



Università Politecnica delle Marche

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Informatica e dell'Automazione

Machine Learning and Deep Learning techniques

Tecniche di Machine Learning e Deep Learning

Candidato:

Guido Laudenzi

Matricola 1079884

Relatore:

Prof. Simone Fiori

Correlatore:

Dr. Shotaro Akaho

Anno Accademico 2019–2020

Index

Abstract	4
1 Machine Learning	6
1.1 Supervised machine learning algorithms	6
1.1.1 Regression	7
1.1.2 Classification	8
1.1.3 Real world applications	8
1.2 Un-supervised machine learning algorithms	8
1.2.1 Dimension reduction	9
1.2.2 Clustering	9
1.2.3 Pattern Mining	9
1.3 Semi-supervised machine learning algorithms	9
1.4 Reinforcement learning algorithms	10
2 Supervised Machine Learning algorithms	11
2.1 Linear Regression	13
2.2 Polynomial Regression	16
2.3 Logistic Regression	19
2.3.1 Confusion Matrix	20
2.4 Naïve Bayes	23
2.4.1 Bayes' theorem	23
2.5 Decision Tree	26
2.6 Random Forest	29
2.7 Support Vector Machines	31
3 Deep Learning	35
3.1 Deep Feed-forward Networks	39
3.1.1 Single-layer Perceptron	40
3.1.2 Multi-layer Perceptron	40
3.2 Convolutional Neural Networks	42
3.2.1 Convolution Operation	42

3.2.2	Pooling	44
3.3	Recurrent Neural Networks	45
3.3.1	Vanishing and exploding gradient problem	45
3.3.2	RNN' extensions	46
4	Code	48
4.1	Linear and Polynomial Regression	48
4.2	Decision Tree	51
4.3	Random Forest	56
4.4	Logistic Regression and SVC	60
4.5	Deep Learning	70
4.5.1	Multilayer Perceptron	70
4.5.2	Convolutional Neural Network	74
4.5.3	Recurrent Neural Network	77
4.5.4	Transfer Learning with pre-trained model	78
	Conclusion	85
	References	87

Abstract

English Version

The purpose of this report is to explain how basic Machine Learning and Deep Learning works with everything that is involved with it: why it is so popular and why it gets even more importance in the programming world day by day, using some real life examples.

First of all, I must explain what is the real meaning of Machine Learning and Deep Learning. I will explain, for each one, some useful algorithms for problems solving.

At the end, I will show the techniques with the help of the programming language Python. In the codes I will be using real datasets (CSV files filled with data) taken by the Kaggle website, an online community of data scientists and machine learners, owned by Google LLC.

Python is an interpreted, high-level, general-purpose programming language. It can be used both OOP (Object-oriented programming) and imperative programming. It offers Scikit-learn, which is a free software machine learning library for the Python programming language.

Regarding Deep Learning, other frameworks are necessary and Tensorflow and Keras are the commonly used. For improved performance, I could take advantage of the remote use of a GPU machine, a processor designed to handle graphics operations, which tends to compute Deep Learning algorithms faster because of its high power.

Abstract

Versione Italiana

Lo scopo di questa tesi è quello di spiegare come funzionano le basi del Machine Learning (in italiano, Apprendimento Automatico), del Deep Learning e tutto ciò che ne deriva: come mai è così popolare e come mai acquista sempre maggiore importanza nel mondo della programmazione giorno dopo giorno, tramite l'utilizzo di alcuni esempi comuni applicabili alla vita reale.

Prima di tutto, devo iniziare a spiegare qual'è il vero significato dietro il Machine Learning e il Deep Learning. Per ognuno spiegherò alcuni algoritmi utili per la risoluzione di problemi (Problem Solving).

Alla fine, sono riportate tutte le tecniche riportate tramite l'utilizzo del linguaggio di programmazione Python. Nel codice si fa utilizzo di dati reali (dei file CSV riempiti di dati) presi dal sito internet di Kaggle, una comunità online di 'data scientists' e 'machine learners' (ovvero praticanti della materia qui trattata), di proprietà di Google LLC.

Python è un linguaggio interpretato, di alto livello e con fini dediti alla programmazione generale. Può essere usato sia come OOP (programmazione orientata agli oggetti) oppure come linguaggio imperativo. Inoltre offre Scikit-learn, una libreria software gratis utilizzabile nel linguaggio di programmazione Python.

Per quanto riguarda il Deep Learning invece, sono necessari altri framework e Tensorflow e Keras sono quelli più utilizzati. Per migliorare i risultati dello studio, ho potuto avere il vantaggio di utilizzare da remoto una macchina GPU, ovvero un processore efficiente nella manipolazione di operazioni grafiche, il quale tende a compilare gli algoritmi di Deep Learning più velocemente grazie alla sua enorme potenza.

Chapter 1

Machine Learning

In general, Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use them learn by themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

Machine learning algorithms are often categorized as supervised or unsupervised, but more generally there are 4 categories (see the taxonomy of Figure 1.1):

- **Supervised machine learning algorithms**
- **Un-supervised machine learning algorithms**
- **Semi-supervised machine learning algorithms**
- **Reinforcement learning algorithms**

1.1 Supervised machine learning algorithms

They can apply what has been learned in the past to new data using labeled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces a function to make predictions about the

output values. The system is able to provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct output and find errors in order to modify the model accordingly.

During the internship, I was mainly focused on this kind of algorithms. In particular, Supervised Machine Learning algorithms are divided in two problems: classification and regression. Both are categorized under the same umbrella of supervised machine learning and both share the same concept of utilizing known datasets (referred to as training datasets) to make predictions. The main difference between them is that the output variable in regression is numerical (or continuous) while that for classification is categorical (or discrete).

In supervised learning, an algorithm is employed to learn the mapping function from the input variable x to the output variable y ; that is $y = f(X)$. The objective of such a problem is to approximate the mapping function f as accurately as possible such that whenever there is a new input data x , the output variable y for the dataset can be predicted. [1]

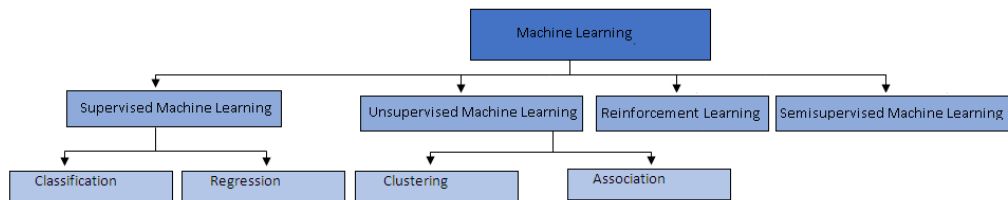


Figure 1.1: Machine Learning taxonomy

1.1.1 Regression

Regression algorithms attempt to estimate the mapping function f from the input variables x to numerical or continuous output variables y . In this case, y is a real value, which can be an integer or a floating point value. Therefore, regression prediction problems are usually quantities or sizes.

Examples of the common regression algorithms include Linear Regression, Polynomial Regression and Kernel Regression.

Attention! Some algorithms, such as logistic regression, have the name “regression” in their names but they are not regression algorithms.

1.1.2 Classification

On the other hand, classification algorithms attempt to estimate the mapping function f from the input variables x to discrete or categorical output variables y . In this case, y is a category that the mapping function predicts. If provided with a single or several input variables, a classification model will attempt to predict the value of a single or several conclusions. [1]

Examples of the common classification algorithms include Logistic Regression, Naïve Bayes, Decision Trees and SVC.

1.1.3 Real world applications

Supervised Machine Learning applications are lots. Everything that can be recognized, can be studied with Machine Learning. Some important example can be: [2]

- **Image recognition:** one of the most common uses of machine learning is image recognition. There are many situations where you can classify the object as a digital image. For digital images, the measurements describe the outputs of each pixel in the image. In the case of a black and white image, the intensity of each pixel serves as one measurement; in the colored image, each pixel considered as providing 3 measurements to the intensities of 3 main color components **RGB**. Examples: face detection or character recognition (Google Lens).
- **Medical Diagnosis:** it is being used for the analysis of the importance of clinical parameters and of their combinations for prognosis, ex. prediction of disease progression, for the extraction of medical knowledge for outcomes research, for therapy planning and support, and for overall patient management. The measurements in this application are typically the results of certain medical tests, medical images or basic physical information.
- **Statistical Arbitrage:** in finance, statistical arbitrage refers to automated trading strategies that are typical of a short-term and involve a large number of securities. In such strategies, the user tries to implement a trading algorithm for a set of securities on the basis of quantities such as historical correlations and general economic variables.

1.2 Un-supervised machine learning algorithms

These algorithms are used when the information used to train is neither classified nor labeled. This means that Unsupervised Machine Learning can not be directly

applied to a regression because it is unknown what the output values could be, therefore it is impossible to train the algorithm how you normally would.[3]

Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data. The system does not figure out the right output, but it explores the data and can find inferences from datasets to describe hidden structures from unlabeled data.

1.2.1 Dimension reduction

Dimensionality is the number of variables present in the dataset. These dimensions are represented as columns, and the goal is to reduce the number of them.

In most cases, those columns are correlated and, therefore, there is some information that is redundant which increases the dataset's noise. This redundant information impacts negatively in Machine Learning model's training and performance and that is why using dimensionality reduction methods becomes important.

There are two main categories of dimensionality reduction:

- **Feature Selection:** we select a subset of features of the original dataset.
- **Feature Extraction:** we derive information from the original set to build a new feature subspace.

1.2.2 Clustering

Clustering is the task of dividing the data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them. [4]

1.2.3 Pattern Mining

Pattern Mining is a rule-based method for discovering interesting relations between variables in large databases. It is intended to identify strong rules discovered in databases using some measures of interestingness.

1.3 Semi-supervised machine learning algorithms

They fall in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training. The systems that use this method are able to improve learning accuracy. In fact, the basic procedure involved is that at first, the programmer will cluster similar data using an unsupervised learning algorithm,

and then use the existing labeled data to label the rest of the unlabeled data. The acquisition of labeled data for a learning problem often requires a skilled human agent or a physical experiment. The cost associated with the labeling process may render a fully labeled training set impossible, while the acquisition of unlabeled data is relatively easy.

1.4 Reinforcement learning algorithms

It is a learning method that interacts with its environment by producing actions and discovers errors or gets rewards. Trial and error search and delayed reward are the most important characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is the best: this is known as the reinforcement signal.

In comparison, Reinforcement algorithms differ by the Supervised ones for the continue generation of its outputs, in order to improve the following one. Instead, Supervised ones are already trained with test outputs, that are independent from each others. Supervised algorithms usually get explicit information of his output, while Reinforcement can get outputs only by rewards. So, if the target action is given to the Reinforcement learning setting, it becomes a Supervised Learning program.

Chapter 2

Supervised Machine Learning algorithms

In this chapter, I will explain various Supervised algorithms that are useful in real life problems to predict the output of the test set. Each algorithm has a different area of application, so the programmer should usually do an accurate analysis of the problem and then decide which one fits the most for the actual program.

The practitioner has some guidelines that helps him to understand which is the appropriate algorithm to use:

1. Collect the data, so the particular dataset of the problem he wants to solve.
2. Check for anomalies, missing data and then clean the data; this is also known as **preprocessing**. It is important because without this passage, the algorithm might not work in the correct way.
3. Perform statistical analysis and initial visualization with the plot of each dataset's attribute. It gives an initial idea of the predictive outputs.
4. Build different mathematical model using each algorithm.
5. Check the accuracy of each model and then choose the best one.
6. At the end, present the results using informative plots of the model chosen.

Everything is performed by a method called **validation**: typically, the dataset is splitted in two different parts in order to train the model and then to state the accuracy during the construction of the model. The train set is the part of the dataset used to build the predictive function. The test set is the second part that we want to use to verify the solution and the precision of the algorithm. The accuracy of

every algorithm is calculated comparing the predicted output with the test output. Validation is important for generalization (the ability of an algorithm to be effective across a range of inputs) and to avoid over-fitting (a model that models the training data too well) other than the model selection.

Another selection method is the **information criteria**: it is an estimator for out-of-sample deviance and the relative quality of statistical models for a given dataset.

Using the Scikit-learn library, the datasets' splitting is really easy because you just need to call a very simple function: basically, it divides the entire dataset in 4 different matrices that are input train, output train, input test, output test.

After the division, all you need to do is just fitting the program with the train data, and then predicting the output. For each algorithm, there are different ways to predict the result.

2.1 Linear Regression

Linear regression is a basic and commonly used type of predictive analysis which usually works on continuous data, as the name says. We will try to understand linear regression based on an example[5]:

Living area (feet ²)	Price (1000\$)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

Figure 2.1: House Cost

Guido is trying to buy a house and is collecting housing data so that he can estimate the “cost” of the house according to the “Living area” of the house in feet. He observes the data and comes to the conclusion that the data is linear after he sketches the scatter plot. For his first scatter plot, Guido uses two variables: ‘Living area’ and ‘Price’.

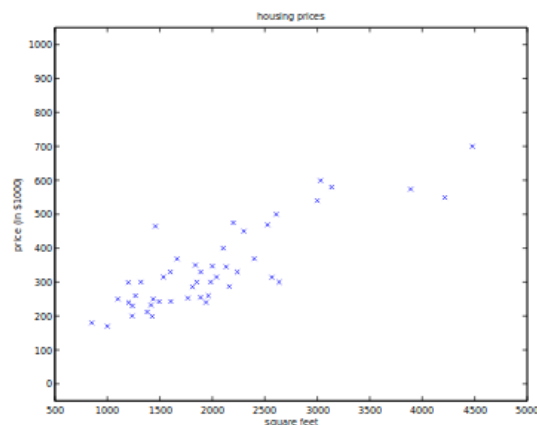


Figure 2.2: Price and feet scatter

As soon as he saw a pattern in the data, he planned to make a **regression line** on the graph so that he can use the line to predict the ‘price of the house’.

Using the training data (in this case ‘Price’ and ‘Living area’), a regression line is obtained which will give the minimum error. To do that, he needs to make a line that is closest to as many points as possible.

This ‘linear equation’ is then used for any new data so that he is able to predict the required output:

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i$$

Here, the β_1 is/are the parameters (called weights), β_0 is the y-intercept (one of the parameters) and ϵ_i is the random error term whose role is to add bias.

The equation above is a simple ‘equation of a line’ that is

$$Y_{predicted} = (\beta_1 * x + \beta_0) + Error$$

The values of the weights and of the y-intercept must be chosen so that they minimize the error. For example, to check the error we have to calculate the sum of squared error and tune the parameters to try to reduce the error.

$$Error = \sum (actualoutput - predictedoutput)^2$$

or

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Where: $Y_{predicted}$ is also called the hypothesis function; $J(\theta)$ is the cost function (error function), our main goal is to minimize the value of the cost; $y^{(i)}$ is the predicted output; $h_{\theta}(x^{(i)})$ is basically the $Y_{predicted}$ value.

How do we reduce the error value? This can be done by using **Gradient Descent**[5]. The main goal of Gradient descent is to minimize the cost value. That is: $minJ(\theta_0, \theta_1)$.

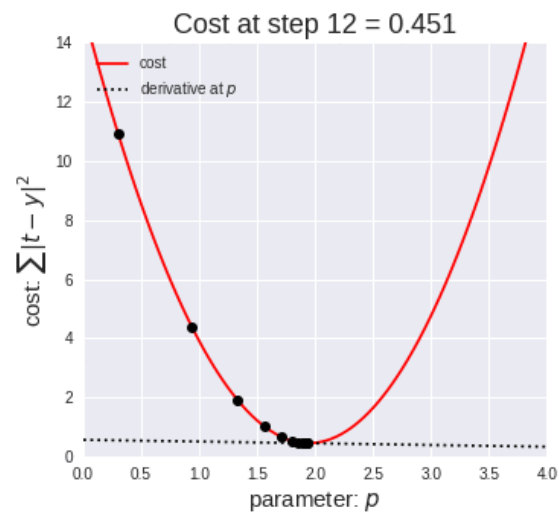


Figure 2.3: Gradient Descent

Gradient Descent is the most used method for error reduction used in Machine Learning and Deep Learning algorithms. Nowadays, the many others techniques that were discovered are all originated by the Gradient Descent.

2.2 Polynomial Regression

In the Linear Regression, we saw two variables in the data set were correlated. But what happens if we know that our data is correlated, but the relationship does not look linear? So depending on what the data looks like, we can do a polynomial regression on the data to fit a polynomial equation to it.

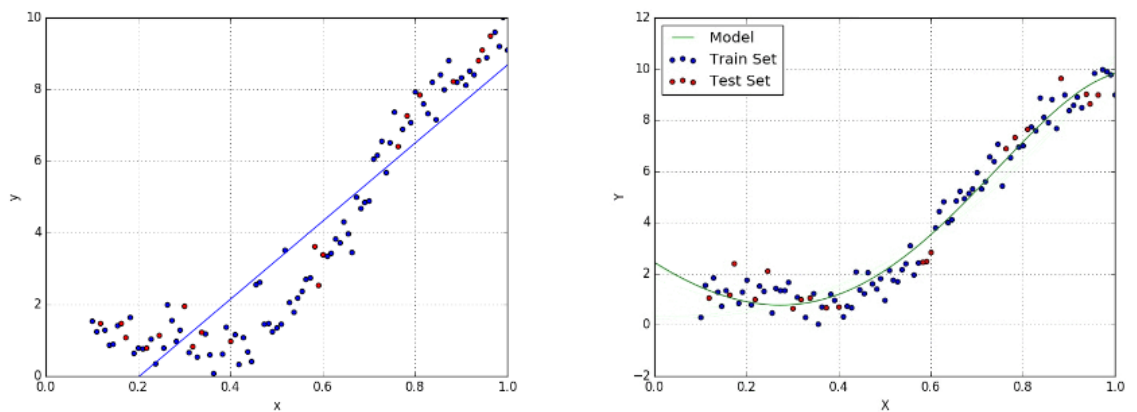


Figure 2.4: Left: Linear Regression, Right: Polynomial Regression

The general equation of the polynomial regression is: [5]

$$Y = \theta_0 + \theta_1 X + \theta_2 X^2 + \dots + \theta_m X^m + \text{Error}$$

The main difference between the linear and the polynomial regression is that you can choose the degree of your curve.

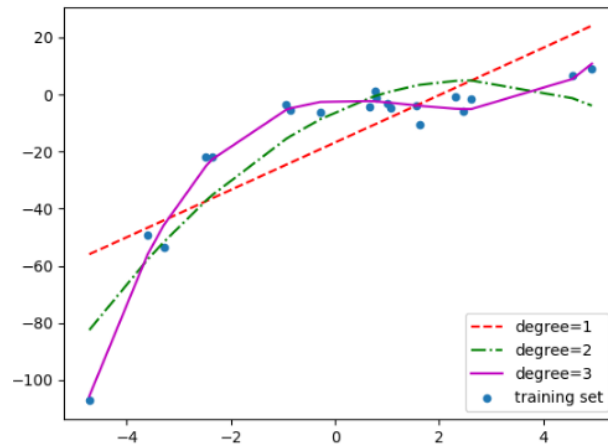


Figure 2.5: Example of degree by 1 (Linear) to 3 (Cubic)

For $degree = 20$, the model is also capturing the noise in the data. This is an example of over-fitting. Even though this model passes through most of the data, it will fail to generalize on unseen data.

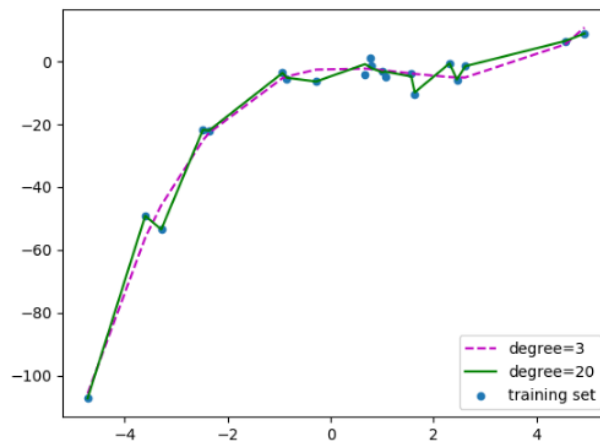


Figure 2.6: Overfitting with degree 20

To prevent over-fitting, we can add more training samples so that the algorithm does not learn the noise in the system and can become more generalized. [6]
This is how we can prevent over-fitting:

- **Bias** refers to the error due to the model's simplistic assumptions in fitting the data. A high bias means that the model is unable to capture the patterns in the data and this results in under-fitting.

- **Variance** refers to the error due to the complex model trying to fit the data. High variance means the model passes through most of the data points and it results in over-fitting the data.

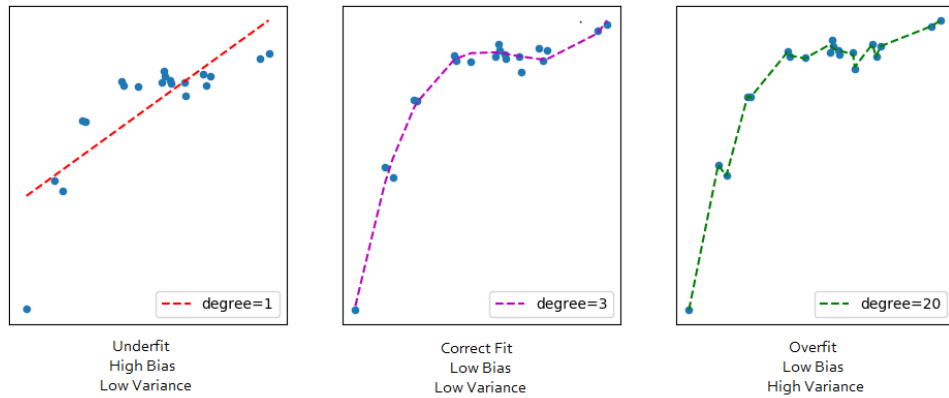


Figure 2.7: Bias and Variance

From the below picture we can observe that as the model complexity increases, the bias decreases and the variance increases and vice-versa. Ideally, a machine learning model should have low variance and low bias. But practically it's impossible to have both. Therefore to achieve a good model that performs well both on the train and unseen data, a trade-off is made. [6]

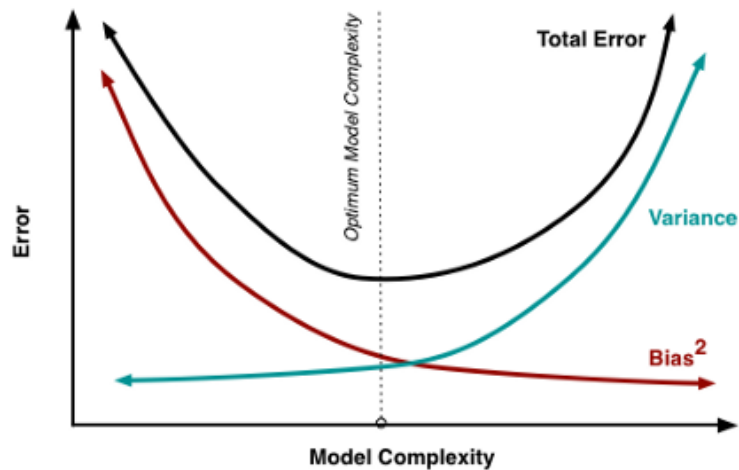


Figure 2.8: Trade-off between Bias and Variance

2.3 Logistic Regression

In the above sections, I have explained about Linear and Polynomial Regression: they are examples of regression, which means that the output is a continuous variable.

Despite of what you are maybe thinking, Logistic Regression is not for regression, but for classification, because its output can be only a binary value.

Logistic regression is named for the function used at the core of the method, the logistic function. The logistic function, also called the **sigmoid** function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1. [?]

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

Unlike the linear, we have sigmoid output: if Z goes to infinity, Y_{pred} will become 1 and if Z goes to negative infinity, Y_{pred} will become 0.

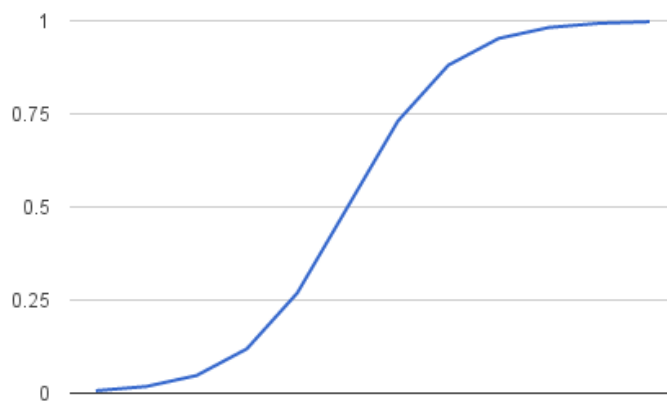


Figure 2.9: Logistic Function

The output from the hypothesis is the estimated probability. This is used to infer how confident can predicted value be actual value when given an input X .

$$P = \frac{1}{1 - e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n)}} = \frac{1}{1 - e^{-(\beta_0 + \sum \beta_i X_i)}}$$

There are different types of Logistic Regression: [7]

- **Binary:** only two outputs exist, like 0 and 1.
- **Multinomial:** there are more than 2 outputs, like 0, 1 and 2 or veg, vegan and non-veg.
- **Ordinal:** there are more categories but they are ordered, like the movie rating by 1 to 5 stars.

We expect our classifier to give us a set of outputs based on probability when we pass the inputs through a prediction function and returns a probability score between 0 and 1. To predict which class a data belongs, a threshold can be set. Based upon this threshold, the obtained estimated probability is classified into classes.

Example: if we have chosen a threshold of 0.5 and if the prediction function returned a value of 0.7, then we would classify this observation as 1. If our prediction returned a value of 0.2 then we would classify the observation as 0.

2.3.1 Confusion Matrix

The Confusion Matrix is a matrix that can be easily built in Python. It tells us the accuracy of every classification model where we use it.

	Predict: 1	Predict: 0
Actual: 1	27	27
Actual: 0	12	88

Figure 2.10: Example of Confusion Matrix

Let's try with the example of a medical disease: diabetes diagnosis. Here is how we can explain the above confusion matrix:

		Predictions	
		Predicted to have diabetes.	Predicted to not have diabetes.
Truth	Does not have Diabetes.	27 people were classified as having diabetes (1) correctly.	27 people were classified to not have diabetes although they do.
	Has Diabetes	12 people were classified to having diabetes without having it.	88 people were classified as not having diabetes (0) correctly.

Figure 2.11: Explanation of Confusion Matrix

As we can see, the main diagonal tells us about how many we predicted correctly about having or not having diabetes. Instead, in general, the off-diagonal tells us about the mistakes that we made in the predicting values. Here are the names of every cell:

		Predictions	
		POSITIVE CLASS	NEGATIVE CLASS
Truth	POSITIVE CLASS	TRUE POSITIVE (TP)	FALSE NEGATIVE (FN)
	NEGATIVE CLASS	FALSE POSITIVE (FP)	TRUE NEGATIVE (TN)

Figure 2.12: Example of Confusion Matrix

27 people were classified to not have diabetes although they do. Even if the model has a really high accuracy, we have to minimize the false negative (FN) value because it is more important to save people from diabetes than having them retrying for further diabetes tests, as the false positive (FP) people will do. In order to succeed, we have to change the Logistic Regression's threshold: the best one will return zero on the false negative cell. After that we can use the **Negative Predictive Value (NPV)** that is a function that finds the new accuracy of the model.

$$NPV = \frac{FalseNegative}{FalseNegative + TrueNegative}$$

There are many other indices that can be found using Confusion Matrix:

- **True Positive Rate (TPR)** or **Sensibility**, **Recall** is the proportion of people who took the test with a condition and were correctly labeled as having the

condition, diabetes in this case.

$$T P R = \frac{TruePositive}{TruePositive + FalseNegative}$$

- **False Positive Rate (FPR)** describes the rate of false alarms. It is the number of people incorrectly thought to have diabetes although they do not.

$$F P R = \frac{FalsePositive}{FalsePositive + TrueNegative}$$

- **Positive Predictive Value (PPV)** or **Precision** expresses the proportion of the data points our model says was relevant actually were relevant.

$$P P V = \frac{TruePositive}{TruePositive + FalsePositive}$$

The **ROC** is a plot of the FPR (false alarms) in the X axis and TPR (finding everyone with the condition who really has it) in the Y axis. Without context, it is a tool to measure classifier performance.

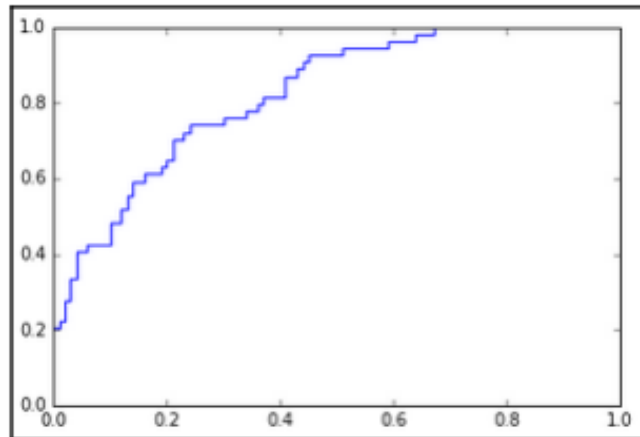


Figure 2.13: ROC plotting

If it is a perfect classifier, ROC gets almost like a rotated L shape. Otherwise, it is an imperfect classifier: ROC is a straight line with a slope of about 1. To measure classifier performance, we can also calculate the **AUC (Area under ROC curve)**. The area of the L-shaped perfect classifier is $1 \times 1 = 1$. The area of the bad classifier is usually around 0.5. [8]

2.4 Naïve Bayes

As you might have guessed watching the title, this requires us to view things from a probabilistic point of view. This is theoretically very important, though its applications are very few.

Using this algorithm, we will be dealing with the probability distributions of the variables in the dataset and predicting the probability of the response variable belonging to a particular value, given the the attributes of a new instance. [9]

This lets us examine the probability of an event based on the prior knowledge of any event that related to the former event. For example, the probability that price of a house is high, can be better assessed if we know the facilities around it, compared to the assessment made without the knowledge of the location of the house.

2.4.1 Bayes' theorem

Bayes' theorem does exactly that.

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

Where A and B are events and:

- **P(A|B)** is the conditional probability that event A occurs, given that B has occurred; also known as the posterior probability.
- **P(A) and P(B)** are the probabilities of A and B without regard of each other.
- **P(B|A)** is the conditional probability that event B occurs, given that A has occurred.

Take a simple machine learning problem, where we need to learn our model from a given set of attributes and then form a relation to a response variable. Then we use this relation to predict a response, given attributes of a new instance. Using the Bayes' theorem, its possible to build a learner that predicts the probability of the response variable belonging to some class, given a new set of attributes.

Consider the previous equation again and now assume that A is the response variable and B is the input attribute. So we have:

- **P(A|B)** is the conditional probability of response variable belonging to a particular value, given the input attributes; also known as the posterior probability.

- $P(A)$ is the prior probability of the response variable.
- $P(B)$ is the probability of training data or simply the evidence.
- $P(B|A)$ is known as the likelihood of the training data.

So:

$$Posterior = \frac{Likelihood * Prior}{Evidence}$$

Now consider a problem where the number of attributes is equal to N and the response is a boolean value, so it can be in one of the two classes. Also, the attributes are categorical (2 categories). Now, to train the classifier, we will need to calculate $P(B|A)$, for all the values in the instance and response space. This means, we will need to calculate $2(2^N - 1)$ parameters for learning this model. This is unrealistic. [9]

The complexity of the Bayesian classifier needs to be reduced, for it to be practical. The naive Bayes algorithm does that by making an assumption of conditional independence (over the training dataset).

The assumption of conditional independence states that, given random variables X , Y and Z , X is conditionally independent of Y given Z , if and only if the probability distribution governing X is independent of the value of Y given Z . Given, N different attribute values, the likelihood now can be written as

$$P(X_1 \dots X_n | Y) = \prod_{i=1}^n P(X_i | Y)$$

X represents the attributes or features, Y is the response variable. Now, $P(X|Y)$ becomes equal to the products of probability distribution of each attribute X given Y .

So, we must say that this algorithm is considered a **Generative Model**: it explicitly models the joint probability distribution $P(B, A)$ and then uses the Bayes rule to compute $P(A|B)$. The opposite of a generative model is the **Discriminative Model** (e.g. Logistic Regression) that directly models $P(A|B)$ (e.g. using the Sigmoid function).

Methods for finding parameters: [9]

- **Maximizing a Posteriori**: we are interested in finding the posterior probability, or $P(Y|X)$. Now, for multiple values of Y , we will need to calculate this expression for each of them.

Given a new instance X_n , we need to calculate the probability that Y will take on any given value, given the observed attribute values of X_n and given the distributions $P(Y)$ and $P(X|Y)$ estimated from the training data. So we will predict the class of the response variable simply taking the most probable or maximum of $P(Y|X)$ values.

- **Maximizing Likelihood:** if we assume that the response variable is uniformly distributed, then we can further simplify the algorithm. With this assumption the prior, or $P(Y)$, becomes a constant value, which is 1 between the categories of the response.
Now, the prior and evidence are independent of the response variable and they can be removed from the equation. Therefore, the maximizing the posteriori is reduced to maximizing the likelihood problem.

Some types of Naïve Bayes classifiers:

- **Multinomial Naïve Bayes:** mostly used for document classification problem, such as a document belonging to the category of sports, politics, technology etc. The features used by the classifier are the frequency of the words in the document.
- **Bernoulli Naïve Bayes:** similar to the multinomial one but the features are boolean variables. So, the parameters that we use to predict the class variable take up only values yes or no.
- **Gaussian Naïve Bayes:** the features take up a continuous value and are not discrete, so we assume that these values are sampled from a Gaussian distribution. Since the way the values are present in the dataset changes, the formula for conditional probability changes to:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

2.5 Decision Tree

A tree has many analogies in real life, and turns out that it has influenced a wide area of machine learning, covering both classification and regression.

In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions. [10]

A decision tree is drawn upside down with its root at the top.

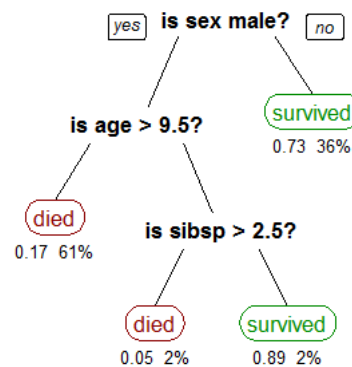


Figure 2.14: Tree example

Just like the figure, a decision tree has internal nodes which are based on some condition. The answer to this condition makes the tree splits into branches. The end of the branch that doesn't split anymore is the leaf: in this case, whether the passenger died or survived.

Although, a real dataset will have a lot more features and this will just be a branch in a much bigger tree. The feature importance is clear and relations can be viewed easily. This methodology is more commonly known as **learning decision tree** and there are two kinds:

- **Classification tree:** the target is to classify results as a discrete number (two or any number of classes), like the example above (Figure 2.14).
- **Regression trees:** represented in the same manner, just they predict continuous values like price of a house.

Growing a tree involves deciding on which features to choose and what conditions to use for splitting, even with knowing when to stop.

The most common technique for splitting is **Recursive Binary Splitting**. We can split the tree in function of the attributes we have. For example: in Figure 2.14, the sex of the person is an attribute used for splitting.

Now we will calculate how much accuracy each split will cost us, using a function. The split that costs least is chosen. This algorithm is recursive as the groups formed can be sub-divided using same strategy.

This algorithm is also known as the **greedy algorithm**, as we want to lower the cost. This makes the root node as best predictor/classifier. [10]

Cost of a split:

- **Classification:**

$$G = \sum_n P_k(1 - P_k)$$

G means Gini, P_k is proportion of same class inputs present in a particular group and the sum is calculated for each n split of the case. A Gini score gives an idea of how good a split is by how mixed the response classes are in the groups created by the split.

Example: a perfect class purity occurs when a group contains all inputs from the same class, in which case P_k is either 1 or 0 and $G = 0$; if a node is having a 50/50 split of classes in a group, it has the worst purity (for a binary classification $P_k = 0.5$ and $G = 0.5$).

Other than Gini function, another option is to measure **entropy** to determine how to split the tree. Entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any conclusions from that information.

$$E = - \sum_n p(x) \log_2 p(x)$$

Where $p(x)$ is a fraction of examples in a given class.

- **Regression:**

$$L = \sum (y - y_{pred})^2$$

L means Loss function. Here, the mean of responses of the training data inputs of particular group is considered as prediction for that group. The above function is applied to all data points and cost is calculated for all candidate splits. Split with lowest cost is chosen.

Once we decided how to split the tree, it remains for us to decide when to stop splitting.

One way is to set a minimum number of training inputs to use on each leaf: for example we can use a minimum of 10 passengers to reach a decision (died or survived), and ignore any leaf that takes less than 10 passengers. Another way is

simply to set a maximum depth of your model, that is the the length of the longest path from a root to a leaf.

The performance of a tree can be further increased by **pruning**. It involves removing the branches that uses features with low importance. This way, we reduce the complexity of tree, and we increase its predictive power by reducing overfitting. Two different methods of pruning exist:

- **Pre-pruning:** also known as forward or online pruning, it prevents the generation of non-significant branches at the start. It consists on using a 'termination condition' to decide when it is desirable to terminate some branches prematurely during the construction.
- **Post-pruning:** also known as backward pruning, it consists in removing non-significant branches after the generation of the decision tree. The main aim is to adjust it after the creation in order to improve the accuracy on unseen instances. There are two principal methods of Post-Pruning: one converts the tree to an equivalent set of rules. The second method aims to retain the tree but replacing some of its subtrees with leaf nodes.

2.6 Random Forest

Random forests are popularly applied to both data science competitions and practical problems. They are often accurate and do not require feature scaling. They can also be more interpretable than other complex models such as neural networks.

Basically, a random forest is a combination of many decision trees into a single model. So, a random forest consists of multiple random decision trees. Two types of randomnesses are built into the trees. First, each tree is built on a random sample from the original data. Second, at each tree node, a subset of features are randomly selected to generate the best split. [11]

Rather than just simply averaging the prediction of trees (that is “**forest**”), this model uses two key concepts that gives it the name random: [12]

- **Random sampling of training data points when building trees:** when training, each tree in a random forest learns from a random sample of the data points. The samples are drawn with replacement (**bootstrapping**), which means that some samples will be used multiple times in a single tree. The idea is that by training each tree on different samples, overall, the entire forest will have lower variance. At test time, predictions are made by averaging the predictions of each decision tree. This procedure of training each individual learner and then averaging the predictions is known as **bagging** (bag = bootstrap aggregation).
- **Random subsets of features considered when splitting nodes:** the other main concept in the random forest is that only a subset of all the features are considered for splitting each node in each decision tree. Generally the number of selected features is set to $\sqrt{N_{features}}$ for classification: if there are 16 features, at each node in each tree, only 4 random features will be considered for splitting the node (the random forest can also be trained considering all the features at every node as is common in regression).

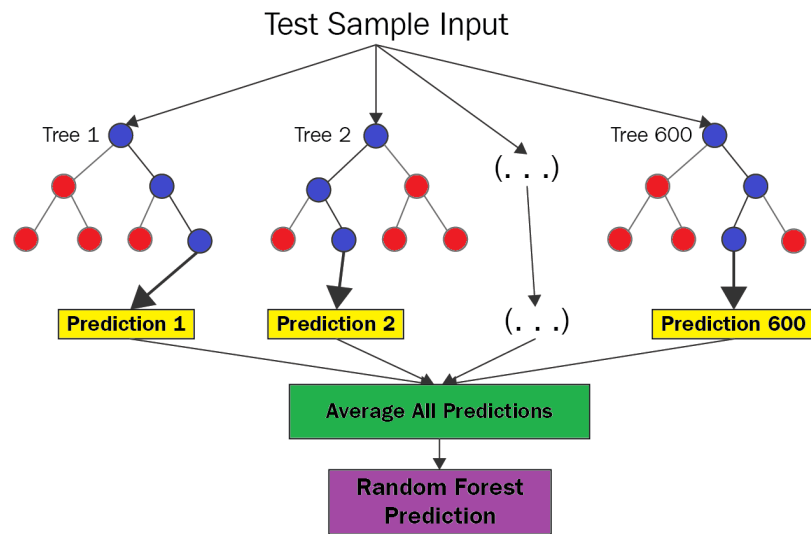


Figure 2.15: Random forest workflow

The gist of the matter is that the random forest algorithm combines hundreds or thousands of decision trees, trains each one on a slightly different set of the observations, splitting nodes in each tree considering a limited number of the features. The final predictions of the random forest are made by averaging the predictions of each individual tree.

2.7 Support Vector Machines

A Support Vector Machine (SVM) is a supervised learning algorithm that can be used for classification (SVC) or regression. Support vector machines are used in applications such as language processing and image recognition.

For classification problems, a SVC constructs an optimal hyperplane as a decision surface where the margin of separation between the two (or more) classes in the data is maximized.

Suppose you have a dataset as shown below and you need to classify the red rectangles from the blue ellipses (positives from the negatives).

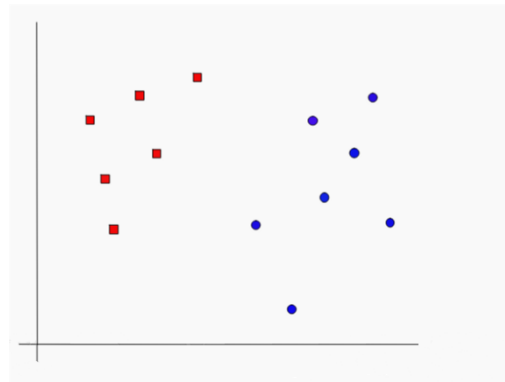


Figure 2.16: Simple dataset

So you have to find a sort of line that separates this dataset in two classes (red and blue). But, as you notice there is not a unique line that does the job. In fact, we have infinite lines that can separate the two classes.

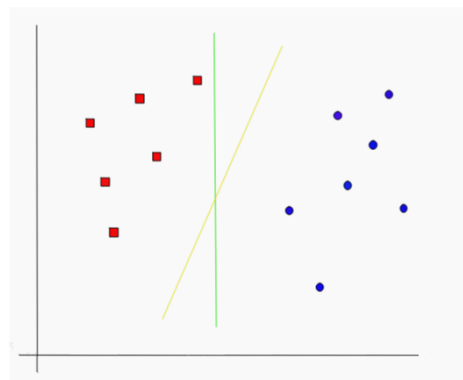


Figure 2.17: Separating lines

So, with SVM algorithm we find the points closest to the line from both the classes, called **support vectors**. Now, we find the distance between the line and the support vectors and we find the **margin**. Our goal is to maximize the margin. The hyperplane for which the margin is maximum is the optimal hyperplane. Support vectors refer to a small subset of the training observations that are used as support for the optimal location of the decision surface.

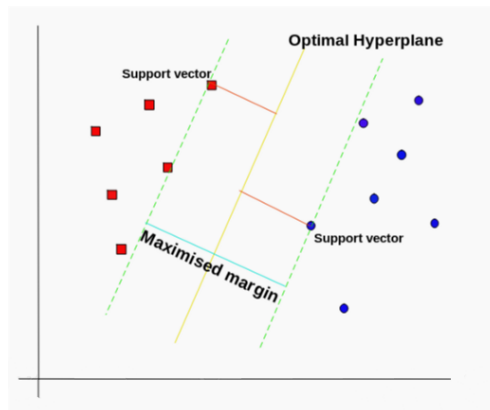


Figure 2.18: Margin and Support Vectors

What if the examples are not linearly separable? SVM fall under a class of machine learning algorithms called kernel methods, which means that SVM uses a set of mathematical functions that are defined as the kernel. The function of kernel is to take data as input and transform it into the required form.

Training for a support vector machine has two phases:

1. Transform predictors (input data) to a high-dimensional feature space. It is sufficient to just specify the kernel for this step and the data is never explicitly transformed to the feature space. This process is commonly known as the kernel trick.
2. Solve a quadratic optimization problem to fit an optimal hyperplane to classify the transformed features into two classes. The number of transformed features is determined by the number of support vectors.

N.B. Only the support vectors chosen from the training data are required to construct the decision surface.

Popular kernels used with SVMs include: [13]

- **Linear:**

$$K(x_1, x_2) = x_1^T x_2$$

- **Polynomial:**

$$K(x_1, x_2) = (x_1^T x_2 + c)^n$$

N is the order of the polynomial and C is a free parameter.

- **Sigmoid:**

$$K(x_1, x_2) = \tanh(\beta_0 x_1^T x_2 + \beta_1)$$

β_0 and β_1 are kernel parameters; it is similar to the sigmoid function of Logistic Regression.

- **Gaussian or Radial Basis Function (RBF):**

$$K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

σ is the width of the kernel. It is a general-purpose kernel; used when there is no prior knowledge about the data.

SVM has even some important parameters called **Tuning Parameters**. Tuning parameters are arguments that you can choose when you are creating your model. They are **Regularization parameter (C)** and **Gamma**:

1. The first one, C , controls the trade off between smooth decision boundary and classifying training points correctly. A large value of C means you will get more training points correctly.

C is a parameter that can be changed inside a SVM's implementation called L_2 **regularization** or **weight decay** [14]. Basically, it is a regularization technique that adds a penalty to large weights in our error function to lower model variance and so over-fitting.

$$w^* = \min_w \text{Error} + \lambda \|w\|^2 \quad \text{where} \quad \lambda = \frac{1}{C}$$

Here, w is a weight vector and the *Error* function is called **hinge loss** function. In the linear SVM it is:

$$\text{Error} = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i(w \cdot x_i - b))$$

Where w are the weights, x is our data, b is the bias. "Hinge" describes the fact that the error is 0 if the data point is classified correctly and it is not too close to the decision boundary, and after that it keeps increasing. [15]

The second term of the L_2 implementation is the regularization term, and λ is the regularization coefficient:

- If λ is too high, the model will be simple, but you run the risk of under-fitting. The model won't learn enough about the training data to make useful predictions.
- If λ value is too low, your model will be more complex, and you run the risk of over-fitting. Your model will learn too much about the training data, and won't be able to generalize the new data.

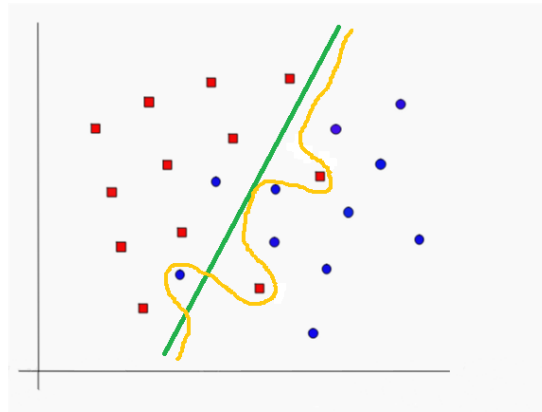


Figure 2.19: C change

In the Figure 2.19, the green line creates a decision boundary which is quite simple and linear but at cost of a few points being misclassified (called **outliers**). Changing the Regularization parameter you can get the orange line: this one is getting almost all the training points correctly, but it is very complicated and it is not going to generalize very well our data. The solution is to try different values of C in order to find a perfectly balanced curve, avoiding over-fitting.

2. The second one, γ , is only used with the Radial Basis Function Kernel.

$$K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right) \quad \text{where} \quad \gamma = \frac{1}{2\sigma^2}$$

It defines how far the influence of a single training example reaches. If gamma has a very high value, then the decision boundary is just going to be dependent upon the points that are very close to the line and ignores some of the points that are very far from the decision boundary. [16]

This is because the closer points get more weight (orange curve in Figure 2.19). On the other hand, if the gamma value is low even the far points get weight and we get a more linear curve (green curve).

Chapter 3

Deep Learning

Until now, we have learned in a really simplistic way what is general Machine Learning. In few words, it is the capability of a AI system to acquire their own knowledge, by extracting patterns from raw data. The performance of the above algorithms depends heavily on the **representation** of the data they are given, so on the features, which are each piece of information of the representation. [17]

Many tasks can be solved just giving to the AI the most correct and the easiest set of features to extract for that task, and then providing them to one of the simple algorithms. But it is not correct for every task: sometimes it might be difficult to understand what features should be extracted. For example, describing each part of a car image that should be detected, it is not so easy. The problem is that the image could be influenced by environmental factors such as sun or shadows.

One solution to this problem is the **Representation Learning**: it discovers not only the mapping to output, but also the representation itself. An example of representation learning algorithm is the Autoencoder, which is a combination of encoder function (converts input data into a different representation) and decoder function (convert the new representation back to the original format).

When designing features, our main plan is to separate the **Factors of variation** that explain the observed data. Factors are often not quantities that you can observe. Some unobserved objects or forces can affect observable quantities. For the above example of car image, the factors of variation include the position of the car, its color or the angle of the sun. So, the major difficulty in many real-world AI application is that the factors of variation can effect every piece of data you need to observe and just a nearly human-level understanding of the data can identify those factors.

Deep learning solves this central problem in representation learning by introducing representations that are split and expressed in terms of simpler ones. So, Deep

Learning enables the system to build complex concepts in function of simpler concepts.

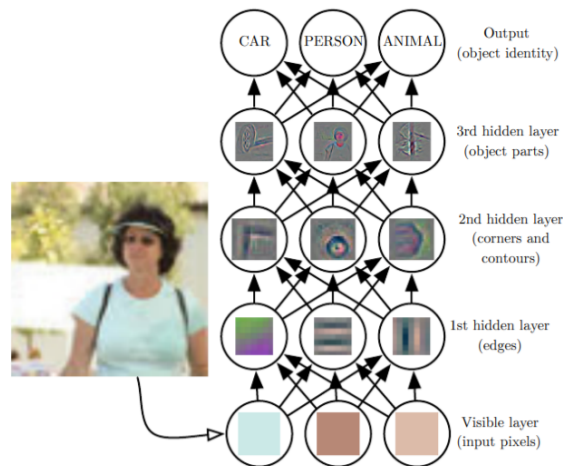


Figure 3.1: Deep learning concepts splitting

The most common deep learning model is the **feed-forward deep network (Multilayer Perceptron or MLP)**. It is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions.

The major aspects of deep learning are the idea of learning the right representation of the data and that the depth enables the system to learn a multi-step program. Sequential instruction offer great power because each step can refer to the results of the previous one. Anyway, not all the information in a layer's activation will encode all the factors of variation that explain the output. It has nothing to do with the content of the input, but it only helps the model to organize its processing.

There are two main ways of measuring the depth of a model, even if it is not always clear which one is most relevant:

- **Computational graph:** it is the number of sequential instructions that must be execute to evaluate the architecture. It is like the length of a decision tree.

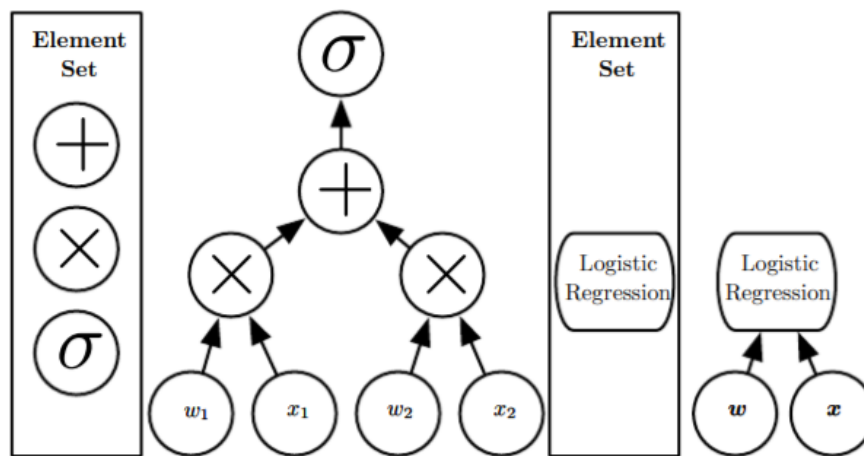


Figure 3.2: Computational graph: in the left, the depth is three because we use addition, multiplication and sigmoids as elements; in the right, the depth is one because logistic regression is an element itself.

- **Probabilistic graph:** it is the depth of the graph describing how concepts are related to each other. It is used in deep probabilistic models and it can be much deeper than the graph only for the concepts.

To summarize, we can say that deep learning is a particular kind of Machine Learning that achieves great power and flexibility by representing the real-world problems as a hierarchy of concepts, with each concept defined in relation to simpler ones.

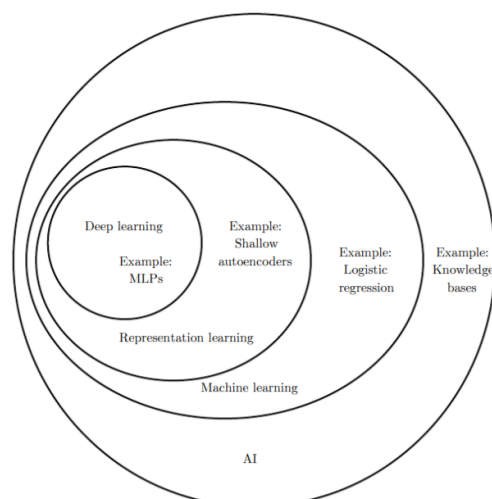


Figure 3.3: AI disciplines

Deep Learning was studied in order to overcome some problems that are impossible to handle with normal Machine learning algorithms. The most important ones are the **Curse of Dimensionality** and his high computational costs. The Curse of Dimensionality is the phenomenon which happens when machine learning problems become too much difficult when the number of dimension in the data is really high. [17]

3.1 Deep Feed-forward Networks

The main goal of a MLP is to approximate some function f^* . For example, for a classifier, $y = f^*(x)$ maps an input x to a category y . A feedforward network defines a mapping $y = f(x, \theta)$ and learns the value of the parameters θ that result in the best function approximation.

These models are called feedforward because information flows through the function evaluated from x , through the computations used to define f , and finally to the output y . There are no feedback connections in which outputs of the model are fed back into itself, so there are not cycles or loop in the network. When feedforward neural networks includes feedback connections, they are called **recurrent** neural networks. [17]

Furthermore, feedforward neural network are called networks because they are composed by many different functions together. For example, with three functions $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ connected in a chain, they form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. These chains are the most common structure of neural networks. $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. The depth of the model is the length of the chain: this is why the name **deep learning**.

In this case, the training data does not show the output of each layer, so they are called **hidden layers**. As the normal machine learning, training data provides us some approximate $f^*(x)$ evaluated at different points. It is then up to the learning algorithm the decision how to use the layer of the chained function to produce the desired output.

Another notifiable thing about the name is **neural**. This is because they are inspired by Neuroscience. The hidden layers are typically vector valued and their dimension determines the **width** of the model. We can think of the layer as many units that act in parallel, where each unit remind to a neuron that receives the input and computes its own activation value. [18]

To understand feedforward networks, it is better to start with linear models and then to pass to how to overcome their limitations. An obvious limitation is that the model capacity is limited to linear functions. To extend them to represent non linear function of x we can just transform the input x to $\phi(x)$ where ϕ is a non linear transformation. After this, we can just apply the kernel trick.

How to choose the mapping ϕ ?

- Very generic ϕ .
- Manually engineer ϕ , this was the dominant approach before deep learning.

- Deep learning's strategy is to learn ϕ . Here we have a model $y = f(x; \theta, w) = \phi(x; \theta)^T * w$. With parameters θ we can learn ϕ from a class of functions, and with parameters w we map from the transformation. Here, ϕ is defining a hidden layer.

3.1.1 Single-layer Perceptron

The simplest kind of neural network is a single-layer network, which consists of a single layer of output nodes; the inputs are fed directly to the outputs via a series of weights. The sum of the products of the weights and the inputs is calculated in each node, and if the value is above some threshold the neuron fires and takes the activated value (typically 1); otherwise it takes the deactivated value (typically -1). Neurons with this kind of activation function are also called artificial neurons or linear threshold units. [19]

Perceptrons can be trained by a simple learning algorithm that is usually called the **delta rule**. It calculates the errors between calculated output and sample output data, and uses this to create an adjustment to the weights, thus implementing a form of gradient descent.

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i$$

Where: α is a small constant called learning rate, $g(x)$ is the neuron's activation function, g' is the derivative of g , t_j is the target output, h_j is the weighted sum of the neuron's inputs, y_j is the actual output and x_i is the i -th input.

A single-layer neural network can compute a continuous output instead of a step function. A common choice is the logistic function: $f(x) = \frac{1}{1+e^{-x}}$. With this choice, the single-layer network is identical to the logistic regression model.

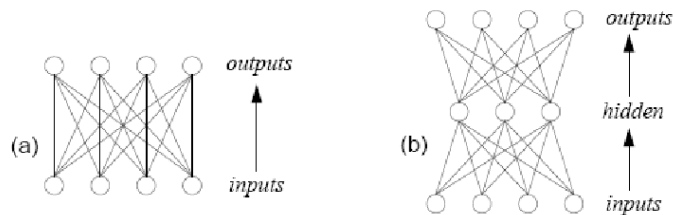


Figure 3.4: (a) Single-layer Perceptron; (b) Multi-layer Perceptron

3.1.2 Multi-layer Perceptron

This class of networks consists of multiple layers interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the sub-

sequent layer. In many applications the units of these networks apply the sigmoid function. [19]

The **universal approximation theorem** for neural networks states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer. This result holds for a wide range of activation functions, e.g. for the sigmoidal functions.

Multi-layer networks use a variety of learning techniques, the most popular is **back-propagation**. Here, the output values are compared with the correct answer to compute the value of some predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small.

To adjust weights properly, one applies the method for non-linear optimization of the **gradient descent**. For this, the network calculates the derivative of the error function with respect to the network weights, and changes the weights such that the error decreases. For this reason, back-propagation can only be applied on networks with differentiable activation functions.

Some typical problems of the back-propagation algorithm are the speed of convergence and the possibility of ending up in a local minimum of the error function. Today there are practical methods that make back-propagation in multi-layer perceptrons the tool of choice for many machine learning tasks.

3.2 Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks or CNNs, are a kind of neural networks for processing data that has a known grid-like topology. CNNs' name is derived by the mathematical operation, called **convolution**, deployed inside the network. In fact, convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. [17]

CNNs are really important in time-series data and image data.

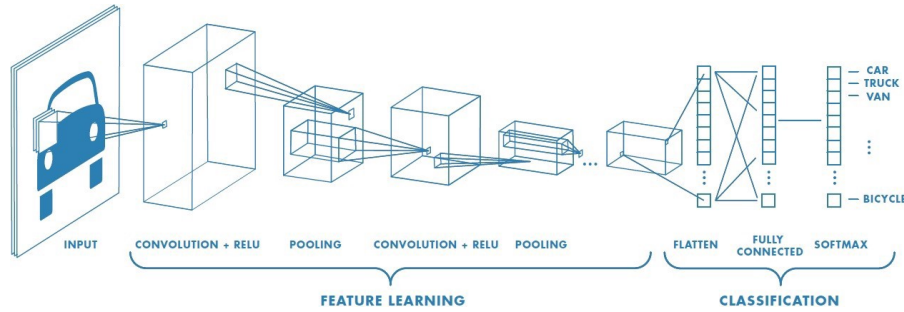


Figure 3.5: CNN scheme

3.2.1 Convolution Operation

In a few words, Convolution is an operation on two functions of a real-valued argument.

Let's suppose we are tracking the location of a spaceship with a laser. The laser has output $x(t)$, so position at time t . They are real-valued, so at each instant of time we have different output.

Now let's suppose that the laser is noisy. To obtain a less noisy output of the position, we should average several measurements considering the recent measurements are the most important ones: this can be done with a weighted average, that means using a weighted function $w(a)$, where a is the age of a measurement. If we apply w at every moment, we obtain a function s which provides a estimation of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This is the convolution operation and it is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In convolution network terminology, the first argument (in the example, function x) to the convolution is the **input**, and the second (the function w , here) is the **kernel**. The output is the **feature map**.

Anyway our example can not be realistic. That is because, when we work with data on a computer, time will be discretized. In our example, the discretized time could be once per second. So the time t can be only an integer value. If we assume that x and w are defined only on integer t , we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In machine learning applications, the input is usually a multi-dimensional array of data, and the kernel is a multi-dimensional array of parameters (referred as tensors). Plus, we assume that these functions are zero everywhere but in the finite set of points for which we store the values. It means that we can implement the infinite summation as a summation over a finite number of array elements. Finally, we can use convolutions to more than one axes in one time. Usually if we use 2-D input, the kernel is 2-D either.

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i-m, j-n)$$

Convolution is commutative, so:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i-m, j-n)K(m, n)$$

The commutative formula is not usually an important property of an implementation. Instead, many neural networks libraries implement a related function called **Cross-correlation** (calling it Convolution), which is the same as convolution but without flipping the kernel like the commutative one:

$$S(i, j) = (I \wedge K)(i, j) = \sum_m \sum_n I(i+m, j+n)K(m, n)$$

It is rare for convolution to be used alone in Machine learning application; instead, it is used simultaneously with other functions.

3.2.2 Pooling

A typical layer of a CNN consists of three stages.

1. At first, the layer performs several convolutions in parallel to produce a set of linear activations.
2. In the second phase, each linear activations in run through a non linear activation function, such as the **ReLU** (Rectified Linear Activation function).

$$f(x) = x^+ = \max(0, x) \text{ where } x \text{ is the input}$$

The rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero. This stage is called **detector stage**.

3. For third, it is common to use a **pooling function** to modify the output of the layer further.

Pooling reduce the number of parameters in your network (pooling is also called **down-sampling** for this reason). A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. One example is the **max pooling**, it reports the maximum output within a rectangular neighborhood.

Pooling helps to make the representation invariant to small translations of the input. It means that, translating the input, the values of most of the pooled outputs will not change. Invariance to translation can be a useful property if we care more about whether some feature is present, than exactly where it is. For instance, if we want to find out if an image contains a face, we do not need to know where the eyes are placed, but only if a left and a right eye are present. Anyway this is not always true. In fact, sometimes it is more important to preserve the location of the feature than just knowing if it is present in the image.

3.3 Recurrent Neural Networks

Recurrent Neural Networks, or RNNs, are a family of neural networks specialized for processing sequential data, such as a sequence of values x^1, \dots, x^T . RNNs are the choice when the problem is about scaling very long sequences of data, also when the sequences have variable length.[17]

The most important idea behind the RNNs is that of sharing parameters across different parts of a model. Parameters sharing make possible the extension of the model to examples of different forms, or length, and make generalizations. Sharing is even more important when a piece of information can occur multiple times inside the sequence, for example when the problem is to find the same string inside different sentences. How do Recurrent Neural Networks share parameters? We can say that each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous ones.

Simple explained, we can say that RNNs operate on a sequence that contains vectors x^t with the time step index t ranging from 1 to τ . Recurrent networks usually operate on minibatches of the sequences, with a different sequence length τ for each member of the minibatch. We can say that the time step is not the real passage of time in real life, but sometimes only refer to the position in the sequence.

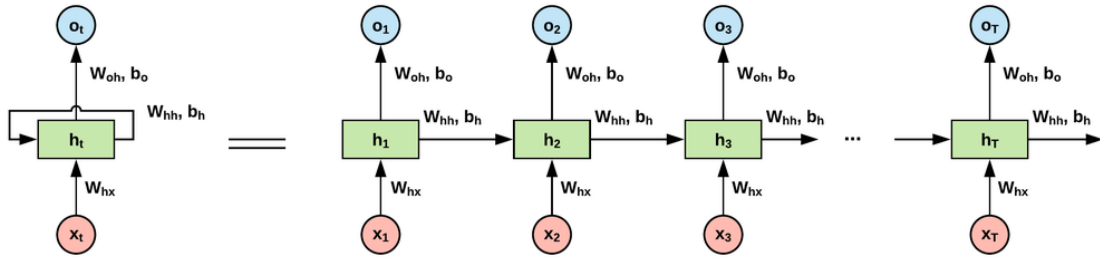


Figure 3.6: RNN scheme

3.3.1 Vanishing and exploding gradient problem

Training a RNN is similar to training a traditional Neural Network. We also use the backpropagation algorithm, but with a little twist. Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. For example, in order to calculate the gradient at $t=4$ we would need to backprop-

agate 3 steps and sum up the gradients. This is called **Backpropagation Through Time (BPTT)**.

But this algorithm has a problem, called **vanishing problem**. It mainly occurs when we are dealing with large time series datasets. In fact, as we go back to the lower layers, gradient often get smaller, eventually causing weights to never change at lower layers. In the most difficult situations, it obliges the training to stop.

The opposite problem is called **exploding problem**: when we go back to the lower layers, gradient start increasing. This alarming increase eventually shoots up the gradient to values of such high magnitudes that model blows up, or crashes.

The solution of these problems can be found in an RNN' extension, so called **LSTM**.

3.3.2 RNN' extensions

Over the years researchers have developed more sophisticated types of RNNs to deal with some of the problems of the vanilla RNN model. Here are some examples: [20]

Bidirectional RNNs are based on the idea that the output at time t may not only depend on the previous elements in the sequence, but also on the future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple, they are just two RNNs stacked on top of each other. The output is then computed based on the hidden state of both RNNs.

Echo state networks or ESNs and **Liquid State Machines** are models created with the idea to overcome a problem of the RNNs. We can say that the recurrent weights mapping from $h(t - 1)$ to $h(t)$ and the input weights mapping from $x(t)$ to $h(t)$ are some of the most difficult parameters to learn in a recurrent network. The approach to avoid this difficulty is to set the recurrent weights so that the recurrent hidden units can do a good job of capturing the history of past inputs, and learn only the output weights. For LSMs, the latter is similar, except that it uses neurons with binary outputs instead of the continuous-valued hidden units used for ESNs. Both ESNs and liquid state machines are termed **reservoir computing** to denote the fact that the hidden units form of reservoir of temporal features may capture different aspects of the history of inputs.

LSTM or Long Short-Term Memory are quite popular these days. LSTMs don't have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. The memory in LSTMs are called cells and you can think of them as black boxes that take as input the previous state $h(t - 1)$ and current input $x(t)$. Internally these cells decide what to keep in and what to erase from memory. They then combine the previous state, the current memory, and the input. It turns out that these types of units are very efficient at capturing long-term dependencies, so they can easily avoid the vanishing or exploding problem.

Chapter 4

Code

In this chapter, I will finally show the examples made during the Machine Learning's studies.

For the basics, I created a program in Jupiter Notebook for each algorithm, but I will not show all of them. I will point out some simple technique to improve the accuracy and the important plots associated with them.

Regarding Deep Learning instead, plotting using a GPU machine would be really difficult, so I will limit myself to show really simple code. In fact, the Tensorflow Frameworks is already prepared for creating a model, only a few lines are enough to build an efficient algorithm.

4.1 Linear and Polynomial Regression

The first program is obviously even the simpler one. Every program starts with the libraries importing, so I might omit them with the others.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
import warnings #had to supress future warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```



```
k = pd.read_csv('Linear Regression - Sheet1.csv')
k.describe()
```

	X	Y
count	300.000000	300.000000
mean	150.500000	102.215556
std	86.746758	57.842711
min	1.000000	1.888889
25%	75.750000	52.388889
50%	150.500000	102.222222
75%	225.250000	152.055556
max	300.000000	201.888889

Figure 4.1: Dataset described

After I have imported the libraries, I have to read the CSV file containing the data I need to study. Pandas' library is useful for this purpose, putting the dataset inside a data structure. Here, the **describe** method shows a summary of the data. We can see that it is composed only by one feature and one output, and it is a Regression problem.

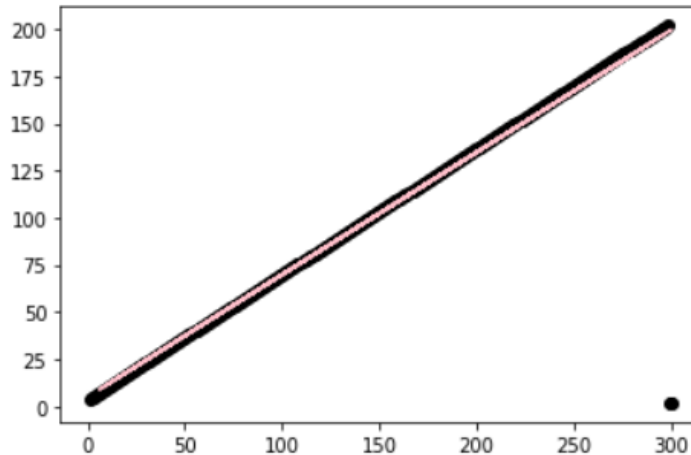
```
X = k.iloc[:, 0].values.reshape(-1,1)
y = k.iloc[:, 1].values.reshape(-1,1)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size = 0.25, random_state = 0)
line = LinearRegression()
line.fit(X_train, y_train)
y_pred = line.predict(X_test)
```

The first two lines are used to reshape the data inside the **k** variable, in order to allocate it in new variables. Then, we are finally going to split into train set and test set using the Sklearn's easy function **train test split**. [8]

The main part of the Machine Learning algorithm starts here and it is the same for everyone: I initialize the Linear Regression, I fit the program using the train set and then I predict the results using the input test.

```
plt.scatter(X,y, color='black')
plt.plot(X_test, y_pred, color='pink', linewidth = 1.5)
```

```
plt.show()
print('Linear accuracy: ', line.score(X_test,y_test))
```



Linear accuracy: 0.844631258824088

Figure 4.2: Linear Regression plotting and accuracy

At the end, using the **matplotlib** library, it is easy to see if the prediction was successful. The black line is composed by every single information of dataset, while the pink one shows every prediction of the program.

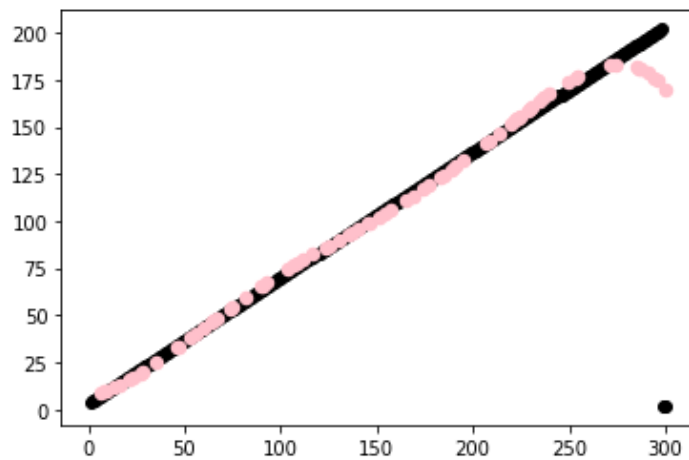
The score method computes by itself the prediction again: this is why the argument passed is only the train data, without the predicted output previously calculated.

Using the same dataset, I implemented the Polynomial Regression algorithm.

```
pol = PolynomialFeatures(degree = 5)
X_pol = pol.fit_transform(X_train)
poly = LinearRegression()
poly.fit(X_pol, y_train)
yp_pred = poly.predict(pol.fit_transform(X_test))
```

The Polynomial Regression's algorithm is almost the same as the Linear one. The Linear Regression has to be initialized, but the Polynomial features have to be created either. This means that the **fit transform** method is important to reshape the input train matrix, with the inclusion of the number of features chosen (5 in the example).

```
plt.scatter(X,y, color='black')
plt.scatter(X_test, yp_pred, color='pink', linewidth = 1.5)
plt.show()
print('Poly accuracy: ',
      poly.score(pol.fit_transform(X_test),y_test))
```



Poly accuracy: 0.8789712546421925

Figure 4.3: Polynomial Regression plotting and accuracy

4.2 Decision Tree

Here, I am trying to use a more difficult dataset, it is a Classification problem and it is composed by four features and one output.

```
w =
    pd.read_csv(r'C:\Users\MHI6\Guido\Datasets\bill_authentication.csv')
w.hist(figsize=(15,9),bins=50)
w.describe()
```

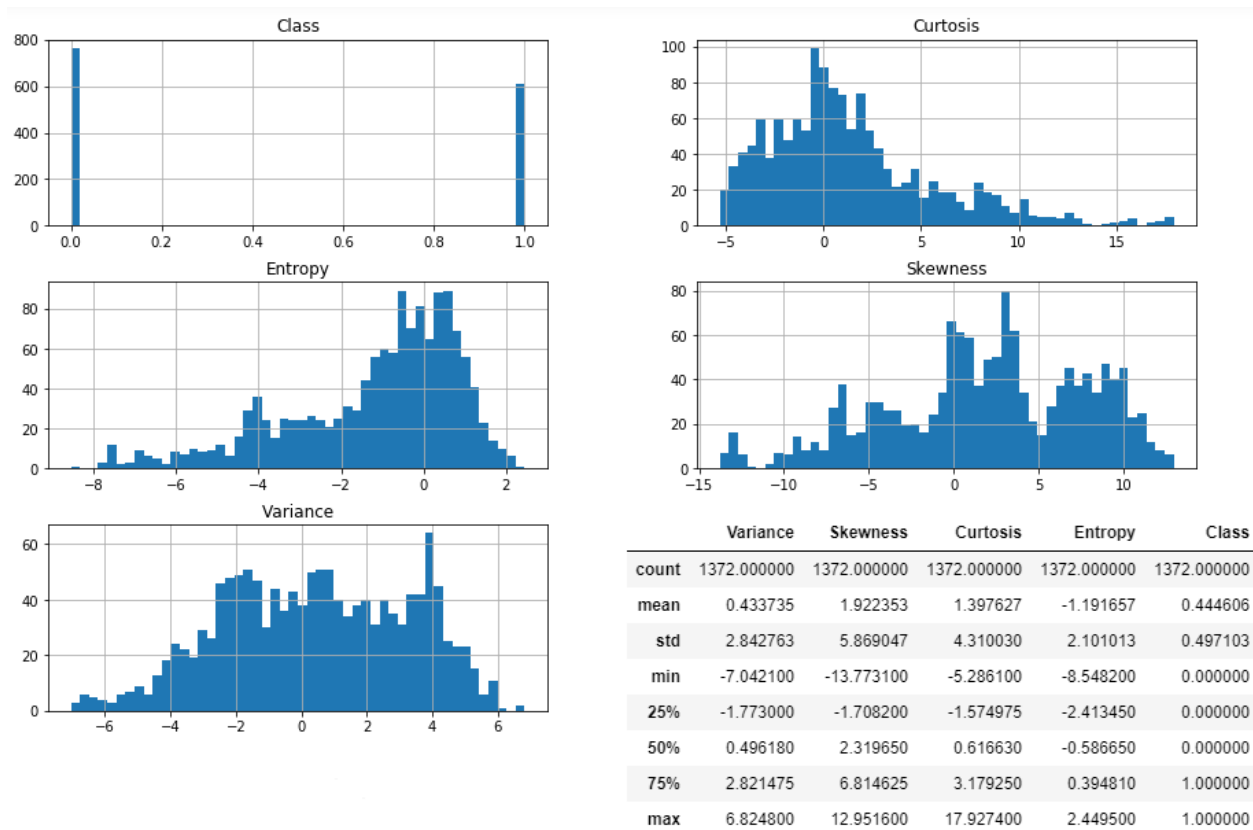


Figure 4.4: Dataset describe and histogram

The description and the histogram help to understand the dataset's content, so it is always useful to include them right at the start.

```

X = w.iloc[:, :4].values
y = w.iloc[:, 4].values
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3, stratify=y)
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)
y_pred = dtc.predict(X_test)
accuracy_score(y_test, y_pred)

```

Out: 0.970873786407767

As the first example, we need to allocate the data inside two variables. We split in train and test, this time using the attribute **stratify**: it splits the data in a stratified fashion when the attributes are not proportional. [8]

After fitting and predicting, we are using another accuracy function which is not associated with the Decision Tree's class. And it is getting a very high value!

```
dot_bill = StringIO()
tree.export_graphviz(dtc, out_file = dot_bill, feature_names =
    w.columns.values[:4])
(graph,) = pydot.graph_from_dot_data(dot_bill.getvalue())
Image(graph.create_png())
```

Here, we are plotting the entire Decision Tree, created with the Gini criterion.

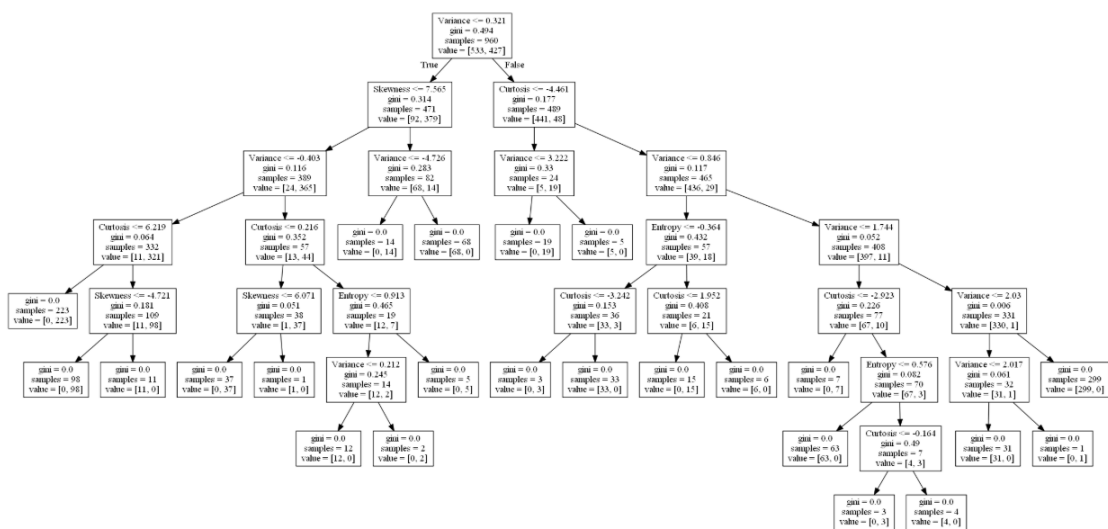


Figure 4.5: Decision Tree with Gini criterion

```
dtc2 = DecisionTreeClassifier(criterion='entropy')
dtc2.fit(X_train, y_train)
y_pred = dtc2.predict(X_test)
accuracy_score(y_test, y_pred)
dot_bill = StringIO()
tree.export_graphviz(dtc2, out_file = dot_bill, feature_names =
    w.columns.values[:4])
(graph,) = pydot.graph_from_dot_data(dot_bill.getvalue())
Image(graph.create_png())
```

Out: 0.9927184466019418

Changing the attribute **criterion** during the Decision Tree's initialization, we can modify it in the Entropy one. And the accuracy improves.

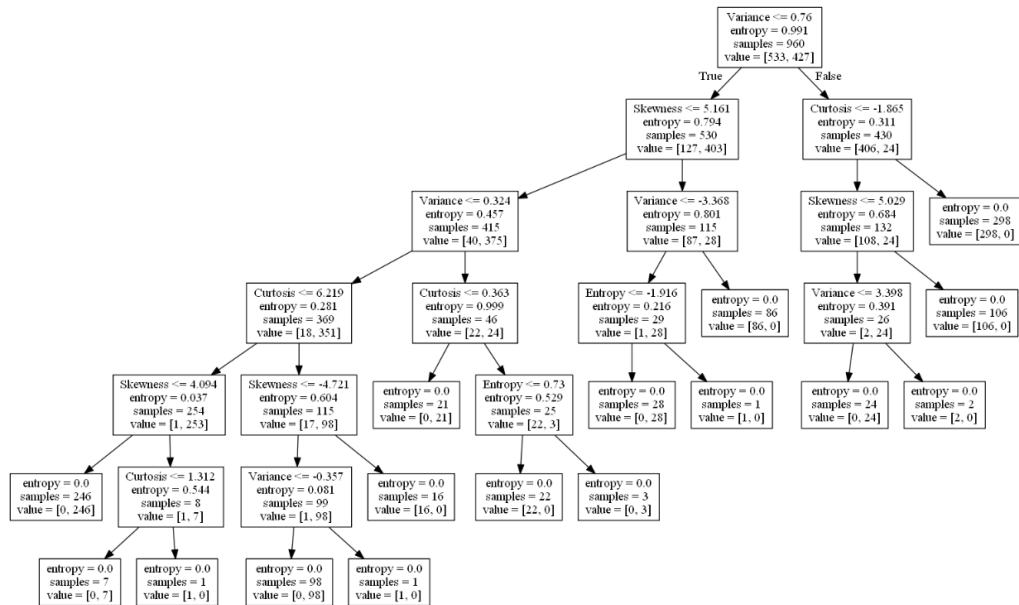


Figure 4.6: Decision Tree with Entropy criterion

```
dtc3 = DecisionTreeClassifier()
param_grid = {'criterion': ['gini', 'entropy'],
              'max_depth': [3, 5, 7, 20]}
gs_inst = GridSearchCV(dtc3, param_grid=param_grid, cv=5)
gs_inst.fit(X_train, y_train)
y_pred_gs = gs_inst.predict(X_test)
accuracy_score(y_test, y_pred_gs)
```

Out: 0.9927184466019418

A function can help in improving the accuracy of the model. The function is the **Grid Search**.

Inside the function, we have to pass our initialized model, the parameters we want to try and the **cross validation** value: in a few words, we vary the split scoring criterion between Gini and Entropy and vary the max depth of a tree, trying them in five different subsets of the training data. [8]

But this time the accuracy is not getting higher.

```
n_classes = 2
```

```

plot_colors = "rb"
plot_step = 0.02
target = np.arange(2)

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3], [1, 2],
                                [1, 3], [2, 3]]):

    #We only take the two corresponding features
    X2 = w.iloc[:, pair].values
    y2 = w.iloc[:, 4].values

    #Train
    tree = DecisionTreeClassifier().fit(X2, y2)

    #Plot the area of the output
    plt.subplot(2, 3, pairidx + 1)
    x_min, x_max = X2[:, 0].min() - 1, X2[:, 0].max() + 1
    y_min, y_max = X2[:, 1].min() - 1, X2[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                          np.arange(y_min, y_max, plot_step))
    plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)
    Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)
    plt.xlabel(w.columns.values[pair[0]])
    plt.ylabel(w.columns.values[pair[1]])

    #Plot the training points
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        plt.scatter(X2[idx, 0], X2[idx, 1], c=color,
                    label=target[i], cmap=plt.cm.RdYlBu, edgecolor='black',
                    s=15)

plt.suptitle("Decision surface of a decision tree using paired
             features")
plt.legend(loc='lower right', borderpad=0, handletextpad=0)
plt.axis("tight")
plt.figure()
plt.show()

```

The last part of the code is helpful to plot the Decision Boundary of the Decision Tree. A Decision Boundary is a hypersurface that partitions the underlying

vector space into two sets (or more), one for each class.

The dataset is composed by many features, so we have to make a Decision Surface for each pair of attributes.

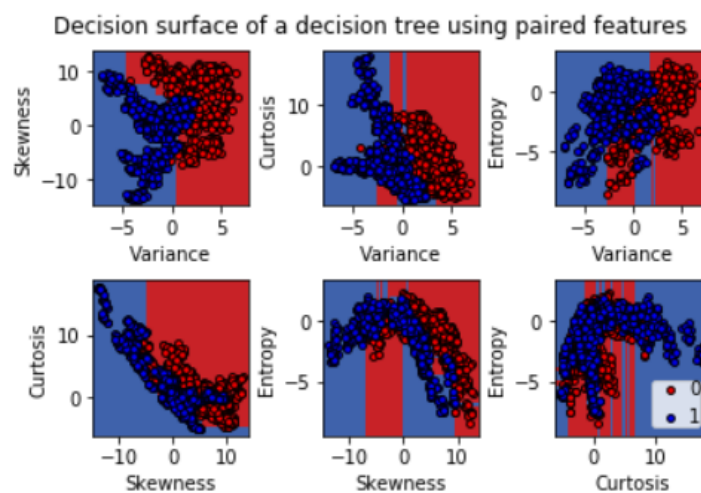


Figure 4.7: Decision Boundary with paired features

4.3 Random Forest

```
adm =  
    pd.read_csv(r'C:\Users\MHI6\Guido\Datasets\Admission_Predict_Ver1.1.csv')  
adm.hist(figsize=(15,9),bins=50)  
adm.describe()
```

The dataset chosen for the Random Forest shows the chances of admission in function of many features. As we can see, the first feature is not important for the program.

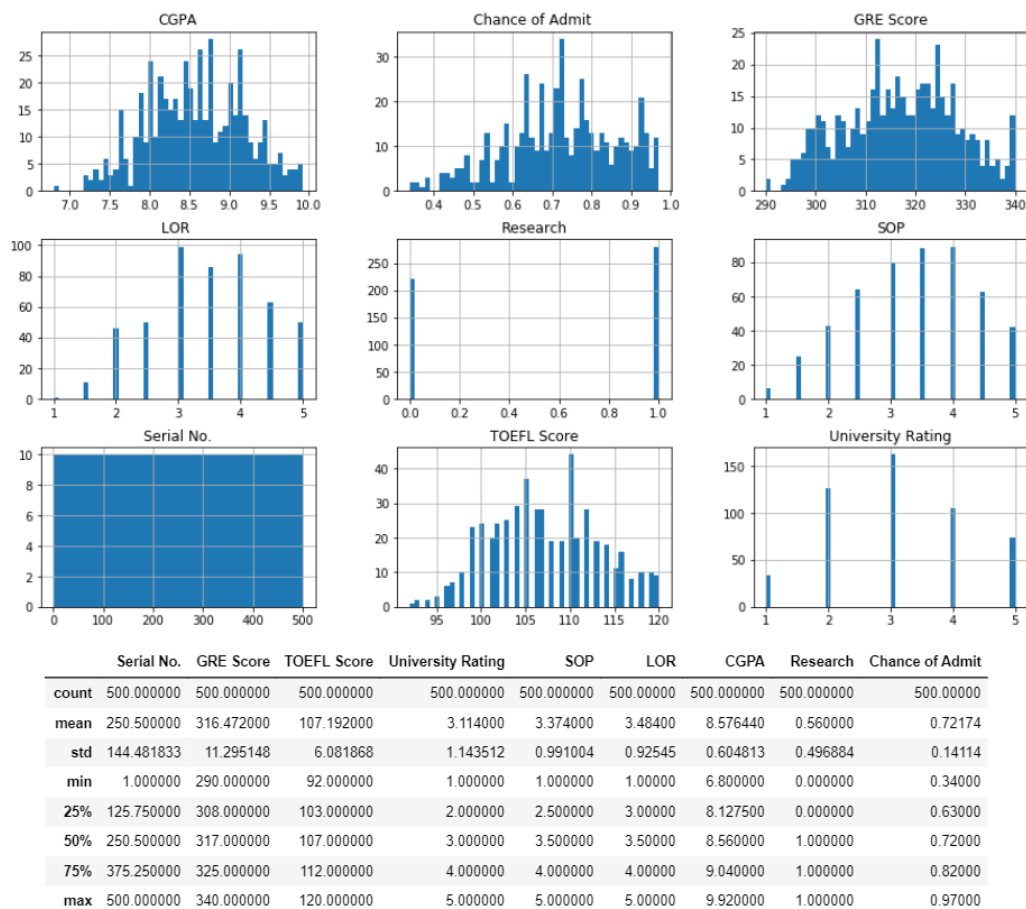


Figure 4.8: Admission dataset describe and histogram

```

X = adm.iloc[:, 1:8].values
y = adm.iloc[:, 8].values
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, stratify = y)
rft = RandomForestRegressor()
rft.fit(X_train, y_train)
y_pred = rft.predict(X_test)

mean_absolute_error(y_test, y_pred)
Out: 0.049673333333333326

(np.abs(y_test - y_pred)/(y_test)).mean()
Out: 0.07940735256505903

```

The **mean absolute error** function calculates the error using only one Decision

Tree of the model, while the second function calculates the mean of the errors of every tree of Random Forest.

```
for i in range (0,9):
    dot_uni = StringIO()
    tree.export_graphviz(rft.estimators_[i], out_file = dot_uni,
        feature_names = adm.columns.values[1:8])
    (graph,) = pydot.graph_from_dot_data(dot_uni.getvalue())
    img = Image(graph.create_png())
    display(img)
```

This loop plots every Decision Tree of the algorithm: the attribute **estimators** outputs an array filled with every single tree, it can be easily read thanks to the use of an index (see Figure 4.9).

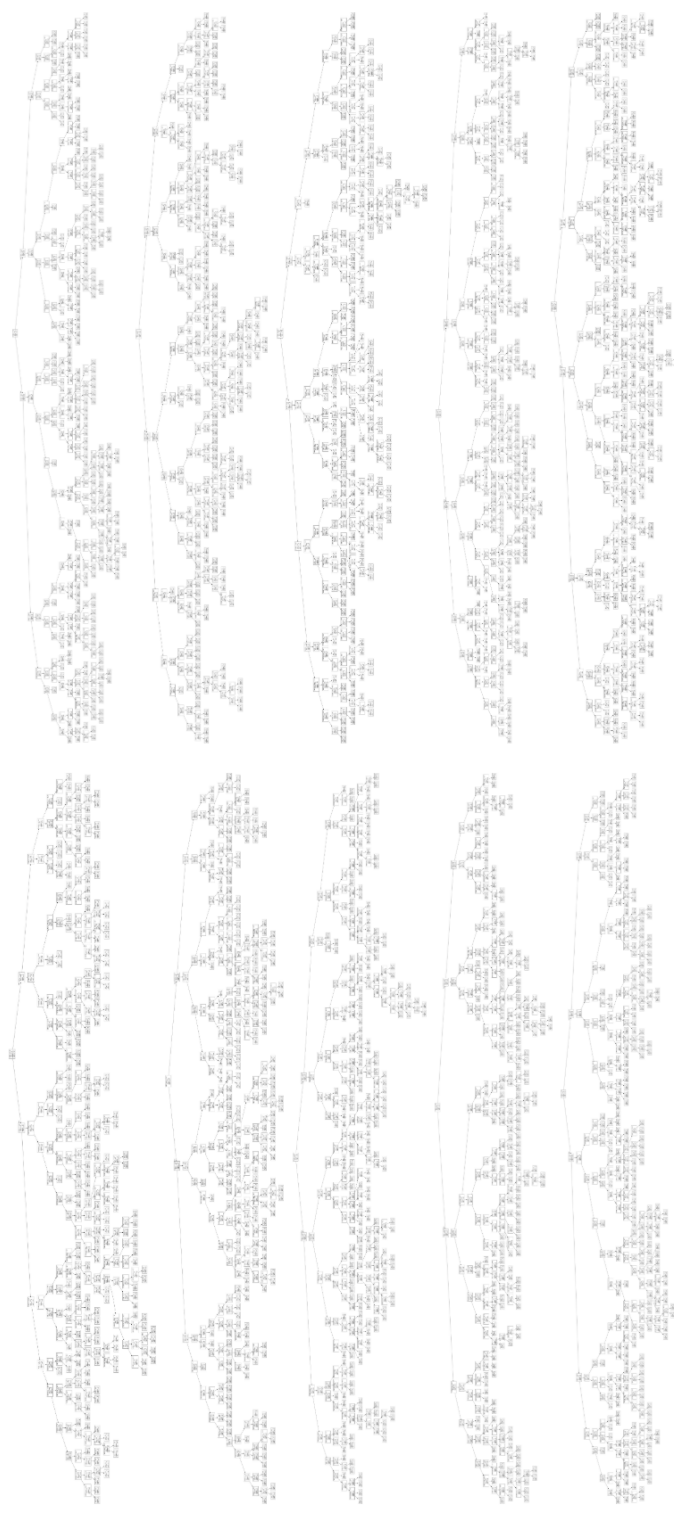


Figure 4.9: All the Random Forest's trees

```
rft.feature_importances_
```

```
Out: array([0.11067352, 0.03755564, 0.0125767 , 0.02357941,  
          0.01848026, 0.7840063 , 0.01312817])
```

```
pd.Series(rft.feature_importances_,  
          index=adm.columns.values[1:8]).nlargest(8).plot(color='red',  
          kind='barh')
```

The **feature importances** model's attribute creates an array containing all the attributes' importances in order to predict the result. Using a simple row of code, there is the plot.

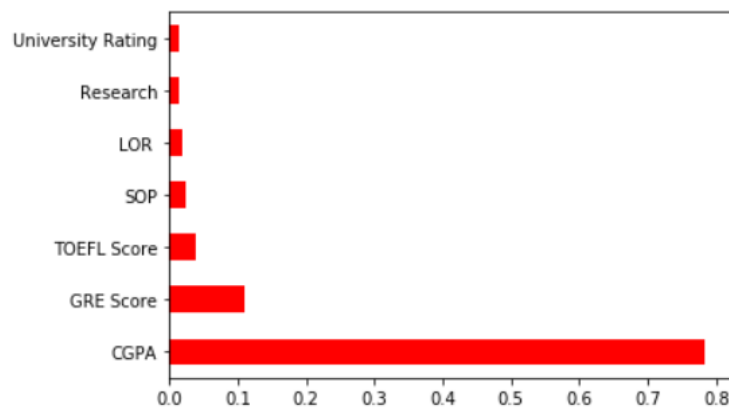


Figure 4.10: Attributes importances

At the end, I tried to make the same Decision Boundary as the Decision Tree example. But here we have a Regression problem, so it is really difficult to find the correct way to visualize every output's area associated with the inputs, even pretending that the outputs are limited like a Classification problem.

4.4 Logistic Regression and SVC

```
w = pd.read_csv(r'C:\Users\MHI6\Guido\Datasets\diabetes2.csv')  
w.hist(figsize=(15,9),bins=50)  
w.describe()
```

In this exercise, I am trying to use the medical parameters to predict the diagnosis of diabetes. Two different algorithms will be shown.

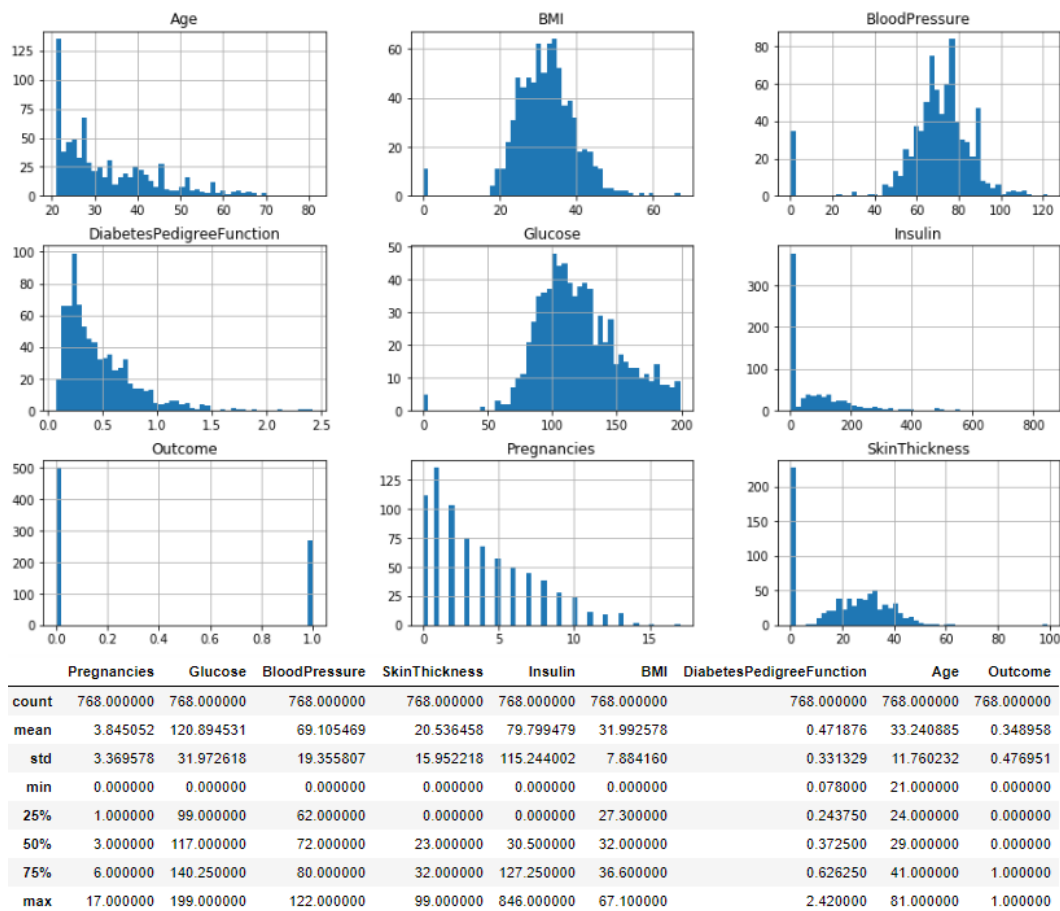


Figure 4.11: Diabetes dataset describe and histogram

```

X = w.iloc[:, :8].values
y = w.iloc[:, 8].values

tsne = TSNE(n_components=2, random_state=0)
tsne_obj = tsne.fit_transform(X)
vis_x = tsne_obj[:, 0]
vis_y = tsne_obj[:, 1]
plt.scatter(vis_x, vis_y, c=y, cmap=plt.cm.get_cmap("jet", 2), s =
    1)
plt.colorbar(ticks=range(2))
plt.show()

```

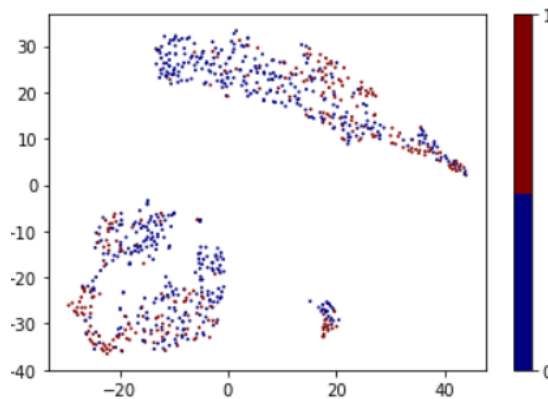


Figure 4.12: t-SNE plot

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets. So, t-SNE gives you a feel or intuition of how the data is arranged in a high-dimensional space. Sklearn's library includes a class which can easily be included in the program.

The **fit transform** function takes the array X (composed of 8 features) and decrease its dimension into **n components** attribute specified in the initialization.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.25, random_state=0)
X_train_2, X_test_2, y_train_2, y_test_2 =
    train_test_split(X_train, y_train, test_size=0.25,
        random_state=0)
svc_clf = SVC(kernel = 'linear', random_state = 0).fit(X_train_2,
    y_train_2)
lr_clf = LogisticRegression(random_state = 0).fit(X_train_2,
    y_train_2)

svc_pred = svc_clf.predict(X_test_2)
print ('Accuracy of SVC:', accuracy_score(y_test_2,svc_pred))
Out: Accuracy of SVC: 0.7777777777777778
print ('Accuracy of SVC on original Test Set:
    ',accuracy_score(y_test, svc_clf.predict(X_test)))
Out: Accuracy of SVC on original Test Set: 0.796875

lr_pred = lr_clf.predict(X_test_2)
print ('Accuracy of LR:', accuracy_score(y_test_2,lr_pred))
Out: Accuracy of LR: 0.7777777777777778
```

In order to improve the accuracy, it is possible to further divide the train set with **cross validation** technique and then we can predict the accuracy with the divided ones. The accuracy is the same for both models. It is anyway easy to predict the accuracy on the original test set.

```
svc_scores = cross_val_score(svc_clf, X_train, y_train, cv=4)
print ('Average SVC scores: ', svc_scores.mean())
print ('Standard Deviation of SVC scores:', svc_scores.std())
Out: Average SVC scores: 0.7568844948155293
Out: Standard Deviation of SVC scores: 0.017842790888631447

lr_scores = cross_val_score(lr_clf, X_train, y_train, cv=4)
print ('Average LR scores: ', lr_scores.mean())
print ('Standard Deviation of LR scores: ', lr_scores.std())
Out: Average LR scores: 0.7533397636845912
Out: Standard Deviation of LR scores: 0.027907292694692474
```

The **cross val score** function creates an array containing the scores reached for each subset of the training set. Being it an array, it is enough the use of some maths functions in order to calculate the mean and the standard deviation.

```
confusion_matrix(y_test_2, lr_pred, labels = [1,0])
Out: array([[29, 26],
           [ 6, 83]], dtype=int64)

y_pred_proba = lr_clf.predict_proba(X_test)
y_pred_low = binarize(y_pred_proba, threshold=0.2)
confusion_matrix(y_test, y_pred_low[:,1], labels=[1,0])
Out: array([[61, 1],
           [79, 51]], dtype=int64)
```

Let's study the confusion matrix of the Logistic Regression model. In the main diagonal, there are 29 people with a correct diagnosis of diabetes and 83 with a correct diagnosis of not having diabetes. In the secondary diagonal instead, there are 26 people with the diagnosis of not having diabetes although they have and 79 with the diagnosis of diabetes although they have not.

As said in the theory, sometimes is better having the positive diagnosis even if it is not correct then let people go home with a wrong diagnosis which can be fatal for their health.

It is possible to change the threshold of the confusion matrix in order to improve its accuracy. At first, we have to calculate the predictions probabilities using the **predict_proba** function. Its output will be a matrix where in the first column there will be the probability for each prediction of being 0, 1 in the second.

It can be used in the confusion matrix function after binarized it. The accuracy is improving but the False Negatives should turn to 0.

```
def npv_func(th): #NPV function for accuracy of threshold
    y_pred_low = binarize(y_pred_proba, threshold=th)
    second_column =
        confusion_matrix(y_test,y_pred_low[:,1],labels=[1,0])[:,1]
    npv = second_column[1]/second_column.sum()
    return npv
npv_func(0.2)
npv_func(0.22)
npv_func(0.18)
Out: 0.9807692307692307
Out: 0.9682539682539683
Out: 1.0
```

The NPV function can help to find the correct threshold for the data studied. Here, the best threshold is 0.18.

```
y_pred_low = binarize(y_pred_proba, threshold=0.18) #best threshold
confusion_matrix(y_test, y_pred_low[:,1],labels=[1,0])
Out: array([[62, 0],
           [92, 38]], dtype=int64)

ths = np.arange(0,1,0.01)
npvs = []
for th in np.arange(0,1.00,0.01):
    npvs.append(npv_func(th))
plt.plot(ths,npvs)
```

The plot of the NPV function can give a more accurate detail of the thresholds.

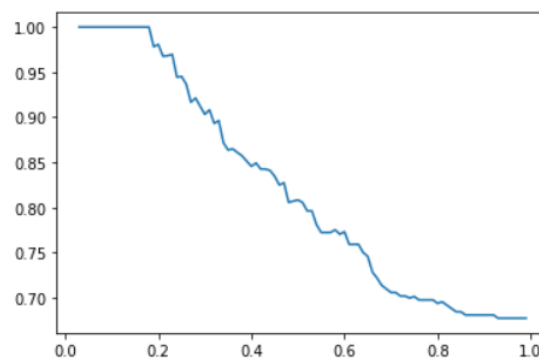


Figure 4.13: NPV plot

```
fpr, tpr, ths = roc_curve(y_test, y_pred_proba[:,1])
plt.plot(fpr,tpr)
auc(fpr,tpr)
```

Out: 0.851985111662531

The **roc curve** function returns the values useful for the plotting of the ROC curve, including an array of thresholds. Plotting the ROC curve and calculating the accuracy through the AUC (area under curve) is easy thanks to the Sklearn library.

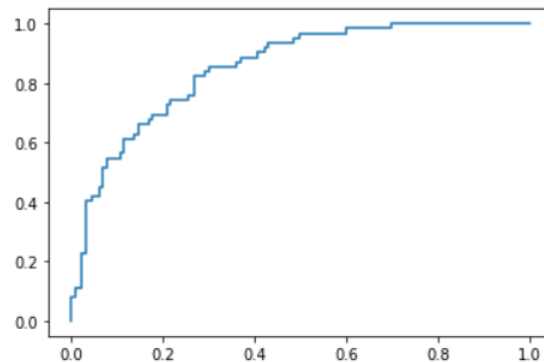


Figure 4.14: ROC plot

```
def model(x):
    return 1 / (1 + np.e**(-x))
for i in range (8):
    cv = StratifiedShuffleSplit(n_splits=7, test_size=0.2,
                                random_state=7) # Cross validation
    logregpipe = Pipeline([('scale', StandardScaler()),
                            ('logreg',LogisticRegression(multi_class="multinomial",solver="lbfgs"))])
    Cs = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100] # 5 parameters for
    cross validation tries
    param_grid = dict(logreg__C=Cs)
    logreg_cv = GridSearchCV(logregpipe,param_grid =
                              param_grid,cv=cv)
    logreg_cv.fit(X_train,y_train)

    bestlogreg = logreg_cv.best_estimator_ # Getting the best
    estimator (best parameter)
    bestlogreg.fit(X_train,y_train)
```

```

bestlogreg.coef_ = bestlogreg.named_steps['logreg'].coef_
bestlogreg.intercept_ =
    bestlogreg.named_steps['logreg'].intercept_

loss = model(X_train * bestlogreg.coef_ +
    bestlogreg.intercept_) # Manually calculating the
    probability
tmp = X_train * bestlogreg.coef_ + bestlogreg.intercept_
plt.scatter(tmp[:,i], y_train, s = 1, color = 'black')
plt.scatter(tmp[:,i], loss[:,i], color = 'red')
plt.xlabel(w.columns.values[i])
plt.ylabel('Probability')
plt.show()

```

A fundamental plot of the Logistic Regression algorithm is the one of the sigmoid function [8]. It explains the relationship that exists between a feature and its probability associated with the Classification output.

For each iteration of the code, a Grid Search is used, in order to find the best regularization parameter for the plot. The regularization parameter applies a penalty to increase the magnitude of parameter values in order to reduce overfitting.

Here, I am defining a function to manually calculate the probability, then using the **coef** and **intercept** attributes of the model. Another simple way is just using the **predict_proba** function already shown above.

In a simple dataset with no more than 2 features, the sigmoid is pretty easy to visualize. This is a multivariate model, so the plots hide the fact that the probability is influenced by the other variables not being shown, creating the horizontal spread.

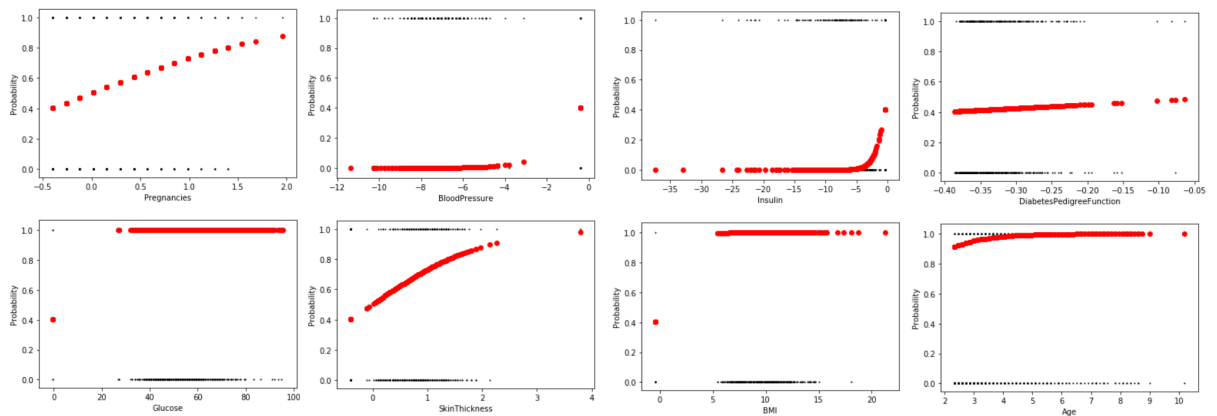


Figure 4.15: Sigmoid plot for each feature

Regarding the SVC model, again the decision boundary is one of the most

important plot. As known, it is possible to decide the kernel of the model. The linear and the polynomial ones are almost the same: the only difference is the possibility of choosing the polynomial degree. I am using the default one (that is the third degree), this is because the program execution using the polynomial kernel is very slow.

```

k = ['linear', 'poly']
for pair in list([[0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6],
                 [0, 7], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [2, 3],
                 [2, 4], [2, 5], [2, 6], [2, 7], [3, 4], [3, 5], [3, 6], [3, 7], [4,
                 5], [4, 6], [4, 7], [5, 6], [5, 7]]):
    X_2 = w.iloc[:, pair].values
    Xy1 = X_2[y==0]
    Xy2 = X_2[y==1]
    X_train, X_test, y_train, y_test = train_test_split(X_2, y,
                                                         test_size=0.2, stratify=y)
    xmin, xmax = np.percentile(X_2[:, 0], [0, 100])
    ymin, ymax = np.percentile(X_2[:, 1], [0, 100])
    test_points = np.array([[xx, yy] for xx, yy in
                             product(np.linspace(xmin, xmax), np.linspace(ymin, ymax))])

    i=0
    for i in range(2):
        svm_inst = SVC(kernel=k[i])
        svm_inst.fit(X_train, y_train)
        test_preds = svm_inst.predict(test_points)
        plt.figure(figsize=(10,7))
        plt.scatter(Xy1[:,0], Xy1[:,1], color = 'red') # Scatter of
            the background area
        plt.scatter(Xy2[:,0], Xy2[:,1], color = 'blue')
        colors = np.array(['r', 'b'])
        plt.scatter(test_points[:, 0], test_points[:, 1],
                    color=colors[test_preds], alpha=0.25) # Scatter of the
            points
        plt.xlabel(w.columns.values[pair[0]])
        plt.ylabel(w.columns.values[pair[1]])
        plt.scatter(X_2[:, 0], X_2[:, 1], color=colors[y])
        plt.title('%s-separated classes' %(k[i]))
        plt.show()

```

For each pair of features I am plotting the Linear and Polynomial kernel decision boundary. Therefore, only few charts of them are shown below.

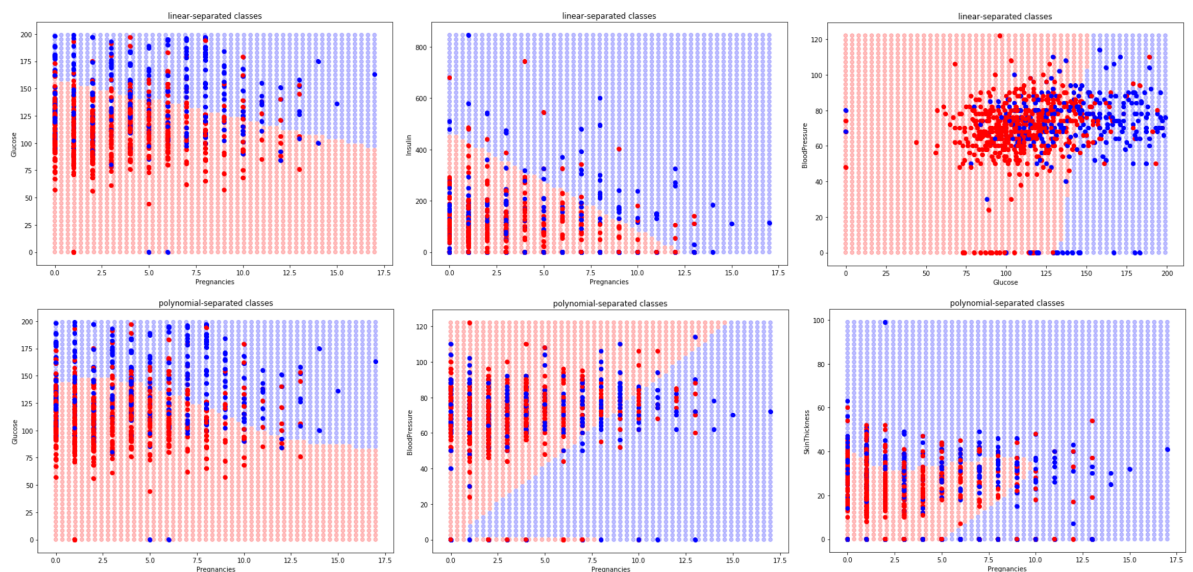


Figure 4.16: Linear and Polynomial kernel

The second part of the code has the same purpose, but applying the RBF kernel. In addition, the Grid Search is used in order to find the best parameters for the plot.

```
for pair in list([[0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6],
                 [0, 7], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [2, 3],
                 [2, 4], [2, 5], [2, 6], [2, 7], [3, 4], [3, 5], [3, 6], [3, 7], [4,
                 5], [4, 6], [4, 7], [5, 6], [5, 7]]):
    X_2 = X[:, pair]
    Xy1 = X_2[y==0]
    Xy2 = X_2[y==1]
    X_train, X_test, y_train, y_test = train_test_split(X_2, y,
                                                         test_size=0.2, stratify=y)
    xmin, xmax = np.percentile(X_2[:, 0], [0, 100])
    ymin, ymax = np.percentile(X_2[:, 1], [0, 100])

    svm_est =
        Pipeline([('scaler', StandardScaler()), ('svc', SVC(kernel="rbf"))])
    cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2,
                               random_state=7) # Cross validation
    Cs = [0.001, 0.01, 0.1, 1, 10] # 5 parameters for cross
        validation tries
    gammas = [0.001, 0.01, 0.1, 1, 10]
    param_grid = dict(svc__gamma=gammas, svc__C=Cs) # Parameters
        given to grid
    grid_cv = GridSearchCV(svm_est, param_grid=param_grid, cv=cv)
```

```

grid_cv.fit(X_train, y_train)
test_points = np.array([[xx, yy] for xx, yy in
    product(np.linspace(xmin, xmax), np.linspace(ymin, ymax))])
test_preds = grid_cv.predict(test_points)
plt.figure(figsize=(10,7))
plt.scatter(Xy1[:,0],Xy1[:,1], color = 'red') # Scatter of
background area
plt.scatter(Xy2[:,0],Xy2[:,1], color = 'blue')
colors = np.array(['r', 'b'])
plt.scatter(test_points[:, 0], test_points[:, 1],
    color=colors[test_preds-1], alpha=0.25) # Scatter of the
points
plt.xlabel(w.columns.values[pair[0]])
plt.ylabel(w.columns.values[pair[1]])
plt.scatter(X_2[:, 0], X_2[:, 1], color=colors[y-1])
plt.title("RBF-separated classes")
plt.show()

```

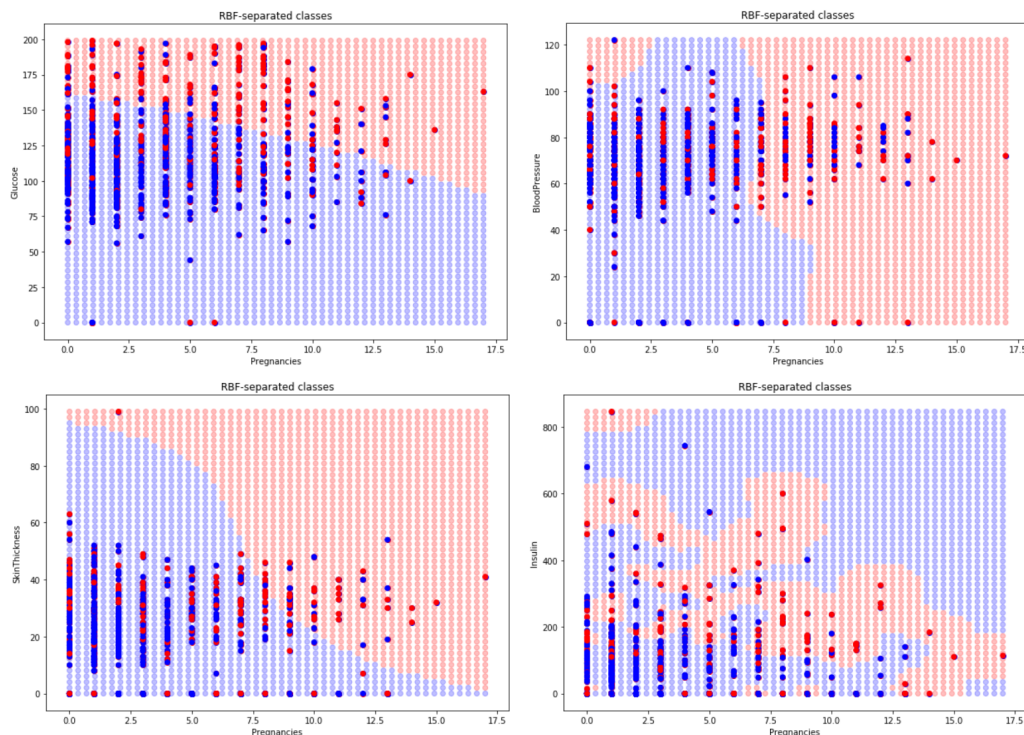


Figure 4.17: RBF kernel

4.5 Deep Learning

Using the GPU machine, I tried to create some simple MLP, CNN and RNN programs using the most common dataset: the MNIST handwritten digit database. It is a huge dataset containing handwritten digits that is commonly used for training various image processing systems. It is well-suited for models such as CNN, but it is not very suitable for others, like the RNN.

For each of the three types of algorithm explained previously, I used two different libraries: Tensorflow, which is essentially a sort of low-level language to build Deep Learning's models, and Keras, which is implemented inside Tensorflow and it is a very simple high-level language for the same purpose.

4.5.1 Multilayer Perceptron

```
#Read the data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

#Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1
n_hidden_1 = 256
n_hidden_2 = 256
n_input = 784 # 28x28
n_classes = 10

#Graph
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])

#Store weights and biases
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
        #784x256
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
        #256x256
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes])),
        #256x10
}
biases = {
```

```

    'b1': tf.Variable(tf.random_normal([n_hidden_1])),          #256x1
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),          #256x1
    'out': tf.Variable(tf.random_normal([n_classes]))           #10x1
}

#Model
def mpl(x, weights, biases):
    print('x:', x.get_shape(), 'W1:', weights['h1'].get_shape(),
          'b1:', biases['b1'].get_shape())
    #Hidden layer with ReLU
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    #Hidden layer with ReLU
    print('layer_1:', layer_1.get_shape(), 'W2:',
          weights['h2'].get_shape(), 'b2:', biases['b2'].get_shape())
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    #Output layer with linear activation
    print('layer_2:', layer_2.get_shape(), 'W3:',
          weights['out'].get_shape(), 'b3:', biases['out'].get_shape())
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    print('out_layer:', out_layer.get_shape())
    return out_layer

#Construct model
pred = mpl(x, weights, biases)

#Loss function and Optimizer
cost =
    tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits
    = pred, labels = y))
optimizer =
    tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

#Initializing the variables
init = tf.initialize_all_variables()

#Launch
with tf.Session() as sess:
    sess.run(init)
    #Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.

```

```

total_batch = int(mnist.train.num_examples/batch_size)
#Loop all batches
for i in range(total_batch):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    #Run optimization and cost op (loss)
    _, c = sess.run([optimizer, cost], feed_dict={x:
        batch_x,y: batch_y})
    #Compute average loss
    avg_cost += c / total_batch
#Display epoch step
if epoch % display_step == 0:
    print ("Epoch:", '%04d' % (epoch+1), "cost=",
        \"{:.9f}".format(avg_cost))
print("Optimization Finished!")
#Test
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y,
    1))
#Accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print ("Accuracy:", accuracy.eval({x: mnist.test.images, y:
    mnist.test.labels}))

```

This is the example using the Tensorflow library.

It is easy to understand how it works: the neural network parameters can be chosen, including the input and the output, the weights and the biases with their shapes. Then the model can be defined like a chained function, so each layer is strictly correlated with the previous one.

The next step is initializing the variables and the model, and defining a cost function and an optimizer. The most used one is the **Adam**: it is an optimizer extension of the already known Stochastic Gradient Descent.

The **cost function** depends on the problem type. If it is a binary classification, the most correct cost function is the binary cross-entropy; for categorical classification, the best one is the categorical cross-entropy; for regression, it is the mean squared error. The purpose is common: it is about calculating the difference between the predicted value and the real one.

The final part is just about training and then testing the model using a Session. Everything written in Tensorflow can be further simplified with the Keras framework, follows the code:

```

#Parameters
batch_size = 128
num_classes = 10
epochs = 15

```



```

#Preprocessing the data
(x_train, y_train), (x_test, y_test) =
    tf.keras.datasets.mnist.load_data() #Splitting
x_train = x_train.reshape(60000, 784) #Correcting the shape
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32') #Correcting the format
x_test = x_test.astype('float32')
x_train /= 255 #Normalize
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

#Convert to binary class matrices (one hot)
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

#Model
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(512, activation='relu',
    input_shape=(784,)))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(), metrics=['accuracy'])

#Train and test
history = model.fit(x_train, y_train, batch_size=batch_size,
    epochs=epochs, verbose=1, validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

```

With Keras, creating the model and evaluating is easier than using the Tensorflow framework. In fact, the model is built per block, without the need of defining the biases and weights. Many function and attributes help to improve and evaluate the model, and the code is better readable than the first one.

```

Layer (type)                 Output Shape                 Param #
-----
dense_1 (Dense)              (None, 512)                 401920
dense_2 (Dense)              (None, 512)                 262656
dense_3 (Dense)              (None, 10)                  5130
-----
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples
Epoch 1/15
60000/60000 [=====] - 7s 110us/step - loss: 0.2226 - acc: 0.9337 - val_loss: 0.1191 - val_acc: 0.9639
Epoch 2/15
60000/60000 [=====] - 3s 52us/step - loss: 0.0792 - acc: 0.9755 - val_loss: 0.0817 - val_acc: 0.9736
Epoch 3/15
60000/60000 [=====] - 3s 57us/step - loss: 0.0511 - acc: 0.9840 - val_loss: 0.0718 - val_acc: 0.9768
Epoch 4/15
60000/60000 [=====] - 3s 55us/step - loss: 0.0369 - acc: 0.9881 - val_loss: 0.0697 - val_acc: 0.9797
Epoch 5/15
60000/60000 [=====] - 3s 49us/step - loss: 0.0262 - acc: 0.9913 - val_loss: 0.0627 - val_acc: 0.9814
Epoch 6/15
60000/60000 [=====] - 3s 53us/step - loss: 0.0248 - acc: 0.9921 - val_loss: 0.0747 - val_acc: 0.9786
Epoch 7/15
60000/60000 [=====] - 3s 50us/step - loss: 0.0188 - acc: 0.9934 - val_loss: 0.0635 - val_acc: 0.9818
Epoch 8/15
60000/60000 [=====] - 3s 50us/step - loss: 0.0162 - acc: 0.9947 - val_loss: 0.0721 - val_acc: 0.9819
Epoch 9/15
60000/60000 [=====] - 3s 48us/step - loss: 0.0169 - acc: 0.9942 - val_loss: 0.0805 - val_acc: 0.9807
Epoch 10/15
60000/60000 [=====] - 3s 49us/step - loss: 0.0137 - acc: 0.9953 - val_loss: 0.0694 - val_acc: 0.9814
Epoch 11/15
60000/60000 [=====] - 3s 52us/step - loss: 0.0099 - acc: 0.9967 - val_loss: 0.0840 - val_acc: 0.9823
Epoch 12/15
60000/60000 [=====] - 4s 63us/step - loss: 0.0113 - acc: 0.9959 - val_loss: 0.0808 - val_acc: 0.9812
Epoch 13/15
60000/60000 [=====] - 3s 50us/step - loss: 0.0114 - acc: 0.9964 - val_loss: 0.0806 - val_acc: 0.9813
Epoch 14/15
60000/60000 [=====] - 3s 50us/step - loss: 0.0122 - acc: 0.9960 - val_loss: 0.0883 - val_acc: 0.9810
Epoch 15/15
60000/60000 [=====] - 3s 51us/step - loss: 0.0093 - acc: 0.9969 - val_loss: 0.0877 - val_acc: 0.9821
('Test loss:', 0.08766472982572664)
('Test accuracy:', 0.9821)
guido@pc097070:~/Guido$

```

Figure 4.18: Keras Multilayer Perceptron Output

4.5.2 Convolutional Neural Network

The differences between the Deep Learning algorithms written in Python are really a few, therefore I will limit myself to explain what changes. In CNN example, only the model changed and now it needs as input the real picture format: it is not anymore an array but a matrix (28x28).

```

#Conv2D
def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1],
        padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)

#MaxPool2D

```

```

def maxpool2d(x, k=2):
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k,
        1],padding='SAME')

#Model
def conv_net(x, weights, biases):
    #Reshape to match picture format [Height x Width x Channel]
    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    #Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    #Max Pooling
    conv1 = maxpool2d(conv1, k=2)
    #Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    #Max Pooling
    conv2 = maxpool2d(conv2, k=2)
    #Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1,
        weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    #Output
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out

```

The CNN model differs to the MLP one because it includes at least one convolutional layer. A convolutional function is defined to simplify the code and the MaxPooling is used to down-sample the input representation.

```

x_train = x_train.reshape((x_train.shape[0], 28, 28, 1)) #Real
    image shape and not an array of 784 elements
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
    input_shape=(28, 28, 1)))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(100, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

```

The Conv2D layer takes an input shape of 28x28x1 (in this case, we are not

working in RGB images) and it will output a shape of 26x26x32. This is because a 32 filter and 3x3 kernel are applied. MaxPooling2D layer basically divides the matrices taking the maximum value for each "2x2 sub-matrix", so the output will be 13x13x32. At the end, Flatten will create a unique array and will be furthermore simplified with the use of the Dense.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2	(None, 13, 13, 32)	0
flatten_1 (Flatten)	(None, 5408)	0
dense_1 (Dense)	(None, 100)	540900
dense_2 (Dense)	(None, 10)	1010
Total params: 542,230		
Trainable params: 542,230		
Non-trainable params: 0		
Train on 60000 samples, validate on 10000 samples		
Epoch 1/20	60000/60000 [=====] - 7s 121us/step - loss: 0.2058 - acc: 0.9366 - val_loss: 0.0865 - val_acc: 0.9721	
Epoch 2/20	60000/60000 [=====] - 4s 63us/step - loss: 0.0594 - acc: 0.9822 - val_loss: 0.0506 - val_acc: 0.9836	
Epoch 3/20	60000/60000 [=====] - 4s 59us/step - loss: 0.0359 - acc: 0.9887 - val_loss: 0.0515 - val_acc: 0.9826	
Epoch 4/20	60000/60000 [=====] - 3s 55us/step - loss: 0.0227 - acc: 0.9930 - val_loss: 0.0407 - val_acc: 0.9863	
Epoch 5/20	60000/60000 [=====] - 3s 52us/step - loss: 0.0152 - acc: 0.9952 - val_loss: 0.0431 - val_acc: 0.9869	
Epoch 6/20	60000/60000 [=====] - 3s 57us/step - loss: 0.0099 - acc: 0.9970 - val_loss: 0.0490 - val_acc: 0.9872	
Epoch 7/20	60000/60000 [=====] - 3s 57us/step - loss: 0.0067 - acc: 0.9980 - val_loss: 0.0500 - val_acc: 0.9864	
Epoch 8/20	60000/60000 [=====] - 3s 58us/step - loss: 0.0043 - acc: 0.9989 - val_loss: 0.0554 - val_acc: 0.9859	
Epoch 9/20	60000/60000 [=====] - 3s 57us/step - loss: 0.0031 - acc: 0.9993 - val_loss: 0.0602 - val_acc: 0.9864	
Epoch 10/20	60000/60000 [=====] - 3s 55us/step - loss: 0.0022 - acc: 0.9994 - val_loss: 0.0643 - val_acc: 0.9855	
Epoch 11/20	60000/60000 [=====] - 4s 67us/step - loss: 0.0016 - acc: 0.9996 - val_loss: 0.0672 - val_acc: 0.9861	
Epoch 12/20	60000/60000 [=====] - 3s 50us/step - loss: 0.0012 - acc: 0.9997 - val_loss: 0.0622 - val_acc: 0.9873	
Epoch 13/20	60000/60000 [=====] - 3s 51us/step - loss: 9.9142e-04 - acc: 0.9998 - val_loss: 0.0704 - val_acc: 0.9865	
Epoch 14/20	60000/60000 [=====] - 3s 53us/step - loss: 9.3090e-04 - acc: 0.9998 - val_loss: 0.0791 - val_acc: 0.9860	
Epoch 15/20	60000/60000 [=====] - 3s 52us/step - loss: 6.0163e-04 - acc: 0.9999 - val_loss: 0.0789 - val_acc: 0.9873	
Epoch 16/20	60000/60000 [=====] - 3s 52us/step - loss: 4.8182e-04 - acc: 0.9999 - val_loss: 0.0721 - val_acc: 0.9871	
Epoch 17/20	60000/60000 [=====] - 3s 55us/step - loss: 5.4744e-04 - acc: 0.9999 - val_loss: 0.0748 - val_acc: 0.9868	
Epoch 18/20	60000/60000 [=====] - 3s 53us/step - loss: 3.7421e-04 - acc: 1.0000 - val_loss: 0.0887 - val_acc: 0.9848	
Epoch 19/20	60000/60000 [=====] - 4s 72us/step - loss: 3.8477e-04 - acc: 1.0000 - val_loss: 0.0796 - val_acc: 0.9872	
Epoch 20/20	60000/60000 [=====] - 3s 56us/step - loss: 3.4484e-04 - acc: 1.0000 - val_loss: 0.0795 - val_acc: 0.9873	
('Test loss:', 0.07945174906843629)		
('Test accuracy:', 0.9873)		
guido@pc097070:~/Guido\$		

Figure 4.19: Keras CNN Output

4.5.3 Recurrent Neural Network

The RNN are used to process sequences of data. To classify images using a recurrent neural network, we consider every image row as a sequence of pixels. Because MNIST image shape is 28x28, we will then handle 28 sequences of 28 steps for every sample.

Tensorflow based program is again similar to the others, only the model's function is changed:

```
def RNN(x, weights, biases):
    #Required shape: 'timesteps' tensors list of shape (batch_size,
    n_input) using unstack
    x = tf.unstack(x, timesteps, 1)
    #LSTM cell
    lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(n_hidden,
        forget_bias=1.0)
    #LSTM cell output
    outputs, states = tf.nn.static_rnn(lstm_cell, x,
        dtype=tf.float32)
    #Output
    return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

And this is the Keras program, where the LSTM hidden layers are added to the model:

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.LSTM(128,
    input_shape=(x_train.shape[1:]),
    activation='relu', return_sequences=True))
model.add(tf.keras.layers.LSTM(128, activation='relu'))
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

The first LSTM layer has 128 output units and the input shape is a 28x28 matrix. Using the **return sequences** argument, we are telling to the layer to return the output at each time step instead of the final time step. So it will study a sequence of 28 arrays.

The output of each unit will be fed into the second layer which has again 128 output units, but in this case they will elaborate just the input of a single sequence.

```

Layer (type)                 Output Shape                 Param #
-----
lstm_1 (LSTM)                (None, 28, 128)            80384
lstm_2 (LSTM)                (None, 128)                131584
dense_1 (Dense)              (None, 32)                 4128
dense_2 (Dense)              (None, 10)                 330
-----
Total params: 216,426
Trainable params: 216,426
Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 56s 930us/step - loss: 1.0736 - acc: 0.6561 - val_loss: 0.3066 - val_acc: 0.9133
Epoch 2/10
60000/60000 [=====] - 50s 825us/step - loss: 0.2424 - acc: 0.9251 - val_loss: 0.2363 - val_acc: 0.9250
Epoch 3/10
60000/60000 [=====] - 49s 821us/step - loss: 0.1368 - acc: 0.9586 - val_loss: 0.1059 - val_acc: 0.9664
Epoch 4/10
60000/60000 [=====] - 48s 801us/step - loss: 0.0945 - acc: 0.9714 - val_loss: 0.0868 - val_acc: 0.9759
Epoch 5/10
60000/60000 [=====] - 49s 823us/step - loss: 0.0751 - acc: 0.9778 - val_loss: 0.0605 - val_acc: 0.9808
Epoch 6/10
60000/60000 [=====] - 50s 829us/step - loss: 0.0618 - acc: 0.9817 - val_loss: 0.0933 - val_acc: 0.9741
Epoch 7/10
60000/60000 [=====] - 49s 821us/step - loss: 0.0541 - acc: 0.9844 - val_loss: 0.0468 - val_acc: 0.9861
Epoch 8/10
60000/60000 [=====] - 50s 828us/step - loss: 0.0473 - acc: 0.9864 - val_loss: 0.0434 - val_acc: 0.9889
Epoch 9/10
60000/60000 [=====] - 49s 824us/step - loss: 0.0417 - acc: 0.9880 - val_loss: 0.0428 - val_acc: 0.9876
Epoch 10/10
60000/60000 [=====] - 49s 810us/step - loss: 0.0363 - acc: 0.9892 - val_loss: 0.0470 - val_acc: 0.9873
('Test loss:', 0.04698368611245314)
('Test accuracy:', 0.9873)
guido@pc097070:~/Guido$

```

Figure 4.20: Keras RNN Output

4.5.4 Transfer Learning with pre-trained model

Transfer Learning is a popular method which allows to build accurate models in a timesaving way: instead of creating a brand new model to resolve the problem, you will start from model patterns already created by other people that solve a different problem than yours. [21]

Transfer Learning is expressed using some pre-trained models. They are weighted models that were trained on a large dataset in order to solve a problem similar to the one we want to solve. Most of the pre-trained models belong to the CNN section of Deep Learning and some typical pre-trained models are **VGG**, **Inception**, **ResNet**.

I tried to implement the VGG16 pre-trained model in a different image classification problem. The VGG16 is a CNN model trained on a datasets of million images belonging to thousand classes, called ImageNet.

The dataset I am using is a big one, it includes 10.000 images belonging to 149 different classes, but this means that the pre-trained model will probably over-fit during the training.

Keras includes many pre-trained model like the VGG16. Instantiated the model,

we can take a look to its complexity using the summary function:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
Total params: 14,714,688		
Trainable params: 14,714,688		
Non-trainable params: 0		

Figure 4.21: VGG16 model's layers

```

vgg_model = tf.keras.applications.VGG16(weights='imagenet',
    include_top=False, input_shape = (224,224,3))
for layer in vgg_model.layers[:-5]:
    layer.trainable=False
es = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
    patience=50, mode='auto', min_delta=0.01)

model = tf.keras.models.Sequential()
model.add(vgg_model)
model.add(tf.keras.layers.Flatten(input_shape=vgg_model.output_shape[1:]))
model.add(tf.keras.layers.Dropout(0.8))
model.add(tf.keras.layers.Dense(256, activation='relu',
    kernel_regularizer=tf.keras.regularizers.l2(l=0.01)))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(149, activation='softmax'))

```

```
model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.0001,
    decay=0.0001/200), loss='categorical_crossentropy',
    metrics=['accuracy'])
```

But the model's output will not be the same of the problem. It is appropriate to add some layers in order to get the output needed: for example, the last Dense layer will oblige the model to classify the input into 149 different classes.

The 256 Dense layer is used to add some **Dropout** between the layers. The best Dropout chosen is 0.8 in the first one and 0.5 in the second one. Dropout is the probability p of a neuron being dropped out during each training stage and it can be seen as a regularization technique. Dropout is important in this case because, at the first try, without any regularization technique, the model was significantly over-fitting: the training accuracy arrived almost at 100% but the validation accuracy got less than 50%. The meaning is that the program achieve a good fit on the training data, while it does not generalize well on unseen data.

But Dropout is not enough to relieve the over-fitting problem. Another ploy can be freezing some weights of the pre-trained model because it could be too deep for the studied case. Here, after many tests, freezing the last 5 VGG's layers seems to be the best choice to improve the validation accuracy.

Another simple regularization technique is the **L2 regularization**, shown in the SVM section. It adds a cost, resulting in smaller weights, in the the Dense layer right after the pre-trained model.[22]

The **Early Stopping callback** is used to stop the training after a certain number of epochs where the accuracy is not improving. It can be helpful to find the correct place where the best performance is happening before entering in over-fitting.

Defined the model, what remains to be done is compiling it. Adam is the chosen optimizer, but the learning rate and decay are modified so as the over-fitting will relieve.

```
train=
    tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255,
        shear_range=0.2, zoom_range=0.2, horizontal_flip=True)
test=
    tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
training_set =
    train.flow_from_directory('datasets/generation/train',
        target_size=(224,224), class_mode = 'categorical')
val_set = test.flow_from_directory('datasets/generation/test',
    target_size=(224,224), class_mode = 'categorical')
```

The GPU libraries does not include Sklearn, so it is impossible to use the split-

ting function. Keras offers a class for images manipulation thought: the **ImageDataGenerator** which it has the **flow_from_directory** method which directly recognizes different classes inside a directory. I had to use my own Python script to split the dataset inside the train and test directories.

The training set is built using a sort of **Data Augmentation**. Usually, Data Augmentation means augmenting the training data using modified images, such as zoomed ones, flipped, negative colors, etc. The ImageDataGenerator does not increase the number of images but replaces them with the modified one. Here, it randomly zooms, shears and horizontal flips some of them.

Another problem of the Sklearn library lack is that the dataset is very imbalanced and it is impossible to solve it automatically. The different 149 classes contains an huge difference in the number of images, where some classes have just a few pictures. This creates two problems:

- We don't get optimized results for the class which is unbalanced as the model never gets sufficient look at the class.
- It creates a problem of making a validation as its difficult to have representation of the classes in case the number of observation for some classes is extremely poor (over-fitting).

The approaches that Sklearn could offer easily were 2: [23]

- **Undersampling:** randomly deletes images from the class which has too many observations. This approach is really simple but there is a possibility that the data that we are deleting may contain important information about the class.
- **Oversampling:** for the poor class, randomly increases the number of observations which are just modified copies of existing samples. This gives a sufficient number of samples to play with, but it may lead to over-fitting.

```
history = model.fit_generator(training_set, steps_per_epoch = 64,
                             epochs = 500, validation_data = val_set, validation_steps = 64,
                             callbacks=[es])

test_image =
    tf.keras.preprocessing.image.load_img('datasets/starter/val/attempt.png',
    target_size=(224, 224))
test_image = tf.keras.preprocessing.image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = model.predict(test_image)
print(training_set.class_indices)
print(result)
```

A cartoon illustration of Charmander, a Fire-type Pokémon. It is depicted in a three-quarter view, facing towards the right. Charmander has an orange body with a lighter yellow-orange belly. It wears a blue scarf around its neck. Its eyes are large and blue. A flame of red and yellow fire is at the end of its tail. The background is a standard transparent checkerboard pattern.

Figure 4.22: The new observation

[illegible]

Figure 4.23: Model's output

The model is getting a very good 95% on training accuracy and 75% in validation accuracy. This means that the model is still over-fitting, but it has improved by 25%! The real challenge of Machine Learning is to improve the validation accuracy, so the performance of the model in predicting new observations. And this is very huge improvement.

The **predict** method outputs an array where each index belongs to a class of the dataset, associated with the probability that the new observation belongs to that class. Here, the indices with their label are shown randomly using the **class indices** attribute of the ImageDataGenerator object. The new image's label is "Charmander" and, looking to the indices, it would be the number 14. Knowing that the 0 index is also included, the program is sure that the image belongs to the correct class, so the probability is 1.

```
N = es.stopped_epoch+1 if es.stopped_epoch != 0 else 500
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0,N), history.history["loss"],
         label="train_loss")
plt.plot(np.arange(0,N), history.history["val_loss"],
         label="val_loss")
plt.plot(np.arange(0,N), history.history["acc"], label="train_acc")
plt.plot(np.arange(0,N), history.history["val_acc"],
         label="val_acc")
plt.title("Training loss and accuracy on dataset")
plt.xlabel("Epoch")
plt.ylabel("Loss/Acc")
plt.legend(loc="lower left")
plt.savefig("plot.png")
```

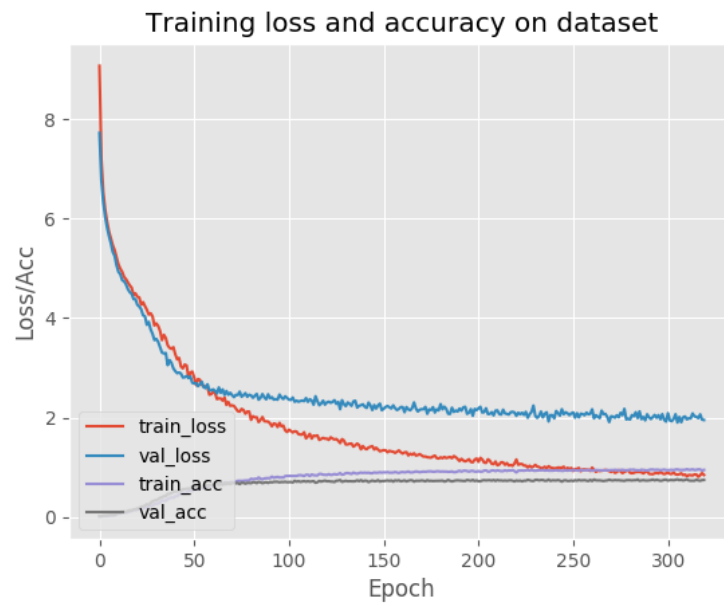


Figure 4.24: Accuracy and losses

This plot shows the trend accuracy and losses of the mode. It has to be saved in an external file and then imported in another computer, because the GPU machine can be accessed just using the command line.

Conclusion

English Version

In the previous chapters, I have explained what really is Machine Learning and Deep Learning, following methodologies which can be implemented in an high level programming language as Python. The models are many, but only some can fit the actual problem. At the end, the purpose is to get the best accuracy on predicting datas, resulting in a perfectly trained machine.

Using Python, learning the basis of Machine Learning is easy: the severe property of encapsulation allows the programmer to work with several functions that do not need to be changed. Anyway, Machine Learning can be implemented with many other languages, where the practitioner is much more free to create his own model, getting a better result overall. The best way to practice is trying with different languages.

My final example shows something: a machine can really recognize where our input belongs to, also with several classes, just as a person can do.

This branch of computer science has been ignored for many years because it needed too much computational time using the very old machines. Today, in contrast, a machine can be easily trained to perform this kind of recognition: a fact that has entered history is the one where a machine, trained using a combination of Supervised Machine Learning and Reinforcement Learning, could beat the world champion of Go, a traditional chinese board game [24]; similar thing is happening with Chess.

Basically, Machine Learning gets more powerful day by day, because of the continuous improvements in GPUs and CPUs, resulting effective in every matter. We have to wait to see what it still has in store for us.

Conclusioni

Versione Italiana

Nel capitolo precedente, ho spiegato, quindi, cos'è davvero il Machine Learning e il Deep Learning, proseguendo poi con metodologie che possono essere implementate in un linguaggio di alto livello come Python. I modelli da utilizzare sono tanti, ma solo alcuni sono precisi per il problema attualmente da risolvere. Alla fine, lo scopo diventa quello di ottenere la migliore precisione nella predizione di dati, ottenendo così una macchina perfettamente allenata.

Utilizzando Python, imparare le basi dell'Apprendimento Automatico è molto facile: la proprietà di incapsulamento permette al programmatore di lavorare con molte funzioni che non hanno bisogno di essere cambiate o riscritte. In ogni caso, gli algoritmi di Machine Learning possono essere implementati con molti altri linguaggi, dove il programmatore ha molta più libertà nella creazione di un suo modello, ottenendo un risultato migliore. Il miglior modo per fare pratica è provare con differenti linguaggi.

L'esempio finale prima delle conclusioni mostra qualcosa di molto importante: una macchina può veramente riconoscere a cosa appartengono determinati input, anche con differenti classi, proprio come un essere umano può fare.

Questa branca dell'informazione è stata ignorata per parecchi anni perchè necessitava di troppo tempo computazionale nelle vecchie macchine. Oggi invece una macchina può essere facilmente allenata per praticare queste tipologie di riconoscimenti: un fatto che è entrato nella storia è quello in cui una macchina, allenata utilizzando una combinazione di Supervised Machine Learning e Reinforcement Learning, è riuscita a battere il campione mondiale di Go, un gioco da tavolo tradizionale cinese; stessa cosa sta succedendo con gli scacchi.

In poche parole, il Machine Learning diventa via via sempre più potente, anche grazie alle continue migliorie fatte a componenti hardware come CPU e GPU, risultando efficace in qualsiasi materia di studio. Dobbiamo però aspettare per vedere cosa ha ancora in serbo in futuro.

References

- [1] M. J. Garbade, “Regression versus classification machine learning: What’s the difference?.” <https://medium.com/quick-code/regression-versus-classification-machine-learning-whats-the-difference-345c56dd15f7>, Aug 2018.
- [2] J. B. Ahire, “Machine learning: Real world applications.” <https://www.datasciencecentral.com/profiles/blogs/machine-learning-real-world-applications>.
- [3] Algorithmia, “Introduction to unsupervised learning.” <https://blog.algorithmia.com/introduction-to-unsupervised-learning/>, Sep 2019.
- [4] S. Priy, “Clustering in machine learning.” <https://www.geeksforgeeks.org/clustering-in-machine-learning/>, Feb 2018.
- [5] A. Pant, “Introduction to linear regression and polynomial regression.” <https://towardsdatascience.com/introduction-to-linear-regression-and-polynomial-regression-f8adc96f31cb>, Jan 2019.
- [6] A. Agarwal, “Polynomial regression.” <https://towardsdatascience.com/polynomial-regression-bbe8b9d97491>, Oct 2018.
- [7] S. Swaminathan, “Logistic regression - detailed overview.” <https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc>, Jan 2019.
- [8] J. Avila, *Scikit-learn Cookbook - Second Edition*. Packt Publishing, 2017.
- [9] P. Gupta, “Naive bayes in machine learning.” <https://towardsdatascience.com/naive-bayes-in-machine-learning-f49cc8f831b4>, Nov 2017.

- [10] P. Gupta, “Decision trees in machine learning.”
<https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052>, Nov 2017.
- [11] H. Deng, “An introduction to random forest.”
<https://towardsdatascience.com/random-forest-3a55c3aca46d>, Jul 2019.
- [12] W. Koehrsen, “An implementation of the random forest in python.”
<https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76>, Aug 2018.
- [13] Mathworks, “Support vector machine.” <https://jp.mathworks.com/discovery/support-vector-machine.html>.
- [14] “The difference between neural network l2 regularization and weight decay.” <https://jamesmccaffrey.wordpress.com/2019/05/09/the-difference-between-neural-network-l2-regularization-and-weight-decay/>, May 2019.
- [15] C. Lounge, “Support vector machine (svm) theory.”
<https://www.commonlounge.com/discussion/a49bcd907bdc4824ae53483c060f0259>.
- [16] S. Patel, “Svm - theory.”
<https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72>, May 2017.
- [17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2017.
- [18] T. Gupta, “Deep learning: Feedforward neural network.”
<https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7>, Dec 2018.
- [19] “Feedforward neural network.”
https://en.wikipedia.org/wiki/Feedforward_neural_network, Oct 2019.
- [20] D. Britz, “Recurrent neural networks tutorial, part 1 – introduction to rnns.”
<http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>, Jul 2016.

- [21] P. Marcelino, “Transfer learning from pre-trained models.”
<https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751>, Oct 2018.
- [22] B. Carremans, “Handling overfitting in deep learning models.”
<https://towardsdatascience.com/handling-overfitting-in-deep-learning-models-c760ee047c6e>, Jan 2019.
- [23] S. Chatterjee, “Deep learning unbalanced training data? solve it like this..”
<https://towardsdatascience.com/deep-learning-unbalanced-training-data-solve-it-like-this-6c528e9efea6>, Aug 2018.
- [24] F. Bo, “Go, ai, go!.”
<https://medium.com/fraglie-digitali/go-ai-go-be914d359921>, Sep 2019.
- [25] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2016.
- [26] J. Brownlee, “Logistic regression for machine learning.”
<https://machinelearningmastery.com/logistic-regression-for-machine-learning/>, Aug 2019.