University POLITEHNICA of Bucharest
Faculty of Automatic Control and Computers

Automatic Grade Prediction using Random Forest and Neural Networks

Author:
Marin Cristian-Emil

Scientific Coordinator:
ŞL.Dr.Ing. Popovici Dan-Matei

Bucharest
September 2016

Contents

Abstract

Keywords

1 Introduction **1**

1.1 Motivation 2

1.2 Contributions 2

2 Background **4**

2.1 State of the Art 4

2.1.1 A very short history of Machine Learning 4

2.1.2 Current interests in the field 5

2.1.3 Trends in educational learning 6

2.2 Related Work 6

3 Model Design and Methods Used **9**

3.1 Getting and Pre-processing the Data 9

3.1.1 Choosing the dataset 9

3.1.2 Dataset Structure 11

3.1.3 Data Standardization 13

3.2 Generating new features 14

3.2.1 The shape of a general dataset 14

3.2.2 Curse of Dimensionality 15

3.2.3 Adding Complexity to the Model 16

3.3 Visualizing the dataset 19

3.3.1 PCA 20

3.3.2 Looking at the data 20

3.4 The Models 22
3.4.1 Simple Models 23
3.4.2 Neural Networks 25
3.4.3 Random Forest 29

4 Implementation **32**
4.1 Parsing the students data 32
4.2 scikit-learn 34
4.3 Testing 37

5 Evaluation **38**
5.1 First Experiments 38
5.2 Neural Network and Random Forest Evaluation 41
5.3 Model Entropy 43

6 Conclusion **45**
6.1 Future Work 46

List of Figures **47**

Bibliography **48**

Abstract

In this project, we explore Neural Networks and Random Forests machine learning models, to build a tool for making predictions and analysis on data coming from an undergraduate Computer Science course. The main reason for this is to automate the performance evaluation of students during the semester and help them to get higher grades if it is probable to fail the final exam. Aditionally, we propose a general structure for the dataset, so this model can be used for any Computer Science course.

Keywords: grade prediction; machine learning; data analysis; neural networks; random forests; data classification; model entropy

Chapter 1

Introduction

Researchers in the last decades spent a lot of time developing tools and models for problems observed in nature and life. Their approach was purely analyitical and, given the fact that some systems behave in a very complex, non-linear way, these methods became harder and harder to build and understand.

But, with the recent advances in computational power, another approach was used to solve those scientific problems. This approach is oriented on empirical observations, being driven by the data, rather than the model and is called Machine Learning. The main purpose of it is to learn complex patterns and relations in data, without providing the rules explicitly.

The educational environment is a good place to apply these automatic models, since a lot of data is present here, it is hard to manually make inferences about the given datasets and a general overview of the structure of data is almost impossible to obtain without the help of intelligent systems.

In this report we are going to use a Machine Learning approach to solve the problem of automatic grade prediction by using two popular models, i.e. Neural Networks and Random Forest. The workflow which our project will use for building the model is show in the next diagram:

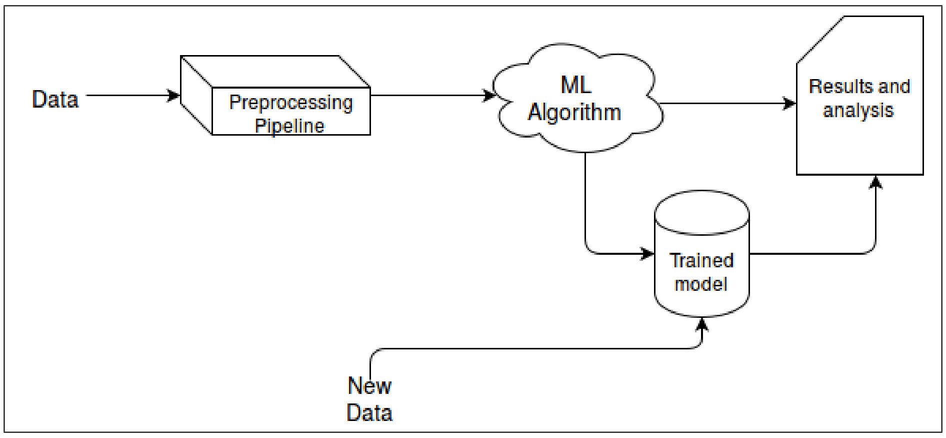

Figure 1.1: Workflow of a ML model

1.1 Motivation

Sometimes, evaluating the students performance in a course during the semester is found to be a difficult issue. The obstacles come from the fact that there are many students in an undergratuate class, and analyzing each of them is not so scalable, due to the limited number of course staff. Moreover, if the course is a difficult one and most students have problems passing it, we want to help these students get on track before taking the final exam and failing it.

For the problem stated above, a Machine Learning solution comes naturally, since it offers a way of classifying students and making predictions about their final grades. Also, with Machine Learning tools we can get insights about what grading components should we look at when we will want to assess the performance of a student before the course ending.

1.2 Contributions

The initial purpose of this project was to build an automated model for grading students based on the semester activity of each student. Later, this aim took the form of a model focused on getting important information from the data. Specifically, we wanted to use this model for helping students with a high probability of failing the course as early as possible during the semester.

The contributions of the project include:

- A research done on Machine Learning techniques and software frameworks appropiate for our use case
- A working model that proves the point of using Machine Learning for automatic grade predictions
- An evaluation of the model, with focus on educational analysis
- A generic way of using the developed model to other undergraduate courses related with the ones we used

Chapter 2

Background

2.1 State of the Art

2.1.1 A very short history of Machine Learning

"You have to know the past to understand the present." - Carl Sagan

The early beginnings of Machine Learning come not after the first electronic computers or after the first computer programs, as many may believe so. The fundamentals of this field took shape centuries ago with the discovery of the so-called Conditional Probability theories. **Bayes Theorem**, named after Thomas Bayes who first proposed a mathematical model for infering probabilities of conditioned events, and further developed by Pierre-Simon Laplace in his essay [1] in 1812, can be seen as one of the first models that can "learn" from given data and predict events based on correlated past events. Another old discovery that is the basis of today's regression models (e.g.: Linear Regression) is the "least squares method", credited to Carl Gauss, but first published by Adrien-Marie Legendre in 1805. This method was first applied in astronomy and allowed explorers to navigate oceans by aproximating the movement of celestial bodies.

Later on, in 1950, Alan Turing proposed in his paper[2] a *learning machine* that is able to learn and become intelligent and do well in the **Imitation Game** (now

[1] Théorie Analytique des Probabilités

[2] Turing, A.M. (1950). Computing machinery and intelligence. Mind, 59, 433-460.

generally called the **Turing Test**).

After this, the discovery of the Percetron and the **Neural Networks** around 1960s drew some attention in the field, but their current limitations had put Machine Learning on an impeding state for almost 10 years. It was only with the invention of the backpropagation algorithm in 1974 by Paul Werbos and the demonstration of its generalization by Geoffrey Hinton in 1986, that allowed it to be applied in multi-layered artificial neural networks. This also gave birth to a new sub-field of Machine Learning that today is called **Deep Learning.**

Along with the research in neural networks, some other models that were developed in that period are worth to mention. The most important ones are Support Vector Machines and kernels (models used for data classification and regression, that can be more time-efficient than neural networks[3] are and provide good performance from a data perspective) and Decision Trees. The latter, in combination with Ensemble Methods helped researchers invent models like **Random Forests** and **Adaptive Boosting** that are now state-of-the-art algorithms for tree models used for a lot of tasks.

2.1.2 Current interests in the field

Coming back to Deep Learning, which is today's main subject of interest of the Machine Learning community, it is a general approach that combines several state-of-the-art models of Machine Learning to solve problems such as image classification, AI for computer games, natural language, etc. Its constituents include neural networks with many hidden layers, convolutional networks, deep belief networks and recurrent networks. Also, the Q-Learning algorithm[4] and the Monte-Carlo search used in combination with convolutional networks allowed researchers to build semi-supervised learning programs that could learn to play computer games by themselves[5] or beat professional human players at games, such as Go (Google AlphaGo's program first beat Lee Sedol in October 2015).

[3]Some say that SVMs actually subsume Neural Networks, because of the flexibility of kernel functions

[4]Watkins, C.J.C.H. (1989). Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England

[5]https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf

2.1.3 Trends in educational learning

All advances in the Machine Learning field also conducted in an incresing interest in educational learning and assessment. With the name of **Educational Data Mining**, this newly emerging discipline deals with studying machine learning and data mining models in order to gain important knowledge about the structure of an educational system data (final grades, performance indicators, course drop-outs). Educational data can be taken from schools and universities (the classical way and also, the way that this thesis explores), online courses (such as MOOCs[6]), or even collaborative learning.

With the use of Machine Learning techniques, students can better decide on what courses they could take (based on past related grades), whether they have a chance or not to pass an exam before taking it and what indicators are relevant to their final asessment. The course department also benefits from the "learned" data because it helps them to better plan the structure of their courses, whether they are on-line or taken at the university.

2.2 Related Work

This part of the chapter will focus on the work on other people about the study on student performance prediction and analysis. Some of their research is similar to that of this thesis, and some others treat only related issues.

(Mehdi Sajjadi et al., 2016)[1] did some work on approximating final grades of students in a course on algorithms. In the grading process they used the peer grading method[7], and then applied Machine Learning (both supervised and usupervised) to aggregate those grades into a final grade. Their results were not so good compared to the simple method of just using the mean of all peer grades per an assessment as the final grade.

(Siddharth Reddy et al., 2015)[2] worked on developing a representational model of combined students and educational content (assessments and lessons). This

[6]Massive Open Online Courses

[7]A process in which students grade work of other students based on a given guideline

representation is actually a semantic space[8] and it can be used to study the relation between course content and students. Several conclusions can be drawn from these representations, such as: probability of passing an assessment or course and knowledge gained from completing a lesson. This article aimed mostly at MOOCs platforms, like Coursera, EdX and Khan Academy, and the model described was tested on synthetic student data and also, on real data from Knewton. Their model's results can be used to personalize the learning process of a course for each student in order to maximize the educational performance. Also, this model successfully predicted assessment results.

(Michael Wu, 2015)[3] wrote an interesting Master Thesis in which describes a Machine Learning Model that simulates MOOC data. Working with data gathered from EdX, the model once trained, can be able to synthesize student data. The model was trained to learn about student types, habits and difficulty of course materials. One of the main results of the thesis was being able to classify students in 20 important clusters.

(Saeed Hosseini Teshnizi and Sayyed Mohhamad Taghi Ayatollahi, 2015)[4] did a comparasion between Logistic Regression and ANNs[9] on a dataset composed of 275 undergraduate students and 16 student characteristics (e.g.: age, gender, parent education, employment status, place of residence, etc.) in order to predict academic failure. They concluded that the neural network models had a better accuracy than Logistic Regression (84.3% versus 77.5%). They tested 9 ANNs from which the one with 15 neurons in the hidden layer provided the best results, so ANNs methods were appropiate to be used in their problem.

Other references[5],[6],[7] also treat prediction of student academic performance mostly with neural networks and provide good accuracy with this model (the accuracy is also dependent of the structure of the dataset, number of examples, number of characteristics and noise in the data).

(Emaan Abdul Majeed and Khurum Nazir Junejo)[8] had some great results using Machine Learning models for predicting student's performance. They claim that they were capable of predicting the final grade with an accuracy of 96%. With a final number of 2500 student records and about 10 attributes for each record,

[8]Semantic similarity between objects represented as a kind of "metric" in space

[9]Artificial Neural Networks

they managed to predict the value of the final Grade attribute (which is a Class Variable that can have 6 values: A, B+, B, C+, C, Fail). Four classifier models were used in their study and we can see in Figure 2.1[8] their performance:

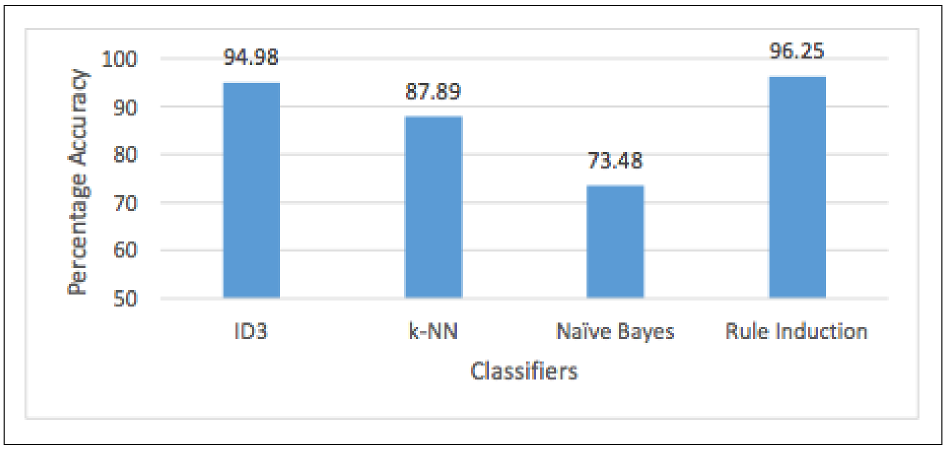

Figure 2.1: Classifiers accuracy

Some other last interesting references for this thesis[9],[10] used students online activity (course logs) to find patterns related to their overall performance. When applied to a MOOCs platform[10], the researchers found that the learners performace is strongly related to the number of videos played in the course platform, posts in course forum and, also, the total number of active days on that platform. With the use of SVMs, they achieved a 95% accuracy[10].

[10]They divided the learners in two classes: earning or not a certificate for the course

Chapter 3

Model Design and Methods Used

3.1 Getting and Pre-processing the Data

3.1.1 Choosing the dataset

First, when this project was in development, we only had access to a single students dataset, i.e. students performance grades from the Analysis of Algorithms course, which took place in the Fall 2015 semester. As progress was being made with the implementation, it became clear that the dataset could not be practically used, due to its high amount of noisy data. The noise was present in the data because of the small size of the dataset (137 students were taken into consideration) and also, the non-trivial distribution of final grades over the semester activity of students. The grading subjectivity of teaching assistants combined with a new experimental grading method for the course were also important factors in this negative result. Hence, with this much noise in the data, the models have not been able to correctly label the examples and, with more complicated models, there was a high tendency for *overfitting* the data.

Later, as the second semester was ending, we had access to another dataset of students and decided to switch to it. We also kept the students final grades for the first course to use them as attributes for the new dataset, in order to increase accuracy. In the 5^{th} chapter of this thesis, a detailed analysis will be made between these two datasets to show how they differ and what is wrong with the first one.

An important note must be made here. To increse the number of examples for our dataset, we could have combined those two above or could have aggregated students from different years into a single dataset but, there are two main reasons we did not do this: one, because the grading method and the number of semester assessments were different from one year to another and, second, because the two courses are conceptually different: one is theoretical, the other is more practical. Also, combining those two sets of examples would just have brought more noise to the second dataset.

Given this situation, the data that is currently used in this thesis comes from our Computer Science Undergraduate course, i.e. Programming Paradigms from 2nd year during the Spring 2016 semester. For that semester, we had a total record of almost 200 registered students, but we removed those whose attributes were missing or were not relevant (the ones that had a very low performance). Since we are interested in classifying students based on their exam grades and final grades, also implying the classification in *passed* and *failed* classes, we kept some students that had a poor grade during the semester and could not participate at the final exam, although they were close to that point. Having said this, we are left with a total of 143 relevant student records. These records must further be split in two sets: the training set - the set that is used for building the model and the testing set - the set used for testing the accuracy and measure the performance indicators of the model.

There is not a standard recipe of how to divide these two sets, but it is recommended that the training set should have a proportion between 60% and 80% of the total dataset's number of examples. For empirical reasons, the ratio of 80-to-20 percent was chosen for our dataset and the examples were sorted alphabetically based on the student's surname (the order is independent from the features and labels). The current dataset is not uniformly distributed over the *failed/passed* classes. There are 40% students that passed the course and 60% that failed it. This proportion is helpful in studying the prediction models (they must predict the failed students with a slightly higher accuracy). In more detail, from those 143 analyzed students, 57 of them had a positive grade (>=5) and 86 a negative one (<=4). From the 86 students that did not pass the course, 55 of them failed the final exam and 31 of them failed the course during the semester (and did not take part in the final examination process). The reason we kept those 31 students in the datased was to give the ML model more data examples, thus increasing the accuracy when predicting *failed* students.

3.1.2 Dataset Structure

To build the dataset, several sources of information were used. The first trivial one is the course semester activity of the students. The second, comes from performance of students achieved at past courses relevant to ours (chosen in different semesters in order to make them as independent from each other as possible), from a topic perspective. The courses that we got the grades from are:

- Computer Programming (PC): first year, first semester
- Data Structures (SD): first year, second semester
- Analysis of Algorithms (AA): second year, first semester

Lastly, the third subset of entire dataset was taken from our on-line e-learning course platform[1]. There, we had access to all the students activity logs recorded during the course progress. The logs were downloaded as *.csv* file format and then, aggregated in four categories for meaningful results.

In Table 3.1 there is a listing with all the attributes (features) belonging to the first subset of features - the course semester activity. There are a total of 10 initial features (will address later on this problem with the number of features). The weight characteristic of the features represents the actual maximum points that a student can get from that assignment (they are not equal, but are summing for a total of 6.0 semester points out of 10), and was not used for building the ML models.

Table 3.2 describes the third subset of our dataset's features: the Moodle logs. Activities including homework forum posts and views (x_1), active days (x_2), course resources opened (x_3) and total number of logs per user (x_4) were used along with the rest of features (together and separately) in the learning models.

Next, we need to provide the values used for labelling the examples. Since this thesis treats both classification and regression problems, different types of labels

[1]Moodle, the Open-source learning platform

Feature	Description	Type	Range	Weight
t_1	Test 1 grade	Float	0-10	0.25
t_2	Test 2 grade	Float	0-10	0.25
t_3	Test 3 grade	Float	0-10	0.25
t_4	Test 4 grade	Float	0-10	0.25
t_f	Final test grade	Float	0-10	1.0
hw_1	Homework 1 grade	Float	0-1	1.0
hw_2	Homework 2 grade	Float	0-1	1.0
hw_3	Homework 3 grade	Float	0-1	1.0
lab	Lab activity	Float	0-0.5	0.5
lecture	Lecture activity	Float	0-0.5	0.5

Table 3.1: Dataset Semester Attribute Details

must be present in the dataset. For classification, we use the final grade (real number between 0 and 10) and the exam grade as integer values. For regression, only the exam/final grade is used and it will be represented as a float number.

Table 3.3 gives a structured view of how the students dataset labels are used in the studied models. For classifying the students into 3 classes based on their exam points or final grade, the real continuous interval $(0, 10]$ was split into 3 different-length subintervals: $(0, 5)$, $[5, 7]$ and $(7, 10]$. For each subinterval there was assigned a class label: 0, 1 and 2, respectively.

Feature	Description	Type
x_1	f(forum_posts, forum_views)	Float
x_2	Number of active days	Float
x_3	Number of course resource views	Float
x_4	Total number of logs per user	Float

Table 3.2: Dataset Logs Attribute Details

The f function from the 3.2 table is actually a linear combination of the number of forum posts and forum views, as follows:

$$f(\#posted, \#viewed) = \#posted + \frac{\#viewed}{\alpha} \tag{3.1}$$

where α is the factor representing the *viewed* to *posted* ratio of all the homework forum logs (40 in our case). This was chosen because we are more interested in forum posts (as they are more relevant), so the forum views are adjusted with this factor to have a smaller value.

Problem type	Label Description		
	Label type	Label Value	Label used
Binary Classification	Integer	0|1	Exam/Final grade
3-class Classification	Integer	0|1|2	Exam/Final grade
Regression	Float	0.0-10.0	Exam/Final grade

Table 3.3: Label Structure

3.1.3 Data Standardization

In Machine Learning, standardization of data is a common requirement for a lot of models (e.g.: neural networks and SVMs). This means that, before applying our learning models, we must first scale the features from our dataset. This scaling implies that the data should have **mean** 0 and **standard deviation** 1. This is done by subtracting the mean value of each feature, then scale the data by dividing the features by their standard deviation (or variance, because standard deviation is the square root of variance, so they are equal).

$$X_{scaled} = \frac{X - \overline{X}}{\sigma} \tag{3.2}$$

where X is a vector of values from one feature, $\overline{X}$ is the mean of X and σ represents the standard deviation of X.

In Figure 3.1[2] we can see a straight-forward visualization of how the data is represented before and after the preprocessing techniques:

Data standardization is important if we are comparing features that have different ranges, but it is also necessary for some of the machine learning models. That

[2]http://cs231n.github.io/neural-networks-2

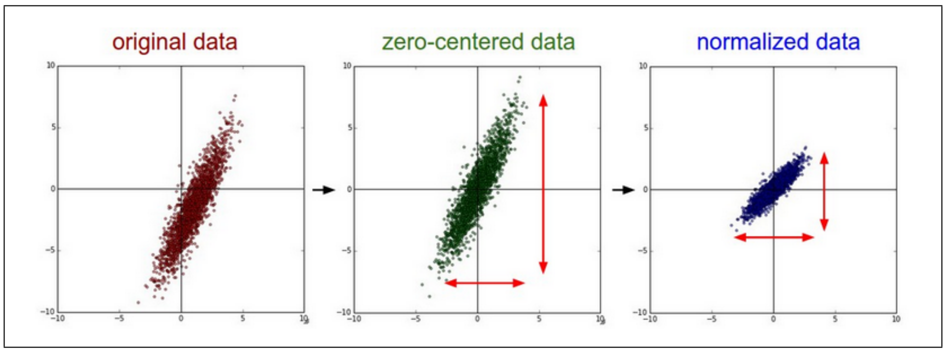

Figure 3.1: Data preprocessing pipeline

is because these models use the *gradient descent*[3] algorithm, which is sensitive to feature scaling.

It is worth noting that the mean and standard deviation values must come only from the training data and the transformation must be done on both training and testing sets, for meaningful results. The reason is that, in general, the built Machine Learning model is applied on unseen data (on real-time data), which is not available when the model is built. So, for accurate calculations on the model's performance and generalization, we must restrict the computation of mean and variance only on the training examples.

3.2 Generating new features

3.2.1 The shape of a general dataset

Because one of the purposes of this work is to apply the learning model on future datasets (from the same course or even from other courses), we must first generalize the model, i.e. find a general set of features. To do this, the dimension of the feature space must have a constant value along multiple datasets and each feature must have the same *meaning*. Therefore, we are going to divide our current feature space discussed in subsection 3.1.2 into several categories, based on their similarity. Besides the above stated purpose, an advantage of this feature modelling is to decrease the dimension of the feature space. Having a small, but

[3]an optimization algorithm that is used to find a local minimum of a cost function

meaningful dimension of the input, the learning models can sometimes perform better, mainly when the number of examples in the dataset is small. The reasons for this statement are described in the following subsection.

Since almost every course has a semester grading method based on homeworks, tests, lecture and lab activity, the split visible in table 3.4 comes natural. With this, the dataset now has a constant number of features: five. If the Moodle logs features (which also have a constant dimension regardless of the dataset used) are further added to this new feature space, there will be a total of nine features. This way, we can use every student dataset, transform it to have this structure and then, apply the model.

A disadvantage of this generalization is that we lose information about features that get aggregated together in a new one. Sometimes this is useful, for example when we want to see what homework or what test was the most relevant in the student's exam or final grade.

Feature	Description	Type	Range
hw_agg	Homework points	Float	0-1
t_agg	Test points	Float	0-10
past_agg	Past results score	Float	0-10
lab	Lab activity	Float	0-0.5
lecture	Lecture activity	Float	0-0.5

Table 3.4: Dataset Semester Aggregated Features

Note: the homework, test, and past results aggregated features are calculated as a weighted arithmetic mean between their components.

3.2.2 Curse of Dimensionality

In general, the number of examples and the dimensionality of each example (the number of features per example) are correlated, taking into consideration the accuracy of the trained model. The *Hughes phenomenon*[11] tells us that if we have a constant number of training examples, the ability of the model's prediction decreses when the dimensionality increses over an optimal value. This is also called the **Curse of Dimensionality** and it can lead to *overfitting* the dataset - the model has a low power of generalization. Finding the best number of features can be a very hard problem (as it requires a lot of manual testing). Actually, this is an *intractable* problem, because we need to generate all possible combinations of features and find the optimal one. This could easily be avoided now by using Feature Selection tools. For example, **Random Forests** are very good models at selecting features that provide the best accuracy to the model. This will be analyzed with more details in the next sections of the thesis.

So, supposing that we have M number of examples in the training set and the feature tensor is one-dimensional, if we add another dimension to the tensor (another feature), ideally, we need to square the training examples. By induction, the number of training example grows exponentially with the dimension of the feature tensor. The reason behind this, is that when adding more features to the dataset by keeping the same number of examples, the space where our examples are distributed becomes sparser. So, in order to keep the same sparseness of the space, we must fill the higher dimensional space with more data, and that data must grow exponentially as the dimension of the space increases. Keeping the same size of the dataset will result in overfitting the data, which is bad for a model.

Figure 3.2[4] shows a representation of how the number of feature dimensions affects the quality of the learning model. It can be seen that keeping the training examples constant and only increasing the number of features, the accuracy drops by an exponential rate. Finding the optimal dimension of the feature space requires a lot of work and testing - there is not an universal recipe that does this yet, but methods such as Feature Selection or Feature Extraction are a good place to start.

3.2.3 Adding Complexity to the Model

There are situations in which is better to add complexity to our data. For example, when our dataset in not linearly separable using one, two, or more features, we can add extra dimensions to the feature tensor in order to make that data separable.

[4]http://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/

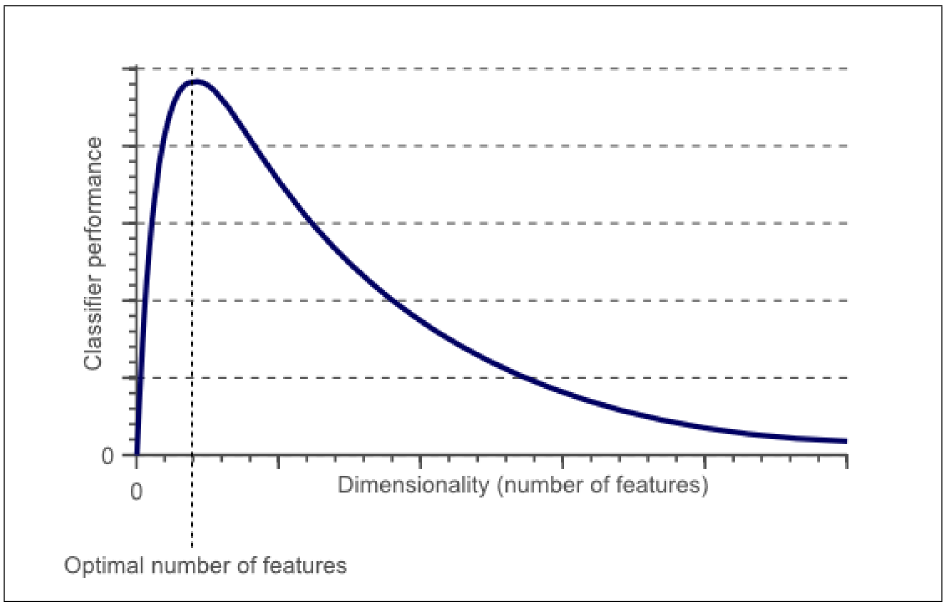

Figure 3.2: The Curse of Dimensionality

SVMs make use of this approach, by using multiple *kernel functions* that have the role to transform an input space in order to easily process data. Intuitively, a kernel is a "shortcut" that allows us to do certain computations, but without being directly involved in higher-dimensional calculations. Kernels can be linear functions, polynomial functions or even sigmoid functions. Using SVMs, we implicitly add complexity to our model.

Having tested some simple models on the above dataset, such as Linear Regression or the Perceptron, adding some complexity to the models was not at all a bad idea. A good method was the one of using **Polynomial Features**, that combines the initial dataset's features into new nonlinear features. Polynomial Features add more dimensions to the feature space, but the key here is that they are correlated, so this can help in achieving better prediction accuracy. Here, there are two different options to consider:

- The first one is generating a list of features (a polynomial of a certain degree) from the current features. Example: If we have the input given as (X_1, X_2), after the polynomial tranformation the example becomes $(1, X_1, X_2, X_1 * X_2, X_1^2, X_2^2)$
- The second approach was to consider only interactions between the features for building a polynomial with the same degree as the number of initial fea-

tures. Example: The features (X_1, X_2, X_3) are transformed by the polynomial into: $(1, X_1, X_2, X_3, X_1 * X_2, X_1 * X_3, X_2 * X_3, X_1 * X_2 * X_3)$, resulting in a total of 2^N final features, where N is the original dimension of the input space.

So, for Linear Regression, the input features are transformed using the first method (to generate non-linear functions like polynomials of degree 2 or 3). This "trick" allows us to use simple linear models that are trained on actual non-linear combinations of the data and are faster than other complex non-linear models. Supposing that we want to train our student's dataset using Linear Regression with Polynomial Features:

Let $\tilde{y}$ be the output vector of the linear model, x the input tensor, $\omega \in \mathbb{R}^{M \cdot N}$ the coefficient tensor (a two-dimensional vector) and β the vector bias. The model computes the following equation, making use of the "least squares" method for calculating ω and β:

$$\tilde{y}(\omega, x) = \omega \cdot x + \beta \tag{3.3}$$

where $x = (x_1, x_2, \ldots, x_9)$. When we add the polynomial features, x is transformed like this:

$$x_{nonlinear} = (x_1, x_2, \ldots, x_9, \ldots, x_i \cdot x_j, \ldots, x_1^2, x_2^2, \ldots, x_9^2); i, j = \overline{1, 9}, i < j$$

The size of the new feature space is: $9 + 9 + C_9^2 = 54$
Substituting $x_{nonlinear}$ in Equation 3.3, we obtain:

$$\tilde{y}(\omega', x_{nonlinear}) = \omega' \cdot x_{nonlinear} + \beta' \tag{3.4}$$

We can observe from the above equation that the linearity is still preserved and the model can fit more complicated data now.

On the other hand, the Perceptron model was tested using both methods, although the second method turned to be more appropiate, i.e. the *interaction features* method. This method was also used for the MLP[5] neural network model. With the size of the feature space of 9, adding interaction features provided a total of $2^9 = 512$ features.

Besides **Polynomial Features**, another way to add more features from existing features is to use **Trigonometric Features**, like $sin(x)$, $cos(x)$, etc. This can be useful when you only have two or three features in the dataset and want to increase the

[5]Multi-Layer Perceptron

input size a little more to see if the model (e.g. neural networks) can fit the data more precisely.

It is worth saying here that this approach of making the model more complicated is a good thing when having a big dataset of training examples. In our case, the training examples have a dimension of 80% of a total of 143 examples, which is approximately 115 examples. With this number, and with an input size of 9 features, generating polynomial features and increasing the input space to a much higher value does not scale very well, even if the features are correlated to each other: it will only make the data more complex and the model will give bad results, no matter how good it is, theoretically.

3.3 Visualizing the dataset

Machine Learning can be very counter-intuitive when we are dealing with the number of dimensions. Often, it implies working with hundreds, or even thousands of dimensions, and our human mind cannot reason easy about this (or not at all) when it comes to visualizing/imagining what is happening with the data or how it looks. Humans can think with no problem in two or three dimensions (even four, with some effort), so researchers in this field came up with some useful tools that do a *dimensionality reduction* of the data. This means transforming data from higher-dimension space to a human-meaningful lower-dimension space, with the purpose of having a view of how the dataset looks before trying to apply some Machine Learning models ot it. Also, another reasing for the reduction of the input space is to decrease the number of features in an optimal way (find the most important ones or create new fewer ones from the existing features), so the models can provide better results.

There are many tools that are used for the dimensionality reduction. Some examples are: **PCA**[6]**, MDS**[7] **and t-SNE**[8]. If visualization is what is needed, **Graph Based Visualizations** models are very helpful, as they give important insights into the structure of the data, i.e. how the points are connected to each other.

[6]Principal Component Analysis
[7]Multidimensional Scaling
[8]t-Distributed Stochastic Neighbor Embedding

3.3.1 PCA

For this work, the *PCA* technique was used to get information about our dataset and see how it looks, visually. PCA was not used for *feature extraction* purposes, since the aim was to focus on the models and current features and to get useful insights about them and their role in the student's performance.

What PCA does is to get the original input space and apply to it an orthogonal linear transformation in such a way that it will reduce the dimension of the input space and the new points will be spreaded the most in the new space. The reason behind this is to capture the most possible *variance* of the new data, so we can get a better look at them. Variance measures how much the data is distributed across the space, i.e. how far a set of points are spread out from their central point (thei mean value).

For example, considering the dataset fits in an *N-dimensional* hypercube, PCA will find the optimal angle to look at the data and then will project that data to two or three orthogonal axes, so it can be visualized. That angle will give the most variation in the data. However, the most variation does not always imply that the new dimensions can be used as new, alternative features for the learning algorithms, mostly when the dataset is not classifiable (it has a very high amount of noise).

3.3.2 Looking at the data

Since our dataset's feature space has a minimum dimension of 4 (when using just the Moodle logs features) and a maximum dimension of 17 (when using all the possible features - past results, semester grades and Moodle logs), PCA was applied multiple times for different subsets of the feature space with the goal of reducing its dimension. In this subsection, the dataset described in subsection 3.2.1 was used as an example.

In figure 3.3, it can be seen how the points are spread across the two principal axes in order to have the most variance. The first one, i.e. the horizontal one, represents the first principal component (with the highest variance) and the second one (the vertical axis) count for the second principal component (which has the second-

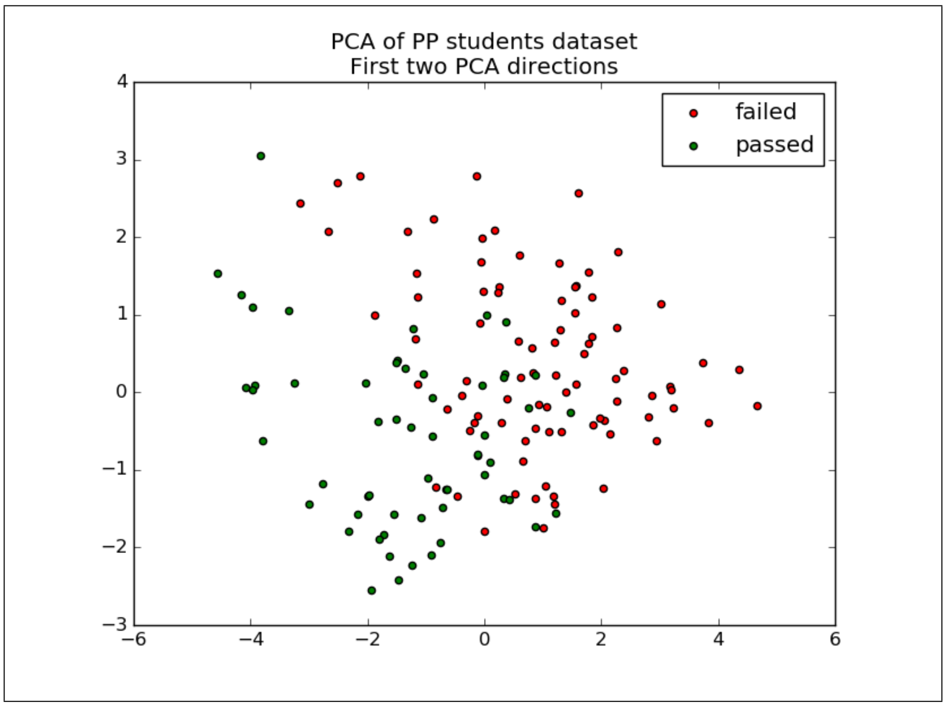

Figure 3.3: 2D PCA of the model

highest variance). After the plot was made, the points were colored based on their class membership. The red dots represent the students that failed the course and the green dots the ones that passed (for this example only the binary split of the examples was taken into consideration). We can see from the picture that the PCA algorithm manages to split the dataset without *knowing* about the label values. This is why PCA can be seen as an unsupervised learning model for classification that finds patterns in the data by its own.

Figure 3.4 show the same PCA analysis of the same dataset, but this time plotted in three dimensions, only to get another viewable perspective. The higher the number of dimensions, the more information the model has. So, using three instead of two dimensions for the PCA, we can get more insight into the structure of the data (e.g.: in three dimensions we can see if the data has a curvature, and this can be helpful).

Looking at the two visualizations made by the PCA analysis, some conclusions can be drawn about the dataset. First, we can observe that there is only a small amount of noise in the data, which is a good thing, because the models used will be able to classify students with a high accuracy. Second, PCA provides

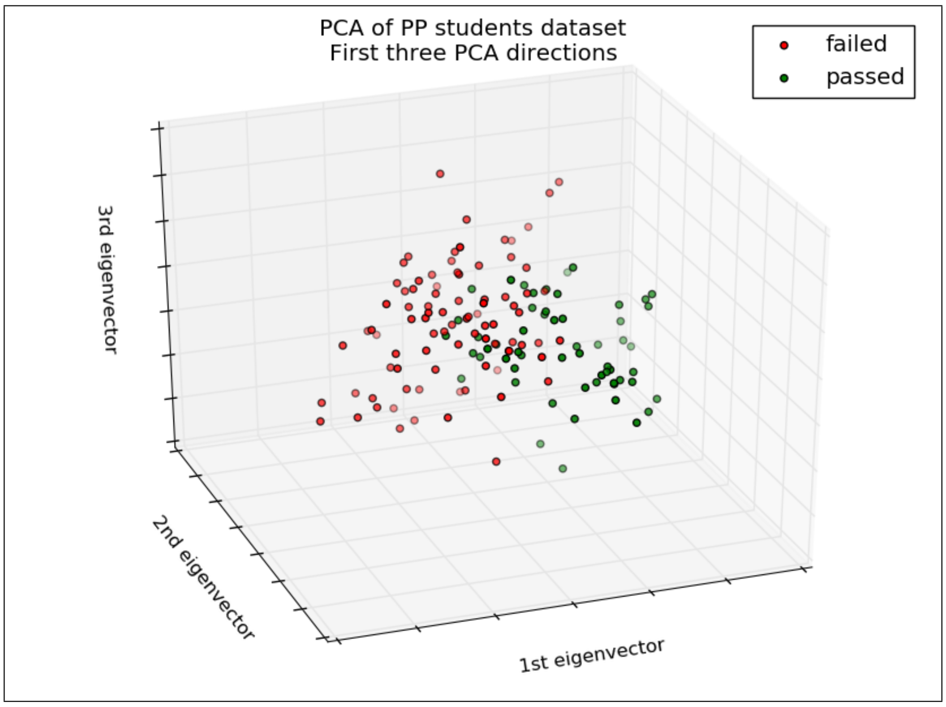

Figure 3.4: 3D PCA of the model

a safety check before going further with building the model (if we do not get a preliminary overview of our dataset, we will not know how well is expected from it to perform).

3.4 The Models

After building the dataset, which implied choosing the examples and the subsets of features, the next step is to choose a Machine Learning model (or more) to start training the data. The first part of this section discusses the design of two simple models (the *Perceptron* and the *Linear Regression* model), while the second subsection expands the architecture and components of a more complex model: *Artificial Neural Networks*. Finally, in the third part the *Random Forest* model is analyzed, with focus on its structure and goals.

3.4.1 Simple Models

The Perceptron

The perceptron is a model used for binary classification. The algorithm is linear: it combines the input vector of features with a vector of weights that are adapted in the learning process, based on the sign value (-1 or +1) of the product between the real class label and the sign value of the linear term $\omega \cdot x + b$. When this process ends, the model can decide if an input belongs to a class or the other. Figure 3.5 shows the model's simple architecture.

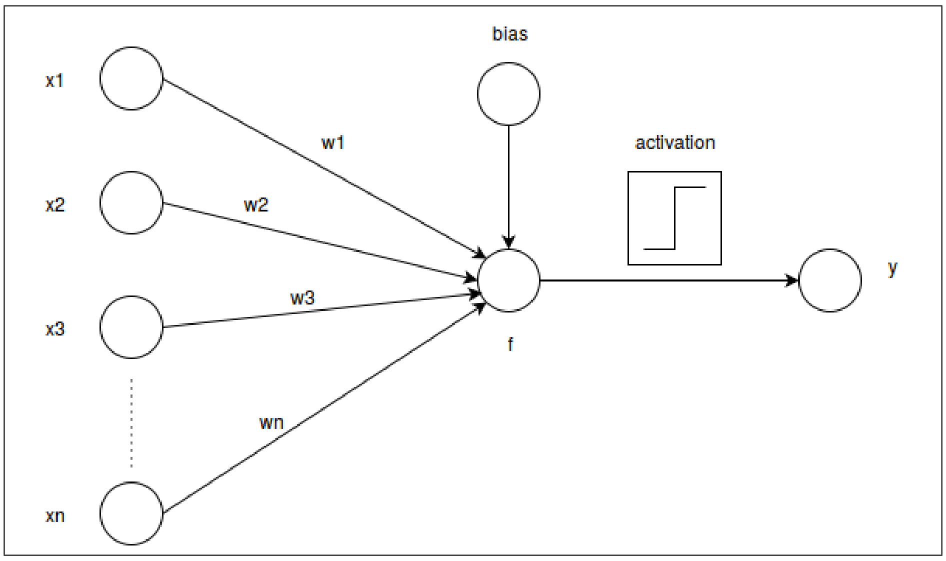

Figure 3.5: The Perceptron

This design was inspired by how a biological neuron works (it activates on certain learned thresholds) and it was the first step into developing *artificial neural networks*, which are a collection of perceptrons arranged in a layered network.

From the figure, we can see certain elements of the model: x and *omega* represents the input vector and the vector of weights, respectively; the *bias* has the role of adjusting the boundary position between the classes; f and the *activation* function represents the *perceptron algorithm* and y is its output, i.e. -1 or 1.

Linear Regression

The Linear Regression model uses the *ordinary least squares* method to solve an optimization problem that has the form:

$$min_{\omega}||\omega \cdot x - y||_2^2 \tag{3.5}$$

Starting from the standard equation $y = \omega \cdot x + \beta$, the method is used to find the ω and β parameters of the equation that satisfy the above optimization problem. The solutions are:

$$\omega = (x^T x)^{-1} x^T y; \ \beta = y - \omega \cdot x \tag{3.6}$$

Note: since this an optimization problem, *gradient descent* cand also be used to find its solution, but with an iterative approach, not analytically like the first method. The cost function used by this algorithm has a form similar to the optimization problem mentioned above in equation 3.5:

$$Loss(\omega) = \frac{1}{2}||\omega \cdot x - y||_2^2 \tag{3.7}$$

where $\frac{1}{2}$ is a factor used only as a convenient when calculating the first derivative of the cost function:

$$\frac{\partial Loss(\omega)}{\partial \omega_j} = \frac{\partial}{\partial \omega_j}\frac{1}{2}(\omega x - y)^2 = (\omega x - y)x_j, for\ all j = \overline{1, N} \tag{3.8}$$

where N is the input size of an example.

The algorithm uses this partial derivative to repeatedly update the weight vector ω, until convergence:

$$\omega_j := \omega_j - \alpha \frac{\partial}{\partial \omega_j} Loss(\omega),\ for\ every\ j. \tag{3.9}$$

where α is the learning rate of the algorithm.

Combining this with the *polynomial features* discussed in subsection 3.2.3, we can extend the model in order to perform better at fitting the data, while still preserving its linearity.

3.4.2 Neural Networks

Model Architecture

To define a **neural network**, first we have to start from the concept of the **perceptron**, discussed in the first part of this section. To remind, a perceptron takes inputs of a certain size and computes a single output, which has a binary value (0/1 or -1/1), thus being able to classify data in a linear approach.

When the dataset is too complicated to be linearly classified, the perceptron model is not capable of doing the job. Therefore, a more complex model had to be developed based on the latter, and that is **neural networks**. A neural network is a generalization of the perceptron model, that is built up by not one, but many single neurons (or perceptron) arranged in a network that has a particular structure. The network consist of layers of neurons (groups of interconnected neurons). The neurons from a layer are independent to each other and only connected to the neurons from the next and last layers.

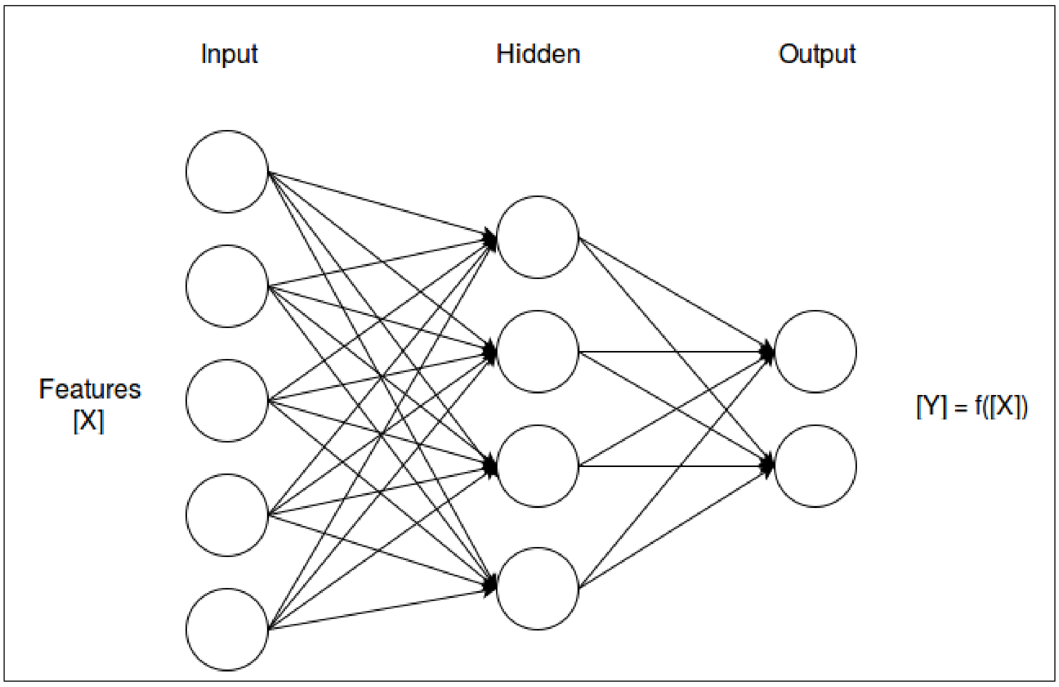

Figure 3.6: Neural network architecture

Figure 3.6 shows a generic architecture of a neural network[9]. In the first layer,

[9]Sometimes called a Multi-layer Perceptron

the input is given to the neural network. This network has a single layer of hidden units, but there are networks with more than one hidden layers, used at very complex computations, such as pattern recognition, image classification, and so on. The hidden layer in the figure makes four basic decisions based on the weights assigned to each edge between the neurons from the input layer and the ones from the hidden layer. The hidden layer and the output layer are also connected to each other, and another set of weights exists between them. The weights from each of the connections can also be seen as a weight matrix of size *number of hidden units* $\times$ *number of last layer size*.

With each of the hidden layers, the model learns new *representations* of data, by making transformations of the space topology in which the data resides. The reason for this is to make the dataset easier to classify in the last layer of the network (in the last representation of the dataset, the model may simply construct a line through it). This is why the neural network models are non-linear and capable of predicting very complex inputs.

Since this model is capable of learning non-linear functions from the data, after each layer (except the input layer) there must exist an activation function similar to the one used in the perceptron model, otherwise the model will still be linear, no matter the number of hidden layers. There exists multiple activation functions for neural networks, but the most popular ones are the *sigmoid function*, the *hyperbolic tangent function* and the *rectifier function* (ReLU). They give different results based on the dataset and are chosen in the process of *cross-validation*[10].

For the learning part of this model, *gradient descent* is used. If the model is trained on large datasets (thousands or millions of examples), a different version of this algorithm is implemented and that is, *stochastic gradient descent*. The classic algorithm must load the entire dataset into memory at each iteration, and this is not scalable if the dataset is big. On the other hand, stochastic gradient descent applies the same computations of a random subset (*mini-batch*) of the data at each step. This gives it a probabilistic flavor (hence the word *stochastic*) and it means that it can either find the minimum faster, or it may never converge to the minimum, theoretically. In practice, the error is neglected, as the computed value

[10]The technique of running multiple models or variations of the same model to a given dataset, in order to choose the most performant one.

is very close to the real minimum.

Usage

In this, part we are going to discuss how the *neural networks* had been applied as part of the development of our model.

We used neural networks for both classifying our dataset in two and three classes and predicting a continuous value based on the labels of the dataset pointed in table 3.3. For the regression type of problem (real value prediction) a single neuron was used in the output layer of the network, with no activation function (or, equivalently, with the *identity function* as the activation function).

The classifier, however, used the *logistic function* for the binary classification to obtain values between 0 and 1. For this case, there is still only one neuron needed at the output layer. In general, for *n*-class classification, the output layer has *n* neurons, and each neuron provides as value the probability for the input example to belong to the class represented by that neuron. But, since the probabilities must sum up to 1, for binary classification we only need one neuron in the output layer, because the second probability value can be infered from the first one.

The 3-class classification uses the *softmax* function at the output layer:

$$softmax(y')_i = \frac{e^{y'_i}}{\sum_{k=1}^{K} e^{y'_k}} \tag{3.10}$$

where y' is the output of the network before passing to the *softmax* layer (raw output) and K is the total number of classes. i represents the i-th component of the class vector.

The *softmax* function also has the property of transforming an input of raw values into a probability distribution over the set of classes. So, when predicting a class label, the value with the highest probability is picked from all the class probability values.

For regression, our network uses the *Loss* function stated in equation 3.7 as the cost function, which it will try to minimize. On the other hand, the both classifiers use a different cost function, since they deal with probabilities this time. The cost function used is called the *Cross-Entropy* function and is defined as follows:

$$Loss(\omega) = \frac{1}{M}\sum_{m=1}^{M}\sum_{k=1}^{K}[y_k^{(m)}log_2(\frac{1}{y'_k}) + (1 - y_k^{(m)})log_2(\frac{1}{1 - y'_k})] \tag{3.11}$$

here, y' is the predicted probability value, y the desired output (encoded as a *1-of-k* vector[11]), K the number of classes, M the number of training examples and ω being represented as the tensor of parameters (weights and biases) of the network that will get updated during the training process in order to minimize this loss functions.

The architecture of our model for the 3-class classification problem is evidentiated in figure 3.7, as an example. The input layer has a size of 9 neurons (corresponding to our number of features), the output layer consists of 3 neurons (given that we are classifying the dataset in three categories) and the hidden layer has 10 neurons. We chose a single hidden layer because of the few number of training examples (if the model is too complex for the dataset, it would overfit the data and not generalize well).

The number of hidden layer units was decided based on some empirical rules[12]:

- The number of hidded neurons should be between the number of input size and the number of output size (typically the mean of those)
- The hidded layer size should not exceed twice the number of the input layer size
- The size of the training examples should an upper-bound for the hidden
- Choosing the size of the hidden layer as $sqrt(N_i \cdot N_o)$, where N_i is the input size and N_o is the output size can lead to good results[12]

The hidden layer architecture is different for the other classification problem and the regression one. However, the same number of layers was preserved (but the size of the layer was changed). Also, the output size has different values: two for the binary classification and one for the regression. Other parts of the model (input, algorithm, learning rate and other model parameters) remained unchanged.

[11]A vector which has 1 for the correct class component and zero in the rest of its components
[12]www.faqs.org/faqs/ai-faq/neural-nets/part3/section-10.html

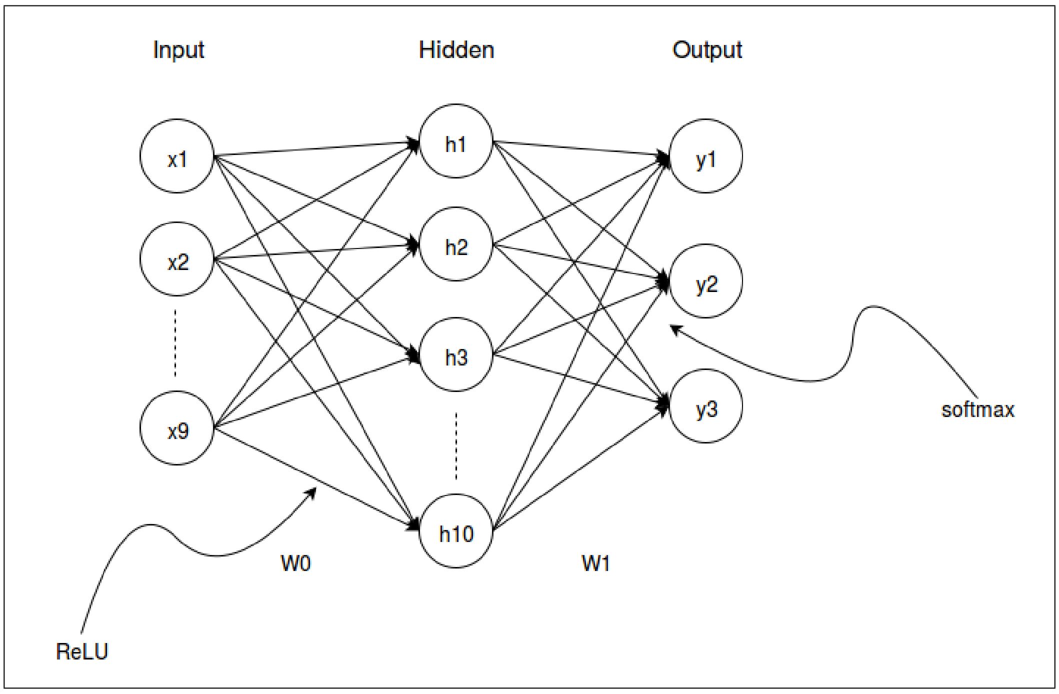

Figure 3.7: 3-class NN classifier with one hidden layer

Note: The learning algorithm used was SGD[13] with a learning rate value of $1E-3$.

3.4.3 Random Forest

Random Forests are probabilistic learning models based on the *Decision Trees* model. Decision Trees are simple tree structures that are used mainly for classifying data. The tree finds the best possible split of the dataset by assigning to each node a feature from the feature space. The feature selection is made by making an objective measurement of the "purity"[14] of the dataset (we select the feature that gives the best split of our data). Therefore, when the algorithm arrives at the leaves of the tree our dataset is classified, since each leaf contains the value of one of our classes. If we run the algorithm multiple times on the same dataset the results will be the same, because all the decisions taken by it are deterministic. Given this, the model is sensitive to small perturbations of the dataset, so it has the habit of overfitting.

To conquer the limitations of the above model, there was a need for a better one,

[13]Stochastic Gradient Descent

[14]The purity is measured by entropy or by the *gini impurity function*

so Random Forests were invented. The Random Forest model extends the notion of Decision Trees, by using an different approach. Instead of building a single complicated deterministic tree, Random Forest uses a collection of simple trees (a forest of trees) generated from random subsets of the data. An overview on the structure of this model can be seen in figure 3.8 [15].

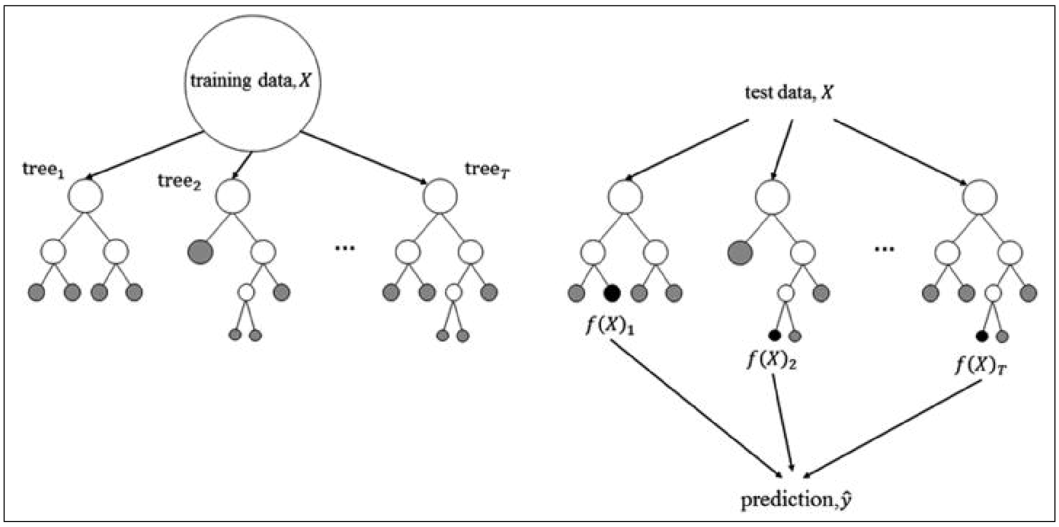

Figure 3.8: Random Forest model

Each tree is trained with a random subset of the whole dataset. At each node, a small subset of all the features is chosen and the split will be made like it is made in the Decision Tree model (a feature will be selected from that random subset that will best split the data). The size of the feature subset is usually chosen as the square root of the total feature size. At the end of each random tree a table with the frequencies of all the labels will be kept in each leaf, based on the partitioning of the training samples. When testing new data with this model, the input goes through all the trees in the forest and each tree will provide a frequency list of the labels. By a majority voting method (or by averaging[16]), the model will choose the class that has the highest probability.

Having this structure, Random Forests are good for reducing the *variance* [17] of the model, by averaging results from multiple random decision trees.

[15]https://www.researchgate.net/figure/278677730_fig2_Conceptual-diagram-of-the-RANDOM-FOREST-algorithm-On-the-left-trees-are-trained

[16]Depending on the implementation, for example, *scikit-learn* uses the averaging method

[17]A measure of how the model behaves on small fluctuations of the data

Besides making predictions, Random Forest models also provide information about the importance of each feature in the prediction process. This is because each node in each tree works with a small random subset of all the features and selects the one that gives the best split. So, features that are chosen at the top of a tree are considered more important because they contribute to a bigger set of training examples than the ones from the bottom. At the end, each of the trees will provide a list of the most important features (the ones that were selected for the splits) and then, a *feature ranking* can be computed by aggregating all these local ranks.

Usage

We used the Random Forest model in this thesis mainly for getting information about our features and how important they are for our model, and also, for making predictions (and comparing the performance with the Neural Network model). The feature importances helps us to see what features, i.e. what performance metrics are most relevant when performing an automatic evaluation for a student, so the staff of the course can work on improving the asessments that were not so relevant is the student's final examination grade.

Also, this can be used as a feature selection tool: the Random Forest model can provide a list of the most relevant features and then, another Machine Learning model can use this selected features as input. The outcome is that the model will give more accurate results and will have a better *running time*, mostly when we are dealing with a large datased and a complex learning model.

For the thesis, a Random Forest model with 10 estimators (random decision trees) was used for all the classification and regression approaches. The number of maximum features to select when buiding a node and finding the best split is $sqrt(N)$ for the classifiers, where N is the size of the total feature space of the model. When using regression, the maximum features to consider is N.

Chapter 4

Implementation

This chapter describes the design and implementation of the models expanded in Chapter 3, from a more technical perspective. The structure of the dataset and also short code listings will be used to show the logic of the implementation.

In the first section of this chapter, we are going to see the approach taken to gather the dataset and parse it to our *Python* program. The user can choose which subset of the features it wants to use in order to apply the ML models and get the desired results.

Next, we are going to focus on the *scikit-learn Python* library, which is an open-source Machine Learning library with many implemented features. Here, we will show how we used our chosen models with the help of this library.

At the end of the chapter we briefly discuss how the whole implementation was tested with data that did not take part in the training process.

4.1 Parsing the students data

All the data used in this project was taken from the course catalog and from the **Moodle** platform. The gathered data was put into `.csv` files, one for the semester grades and one for the logs taken from the on-line course platform.

The first line of each `.csv` file holds the names for the values found in the next line. Each further line contains the key string of the example along with the feature values and label values that were measured for it. An example taken from the *grades* file is given in the following listing:

```
id,h1,t1,t2,h2,t3,h3,t4,tf,lab,lecture,PC,AA,SD,exam,final
xxxxxxxx,0.9375,5,2.5,1,7.5,0,5.63,0.23,0.5,0,6,4,5,5,5.18
```

The second file completes the input features of the first one, with the mention that the final `.csv` file was build by a *python* script, because we had to aggregate all the measurements present in multiple `.csv` files into a single file, with the same structure and order as the one showed above. So, the second file contains the course platform log features that were described in table 3.2.

For loading the dataset into memory, we used a source file called `data_loader.py`. Here, all the data from both `.csv` file is loaded into a `python dict()` data structure. Then, using the options defined visible in the code listing example 4.1, we can choose which subset of our total extracted features are we going to use.

Listing 4.1: Parsing the data

```
class Options(object):
    GRADES_ONLY = 'grades_only'
    AGG_GRADES_ONLY = 'agg_grades_only'
    ALL_FEATURES = 'all_features'
    ALL_FEATURES_AGG = 'all_features_agg'
    ALL_LOGS = 'all_logs'

    def __init__(self):
        pass

students_data = load_data()

option = Options.ALL_FEATURES_AGG
data = get_data(option, students_dataset)
data = preprocess_data(data, poly_features=False)
```

In the above code, the desired dataset is loaded into a `DatasetContainer` class with the `get_data(opt, dataset)` function. The `load_data` function reads the content of the `.csv` files and puts it into a dictionary. After this, the current dataset is preprocessed (scaled and added polynomial features to it, if desired) by the `preprocess_data` function. Also, in this function, we implemented the PCA analysis of the dataset.

4.2 scikit-learn

The *scikit-learn* library is a free machine learning library implemented in the *Python* programming language, with some base algorithms written in *C*, for performance considerations. It is simple to use and has a wide range of tools for data analysis and machine learning projects. The backend of the library uses **Numpy, SciPy and matplotlib** packages developed for scientific purposes.

The library provides support for all the machine learning subfields, from classification to regression and clustering problems. Throughout the development of this project, we experimented with machine learning algorithms implemented in this library and also, tools for preprocessing and visualizing the data and the results of our models.

General Usage

For preprocessing we used the `StandardScaler()` object to scale our raw data, with the goal of having zero mean and unit variance of the data. Also, for generating our polynomial features based on the initial features, `PolynomialFeatures()` object from the `scikit-learn` library was used.

The *PCA* algorithm is also implemented in the library, so we used it to generate our two- and three-dimensional plots exposed in Chapter 3. For our classifier models evaluation, we needed to calculate and visualize the *confusion matrix* which evaluates the performance of the classifier (more details on this will be given in the following chapter). We used the `confusion_matrix` function from the library to calculate the matrix.

The *scikit-learn* library also has implementations for many evaluation metrics of the models, which can be found in the `sklearn.metrics` module. We used those metrics in our evaluation process and also for comparing the models.

Machine Learning Algorithms

The models discussed in section 3.4, i.e. the **Linear Regression, Perceptron, Neural Network and Random Forest** models were all used from the *scikit-learn* library. For the neural networks models (the classifier and the regressor), we had to use a *development* version of the library, since they are not currently implemented in the stable version.

In the code listing 4.2, a neural network binary classifier is used to train our dataset. The `model_eval` function evaluates the model based on training and testing data performance metrics.

Listing 4.2: Neural Network Classifier

```
from sklearn.neural_network import MLPClassifier

LABEL_NAMES_BIN = ['failed', 'passed']

def nn_binary_classifier(data):
    # Binary classification with neural networks.
    X_nn = data['train_data']
    X_nn_test = data['test_data']
    # use the exam grades as labels
    Y_nn = data['train_labels'][:,0]
    Y_nn_test = data['test_labels'][:,0]
    # Transform Y_nn and Y_nn_test for binary
    # classification
    Y_nn[Y_nn < 5] = 0
    Y_nn[Y_nn >= 5] = 1
    Y_nn_test[Y_nn_test < 5] = 0
    Y_nn_test[Y_nn_test >= 5] = 1
    # Use the MLP classifier
    clf = MLPClassifier(algorithm='sgd',
                        activation='relu',
                        hidden_layer_sizes=(10,),
                        max_iter=1000,
                        batch_size='auto',
                        learning_rate_init=0.001,)
    # Training process
    clf.fit(X_nn, Y_nn)
    # Evaluate the trained model
    model_eval(clf, X_nn, Y_nn, X_nn_test, Y_nn_test, LABEL_NAMES_BIN)
```

Using the random forest classfier only requires changing the `clf` variable in the code listed above to: `clf = RandomForestClassifier(n_estimators=10, max_features='sqrt')` and the rest of the code can remain unchanged (the library gives us a standard interface for the training algorithms, no matter which one is used).

If we want to see the feature ranking computed by the random forest model, we need to use the `clf.feature_importances_` attribute of the `clf` classifier object. Having this array of values - a probability distribution over the feature set, it is easy to plot the feature importances and their names on a graph, along with their standard deviations. The 4.3 code listing shows how we got the feature ranks from our trained model in order to plot them.

Listing 4.3: Plot Feature Ranks

```
importances = clf.feature_importances_
# Calculate the std dev of each feature
std = np.std([t.feature_importances_ for t in clf.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]
idx_names = list()
for i in range(X_train.shape[1]):
    idx_names.append(data['feature_names'][indices[i]])

# Print the feature ranking
print('Feature␣ranking:')
for f in range(X_train.shape[1]):
    print('%d.␣feature␣%d,␣name:␣%s,␣(%f)' % \
          (f + 1, indices[f], data['feature_names'][indices[f]],
           importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure()
plt.title('Feature␣importances')
plt.bar(range(X_train.shape[1]), importances[indices],
        color="r", yerr=std[indices], align="center")
plt.xticks(range(X_train.shape[1]), idx_names)
plt.xlim([-1, X_train.shape[1]])
plt.show()
```

4.3 Testing

The testing process was done after we trained each of the models. From the 4.2 code listing it can be seen that we have kept a subset of our entire dataset for the testing purposes (actually 20% of the dataset). This testing data was used to evaluate the model and assess its accuracy, i.e. how well the model behaves with new data.

Also, the implementation provides a testing source file named `test_models.py` that serves as a POC[1] for the functionality of the project. Here, our implemented models will be automatically loaded and trained and then, compared to each other. The test module will provide an easy command line interface that will allow the user to give as input to the program some examples and then get insights about them. The user will provide the features for their examples and will choose the subset of features and also the model that will be used for prediction.

[1]Proof-of-Concept

Chapter 5

Evaluation

In this chapter, we describe our approach to evaluating the models we used, with focus on performance, accuracy, features, as well as how confident we are in the answers given by the models. We are going to describe the results of our different subsets of the dataset used and also the poor results obtained from the first tried dataset that was described at the beginning of Chapter 3.

5.1 First Experiments

As we mentioned in the introduction of the first chapter, another dataset of students was used as a starting point. The main problem with this dataset was that it had a lot of noisy data, i.e. it was not clear to distinguish the failed students from the ones with higher grades. Thus, our trained models gave unrealiable results.

To get an image of how this first dataset looks like, we are going to use the same PCA analysis which was also done with the second dataset (the one that was used during this project). The next figures shows the 2D plots resulted from the Principal Component Analysis. In both figures, only the grade features were used (without the Moodle logs features, because we did not have at the time we tested the first dataset).

From the first figure we can observe that, althogh most the *passed* students are found in the left side of the plot, not much we can say about the structure of the

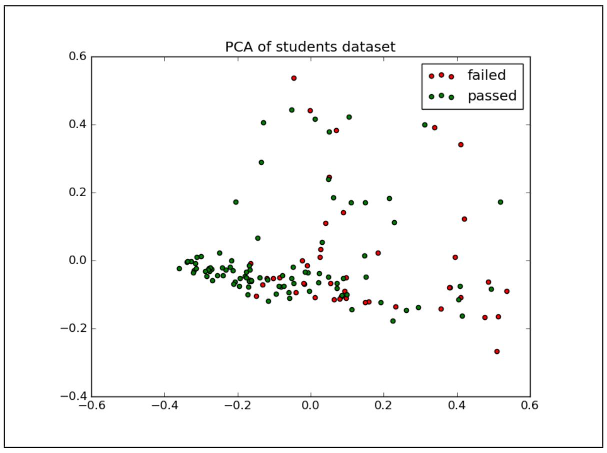

Figure 5.1: 2D PCA of AA dataset

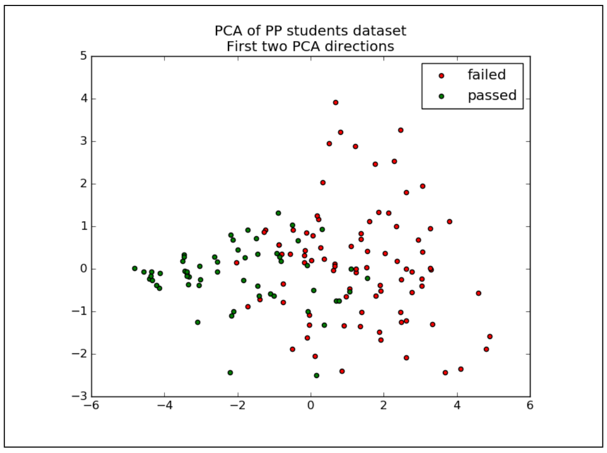

Figure 5.2: 2D PCA of PP dataset

red points, which represents the *failed* students. So, given this preliminary result, we can expect from the models to have a low accuracy when predicting the class of the *failed* students.

The second picture however, separates better the two classes, this being a sign that our models should classify the data with a lower error rate.

To show how the accuracy of the predictors differ between those two datasets, we will use the *confusion matrix* as a way of visualizing the prediction error of the model applied to each dataset. As a choice for the model, the two datasets will be trained using a neural network binary classifier. The confusion matrix shows on each column the number of examples predicted in each class, with the rows representing the examples in the true class.

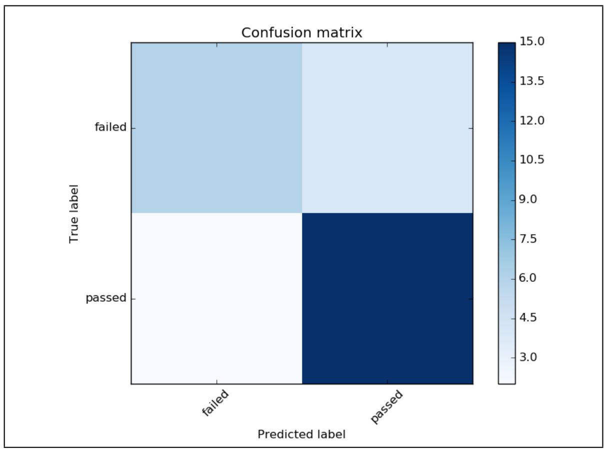

Figure 5.3: AA Confusion Matrix

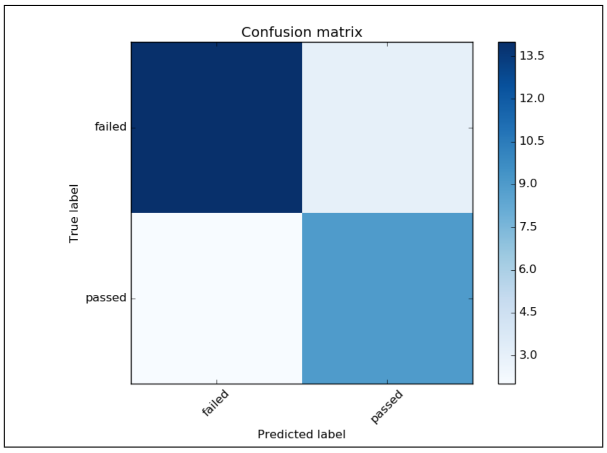

Figure 5.4: PP Confusion Matrix

The first confusion matrix shows in the first row that some of the *failed* students were incorrectly predicted as *passed* (this is expected after visualizing the PCA plot). In the second confusion matrix we can see that the model managed to better predict the *failed* students (which is mainly what we want), and also the *passed* ones, which are fewer.

The accuracy of the first dataset trained with the neural network was 68% on the testing set, while the second dataset gave an accuracy of 82% on this classifier model. The features used were the same that we used for the above PCAs.

5.2 Neural Network and Random Forest Evaluation

In this section we will perform a comparison between different subsets of our data and different learning algorithms on the two problems which we had treated: the classification and the regression problems.

Model		
Problem	Dataset features	Testing set Accuracy (%)
binary classification	only grades	82.14
binary classification	all features	85.71
binary classification	all features with aggregation	85.71
3-class classification	only grades	75.00
3-class classification	all features	78.57
3-class classification	all features with aggregation	78.57
regression	only grades	71.69
regression	all features	79.07
regression	all features with aggregation	80.29

Table 5.1: Neural networks results

In table 5.1 there are different variations of the model tested and we can see their accuracies. We can notice that the error rate of our predictions is lower when using more features. The neural network architecture is the one described in section 3.4.2 (the hidden layer size of the network was adapted considering the size of the input). Better results are achieved with the binary classification, while the regression problem gives the lowest performance (is harder to predict the value of the grade with a low error).

For the 3-class classification problem the confusion matrix is relevant. We can view from the 5.5 figure that our model is good at predicting very bad and very good grades. The problem comes when it tries to predict the grades in the "middle" (between 5 and 7), as it "thinks" they also count for *failed* examples mostly. Looking at the PCA plot from Chapter 3, it can be seen that there is a mix of *failed* and *passed* students in the middle of the plot, meaning the model will not be able to distinguish between those two and, given the fact that there are more *failed* examples in the dataset, the model will choose this class.

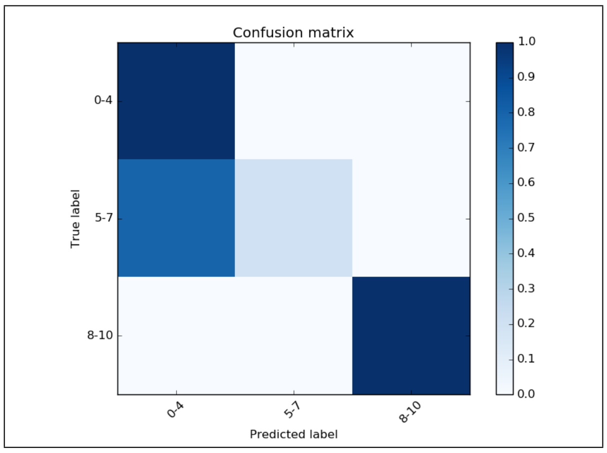

Figure 5.5: Confusion Matrix for 3 classes

The next table provides accuracy results for the Random Forest model.

Model		
Problem	Dataset features	Testing set Accuracy (%)
binary classification	only grades	78.57
binary classification	all features	78.57
binary classification	all features with aggregation	78.57
3-class classification	only grades	75.00
3-class classification	all features	71.42
3-class classification	all features with aggregation	78.57
regression	only grades	75.52
regression	all features	77.03
regression	all features with aggregation	78.17

Table 5.2: Random Forest results

The Random Forest has good results, regardless of the subset of features used. This is because the model chooses the best possible features when making decisions. So, the number of features does not matter here.

The last important result to mention in this section is the *feature ranking* provided by the Random Forest algorithm. In figure 5.6, the ranks of our chosen features are shown. We can notice that the first three features that give the best accuracy are the aggregated grades (taken from homeworks, tests and past results). Also, the lecture activity is also important, given this plot and the least important grade feature is the lab activity. The Moodle logs do not seem to provide high accuracy to the prediction model, but we can see that the first feature, i.e. the course forum activity feature has the greatest mean importance compared to the other three log features.

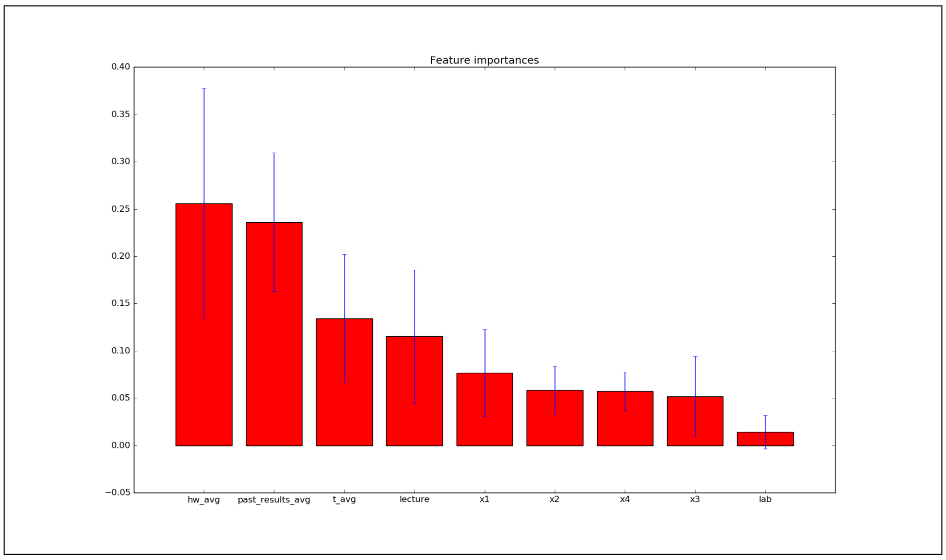

Figure 5.6: Feature ranking

5.3 Model Entropy

In the last section we will focus on the details of another metric that measures the performance of a machine learning model. When using classifiers, for example a binary classifier, our model will output a pair of probabilities for each input example that is given. Actually, this is a probability distribution for that input, consisting of two values for the random variable y_k.

If, for example, one of our input is labeled as *failed* with a probability of 0.8 and the correct label is also *failed* we want to see how good is that the model predicted the label correctly with that value.

When using the accuracy metric, we cared about the ratio of our correct answers. Here, on the contrary, we care about the confidence in the results that we got from the model. The metric used to measure this confidence is called the **KL**

Divergence[1] and is related to the **Cross-Entropy** function. This measure can be seen as a "distance" between two probability distribution. When applied to machine learning, the KL divergence measures the distance between the predicted probability distribution and the real distribution of the data.

Using our dataset, we achieved a KL divergence value of **0.32** with the neural network model and **0.41** for the random forest (the lower, the better, since the divergence is a sum over logarithms of probabilities). So, besides the high accuracy of prediction, there is also a high confidence that the prediction is correct.

[1]Kullback-Leibler divergence

Chapter 6

Conclusion

In this thesis, we described the approaches taken to adding machine learning models such as neural networks and random forest to automate the prediction of grades for students in a Computer Science class. We analyzed in detail the structure of our dataset and proposed a generic solution, that can be used in any Computer Science course. Then, we moved forward to discuss the chosen models and their mathematical formulations. In the last part of the report, we commented on the implementation of the project and then, evaluated the implementation with various metrics.

We believed that we managed to solve the problems and goals for this thesis. After analyzing our data and getting the results of the experiments used, we infered some valuable information about the course grading methods.

Then, choosing the best model for automatic learning is not a direct response. Neural networks can be slower but provide better results, whereas random forests are fast and give more information about the importance of features, but their predicting performance can be weaker.

We encountered some problems during the implementation of this project, but managed to find a proper solution with the little amount of data we got access to. There are also some ideas on improving this project, as pointed in the following section.

6.1 Future Work

Some next steps that we can take in the future with this work include:

- Getting more data from different years and different courses to increase the number of training examples and, implicitly, the performance of the model
- Implement an easy user interface for the model, so the course staff can benefit from this work
- Finding more relevant features that can be used in the prediction. An example can be analyzing social interactions between students or finding patterns in their course attendance (lectures and seminars)

List of Figures

1.1 Workflow of a ML model 2

2.1 Classifiers accuracy 8

3.1 Data preprocessing pipeline 14
3.2 The Curse of Dimensionality 17
3.3 2D PCA of the model 21
3.4 3D PCA of the model 22
3.5 The Perceptron 23
3.6 Neural network architecture 25
3.7 3-class NN classifier with one hidden layer 29
3.8 Random Forest model 30

5.1 2D PCA of PP dataset 39
5.2 2D PCA of PP dataset 39
5.3 AA Confusion Matrix 40
5.4 PP Confusion Matrix 40
5.5 Confusion Matrix for 3 classes 42
5.6 Feature ranking 43

Bibliography

[1] Mehdi S. M. Sajjadi, Morteza Alamgir, Ulrike von Luxburg. *Peer Grading in a Course on Algorithms and Data Structures: Machine Learning Algorithms do not Improve over Simple Baselines.* In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale,* pp. 369-378, 2016.

[2] Siddharth Reddy, Igor Labutov, Thorsten Joachims. *Learning Representations of Student Knowledge and Educational Content. Cornell University,* 2015.

[3] Michael Wu. *The Synthetic Student: A Machine Learning Model to Simulate MOOC Data,* MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015.

[4] Saeed Hosseini Teshnizi and Sayyed Mohhamad Taghi Ayatollahi. *A Comparison of Logistic Regression Model and Artificial Neural Networks in Predicting of Student's Academic Failure.* In *Acta Inform Med.,* 23(5): 296–300, 2015 Oct.

[5] Jeng-Fung Chen, Ho-Nien Hsieh and Quang Hung Do. *Predicting Student Academic Performance: A Comparison of Two Meta-Heuristic Algorithms Inspired by Cuckoo Birds for Training Neural Networks*. In *Algorithms,* 7(4), 538-553, 2014.

[6] B.A., Kalejaye, O., Folorunso and O.L., Usman. *Predicting Students' Grade Scores Using Training Functions of Artificial Neural Network*. In *Journal of Natural Sciences, Engineering and Technology, Volume 14,* 2015.

[7] V.O. Oladokun, Ph.D., A.T. Adebanjo, B.Sc., and O.E. Charles-Owaba, Ph.D. *Predicting Students' Academic Performance using Artificial Neural Network: A Case Study of an Engineering Course*. In *The Pacific Journal of Science and Technology, Volume 9,* pp. 72-79, 2008.

[8] Emaan Abdul Majeed and Khurum Nazir Junejo. *GRADE PREDICTION USING SUPERVISED MACHINE LEARNING TECHNIQUES.* In *E-Proceedin*

of the 4th Global Summit on Education, pp. 224-234, 2016.

[9] Amelia Zafra and Sebastián Ventura. *Predicting Student Grades in Learning Management Systems with Multiple Instance Genetic Programming*. In *Educational Data Mining*, 2009.

[10] Bin Xu and Dan Yang. *Motivation Classification and Grade Prediction for MOOCs Learners.* In *Comput Intell Neurosci*, 2016 Jan.

[11] G. Hughes. *On the mean accuracy of statistical pattern recognizers. In IEEE Transactions on Information Theory, Volume 14*, pp. 55-63, 1968.

[12] K. Gnana Sheela and S. N. Deepa. *Review on Methods to Fix Number of Hidden Neurons in Neural Networks*. In *Mathematical Problems in Engineering Volume 2013, Article ID 425740*, 2013.