(-1, 1)

y = data['medv'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)

plt.figure(figsize=(11, 5))

plt.title("Boston Housing Dataset")
plt.xlabel('% lower status of the population')
plt.ylabel('Median value')

plt.scatter(x = X, y = y, color=ORANGE)
plt.show()
```



2.4.6 Regression with Sci-Kit Learn

Classification seems simple enough, but just like the *Decision Trees*, *Random Forest* can be used for regression too. Again, **bagging** is used during training, but instead of the voting process, the results of all learners are **averaged**. Here is how we can use it for the *Boston Housing* dataset:

```
rfr_model = RandomForestRegressor(n_estimators=111, max_leaf_nodes=16,)
rfr_model.fit(X_train, y_train)

rfr_predictions = rfr_model.predict(X_test)

print_evaluation_metrics(y_test, rfr_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'RandomForestRegressor Prediction': rfr_predictions,
})
```

```
MAE: 3.7711599513447256
MSE: 24.157626367620978
RMSE: 4.915040830717582
R Squared: 0.6627653295633458
```

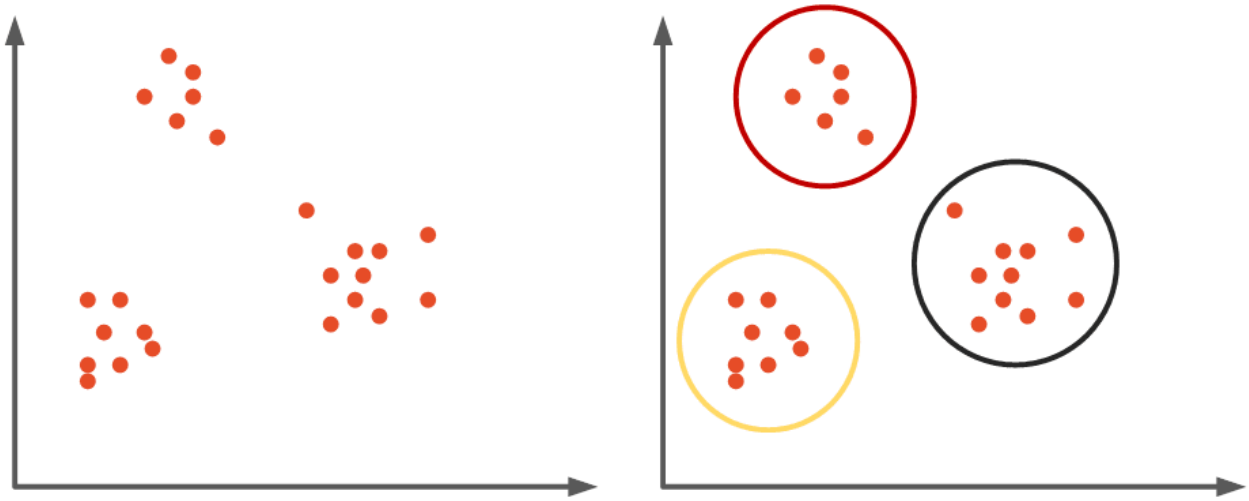
	Actual Value	RandomForestRegressor Prediction
0	20.5	16.191060
1	5.6	12.421497
2	13.4	12.418455
3	12.6	16.547765
4	21.2	28.846192
...	...	...
97	25.0	22.444650
98	19.5	16.121502
99	19.9	16.570633
100	15.4	13.332339
101	21.7	20.567007

Again, compared to the Decision Tree results, it seems a bit better. Graphically, it looks better too:

2.5 Clustering Algorithms

Up to this point, we explored algorithms that use supervised learning. This means that we always had input and **expected** output data that we used to train our machine learning models. In this type of learning, the training set contains inputs and desired outputs. This way, the algorithm can check its calculated output the same as the desired output and take appropriate **actions** based on that.

However, in real life, we often **don't have** both input and output data, but only input data. This means that the algorithm in itself needs to figure out **connections** between input samples. For that, we use **unsupervised learning**. In unsupervised learning, the training set contains only **inputs**. Just like we solve regression and classification problems with supervised learning, we solve **clustering** problems with unsupervised learning. This technique attempts to identify similar inputs and put them into **categories**, i.e., it clusters data. Generally speaking, the goal is to detect the hidden **patterns among** the data and group them into clusters. This means that the samples which have some shared properties will fall into one group – **cluster**.



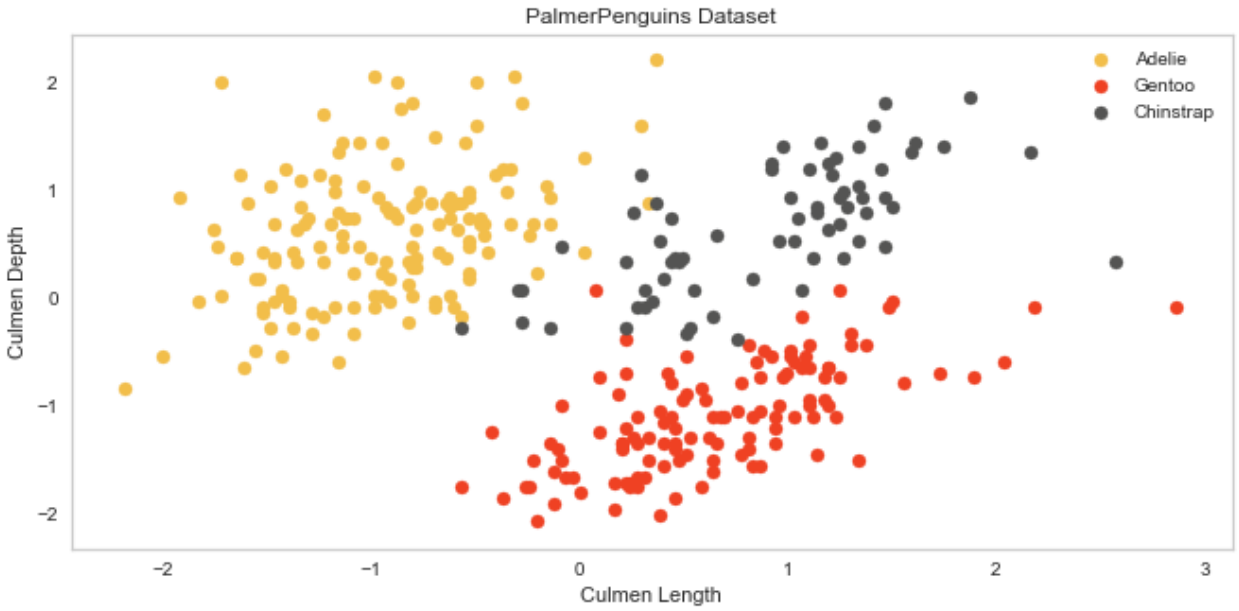
There are many clustering algorithms out there and, in this chapter, we cover three of them: **K-Means** Clustering, **Agglomerative** Clustering and **DBSCAN**. As one can imagine, since the dataset is completely unlabeled, deciding which algorithm is **optimal** for the chosen dataset is much more **complicated**. Usually, the performance of each algorithm depends on the unknown properties of the probability distribution the dataset was drawn from.

2.5.1 Preparing Data for Clustering

Ok, let's prepare data for processing.

```
data = pd.read_csv('./data/penguins_size.csv')

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)

X = data.drop(['species'], axis=1).values
ss = StandardScaler()
X = ss.fit_transform(X)

y = data['species']
species = {'Adelie': 0, 'Gentoo': 1, 'Chinstrap': 2}
y = [species[item] for item in y]
y = np.array(y)
```

First, we **load** it from the *.csv* file and remove the features we don't want to consider in this tutorial. Then, we split the input data *X* from the labels *y* and we scale the input data. The output data is used only for **checking** out how well our clustering algorithms work. Here is how the data looks like when we **plot** it:



2.5.2 K-Means Clustering

K-Means is one of the most **popular** clustering algorithms. It is definitely a go-to option when you **start** experimenting with your unlabeled data. This algorithm groups *n* data points into *K* number of clusters, as the name of the algorithm suggests. This algorithm can be split into several stages:

- In the first stage, we need to set the hyperparameter k. This represents the number of clusters or groups that K-Means Clustering will create once it is done.
- K random vectors are picked in the feature space. These vectors are called centroids. They are changed during the training process and the goal is to put them into the "center" of each cluster.
- Distances from each input sample x to each centroid c are calculated using a certain metric, like the Euclidean distance. The closest centroid is assigned to each sample in the dataset. Basically, we create clusters at this stage.
- For each cluster, the average feature vector is calculated using samples that are assigned to it. This value is considered as a new centroid of the cluster.
- Steps 2-4 are repeated for a fixed number of iterations or until the centroids don't change, whichever comes first.

Mathematically speaking, each sample *x* is assigned to a cluster based on:

$$arg(min(c_i \in c) * (c_i, x)^2)$$

where c_i is the centroid of the cluster *i* and *D* is the Euclidean distance calculated using the formula:

$$D(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$$

For finding the new centroid from the clustered group of points, we use the formula:

$$c_i = \frac{1}{|S_i|} * \sum_{x_i \in S_i} x_i$$

As we already mentioned, the hyperparameter *k*, i.e. the **number** of clusters, has to be tuned manually. This is quite tiresome. Later in this tutorial, we consider one technique for selecting the **correct** number of clusters. However, let's first implement this simple yet powerful algorithm using *Python* from **scratch**.

2.5.2.1 Python Implementation

This algorithm is implemented within the ***MyKMeansClustering*** class. The process is quite straightforward, so let's check out the implementation:

```
class MyKMeansClustering():
    def __init__(self, k=2):
        self.k = k
```

```
    def fit(self, X, max_num_iterattions=300, sse_treshold = 0.001):

        self.centroids = {}

        # Initialize Centroids
        for i in range(self.k):
            self.centroids[i] = X[i]

        for i in range(max_num_iterattions):

            # Initialize Clusters
            self.clusters = {}
            for i in range(self.k):
                self.clusters[i] = []

            # Euclidean distance for each point
            for sample in X:
                distances = [np.linalg.norm(sample-self.centroids[centroid])
                             for centroid in self.centroids]
                cluster = distances.index(min(distances))
                self.clusters[cluster].append(sample)

            # Update centroids
            previous_centroids = dict(self.centroids)
            for cluster in self.clusters:
                self.centroids[cluster] = np.average(self.clusters[cluster],
                                                     axis=0)

            # Check if centroids changed
            centroids_changed = True

            for centroid in self.centroids:
                sse = np.sum((self.centroids[centroid] -
                              previous_centroids[centroid])/
                              previous_centroids[centroid]*100.0)
                if  sse > sse_treshold:
                    centroids_changed = False

            if centroids_changed:
                break

    def predict(self, sample):
        distances = [np.linalg.norm(sample - self.centroids[centroid]) for
                     centroid in self.centroids]
        cluster = distances.index(min(distances))
```

```
        return cluster
```

There are two public methods, *fit* and *predict*. The number of clusters *k* is passed through the **constructor**. The *fit* method is basically where the magic happens. First, we **initialize** centroids by random. Then we calculate the Euclidean distance from each **point** in the dataset to each **centroid**. We assign samples to the closest centroid, i.e., to **cluster**. Then we **update** centroids by taking the average value for each cluster. Finally, check if the centroids have changed their value. If they have, we **stop** the process; otherwise, we repeat it until *max_num_iterattions* is reached. Also, notice the *threshold parameter*. Using this parameter, we can **fine-tune** this check. The *predict* method is used to predict the cluster value for each new sample.

Cool. Let's try it out on the *PalmerPenguins* dataset:

```
model = MyKMeansClustering(k = 3)
model.fit(X, max_num_iterattions=700, sse_treshold= 0.00001)

clusters = []
for sample in X:
    clusters.append(model.predict(sample))

print(metrics.classification_report(y, clusters))
```

```
              precision    recall  f1-score   support

           0       0.96      0.97      0.97       146
           1       0.93      0.94      0.94       120
           2       0.85      0.81      0.83        68

    accuracy                           0.93       334
   macro avg       0.91      0.91      0.91       334
weighted avg       0.93      0.93      0.93       334
```

This is a bit unorthodox, but in the end, we are **checking** the various metrics like we perform classification. In normal conditions, we wouldn't have the output information and thus we would not know the **result** of our actions. However, since this is an educational tutorial, we can see that the K-Means algorithm has done a good job, with an accuracy of ~93%. Let's print out the output and compare it with the real values:

Apparently, there are several missing samples in the middle of the distribution, but overall not bad, not bad at all.

2.5.2.2 Using SciKit Learn

Using this algorithm with Sci-Kit is even easier:

```
kmeans = KMeans(n_clusters=3, max_iter = 700, random_state=6)
kclusters = kmeans.fit_predict(X)
print(metrics.classification_report(y, kclusters))
```

```
              precision    recall  f1-score   support

           0       0.96      0.97      0.97       146
           1       0.93      0.94      0.93       120
           2       0.84      0.79      0.82        68

    accuracy                           0.93       334
   macro avg       0.91      0.90      0.91       334
weighted avg       0.92      0.93      0.92       334
```

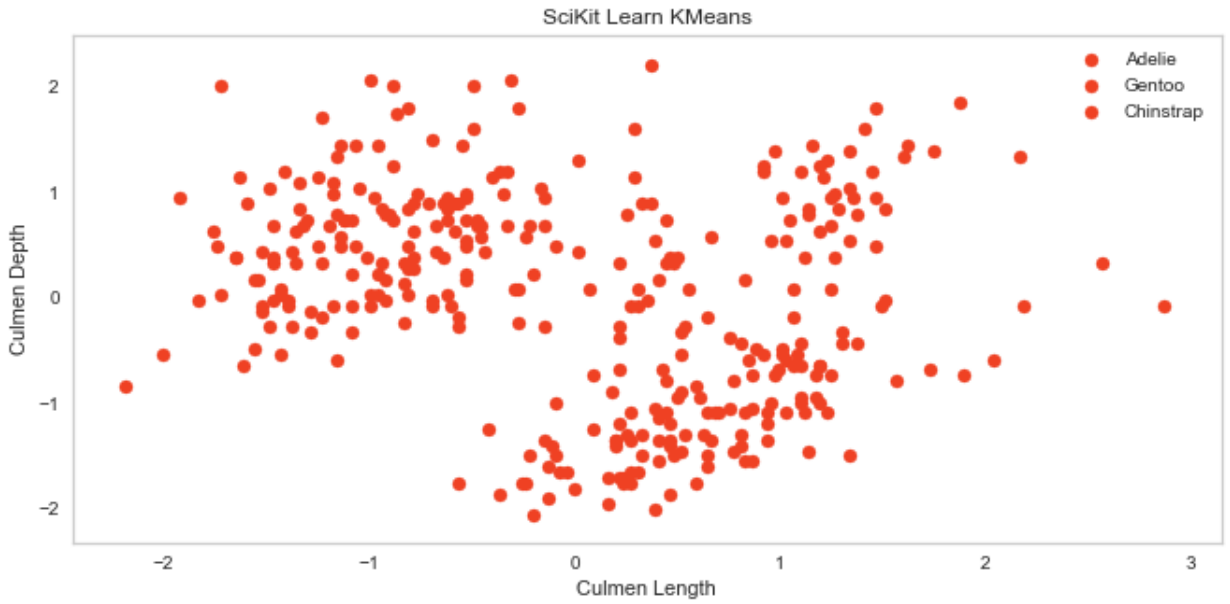
Since we are using the *predict* method on the input data as well, we can utilize the *fit_predict* method. Accuracy is still ~92% and the visual representation looks almost the same as our implementation:

Again, there are several points in the middle that are not properly clustered, but that data is specifically hard because it represents **edge** cases.

Things look pretty easy when we **know** the number of clusters beforehand, but what should we do when we get unlabeled data in the real world? How are we supposed to know the number of clusters? There are many techniques that can help us achieve that. One of the most popular ones is the **Elbow method**.

2.5.2.3 The Elbow Method

We know that we have tree classes in our dataset. However, let's **forget** about all that for a second and let's **plot** out data using the same color for all classes:

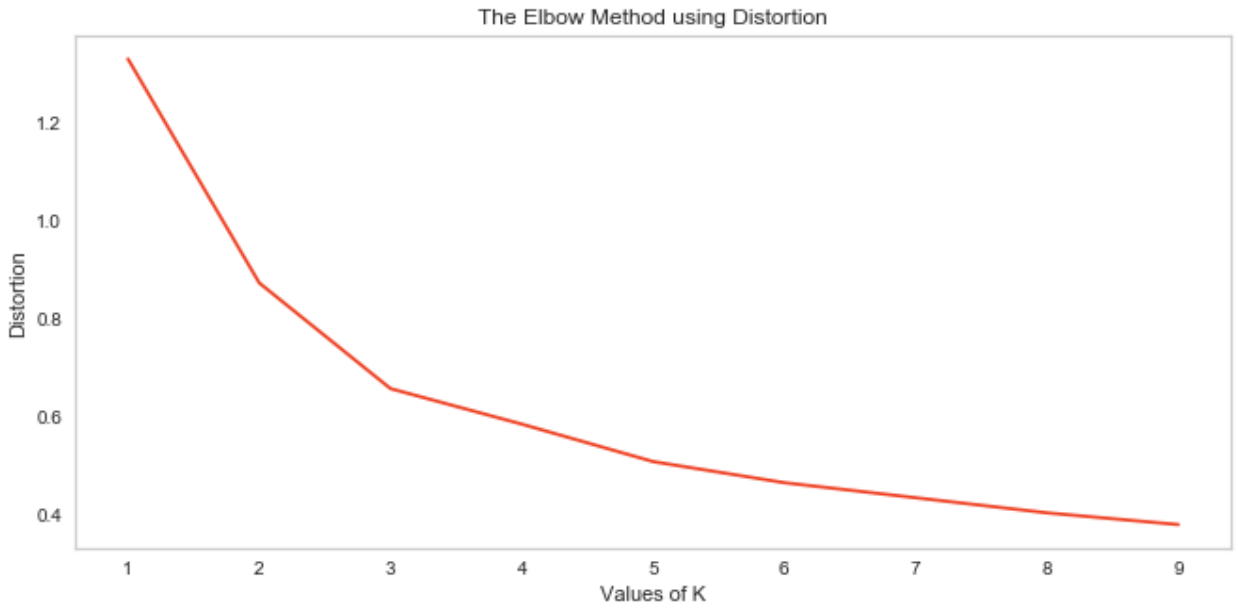


How many clusters do you see? We could say around 3, but we can not be sure. Plus, there is that data in the middle, which is super sketchy. As we said, one of the most popular methods for determining the number of clusters in the dataset is called the **Elbow Method**. It can be built on top of two metrics: distortion and inertia. **Distortion** is calculated as the average of the squared distances (let's say the Euclidean distance) from the cluster centers of the respective clusters. **Inertia** represents the sum of squared distances of samples to their closest cluster center.

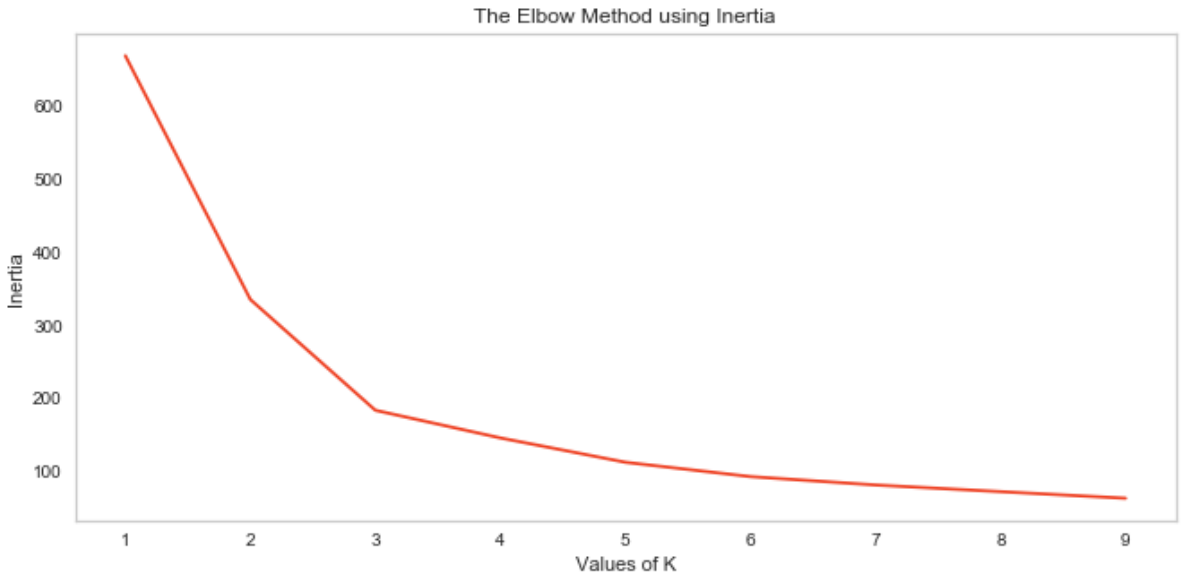
What we can do is run our clustering algorithm with a **variable** number of clusters and calculate distortion and inertia. Then we can plot the results. There we can look for the "elbow" point. This is the point after which the distortion/inertia starts decreasing in a **linear** fashion as the number of clusters grows. This point tells us the **optimal** number of clusters. Let's calculate both metrics first:

```
distortions = []
inertias = []

for k in range(1,10):
    kmeanModel = KMeans(n_clusters=k)
    kmeanModel.fit(X)

    distortions.append(sum(np.min(cdist(X, kmeanModel.cluster_centers_,
'euclidean'),axis=1)) / X.shape[0])
    inertias.append(kmeanModel.inertia_)
```

Now let's **plot** the distortion value along with the number of clusters:



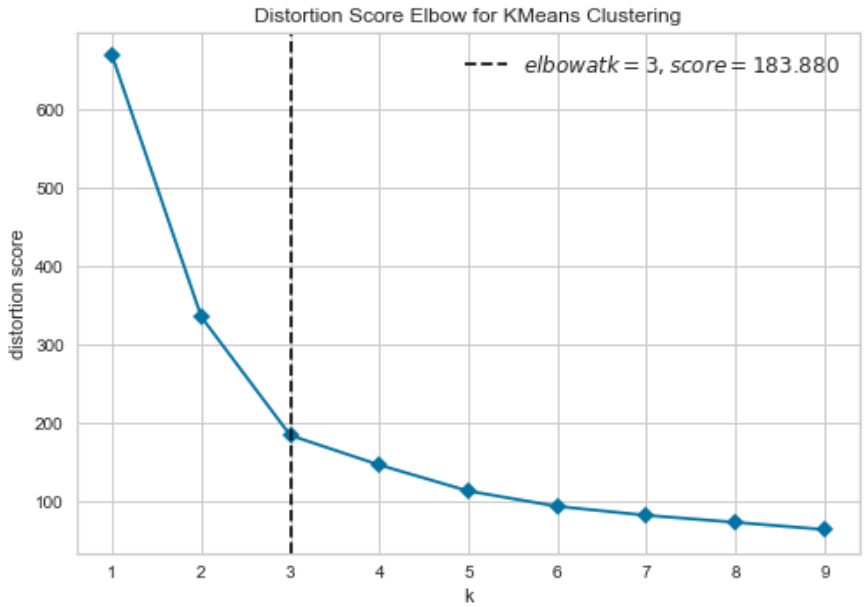
From this image, we can conclude that after 3 clusters, the distortion decreases in a linear fashion, i.e., 3 is the optimal number of clusters. How about inertia?



We can conclude the same thing from this image as well.

One library can help us with this process – **Yellowbrick**. This library is built on top of the Sci-Kit Learn and it provides a bunch of useful visualizations. Here is how we can get the above result by using it:

```
model = KMeans()
visualizer = KElbowVisualizer(model, k=(1,10))

visualizer.fit(X)
visualizer.show()
```



Again this visualizer used distortion as a metric. It can be useful when we are not quite sure when our metric starts to be linear.

2.5.3 Agglomerative Clustering

As we can see one of the biggest challenges of working with *K-Means* is that we need to **determine** the number of clusters beforehand. Another challenge is that *K-Means* tries to make clusters of the same **size**. These challenges can be addressed with other algorithms like **Hierarchical Clustering**. In general, every *Hierarchical Clustering* method starts by putting all samples into separate single-sample clusters. Then, based on some similarity metrics, samples or clusters are **merged together** until the point when **all** samples are put into a **single** cluster. This means that we are building the **hierarchy** of clusters, hence the name. In this article, we explore **Agglomerative Clustering,** which is a specific type of *Hierarchical Clustering*. The metric that it uses for merging clusters is the **distance**, i.e., it merges the closest pair of clusters based on the distance among centroids and repeats this step until only a single cluster is left. For this purpose, the **proximity matrix** is used. This matrix stores the distances between each point.

Let's break this into steps:

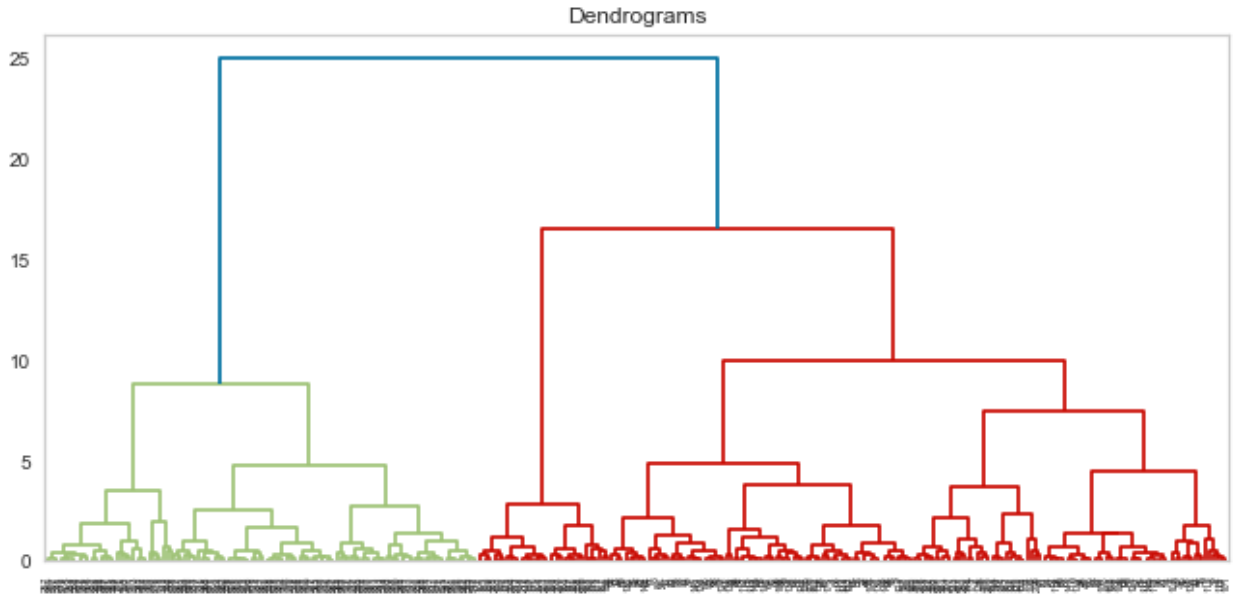
- Every point is stored in its own cluster.
- The proximity Matrix is calculated.
- Closest points are detected and merged. They are the cluster and the centroid is calculated.
- The proximity matrix is updated using the centroid of the created cluster.
- Steps 3 and 4 are repeated until one cluster is created.

At this point you might ask yourself: "How does this help us with deciding the number of clusters? To do so, we utilize an awesome concept – the **Dendrogram**.

2.5.3.1 Dendrogram

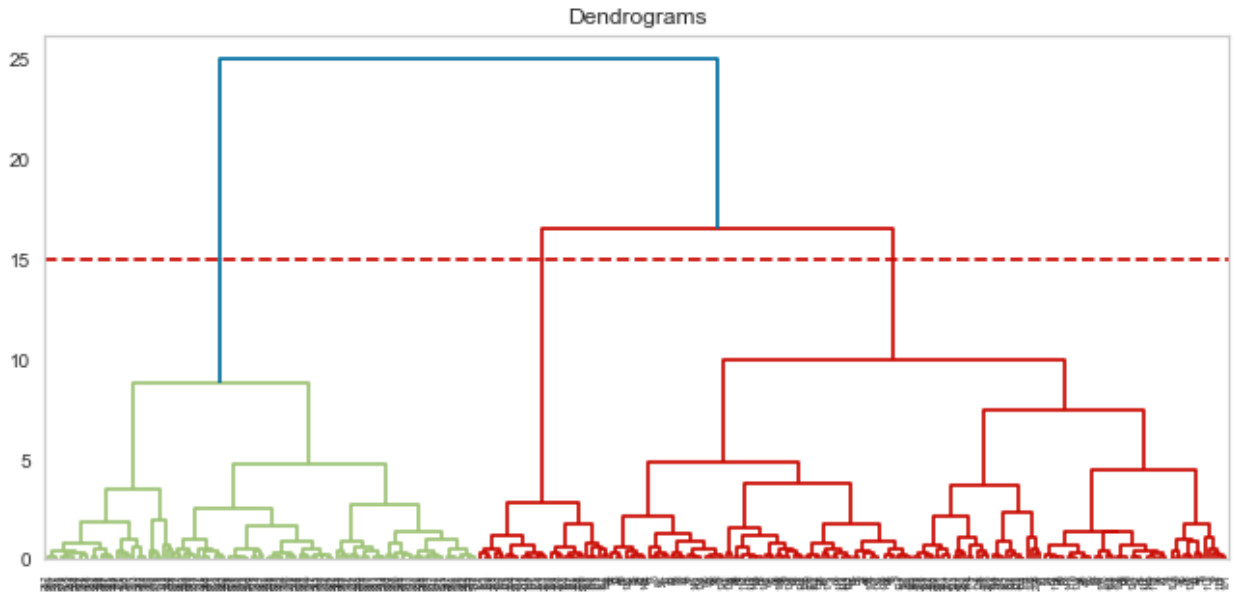
This is a tree diagram that **records** all merges that happened during the training process. So, every time we merge two points or clusters, this is **stored** in the dendrogram. Here is how we can create it using *SciPy*:

```
plt.figure(figsize=(11, 5))
plt.grid(False)
plt.title("Dendrograms")

dend = shc.dendrogram(shc.linkage(X, method='ward'))
```



What are we looking at? Well, on the x-axis we have all **points** in the dataset, while on the y-axis we have the **distance between** those points. Every time the points or clusters are merged it is represented by a **horizontal** line. The **vertical line** represents the distances between merged points/clusters. Longer vertical lines in the dendrogram indicate that the distance between clusters is **bigger**. In the next step, we need to set a **threshold** distance and draw a horizontal line in this image. In general, we try to set the threshold in such a way that it cuts the **tallest vertical line**. In our example, we set it to 15. Here is how that is done:

```
plt.figure(figsize=(10, 7))
plt.title("Dendrograms")
dend = shc.dendrogram(shc.linkage(X, method='ward'))
plt.axhline(y=15, color='r', linestyle='--')
```



The number of clusters is determined by the number of vertical lines that are being intersected by the threshold line. As we can see, the number of clusters in our example is 3.

2.5.3.2 Using Sci-Kit Learn

Alright, now we know how *Agglomerative Clustering* works behind the curtains. Let's see how we can use it with *Sci-Kit Learn*:

```
aglo = AgglomerativeClustering(n_clusters=3)
aclusters = aglo.fit_predict(X)
print(metrics.classification_report(y, aclusters))
```

```
              precision    recall  f1-score   support

           0       0.83      1.00      0.91       146
           1       0.97      0.98      0.98       120
           2       1.00      0.54      0.70        68

    accuracy                           0.90       334
   macro avg       0.93      0.84      0.86       334
weighted avg       0.92      0.90      0.89       334
```

The accuracy is a bit smaller than when we were using *K-Means Clustering*. Let's plot data and see how it compares to the real situation:

We can see that the cluster of 'Chinstrap' instances is much smaller than it is in reality, and that is where the lower accuracy comes from.

2.5.4 DBSCAN

Unlike *K-Means* and *Hierarchical Clustering*, which are **centroid-based** algorithms, DBSCAN is a **density-based** algorithm. Effectively, this means that you don't need to determine how many clusters you need. We saw this at *Hierarchical clustering*, but *DBSCAN* takes it to another level. Instead of defining hyperparameter *k*, we define two **hyperparameters for** distance ε and the number of samples per cluster – *n*. Let's break it into steps and it will be more clear:

- First, we assign a random sample x to the first cluster.
- We count how many samples have a distance from the sample x that is less or equal to ε. If the number of such samples is greater or equal to n, we add them in the cluster.
- We observe each new member of the cluster and perform step 2 for them too, i.e., we calculate the number of samples within the ε area of the sample and if that number is larger than n, we add them to the cluster. We repeat this process recursively until there are no more samples left to put in it.
- Repeat steps from 1 to 3 for a new random unclustered sample.
- The process is repeated like this until all samples are either clustered or marked as outliers.

The main advantage of this approach is that clusters have different and random **shapes**. The centroid-based algorithms always create clusters that have the shape of a **hypersphere**. That is why *DBSCAN* can be specifically useful for some data. Of course,

the main problem is choosing **optimal** values for ε and *n*. This problem is optimized with a variation of this algorithm called **HDBSCAN**, i.e., high-performance DBSCAN. This algorithm eliminates the use of the ε hyperparameter; however, it is out of the scope of this tutorial.

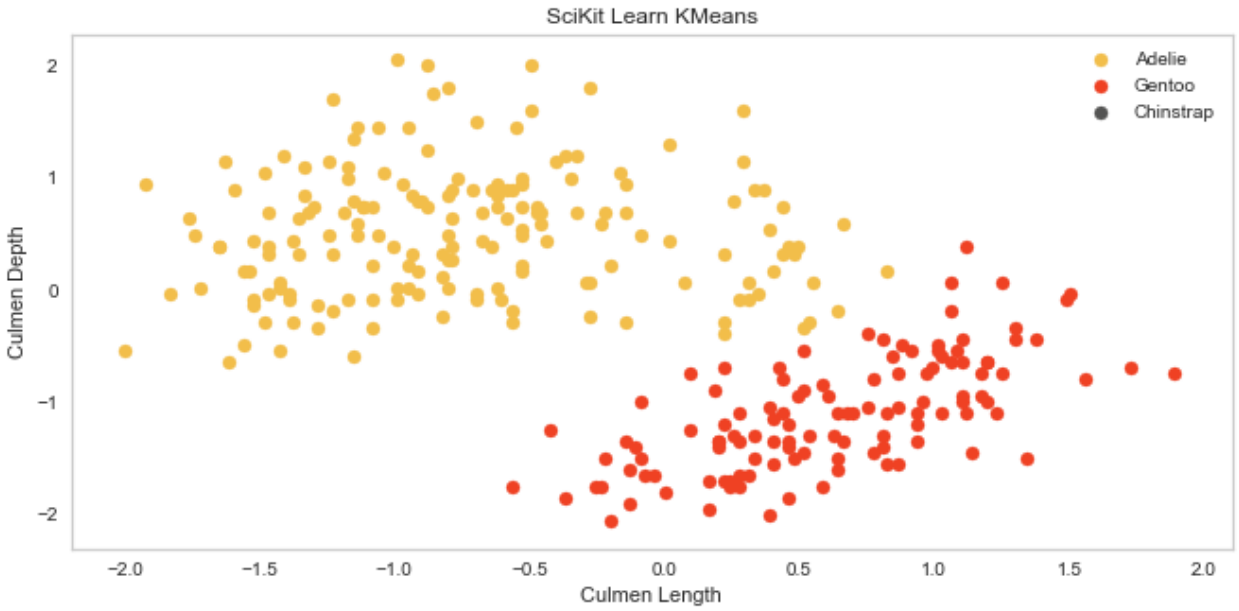
2.5.4.1 Using Sci-Kit Learn

Ok, let's run the *PalmerPenguins* data through *DBSCAN* with the *Sci-Kit Learn* implementation:

```
dbscan = DBSCAN(eps = 0.89, min_samples=74)
dbclusters = dbscan.fit_predict(X)
print(metrics.classification_report(y, dbclusters))
```

```
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
           0       0.83      0.97      0.89       146
           1       0.97      0.96      0.97       120
           2       0.00      0.00      0.00        68

    accuracy                           0.77       334
   macro avg       0.45      0.48      0.46       334
weighted avg       0.71      0.77      0.74       334
```

With this algorithm, we got the worst accuracy. However, keep in mind that this doesn't mean that this is a bad algorithm, just that it is not suitable for this data, or we need to further experiment and use better hyperparameters. When we plot the data and compare it to the real situation:



Interesting results indeed. We can see why we got such bad metric values - we are missing one complete cluster. This result is especially interesting if we take into consideration that this is done based on the density.

3. Regularization

When we talk about the **performance** of the machine learning model, we always talk about some form of accuracy. We want our model to produce accurate predictions; that is why we do all this, right? During the training process of any machine learning model, we try to find the **optimal** function that will be able to best describe output values, i.e., we try to optimize the function *f(X)*:

$$Y = f(X) + e$$

During the training process, we calculate how well the model performs and modify the **parameters** of the *f(X),* so our result is closer to the real values of *Y*. While we are doing that, we calculate the **error** of our model. We put in the sample, calculate the error based on the real *Y* value and modify the parameters of the *f(X)*. The error produced this way is called the **reducible error** because it can be minimized and even completely removed (*not a good idea btw*). The other part of the equation from above is *e* – **irreducible error**.

This error comes from the fact that no matter how good our machine learning model is, *X* is not carrying all the information that we need to predict *Y* in every possible **scenario**. Our model is an approximation, after all, thus it will not be 100% accurate. As George Box said, *"All models are wrong, but some are useful"*. Essentially, all we can do is **optimize parameters** of *f(X)* and get the best possible **approximation**; we always try to optimize reducible error because there is nothing we can do about the irreducible error.

This leads us to use different functions for *f(X)*. Sometimes, as we were able to see, they are quite easy, understandable and produce a model that is easy to **interpret**. For example, the multiple linear regression output value *Y* as a combination of features of *X*:

$$Y \approx \beta_0 + \beta_1 * X_1 + \beta_2 * X_2 + \ldots + \beta_n * X_n$$

When we get values of parameters, we can say how each feature affects the final result. Like a recipe. However, if this model doesn't provide good enough results, we might want to use some **other** function, something that will better **fit** the data. So, we go and use more **complex** functions like *polynomial* linear regression. This gives us better results, but it is more **complex** and harder to explain. Also, the complex model might make **assumptions** that are not really correct, or pick up some patterns that are caused by **random chance** rather than by true **properties,** and make bad predictions on test/real data.

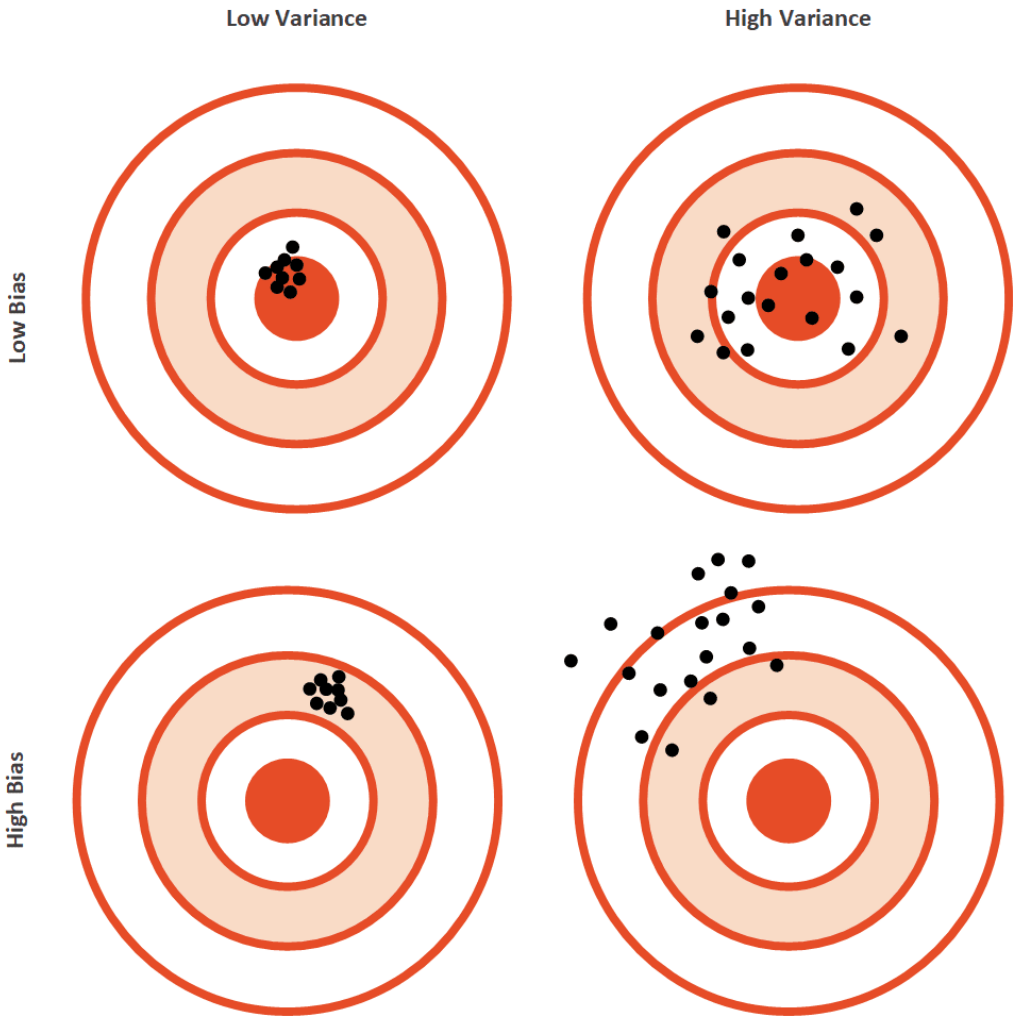
Here we can observe two possibilities: either we choose a model that is not that complex and get bad results on **training data**, or choose a complex model that might give bad results on **test** data. The first case is called **underfitting** – the inability of the machine learning model to predict the labels of the data it was trained on well. As mentioned, this happens when we use a model that is not complex enough, but it can happen if the features in the dataset are not informative enough.

The other case is **overfitting** – the model predicts very well on the training data but makes a lot of errors when applied to samples that were not processed by it during the training phase. These models have bad performance in the testing phase or during production. This happens if the model is too complex for the data or/and we have too many features but a small number of samples in the dataset.

In order to avoid both underfitting and overfitting, we need to represent the reducible error as a sum of **bias** and **variance**. This means that models complete the **generalization error** and it can be represented as:

$$Error = Bias + Variance + Irreducible$$

Bias is the error that is introduced when we are approximating a real-life problem, which may be extremely **complicated** with a **simple model**. This, of course, leads to underfitting, and we say that such models have a **high bias**. However, when a model shows high **sensitivity** to small variations in training data, we say that it has a **high-variance**. In essence, the **variance** can be defined as the **amount** by which your estimate of *f(X)* would change if we estimated it using a different training data set. High-variance leads to **overfitting** and it is the biggest problem with flexible (and complex) models.



In literature, this problem is often called a **bias-variance tradeoff**. We can define a rule like this – as a machine learning model tries to match data points more accurately, or when a more complex method is used, the **bias reduces**, but **variance increases**. In order to minimize the generalization error, we need to select a machine learning model that simultaneously achieves low variance and low bias.

One way to minimize the chance of **overfitting** is by using **regularization**. In general, this is an umbrella term that refers to different techniques that force the learning algorithm to build a less complex model by **constraining** it in one way or another. The idea is that if the model has less freedom, the harder it will be for it to overfit the data. In practice, this often leads to higher bias but reduces the variance. Some of the most popular regularization techniques constrain the model's weights/coefficients. In this chapter, we explore three such methods – **Lasso**, **Ridge** and **ElasticNet**. Also, we explore the **Early Stopping Technique**, which is a different regularization technique, but more on that later. Before going into more details of each technique, let's first load the data and **prepare** it for processing.

3.1 Preparing the Data

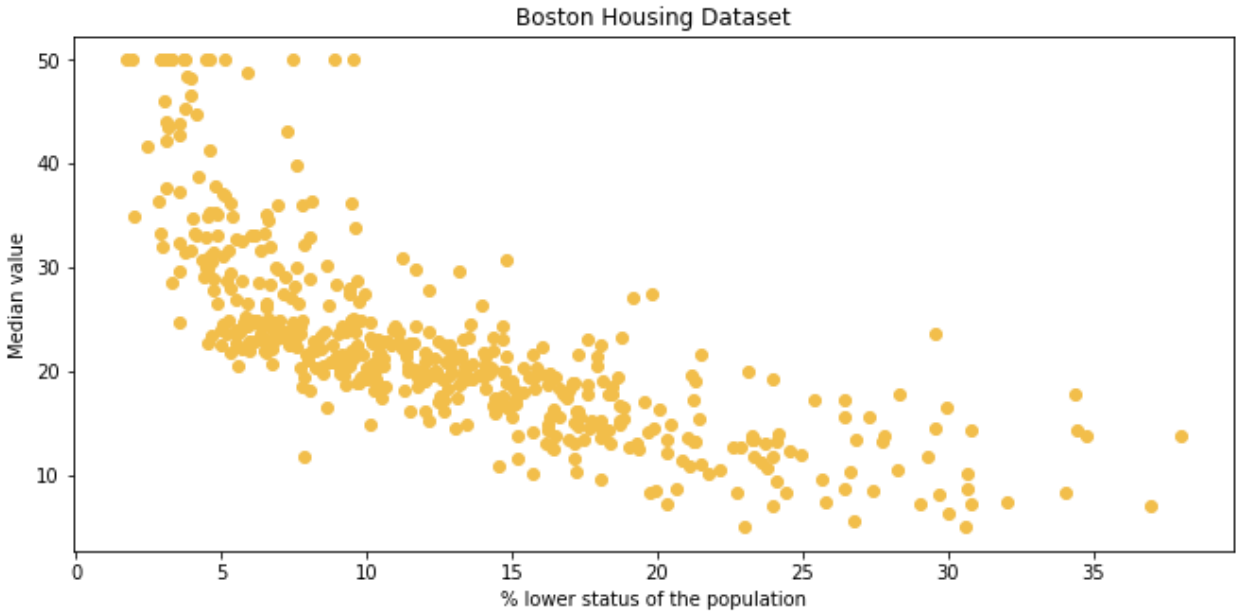
The Boston Housing Dataset is located in the .csv file, so let's load it:

```
data = pd.read_csv('./data/boston_housing.csv')
data.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

In this article, we use only the *'lstat'* feature as an input and the '*medv*' feature as an output. Here is what that looks like:

```
data = data.dropna()

X = data['lstat'].values
X = X.reshape(-1, 1)

y = data['medv'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)

plt.figure(figsize=(11, 5))

plt.title("Boston Housing Dataset")
plt.xlabel('% lower status of the population')
plt.ylabel('Median value')

plt.scatter(x = X, y = y, color=ORANGE)
plt.show()
```



Also, let's prepare the function that we use to print out all regression metrics for some predictions:

```
def print_evaluation_metrics(actual_values, predictions):
    print (f'MAE: {metrics.mean_absolute_error(actual_values, predictions)}')
    print (f'MSE: {metrics.mean_squared_error(actual_values, predictions)}')
    print(f'RMSE: {sqrt(metrics.mean_squared_error(actual_values,
predictions))}')
    print (f'R Squared: {metrics.r2_score(actual_values, predictions)}')
```

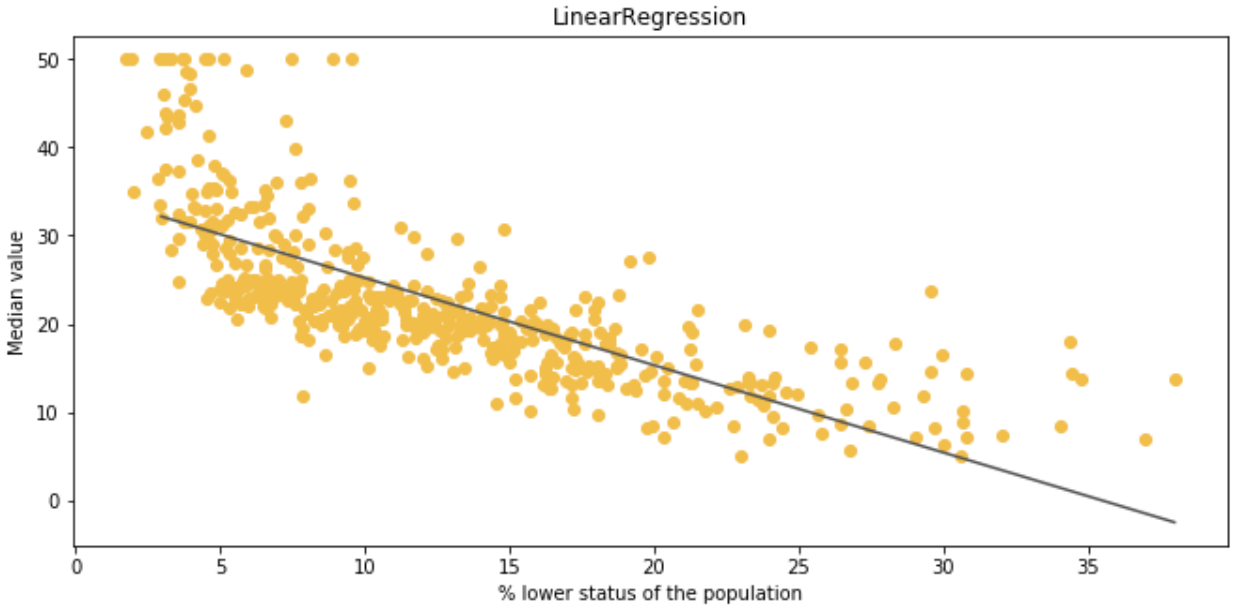
In one of the previous chapters, we applied the **linear regression** to this data and here is what we got:

```
linear_reg = make_pipeline(StandardScaler(), LinearRegression())
linear_reg.fit(X_train, y_train)
linear_reg_predictions = linear_reg.predict(X_test)

print_evaluation_metrics(y_test, linear_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'Linear Regression Prediction': linear_reg_predictions,
})
```

```
MAE: 4.349924284477385
MSE: 33.71853518466314
RMSE: 5.80676632771314
R Squared: 0.5292973354432082
```

	Actual Value	Linear Regression Prediction
0	20.5	17.345491
1	5.6	8.581341
2	13.4	12.031481
3	12.6	18.822707
4	21.2	25.653589
...	...	...
97	25.0	25.395820
98	19.5	16.621755
99	19.9	18.971420
100	15.4	13.855694
101	21.7	22.233191

Here is what that looks like when we **plot** it out:



3.2 Ridge Regression

Ridge Regression, also known as the *Tikhonov regularization* or *L2 norm*, is a modified version of Linear Regression where the cost function is modified by adding the "*shrinkage quality*". In essence, the **regularization term** is added to the cost function:

$$\alpha * \sum_{i=1}^{n} \theta_i^2$$

This is done during the **training** process (not during testing, just during training). This forces the machine learning algorithm to better **fit** data and to keep the model as simple as possible because the linear regression coefficients are optimized in a way to **minimize** this modified function.

Notice that this additional term prevents coefficients from **rising** too high. Apart from that, notice that α drives the level of regularization and that we need to be careful when we pick it. If it is close to 0, the regularization has no effect, but as $\alpha \to \infty$ regression, coefficients are getting closer to 0. So, if we use the *MSE* (mean squared error) as a loss function, the Ridge regression cost function would be:

$$C = MSE + \alpha * \sum_{i=1}^{n} \theta_i^2$$

Note that the bias term *θ0* is **not regularized** because the sum starts from *i=1*, not from *i=0*. In essence, the regularization term can be defined with the formula:

$$\frac{1}{2}(\|w\|_2)^2$$

where *w* is the vector of feature coefficients (from *θ1* to *θn*) and $\|\cdot\|2$ represents the **L2 norm**. That is why this regularization technique is often called the L2 norm. The cool thing about it is that this function is **differentiable**, so we can use the Gradient descent for optimizing the objective function. Another thing that we need to be careful about is that our data has to be **scaled** if we want to use this type of regularization.

3.2.1 Using Sci-Kit Learn

Let's see how we can use this technique with **Sci-Kit Learn**. As we mentioned, when we use different values for ***alpha***, we get different results. In this example, we use the value 10 for this **hyperparameter**:

```
ridge_reg = make_pipeline(StandardScaler(), Ridge(alpha=1, solver="cholesky"))
ridge_reg.fit(X_train, y_train)
ridge_reg_predictions = ridge_reg.predict(X_test)

print_evaluation_metrics(y_test, ridge_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'Ridge Prediction': ridge_reg_predictions,
})
```

	Actual Value	Ridge Prediction
0	20.5	17.359049
1	5.6	8.616538
2	13.4	12.058160
3	12.6	18.832617
4	21.2	25.646633
...	...	...
97	25.0	25.389500
98	19.5	16.637100
99	19.9	18.980963
100	15.4	13.877868
101	21.7	22.234680

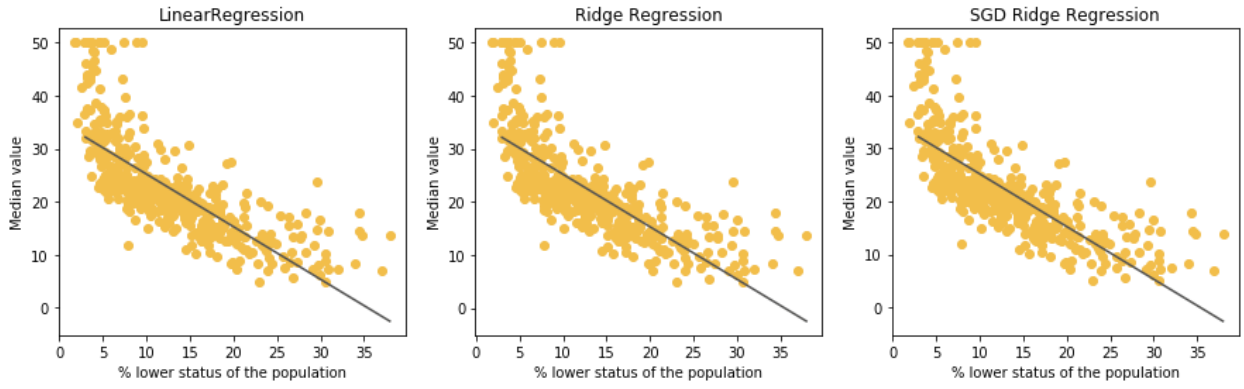
We can see that results are a bit better for higher values. As mentioned, this can be used with the stochastic gradient descent regression like this:

```
sgd_ridge_reg = make_pipeline(StandardScaler(), SGDRegressor(penalty="l2"))
sgd_ridge_reg.fit(X_train, y_train)
sgd_ridge_reg_predictions = sgd_ridge_reg.predict(X_test)

print_evaluation_metrics(y_test, sgd_ridge_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'SGD Ridge Prediction': sgd_ridge_reg_predictions,
})
```

```
MAE: 4.357165500878059
MSE: 33.78949776374881
RMSE: 5.812873451551204
R Squared: 0.5283067148579271
```

	Actual Value	SGD Ridge Prediction
0	20.5	17.357584
1	5.6	8.572811
2	13.4	12.031070
3	12.6	18.838276
4	21.2	25.685232
...	...	...
97	25.0	25.426856
98	19.5	16.632145
99	19.9	18.987339
100	15.4	13.859575
101	21.7	22.256785

Let's see how this compares to linear regression when we plot the results:



As we can see the results are close to each other.

3.3 Lasso Regression

The **Least Absolute Shrinkage and Selection Operator Regression** or the **Lasso** regularization technique follows similar principles as Ridge. It adds the **regularization terms** to the cost function, but it uses *L1 Norm*. The regularization parameter is $\alpha\Sigma|\theta i|$, meaning that if we use MSE as a cost function, the new cost function looks like this:

$$C = MSE + \frac{1}{2}\sum_{i=1}^{n}|\theta_i|$$

From the equation, you can see that this approach penalizes only **high** coefficients. The really interesting thing about this regression is its ability to completely eliminate the coefficients of the **least important** features. This means that it does some kind of feature selection, by choosing features that are **essential** to predictions. Eventually, it produces a sparse model, which increases model **explainability**. The downside is that Lasso regression function is not differentiable at $\theta i = 0$ *(for* $i = 1, 2, \cdots, n$*)*. However, we can still use the Gradient Descent with a **subgradient vector**.

3.3.1 Using Sci-Kit Learn

We use the **Lasso** from *Sci-Kit Learn* to perform this operation on the *Boston Housing* dataset:

```
lasso_reg = make_pipeline(StandardScaler(), Lasso(alpha=1))
lasso_reg.fit(X_train, y_train)
lasso_reg_predictions = lasso_reg.predict(X_test)

print_evaluation_metrics(y_test, lasso_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'Lasso Prediction': lasso_reg_predictions,
})
```

```
MAE: 4.191410465618733
MSE: 32.00917258613768
RMSE: 5.657664941134079
R Squared: 0.5531596273670172
```

	Actual Value	Lasso Prediction
0	20.5	18.143094
1	5.6	10.652017
2	13.4	13.600993
3	12.6	19.405730
4	21.2	25.244364
...	...	...
97	25.0	25.024038
98	19.5	17.524487
99	19.9	19.532842
100	15.4	15.160222
101	21.7	22.320810

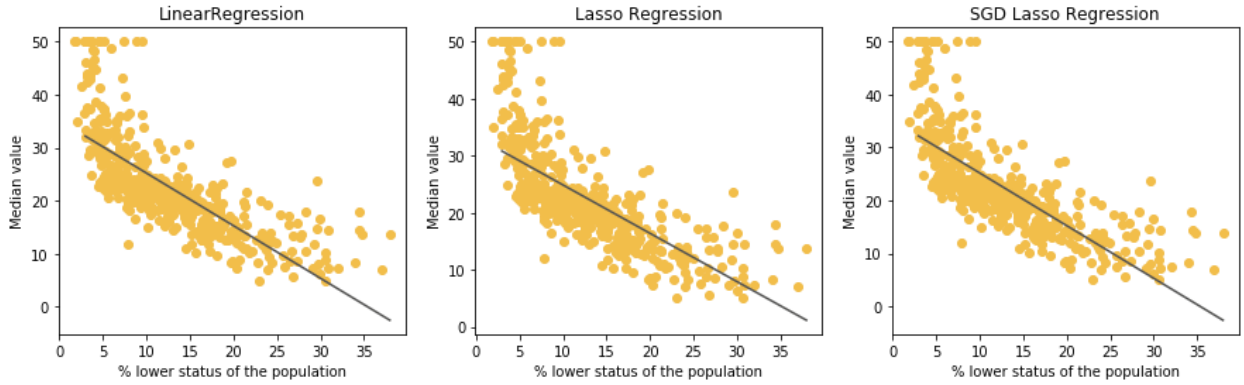
We can see that the results are different and interesting. When we use it with the *Stochastic Gradient Descent*, we get this:

```
sgd_lasso_reg = make_pipeline(StandardScaler(), SGDRegressor(penalty="l1"))
sgd_lasso_reg.fit(X_train, y_train)
sgd_lasso_reg_predictions = sgd_lasso_reg.predict(X_test)

print_evaluation_metrics(y_test, sgd_lasso_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'SGD Lasso Prediction': sgd_lasso_reg_predictions,
})
```

```
MAE: 4.351841291760927
MSE: 33.76185189745489
RMSE: 5.810494978696298
R Squared: 0.5286926445211602
```

	Actual Value	SGD Lasso Prediction
0	20.5	17.332252
1	5.6	8.563336
2	13.4	12.015353
3	12.6	18.810271
4	21.2	25.644868
...	...	...
97	25.0	25.386959
98	19.5	16.608122
99	19.9	18.959065
100	15.4	13.840557
101	21.7	22.222610

Finally, let's plot out the compared results:



3.4 Elastic Net

However, what should we do if neither of these options produces the aimed result? Well, then we can use **both** of them. Kinda. The **Elastic Net** is a regularization technique that combines *Lasso* and *Ridge*. Both regularization terms are **added** to the cost function, with one additional **hyperparameter** *r*. This hyperparameter controls the Lasso-to-Ridge ratio. In a nutshell, if *r = 0*, the Elastic Net performs the *Ridge* regression, and if *r = 1*, it performs the *Lasso* regression. This means that the cost function of the Elastic Net can be represented as (taken that MSE is the cost function we want to use):

$$C = MSE + r\sum_{i=1}^{n}\theta_i^2 + \frac{1-r}{2}\sum_{i=1}^{n}|\theta_i|$$

3.4.1 Using Sci-Kit Learn

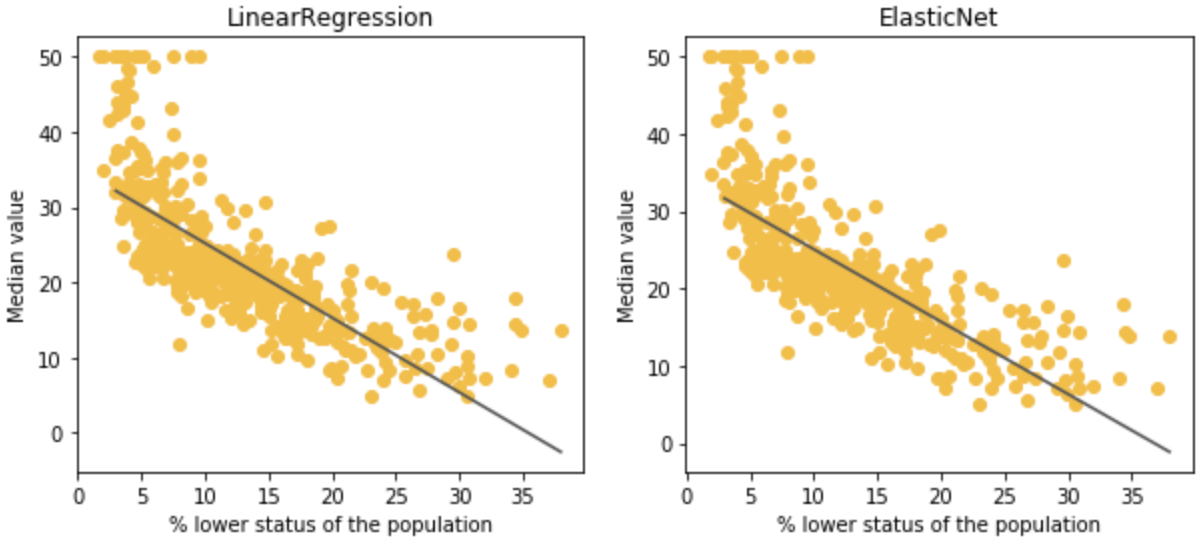
Here is how we use it with *Sci-Kit Learn*:

```
elastic_net_reg = make_pipeline(StandardScaler(), ElasticNet(alpha=0.1,
l1_ratio=0.5))
elastic_net_reg.fit(X_train, y_train)
elastic_net_reg_predictions = elastic_net_reg.predict(X_test)

print_evaluation_metrics(y_test, elastic_net_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'ElasticNet Prediction': elastic_net_reg_predictions,
})
```

```
MAE: 4.279460546855516
MSE: 32.776048508799335
RMSE: 5.725036987548581
R Squared: 0.5424542234043481
```

	Actual Value	ElasticNet Prediction
0	20.5	17.644944
1	5.6	9.358756
2	13.4	12.620739
3	12.6	19.041597
4	21.2	25.499949
...	...	...
97	25.0	25.256238
98	19.5	16.960677
99	19.9	19.182200
100	15.4	14.345466
101	21.7	22.266087

And plot it out:



In general, it is good to know that it is always good to have at least a bit of regularization. This means that we should avoid using **pure** Linear Regression. *Ridge* is the way to go if we want good **performance,** and it is a great **default**. If we suspect that there might be features that are useless in our dataset, we can go with the *Elastic Net* or the *Lasso*.

3.5 Early Stopping

Another regularization technique is **Early Stopping**. This is a non-mathematical technique that Geoffrey Hinton called a "*beautiful free lunch*". The main trick of this technique is to **stop training** as soon as the **validation error** reaches the **minimum**. Essentially, the model is saved after each epoch and the performance assessment of the model on the validation set. When we use the Gradient Descent, the cost should decrease after each epoch and with it, the validation error as well. Theoretically, at least.

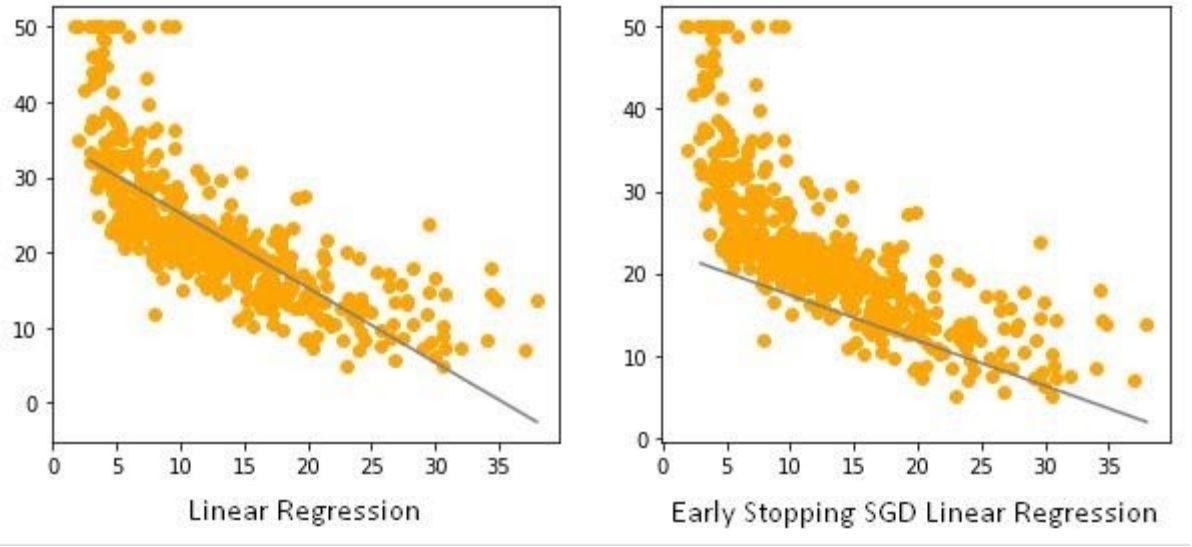
However, if the model starts **overfitting**, the validation error can start **increasing,** which is an indication that the performance of the model **decreases**. There are two ways that this technique can be implemented. The first one is to **stop the training** right after the validation error starts increasing. The other way is training the model for a fixed number of epochs, saving models at each epoch (**checkpoint**) and then picking the one with the **best** results. The second approach is typical for training neural networks. As you already know, machine learning algorithms in the *Sci-Kit Learn* Library have **hyperparameters** that you can use to set up this process easily. Let's see where we will land with the *Boston Housing Dataset:*

```
sgd_reg = SGDRegressor(penalty="elasticnet", early_stopping = True, alpha=11)
X_train = StandardScaler().fit_transform(X_train)

sgd_reg.fit(X_train, y_train)
sgd_reg_predictions = sgd_reg.predict(X_test)

print_evaluation_metrics(y_test, sgd_reg_predictions)

pd.DataFrame({
    'Actual Value': y_test,
    'SGD with Early Stopping': sgd_reg_predictions,
})
```

```
MAE: 5.432452178486986
MSE: 60.46609011128903
RMSE: 7.775994477318578
R Squared: 0.15590788345810025
```

	Actual Value	SGD with early stopping
0	20.5	12.977140
1	5.6	8.107905
2	13.4	10.024753
3	12.6	13.797860
4	21.2	17.592999
...	...	...
97	25.0	17.449786
98	19.5	12.575043
99	19.9	13.880483
100	15.4	11.038259
101	21.7	15.692675

We got some really weird results like this; just look at the plot:



Ugh! As it turned out, we need additional experiments with this approach and further hyperparameter tuning, or it might not be a good approach to our problem at all. Anyhow, now you know the meaning of this process and how to use it with *Sci-Kit Learn*.

4. Optimization

In general, every machine learning algorithm is composed of three integral parts:

1. A **loss** function.
2. Optimization criteria based on the loss function, like a **cost** function.
3. **Optimization** technique – this process leverages training data to find a solution for optimization criteria (cost function).

As you were able to see in the previous chapters, some algorithms were created intuitively and didn't have optimization criteria in mind. In fact, mathematical **explanations** of why and how these algorithms work were done later. Some of these algorithms are **Decision Trees** and **kNN**. Other algorithms, which were developed later, had this thing in mind beforehand. **SVM** is one example.

During the training, we change the parameters of our machine learning model to try and

minimize the loss function. However, the question of how you change those parameters arises. Also, how much should we change them during training and when? To answer all these questions, we use **optimizers**. They put all different parts of the machine learning algorithm together.

4.1 Cross-Entropy

While we are training a model, we give certain samples as inputs and as outputs and we get an array of probabilities. For example, if we put a sample into our model and we get an output with three numbers, then each of them represents the probability of a single class, i.e., *y'= [0.4, 0.5, 0.1]*. If we use supervised learning, we know that this differs from the expected value *y = [0 1 0]*. To get better, the model changes parameters to get from *y'* to *y*. However, this leaves us with several questions, like "What does getting better actually mean?", "What is the measure or quantity that tells me how far y' is from y?" and "How much should I tweak the parameters in my model?" Cross-entropy is a possible solution; one possible tool for this. It tells us how badly our model is doing, meaning it tells us in which "direction" we should tweak the parameters of the model.

To understand cross-entropy, we need to travel back to 1948 when the mathematician and electrical engineer *Claude Shannon* was trying to figure out ways to send messages without losing any information. He was thinking in terms of an average message length, meaning that he tried to encode a message using the smallest number of bits. Apart from that, he assumed that the decoder should be able to restore that message losslessly, meaning there should be no loss of information at all. That is how he invented the concept of entropy in his paper entitled *"A Mathematical Theory of Communication"*.

Imagine you have some source of the message that sends an encoded message to the destination. In this situation, the encoding must be done in a way that no information is lost during the transmission. Entropy is defined as the minimum average encoding size per transmission that guarantees that no data will be lost in the process. Mathematically, we can use probability distribution to define entropy (denoted as H). If we are talking about discrete variables, that formula looks something like this:

$$H(P) = -\sum P(i)\log(P(i))$$

When we consider numerical variables, we use the integral form:

$$H(P) = -\int P(x)\log(P(x))$$

x is a quantitative variable, and *P(x)* is the probability density function.

Hope you can anticipate where we are going with this. In our example from the beginning of the chapter, what we get as output are the probabilities of the class of image we got on the input; e.g., we get the probability distribution. This can be observed as our encoding tool. Basically, we use probability distribution as a means to encode input. Our optimal tool would be entropy. In this case, distribution *y*. However, we have the distribution *y'*. This means that Cross-entropy can be defined as the number of bits we need to encode information from *y* using the wrong encoding tool *y'*. Mathematically, this can be written like this:

$$H(y, y') = -\sum y \cdot \log(y')$$

The other way to write this expression is using expectation:

$$H(y, y') = E_{x \sim y}[-\log(y'(x))]$$

H(y, y') represents an expectation using y and the encoding size using *y'*. From this, we can conclude that *H(y, y')* and *H(y', y)* are not the same, except when *y = y'*; e.g., this calculation becomes entropy itself. Now, entropy is the theoretical minimum average size and the cross-entropy is higher than or equal to entropy, but not less than that.

To sum up, entropy is the optimal distribution that we want to get on our output. However, we get another distribution – cross-entropy, which is always larger than entropy. Now, all we need to do is to comprehend the difference between them so that we can improve our model. Here we need to introduce one more term: the *Kullback–Leibler* divergence.

4.2 KL Divergence

This term was coined by Solomon Kullback and Richard Leibler back in 1951 as the directed divergence between two distributions. Kullback preferred the term '*discrimination information*'. This topic is heavily discussed in *Kullback's 1959* book called *Information Theory and Statistics*. Essentially, the KL divergence is a difference between cross-entropy and entropy. It can be written down like this:

$$KL(y||y') = -\sum y \cdot \log(y) + \sum y \cdot \log(y') = \sum y \cdot \log\left(\frac{y}{y'}\right)$$

We can say that it measures the number of extra bits we'll need on average if we encode the output with *y'* instead of y. This value is never negative, and by optimizing cross-entropy, we are trying to get as close to 0 as possible. This means that by minimizing cross-entropy, we are minimizing the KL divergence.

To sum up, we'll be using our example from the beginning. During our training process, we put an image of Samuel L. Jackson in our output. However, we don't get the correct label for it but some probabilities for each class of the image. For example, instead of *y = [0 0 1]* we get something like this *y' = [0.1 0.2 0.7]*. This means that instead of the perfect encoding (entropy) *y*, we got the imperfect encoding (cross-entropy) *y'*. Using these values, we calculate the KL divergence and we aim to minimize this value. That is how we know how to modify the parameters of our model.

4.3 Optimization Algorithms

So far, we mentioned the **Gradient Descent** as an optimization technique, but we haven't explored it in more detail. In this article, we focus on that and we cover the **grandfather** of all optimization techniques and its variation. Note that these techniques are **not** machine learning algorithms. They are solvers of **minimization** problems in which the function to minimize has a gradient in most points of its domain.

Note that we also use simple **Linear Regression** in all examples. Due to the fact that we explore **optimization techniques**, we picked the easiest machine learning algorithm. As a quick reminder, the formula for linear regression goes like this:

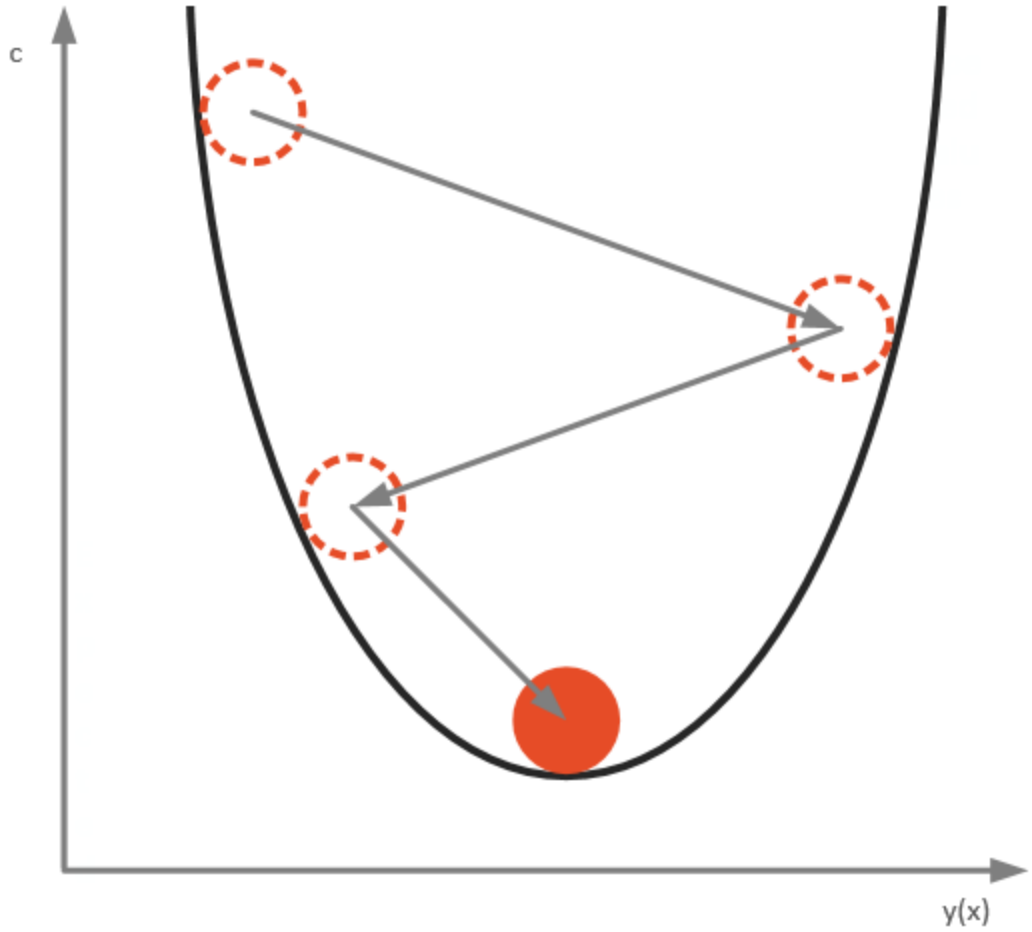
$$f(X) = w * X + b$$

where *w* and *b* are parameters of the machine learning algorithm. The entire point of the training process is to set the correct values to the *w* and *b*, so we get the desired output from the machine learning model. This means that we are trying to make the value of our **error vector** as small as possible, i.e., to find a **global minimum of the cost function**.

One way of solving this problem is to use calculus. We could compute derivatives and then use them to find places where an extremum of the cost function is. However, the cost function is not a function of one or a few variables; it is a function of all parameters of a machine learning algorithm, so these calculations will quickly grow into a monster. That is why we use these optimizers.

4.3.1 Gradient Descent and its Variations

We already had a chance to learn about the Gradient Descent basics. However, we haven't explored details. So, let's see how the Gradient Descent works with Linear Regression.



In a nutshell, the training process of Linear Regression with the Gradient Descent can be described like this:

1. put the training set in the machine algorithm and get the output for each sample in it;
2. the output is compared with the desired output and error is calculated using a cost function;
3. based on the error value and used cost function, a decision on how the *w* and *b* should be changed is made in order to minimize the error value;
4. the process is repeated until the error is minimal.

Let's formalize this in more mathematical terms. We don't know what the optimal values for *w* and *b* are in the Linear Regression formula:

$$f(X) = w * X + b$$

We use Mean Squared Error as the loss function:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - (w * x_i + b))^2$$

where N is the number of samples in the dataset, *yi* is the real output value and *xi* is the input vector (where each feature is represented with a separate coordinate). The first step in the Gradient Descent would be to define the **partial derivatives** for each parameter. For *w*, we use the chain rule and get the result:

$$\frac{\partial MSE}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (w * x_i + b))$$

and for *b* we get:

$$\frac{\partial MSE}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} -2(y_i - (w * x_i + b))$$

Once we know how to calculate partial derivatives for all the parameters in the machine learning model (in this case *w* and *b*) for the defined cost function (in this case *MSE),* then we can initialize values for those parameters (*w* and *b)*. The initialization process is a completely different topic outside of the scope of this tutorial. So, in this chapter, we will initialize those values to 0. Then we start iteration through the training set examples and update *w* and *b* by utilizing partial derivatives after each sample:

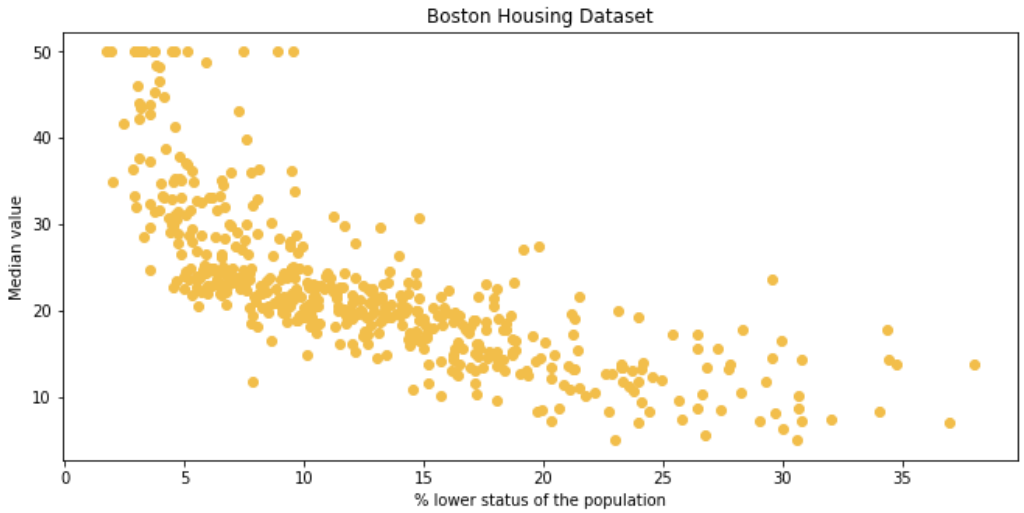
$$w_i \leftarrow \alpha \frac{-2x_i(y_i - (w_{i-1} * x_i + b_{i-1}))}{N}$$
$$b_i \leftarrow \alpha \frac{-2(y_i - (w_{i-1} * x_i + b_{i-1}))}{N}$$

where alpha is the learning rate hyperparameter. This hyperparameter controls how "strong" an update is. When we go through all examples in the training set, we call that an **epoch**. In general, we train our machine learning algorithms for multiple epochs.

4.3.1.1 Prepare Data

Before we dwell into details and implement the Gradient Descent, let's prepare data. We perform the same preparation steps on the Boston Housing Dataset as we did for Linear Regression.

```
data = pd.read_csv('./data/boston_housing.csv')
data = data.dropna()

scaler = StandardScaler()

X = data['lstat'].values
X = X.reshape(-1, 1)
X = scaler.fit_transform(X)

y = data['medv'].values
y = y.reshape(-1, 1)
y = scaler.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```



4.3.1.2 Python Implementation

Ok, that's enough theory; let's implement this with *Python.* The complete implementation is done within *MyGradientDescent* class:

```
class MyGradientDescent():
    def __init__(self, learning_rate):
        self.learning_rate = learning_rate
        self.w = 0
        self.b = 0

    def fit(self, X, y, epochs = 100):
        N = len(X)
        history = []

        for e in range(epochs):
            for i in range(N):
                Xi = X[i, :]
                yi = y[i, :]

                f = yi - (self.w * Xi + self.b)

                self.w -= self.learning_rate * (-2 *
Xi.dot(f.T).sum() / N)
                self.b -= self.learning_rate * (-2 * f.sum() / N)

            loss = mean_squared_error(y, (self.w * X + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

It is a pretty simple class. Note that the name of this class is maybe not completely accurate. In essence, we created an algorithm that uses *Linear regression* with *Gradient*

Descent. This is important to say. Here the algorithm is still *Linear Regression*, but in the method that helped us, we learn *w* and *b* are the **Gradient Descent**. We could switch to any other learning algorithm.

In the constructor of the class, we initialize the value of *w* and *b* to zero. Also, we initialize the **learning rate hyperparameter**. There are two public methods:

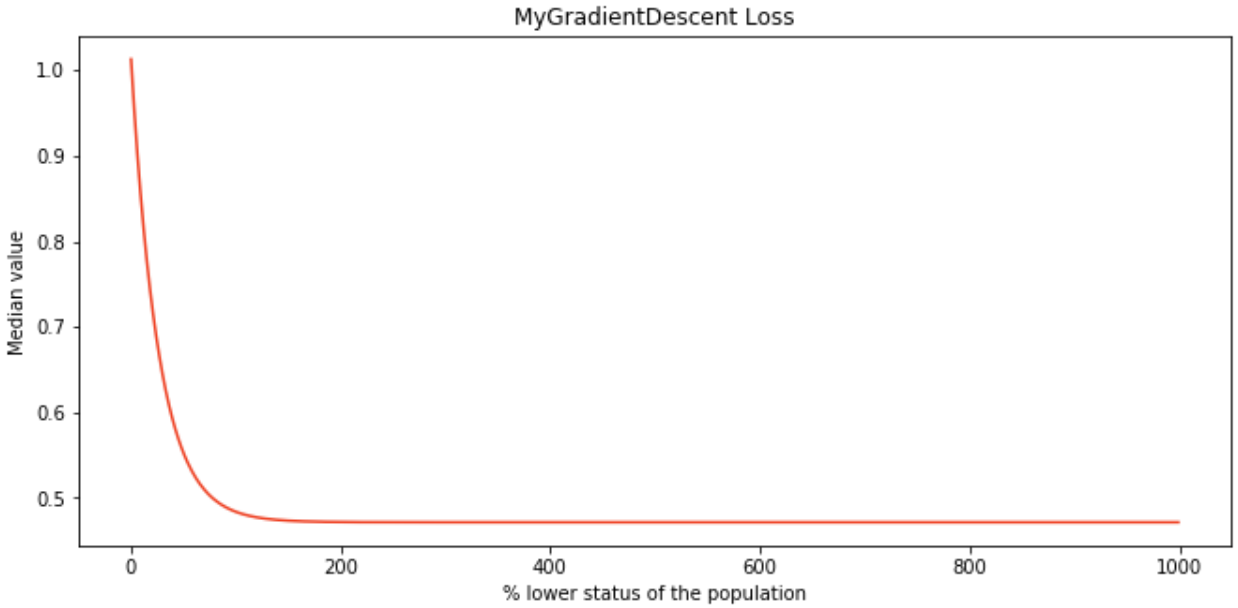
- **fit** – This is the method that performs the training process. Note that it returns history where we record the loss during training. This is done for future visualizations. Apart from that, for updating *w* and *b* we used formulas defined above.
- **predict** – Method that predicts value.

Note that for the loss, we use the MSE and in the code, we use the Sci-Kit *learn* function to calculate it. Let's utilize this class on the data:

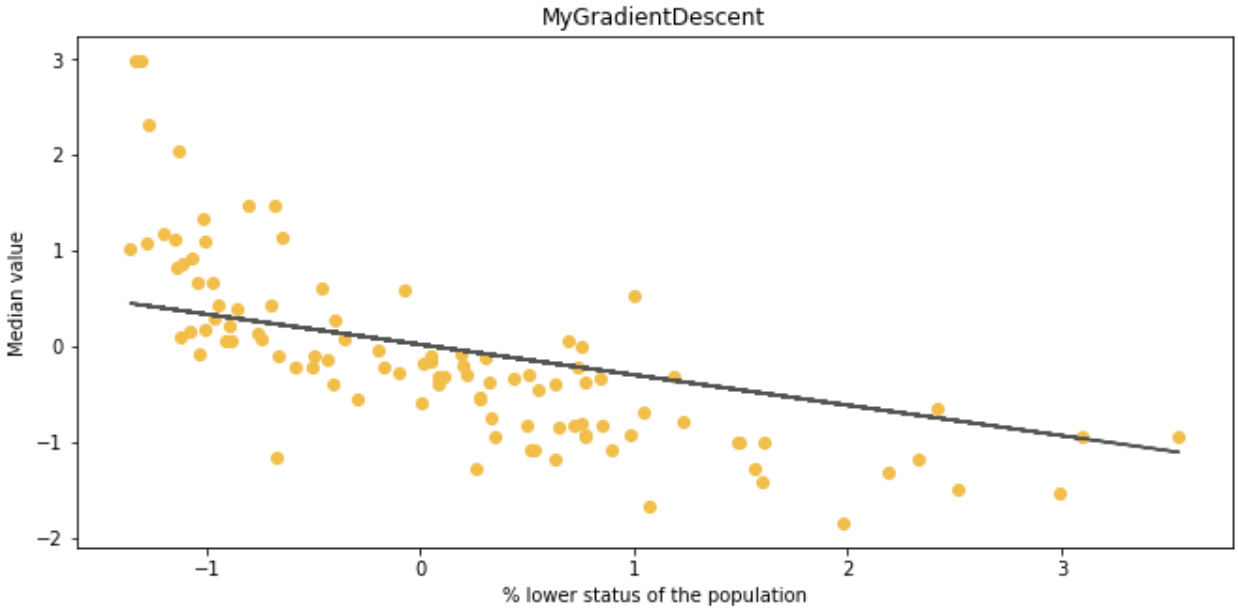
```
model = MyGradientDescent(learning_rate = 0.01)
history = model.fit(X_train, y_train, 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.0129236386713547)
Epoch: 100, Loss: 0.4836365268936342)
Epoch: 200, Loss: 0.4717052013642623)
Epoch: 300, Loss: 0.4714309129514397)
Epoch: 400, Loss: 0.4714244612434121)
Epoch: 500, Loss: 0.47142429776534295)
Epoch: 600, Loss: 0.4714242918194498)
Epoch: 700, Loss: 0.4714242913200272)
Epoch: 800, Loss: 0.47142429124768254)
Epoch: 900, Loss: 0.47142429123592733)
```

We can see how our algorithm is diverging because the loss is slowly going down. Once it is done, we can plot the history and see how the loss function decreased during the training process:



Another thing we can do is plot the final model:



Also, we can compare real values with predictions (keep in mind that data is **scaled**):

	Actual Value	Predictions
0	-0.221246	-0.564486
1	-1.842924	-1.518349
2	-0.993992	-1.142846
3	-1.081062	-0.403710
4	-0.145059	0.339743
...	...	...
97	0.268523	0.311688
98	-0.330083	-0.643255
99	-0.286548	-0.387525
100	-0.776317	-0.944305
101	-0.090641	-0.032523

The biggest problem of the Gradient Descent is that it can converge towards a **local** minimum and not to a **global** one. In our case, that is not a problem since the MSE is a convex function that has just one minimum – a **global** one. The other problem is that, for big datasets, this approach can take some time.

4.3.2 Batch Gradient Descent

However, in practice, it would be inefficient to calculate the partial gradient of the cost function for **each** small change in the *w* or *b*, for **each** sample. You could see that it is not the fastest approach. That is why simple optimization is done, i.e., a partial derivative is calculated for a complete set of parameters in **one go**. For MSE, that is done using the formula:

$$\frac{\partial MSE}{\partial \theta_j} = \frac{2}{N}\sum_{i=1}^{N} x_i^j(\theta^T * x_i - y_i)$$

$$\nabla MSE = \begin{pmatrix} \frac{\partial MSE}{\partial \theta_0} \\ \frac{\partial MSE}{\partial \theta_1} \\ \frac{\partial MSE}{\partial \theta_2} \\ \dots \\ \frac{\partial MSE}{\partial \theta_k} \end{pmatrix} = \frac{2}{N}X^T * (X * \theta - y)$$

where **θ** is the set of all machine learning parameters. In our case, **θ0** is *b* while other **θ** values come from *w.* This optimized version of gradient descent is called the **batch gradient descent** due to the fact that the partial gradient descent is calculated for the complete input X (i.e. batch) at each gradient step. This means that *w* and *b* can be updated using the formulas:

$$w_i = w_{i\_1} - \alpha \nabla MSE$$
$$b_i = b_{i\_1} - \alpha \nabla MSE$$

4.3.2.1 Python Implementation

The implementation of this algorithm is very similar to the implementation of the "vanilla" Gradient Descent. We do so in the class *MyBatchGradientDescent*:

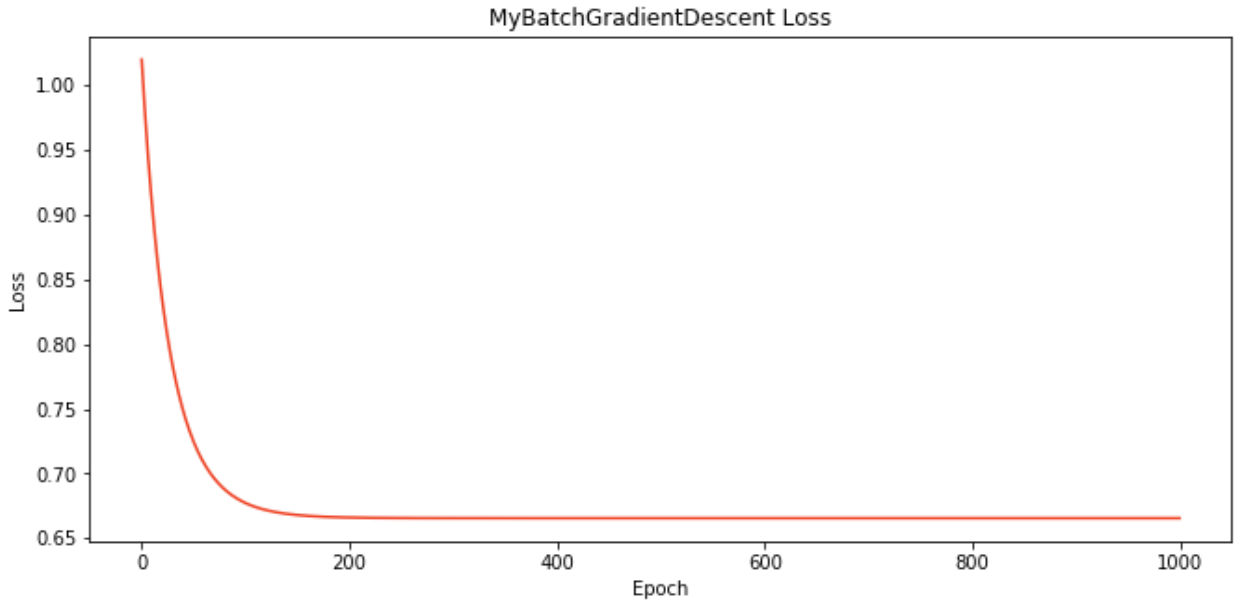
```
class MyBatchGradientDescent():
    def __init__(self, learning_rate):
        self.learning_rate = learning_rate
        self.w = 0
        self.b = 0

    def fit(self, X, y, epochs = 100):
        N = len(X)
        history = []

        for e in range(epochs):
            f = y - (self.w * X + self.b)

            self.w -= self.learning_rate * (-2 * X.dot(f.T).sum() /
N)
            self.b -= self.learning_rate * (-2 * f.sum() / N)

            loss = mean_squared_error(y, (self.w * X + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

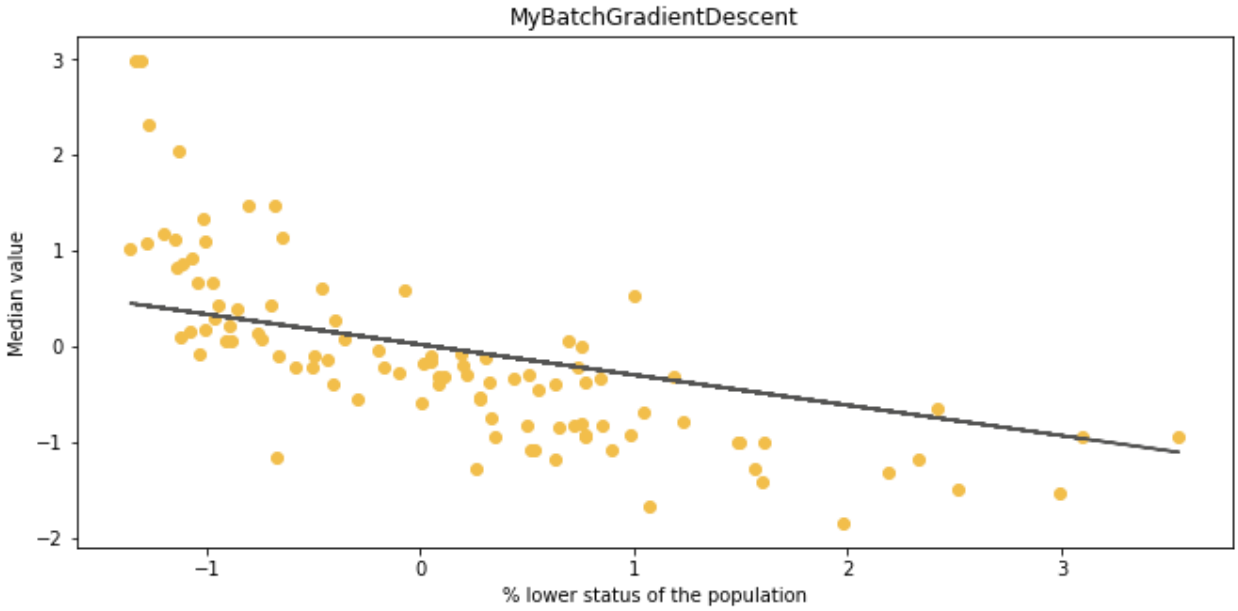
The only difference is the way we update *w* and *b*. When we fit our data in this algorithm, here is what we get:

```
model = MyBatchGradientDescent(learning_rate = 0.01)
history = model.fit(X_train, y_train, 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.0196646159305187)
Epoch: 100, Loss: 0.6771864255196849)
Epoch: 200, Loss: 0.6660374373514026)
Epoch: 300, Loss: 0.6655591433109558)
Epoch: 400, Loss: 0.6655383319190815)
Epoch: 500, Loss: 0.6655374258171037)
Epoch: 600, Loss: 0.6655373863654891)
Epoch: 700, Loss: 0.6655373846477667)
Epoch: 800, Loss: 0.6655373845729767)
Epoch: 900, Loss: 0.6655373845697203)
```

Even though this approach is faster, we got some different results with it; loss is a bit higher than the one when we used the simple gradient descent. Perhaps we should train it more. This difference can also be seen with the model that this approach created:





And in the results that it outputs:

	Actual Value	Predictions
0	-0.221246	-0.213164
1	-1.842924	-0.606138
2	-0.993992	-0.451438
3	-1.081062	-0.146928
4	-0.145059	0.159361
...	...	...
97	0.268523	0.147803
98	-0.330083	-0.245616
99	-0.286548	-0.140260
100	-0.776317	-0.369643
101	-0.090641	0.005994

4.3.3 Stochastic Gradient Descent

The **Stochastic gradient descent** (*SGD*) is an updated version of the Batch Gradient Descent algorithm that speeds up the computation by approximating the gradient using smaller subsets of the training data. These subsets are called mini-batches or just batches. Sometimes in literature, you will find that the Stochastic Gradient Descent is a version of the Gradient Dataset that picks one random sample from the input dataset and that **Mini-Batch Gradient Descent** takes a subset of samples from the input dataset. However, those formal lines are a bit blurred in the day-to-day work. If someone says that they use stochastic gradient descent, they are most probably referring to the one that uses the mini-batches. After all, one sample is just a subset with one element. In this chapter, we are using this context – that Stochastic Gradient Descent uses mini-batches which are the subset of the input dataset.

Since this algorithm works with considerably less data than Batch Gradient Descent, it is faster. Also, this means that this algorithm can be used on big datasets. However, due to its random nature, this process is much less **regularized**. The loss will not linearly go to its minimum, but it will bounce up and down until it **stabilizes** and diverges. Let's see what the implementation of this algorithm looks like in *Python*.

4.3.3.1 Python Implementation

Ok, the only thing that we need to improve from the previous implementation is to give the user of our class the ability to define the size of the batch. Here is how we do it in the class *MySGD*:

```
class MySGD():
    def __init__(self, learning_rate):
        self.learning_rate = learning_rate
        self.w = 0
        self.b = 0

    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def fit(self, X, y, batch_size = 32, epochs = 100):
        N = len(X)
        history = []

        for e in range(epochs):
            indexes = np.random.randint(N, size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            f = y_batch - (self.w * X_batch + self.b)
```

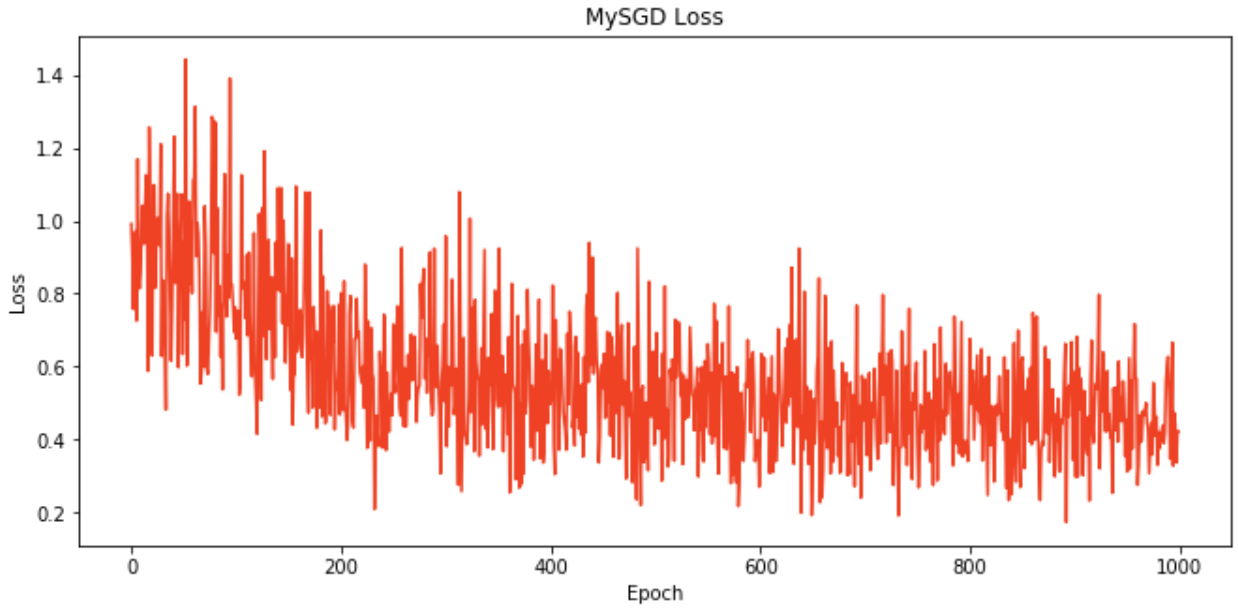
```
            self.w -= self.learning_rate * (-2 *
X_batch.dot(f.T).sum() / N)
            self.b -= self.learning_rate * (-2 * f.sum() / N)

            loss = mean_squared_error(y_batch, (self.w * X_batch +
self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

You can see one additional private function *_get_batch*. This method retrieves a random **subset** of input and output data for the training. It is the only difference from the *MyBatchGradientDescent* class. The *fit* method is modified to utilize this method and generate *X_batch* and *y_batch*, which are later on used in the training. Ok, let's try this on our dataset:

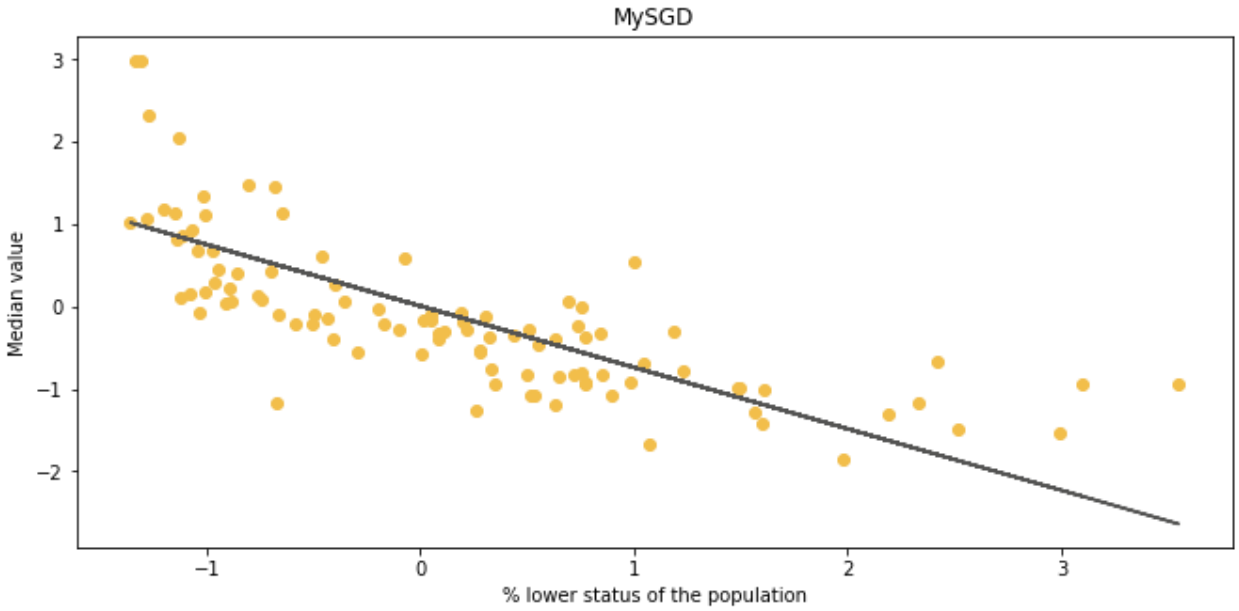
```
model = MySGD(learning_rate = 0.01)
history = model.fit(X_train, y_train, batch_size = 64, epochs = 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 0.6988926574320093)
Epoch: 100, Loss: 0.9368940114609687)
Epoch: 200, Loss: 0.38274490789600646)
Epoch: 300, Loss: 0.4015009827201944)
Epoch: 400, Loss: 0.6096862052672033)
Epoch: 500, Loss: 0.4814101552252322)
Epoch: 600, Loss: 0.3249126077150954)
Epoch: 700, Loss: 0.6309038218016978)
Epoch: 800, Loss: 0.6394188763053792)
Epoch: 900, Loss: 0.6521754136964174)
```

From the output, we can see that loss is going up and down, which we expected, but notice how the overall loss is going down. We can observe that when we plot the history:



Even though this seems a bit odd at first, observe what happens when we plot the predictions and compare them with the results we got from the Batch Gradient Descent:



We got a better approximation of the data! And with that, we got more accurate predictions:

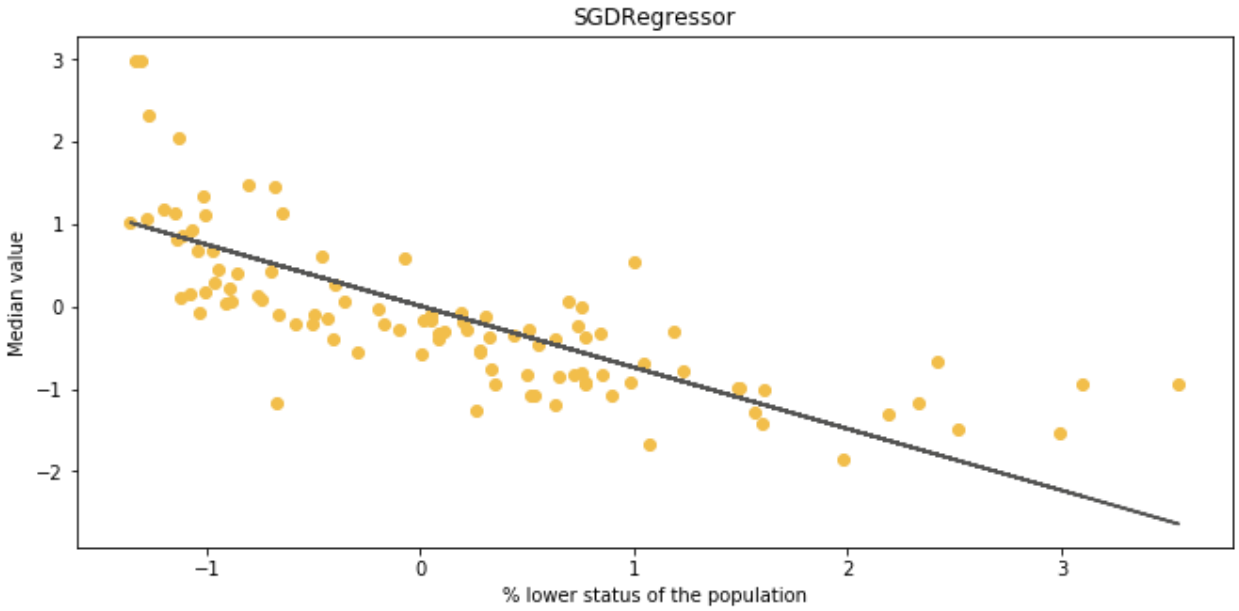
	Actual Value	Predictions
0	-0.221246	-0.529533
1	-1.842924	-1.428056
2	-0.993992	-1.074338
3	-1.081062	-0.378085
4	-0.145059	0.322235
...	...	...
97	0.268523	0.295808
98	-0.330083	-0.603732
99	-0.286548	-0.362838
100	-0.776317	-0.887316
101	-0.090641	-0.028433

4.3.3.2 Using Sci-Kit Learn

As it usually goes, everything we write from scratch already exists in the *Sci-Kit Learn* library, which makes our lives a lot easier. Let's use the **SGDRegressor** for our problem:

```
sgd_model = SGDRegressor(max_iter=10000, alpha=0.1)
sgd_model.fit(X_train, y_train)

sgd_predictions = sgd_model.predict(X_test)
```

Quite straightforward, nothing more than we have seen so far. Note that the *alpha* parameter is the learning rate and *max_iter* is the maximal number of epochs. When we plot the result, we get more or less the same thing as our implementation:



The same goes for the table of results:

	Actual Value	SGD Model Prediction
0	-0.221246	-0.509294
1	-1.842924	-1.373452
2	-0.993992	-1.033263
3	-1.081062	-0.363639
4	-0.145059	0.309896
...	...	...
97	0.268523	0.284479
98	-0.330083	-0.580656
99	-0.286548	-0.348976
100	-0.776317	-0.853393
101	-0.090641	-0.027360

4.3.4 Momentum Optimizer

In order to improve this, a new version of optimizers was born – **Momentum Optimizers**. Essentially, this **idea** was born back in 1974 by Boris T. Polyak and the optimizers are focused on helping models converge as **fast** as possible. The idea comes from observing a ball rolling down a gentle slope. It will start off slowly, but it will gain **momentum**. So, the core of the idea is to focus on movement in the correct direction. This is done by reducing the **variance** in every other insignificant direction, thus accelerating gradient descent.

Moment Optimization introduces the **momentum vector**. This vector is used to "store" changes in previous gradients. It helps **accelerate** the stochastic gradient descent in the relevant direction and dampens oscillations. At each gradient step, the local gradient is **added** to the momentum vector. Then, parameters are updated just by subtracting the momentum vector from the current parameter values. Since the whole idea kinda comes from physics, a new hyperparameter β is introduced – **momentum**. It simulates the friction mechanism and regulates the momentum value so it doesn't explode. Typically, this value is set to 0.9. To sum up, momentum optimization is performed in two steps:

1. Calculating the momentum vector at each iteration using the formula:

$$m \leftarrow \beta m + \alpha \nabla MSE$$

where m is the momentum vector, β is the momentum, α is the learning rate, θ is the set of machine learning parameters and ∇MSE is the partial derivative of the cost function (**Mean Squared Error**, in this case).

2. Update the parameters by subtracting the momentum vector.

$$\theta \leftarrow \theta - m$$

4.3.4.1 Python Implementation

We implement this algorithm within *MyMomentumOptimizer* class:

```
class MyMomentumOptimizer():
    def __init__(self, learning_rate, momentum = 0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum
```

```
        self.w = 0
        self.b = 0

        self.momentum_vector_w = 0
        self.momentum_vector_b = 0

    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def _get_momentum_vector(self, X_batch, y_batch):
        f = y_batch - (self.w * X_batch + self.b)

        self.momentum_vector_w = self.momentum * self.momentum_vector_w + \
              self.learning_rate * (-2 * X_batch.dot(f.T).sum() / len(X_batch))
        self.momentum_vector_b = self.momentum * self.momentum_vector_b + \
              self.learning_rate * (-2 * f.sum() / len(X_batch))

    def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            self._get_momentum_vector(X_batch, y_batch)

            self.w -= self.momentum_vector_w
            self.b -= self.momentum_vector_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

Ugh, that is a lot of code; let's segment it and explain piece by piece. In the **constructor**

of this class, we initialize the necessary attributes and set **hyperparameter** values. The learning rate and momentum are set, and algorithm parameters *w* and *b* are initialized to 0. The same goes for momentum vectors. Note that we could put all the parameters of the algorithm (*w and b*) within one array, but we wanted everything to be as clear as possible. The code can, of course, be improved.

```
def __init__(self, learning_rate, momentum = 0.9):
      self.learning_rate = learning_rate
      self.momentum = momentum
            self.w = 0
      self.b = 0
            self.momentum_vector_w = 0
      self.momentum_vector_b = 0
```

Two private methods **_get_batch** and **_get_momentum_vector**, are very important. The *_get_batch* method is used to pick the batch from the input and output datasets. The *_get_momentum_vector* method does the dirty work we defined in the momentum algorithm. Here, the gradient for each parameter is calculated and used to update the momentum vector for each parameter.

```
def _get_batch(self, X, y, batch_size):
      indexes = np.random.randint(len(X), size=batch_size)
      return X[indexes,:], y[indexes,:]

def _get_momentum_vector(self, X_batch, y_batch):
      f = y_batch - (self.w * X_batch + self.b)
            self.momentum_vector_w = self.momentum * self.momentum_vector_w +
\
                        self.learning_rate * (-2 * X_batch.dot(f.T).sum() /
len(X_batch))
      self.momentum_vector_b = self.momentum * self.momentum_vector_b + \
                        self.learning_rate * (-2 * f.sum() / len(X_batch))
```

Finally, in the *fit* method, the actual training is performed. For each epoch, this algorithm first fetches the batch and calculates the momentum vectors for each parameter. In the end, *w* and *b* are **updated** using these vectors. Apart from that, we calculate the loss, print it out, and store it in history. This is done just so we can follow what is happening during the **training** process. The method *predict* just predicts values for the input.

```
def fit(self, X, y, batch_size = 32, epochs = 100):
      history = []
            for e in range(epochs):
                        indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)
                        self._get_velocity(X_batch, y_batch)
                        self.w -= self.velocity_w
            self.b -= self.velocity_b
                  loss = mean_squared_error(y_batch, (self.w * X_batch +
self.b))

            if e % 100 == 0:
                  print(f"Epoch: {e}, Loss: {loss})")
                        history.append(loss)

      return history
                  def predict(self, X):
      return self.w * X + self.b
```

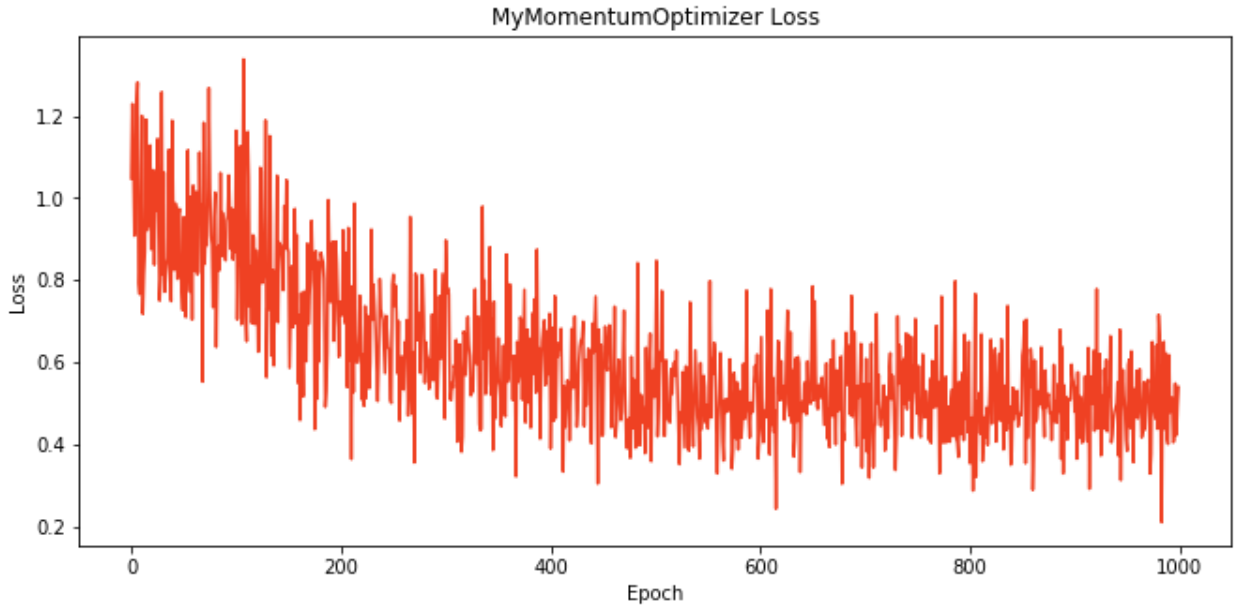
Now when we understand what is happening in this class, let's run this on loaded data:

```
model = MyMomentumOptimizer(learning_rate = 0.0001)
history = model.fit(X_train, y_train, batch_size = 128, epochs = 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.023970350020749)
Epoch: 100, Loss: 0.8915269977932345)
Epoch: 200, Loss: 0.6160710091923012)
Epoch: 300, Loss: 0.7748865586040383)
Epoch: 400, Loss: 0.7624729068019713)
Epoch: 500, Loss: 0.7386694964887148)
Epoch: 600, Loss: 0.4590078364639633)
Epoch: 700, Loss: 0.45645337845296935)
Epoch: 800, Loss: 0.548081143220535)
Epoch: 900, Loss: 0.5855308679385105)
```

We can note that this algorithm goes to the global minimum quite quickly, which is what we wanted. Also, note that it is almost the same as the momentum optimizer. It goes

fast "ahead" and then backs up. If we plot the loss history, we get something like this:



And if we plot the model it looks something like this:

Seems quite cool that we were able to get really good results with this approach, taken into consideration that we are using Linear Regression as our chosen algorithm.

4.3.5 Nesterov Accelerated Gradient

Yurii Nesterov noticed, back in 1983, that it is possible to improve momentum-based

optimization and make it go to the global minimum even **faster**. Let's use our ball analogy once again. Imagine having a smarter ball that will detect when it rolled over the minimum and slow down even more. That is an analogy that we can take when talking about the difference between classical momentum optimization and **Nesterov Accelerated Gradient.**

The trick is just to measure the gradient of the cost function "*in the future*" and adapt according to that. What does this mean? Well, since we know the momentum vector, we can calculate the gradient of the cost function slightly **ahead** in the **direction** of the momentum vector. This gives us an **approximation** of the next position of the parameters, i.e., a rough idea about future values of our parameters. Here is how we update the momentum vector then:

$$m \leftarrow \beta m + \alpha \nabla MSE(\theta + \beta m)$$

The meaning of all symbols is the same as in the previous formula. Essentially, the only difference is that we calculate the partial gradient of cost function not with regards to parameters θ, but with regards to $\theta + \beta m$. We update the parameters in the same way as for the classical momentum:

$$\theta \leftarrow \theta - m$$

This small change of perspective results in a faster algorithm than a simple momentum. This is especially useful when working with neural networks. Let's find out what that looks like in code.

4.3.5.1 Python Implementation

The implementation of this algorithm looks almost the same as the *MyMomentumOptimizer* class. Take a look at *MyNestrovAcceleratedGradient* class that contains that implementation:

```
class MyNestrovAcceleratedGradient():
    def __init__(self, learning_rate, momentum = 0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum

        self.w = 0
        self.b = 0
```

```
        self.momentum_vector_w = 0
        self.momentum_vector_b = 0

    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def _get_momentum_vector(self, X_batch, y_batch):
        f = y_batch - ((self.w + self.momentum * self.momentum_vector_w) *
X_batch + \
          (self.momentum * self.momentum_vector_b))

        self.momentum_vector_w = self.momentum * self.momentum_vector_w + \
                                 self.learning_rate * (-2 *
X_batch.dot(f.T).sum() / len(X_batch))

        self.momentum_vector_b = self.momentum * self.momentum_vector_b + \
                                 self.learning_rate * (-2 * f.sum() /
len(X_batch))

    def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []
        momentum_vector = np.zeros_like(1)

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            self._get_momentum_vector(X_batch, y_batch)

            self.w -= self.momentum_vector_w
            self.b -= self.momentum_vector_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

Ok, this is almost the same. The only difference can be seen in the function *_get_momentum_vector*, which is in charge of calculating the momentum vector when we calculate the loss function. For the classical momentum we use:

```
f = y_batch - (self.w * X_batch + self.b)
```

However, instead of that, for *Nesterov Accelerated Gradient* we use:

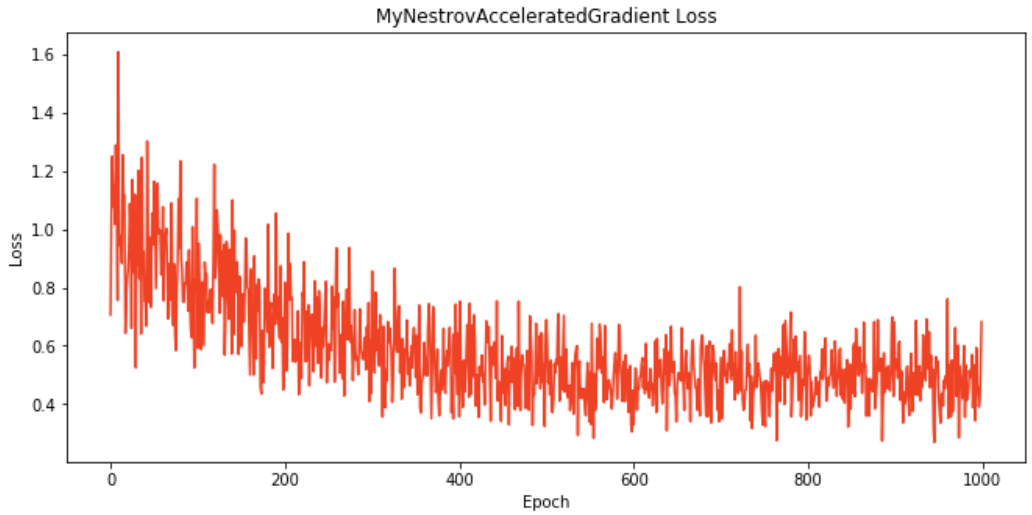
```
f = y_batch - ((self.w + self.momentum * self.velocity_w) * X_batch + \
    (self.momentum * self.velocity_b))
```

Apart from that, the rest of implementation is the same. The API is the same so we can use it just like the other algorithms from this series:

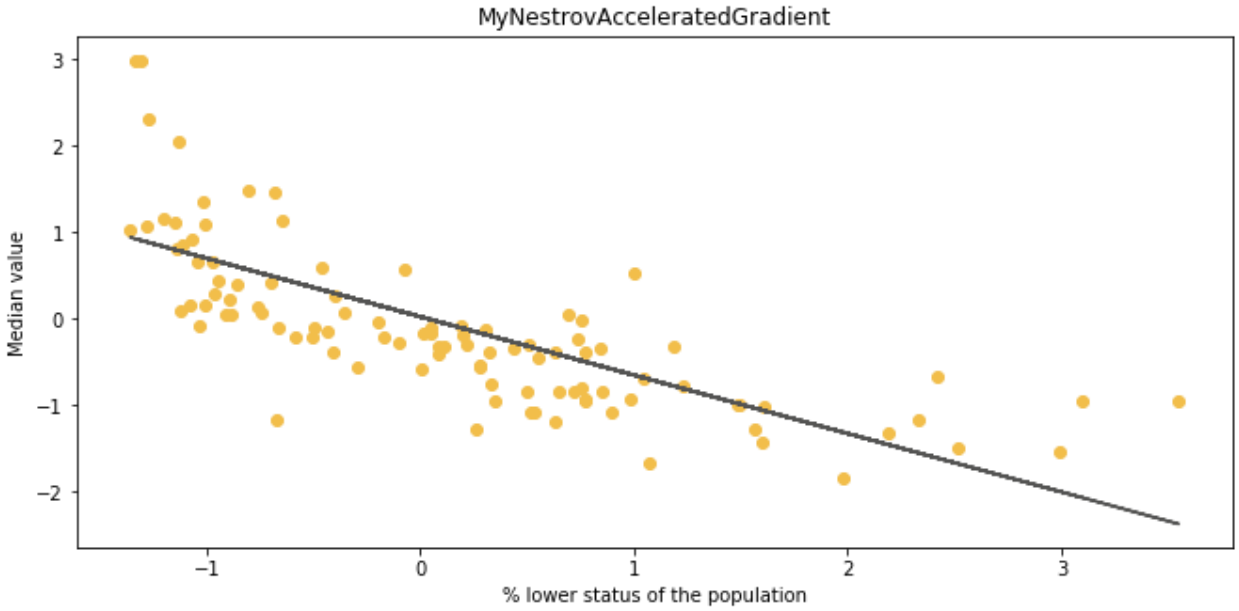
```
model = MyNestrovAcceleratedGradient(learning_rate = 0.0001)
history = model.fit(X_train, y_train, batch_size = 128, epochs = 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.0428307224941038)
Epoch: 100, Loss: 0.7708240506795851)
Epoch: 200, Loss: 0.9109097502153849)
Epoch: 300, Loss: 0.576534368060239)
Epoch: 400, Loss: 0.3422810954841916)
Epoch: 500, Loss: 0.556215628177696)
Epoch: 600, Loss: 0.35335516341041334)
Epoch: 700, Loss: 0.39640988193508453)
Epoch: 800, Loss: 0.5852076539641191)
Epoch: 900, Loss: 0.5284027213990843)
```

Notice that loss is a bit smaller. Here is what the history looks like:



Finally, we can plot out the model:



There is one more way we can improve optimization. Momentum-based optimizers focused on adding some value to the complete formula, but the other approach is to modify the learning rate, i.e. to control the scaling of the gradient. One such algorithm is **AdaGrad**.

4.3.6 AdaGrad

AdaGrad and similar algorithms focus on adaptively **scaling** the learning rate for each dimension. If we go back to our ball analogy (I would like to say for the last time but I can't promise that :)), this algorithm quickly **detects** where the global minimum is early on and pushes the ball in its direction. It is almost like putting a magnet in at the **global minimum**.

How does it work? Well, it is performed in two steps. First, we calculate the scale vector – *s*:

$$s \leftarrow s + \nabla MSE^2$$

this vector accumulates the squares of the partial derivative of the cost function with regards to the parameter. The vector *s* is then used to **scale** the learning rate:

$$\theta \leftarrow \theta - \alpha \frac{\nabla MSE}{\sqrt{s - \varepsilon}}$$

where ε is the so-called smoothing term hyperparameter, which is set to a small value like 10^-10. Essentially, this means that this algorithm adapts the learning rate to the parameters, performing **smaller** updates for parameters associated with **frequently** occurring features, and larger updates for parameters associated with infrequent features. This is why this optimizer is frequently used in combination with neural networks and sparse data.

4.3.6.1 Python Implementation

The implementation of this algorithm can be found in *MyAdaGrad* class:

```
class MyAdaGrad():
    def __init__(self, learning_rate, epsilon = 10 ** -10):
        self.learning_rate = learning_rate
        self.epsilon = epsilon

        self.w = 0
        self.b = 0

        self.scale_w = 0
        self.scale_b = 0

    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def _get_scale(self, X_batch, y_batch):
        f = y_batch - (self.w * X_batch + self.b)

        gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
        gradient_b = (-2 * f.sum() / len(X_batch))

        self.scale_w += np.multiply(gradient_w, gradient_w)
        self.scale_b += np.multiply(gradient_b, gradient_b)

    def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []
        momentum_vector = np.zeros_like(1)

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            self._get_scale(X_batch, y_batch)
```

```
            f = y_batch - (self.w * X_batch + self.b)

            divider_w = np.sqrt(self.scale_w + self.epsilon)
            divider_b = np.sqrt(self.scale_b + self.epsilon)

            gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
            gradient_b = (-2 * f.sum() / len(X_batch))

            self.w -= self.learning_rate * gradient_w / divider_w
            self.b -= self.learning_rate * gradient_b / divider_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

Let's break it down! In the constructor, we initialize all the necessary parameters (*w* and *b*); however, we initialize the *s* values for those parameters as well. Apart from that, note that the learning rate is not the only hyperparameter, but we have the ε value which we set to a small value close to zero.

```
def __init__(self, learning_rate, epsilon = 10 ** -10):
        self.learning_rate = learning_rate
        self.epsilon = epsilon

        self.w = 0
        self.b = 0

        self.scale_w = 0
        self.scale_b = 0
```

While the *_get_batch* method is the same as for previous algorithms, we have a new *_get_scale* method that calculates the *s* vector for each parameter. That is done by multiplying the gradient of the cost function with itself:

```
def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

def _get_scale(self, X_batch, y_batch):
        f = y_batch - (self.w * X_batch + self.b)

        gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
        gradient_b = (-2 * f.sum() / len(X_batch))

        self.scale_w += np.multiply(gradient_w, gradient_w)
        self.scale_b += np.multiply(gradient_b, gradient_b)
```

Finally, in the *fit* method, we first get the batch and calculate the *s* vector. Then we calculate "the divider", i.e. the factor we use to divide the learning rate. After that, the parameters *w* and *b* are updated:

```
def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []
        momentum_vector = np.zeros_like(1)

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            self._get_scale(X_batch, y_batch)

            f = y_batch - (self.w * X_batch + self.b)

            divider_w = np.sqrt(self.scale_w + self.epsilon)
            divider_b = np.sqrt(self.scale_b + self.epsilon)

            gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
            gradient_b = (-2 * f.sum() / len(X_batch))

            self.w -= self.learning_rate * gradient_w / divider_w
            self.b -= self.learning_rate * gradient_b / divider_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")
```

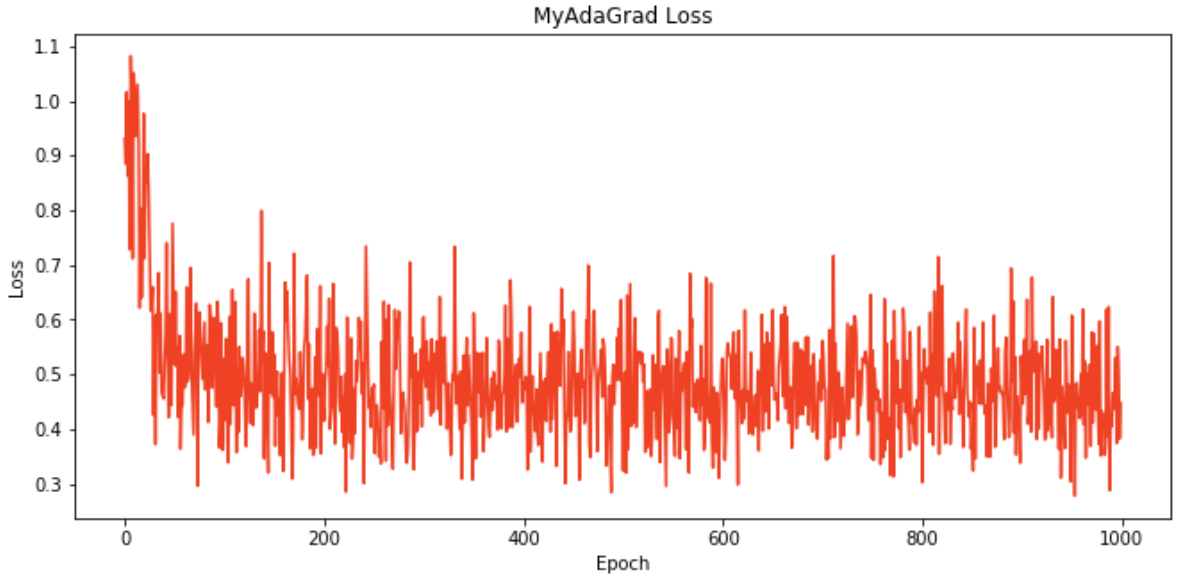
```
            history.append(loss)

        return history

def predict(self, X):
        return self.w * X + self.b
```

When we use this implementation with the data, here is what we get:

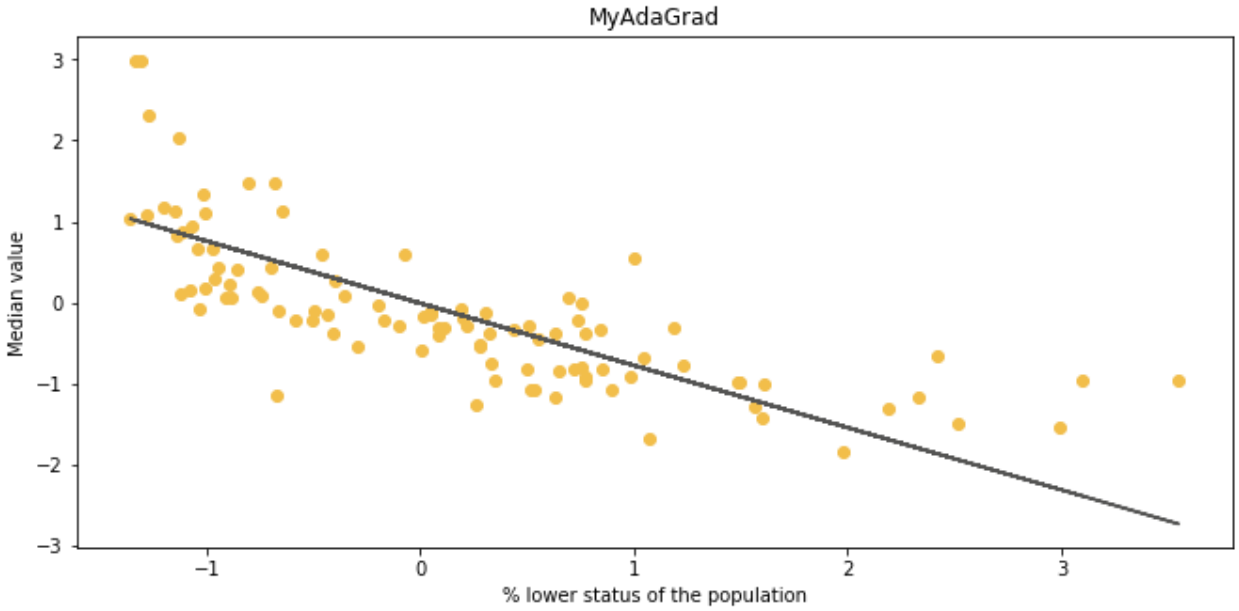
```
model = MyAdaGrad(learning_rate = 0.1)
history = model.fit(X_train, y_train, batch_size = 128, epochs = 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 0.7979041986313373)
Epoch: 100, Loss: 0.5538556882638777)
Epoch: 200, Loss: 0.47179562048530943)
Epoch: 300, Loss: 0.6090468657219893)
Epoch: 400, Loss: 0.29050695867780074)
Epoch: 500, Loss: 0.3170492333098174)
Epoch: 600, Loss: 0.5305147131291987)
Epoch: 700, Loss: 0.45605887968605396)
Epoch: 800, Loss: 0.4405627012256315)
Epoch: 900, Loss: 0.6149532862507887)
```

If we plot out the loss, we will notice that it goes down way faster than with momentum-based algorithms.



The model looks pretty much the same:



4.3.7 RMSProp

RMSProp builds on *AdaGrad* ideas. It is another adaptive learning rate method proposed by Geoff Hinton. Since Adagrad tends to be too aggressive, never converging to the global optimum, RMSProp proposes a solution for that. In essence, this algorithm restricts the accumulation of gradients by using a decay hyperparameter. So, instead of adding a complete square gradient to the s vector's every iteration, it does it like this:

$$s \leftarrow \beta s + (1 - \beta) \nabla MSE^2$$

where betta is the decay hyperparameter. Hinton proposed a value of 0.9 for β and 0.001 for the learning rate. The parameter update is done in the same way as for *AdaGrad*:

$$\theta \leftarrow \theta - \alpha \frac{\nabla MSE}{\sqrt{s - \varepsilon}}$$

4.3.7.1 Python Implementation

As you can imagine, the implementation of this algorithm is similar to *MyAdaGrad* class. Here is what code of *MyRMSProp* class looks like:

```
class MyRMSProp():
    def __init__(self, learning_rate, decay_rate = 0.9, epsilon = 10 ** -10):
        self.learning_rate = learning_rate
        self.epsilon = epsilon
        self.decay_rate = decay_rate

        self.w = 0
        self.b = 0

        self.scale_w = 0
        self.scale_b = 0

    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def _get_scale(self, X_batch, y_batch):
        f = y_batch - (self.w * X_batch + self.b)

        gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
        gradient_b = (-2 * f.sum() / len(X_batch))

        self.scale_w = self.decay_rate * self.scale_w + \
                   (1 - self.decay_rate) * np.multiply(gradient_w, gradient_w)
        self.scale_b = self.decay_rate * self.scale_b + \
                   (1 - self.decay_rate) * np.multiply(gradient_b, gradient_b)

    def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []
        momentum_vector = np.zeros_like(1)

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            self._get_scale(X_batch, y_batch)

            f = y_batch - (self.w * X_batch + self.b)
```

```
            divider_w = np.sqrt(self.scale_w + self.epsilon)
            divider_b = np.sqrt(self.scale_b + self.epsilon)

            gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
            gradient_b = (-2 * f.sum() / len(X_batch))

            self.w -= self.learning_rate * gradient_w / divider_w
            self.b -= self.learning_rate * gradient_b / divider_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

The difference we can notice here is the **decay** hyperparameter. It is initialized through the constructor and used for the scale vector calculation. For *AdaGrad* we calculated the scale vector like this:

```
self.scale_w += np.multiply(gradient_w, gradient_w)
self.scale_b += np.multiply(gradient_b, gradient_b)
```

In this algorithm we do so like this:

```
self.scale_w = self.decay_rate * self.scale_w + \
             (1 - self.decay_rate) * np.multiply(gradient_w, gradient_w)
self.scale_b = self.decay_rate * self.scale_b + \
             (1 - self.decay_rate) * np.multiply(gradient_b, gradient_b)
```

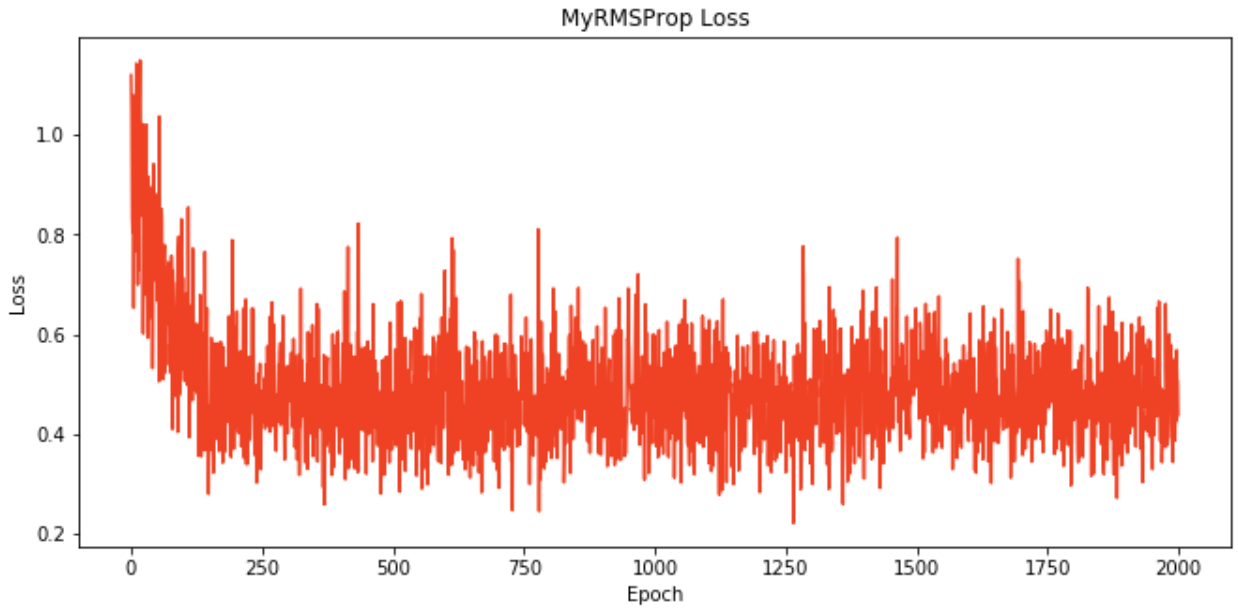
Here is what we get when we use *RMSProp* with the Boston Housing data:

```
model = MyRMSProp(learning_rate = 0.01)
history = model.fit(X_train, y_train, batch_size = 128, epochs = 2000)
```

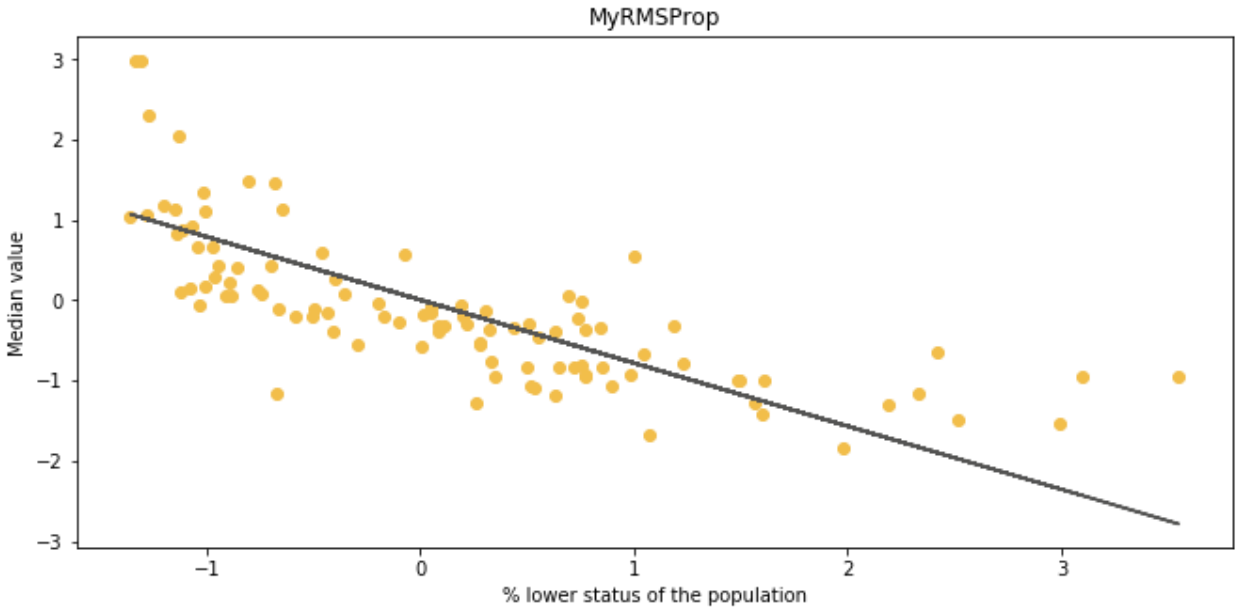
```
predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.0067302568004783)
Epoch: 100, Loss: 0.5548727653729529)
Epoch: 200, Loss: 0.4650584261906152)
Epoch: 300, Loss: 0.5603081231881148)
Epoch: 400, Loss: 0.6147792885402231)
Epoch: 500, Loss: 0.5129763511682113)
Epoch: 600, Loss: 0.591699424069597)
Epoch: 700, Loss: 0.5413949069489516)
Epoch: 800, Loss: 0.5215547915706193)
Epoch: 900, Loss: 0.48122559852723423)
Epoch: 1000, Loss: 0.497310753056444)
Epoch: 1100, Loss: 0.38403636517387396)
Epoch: 1200, Loss: 0.3829499207052695)
Epoch: 1300, Loss: 0.4287416366915751)
Epoch: 1400, Loss: 0.48688055753943527)
Epoch: 1500, Loss: 0.41232863743069914)
Epoch: 1600, Loss: 0.3436122983294806)
Epoch: 1700, Loss: 0.4730678956784552)
Epoch: 1800, Loss: 0.4586441383688593)
Epoch: 1900, Loss: 0.4893198589718893)
```

The loss definitely goes down more slowly, but it converges. Take a look at the loss history:



Here is what we get when we plot the model:



4.3.8 Adam

It is pretty standard that several techniques are combined in the world of data science. That is how the **Adam** optimizer was born – by combining good things from momentum-based optimizers, mixed with good things from adaptive learning rate optimizers. By the way, Adam stands for adaptive moment estimation, but the algorithm is definitely more intuitive than its name. It combines ideas from momentum-based optimization and the *RMSProp*, meaning it keeps **both** exponentially decaying averages of past gradients and squared gradients. Here is how this algorithm is defined:

$$m \leftarrow \beta_m m + (1 - \beta_m) MSE$$

$$m \leftarrow \frac{m}{1 - \beta_m^t}$$

$$s \leftarrow \beta_s s + (1 - \beta_s) MSE^2$$

$$s \leftarrow \frac{s}{1 - \beta_s^t}$$

$$\theta \leftarrow \theta - \alpha \frac{MSE}{\sqrt{(s + \varepsilon)}}$$

Where *βm* is **momentum decay**, *βs* is **scale decay** and *t* is the **epoch** (starting from 1). The *βm* hyperparameter is usually set to 0.9 and *βs* to a value close to one, like 0.95 or 9.99. The second and fourth steps of this process might be a bit confusing, so let's reflect on them for a bit. Because momentum and scale vectors are **initialized** to 0 at the beginning of this algorithm, they can be **biased** to 0 at the beginning of training. These steps are making sure that doesn't happen. At the moment, this is one of the most popular optimizers. Let's implement it.

4.3.8.1 Python Implementation

The *MyAdam* class contains the implementation of the *Adam* algorithm. It is a combination of *MyMomentumOptimizer* class and *MyRMSProp* class with a twist. Take a look:

```
class MyAdam():
    def __init__(self, learning_rate, momentum_decay = 0.9, scaling_decay =
0.95, epsilon = 10 ** -8):
        self.learning_rate = learning_rate
        self.epsilon = epsilon
        self.momentum_decay = momentum_decay
        self.scaling_decay = scaling_decay
        self.w = 0
        self.b = 0
        self.scale_w = 0
        self.scale_b = 0
        self.momentum_vector_w = 0
        self.momentum_vector_b = 0
```

```
    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def _get_gradients(self, X_batch, y_batch):
        f = y_batch - (self.w * X_batch + self.b)

        gradient_w = (-2 * X_batch.dot(f.T).sum() / len(X_batch))
        gradient_b = (-2 * f.sum() / len(X_batch))

        return gradient_w, gradient_b

    def _get_momentum_vector(self, X_batch, y_batch, gradient_w, gradient_b,
                             epoch):
        self.momentum_vector_w = self.momentum_decay * self.momentum_vector_w +
                                 (1 - self.momentum_decay) * gradient_w
        self.momentum_vector_b = self.momentum_decay * self.momentum_vector_b +
                                 (1 - self.momentum_decay) * gradient_b

        self.momentum_vector_w = self.momentum_vector_w / (1 -
                                 self.momentum_decay**epoch)
        self.momentum_vector_b = self.momentum_vector_b / (1 -
                                 self.momentum_decay**epoch)

    def _get_scale(self, X_batch, y_batch, gradient_w, gradient_b, epoch):
        self.scale_w = self.scaling_decay * self.scale_w + (1 -
                       self.scaling_decay) * np.multiply(gradient_w, gradient_w)
        self.scale_b = self.scaling_decay * self.scale_b + (1 -
                       self.scaling_decay) * np.multiply(gradient_b, gradient_b)

        self.scale_w = self.scale_w / (1 - self.scaling_decay**epoch)
        self.scale_b = self.scale_b / (1 - self.scaling_decay**epoch)

    def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []
        momentum_vector = np.zeros_like(1)

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            gradient_w, gradient_b = self._get_gradients(X_batch, y_batch)

            self._get_scale(X_batch, y_batch, gradient_w, gradient_b, e + 1)
            self._get_momentum_vector(X_batch, y_batch,gradient_w, gradient_b,
                                      e + 1)
```

```
            divider_w = np.sqrt(self.scale_w + self.epsilon)
            divider_b = np.sqrt(self.scale_b + self.epsilon)

            self.w -= self.learning_rate * self.momentum_vector_w * gradient_w
                      / divider_w
            self.b -= self.learning_rate * self.momentum_vector_b * gradient_b
                      / divider_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

There are some new moments there. The constructor is huge now because it has to handle **additional** hyperparameters and initialize both scale and momentum vectors. We also extracted the *_get_gradients* method, so we can quickly get gradients for both parameters. The *_get_scale* and *_get_velocity* methods are similar to the previous implementations, but they are using *βm* and *βs* **hyperparameters** according to the *Adam* algorithm definition. The fit method follows these steps:

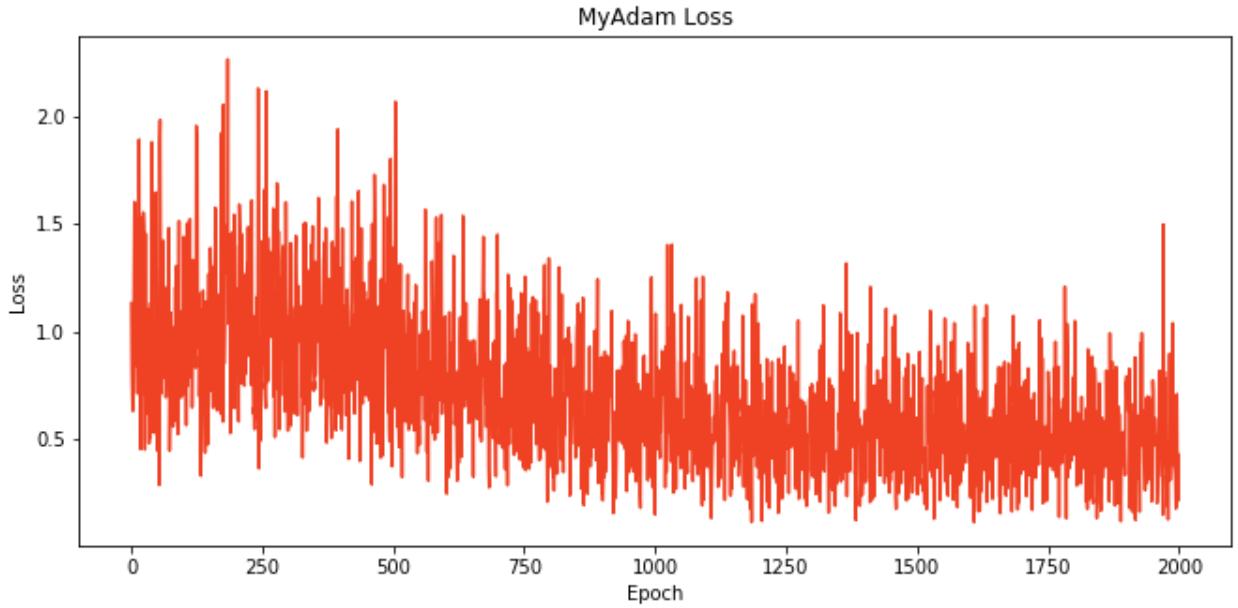
- Get batch
- Calculate gradients
- Calculate momentum vector
- Calculate scale vector
- Update parameters

When we apply it to our data:

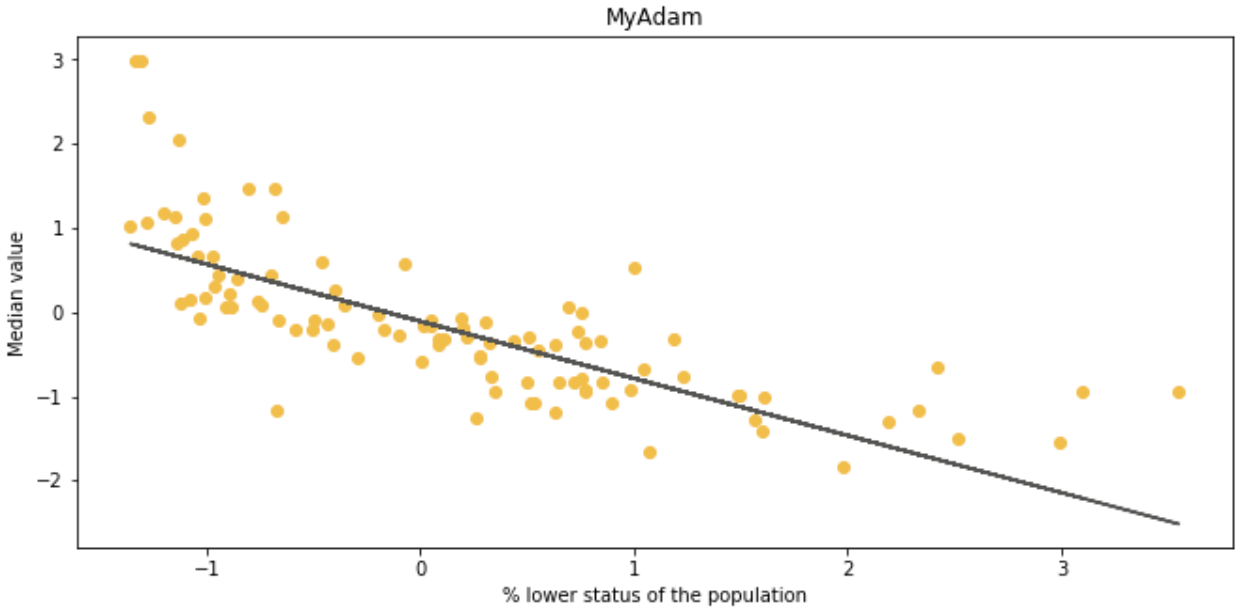
```
model = MyAdam(learning_rate = 0.001)
history = model.fit(X_train, y_train, batch_size = 32, epochs = 2000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.6016804863141227)
Epoch: 100, Loss: 1.0708601616598528)
Epoch: 200, Loss: 0.8396258429842761)
Epoch: 300, Loss: 0.9498729862073851)
Epoch: 400, Loss: 1.422709838725747)
Epoch: 500, Loss: 0.951641965926126)
Epoch: 600, Loss: 0.5114273687678832)
Epoch: 700, Loss: 0.9474342613915748)
Epoch: 800, Loss: 0.18370682522157994)
Epoch: 900, Loss: 0.551880807891246)
Epoch: 1000, Loss: 0.8319923072193909)
Epoch: 1100, Loss: 0.47982450339753563)
Epoch: 1200, Loss: 0.6607618450191821)
Epoch: 1300, Loss: 0.5550136613614162)
Epoch: 1400, Loss: 0.3261014705384321)
Epoch: 1500, Loss: 0.8596159351953009)
Epoch: 1600, Loss: 0.4764163861345504)
Epoch: 1700, Loss: 0.4233869971005261)
Epoch: 1800, Loss: 0.8254322188333079)
Epoch: 1900, Loss: 0.5054093932889874)
```

Here is what history of loss looks like:



The model still looks good as well:



4.4 Hyperparameter tuning

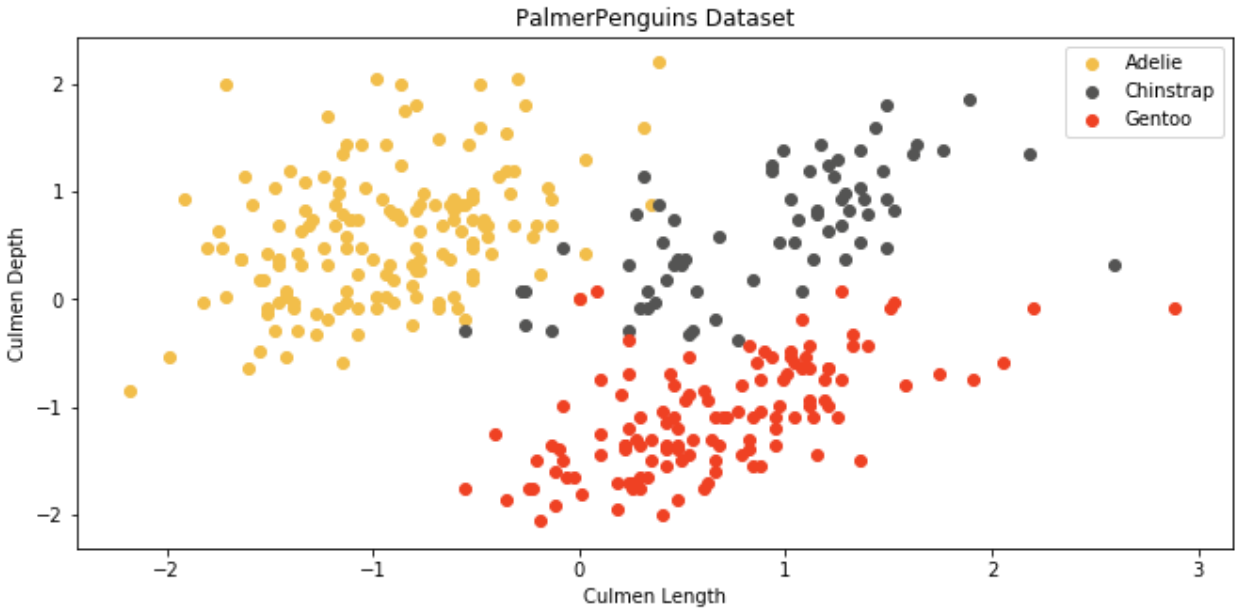
Hyperparameters are an integral part of every machine learning and deep learning algorithm. Unlike standard machine learning parameters that are learned by the algorithm itself (like w and b in linear regression, or connection weights in a neural network), hyperparameters are set by the engineer before the training process. They are an external factor that controls the behavior of the learning algorithm fully defined by the engineer. Do you need some examples?

The learning rate is one of the most famous hyperparameters; C in SVM is also a hyperparameter, maximal depth of the Decision Tree is a hyperparameter, etc. These can be set manually by the engineer. However, if we want to run multiple tests, this can be tiresome. That is where we use hyperparameter optimization. The main goal of these techniques is to find the hyperparameters of a given machine learning algorithm that deliver the best performance as measured on a validation set. In this tutorial, we explore several techniques that can give you the best hyperparameters.

4.4.1 Preparing the Data

Let's load and prepare the PalmerPenguins dataset. First, we load the dataset and remove features that we don't use in this chapter:

```
data = pd.read_csv('./data/penguins_size.csv')

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'],
axis=1)
Then we separate input data and scale it:

X = data.drop(['species'], axis=1)

ss = StandardScaler()
X = ss.fit_transform(X)

y = data['species']
species = {'Adelie': 0, 'Chinstrap': 1, 'Gentoo': 2}
y = [species[item] for item in y]
y = np.array(y)
Finally, we split data into training and testing datasets:

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```



4.4.2 Grid Search

Manual hyperparameter tuning is slow and tiresome. That is why we explore the first and simplest hyperparameters optimization technique – Grid Search. This technique is speeding up that process and it is one of the most used hyperparameter optimization

techniques. In its essence, it automates the trial and error process. For this technique, we provide a list of all hyperparameter values and this algorithm builds a model for each possible combination, evaluates it, and selects values that provide the best results. It is a universal technique that can be applied to any model.

In our example, we use the SVM algorithm for classification. There are three hyperparameters that we take into consideration – C, gamma and kernel. To understand them in more detail, check out this chapter. For C, we want to check the following values: 0.1, 1, 100, 1000; for gamma we use values: 0.0001, 0.001, 0.005, 0.1, 1, 3, 5, and for the kernel we use values: 'linear' and 'rbf'. Here is what that looks like in the code:

```
hyperparameters = {
    'C': [0.1, 1, 100, 1000],
    'gamma': [0.0001, 0.001, 0.005, 0.1, 1, 3, 5],
    'kernel': ('linear', 'rbf')
}
```

We utilize Sci-Kit Learn and its SVC class which contains the implementation of SVM for classification. Apart from that, we use GridSearchCV class, which is used for grid search optimization. Combined, that looks like this:

```
grid = GridSearchCV(
        estimator=SVC(),
        param_grid=hyperparameters,
        cv=5,
      scoring='f1_micro',
      n_jobs=-1)
```

This class receives several parameters through the constructor:

- *estimator* – the instance machine learning algorithm itself. We pass the new instance of the SVC class there;
- *param_grid* – contains a hyperparameter dictionary;
- *cv* – Determines the cross-validation splitting strategy;
- *scoring* – The validation metrics used to evaluate the predictions. We use F1 score;
- *n_jobs* – Represents the number of jobs to run in parallel. Value -1 means that it

is using all processors.

The only thing left to do is run the training process by utilizing the fit method:

```
grid.fit(X_train, y_train)
```

Once the training is complete, we can check the best hyperparameters and the score of those parameters:

```
print(f'Best parameters: {grid.best_params_}')
print(f'Best score: {grid.best_score_}')
```

```
Best parameters: {'C': 1000, 'gamma': 0.1, 'kernel': 'rbf'}
Best score: 0.9626834381551361
```

Also, we can print out all the results:

```
print(f'All results: {grid.cv_results_}')
```

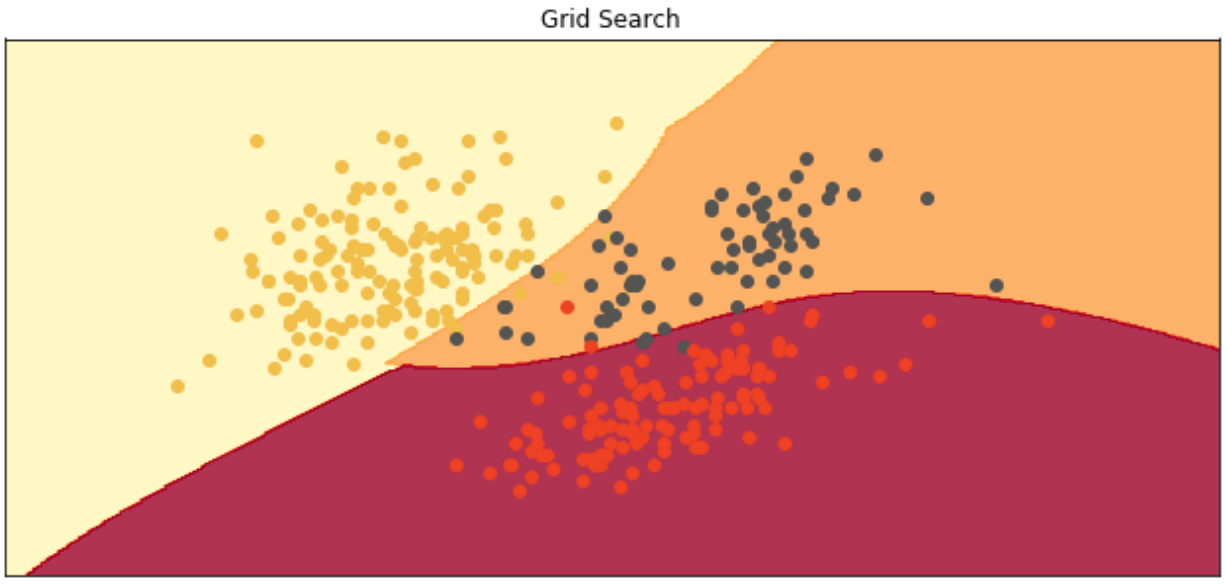
```
All results: {'mean_fit_time': array([0.00780015, 0.00280147, 0.00120015,
0.00219998,
       0.00739942, 0.00059962, 0.00600033, 0.0009994 , 0.00279789,
       0.00099969, 0.00340114, 0.00059986, 0.00299864, 0.000597  ,
       0.00340023, 0.00119658, 0.00280094, 0.00060058, 0.00179944,
       0.00099964, 0.00079966, 0.00099916, 0.00100031, 0.00079999,
       0.002     , 0.00080023, 0.00220037, 0.00119958, 0.00160012,
       0.02939963, 0.00099955, 0.00119963, 0.00139995, 0.00100069,
       0.00100017, 0.00140052, 0.00119977, 0.00099974, 0.00180006,
       0.00100312, 0.00199976, 0.00220003, 0.00320096, 0.00240035,
       0.001999  , 0.00319982, 0.00199995, 0.00299931, 0.00199928,
...
```

Ok, let's now build this model and check how well it performs on the test dataset:

```
model = SVC(C=500, gamma = 0.1, kernel = 'rbf')
model.fit(X_train, y_train)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98        23
           1       0.92      0.92      0.92        13
           2       0.97      1.00      0.98        31

    accuracy                           0.97        67
   macro avg       0.96      0.96      0.96        67
weighted avg       0.97      0.97      0.97        67
```

Cool, our model with the proposed hyperparameters got the accuracy ~97%. Here is what the model looks like when plotted:



4.4.3 Random Search

The Grid search is super simple. However, it is also expensive,especially in the area of deep learning, where training can take a lot of time. Also, it can happen that some of the hyperparameters are more important than others. That is why the idea of Random

Search was born. In fact, this study shows that random search is more efficient than the grid search for hyperparameter optimization in terms of computing costs. This technique allows the more precise discovery of good values for the important hyperparameters too.

Just like Grid Search, the Random Search creates a grid of hyperparameter values and selects random combinations to train the model. It's possible for this approach to miss the most optimal combinations; however, it surprisingly picks the best result more often than not, and in a fraction of the time compared to Grid Search. Let's see how that works in the code. Again we utilize Sci-Kit Learn's SVC class, but this time we use RandomSearchCV class for random search optimization.

```
hyperparameters = {
    "C": stats.uniform(500, 1500),
    "gamma": stats.uniform(0, 1),
    'kernel': ('linear', 'rbf')
}

random = RandomizedSearchCV(
                estimator = SVC(),
                param_distributions = hyperparameters,
                n_iter = 100,
                cv = 3,
                random_state=42,
                n_jobs = -1)

random.fit(X_train, y_train)
```

Note that we used uniform distribution for C and gamma. Again, we can print out the results:

```
print(f'Best parameters: {random.best_params_}')
print(f'Best score: {random.best_score_}')
```

```
Best parameters: {'C': 510.5994578295761, 'gamma': 0.0230624250414157 57,
'kernel': 'linear'}
Best score: 0.9700374531835205
```

Note that we got close but different results than when we used Grid Search. The value of the hyperparameter C was 500 with Grid Search, while with Random Search, we got 510.59. From this alone, you can see the benefit of Random Search since it is unlikely

that we would put this value in the grid search list. Similarly, for the gamma, we got 0.23 for Random Search against 0.1 for Grid Search. What is really surprising is that Random Search picked the linear kernel and not RBF, and that it got a higher F1 Score with it. To print all results, we use the cv_results_ attribute:

```
print(f'All results: {random.cv_results_}')
```

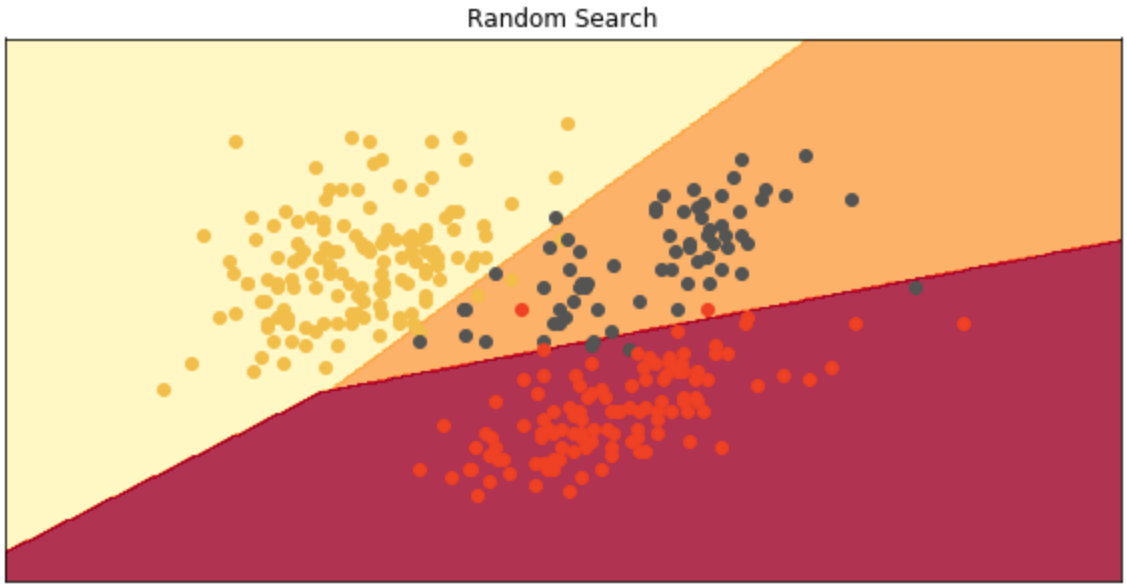
```
All results: {'mean_fit_time': array([0.00200065, 0.00233404, 0.00100454,
0.00233777,
       0.00033339, 0.00099715, 0.00132942, 0.00099921, 0.00066725,
       0.00266568, 0.00233348, 0.00233301, 0.0006667 , 0.00233285,
       0.00100001, 0.00099993, 0.00033331, 0.00166742, 0.00233364,
       0.00199914, 0.00433286, 0.00399915, 0.00200049, 0.01033338,
       0.00100342, 0.0029997 , 0.00166655, 0.00166726, 0.00133403,
       0.00233293, 0.00133729, 0.00100009, 0.00066662, 0.00066646,

        ....
```

Let's do the same thing as we did for Grid Search: create the model with proposed hyperparameters, check the score on the test dataset and plot out the model.

```
model = SVC(C=510.5994578295761, gamma = 0.023062425041415757, kernel =
'linear')
model.fit(X_train, y_train)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98        23
           1       0.92      0.92      0.92        13
           2       0.97      1.00      0.98        31

    accuracy                           0.97        67
   macro avg       0.96      0.96      0.96        67
weighted avg       0.97      0.97      0.97        67
```

Wow, the F1 score on the test dataset is exactly the same as when we used the Grid Search. Check out the model:



4.4.4 Bayesian Optimization

A really cool fact about the previous two algorithms is that all the experiments with various hyperparameter values can be run in parallel. This can save us a lot of time. However, this is also their biggest drawback. Since every experiment is run in isolation, we can not use the information from past experiments in the current one. There is a whole field that is dedicated to the problem of sequenced optimization – sequential model-based optimization (SMBO). Algorithms that are explored in this field use previous experiments and observations of the loss function. Based on them, they try to determine the next optimal point. One of such algorithms is Bayesian Optimisation.

Just like other algorithms, the ones from the SMBO group use previously evaluated points (in this case those are hyperparameter values, but we can generalize) to compute the posterior expectation of what the loss function looks like. This algorithm uses two important math concepts – The Gaussian process and the acquisition function. Since Gaussian distribution is done over random variables, the Gaussian process is its generalization over functions. Just like Gaussian distribution has mean value and covariance, the Gaussian process is described by the mean function and the covariance function.

The acquisition function is the function we use to evaluate the current loss value. One way to observe it is as a loss function for loss function. It is a function of the posterior

distribution over loss function that describes the utility for all values of the hyperparameters. The most popular acquisition function is expected improvement:

$$EI(x) = E[max0, f(x) - f(x')]$$

where *f* is a loss function and *x'* is the current optimal set of hyperparameters. When we put it all together, Byesian optimization is done in 3 steps:

- Using the previously evaluated points for loss function, the posterior expectation is calculated using Gaussian Process.
- The new set of points that maximizes expected improvement is chosen.
- The loss function of the new selected points is calculated.

The easiest way to bring this to code is by using the Sci-Kit optimization library, often called skopt. Following the process that we used on previous examples, we can do the following:

```
hyperparameters = {
    "C": Real(1e-6, 1e+6, prior='log-uniform'),
    "gamma": Real(1e-6, 1e+1, prior='log-uniform'),
    "kernel": Categorical(['linear', 'rbf']),
}

bayesian = BayesSearchCV(
                estimator = SVC(),
                search_spaces = hyperparameters,
                n_iter = 100,
                cv = 5,
                random_state=42,
                n_jobs = -1)

bayesian.fit(X_train, y_train)
```

Again, we defined a dictionary for a set of hyperparameters. Note that we used Real and Categorical classes from the Sci-Kit Optimisation library. Then we utilize the BayesSearchCV class in the same way we use GridSearchCV or RandomSearchCV. After the training is done, we can print out the best results:

```
print(f'Best parameters: {bayesian.best_params_}')
print(f'Best score: {bayesian.best_score_}')
```

```
Best parameters:
OrderedDict([('C', 3932.2516133086), ('gamma', 0.0011646737978730447),
('kernel', 'rbf')])
Best score: 0.9625468164794008
```

Its:It's interesting, isn't it? We got quite different results using this optimization. Loss is a bit higher than when we used Random Search. We can even print out all resu

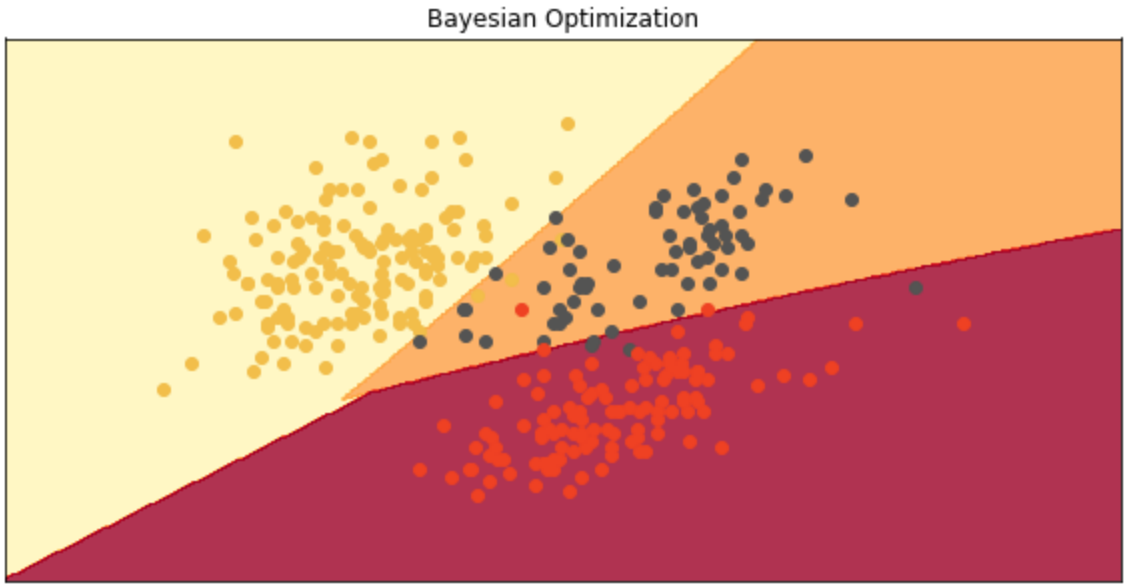
```
print(f'All results: {bayesian.cv_results_}')
```

```
All results: defaultdict(<class 'list'>, {'split0_test_score':
[0.9629629629629629,
  0.9444444444444444, 0.9444444444444444, 0.9444444444444444,
0.9444444444444444,
  0.9444444444444444, 0.9444444444444444, 0.9444444444444444,
0.46296296296296297,
  0.9444444444444444, 0.8703703703703703, 0.9444444444444444,
0.9444444444444444,
  0.9444444444444444, 0.9444444444444444, 0.9444444444444444,
0.9444444444444444,
  .....
```

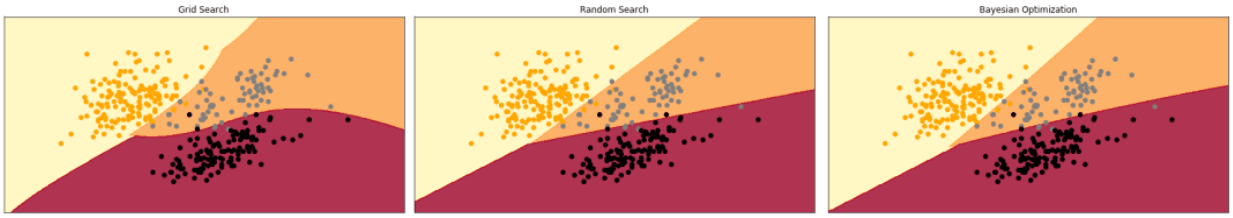
How does the model with these hyperparameters perform on the test dataset? Let's find out:

```
model = SVC(C=3932.2516133086, gamma = 0.0011646737978730447, kernel =
'rbf')
model.fit(X_train, y_train)

predictions = model.predict(X_test)
print(metrics.classification_report(y_test, predictions))
```

This is super interesting. We got better scores on the test dataset even though we got worse results on the validation dataset. Here is the model:



Just for fun, let's put all these models side by side:



4.4.5 Alternatives

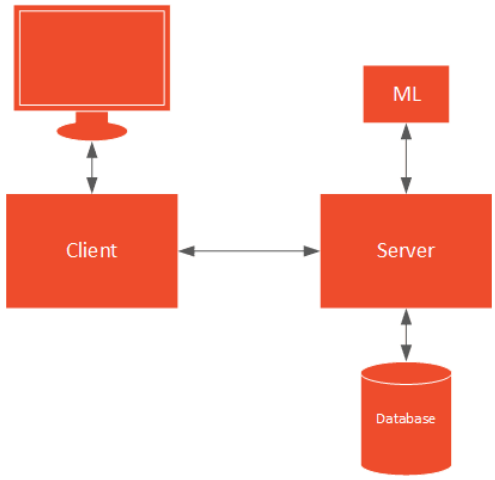
In general, the previously described methods are the most popular and the most frequently used. However, there are several alternatives that you can consider if the previous ones are not working out for you. One of them is Gradient-Based optimization of hyperparameter values. This technique calculates the gradient with respect to hyperparameters and then optimizes them using the gradient descent algorithm. The problem with this approach is that for the gradient descent to work well, we need a function that is convex and smooth, which is often not the case when we talk about hyperparameters. The other approach is the use of Evolutionary algorithms for optimization.

5. Deploying Machine Learning Model

In this chapter, we explore how we can prepare a machine learning model for production and **deploy** it inside of a simple Web application. Deployment of Machine Learning models is an **art** in itself. In fact, to successfully put a machine learning model into production goes **beyond** data science knowledge and engages a lot of software development and DevOps skills. Why should you care about all this? Well, at the moment, Machine Learning engineers have the most valued roles in data science teams. This role gathers the best of both worlds.

These engineers don't only have to know how to apply different Machine Learning and **Deep Learning** models to a proper problem, but how to test them, verify them and finally deploy them as well. Having a person that is able to put Machine learning models into production became a huge **asset** to any company. In order to become a successful Machine Learning engineer, you need to have a variety of skills that are not focused only on the data. In fact, the amount of code that is written for Machine Learning models is much smaller than the amount of the code that supports testing and serving that model.

MLOps, an engineering term that was built around this need, is becoming a new trend. That is why in this chapter, we focus on several techniques you need to know in order to **build** one successful machine learning application that users would like. The complete solutions provided in this tutorial are composed of two components: the server-side and the client-side. This is the basic web architecture, where the client-side interacts with the user of the application and sends it to the server-side. The server-side does the processing of the data, or in this case, running the predictions, and returns the result to the client-side, which presents it to the user. In essence, in this chapter, we build a small system that can be represented with the following image:



5.1 Rest API

Before we start with the implementation of the Web application with Python and Flask, let's first find out what the REST API is. Now, I believe that you have seen this term once or twice in your life. The second part of the term – API, stands for an application programming interface. Essentially, the API represents the set of rules that programs use to communicate with each other. For example, the server-client architecture server side of the application is programmed in a way that exposes methods that can be called by the client-side of the application. What this means is that the client can call a method on the server inside of its code and get a certain result from it. REST stands for "Representational State Transfer". This represents a set of rules that developers should follow when they build their APIs. It defines what the API should look like, so APIs are standardized.

One of the rules defines that data or resources could be gathered when you link a specific URL. For example, you can link 'api.rubikscode.com/blogs' and get the list of blogs as a response. URL ' api.rubikscode.com/blogs ' is called a request, and the list of clients that you get back is called a response. Every request is composed of 4 parts:

- **The Endpoint** (route) – This is the URL we mentioned previously. It is structured like this – "root-endpoint/?". The root-endpoint is the starting point, and it can be followed by the path and query parameters. The path defines what specific resource is required. For example, the root-endpoint of GitHub's API is 'api.github.com', while the full endpoint to the list of my repositories on GitHub is https://api.github.com/users/nmzivkovic/repos.
- **The Method** – There are five types of requests that can be sent, and the method defines this type:
    - **GET** – Used to get or read information.
    - **POST** – Used to create a new resource.
    - **PUT** and **PATCH** – They are used to update resources.
    - **DELETE** – Deletes resource.
- **The Headers** – The headers are used to provide additional information to both client and server in the form of property-values pairs. The list of valid headers on MDN's HTTP Headers Reference.
- **The Body** – This section contains information that the client sends to the server. It is not used in GET requests.

What we want to create in this chapter is the Web server, which serves as a model for the Iris predictions. We want to build an API using which the client-side of the application can get predictions from the model.

5.2 Saving Model into a File

The systems that use machine learning are usually built using microservices architecture. From the machine learning point of view, this means that there is one service that performs the training of the model and another one that uses it. This way, we have a clear separation of concerns. The communication between these two microservices is done via some file in which the model is stored. Something like this:

This way, the "*training microservice*" is in charge of training the model, and versioning, testing and propagating the correct file to the *usage microservice*.

Here is how we can save a model into file when we use *Sci-Kit Learn*. Let's say that we want to train *Random Forest* for the *Palmer Penguins* dataset. We need to do something like this:

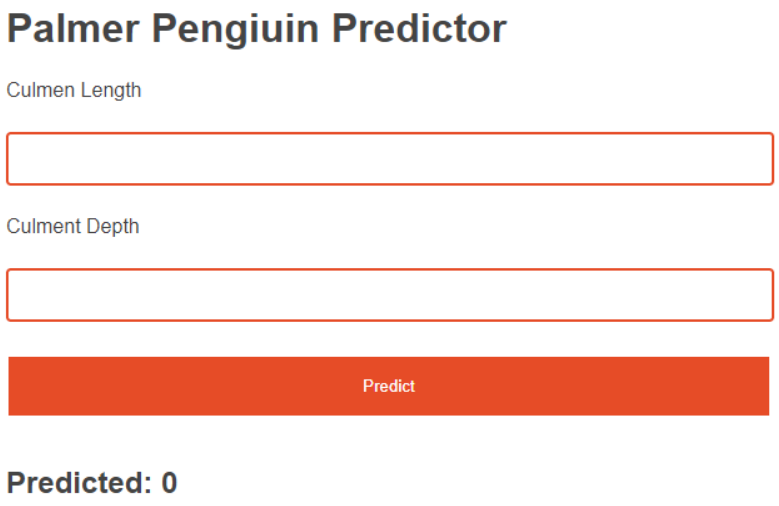
```
import pandas as pd
import numpy as np
from joblib import dump

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Load and preprocess data.
data = pd.read_csv('./data/penguins_size.csv')

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)
data = data[data['species'] != 'Chinstrap']

X = data.drop(['species'], axis=1).values

y = data['species']
species = {'Adelie': 1, 'Gentoo': 2}
y = [species[item] for item in y]
```

```
y = np.array(y)

X = np.delete(X, 182, axis=0)
y = np.delete(y, 182, axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)

# Train model.
model = RandomForestClassifier(n_estimators=11, max_leaf_nodes=16, n_jobs=-1)
model.fit(X_train, y_train)

# Save model into file.
dump(model, './static/model/classifier.joblib')
```

First, we load and prepare the data. Then we run the training using the *fit* method, and finally we save it using the *dump* function from *joblib* library. Note that the file needs to have a *.joblib* extension.

5.3 Deployment with Flask

In this section, we build a simple web application that provides the interface through which users can enter properties of the penguin. This application also uses the model that we saved in the previous chapter, runs predictions on newly entered data and displays the result to the user.



5.3.1 Flask Basics

As already mentioned, Flask is a python based web framework which has almost no dependencies to external libraries. This makes it very light. Since we are creating a simple Web application, this makes Flask the perfect choice for our implementation. The standard folder structure for the Flask folder looks like this:

```
>tree flask_app
Folder PATH listing for volume Data
Volume serial number is 42E0-3374
G:\DEEP_LEARNING\FLASK_APP
├───static
│   ├───css
│   └───model
└───templates
```

Files that are not changed and are assets are located in the *static* folder. This is the place for the *model* that we created so we can copy it over there. The *template* folder is reserved for the HTML files. The flask looks into this folder when it needs to serve dynamic files. We create *base.html* which looks like this:

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="{{url_for('static', filename='css/main.css')}}"
/>
  {% block head%}{% endblock %}

</head>

<body>
  {% block body %}{% endblock %}
</body>
</html>
```

This is just a basic HTML structure with some curly brackets. This way, we are able to quickly inherit this basic structure and implement custom pages. Notice that we added the main.css file in the *css* folder. Ok, let's install Flask using *pip*:

```
pip install flask
```

Once *Flask* is installed, its flask command-line script is installed as well. When you initiate the command:

```
flask run
```

it will start the Flask package to run the HTTP server using an app object. Where is the app object, one might ask? Let's create *app.py* file in the root folder and define this object:

```
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, world!'
```

As you can see, the app object is an instance of the Flask object. This is our HTTP server. Routes are defined using the *@app.route('endpoint')* decorator. In the example above, we define the only route on the *localhost:8080/*. When we run this using the *flask run* or using the *python app.py*, we get this in the browser:

Nice and easy, right? We got all the pieces, model and web server. All we have to do is put them together.

5.3.2 Putting it all together

In the beginning, we focus on UI, meaning we want to create an *HTML* page that handles this functionality. Thanks to the *base.html* we created previously, we can create a simple *index.html* page like this:

```
{% extends 'base.html' %}
{% block head%}
{% endblock %}
{% block body %}
<div>
    <h1>Palmer Penguin Predictor</h1>
    <form action="/predict" method="POST">
        <div>
            <p>Culmen Length</p>
            <input type="text" name = "culmen_length" id="culmen_length">
        </div>
```

```
        <div>
            <p>Culment Depth</p>
            <input type="text" name = "culmen_depth" id="culmen_depth">
        </div>
        <input type="submit" value="Predict">
    </form>
    <h2>Predicted: {{pred}}</h2>
</div>
{% endblock %}
```

This page extends *base.html*. As you can see, we added one form in the body of the *HTML* page. It is quite simple; it has four text inputs for the two features and the *Submit* button. Note that this button sends POST requests to the *'/predict'* endpoint with this information. In the end, it displays whatever is stored in the *pred* parameter.

Also, we extended the *main.css* file with some stylings and now it looks like this:

```
body {
    margin: 0;
    padding: 0;
    margin-left: 20px;
    font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
    color: #444;
}

input[type=text] {
    border: 2px solid #E64C27;
    border-radius: 4px;
    width: 30%;
    padding: 12px 20px;
    margin: 8px 0;
 }

input[type=button], input[type=submit], input[type=reset] {
    background-color: #E64C27;
    border: none;
    color: white;
    padding: 16px 32px;
    width: 32%;
    text-decoration: none;
    margin: 20px 2px;
    cursor: pointer;
}
```

Now, we focus on the *app.py* file and the *Flask app object*. Importing necessary libraries and creating the *app* object is the obvious first step:

```
from flask import Flask, request, jsonify, render_template
import numpy as np
from joblib import load

app = Flask(__name__)
spicies_map = {1: 'Adelie', 2: 'Gentoo'}
```

Then, we want to import the created model into the app object. We do that in the function *load_model_to_app()* which has the decorator *@app.before_first_request*. Basically, this method will be executed during the initialization of the application. In this method, we load the trained machine learning algorithm from the file:

```
@app.before_first_request
def load_model_to_app():
    app.predictor = load_model('./static/model/model.h5')
```

Then we define an API method that is called if the root endpoint is hit, i.e., this endpoint – '/'. For that we use a method *index* with the decorator *@app.route('/')*. This method returns a rendered template *index.html*. Here is what that looks like:

```
@app.route("/")
def index():
    return render_template('index.html', pred = 0)
```

Finally, we create a method *predict*() for the endpoint *'/predict'*. This method, of course, has a decorator that looks like this *@app.route('/predict', methods=['POST'])*. We define this endpoint with the POST method. In this method, we prepare the data received from the input and get predictions from the model. Here is what this method looks like:

```
@app.route('/predict', methods=['POST'])
def predict():
    """Provide main prediction API route. Responds to both GET and POST
requests."""
    data = [request.form['culmen_length'], request.form['culmen_depth']]

    data = np.array([np.asarray(data, dtype=float)])

    species = app.model.predict(data)

    print('INFO Predictions: {}'.format(spicies_map[species[0]]))

    return render_template('index.html', pred=spicies_map[species[0]])
```

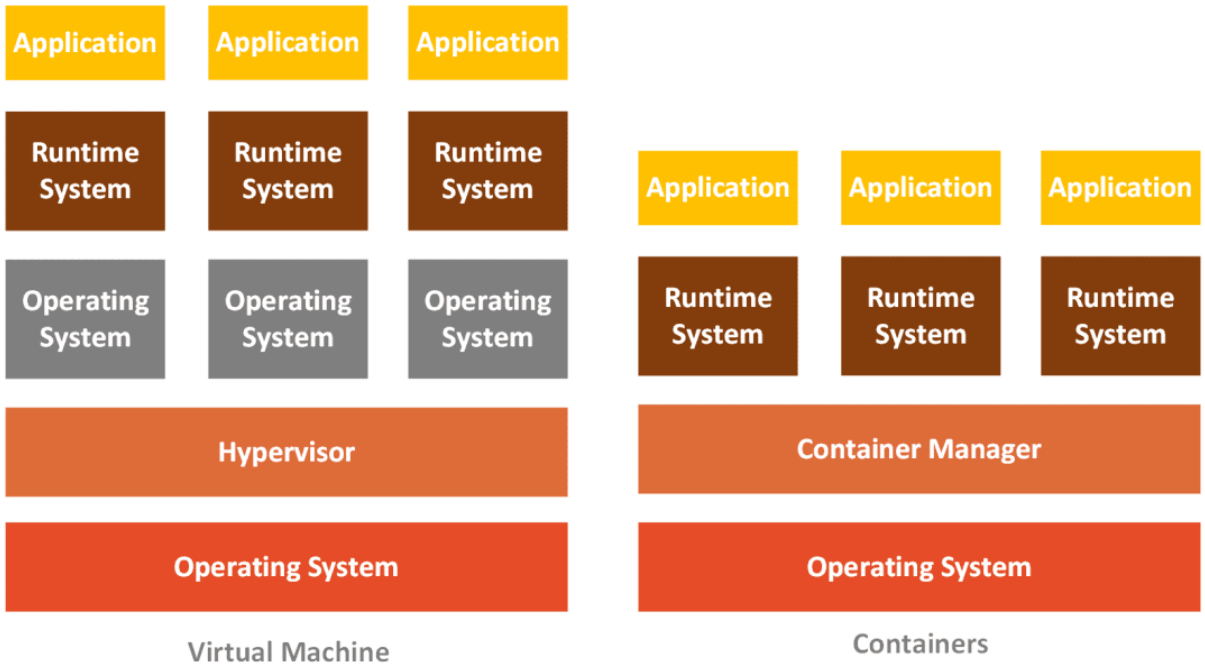
Here is the complete *app.py* file:

```
from flask import Flask, request, jsonify, render_template
import numpy as np
from joblib import load

app = Flask(__name__)
spicies_map = {1: 'Adelie', 2: 'Gentoo'}

@app.before_first_request
def load_model_to_app():
    """Load model before any other actions."""
    app.model = load('./static/model/classifier.joblib')

@app.route("/")
def index():
    """In the home route just load index."""
    return render_template('index.html', pred = 0)

@app.route('/predict', methods=['POST'])
def predict():
    """Provide main prediction API route. Responds to both GET and POST
requests."""
    data = [request.form['culmen_length'], request.form['culmen_depth']]

    data = np.array([np.asarray(data, dtype=float)])

    species = app.model.predict(data)

    print('INFO Predictions: {}'.format(spicies_map[species[0]]))

    return render_template('index.html', pred=spicies_map[species[0]])

def main():
    """Run the app."""
    app.run(host='0.0.0.0', port=8000, debug=False)  # nosec

if __name__ == '__main__':
    main()
```

5.4 Docker Basics

In this section, we focus on the **scalability** problem, and we explore model deployment with **Docker**. Docker and **containers** were born out of the need to better utilize servers. Basically, at one moment in history, servers became very powerful in terms of processing power. However, we never create processes there that **require** and utilize that power. Web Applications don't need that. That is when concepts of virtualization and virtual machines were born. These concepts gave servers the power to run more applications on different operating systems at the **same time**. Then, big companies like Amazon saw this as a business opportunity and provided cheap cloud solutions based on it. This is how the **cloud** was created. During this time, applications became bigger in terms of dependencies. It became hard to develop and maintain them because a developer should take care of **all** external libraries, frameworks, and operating systems. Docker and containers solved this problem by providing a means to run any application **regardless** of the operating system.

At this point, one might ask, what is a container? The container packages and unites the application code with all its dependencies. This way, it becomes a **unit** that can run on any computing environment. Docker is a tool that helps us **build and manage** containers. The difference between the container and the virtual machine is in the way that hardware is **utilized**. In general, virtual machines are managed by Hypervisors. However, Hypervisors are only available on processors that support the virtual replication of hardware. This essentially means that virtual machines run software on **real hardware** while providing **isolation** from it. On the other hand, containers require an operating system with basic services and use virtual-memory support for isolation. To sum up, a virtual machine provides an **abstract machine** (along with device drivers required for that abstract machine), while a container provides an **abstract operating system**.



To install Docker, follow instructions provided on this **page**. Docker comes with the UI, which we will not consider in this chapter. We utilize only docker **CLI**, which comes with this installation as well. You know, like real hackers.

There are three important Docker components that you should be aware of: Docker container **Image**, **Dockerfile** and Docker **Engine**. Docker container image is a lightweight file-system that includes everything that the application needs to run. It has the system tools and libraries, runtime and the application code. These images are **turned** into Docker containers once a user runs the *docker build* command, which will be explained in detail in a little while. Once this command is initiated, Docker uses *Dockerfile* to create a Docker container from Docker Image. This container is then run by the Docker engine. There are a lot of pre-cooked Docker Images available at **Docker Hub**. For example, if you have a need for a docker image for *Ubuntu*, you can find it at Docker Hub. Docker image is obtained by initiating command: *docker pullimage_name*. For the purpose of this chapter, we need three images, so let's download them right away:

```
docker pull ubuntu
docker pull tensorflow/tensorflow
docker pull tensorflow/serving
```

Note that every Docker image has multiple **tags**, which can be observed as a specific image **version** or variant.

The *Dockerfile* is the file that defines what the container…well, contains. In this file, a user defines which Docker image should be used, which dependencies should be installed and which application should be run and how. In essence, a Docker image consists of read-only **layers**. Each of these layers is represented with one Dockerfile **instruction**.

Usually, we start a *Dockerfile* with FROM instruction. With it, we define which image is used. Then we use different instructions to **describe** the system and the application we want to run. We may use COPY instruction to copy files from the local environment to the container, or we can use WORKDIR instruction to set the working directory. We can run various commands within the container with CMD instruction and run the application with the RUN command. A full **list of** Docker file instructions and how they can be used can be found **here**.

Let's observe one example of Dockerfile:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

In this file, the first instruction – FROM defines that we use ubuntu:18.04 Docker image. COPY instruction adds files to the container's *app* directory. RUN builds the application with *make* and CMD runs command '*python app/app.py*' which in turn runs the application.

Once the *Dockerfile* is ready, we can proceed with **building** and **running** the Docker container. We already mentioned some of the Docker commands that we can use, but let's get into more details. With Docker installation comes the rich *Docker CLI*. Using this CLI we can **build**, **run** and **stop** our container. In this chapter, we explore some of the most important commands. The first on the list is definitely the *docker build* command. When this command is issued, the current working directory becomes a so-called **build context**. All files from this directory are sent to the container. Docker assumes that the *Dockerfile* is located in this directory, but you can define different locations as well. Once this command finishes its job, the user can run the *docker run* command and run the container.

Docker CLI has other useful commands. For example, you can check the **list** of running containers with the command *docker ps*. Also, you can **stop** running a container with the command *docker container stop CONTAINER_ID*. If you need to run some commands within the *bash* of the container, you can use the *docker exec -it /bin/bash* to run the bash. Ok, this is all cool in theory, but let's do something practical. We want to run the Flask application we created in the previous section within the Docker container.

5.5 Deployment with Flask and Docker

First, we regroup our files a little bit into this kind of structure:

```
└──app
    ├──static
    │   ├──css
    │   └──model
    └──templates
```

Because there are several **requirements** we installed with *pipenv*, we need to make sure that these requirements are installed in the container as well. In order to make our lives a little bit easier, we **convert** requirements from pipenv lock file into *.txt* file. This is done with the command:

```
pipenv lock --requirements --keep-outdated > requirements.txt
```

Don't forget to run *pipenv synced -d* before it though. This will create *requirements.txt* which looks something like this:

```
-i https://pypi.org/simple
 absl-py==0.9.0
 astor==0.8.1
 cachetools==4.0.0
 certifi==2019.11.28
 chardet==3.0.4
 click==7.0
 flask==1.1.1
 gast==0.2.2
 google-auth-oauthlib==0.4.1
 google-auth==1.11.0
 google-pasta==0.1.8
 grpcio==1.26.0
 h5py==2.10.0
 idna==2.8
 itsdangerous==1.1.0
 jinja2==2.11.1
 keras-applications==1.0.8
 keras-preprocessing==1.1.0
 markdown==3.1.1
 markupsafe==1.1.1
 numpy==1.18.1
 oauthlib==3.1.0
 opt-einsum==3.1.0
 protobuf==3.11.3
 pyasn1-modules==0.2.8
 pyasn1==0.4.8
 requests-oauthlib==1.3.0
 requests==2.22.0
 rsa==4.0
 six==1.14.0
 tensorboard==2.0.2
 tensorflow-estimator==2.0.1
 tensorflow
 termcolor==1.1.0
 urllib3==1.25.8
 werkzeug==0.16.1
 wheel==0.34.2 ; python_version >= '3'
 wrapt==1.11.2
```

Now we can create the *Dockerfile* within the **root** folder. Here is what that file looks like:

```
#Start from the latest Long Term Support (LTS) Ubuntu version
FROM ubuntu:18.04

#Install pipenv
RUN apt-get update
RUN apt-get install -y build-essential python3.6 python3.6-dev python3-pip
python3.6-venv

#Create the working directory
RUN set -ex && mkdir /app
WORKDIR /app

#Copy only the relevant directories to the working directory
COPY app ./app

#Install Python dependencies
RUN set -ex && pip3 install -r app/requirements.txt

#Run the web server
EXPOSE 8000
ENV PYTHONPATH /app
CMD python3 ./app/app.py
```

In this file we defined these things:

- **Use** image ubuntu 18.04
- **Install** *pipenv* and Python 3.6
- Create a **working directory** and copy over necessary files
- **Install** everything from *requirements.txt*
- **Run** the web server

Now we can run the *docker build* command:

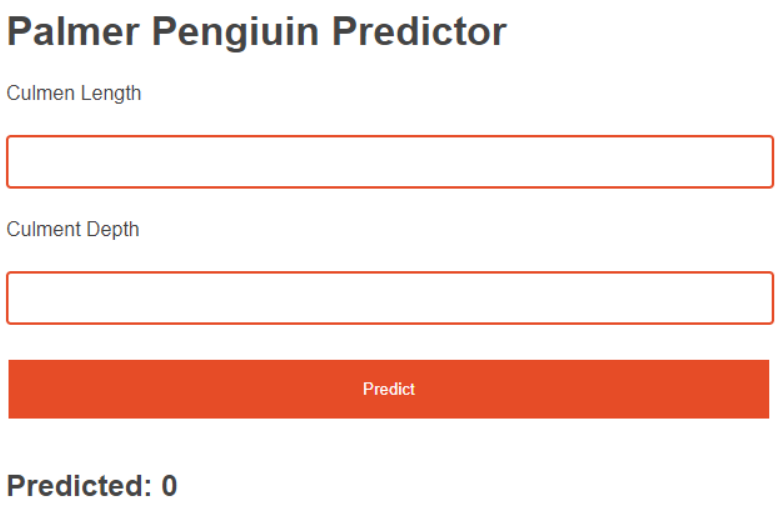
```
docker build -t dl_flask_rest_api  -f Dockerfile .
```

And after that we can run the *docker run* command:

```
docker run -p 8000:80 dl_flask_rest_api
```

```
Serving Flask app "app" (lazy loading)
Environment: production
WARNING: This is a development server. Do not use it in a production
deployment.
Use a production WSGI server instead.
Debug mode: off
Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
```

Finally, we can go to the **IP** we defined and see that our application is running just like in the previous section:



Once we enter new values for Culmen length and Culmen Depth and press the *Predict* button. we will get the predictions for that sample.

5.6 Deployment with Fast API

In this chapter, we use **FastAPI** to build REST API. Why do we use this library? There are several reasons for it. *FastAPI* is **faster** when compared with other major Python frameworks like *Flask* and *Django*. Also, this framework supports **asynchronous** code out of the box using the **async/await** *Python* keywords. This improves its performance even further. Probably the most distinct feature that makes *FastAPI* one of the best API libraries out there is the built-in interactive **documentation**. We explore this feature in more detail a little bit later. Finally, applications built using *FastAPI* are very easy to test and deploy. Because of all of this, *FastAPI* became a standard when it comes to building web API applications with *Python*.

To install *FastAPI* and all its dependencies, use the following command:

```
pip install fastapi[all]
```

This includes *Uvicorn*, an ASGI server that runs your code. If you are more comfortable with some other ASGI server like *Hypercorn*, that is also fine and you can use it for this tutorial.

For the web application, we use **Angular**. For that, you need to install **Node.js** and *npm*. The most common way to manipulate with Angular framework is to use the Angular Command Line Interface – **Angular-CLI**. One of the benefits of this tool is that once we initialize our application with it, we can use *TypeScript* and it will be automatically translated to *JavaScript*. Installing this interface is done through *npm*, of course, by running the command:

```
npm install -g angular-cli
```

When you want to create a new *Angular* application, you can do so with the command:

```
ng new application_name
```

This command will create a folder structure that will be used by our application. To run this application, position the shell in the just created root folder of your application (*cd application_name*), and call command:

```
ng serve
```

If you follow the previous steps, once you go to your browser and open localhost:4200, you will be able to see the default Angular application.

5.6.1 FastAPI Basics

Ok, let's build our first FastAPI application. The cool thing is that a simple *HelloWorld* example with *FastAPI* can be created with 5 lines of code. I am not joking. Are you ready? Here is what you need to do:

- Create a new **folder** and name it 'my_first_fastapi' (or whatever you feel like :))
- Create a new *Python* **script** called main.py (or whatever you feel like :))
- Add the following **code** to the main.py:

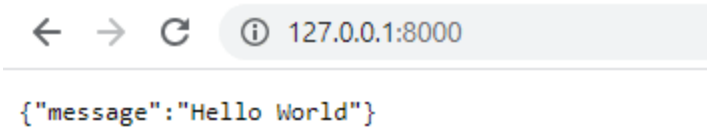
```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
  return {"message": "Hello World"}
```

- Go to the **terminal** and position into my_first_fastapi folder
- Run this **command**:

```
uvicorn main:app --reload
```

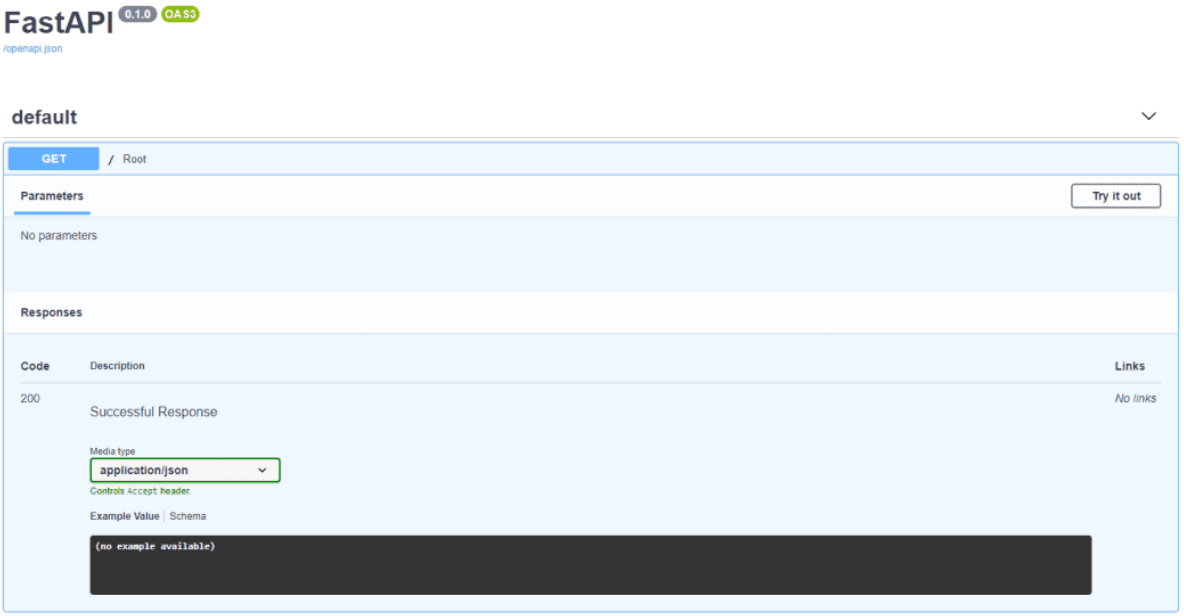
```
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [28720]
INFO: Started server process [28722]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Awesome, now you have a web server running in the *http://127.0.0.1:8000*. If you go to this address in your browser, you will see something like this:



Notice that *@app.get("route")* decorator in the code above? This decorator determines the type of request that can be issued on a particular endpoint. This means, for example, that if we put the *app.get("/users")*, we define REST API endpoint "users" on which we can issue GET http requests.

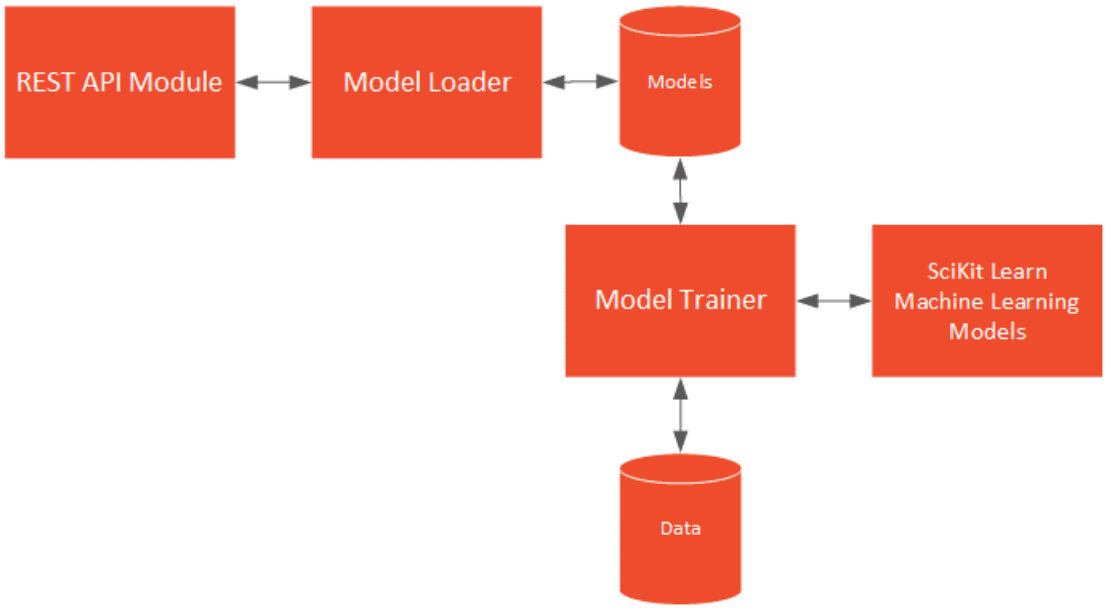
One of the coolest things about *FastAPI* is **built-in** interactive **documentation**. It is presented with *Swagger UI*. If we use the example that we have just built and go to the *http://127.0.0.1:8000/docs*, we will see the documentation page for the API:



You can click on any of the endpoints, explore further and learn about them. Probably the biggest benefit of this documentation is that you can perform live in-browser tests by clicking on the *Try it out* button. Ok, back to our application and let's see how we can utilize *FastAPI* for Machine Learning **Deployment**.

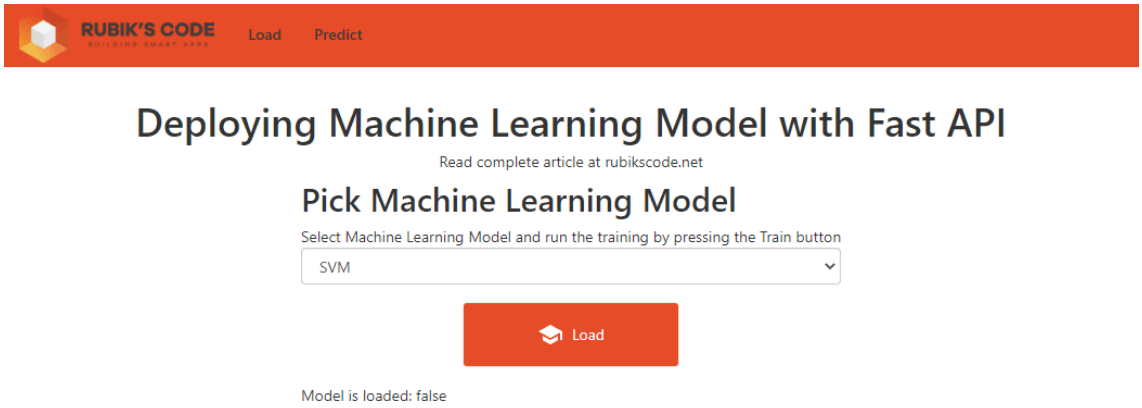
5.6.2 Putting it all together

Let's see what a **real-world** example looks like. In this solution, one component is in charge of gathering the data, the other is in charge of processing that data, and the third one is in charge of periodically training the models and **storing** them in some location. These stored models are then **utilized** by the REST API. Now, all of this is too much for this simple section, but still. The architecture of this solution looks like this:



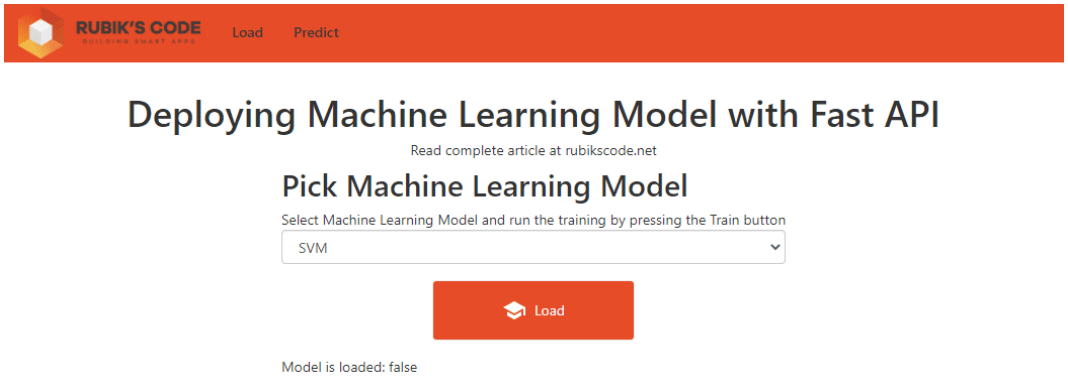
5.6.2.1 Client Side

Ok, in order for the user to communicate with our model, we need some sort of **user interface**. There are many frameworks and technologies available out there that you can do that with. In the previous chapter, we used **Flask and Python**. In this tutorial, we use **Angular** to build a web application that users can interact with. Here is what it looks like:

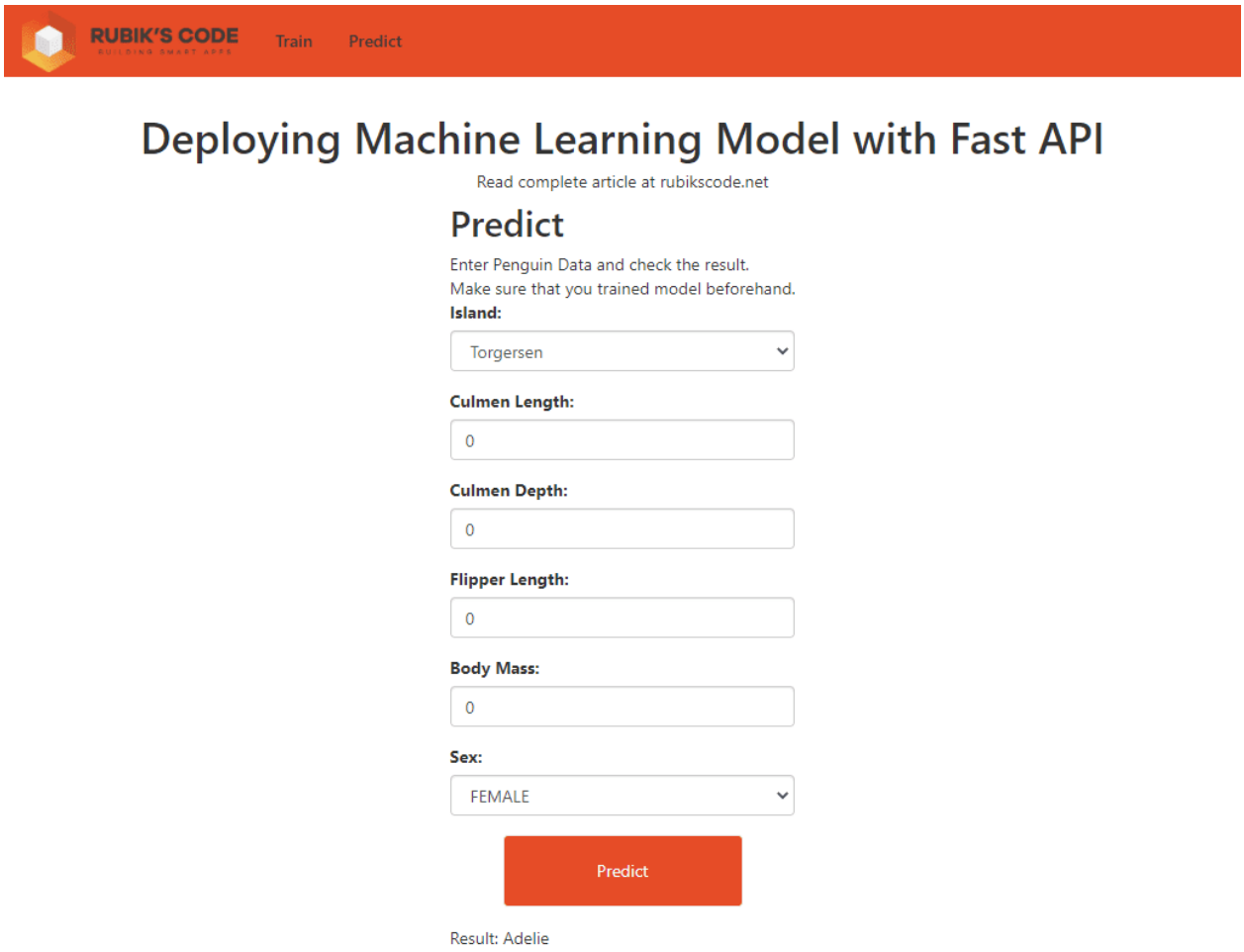


We will not go into implementation details too much here since the focus is on the *FastAPI* and Machine Learning model. In essence, there are two items in the toolbar of the application **Load** and **Predict**, which leads us to two web pages with the same names. The complete thing is implemented within two components: *load component* and *predict component*. There is one service too – *fastapi.service*. This service contains the calls to the REST API that our server-side implements.

In the *load* tab, you are able to pick the model that you want to load. Once you press the *Load* button, the HTTP request with this information is sent to the server-side and we expect it will load the defined machine learning model.



In the predict tab, the user is able to make **predictions** with the machine learning model that is trained via the previous tab. Here, users can enter parameters that describe a penguin, for which the user wants to make a prediction. Once the *Predict* button is pressed, this information is sent to the server which uses a machine learning model to make predictions and sends it **back** to the user.



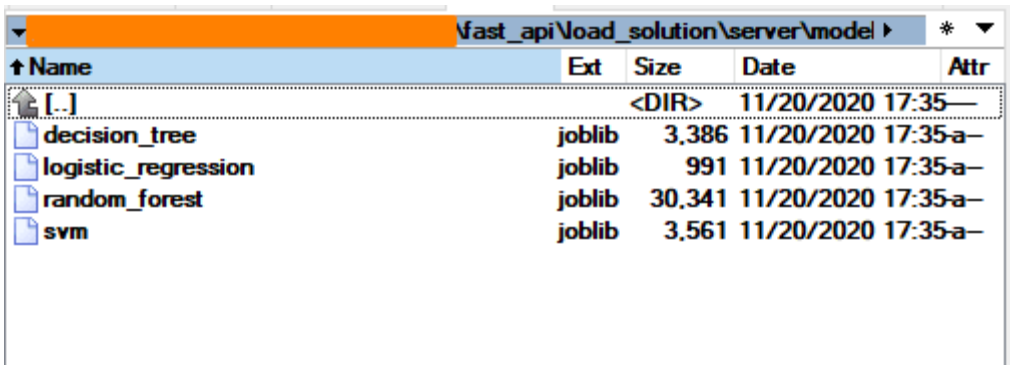
To run this application, open terminal and position into "*Chapter 5. Deployment\Chapter 5.2 Deployment with Fast API\client"* and run these commands:

```
npm install
ng serve
```

Once this is done, this web app will be available at *localhost:4200*. Ok, let's see what the server-side implementation looks like.

5.6.2.2 Model Trainer

You may have already noticed a new folder '*models*' in the solution folder. In this folder you can find following files:



These are already trained models. In fact, there is a script *train_models_script.py* that you can run if you want to **retrain** the following models again. It utilizes bits and pieces from the previous implementation, with one major difference. Models are not stored in the memory but are stored on the hard disk. Here is the script:

```
from sklearn.ensemble import RandomForestClassifier

from joblib import dump

def load_data(scale = True):
    data = pd.read_csv('./data/penguins_size.csv')

    data['culmen_length_mm'].fillna((data['culmen_length_mm'].mean()),
inplace=True)
    data['culmen_depth_mm'].fillna((data['culmen_depth_mm'].mean()),
inplace=True)
    data['flipper_length_mm'].fillna((data['flipper_length_mm'].mean()),
inplace=True)
    data['body_mass_g'].fillna((data['body_mass_g'].mean()), inplace=True)

    data["species"] = data["species"].astype('category')
    data["island"] = data["island"].astype('category')
    data["sex"] = data["sex"].astype('category')

    data["island_cat"] = data["island"].cat.codes
    data["sex_cat"] = data["sex"].cat.codes

    X = data.drop(['species', "island", "sex"], axis=1)
```

```
    if(scale):
        scaler = StandardScaler()
        X = scaler.fit_transform(X)

    y = data['species']
    spicies = {'Adelie': 0, 'Chinstrap': 1, 'Gentoo': 2}
    y = [spicies[item] for item in y]
    y = np.array(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
    return X_train, X_test, y_train, y_test

# Train SVM
X_train, X_test, y_train, y_test = load_data(scale=True)
model = SVC(kernel="rbf", gamma=0.1, C=500, verbose=True)
model.fit(X_train, y_train)
dump(model, './models/svm.joblib')

# Train Decision Tree
X_train, X_test, y_train, y_test = load_data(scale=False)
model = DecisionTreeClassifier(max_depth=5)
model.fit(X_train, y_train)
dump(model, './models/decision_tree.joblib')

# Train Random Forest
X_train, X_test, y_train, y_test = load_data(scale=False)
model = RandomForestClassifier(n_estimators=11, max_leaf_nodes=16, n_jobs=-1,
verbose=True)
model.fit(X_train, y_train)
dump(model, './models/random_forest.joblib')

# Train Logistic Regression
X_train, X_test, y_train, y_test = load_data(scale=True)
model = LogisticRegression(C=1e20, verbose=True)
model.fit(X_train, y_train)
dump(model, './models/logistic_regression.joblib')
```

The load_data function loads the data and does all the necessary preprocessing, just as the DataLoader did in the previous solution. Then we train models and save them in files using *joblib's* **dump** function. You can run this script like this:

```
python train_models_script.py
```

5.6.2.2 Server Side

The server side is located in the *load_solution/server* folder. Let's go through each of the components.

5.6.2.2.1 Data Contracts

The *train_parameters.py* file contains the model that is passed from the *load* tab of the web application. Essentially, this file contains a data contract that is used when the HTTP request is sent.

```
from pydantic import BaseModel

class TrainParameters(BaseModel):
    '''DTO for data coming from client-side.'''
    model: str
```

Note that we use ***Pydantic*** data validation and settings management library for type annotations. This library enforces type hints at runtime and provides user-friendly errors when data is invalid. The same library we use in *penguin_sample.py*, in which we define the data contract that is used when the HTTP request is sent from the *predict* page of the web application:

```
from pydantic import BaseModel

class PenguinSample(BaseModel):
    island: str
    culmenLength: float
    culmenDepth: float
    flipperLength: float
    bodyMass: float
    sex: str
    species: str
    def __getitem__(self, item):
        return getattr(self, item)
```

5.6.2.2.2 Model Loader

This component is in charge of **loading** the trained model. Through the constructor, it receives information about the algorithm that the user **picked** and it handles it from there on. Here is what it looks like:

```
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from penguin_sample import PenguinSample
from sklearn.preprocessing import StandardScaler
from joblib import load
import numpy as np

class ModelLoader():
    def __init__(self, algorithm):
        self.scaledData = True
        self._island_map = {'Torgersen': 2, 'Biscoe': 0, 'Dream': 1}
        self._sex_map = {'MALE': 2, 'FEMALE': 1}
        if(algorithm == 'svm'):
            self.model = load('./models/svm.joblib')

        if(algorithm == 'logistic regression'):
            self.model = load('./models/decision_tree.joblib')

        if(algorithm == 'decision tree'):
            self.scaledData = False
            self.model = load('./models/random_forest.joblib')

        if(algorithm == 'random forest'):
            self.scaledData = False
            self.model = load('./models/logistic_regression.joblib')

        self.scaler = StandardScaler()

    def prepare_sample(self, raw_sample: PenguinSample):
        island = self._island_map[raw_sample.island]
        sex = self._sex_map[raw_sample.sex]

        sample = [raw_sample.culmenLength, raw_sample.culmenDepth,
raw_sample.flipperLength, \
                raw_sample.bodyMass, island, sex]
        sample = np.array([np.asarray(sample)]).reshape(-1, 1)

        if(self.scaledData):
            self.scaler.fit_transform(sample)

        return sample.reshape(1, -1)
```

```
    def predict(self, data: PenguinSample):
        prepared_sample = self.prepare_sample(data)
        return self.model.predict(prepared_sample)
```

Based on the parameters that we receive from the client-side, we load the correct model using *joblib's* **load** function. The two methods are there to perform predictions. The *prepare_sample* prepares a new sample for model processing, while the *predict* method runs the sample through the model and retrieves predictions.

5.6.2.2.3 REST API Module

The most important file is the *main.py* file. This file puts all other pieces together and builds the REST API with ***FastAPI***. Here is what that looks like:

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from model_loader import ModelLoader
from train_parameters import TrainParameters
from penguin_sample import PenguinSample

origins = [
    "http://localhost:8000",
    "http://127.0.0.1:8000/predict",
    "http://127.0.0.1:8000/load",
    "http://localhost:4200"
]

app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.model = ModelLoader('svm')
@app.post("/train")
async def train(params: TrainParameters):
    print("Model Loading Started")
    app.model =  ModelLoader(params.model.lower())
    return True
@app.post("/predict")
async def predict(data:PenguinSample):
```

```
    print("Predicting")
    spicies_map = {0: 'Adelie', 1: 'Chinstrap', 2: 'Gentoo'}
    species = app.model.predict(data)
    return spicies_map[species[0]]
```

First, we import all the necessary libraries. Since our server runs in *http://localhost:8000* and client runs at *http://localhost:4200*, we need to handle *CORS* policies. That is why we import ***CORSMiddleware***. This class is utilized during app initialization:

```
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
app.model = ModelLoader('svm')
```

Note that we added an instance of *ModelLoader* there. This instance is just a **placeholder** and it is replaced later on. There are two endpoints, *"/train"* and *"/predict"*, both accepting POST HTTP requests. With that, we have two functions to handle these requests. The *train* method creates a new instance of the *ModelLoader* based on the parameters received from the client. Then it runs the training of the model and returns the accuracy of the model:

```
async def train(params: TrainParameters):
    print("Model Loading Started")
    app.model =  ModelLoader(params.model.lower())
    return True
```

The *predict* method is receiving data from the predict tab of the web application. It calls the predict method from the *ModelLoader* and returns the predicted value:

```
@app.post("/predict")
async def predict(data:PenguinSample):
    print("Predicting")
    spicies_map = {0: 'Adelie', 1: 'Chinstrap', 2: 'Gentoo'}
    species = app.model.predict(data)
    return spicies_map[species[0]]
```

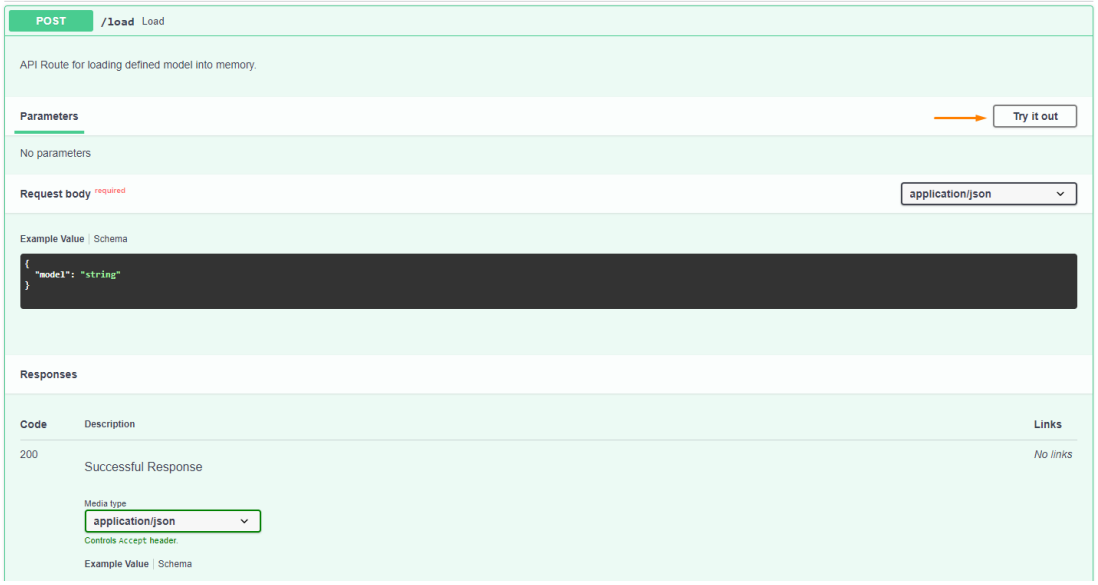
5.6.2.2.4 Testing the server-side

In order to run this solution, open a terminal, and go to the server folder, run this command:

```
uvicorn main:app --reload
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [28720]
INFO: Started server process [28722]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Once the app is running, go to localhost:8000/docs in the browser. You should see something like this:

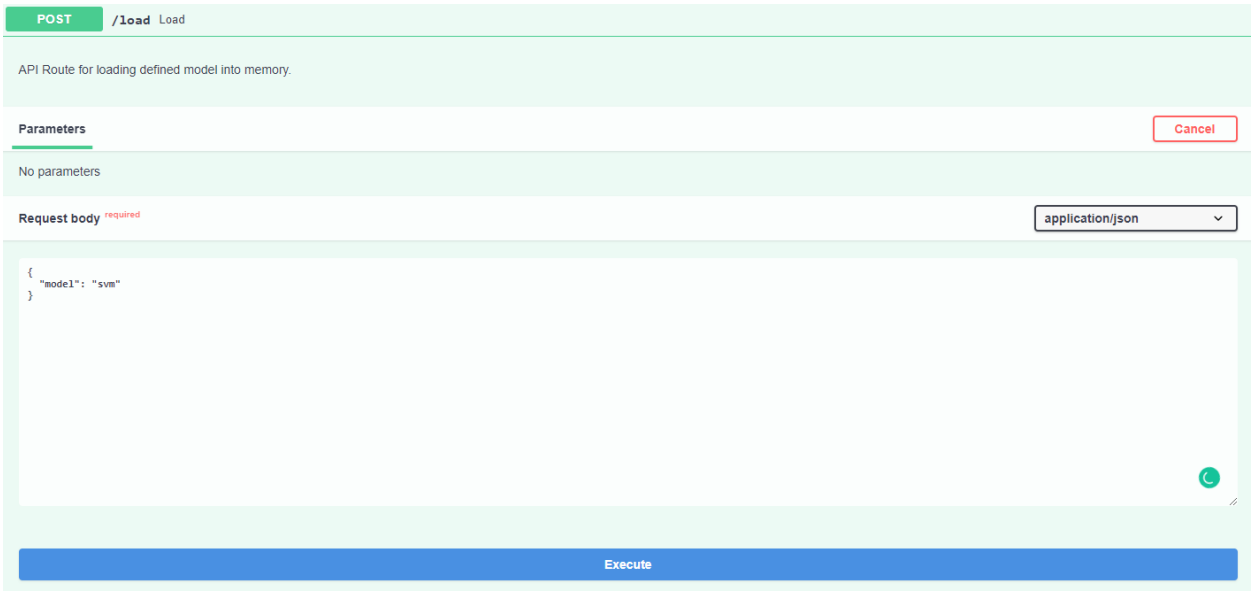
First, let's test the train endpoint. Expand it and click on the Try it out button:



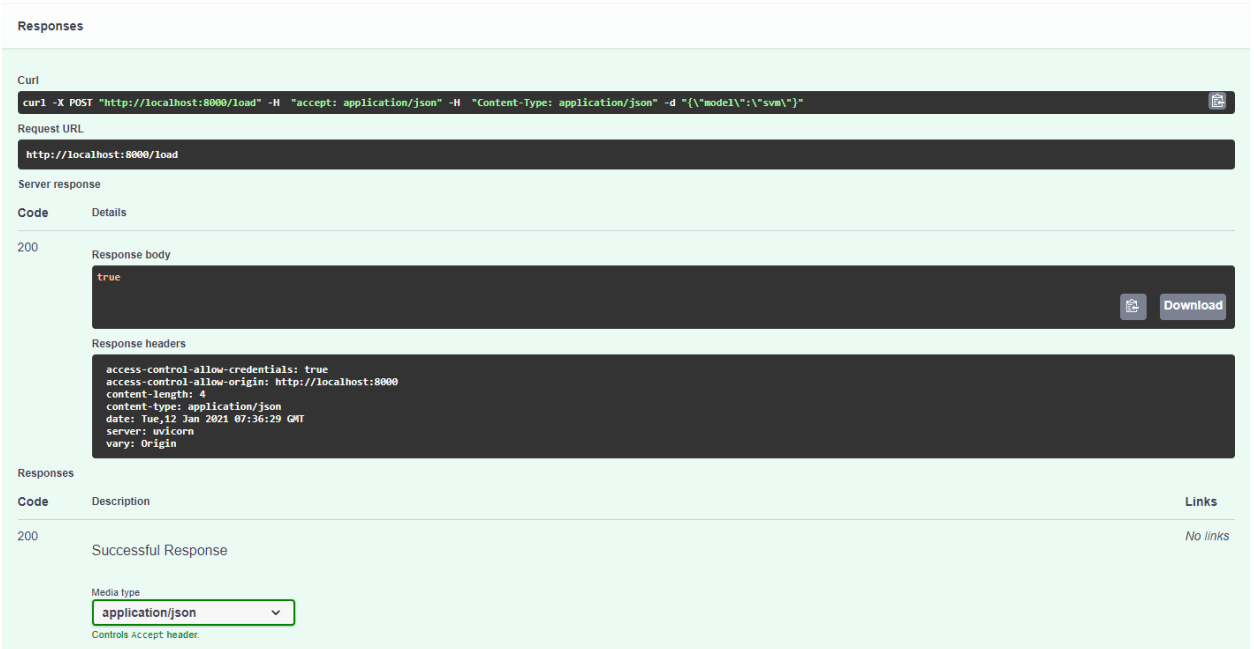
As a request body pass this json object:

```
{
    "model": "svm",
}
```

Then click on *Execute* button:



Here is what we get as a response:



In a similar way, we can test the *predict* endpoint.

5.6.3 Running it all together

To run the server side, you need to go to the *server* folder and use the command:

```
uvicorn main:app --reload
```

```
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [28720]
INFO: Started server process [28722]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

In another terminal, you need to go to the *load_solution\client* folder and run:

```
npm install
ng serve
```

```
√ Browser application bundle generation complete.

Initial Chunk Files   | Names         |      Size
vendor.js             | vendor        |   3.02 MB
polyfills.js          | polyfills     | 481.27 kB
styles.css, styles.js | styles        | 340.83 kB
main.js               | main          |  93.72 kB
runtime.js            | runtime       |   6.15 kB

                      | Initial Total |   3.92 MB

Build at: 2020-11-22T10:33:00.423Z - Hash: bf345d81dfd56983facb - Time: 10867ms
** Angular Live Development Server is listening on localhost:4200, open your
browser on http://localhost:4200/ **
√ Compiled successfully.
```

Once you have done so, you can go to *localhost:4200* and test the application:

6. Deep Learning

Thus far, we covered only algorithms that are usually called shallow learning algorithms. They are called that because they learn the model's parameters directly from the features in the training example. In general, most supervised learning algorithms are shallow, with the exception of artificial neural networks. These networks are built into several layers (we will see what that means in a bit) and are often called deep neural networks. That is why the entire field that uses deep neural networks is referred to as deep learning. However, we digress. In deep neural networks, unlike in shallow machine learning algorithms, the parameters are not learned from the features but from the output of the preceding layers. This means that some of the parameters of this specific machine learning algorithm are learned from other machine learning parameters. Mind-blowing, I know. So, let's learn where the motivation for deep neural networks comes from, how they function, and how we can implement them and use them.

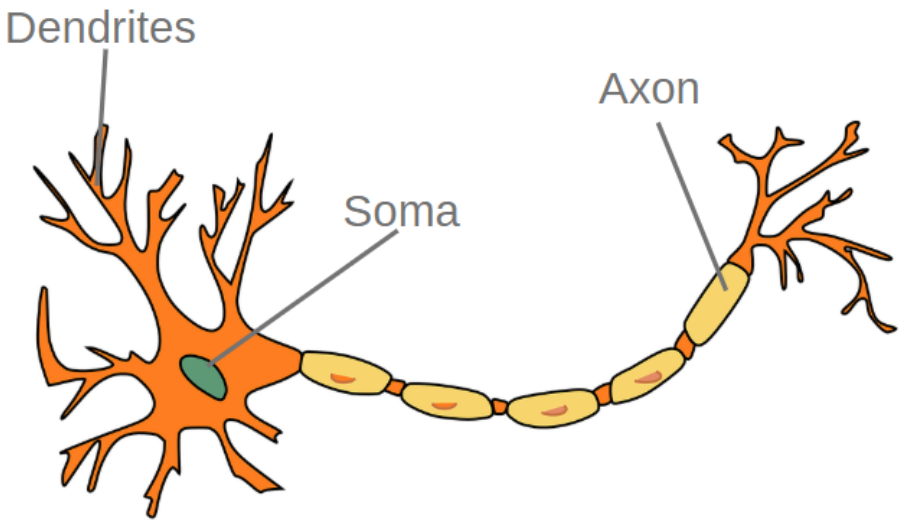
6.2 The biology behind Neural Networks

Deep learning has a seemingly simple premise: if it works in nature, it can work in computers. On paper, computers even have certain advantages. Computers can process complex numerical calculations in the blink of an eye. In theory, they even have bigger processing power than the human brain – *109* transistors with a switching time of *10-9* seconds against *1011* neurons, with a switching time of about *10-3* seconds. Still, brains outperform computers in almost every aspect. Entire fields of deep learning and artificial intelligence strive to make computers learn and make decisions based on that knowledge.

Neural networks are trying to imitate a few abilities the brain has, and computers don't. The first of those abilities is adaptability. As I have previously suggested, a computer should theoretically outperform the brain in any aspect, but the computer is static and rigid. It has great potential, but it fails to utilize it. On the other hand, the brain is reconfigurable, so to speak. A few years back, my dentist accidentally slightly damaged one of my nerves and, as a result, part of my lip was numb. In a matter of days, other nerves took over the function of the damaged nerve, and my lip was fine again. My dentist told me a bunch of stories where people had nerves cut in half and were still able to reach full recovery.

Computers cannot do that. I still haven't heard that a computer reconfigured itself in case of an error and that a hard disk took over the processor's functions, for example. The other problem is that the largest part of computers is just data storage, meaning that those parts don't do any processing whatsoever. The nerves in the brain, on the other hand, process data all the time, and they do it simultaneously. That is why the brain outperforms computers because there are numerous parallel little operations happening in your brain as you are reading this.

So, which processes from nature are Neural Networks trying to imitate exactly? As you are probably aware, the smallest unit of the nervous system is a neuron. These are cells with a similar and simple structure. Yet, by continuous communication, these cells achieve enormous processing power. If you put it in simple terms, neurons are just switches. These switches generate an output signal if they receive a certain amount of input stimuli. This output signal is input for another neuron. We can see the drawing of the real neuron in the image below:



There we can notice that each neuron has several components:

- Body, also known as soma
- Dendrites
- Axon

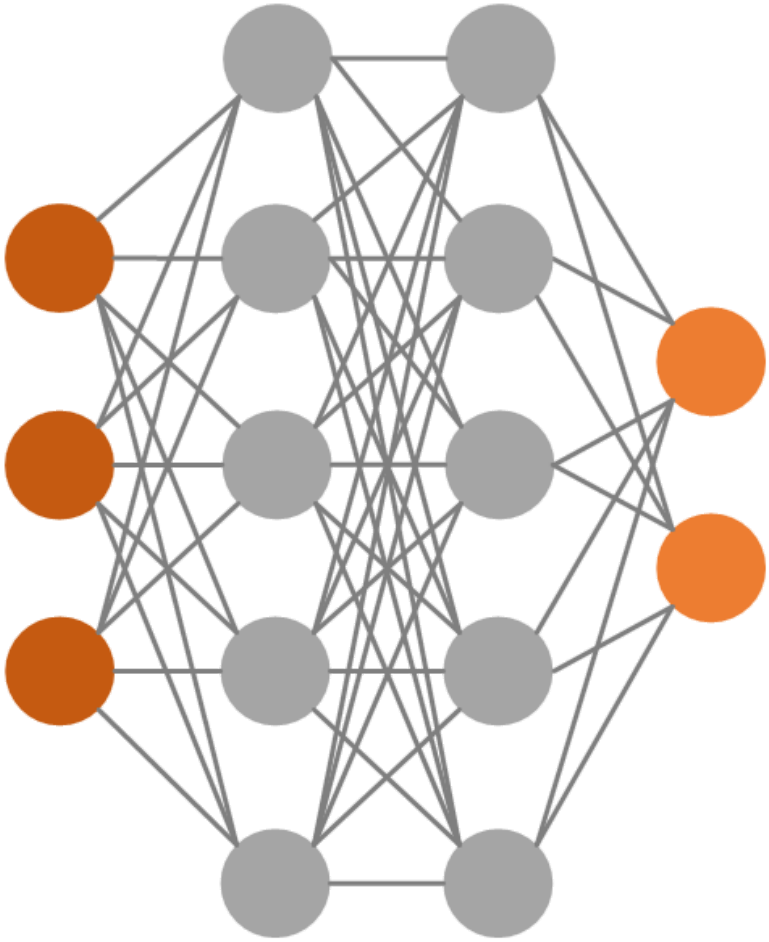
The body (soma) of a neuron carries out the basic life processes of a neuron. It controls the whole functionality of the neuron. Every neuron has a single axon, which is the long part of the cell; in fact, some of these go through the entire length of the spine. It acts like a wire, and it is an output of the neuron. Dendrites, on the other hand, are inputs of neurons, and each neuron has multiple dendrites. These inputs and outputs, axons and dendrites of different neurons, never touch each other, even though they come close. These gaps between axons and dendrites are called synapses. Signals are carried through synapses by neurotransmitter molecules. In general, various neurotransmitter chemicals exist, and each of them serves a different type of neuron. Among them are the famous hormones serotonin and dopamine, which are in charge of happiness. The amount and type of these chemicals will dictate how "strong" the input to the neuron is. And, if there is enough input signal on all dendrites, the soma will "fire up" the signal on

the axon and transmit it to the next neuron.

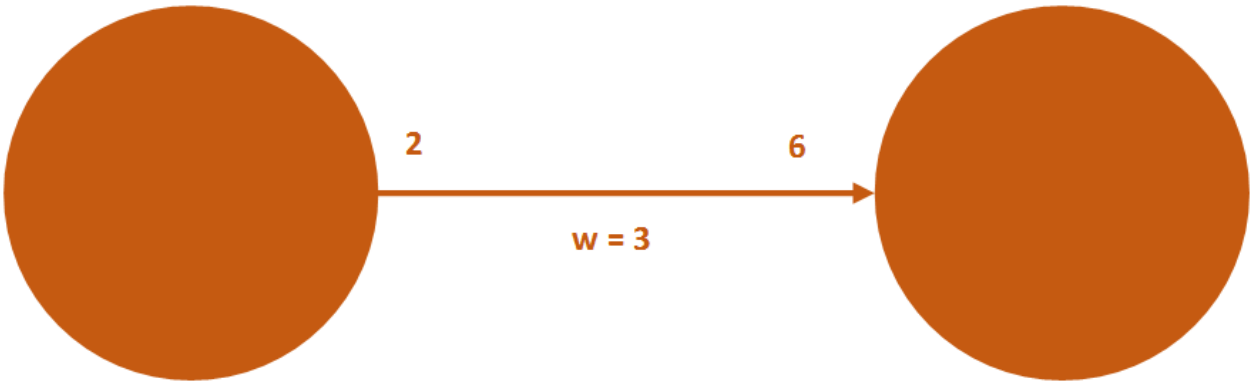
6.3 The Main Components and Concepts of Neural Networks

Before we dive into the concepts and components of neural networks, let's reflect on the goal of neural networks. We want these systems to learn certain processes the way humans do, without being explicitly programmed. For example, after we show an image of a dog to the neural network a few times, we can expect that next time we show it, our network will be able to deduce something like: "Ok, this is a dog", or something like: "There is a 93% chance that this is a dog". Looking from the outside, we can imagine them like a kind of black boxes, where we give them input information and they give us a result. So, how do we make them learn and behave in that kind of manner?

Based on the elements of the nervous system, Artificial Neural Networks are composed of small processing units – neurons and weighted connections between them.

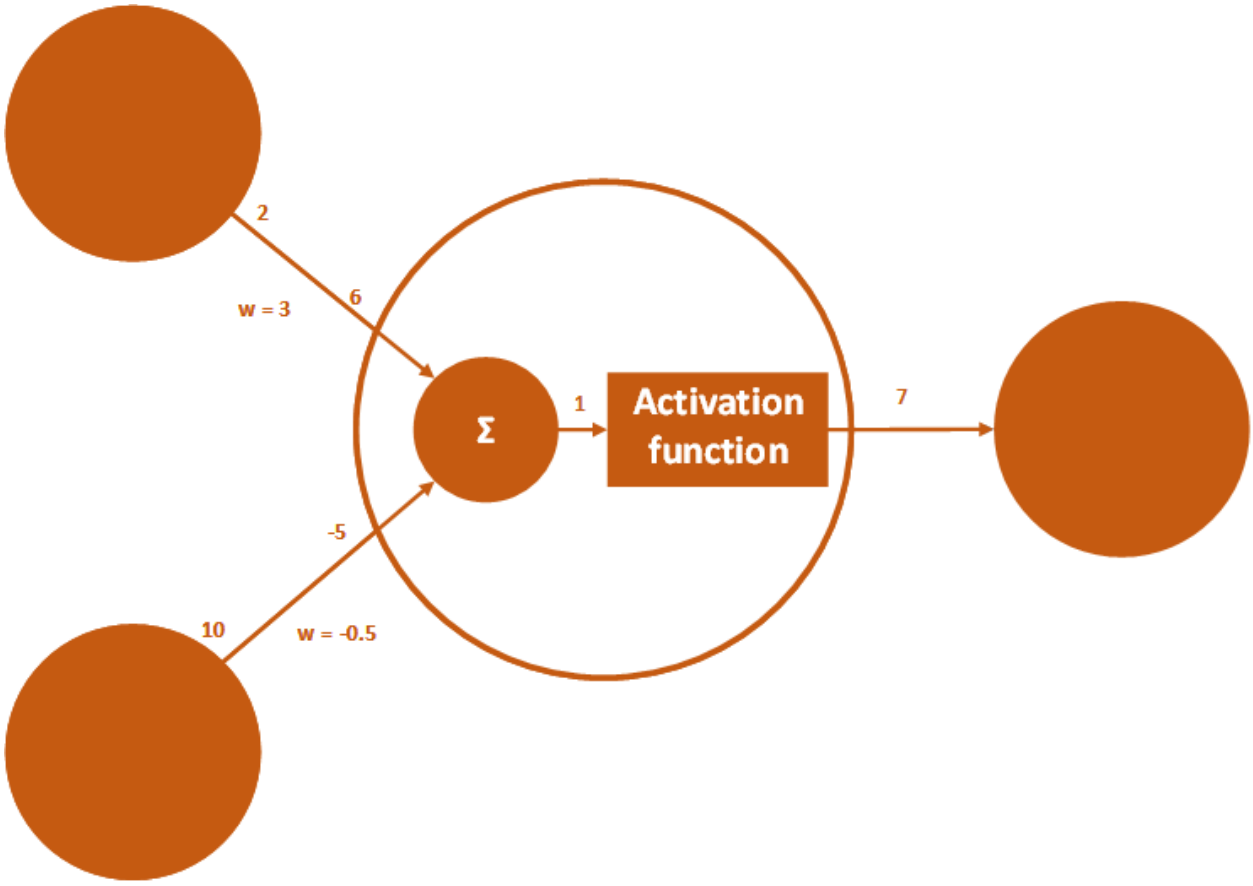


The weight of a connection simulates a number of neurotransmitters transferred among neurons, described in the previous chapter. Mathematically, we can define the Neural Network as a sorted triple *(N, C, w)*, where *N* is the set of neurons, *C* is the set *{(i, j)|i, j ∈ N}* whose elements are connections between neurons *i* and *j*, and *w(i, j)* is the weight of the connection between neurons *i* and *j*. For example, if the output of the neuron *i* is 2, and the weight of the connection w_{ij} is 3, the input to the neuron *j* is 2*3 = 6.



Neurons have this simple structure, and one might say that they alone are useless. Nevertheless, when they are connected, they become a powerful tool—something like a bee and a hive. A bee cannot produce that much honey alone, but thanks to the efforts of many bees working together, you have honey in your kitchen.

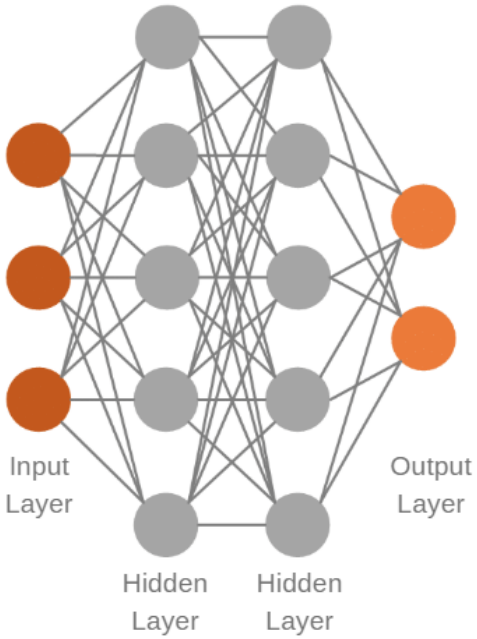
Usually, a neuron receives outputs from many other neurons as its input. Every output is transformed in consideration of the connecting weights to the so-called input network of that neuron. If you go through literature, this process might be referred to as a propagation function. In most cases, what you will see is that the propagation function is just the sum of the weighted inputs – weighted sum. Mathematically, it is defined like this: *net = Σ (i*w)*, where net is input network, *i* is the value of each individual input, and *w* is the weight of the connection through which he input value came. This value is then regulated with bias. Biases and weights of connections are the only interchangeable factors of neural networks. By changing these values, the neural networks learn. After the value coming from the input network is corrected by bias, it is then processed by something called the activation function. This function decides if the output of the neuron will be active. This function simulates the functionality of the biological soma, which will ignite the output only if there are strong enough stimuli on the input.



Every biological neuron has a certain threshold which has to be reached in order for a neuron output to be fired, so the activation function has to take this into consideration. Another aspect that this function has to be aware of is the previous state of the neuron. Biologically, this has no justification, but it simplifies the implementation of these networks. So, to sum up, the activation function transforms the input network of the neuron and the previous state of the neuron, into the neuron output, where the threshold plays an important role.

Now, as we can see, there are just two variable components of each neuron: the connection weight and neuron bias. By modifying the values of these components, we are making a learning mechanism or learning strategy. This mechanism is basically an algorithm which we use to train our network.

So far, we have learned that neural networks are composed of neurons and connections and that there are some adjustable parameters that we can use to create learning strategies. Still, there is another aspect that we haven't mentioned yet. The way that neurons are organized is also very important for the neural network. We are referring to this organization as the topology or architecture of a neural network. Usually, neurons are organized in several layers. Some neurons are just used to receive input from the outside world – the input neuron layer, while others are used exclusively to provide an output to the outside world – the output neuron layer.



Between these two layers, there can be an optional number of processing layers that are called hidden neuron layers. There are numerous architectures in the deep learning zoo today. In this book, we will cover only the portion of the most important ones. The architecture of the neural network is aligned with the business problem it tries to solve.

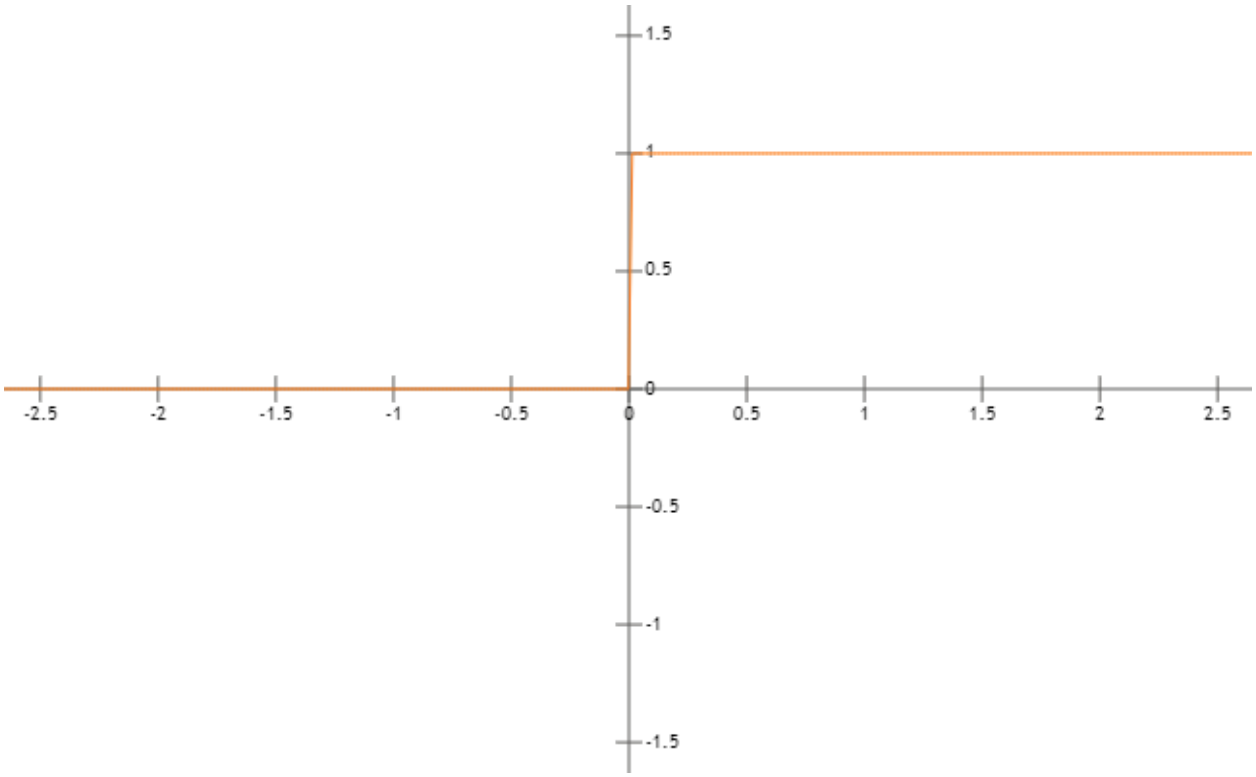
6.4 Activation Function

Now, we talked about the main parts of every neuron and how they are separated into layers. Also, we mentioned that the activation function will define the output value of our artificial neuron, i.e., whether that neuron is "activated" or not. Biologically speaking, this function can be modeled as the expected firing rate of the total input currently arising out of the incoming signals of synapses. This function in global will define how "smart" our Neural Network is and how hard it will be to train it, as well. The most common way is to assign one activation function to the whole layer of neurons, as we will explore further during the implementation sections of this book. One of the first activation functions that was ever used was the step function, also known as the **perceptron**.

6.4.1 Perceptron

This activation function has an interesting piece of history attached to it. This algorithm was first used back in 1957 in the custom-made computer called *Mark 1 Perceptron*, and it was used for image recognition. It was considered the future of artificial intelligence during the first take-off of the field. Even the author of the algorithm Frank Rosenblatt said that perceptron is "the embryo of an electronic computer that the Navy

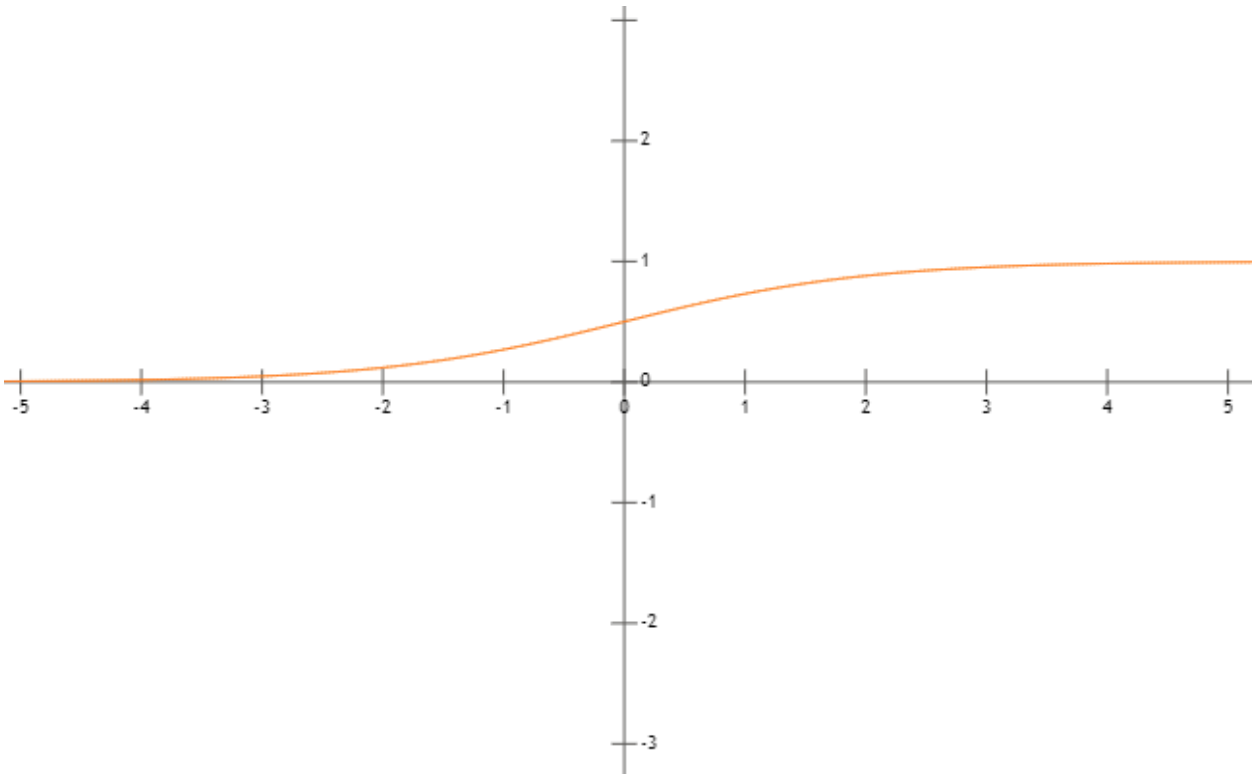
expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." Although it seemed promising in the beginning, it was proved that perceptron was not able to recognize many classes of patterns. This caused the field of neural networks to stagnate for a while.



In its essence, the perceptron is a step function that maps its real-valued vector input to a single binary output value. So, if the summed value of the input reaches a certain threshold, the value on the neuron's output will be 1; otherwise, it will be 0. Very straightforward and especially useful in the last layer of a network.

6.4.2 Sigmoid function

This function is smoother and more biologically plausible than a simple step function. Consider that one neuron gets fired out, meaning that it starts sending neurotransmitters through the synapses to another neuron. This will happen gradually over time and it will take some time to pass all the neurotransmitters. This is, of course, a very simplified description of that scenario. The sigmoid function looks like this:



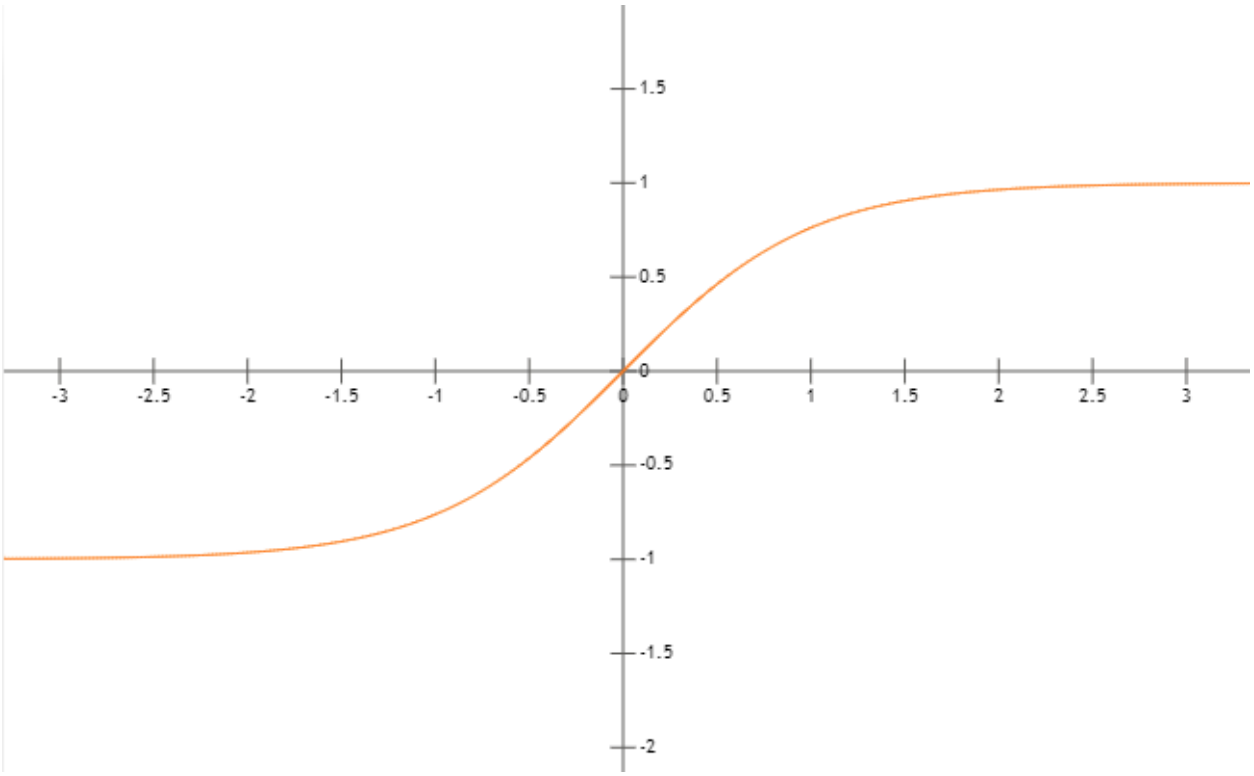
When using this function, we are calculating the neuron's output value using the formula:

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

where x is the *input* of the neuron. Basically, the weighted input is multiplied by a slope parameter. This function is heavily used for **linear regression** – one of the most well-known algorithms in statistics and machine learning. This function is also heavily used for the output layer of the neural network, especially for probability calculations.

6.4.3 Hyperbolic Tangent Function

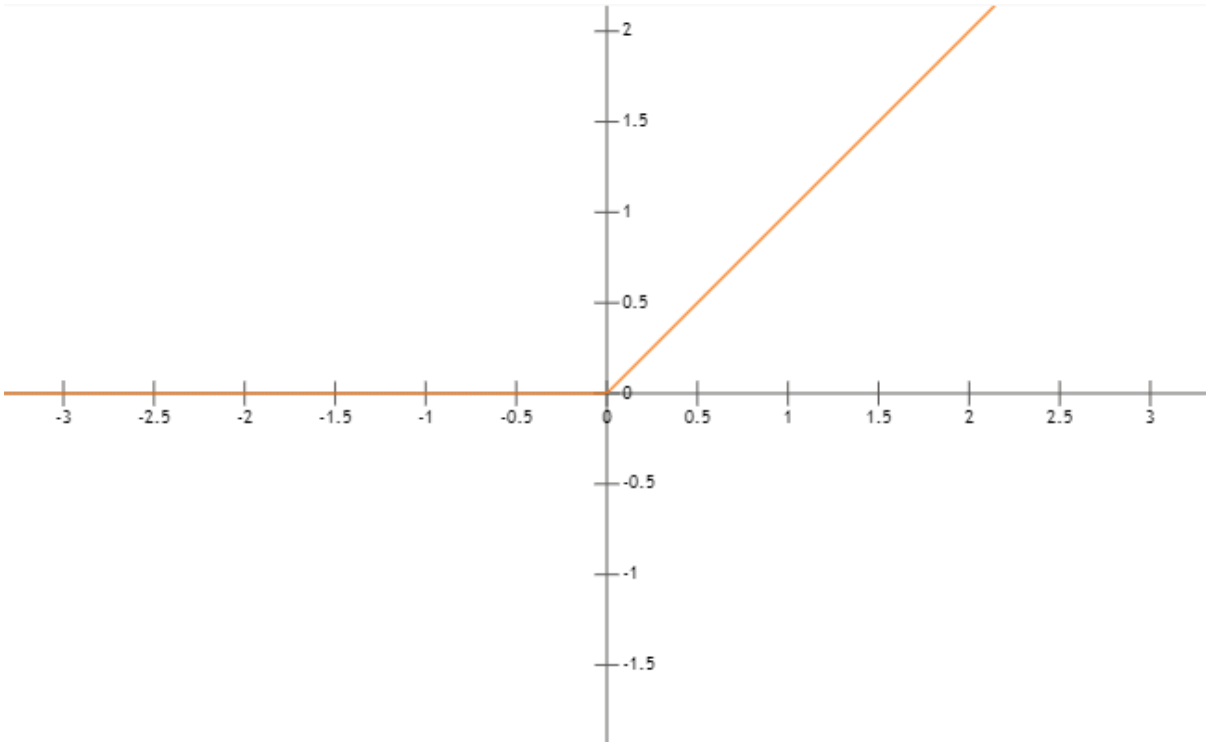
This function is similar to the *sigmoid* function. Output values of this function can vary from -1 to 1, indifference to the sigmoid function which covers values from 0 to 1. Although this is not what happens in neurons, biology-wise, this function gives better results when it comes to training neural networks in some cases. Sometimes, neural networks get "stuck" during training with the *sigmoid* function, meaning that when provided with the strongly negative input, the output of these networks is very near zero. This, in turn, messes up the learning process. Nevertheless, this is what the *hyperbolic tangent* function looks like:



The formula that we are using is *tahn(x)*, where *x* is *the input* of the neuron. It is used for the same purposes as the *sigmoid* function but in networks that have negative inputs. What is interesting to notice here is how this function gives better results in some cases, even though it doesn't have a strong biological interpretation.

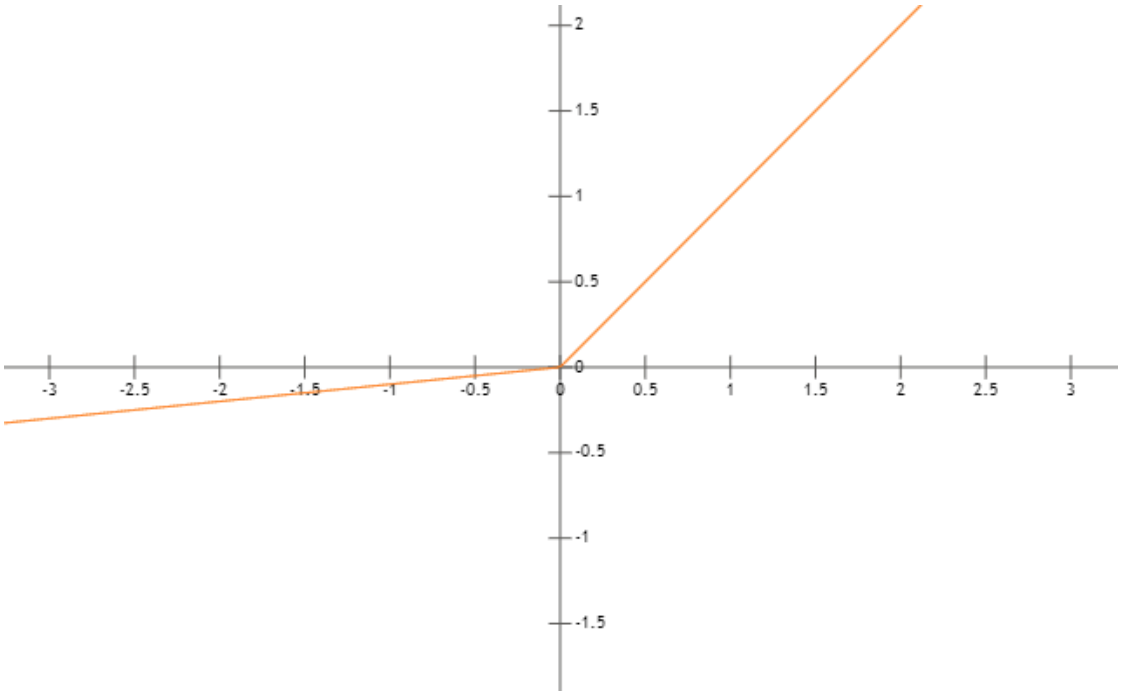
6.4.4 Rectifier Function

The rectifier function or *ReLU* for short, is probably the most popular activation function in the realm of neural networks. It is heavily used to solve all kinds of problems out there and for a good reason. This function is the most biologically plausible of all the functions described so far, and the most efficient function when it comes to training neural networks. Here is what it looks like:



The formula that we are using is *max(0, x)*, where *x* is the network input of the neuron. This is one of the reasons this function is very efficient, as it doesn't require normalization or other complicated calculations.

The very useful adaptation of this function is called *Leaky ReLU*. It looks somewhat different:



Every value that is 0 or below is giving value 0 if we use the standard *rectifier* function. This can cause a problem for networks that have negative values for weights (vanishing gradient). This situation is not happening in nature, of course, but in this world it's common. This means that a lot of values will be 0 and our network will not update during the learning process. So, we use the *Leaky ReLU* function, which will never come across this problem. Mathematically, it is expressed like this: $f(x)=1(x<0)(\alpha x)+1(x>=0)(x)f(x)=1(x<0)(\alpha x)+1(x>=0)(x)$.

6.2 Intro to Pytorch

Elon Musk made it clear that he doesn't like *Facebook*, however, *Tesla* still uses one piece of **technology** that comes from Zukenberg's laboratory. We can bet that it made its way into *SpaceX* as well and that it was an important part of the recent launch too. This technology has a big advocate in the AI world and that is Yan LeCunn, father of modern Convolutional Neural Networks. That is correct, we are talking about **PyTorch**. This deep learning framework is extremely popular among the research community and we can witness that in the past couple of years, it gained popularity in the industry as well. In this chapter, we explore the basic principles of *PyTorch* and how they can be used for simple tasks.

Generally speaking, *PyTorch* as a tool has two big **goals**. The first one is to be *NumPy* for **GPUs**. This doesn't mean that *NumPy* is a bad tool; it just means that it doesn't utilize the power of GPUs. The second goal of *PyTorch* is to be a **deep learning** framework that provides speed and flexibility. That is why it is so popular in the research community; it provides a platform in which users can quickly perform experiments. Apart from that, building models in *PyTorch* is very easy.

6.2.1 Installation

Installing *PyTorch* is quite easy and you can create a command that you need to run for the installation on **this page**. All you need to do is enter your environment details and run the command. Here is how I've set it up for my machine:

To verify that you have installed it correctly, run *Jupyter Notebook* and try to import *torch*:

```
import torch

torch.__version__
'1.5.0'
```

6.2.2 Tensors and Gradients

In general, a lot of concepts in machine learning and deep learning can be abstracted using multi-dimensional matrices – **tensors**. Mathematically speaking, tensors are described as geometric objects that describe linear relationships between other geometric objects. For the simplification of all concepts, tensors can be observed as **multi-dimensional** arrays of data. When we observe them like n-dimensional arrays, we can apply matrix operations easily and effectively. That is what *PyTorch* is actually doing. In this framework, a tensor is a **primitive unit** used to model scalars, vectors, and matrices located in the central class of the package *torch.Tensor*. We can do various operations with tensors, but first, let's see how we can create and initialize them.

6.2.3 Creating and Initializing Tensors

Here is how we can define scalar, vector, and matrix:

```
# Scalar
tensor_1 = torch.tensor(11.)
print('------ Scalar ------')
print(tensor_1)
print(tensor_1.dtype)
print(tensor_1.shape)
print('\n')

# Vector
tensor_2 = torch.tensor([11, 23, 9, 33])
print('------ Vector ------')
print(tensor_2)
print(tensor_2.dtype)
print(tensor_2.shape)
print('\n')

# Matrix
tensor_3 = torch.tensor([[5., 6],
```

```
                  [7, 4],
                  [9, 11]])

print('------ Matrix ------')
print(tensor_3)
print(tensor_3.dtype)
print(tensor_3.shape)
print('\n')

# 3D Array
tensor_4 = torch.tensor([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])

print('------ 3D Array ------')
print(tensor_4)
print(tensor_4.dtype)
print(tensor_4.shape)
print('\n')
```

In the code snippet above, we created a scalar, vector, matrix and 3D array. Note that for vectors, we use integers. For matrix and scalar we use floats. Also, we printed out their data types and shapes. Here is the output:

```
------ Scalar ------
tensor(11.)
torch.float32
torch.Size([])


------ Vector ------
tensor([11, 23,  9, 33])
torch.int64
torch.Size([4])


------ Matrix ------
tensor([[ 5.,  6.],
        [ 7.,  4.],
        [ 9., 11.]])
torch.float32
```

```
torch.Size([3, 2])


------ 3D Array ------
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
torch.float32
torch.Size([2, 2, 3])
```

As you can see, it is quite easy to create all the types of data that we need quickly. There are various tensor initialization functions. For example, you can create an uninitialized tensor, or a tensor initialized with zeros, ones or with a random value:

```
# Uninitialized (does not contain definite known values before it is used.)
tensor = torch.empty(2, 3)
print(tensor)

# Tensor with all zeroes
tensor = torch.zeros(2, 3)
print(tensor)

# Tensor with all ones
tensor = torch.ones(2, 3)
print(tensor)

# Tensor with randomly initialized values
tensor = torch.rand(2, 3)
print(tensor)
```

```
tensor([[1.0286e-38, 9.0919e-39, 9.3674e-39],
        [9.2755e-39, 1.4013e-43, 0.0000e+00]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0.1093, 0.3990, 0.5312],
        [0.2732, 0.6488, 0.4613]])
```

As we mentioned, one of the benefits of using *PyTorch* is that you can perform *NumPy* operations in parallel on GPU. This means that we can create tensors from *NumPy* arrays, and vice-versa create *NumPy* arrays from *PyTorch* tensors. The cool thing is that the tensor and *NumPy* array share underlying memory locations, so the changing one will change the other:

```
import numpy as np

numpy_array = np.array([[1, 2], [3, 4.]])
print(numpy_array)
print('\n')

tensor = torch.from_numpy(numpy_array)
print(tensor)
print('\n')

re_numpy_array = tensor.numpy()
print(re_numpy_array)
```

```
[[1. 2.]
 [3. 4.]]


tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)


[[1. 2.]
 [3. 4.]]
```

6.2.4 Tensor Operations

Torch tensors can easily be manipulated and there are various options to perform mathematical operations on them. For example, we can do something like this:

```
x = torch.tensor(3.)
w = torch.tensor(4.)
b = torch.tensor(5.)

res = w*x + b
res
```

```
tensor(17.)
```

There is also a different syntax of these operations that you can use. For example, if you want to add two tensors, there are several options that will give the same results:

```
x = torch.rand(2, 3)
y = torch.rand(2, 3)

result1 = x + y
result2 = torch.add(x, y)

print(result1)
print(result2)
```

```
tensor([[1.8197, 1.0063, 1.8231],
        [0.2407, 1.0290, 0.8837]])
tensor([[1.8197, 1.0063, 1.8231],
        [0.2407, 1.0290, 0.8837]])
```

Apart from this, you can reshape tensors using the *view* function:

```
tensor = torch.randn(3, 3)
reshaped = tensor.view(9)

print(tensor, tensor.size())
print(reshaped, reshaped.size())
```

```
tensor([[-0.4982,  1.0700,  1.0479],
        [-0.2169,  0.0897,  0.2885],
        [ 1.3385,  0.0141, -2.3748]]) torch.Size([3, 3])
tensor([-0.4982,  1.0700,  1.0479, -0.2169,  0.0897,  0.2885,  1.3385,  0.0141,
        -2.3748]) torch.Size([9])
```

Finally, it is important to mention that you can index tensors just like you would index *NumPy* arrays:

```
tensor = torch.randn(3, 3)
print(tensor[:, 1])
```

```
tensor([ 1.0461, -0.7462,  0.0028])
```

There are many other functions that you can do with *PyTorch* tensors, but for a start, these will get you going.

6.2.5 Gradients

Another core concept of *Pytorch* is the notion of gradients. They are located in the *torch.autograd* package. In essence, this package performs automatic differentiation and it is defined-by-run. This means that backpropagation is defined by how your code is run. In order to utilize this functionality, all you have to do is set the *PyTorch* tensor attribute .requires_grad as True. This will signal *PyTorch* to record all operations performed on that tensor. Once the computation or some interaction is finished, you can call function .backward() and have all the gradients computed automatically. The gradient for each tensor is stored into the *.grad* attribute of the class. Here is an example. First, let's create the 3×3 tensor:

```
tensor = torch.randn(3, 3)
print(tensor[:, 1])
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], requires_grad=True)
```

Then, let's add a scalar to it:

```
increased = tensor + 2
print(increased)
```

```
tensor([[3., 3., 3.],
        [3., 3., 3.],
        [3., 3., 3.]], grad_fn=<AddBackward0>)
```

Note that the grad attribute is now filled and that performed operations are recorded. This means that we can perform some more operations, calculate gradients and print them out:

```
temp = increased * increased * 3
output = temp.mean()
output.backward()

print(tensor.grad)
```

```
tensor([[4., 4., 4.],
        [4., 4., 4.],
        [4., 4., 4.]])
```

6.2.6 Linear Regression

Ok, this should be enough *PyTorch* background to help us implement a simple example that uses Multiple Linear Regression. Note that we are not following strict definitions here, and we are using some techniques that are more used in deep learning than in standard machine learning. This is because *PyTorch* is primarily a deep learning framework and in the next several chapters, we will use it to develop neural networks. Anyhow, as you probably know by now, we can describe Multiple Linear Regression by using the formula:

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

where *x* is the input vector (presented as *x1, x2, …, xn)* and *y* is the output target. The goal of Linear regression is to predict correct weights vector *w* and bias *b* that will for new values for input *x* give correct values for output *y.* Linear regression learns these values during the training process, where *y* and *x* values are known. Here is the dataset that we use for this example:

Region	Temperature	Rainfall	Humidity	Raspberry	Blueberry
Vojvodina	**73**	**67**	**43**	**56**	**70**
Macva	**91**	**88**	**64**	**81**	**101**
Pomoravlje	**87**	**134**	**58**	**119**	**133**
Istocna Srbija	**102**	**43**	**37**	**22**	**73**
Zapadna Srbija	**69**	**96**	**70**	**103**	**119**

So, we have five regions with three input values: the average temperature in Fahrenheit, average rainfall in mm, and average humidity in %. As the output, we have two values, crop yields for raspberry and blueberry. The goal is to create the linear regression model that will be able to learn to predict correct crop yields from the given data. Technically, our linear regression formula looks something like this:

$$y_{raspberry} = w_{11}rainfall + w_{12}humidity + w_{13}rainfall + b_1$$
$$y_{blueberry} = w_{21}rainfall + w_{22}humidity + w_{23}rainfall + b_2$$

Ok, so let's implement Linear Regression from scratch. First, we need to define input and output data and create *PyTorch* tensors from them:

```
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')

outputs = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')

inputs = torch.from_numpy(inputs)
outputs = torch.from_numpy(outputs)
```

Apart from that, we need to initialize weights and biases:

```
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)
```

In this case, our model is simple; it needs to multiply input values with weights and add bias to it.

$$X \quad \times \quad W^T \quad + \quad b$$

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}$$

We can define the model as a function:

```
def model(x):
    return x @ w.t() + b
```

Operator @ in *PyTorch* represents the matrix multiplication, and the *t()* function on a tensor returns the transposed value of it. Another thing we need to do is to define the loss function. In this example, we use the *Mean Squared Error* function. It does exactly what the name suggests; here is the formula:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f_i - y_i)^2$$

In essence, this function calculates the differences between predicted values and real values, squares them and calculates the average. Here is the implementation:

```
def mse(pred, real):
    difference = pred - real
    return torch.sum(difference * difference) / difference.numel()
```

The method torch.sum returns the sum of all the elements in a tensor, and the .numel method returns the number of elements in a tensor. Finally, we need to decide how to utilize this error in order to modify weights and biases. We do that using the optimization

technique Gradient Descent. In essence, what we are trying to do is calculate the best values for weights and biases that will minimize the loss. We do so by training our model in several epochs. During every epoch, we will calculate predictions, loss and gradients. Based on the gradients, we will adjust the weights by subtracting a small quantity proportional to the gradient. Here is what the training for 200 epochs looks like:

```
for i in range(200):
    predictions = model(inputs)
    loss = mse(predictions, outputs)

    print(f'Epoch: {i} - Loss: {loss}')

    loss.backward()
    with torch.no_grad():
        w -= w.grad * 1e-5
        b -= b.grad * 1e-5
        w.grad.zero_()
        b.grad.zero_()
```

Note that, every time, we call the *zero_()* method in the end to reset the gradients. The output looks like this:

```
...
Epoch: 187 - Loss: 135.3285675048828
Epoch: 188 - Loss: 134.859375
Epoch: 189 - Loss: 134.3938751220703
Epoch: 190 - Loss: 133.9319305419922
Epoch: 191 - Loss: 133.4735565185547
Epoch: 192 - Loss: 133.01876831054688
Epoch: 193 - Loss: 132.56735229492188
Epoch: 194 - Loss: 132.11936950683594
Epoch: 195 - Loss: 131.6748046875
Epoch: 196 - Loss: 131.23353576660156
Epoch: 197 - Loss: 130.7956085205078
Epoch: 198 - Loss: 130.36087036132812
Epoch: 199 - Loss: 129.929443359375
```

Note how the loss, i.e., the prediction error is getting smaller and smaller. In the end, we can compare predictions after 200 epochs and real values:

```
predictions = model(inputs)
predictions
```

```
tensor([[ 59.0074,  72.3907],
        [ 81.6869,  90.8632],
        [116.9137, 151.9167],
        [ 31.7110,  47.7871],
        [ 94.7926,  95.8309]], grad_fn=<AddBackward0>)
```

```
outputs
```

```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

Considering that we are performing validation on the training set, and that we still have some differences, we can say that these results could be better. Anyhow, we managed to get close and, in the process, learn how to use *PyTorch.* Now, we can proceed and build more powerful tools like neural networks.

6.3 Building Feed Forward Neural Networks

Finally, let's start with the *PyTorch* **implementation** of neural networks. First, we need to **import** all the necessary modules:

```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import FashionMNIST
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split
%matplotlib inline
```

You can see that we are pretty much only using *PyTorch* modules (except *NumPy* and *Matplotlib*). Using the *nn* module, we are able to create different neural network layers, and using *nn.functional* we can implement different **activation functions**. Apart from *PyTorch* libraries, we use some modules from *torchvision* library. Namely, we use the Fashion MNIST module, which contains FashionMNIST data. So, let's load this data:

```
data = FashionMNIST(root='data/', download=True, transform=ToTensor())

validation_size = 10000
train_size = len(data) - validation_size

train_data, val_data = random_split(data, [train_size, validation_size])

batch_size=128

train_loader = DataLoader(train_data, batch_size, shuffle=True, num_workers=4,
pin_memory=True)
val_loader = DataLoader(val_data, batch_size*2, num_workers=4, pin_memory=True)
```

First, we create an **object** of the Fashion MNIST class, which essentially contains all the necessary data. We **split** this data into training and validation sets. The training data is used during the training process of supervised learning, which is a method neural networks use to **learn** from the data. The validation of data is also used during the training process to evaluate how well neural networks perform. Usually, we would create one test set as well for the final evaluation of neural network performance on thus far unseen data, but for this simple tutorial, this is enough. Then we use the *DataLoader* class to shuffle the data and separate it into **batches** that are fed to neural networks during each training step.

Ok, moving on to the fun stuff! Let's build a neural network with *PyTorch*. Here is the complete *FFNN* class:

```
class FFNN(nn.Module):
    """Simple Feed Forward Neural Network with n hidden layers"""
    def __init__(self, input_size, num_hidden_layers, hidden_size, out_size,
accuracy_function):
        super().__init__()
        self.accuracy_function = accuracy_function

        # Create first hidden layer
        self.input_layer = nn.Linear(input_size, hidden_size)

        # Create remaining hidden layers
```

```
        self.hidden_layers = nn.ModuleList()
        for i in range(0, num_hidden_layers):
            self.hidden_layers.append(nn.Linear(hidden_size, hidden_size))

        # Create output layer
        self.output_layer = nn.Linear(hidden_size, out_size)

    def forward(self, input_image):
        # Flatten image
        input_image = input_image.view(input_image.size(0), -1)

        # Utilize hidden layers and apply activation function
        output = self.input_layer(input_image)
        output = F.relu(output)

        for layer in self.hidden_layers:
            output = layer(output)
            output = F.relu(output)

        # Get predictions
        output = self.output_layer(output)
        return output

    def training_step(self, batch):
        # Load batch
        images, labels = batch

        # Generate predictions
        output = self(images)

        # Calculate loss
        loss = F.cross_entropy(output, labels)
        return loss

    def validation_step(self, batch):
        # Load batch
        images, labels = batch

        # Generate predictions
        output = self(images)

        # Calculate loss
        loss = F.cross_entropy(output, labels)

        # Calculate accuracy
```

```
        acc = self.accuracy_function(output, labels)

        return {'val_loss': loss, 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]

        # Combine losses and return mean value
        epoch_loss = torch.stack(batch_losses).mean()

        # Combine accuracies and return mean value
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch: {} - Validation Loss: {:.4f}, Validation Accuracy:
{:.4f}".format( \
                                    epoch, result['val_loss'],
result['val_acc']))
```

Essentially, when you want to build some model using *PyTorch*, you can **inherit** *nn.Module* class. This way, you can just create different types of neural networks just by overriding several methods. This is one of the main reasons why *PyTorch* is so popular in the research community; it gives you "*precooked*" solutions with enough **flexibility**. We utilize that to create a model that receives several parameters through the constructor. It receives input size (i.e. the number of neurons in the input layer), the number of hidden layers and their size, the output size (i.e. the number of neurons in the output layer/number of categories), and the activation function that is going to be used in each layer.

In order for *PyTorch to know* that a model has certain layers, you need to create a class **attribute** for each layer. That is why we create *self.input_layer* and *self.output_layer* attributes. Note that for hidden layers, we use a different approach. We create an array of layers with *nn.ModuleList()*, since this is configurable through the *num_hidden_layers* parameter. For every layer, we use *nn.Linear*, which creates a simple layer with a defined number of **neurons**. Layers of this type perform a simple $y = wx + b$ function.

Apart from this, we need to override one important *nn.Module* method – **forward**. This function defines how the input will be processed in our neural network. This function basically connects all layers we defined in the constructor. Let's examine it in more detail:

```
def forward(self, input_image):
      # Flatten image
      input_image = input_image.view(input_image.size(0), -1)
            # Utilize hidden layers and apply activation function
      output = self.input_layer(input_image)
      output = F.relu(output)
            for layer in self.hidden_layers:
            output = layer(output)
            output = F.relu(output)
            # Get predictions
      output = self.output_layer(output)
      return output
```

First, we **flatten** the image, meaning we reshape it into an array. We do this because the input layer of our neural network can not receive 2D inputs. Then we pass this information through each linear layer and apply the rectifier or **ReLu** activation function. This activation function is most commonly used for hidden layers since it gives the best results. It is defined with the formula *relu(x) = max(0,x)*. Note that we don't use ReLu after the output layer. This is because, on the output, we expect to get **probabilities** for each category. Apart from the *forward* function, there are various other methods we implement in order to better control the **training** of the network. Methods *training_step* and *validation_step* define what is done during every training and validation pass:

```
def training_step(self, batch):
      # Load batch
      images, labels = batch
            # Generate predictions
      output = self(images)
            # Calculate loss
      loss = F.cross_entropy(output, labels)
      return loss

def validation_step(self, batch):
      # Load batch
      images, labels = batch

      # Generate predictions
      output = self(images)
            # Calculate loss
      loss = F.cross_entropy(output, labels)

      # Calculate accuracy
      acc = self.accuracy_function(output, labels)
            return {'val_loss': loss, 'val_acc': acc}
```

The *training_step* function takes the **batch** of images provided by the *DataLoader* and pushes them through the network to get the **prediction**. Underneath, PyTorch uses a forward function for this. Once this is done, we detect how well the neural network performed by calculating loss. The different functions can be used to measure the difference between the predicted data and the real data. In this example, we use **cross-entropy**. The method *validation_step* looks similar, but this method also calculates the accuracy of our predictions using the *accuracy_function* we passed through the constructor, stores loss, and returns the dictionary with this information. Once the validation epoch ends, we combine all these into an array, so we can see the history of the training process. Also, at the end of every epoch, we **print** out the information *validation_step* returned. The last two functionalities are implemented within *validation_epoch_end* and *epoch_end* methods:

```
def validation_epoch_end(self, outputs):
      batch_losses = [x['val_loss'] for x in outputs]
            # Combine losses and return mean value
      epoch_loss = torch.stack(batch_losses).mean()
            # Combine accuracies and return mean value
      batch_accs = [x['val_acc'] for x in outputs]
      epoch_acc = torch.stack(batch_accs).mean()
      return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

def epoch_end(self, epoch, result):
      print("Epoch: {} - Validation Loss: {:.4f}, Validation Accuracy:
            {:.4f}".format(epoch, result['val_loss'], result['val_acc']))
```

In order to automate the **training** process of the neural networks, we implement one more class *ModelTrainer*:

```
class ModelTrainer():
  def fit(self, epochs, learning_rate, model, train_loader, val_loader,
opt_func=torch.optim.SGD):
        history = []
        optimizer = opt_func(model.parameters(), learning_rate)

        for epoch in range(epochs):
            # Training
            for batch in train_loader:
                loss = model.training_step(batch)
                loss.backward()
                optimizer.step()
```

```
                optimizer.zero_grad()

            # Validation
            result = self._evaluate(model, val_loader)
            model.epoch_end(epoch, result)
            history.append(result)

        return history

    def _evaluate(self, model, val_loader):
        outputs = [model.validation_step(batch) for batch in val_loader]
        return model.validation_epoch_end(outputs)
```

This class has two methods, *fit* and *_evaluate*. Method *fit* is used for training. It receives a model, number of epochs (number of times the whole dataset will be passed through the network), the learning rate, and data loaders. For each **epoch**, we get baches from the loader and run it through the network by calling the *training_step* method. Then we get the loss and use the *backward* method to calculate **gradients**. Finally, we use the optimizer to update the **weights** of the network.

Alright, those are the classes that describe the **general** neural network and the general training process. We need to get more specific and utilize this class for our problem. To do so, we first need to implement the accuracy function:

```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))
```

Also, we define the helper function for plotting history:

```
def plot_history(history):
    losses = [x['val_loss'] for x in history]
    plt.plot(losses, '-x')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Loss and Accuracy');
```

Finally, we can put all these pieces together and create the object of FFNN. We create a neural network with 3 hidden layers and with 32 neurons in each hidden layer. Note that the input size is 28×28=784 and the output size is 10 since we have 10 **categories** of clothes:

```
input_size = 784
num_classes = 10

model = FFNN(input_size, num_hidden_layers, 32, out_size=num_classes,
accuracy_function=accuracy)
print(model)
```

```
FFNN(
  (input_layer): Linear(in_features=784, out_features=32, bias=True)
  (hidden_layers): ModuleList(
    (0): Linear(in_features=32, out_features=32, bias=True)
    (1): Linear(in_features=32, out_features=32, bias=True)
    (2): Linear(in_features=32, out_features=32, bias=True)
  )
  (output_layer): Linear(in_features=32, out_features=10, bias=True)
)
```

Let's train it and plot the history and accuracy:

```
model_trainer = ModelTrainer()

training_history = []
training_history += model_trainer.fit(5, 0.2, model, train_loader, val_loader)

plot_history(training_history)
```

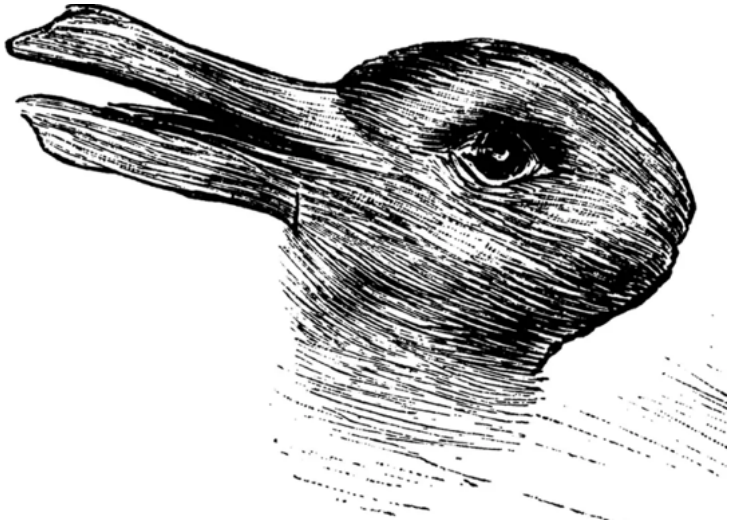
```
Epoch: 0 - Validation Loss: 0.7095, Validation Accuracy: 0.7266
Epoch: 1 - Validation Loss: 0.5858, Validation Accuracy: 0.7959
Epoch: 2 - Validation Loss: 0.5417, Validation Accuracy: 0.7789
Epoch: 3 - Validation Loss: 0.4696, Validation Accuracy: 0.8247
Epoch: 4 - Validation Loss: 0.4303, Validation Accuracy: 0.8371
```

Notice how loss is getting lower and the accuracy is getting better. In the end, after only 5 epochs, we reached the accuracy of 83%.

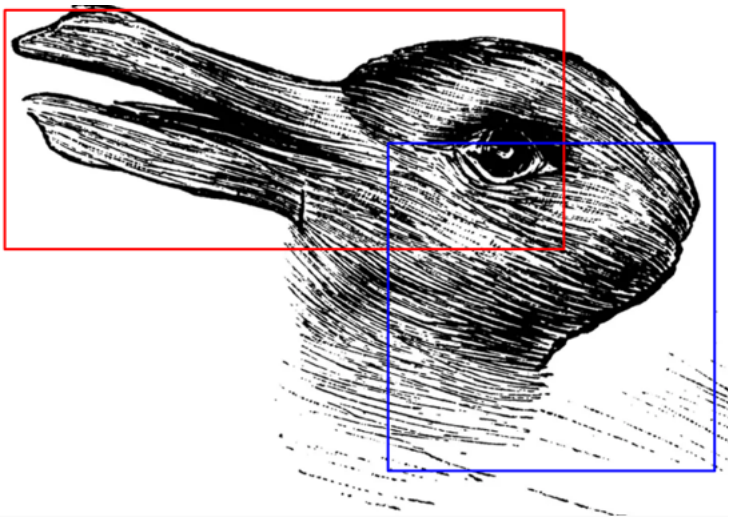
6.4 Convolutional Neural Networks

Have you ever wondered how *Facebook* knows how to suggest the right friend to tag? Speaking of, how does Google's image search algorithm work? Yes, your assumptions are correct; there is a neural network involved in both tasks. To be more precise, we are talking about a special kind of neural network - *Convolutional Neural Networks*. Even though it sounds like a weird mixture of biology and computer science (everything related to neural networks kinda does), this is one very effective mechanism used for image recognition. Of course, it is motivated by biological systems and the ways the brain works, specifically, the visual cortex.

Take a look at the image below and tell me what you see:



You will say it depends, right? Either a rabbit or a duck, depending on how you observe the image. Or something down that line. What does that mean, "depending how you observe it"? Take a look now at the modified image below.



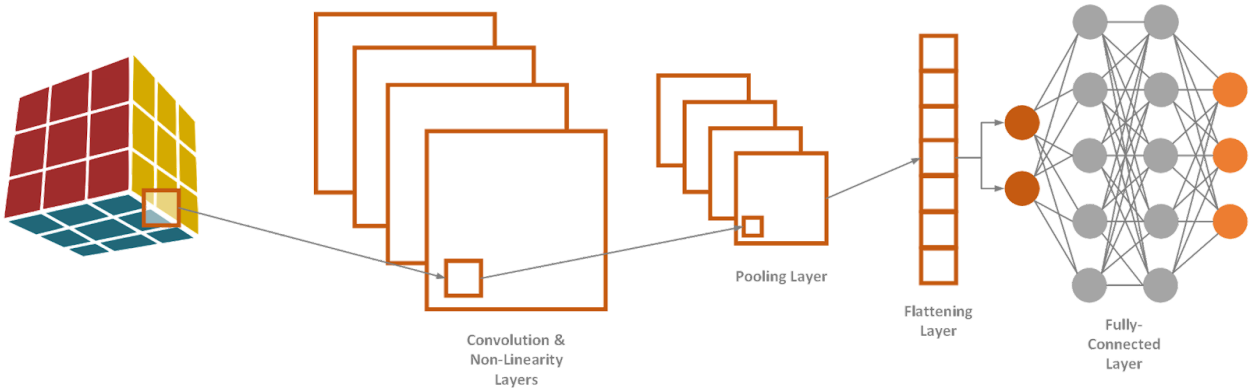
If you pay more attention to the part of the image in the red rectangle, your answer to the question "what do you see in this image" will be "a duck". On the other hand, if you observe part of the image under a blue rectangle, you would answer - "a rabbit". This is because of the way that our visual cortex functions.

Individual neurons in this section of the brain respond to stimuli only in a restricted region of the visual field known as the receptive field. Because these fields of different neurons overlap, together, they make the entire visual field. This effectively means that certain neurons were activated only if there is a certain attribute in the visual field, for example, a horizontal edge. So, different neurons will be "fired up" if there is a horizontal edge in your visual field, and different neurons will be activated if there is, let's say, a vertical edge in our visual field. Based on the fired neurons, our brain will classify an image a certain way. Meaning, if we observe a part of the above image within the red square, only certain neurons of our visual cortex will be activated, and based on this, our brain will say: "OK, that is a duck". In a nutshell, if there is a certain feature in our visual field, specific neurons will be activated, and others won't. This was proven by a fascinating experiment done by Hubel and Wiesel in 1962.

So, how do Convolutional Neural Networks use this for image recognition? Well, they use this idea to differentiate between given images and figure out the unique features that make a rabbit a rabbit or a duck – a duck. This process is happening in our minds subconsciously. Convolutional Neural Networks do the same thing, but they are first detecting the lower level features like curves and edges and then they build it up to more complex concepts, like eyes and ears.

In order to achieve the functionality we talked about, the Convolutional Neural Network processes an image through several layers. Let's do an overview of them and their purposes:

- Convolutional Layer – used to detect features
- Non-Linearity Layer – introducing non-linearity to the system
- Pooling (Downsampling) Layer – reduces the number of weights and controls overfitting
- Flattening Layer – prepares data for Classical Neural Network
- Fully-Connected Layer – standard Neural Network used for classification



Basically, in the end, the Convolutional Neural Network uses standard feed forward Neural Network for solving classification problems, but it uses other layers to prepare data and detect certain features before that. Let's dive into the details of each layer and their functionalities.

6.4.1 Convolutional Layer

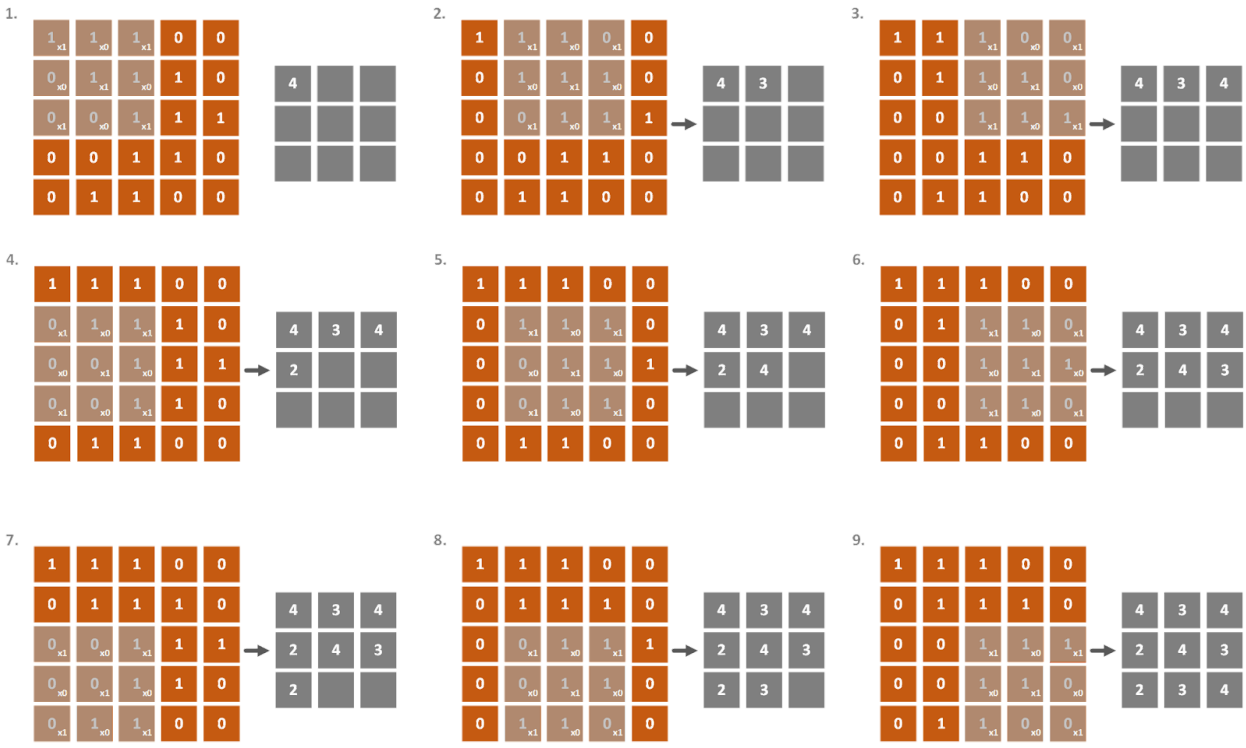
This is the main building block of Convolutional Neural Networks. It does the heavy lifting without which the rest of the activities would be impossible. It is in charge of detecting features. Effectively, this means that it detects special attributes of an image. This is done by applying a certain filter to the image that will extract some of the low-level and later higher-level attributes on an image. When applied, these filters look something like Photoshop filters. For example, we can use a filter that will detect edges. A filter is usually a multi-dimensional array of pixel values – for example, 5x5x3. Images usually have three channels - RGB - and their values are represented in separate matrices. That is why we represent images with tensors. In this particular case, we have an image (filter) that is 5x5 and has the channels. So, how do we apply the filter? Let's process one channel of the image that looks like this:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

For detecting features in this chanel, we will use one channel of the filter that looks like this:

1	0	1
0	1	0
1	0	1

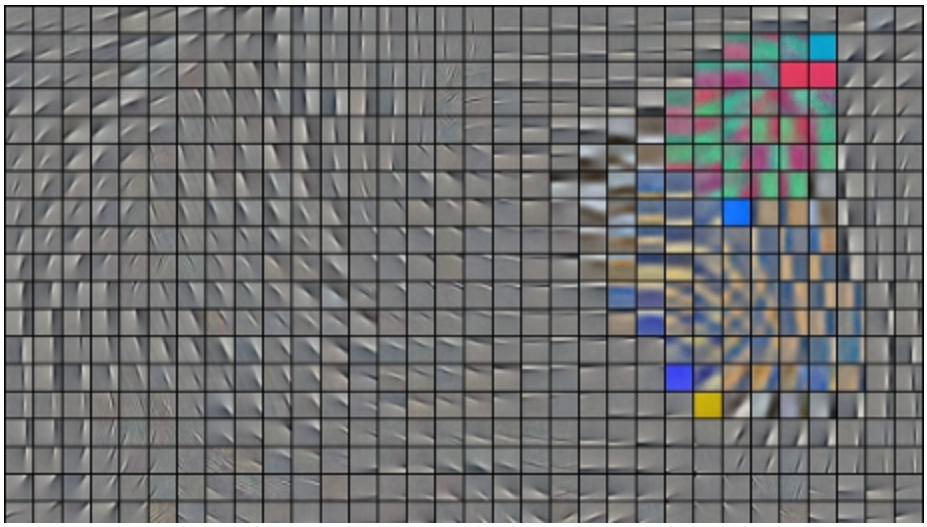
Now, take a look at how this process of applying the filter, also known as convolution, is done:



Firstly, we position the filter in the first location of the image, the top left corner. There we use an element-wise multiplication between two matrices and store the result in the output matrix. Then we move that filter to the right by 1 pixel (also known as "stride") and repeat the process. We do that as long as we can move our filter to the right. When we cannot do that anymore, we go to the next row and apply the filter in the same way. We do this until our output matrix is complete. The reason our output matrix is 3×3 is that there are nine positions in which you can fit the 3×3 feature in the 5×5 image. These 9 numbers are mapped to the 3×3 matrix.

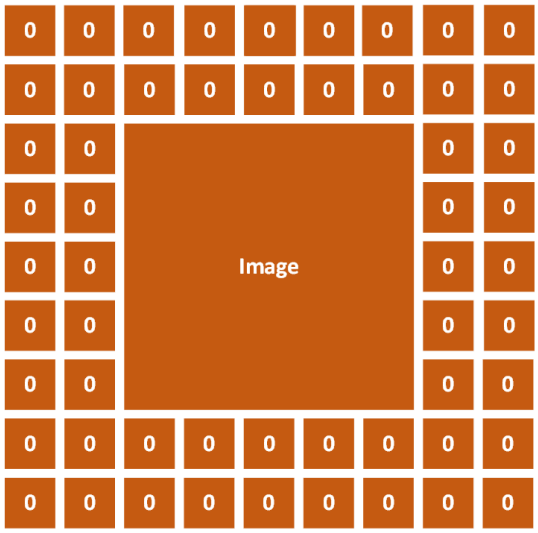
What is the meaning of the output matrix? This matrix is often called Feature Map. It indicates where the feature, represented by the filter, is located in the image. In a nutshell, by moving the filter through the image and using simple matrix multiplication, we detect certain features in the image. Usually, we use more than one filter and detect multiple features, which means that we have more than one convolutional layer in the network.

When we apply the first filter, we are creating one Feature Map and detect one kind of feature. Then we use the second filter to create a second Feature Map that detects another kind of feature. These filters can be simple, as we could see in the example, but they can get quite complicated if we want to extract some complex features from the image. Take a look at the image below, where multiple filters are represented.



We can apply another convolution process to the output matrix. That way, we are moving from the low-level features, like a horizontal or vertical line, to more complex features. After that, we could apply another convolutional layer and detect even more complex features and so on. This is how we get such good results with Convolutional Neural Networks.

Another thing that we have mentioned earlier, but didn't explain in detail, is stride. This term is usually used in combination with the term padding. The stride controls how the filter is convolved around the input image. In the example above, the stride was 1 pixel. Meaning the filter was moved pixel by pixel over the image. This value is, of course, configurable. In the end, the stride affects the size of the output matrix of the Feature Map. At the early stages of the Convolution Neural Network, when we are applying our first layers, we want to preserve as much information as possible for other Convolutional Layers. That is why padding is used. You may notice that the Feature Map is smaller than the original input image. Padding would add zero values to this map to preserve the size, like on the image below:



6.4.2 Non-linearity

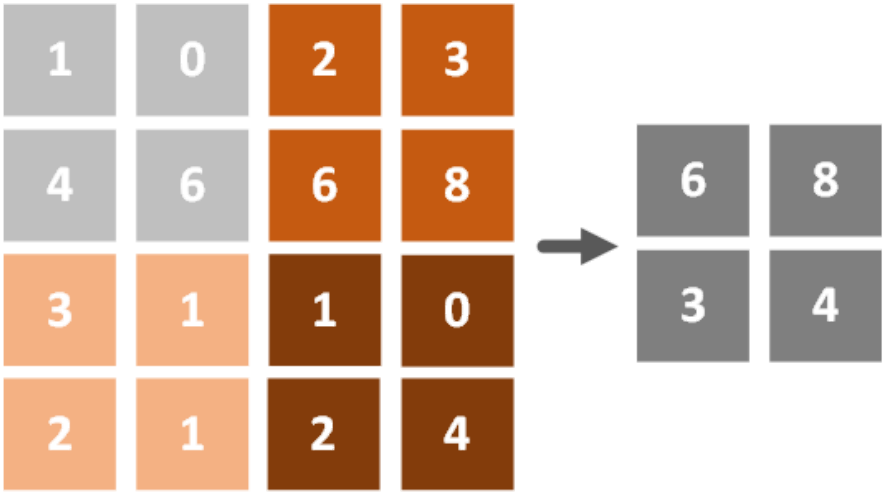
After every convolutional layer, we usually have a non-linearity layer. Why is linearity in the image a problem? Just like a regular Neural Network, the Convolutional Neural Network would behave just like a single perception because the sum of all the layers would still be a linear function, meaning the output could be calculated as a linear combination of the outputs. This layer is also called the activation layer because we use one of the activation functions. In the past, nonlinear functions like sigmoid and *tahn* were used, but it turned out that the function that gives the best results when it comes to the speed of training of the Neural Network, is the Rectifier function. So, this layer is often a ReLU Layer, that removes linearity by setting values that are below 0 to 0 since the Rectifier function is described with the – $f(x) = max(0, x)$. Here is how it looks once applied to one of the feature maps:

On the second image, the feature map, the black values are the negative ones, and after we apply the rectifier function, the black ones are removed from the image.

6.4.3 Pooling Layer

This layer is commonly inserted between the successive convolutional layers in Convolutional Neural Networks. The functionality of this layer is to reduce the spatial size of the representation and with that, the number of parameters and computation in the network. This way, we are also controlling the overfitting in our network. There are different pooling operations, but the most popular one is called max pooling. The other pooling algorithms, like average pooling, or L2-norm pooling, work on the same principle.

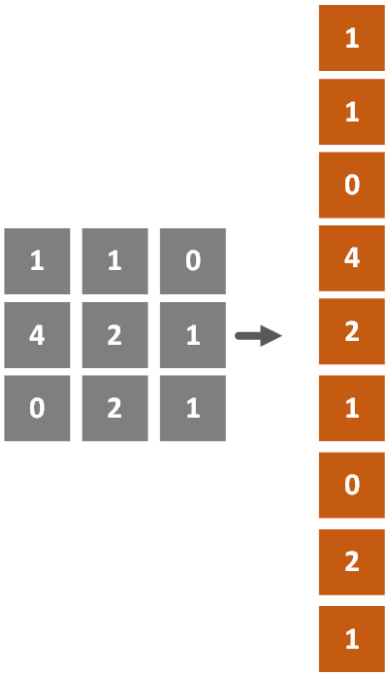
Here we will have a sort of a filter once again. Take a look at the picture below. We used the max pooling filter size 2×2 on the 4×4 image. As you have already guessed, the filter picks the largest number of the part of the image it covers. This way, we end up with smaller representations that contain enough information for our Neural Network to make correct decisions.



However, a lot of people are against pooling layers, and replace them with additional Convolutional Layers with a larger stride. Apart from that, newer generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs), discard pooling layers completely. It seems that this layer will soon be obsolete.

6.4.4 Flattening Layer

This is a simple layer that is used to prepare data to be the input of the final and most important layer – Fully-Connected Layer (which is our Neural Network). Since, in general, neural networks receive data in one dimension in the form of an array of values, this layer uses data passed from the pooling layer or convolutional layer and squashed the matrices into arrays. Then, these values are used as an input to the neural network. Here is a visual representation of the flattening process:



6.4.5 Fully-Connected Layer

The final layer and the layer that does the actual classification is the so-called Fully-Connected layer. This layer takes input from the flattening process and feeds and forwards it on through the feed-forward neural network we examined in the previous chapters.

6.4.6 The Architectures of Convolutional Neural Networks

A common way of building Convolutional Neural Networks is to stack a few Convolutional Layers and, after each of them, add a ReLU layer. After that, they are followed by pool layers and the flattening layer. Finally, several Fully-connected layers along with additional ReLU layers are added. Keep in mind that, in the end, there should always be fully-connected layers, which looks something like this:

Input Image -> [[Conv -> ReLU]*N -> Pool?]*M -> Flattening -> [FC -> ReLU]*K -> FC

Some of the architectures in the field of Convolutional Networks are quite famous:

- *LeNet* – this was the first successful application of Convolutional Networks. It was developed by Yann LeCun in the 1990s and it was used to read zip codes, simple digits, etc.
- *AlexNet* – this was the network that was presented in the *ImageNet ILSVRC* challenge back in 2012. It is actually the network that popularized the Convolutional Networks since it outperformed all other contestants by far. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton.

- *GoogLeNet* – The ILSVRC 2014 winner was a Convolutional Network from Google. They used average pooling layers to dramatically minimize the number of parameters in the network. There are several follow-up versions to the GoogLeNet.
- *VGGNet* – Convolutional Neural Network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. This network proved that the depth of the network is crucial for good performances. It has sixteen convolutional layers.
- *ResNet* – Developed by Kaiming He et al. was the winner of ILSVRC 2015. You can watch this cool video where the topic is described in depth.

6.5 Building the Convolution Neural Network

First, convolution layers **detect** features (line, curve, etc.) of the image using filters. They create the so-called **feature maps** that contain information about where in the image a certain feature is located. These maps are further compressed by the **pooling** layers, after which they are **flattened** into 1D array. Finally, a feed-forward network is used for classification, which is in this context called **fully connected**. *PyTorch nn* module provides a number of other layer types, apart from the *Linear* that we already used. Here is how we can implement the process described above:

```
class CNN(nn.Module):
    """Simple Convolutional Neural Network"""
    def __init__(self, accuracy_function):
        super().__init__()
        self.accuracy_function = accuracy_function

        # Create Convolutional Layers
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

        # Create Fully Connected Layers
        self.fc1 = nn.Linear(1600, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, input_image):
        # Convolution, ReLu and MaxPooling
        output = self.conv1(input_image)
        output = F.relu(output)
        output = F.max_pool2d(output, (2, 2))

        output = self.conv2(output)
        output = F.relu(output)
        output = F.max_pool2d(output, 2)
```

```
        # Flatten
        output = output.view(-1, self.num_flat_features(output))

        # Fully Connected
        output = self.fc1(output)
        output = F.relu(output)
        output = self.fc2(output)

        return output

    def training_step(self, batch):
        # Load batch
        images, labels = batch

        # Generate predictions
        output = self(images)

        # Calculate loss
        loss = F.cross_entropy(output, labels)
        return loss

    def validation_step(self, batch):
        # Load batch
        images, labels = batch

        # Generate predictions
        output = self(images)

        # Calculate loss
        loss = F.cross_entropy(output, labels)

        # Calculate accuracy
        acc = self.accuracy_function(output, labels)

        return {'val_loss': loss, 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]

        # Combine losses and return mean value
        epoch_loss = torch.stack(batch_losses).mean()

        # Combine accuracies and return mean value
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
```

```
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch: {} - Validation Loss: {:.4f}, Validation Accuracy:
{:.4f}".format( \
                                    epoch, result['val_loss'],
result['val_acc']))

    def num_flat_features(self, image):
        size = image.size()[1:]
        num_features = 1
        for s in size:
            num_features *= s

        return num_features
```

The differences from *FFNN* are located in the constructor and the *forward* method. We know upfront which layers we want to use, and we add two convolutional layers using *Conv2d* class and two fully connected layers using the *Linear* class like before. In the *forward* function, we use *max_pool2d* function to perform max pooling. Other methods are the same as for the *FFNN* implementation. We can utilize the *ModelTrainer* that we already implemented before and train this network:

```
input_size = 784
num_classes = 10

model = CNN(accuracy_function=accuracy)
print(model)
```

```
CNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=1600, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

Now we can use the *ModelTrainer* class to train this CNN:

```
model_trainer = ModelTrainer()

training_history = []
training_history += model_trainer.fit(5, 0.2, model, train_loader, val_loader)

plot_history(training_history)
```

```
Epoch: 0 - Validation Loss: 0.5739, Validation Accuracy: 0.7846
Epoch: 1 - Validation Loss: 0.3864, Validation Accuracy: 0.8598
Epoch: 2 - Validation Loss: 0.4431, Validation Accuracy: 0.8344
Epoch: 3 - Validation Loss: 0.3124, Validation Accuracy: 0.8859
Epoch: 4 - Validation Loss: 0.3125, Validation Accuracy: 0.8873
```

We got a bit better results than with feed-forward neural networks. Accuracy is 88%. We can further improve these results by adding more convolutional layers, train networks longer, and modify the learning late. Give it a try.

6.6 Deploying PyTorch Neural Networks

In this section, we cover **TorchServe**, a new way to deploy *PyTorch* models. This is still a new technology; its current version is 0.1 and it is highly experimental, but it is very promising. In essence, the *TorchServe* removes additional servers that you would otherwise have to write manually. It is similar to *TensorFlow Serve* and it makes creating modern **Deep Learning** applications much easier.

6.6.1 TorchServe Architecture

The main goal of the *TorchServe* and similar applications is to provide **API** through which other parts of the system can communicate with the **model**. It exposes three types of API:

- **API description** – retrieves a description of the API using OpenAPI 3.0 specification.

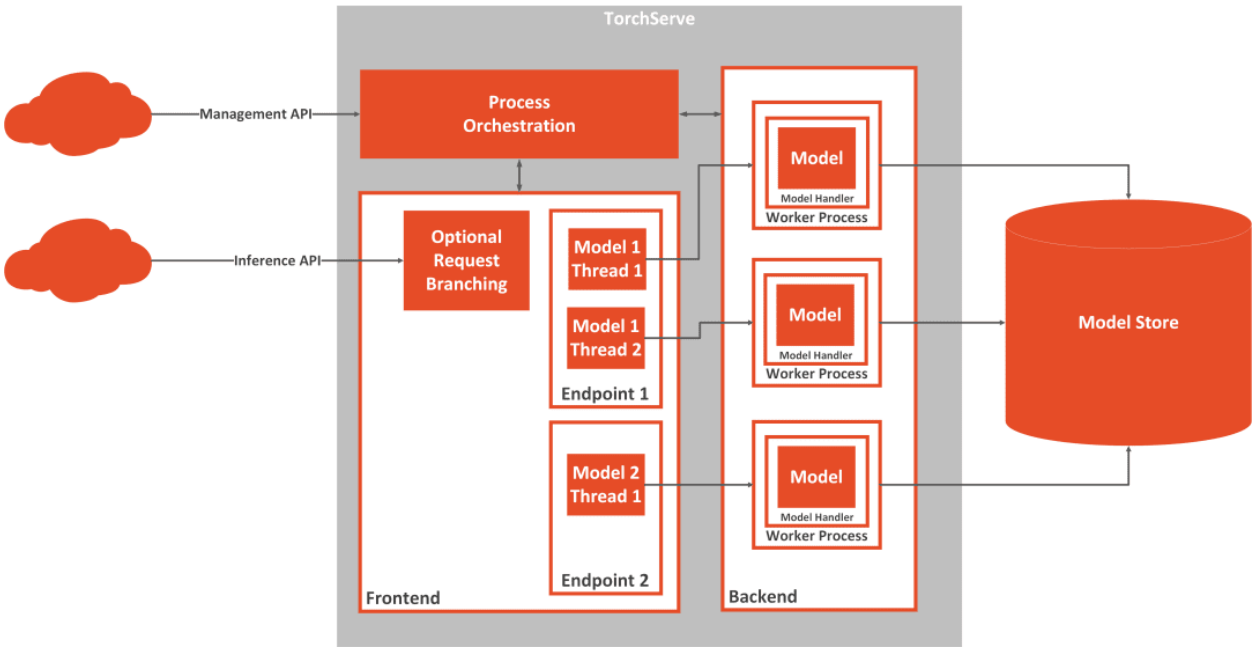
  Call example: *curl -X OPTIONS* http://localhost:8080
- **Health check API** – retrieves the status of the TorchServe.

  Call example: *curl http://localhost:8080/ping*

- **Predictions API** – make predictions API calls to the models that are served with TorchServe.

  Call example: *curl -X POST http://localhost:8080/predictions/{model_name} -T {input_data}*

In order to provide these APIs, *TorchServe* is composed of several parts. Here is what it looks like from a high-level perspective:



All API requests are routed through the so-called **Frontend**. This component of *TorchServe* is, apart from handling all requests and responses from the client, in charge of the model's life cycle. The instances of the models are hosted by **Model Workers**. These components run the actual interfaces of each model. All loadable models are stored within a directory (cloud or a local one) – **Model Store**. In a nutshell, once you run *TorchServe* (we will see how that is done in a bit), you load different models that are available in *Model Store*. Then you can start each of these models, which will create a new *Model Worker* instance that will expose the **interface** to that specific model. Then you can send API calls that are handled by the *Frontend* and routed to the correct *Model Worker*.

6.6.2 Installation

At the moment, *TorchServe* supports only *Linux* and *MacOS*. If you are a *Windows* user, you can still use *TorchServe* with *Docker*. TorchServe also requires *Java 11* SDK, so make sure that you install that first. For *Linux* run this command:

```
sudo apt-get install openjdk-11-jdk
```

For MacOS run this:

```
brew tap AdoptOpenJDK/openjdk
brew cask install adoptopenjdk11
```

Then you can install *TorchServe* with either pip:

```
pip install torch torchtext torchvision sentencepiece psutil future
pip install torchserve torch-model-archiver
```

Or with Conda:

```
conda create --name torchserve torchserve torch-model-archiver psutil \
future pytorch torchtext torchvision -c pytorch -c powerai
```

If you want to install GPU version use this:

```
conda create --name torchserve torchserve torch-model-archiver psutil \
future pytorch torchtext torchvision cudatoolkit=10.1 -c pytorch -c powerai
```

Note that these commands will install *TorchServe* and *Torch Model Archiver*.

6.6.3 Saving Trained Model

Before running and serving models with TorchServe, first you need to **save** it. Let's do that with the Feed-Forward Neural network model we created in the **previous section**.

After the training process, we can save it using the *save()* method and the model's state dictionary. When you train the model using *PyTorch*, all its weights and biases are

stored within the *parameters* attribute of *torch.nn.Module*. You can access these parameters using the parameters function *model.parameters()*. The **state dictionary** is a *Python* dictionary object that maps each layer of the model to its parameter **tensor**. You can access it using the *state_dict* attribute of the model. Note that optimizer objects from torch.optim have state_dict as well, but it contains information about used hyperparameters and optimizer state. Here is how you can use *state_dict* to save the model:

```
torch.save(model.state_dict(), os.path.join('.\models', 'ffnn.pth'))
```

Quite easy, isn't it? Your model will be located in the path you used for the second parameter. As the output, you will find the *ffnn.pth* located in the *./models* folder.

6.6.4 Serving Model

Now, we have all the necessary pieces for serving the models using *TorchServe*. Let's start the *model archiver* and add model to the *Model Store*:

```
torch-model-archiver --model-name ffnn --version 1.0 --serialized-file
./models/ffnn.pth \
--export-path ./model_store --handler image_classifier
```

With this command, we moved the *ffnn* model we created in the previous section. However, we also gave it a version and a name. Make sure that you have **created** a folder for the *Model Store* before calling this command (in our example, that is *./model_store* location). After this call, you will find *ffnn.mar* **file** in the *./model_store*. Finally, we can start *TorchServe*:

```
torchserve --start --ncs --model-store model_store --models ffnn.mar
```

Our model is available at *localhost* port *8080* and we can utilize APIs we covered previously. For example:

```
curl POST http://localhost:8080/predictions/ffnn -T data/sample_image.png
```

Also you can acquire list of available models:

```
curl http://localhost:8081/models
{
    "models": [
        {
            "modelName": "ffnn",
            "modelUrl": "ffnn.mar"
        }
    ]
}
```

If you want to stop *TorchServe*, all you need to do is call the **command**:

```
torchserve --stop
```

Another cool thing is that TorchServe exposes configuration, using which you can configure a number of worker **threads** on CPU and GPU. This can be very useful if your server is under a heavy **workload**. For example, you might want to use *number_of_gpu* which limits the number of GPUs used GPU per model.

6.6.5 TorchServe and Docker

Another option when it comes to serving *PyTorch* models with *TorchServe* is to use it in combination with **Docker**. Just like for other tools, there is *a TorchServe* image available on Docker Hub so that you can pull it from there. To start a CPU-based image, run this command:

```
docker run --rm -it -p 8080:8080 -p 8081:8081 pytorch/torchserve:latest-cpu
```

It is similar for the GPU-based image run:

```
docker run --rm -it --gpus '"device=1,2"' -p 8080:8080 -p 8081:8081
pytorch/torchserve:latest-gpu
```

However, if you want to create a *.mar* file, you need to take some additional steps. After the *Docker* container is started, acquire the name of the container:

```
docker ps
```

Connect to it's bash prompt:

```
docker exec -it <container_name> /bin/bash
```

Finally, run *Model Archiver:*

```
torch-model-archiver --model-name ffnn --version 1.0 --serialized-file
/home/modles/ffnn.pth \
--export-path /home/model-server/model-store --handler image_classifier
```

Don't forget to take care of production parameters when you are deploying *TorchServe* in Production with *Docker*. For example, you might want to use something like this:

```
docker run --rm --shm-size=2g \
        --ulimit memlock=-1 \
        --ulimit stack=67108864 \
        -p8080:8080 \
        -p8081:8081 \
        --mount type=bind,source=path_to_model_store,target=/tmp/models
<container> \
          torchserve --model-store=/tmp/models
```

This way you can set up shared memory size, user limits for system resources and expose ports, as well as avoid potential problems on your server.

7. Some Rules and Best Practices

Machine Learning became an integral part of our businesses and startups. This affects software development too; in fact, it goes even further. We can't observe machine learning components just as another part of the ecosystem because they are part of the system that makes **decisions**. These components are also shifting our focus to data, which brings a different **mindset** when it comes to the infrastructure. Because of all these things, building machine learning-based applications is **not** an easy task. There are several areas where data scientists, software developers, and DevOps engineers need to work together to make a **high-quality** product.

In this chapter, we cover some machine learning **practices** that we think will help you achieve that. These practices are divided into 5 sections. Each section is composed of several tips and tricks that may help you build **awesome** machine learning applications.

7.1 Objective and Metrics Best Practices

In this section, we consider the **business** aspects of machine learning applications. This step is arguably the most **important** one. At the beginning of every project, we need to define a **business problem** we are trying to solve. This will drive everything, from the features of your application to the infrastructure and steps when it comes to gathering the data. Here are some of the things you should pay special **attention** to during this process.

7.1.1 Start with a Business Problem Statement and Objective

As we mentioned, making a business problem statement is crucial when it comes to building machine learning applications. However, since it is not techy and exciting, a lot of people de-prioritize and overlook it. So, the advice is – spend some time on your problem, think about it and think about what you are trying to achieve. Define how the problem affects the profitability of your company. Don't just look at it from the perspective of "I want more clicks on my website"or "I want to earn more money". A well-defined problem looks something like this – "What helps me sell more ebooks?". Based on this, you should be able to define the objective. The objective is a metric that you are trying to optimize. It is of great importance to establish the right success metric because it will give you the feel of progress. Also, the objective might (and probably will) change over time as you learn more about your data.

7.1.2 Gather Historical Data From Existing Systems

Sometimes the requirements are not that clear, so you are not really able to come up with the proper objective straight away. This is often the case when working with legacy

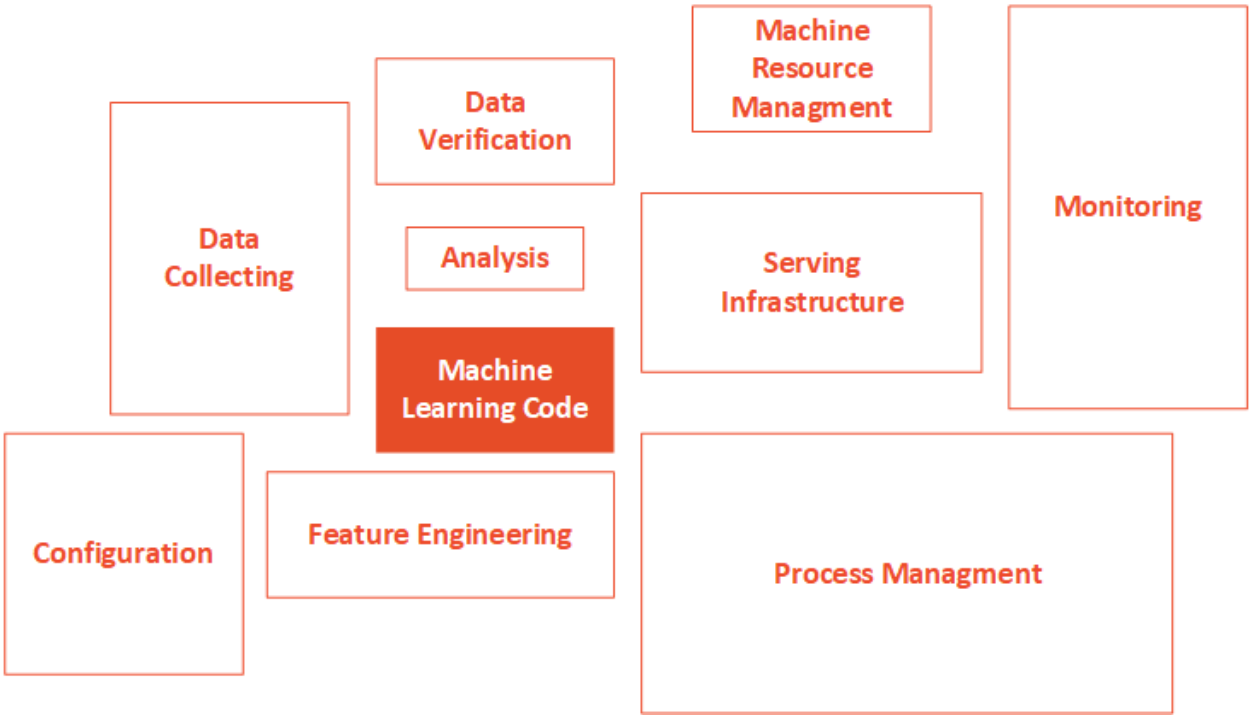
systems and introducing machine learning into them. Before you go to the nuances of what your application will do and which role machine learning plays in it, gather as much as possible from the current system. This way, historical data can help you with the task at hand. Also, this data can already indicate where the optimization is necessary and which actions will give the best result.

7.1.3 Use Simple Metric for First Objective

Making a successful machine learning project is an incremental process. In order to get to the final goal, be ready to iterate through several solutions. That is why it is important to start small. Your first objective should be a simple metric that is easily observable and attributable. For example, user behavior is the easiest feature to observe. Things like "Was the recommended item marked as spam?". You should avoid modeling indirect effects, at least in the beginning. Indirect effects can give your business enormous values; later on, however, they use complicated metrics.

7.2 Infrastructure Best Practices

Infrastructure has multiple roles when it comes to machine learning applications. One of the major tasks is to define how we gather, process, and receive new data. After that, we need to decide how we train our models and version them. Finally, deploying the model in production is a topic that we need to consider as well. In all these tasks, infrastructure plays a crucial role. In fact, chances are that you will probably spend more time working on the infrastructure of your system than on the machine learning model itself:



Here are some tips and tricks that you need to consider when building it.

7.2.1 Infrastructure is Testable without Model

Complete infrastructure should be independent of the machine learning model. In essence, you should strive to create an end-to-end solution where each aspect of the system is **self-sufficient**. The machine learning model should be **encapsulated**, so the rest of the system does not depend on it. This way, you are able to manipulate and restructure the rest of the system fairly easily if necessary. By isolating parts of the system that gather and **pre-process** the data, train model, test model, serve model, and so on, you will be able to mock and replace parts of the system with more ease. It is like practicing the **Single Responsibility Principle** on a higher level of abstraction.

7.2.2. Deploy Model only After it Passes Sanity Checks

Tests are an important **barrier** that separates you from the problems in the system. In order to provide the best experience to the users of your machine learning application, make sure that you do **tests** and **sanity checks** before **deploying** your model. This can be automated too. For example, you train your model and perform tests on the test dataset. You can check if the **metrics** you have chosen for your model are providing good results. You can do that with standard metrics like *accuracy*, *f1 score*, and *recall* as well. If the model provides satisfying results, only then will it be **deployed** to production.

7.2.3 On-Premise or Cloud

Hardware or cloud? Bare metal or someone else servers? An age-old question. The benefit of choosing a cloud is that it **saves time**, it is easier to **scale**, and it includes a low financial barrier to **entry**. You have the support of the provider too. Talking about providers, there many options out there, with the big players like *Microsoft Azure*, *AWS* and *GCP*. However, on-premise hardware is a **one-time** investment on which you can run as many experiments without **affecting** the costs. Today, there are many pre-built deep learning servers available, such as *Nvidia workstations* and *Lambda Labs*.

7.2.4 Separate Services for Model Training and Model Serving

This one is a sort of conclusion that you can make from points 4 and 5. However, it is really important, so it is good to mention it separately. In general, you should always strive to separate the training model component from the serving model component. This will give you the ability to test your infrastructure and model. Apart from that, you will have bigger control of your model in production.

7.2.5 Use Containers and Kubernetes in Deployment

Microservices architecture can help you achieve previous points. By using technologies like Docker and Kubernetes, you should be able to encapsulate separate parts of the system. This way you can make incremental improvements in each of them and replace each component if necessary. Also, scaling with Kubernetes is a painless process.

7.3 Data Best Practices

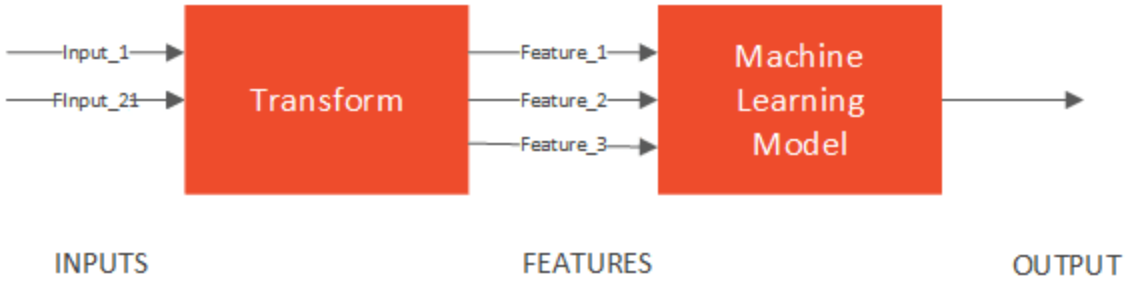
All of these "Software 2.0" solutions would not be possible without data. Data can come in many shapes and forms, and we often need to work really hard to distill information from it. In this chapter, we cover some of the best practices when it comes to data gathering and pre-processing.

7.3.2 Data Quantity

In order to make good predictions or pattern detection, you need a lot of data. That is why it is important to set the proper component in your system that will gather data for you. If you have no data, it is good to invest in some existing dataset and then improve the model over time with the data gathered from your system. Finally, sometimes you can short-circuit the initial lack of data with transfer learning. For example, if you are working on an object detection app, you can use YOLO.

7.3.1 Data Quality and Transformations

Real-world data is messy. Sometimes it is incomplete and sparse; other times, it is noisy or inconsistent. In order to make it better, it is necessary to invest in data pre-processing and feature engineering. If you properly encapsulated it, you can get the data from the data gathering component and apply necessary transformations (like imputation, scaling, etc.) in the transformation component. This component has a twofold purpose. It prepares the training data and uses the same transformations on the new data samples that come into your system. In essence, it creates features that are extracted from the raw inputs.



7.3.2 Document each Feature and Assign Owner

Machine Learning Systems can become large, and datasets can have many features. Also, features can sometimes be created from other features. It is good to assign each feature to one team member. This team member will know why a certain transformation has been applied and what this feature represents. Another good approach is to create a document with a detailed description of each feature.

7.3.3 Plan to Launch and Iterate

Don't be afraid to get into it and get better over time. Your features and models will

change over time, so it is important to have this in mind. Also, it might happen that the UI of your application is changed and you are now able to get more data from the user behavior. In general, it is good to keep an open mind about this and be ready to start small and improve over iterations.

7.4 Model Best Practices

It may seem that even though machine learning applications revolve around the power of machine learning models, they are usually neatly tucked behind large infrastructure components. This is true to a certain degree, but there is a good reason for it. In order to actually utilize that power, those other components are necessary as well, but they are useless without a good machine learning model to put it all together. Here are some tips and tricks that you should keep in mind while working with machine learning and deep learning models.

7.4.1 Starting with an Interpretable Model

Keep the first model simple and get the infrastructure right. This model is often called the baseline. Don't start with complicated neural network architectures right away. Maybe try to solve a problem with a simple Decision Tree first. There are multiple reasons for this. The first one is that building the complete system takes time. Getting other components and data right at the beginning will give you the ability to extend your experiment later. The second reason is that businesses will understand what is happening in the interpretable model, which can give you more confidence and trust to continue with fancier models.

7.4.2. What to do if the model does not work?

There are several things you can do if your machine learning model is not working. First, check your objective. Maybe the model is working well, but you are using the wrong metric. Also, check your baseline. What was the most informative feature there? What is the mean value of the target?

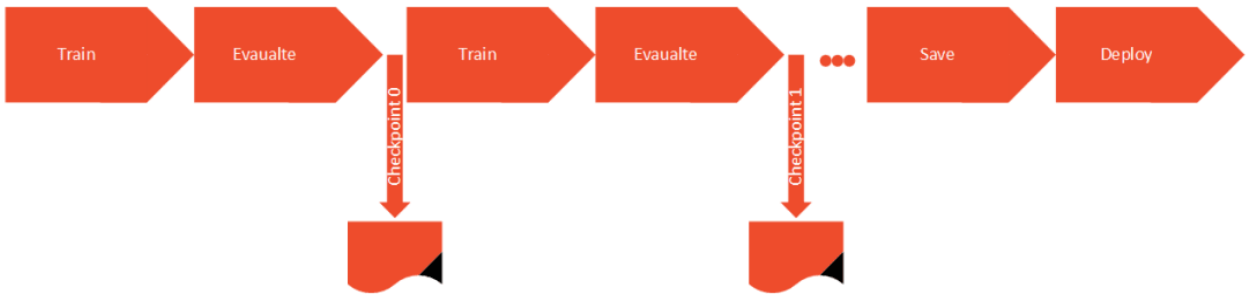
Another thing you can do is go back to the data and do exploratory data analysis again. Check whether the features are informative and if there are some strange values. Look for outliers and correlations between features and the target. Also, use the explainability method to understand the contributions of each feature.

You can use a few examples that the model got wrong and try to figure out what the problem is. If you are creating a deep learning model, try to remove regularization, dropout and data augmentation, increase learning rate and overfit the data. Try to bring the loss to 0.

7.4.2 Use Checkpoints

Probably the best advice that one can give you when working with machine learning models is that you should use checkpoints. A checkpoint is an intermediate dump of a

model's internal state (parameters and hyperparameters). Using these machine learning frameworks, you can resume the training from this point at any given time. This will give you the ability to incrementally train the model and make a good trade when it comes to performance vs. training time. Also, this way, you are more resilient to hardware or cloud failures.



7.4.3 Performance over fancy metric

A lot of times, data scientists can lose themselves in various metrics. This may lead to trying actions that may improve various vanity metrics but lower the performance of the system as a whole. This is, of course, a bad approach and the performance of the complete system should always be in the first place. Thus, if there is some change that improves log loss but degrades the performance of the system, look for another feature. If this starts happening often, it is time to rethink the objective of the model.

7.4.4 Production Data to Training Data

The best way to improve your model over time is to use the data used during serving time for the next training iteration. This way you will move your model to the real-world scenario and improve the correctness of your predictions. The best way to do this is to automate this, i.e., store every new sample that comes from the serving model and then use it for training.

7.5 Code Best Practices

All that math, planning, and design need to be coded. It is the piece that holds it all together. It is important to focus on your code if you want to make a long-lasting solution. In this chapter, we share several tips and tricks that you should pay attention to when it comes to the code of your project.

7.5.1 Write Clean Code

Learn how to write code properly. Name your variables and functions like a grown-up, add comments, and pay attention to the structure. You can choose to use object-oriented programming or functional programming and write a lot of tests. Even if you are working alone on the project, make sure you nail this because sooner or later, you will work in a team. Clean code helps all members of the team be in sync and on the same page. Never forget that the team is larger than the individual, and clean code is one of the tools for building a great team.

7.5.2 Write a lot of Tests

Automate as many tests as possible. These are guards of continuous progress. There are several levels of tests that one can write when it comes to building one machine learning application. In general, you should write a lot of unit tests to verify the functionalities of each component of the system. For this, you can use the Test Driven Development approach. Integration tests are good for testing how components work with each other. Finally, system tests are there to test your solution end-to-end. Additional tests for the model are also part of this. Don't save your model if it is not passing the sanity checks or put them in production. Performance tests can help you with this.

Final Words

Well, we came to the end of this book. We tried to cover quite a large area with it. The journey through the world of machine learning is not easy, and it has many hidden paths and additional quests. It seems that the biggest problem for beginners is to keep a bigger picture in mind and simultaneously be focused on learning about the specifics of the complete process. We tried to get into as many parts as possible and present a complete system but explore every section independently. Of course, we haven't covered everything. Every topic that we have presented has many layers to it. Every topic can be further explored from a mathematical or technical perspective. We encourage you to do so, to find better explanations for all these concepts and those more suitable for you. Hopefully, this book is a good starting point and a little bit more than just that. Apart from that, don't forget to use all the knowledge for practical purposes. The best way to learn about all these things is putting them into practice, i.e., performing them. So head out to *Kaggle*, download some datasets, and try to apply the principles presented in this book. Then invent your own principles and go further. Don't forget that you can always find out more.

Keep learning machine learning!