Ca' Foscari
University
of Venice

Master's Degree Programme

in Economia-Economics
"Second Cycle (D.M. 270/2004)"

Final Thesis

Can an Artificial Neural Network automate the credit rating process of small and medium-sized enterprises?

Supervisor
Ch. Prof. Marco Corazza

Graduand
Leonardo Nadali
Matriculation Number 838182

Academic Year
2016 / 2017

Index

Introduction

1.1
Purpose and motivation

The objective of this study consists in investigating whether a particular machine learning technique, known as Artificial Neural Network, is able to predict with some degree of confidence the failure of Medium to Small Enterprises (SMEs) by trying to reproduce the results obtained in literature in a large set of Italian companies. Also, a new technique that can improve the obtained results will be discussed.

The hunt for a system that could model the bankruptcies of firms, or more in general the insolvency of borrowers, is a very old field in finance and economics. The utility of such a tool is hard to overestimate:
First of all, among all the sources of risk, credit is the one that can have the largest impact on a corporate level, if we exclude natural disasters. Being able to reduce it by having a reliable tool that can predict exactly which borrowers must be trusted and which should be avoided could raise the profitability of a firm dramatically and liberate large capitals set aside to cover for insolvent borrowers.
Secondly, all the businesses and financial institutions whose business model depends strictly on the outcome of a loan (primarily banks, that profit from loans to firms and customers, but also insurance companies or any other related activity) have of course a high interest in being able to predict insolvencies. Both the ability to avoid bad borrowers and the losses that they create and especially the possibility to individuate good customers that

other competitors would have rated as bad can significantly improve the odds of these firms to outrank the competition and have a higher profitability.

Thirdly, a tool that can analyze the account sheets of a firm and assess its financial stability would be of huge help to that firm itself. By performing some auto-analysis firms can become aware of risky situations before the outcome is too certain to be avoided. It is a well proven fact that humans are in general over-confident in their ability to succeed and consistently underestimate risk[1], and entrepreneurs are no exception from this phenomenon. An external judgment, provided by an objective instrument, could provide a strong counterweight to one's own self-judgment and this way could save some enterprises that would otherwise be unsuccessful.

Of course, rating agencies and other external auditors already provide a service which is aimed at doing exactly this, but their work has often being criticized for being too discretionary and expensive. An automatic tool derived from a machine learning approach would not only help those agencies in making better and more objective decision, but would also provide anyone with the ability to do its own preliminary rating evaluations in an inexpensive way. This would be massively beneficial to a world economy which recently suffered from the consequences of a terrible crisis that originated from a credit bubble caused by excessive lending, and which is still recovering from the subsequent credit crunch due in part to the general aversion of agents towards credit risk.

When dealing with a modelization of credit risk, the most fundamental question that must be addressed is whether the agent taken into consideration is a consumer, who applies for a loan issued by a bank or any other agency, or a firm, whose general financial stability and risk of failure

1Moore, D. A., & Healy, P. J. (2008). The trouble with overconfidence. *Psychological Review, 115*(2), 502-517.

has to be assessed. In this work, the choice to focus mainly on firms was made, for the following reason:

The study of machine learning tools that can predict the insolvency of customers has seen a huge rise in the last years, with tens of papers published every year, often with very good results (for a detailed and exhaustive review, see Louzada et al. 2016), while the study of the same phenomenon in firms did not see a comparable interest. This, however, is in contrast with the fact that corporate failures have an economic impact that is perhaps much higher than that of personal credit.

It is also important to highlight the fact that some of the most common machine learning algorithms are most suited to deal with the sequential and time-based data that only corporate accounts can provide, as will be discussed in the following chapters.

In this study the focus will be on Small and Medium sized Enterprises, as defined by the European Commission[2]. This choice was made as these firms are the ones that would benefit the most from an automated and inexpensive credit rating mechanisms. In fact, those enterprises are often the ones that lack any external judgment on their financial stability both because it is much harder to collect reliable data on them and because they can rarely sustain the cost of such procedures. Many traditional models of credit risk rely on the possibility to gather additional data that is not present in the account sheets of these firms, and so they are precluded from the use of such tools.

1.2
Structure of this work

In Chapter 1, a brief review on the history and main principles of Artificial Neural Networks will be provided. In particular, the two Neural

2 http://ec.europa.eu/growth/smes/business-friendly-environment/sme-definition_it

Network architectures used in this study will be presented: the Multi Layer Perceptron, and the Recurrent Neural Network.

In Chapter 2, the current state of the research in this field will be discussed. Some examples of studies that provided useful insights on the problem that constitutes the motivation of this study will be summarized.

Chapter 3 is dedicated to the discussion of the collection and cleaning of the data used for the experiment that constitutes the core of this research. The choice of the database, and the selected firms and variables will be motivated by comparing them against a set of guiding principles. Some useful statistics on those data will be provided, and the procedure that led to the "cleaning" of the data will be explained in detail.

Finally, Chapter 4 will present the procedure and the results of the experiments that, using two architectures of Artificial Neural Networks, tried to predict the failures of Italian SMEs.

Chapter 1

Artificial Neural Networks

2.1

Machine learning and Perceptrons

"Machine Learning" is a very broad concept that encompasses many very different techniques. The main difference between the definition of Artificial Intelligence and its subset ML is that while AI studies in general any automatic decision making procedure, even those where a human has to list all the possible actions the machine needs to performs based on the inputs it receives ("imperative AI"), the latter comprehends any algorithm that gives a computer the ability to solve a problem without being explicitly trained for that particular application. In this sense, a ML model is a particular structure that has the ability to be applied to a variety of problems just by changing the input data on which it will perform a "learning" process.

In this context, models like Ordinary Least Squares or properly coded non-linear regressors are also subsets of ML. However, the advent of machines that had significant computing power sparked a frantic research of new methods that could best exploit this new tool, and new ways to improve "old" standard statistical models. Among the most famous of those: Binary Search Trees, Support Vector Machines, and, of course, Artificial Neural Networks.

Artificial Neural Networks (ANNs) were first developed as a result of biomimicry: the intention was to find a new way of teaching machines how to solve problems by imitating the way neurons in animal brains work. The first result produced by this search was the invention of the Perceptron [Rosenblatt 1957], which is a structure that performs a binary classification by simulating a simple model of how a neuron works (Figure 1).

A perceptron is described by the formula:

$$f(X)=\theta(\sum(w_i x_i+b))$$

where

x_i are the components of the input vector X ,

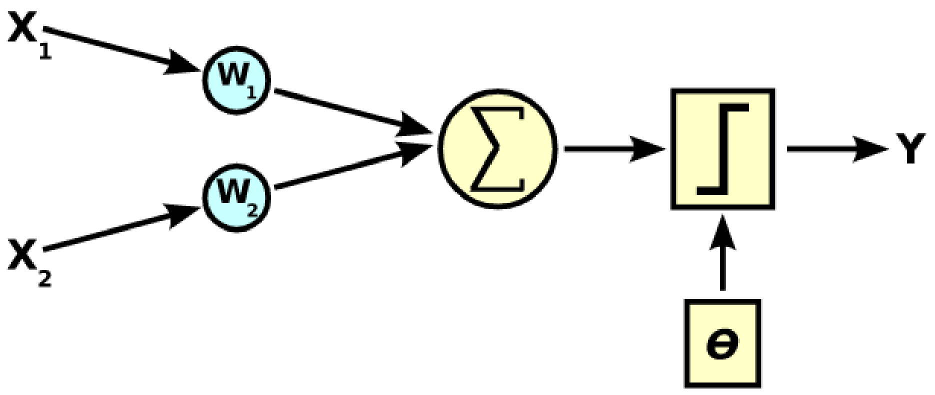
w_i and b are parameters respectively called “weights” and “bias”,

$\theta(z)$ is the “activation function”.

The activation function has the purpose to output a value which will be used to decide whether or not the perceptron is active, thus giving its binary classification property. A common choice for the activation function is the sigmoid function, which outputs values from -1 to 1, that can be interpreted as “if positive, the perceptron is active, otherwise it is dormient”.

Just like neurons take multiple inputs, process them, and communicates whether it is “active” or “dormient”, a perceptron artificially performs the same function. This is why perceptrons are most commonly referred to as “artificial neurons”, or simply “neurons”, which will be the term used in this work too.

Figure 1



Schematic representation of the structure of a Perceptron with two inputs: inputs ***X*** *are multiplied by weights* ***W****, summed up and passed through an activation function* ***θ****.*

2.2

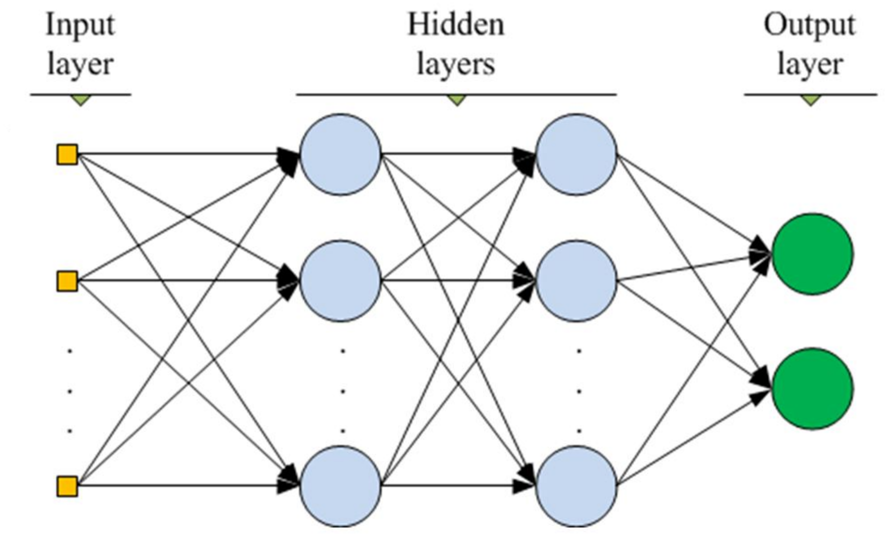
Networks of perceptrons

In the same manner as evolution, which combined the power of a multitude of neurons all interconnected with each other to produce such a powerful tool as our brains, connecting multiple perceptrons together produces an Artificial Neural Network, a structure that when properly set can encode very complex phenomena. A wide variety of ANN architectures have been successfully tested, but the most popular and perhaps the best to explain the general idea of ANNs is called “Multi Layer Perceptron”.

As the name suggests, a MLP is constituted by a series of perceptrons organized in consecutive layers (Figure 2). In this configuration, the first layers represents the input data, and has one node per feature. Each of these nodes sends a “signal” to the first layer of neurons, those neurons perform the computation described in the previous paragraph and send a copy of the results to each neuron in the following layer (the number of which is arbitrary). This operation is repeated a number of times equal to the chosen number of hidden layers (which is two in the example shown in Figure 2). Finally, the signals of the last hidden layer are combined to produce the output, which, depending on the task that must be performed, can be a single neuron or any other number. The main idea behind this structure is that every time a signal is passed from one layer to the next, the informations contained in it are analyzed and summarized to a higher degree of abstraction.

A key result that supports the use of the use of the MLP machine is the “Universal Approximation Theorem”, which states that an appropriately tuned one-layered MLP network is able to approximate any function (whose domain is a compact subset of $\mathbb{R}^n$). The exact formulation given by Cybenko [1989] is as follows:

Figure 2



Representation of a Multi Layer Perceptron with two hidden layers and two output nodes.

Let θ be any continuous discriminatory function[3], then finite sums of the form:

$$f(X)=\sum \alpha_j \theta(W_j X+b_j)$$

are dense in $C(I_n)$.

In other words, given any $G \in C(I_n)$ and $\epsilon>0$, there is a sum $f(X)$ of the above form for which:

$$|G(X)-f(X)|<\epsilon \quad \text{for all} \quad X\in I_n$$

where

X , $\theta(z)$ and b are defined as in paragraph 2.1,

I_n is the n-dimensional unit cube $[0,1]^n$,

$C(I_n)$ is the space of the continuous functions on I_n .

This has subsequently been proved to hold true even when considering $\mathbb{R}^n$ in place of the n-dimensional unit cube. Such a result has boosted the confidence in the use of neural networks for a wide variety of tasks, leading to the late 2000s when the incredible fast growth of machine's computing power (thanks also to the perfection of GPU computing) met with the

3 A "discriminatory function" is a function that divides its inputs in two categories. For example, *tan^{-1} (x)* can be used for this purpose by classifying in a first category inputs that produce an output smaller than zero, and in the second category the ones that output values greater than or equal to zero.

availability of enormous databases, exploding the range of possible applications of this technology. Nowadays, neural networks are well tested tool that is used in many tasks, such as: face and object recognition from images, translation, voice recordings transcription, self driving vehicles, personalized marketing, and many more.

Of course, those networks would be useless if there was no technique to “train” them to solve a particular problem. As with many other aspects in this field, there exists a wide variety of possible ways to handle this, however those can be divided in three main categories:

- *Supervised Learning*

  This method consists in presenting the network with two lists of data: the inputs and the labels. Every value in the in input space is associated with one label, and the network is trained to on this data in a way that makes it possible, after the completion of the training phase, to correctly identify never seen before inputs and associate the proper label to them.
- *Unsupervised Learning*

  In this case, a network is presented with data whose label and classification is unknown, only a specified set of features of the data is available. The Network is then trained to autonomously find some similarities in the inputs, and return a rule that divides them in different classes.
- *Reinforcement Learning*

  Just like humans and animals can learn by trial and error, by repeating actions that produced wanted outcomes and avoiding actions that led to undesired consequences, machines can be programmed to do the same. In this case, instead of feeding a network some values that it must learn to replicate, the algorithm is given the possibility to explore the space of possible actions it can take, and it is provided with a way to change its behavior depending on the feedback it receives.

In this work, a supervised approach will be used. The networks architectures that were used (discussed in Chapter 4) was fed with a set of inputs constituted by the accounting balance sheets of Italian firms, and a set of labels that indicate whether or not any particular firm failed.

The way Supervised Learning is usually implemented is via a technique called "Backpropagation through Gradient Descent" (for a history of Gradient Descent use in ANNs, and a detailed discussion of ANNs in general, see [Schmidhuber 2014]). The idea behind it is simple: the gradient of a function represents the direction of maximum growth of that function, while its opposite is the direction of maximum descent. In an analytical representation:

$f(x)$ is the function that must be minimized,

x_i is an estimate of the best solution,

$\nabla f(x)$ is the gradient of $f(x)$,

λ is a positive (sufficiently small) constant called "learning rate".

Gradient Descent finds the next best solution by this iteration:

$$x_{i+1} = x_i - \lambda \nabla f(x_i)$$

In practical applications, this procedure is implemented in the following way: first, the weights and biases in a network are initialized randomly. Then, inputs are fed through the network and the result is compared to the desired feature. The difference between the desired feature and the produced input is calculated by a properly specified loss function (the exact formulation of which must be chosen depending on the specific problem considered), and the gradient of this function with respect to the weights and biases of the last layer is computed. The process is repeated for every layer until the input, and then the weights and biases are updated based on overall direction of the gradient through all the layers, computed through the chain rule.

This process is not granted to find the best possible solution, but will instead converge to the local optimum in the space of the ANN parameters. For this reason, optimal tuning of the hyper-parameters[4] of both the network and the GD algorithm is required to achieve good results. More details on how this procedure was implemented in this study will be discussed in Chapater 5.

2.3
Types of Artificial Neural Networks

The Multi Layer Perceptron is just the simplest form of an ANN, and through the years hundreds of different architectures have been developed, each one of them perfected to perform the best in a particular problem. Two of the most popular, and among the first to emerge, are the Convolutional Neural Network and the Recurrent Neural Network.

The Convolutional Neural Network (CNN) was invented to deal in particular with the classification of images [LeCun et al. 1989]. The problem of images as inputs in a problem is that the data they represent is characterized by a very high dimensionality: each pixel of an image, in fact, can be considered as one variable in the problem. While it is possible to process an image with a standard MLP, this would not only require a huge computational power, but has also been proven to be highly inefficient with respect to more modern techniques such as CNNs.

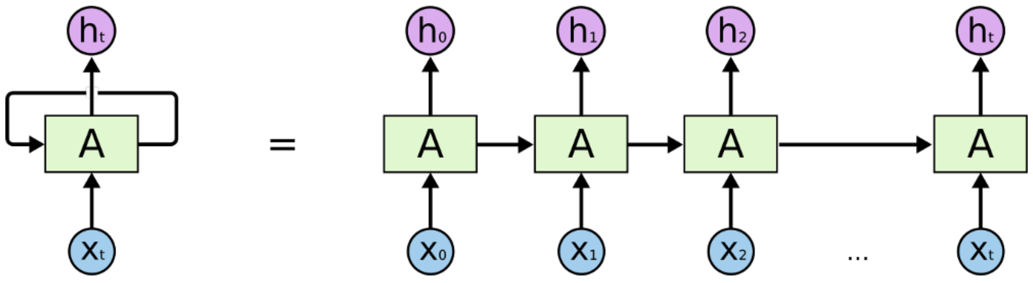
4 Usually, in Machine Learning research the word "parameters" is used to refer to the values in a network to be optimized during learning, which are the weights and the biases, while "hyper-parameters" refer to the values that define the network structure and algorithm, such as: the number of nodes per layer, the number of hidden layers, the learning rate, etc.

CNNs solve this problem by performing two operations on the data: convolutions, that highlight the most meaningful features of an image while removing noise, and pooling, which reduces the dimensionality of the images. Both those operation are done by moving a “filter” of an arbitrary size around the image that must be processed. This operation can be repeated any number of time, and then a standard MLP can be applied to its results. Then, both the MLP and the filters parameters are updated using GD.

Neither MLPs or CNNs, however, have an intrinsic ability to correctly represent ordered data, in both those architectures the input $[x_1, x_2, x_3]$ does not carry any different meaning than the input $[x_3, x_2, x_1]$. This can be a huge problem when a problem has an inherently sequential nature, like it is the case when dealing with written and spoken language processing, or time series data. Imagine the objective of a research is to predict the unemployment level in an economy, based on growth in the previous three years, and that ANNs are used to find the correct prediction. It is easy to understand the feeding the network with the values $[4\%,\ 1\%,\ -5\%]$ is very different from feeding $[-5\%,\ 1\%,\ 4\%]$ as an input.
This is the issue that Recurrent Neural Networks (RNNs) were invented to solve [Lipton et al. 2015]. The idea behind a RNN is to make each of the neurons that compose it “recurrent”, meaning that its input is made both of the input data that the network must learn to represent and model, and of the previous value of the neuron itself (Figure 3). Using as an example the unemployment prediction framework described above, an RNN would function this way:

- The first input, “4%”, is fed through the network in *t=0*. A neuron with a weight on the input equal to -0,5; a bias equal to 3, and a linear activation function takes this input and computes the unemployment prediction $(-0.5\times4)+3=1\%$.

Figure 3



A Recurrent Neural Network in its looped and "unrolled" representation. ***X_t*** *are the inputs at time* ***t****, that are transferred to* ***A*** *(a neuron or, more commonly, a layer of neurons). For every* ***t, A*** *produces an output* ***h_t*** *and also sends it as an input to the network in* ***t+1****.*

- In *t=1*, the value of 1% is fed through the same neuron, but this time the output of the previous time step is also included. Let us say that the weight on the recurrent gate (which is the weight on the value coming from *t-1*) is 0.5, and the bias is 0. Then, this time the neuron outputs a value of $(-0.5\times1+3)+(0.5\times1)=3\%$.
- In *t=2*, we proceed with the same computation of the previous step, to get a value of $(-0.5\times-5+3)+(0.5\times3)=7\%$.

We can see that the network is able to predict that a decrease in growth leads to an increase in unemployment. But will the results be consistent with this model when the network is fed the data coming in the opposite order?

- In *t=0*, the output is $(-0.5\times-5+3)=5.5\%$.
- In *t=1*, $(-0.5\times1+3)+(0.5\times5.5)=5.25\%$.
- In *t=2*, $(-0.5\times4+3)+(0.5\times5.25)\approx3.6\%$.

Yes, the simple RNN in this example is already able to model a relation very similar to Okun's law[5] even if it only is constituted of a single recurrent neuron. Because of the great properties of RNNs when dealing with time series, this model was used in this study (together with an MLP network) to try to predict the failures of Italian SMEs. The results of this experiment will be discussed in Chapter 4.

5 The macroeconomic formula that relates unemployment and GDP, see [Okun 1962].

Chapter 2

ANNs for Credit Scoring

3.1
Comparability of the results

Most of the works in the field of machine learning applications to credit risk actually deal with a different environment than the one that constitutes the objective of this work, as they concentrate on consumers' risk of insolvency, rather than corporate risk. This is a similar problem, but still different enough that results in those two settings are hard to compare, especially as in general accuracy scores in this field are heavily reliant on the chosen dataset [Louzada et al. 2016]. Also, among the studies that were carried out on corporate credit risk datasets, only very few of those considered here actually present the breakdown of their results between Type 1 and Type 2 error (where Type 1 error is the share of failed firms among those that are predicted to fail, and Type 2 is the share of non-failed among those predicted not to fail). This makes it very hard to tell whether the accuracy scores reported actually carry any meaning, as for example a dataset containing 90% of "healthy" cases and 10% of bankruptcies would get an overall accuracy of 0.9 just by adopting the trivial strategy of always predicting firms to be non-failed.

Even when different studies use the same dataset for their analysis (which is very common as there are two consumer credit risk datasets in particular that have become sort of a standard for research in this field, called the "Australian" and the "German" datasets [Bache, Lichman 2013]) it may be hard to compare them as, for example, one study may use some criteria to exclude outliers from the data, thus making it “easier” to train an efficient model on it, or it could decide to use a different set of explanatory variables.

3.2
Examples from previous studies

In this paragraph some examples taken from the research in this field will be discussed, while in Chapters 3 and 4 the details of the new experiment run for this study will be presented.

- Angelini et al. 2008

This study uses a dataset of 76 firms provided by and Italian bank. The criterion for the selection of these firms is not specified, but from the nature of the fields collected it can be inferred that only firms whose credit application was accepted by the bank are included. This could be a cause of bias in the results.

The features of the data consist in 11 fields, 8 of which are balance sheet indexes and the three remaining are related to the quality of the repayment ability of the firms. This makes this dataset very specific to the situation it is studying: the predictions are not based only on the informations coming directly from the accounts, but fields that already indicate a state of distress are included. This could be a cause of endogeneity in the data. Every one of the 11 fields is collected in three consecutive years.

Two ANN architectures are trained: the first one is a standard MLP, that takes 33 inputs and has a single output neuron. The second one performs a kind of convolution on the input data, by grouping the three years of data for every feature together and connecting them to one single neuron, the 11 convolutional neurons are then used as an input for a fully connected MLP.

In the standard network the Type 1 error is as low as 12,25%, Type 2 10,27%. In the "convolutional" network the errors are respectively 6,13% and 9,15%.

- Yu et al. 2008

In this study, a procedure called "ensemble learning" is used. The idea behind it is to train a number of different networks (that can be differentiated both by their architecture and parameters or on the initial location of weights and biases and the selection of the training set), select among all of the trained networks the ones that give the most different predictions (based on a correlation matrix), and use a rule that summarizes their results.

Two main techniques are used to perform this "ensemble" summarization: majority voting, which simply ranks an entry as insolvent or not whether the majority of the networks that participated in the vote agree on that prediction; and reliability-based voting, in which the vote of each network is assigned a weight based on a metric that measures mainly how accurate those networks are, and then proceeds to sum up these votes.

The analysis is performed on two datasets, one dealing with consumer credit risk evaluation and one dedicated to corporate credit risk. The corporate risk dataset covers 60 firms, half failed and half healthy, the training set is made of 30 firms in total. In this case too, the sample size is extremely small, and this has of course an influence on the results.

Type 1 and Type 2 errors of the corporate risk case in the majority voting environment are respectively 19,85% and 17,94%. The reliability-based voting procedure produces 17,83% and 14,37% error rates.

- Kashman 2010

This work is mostly concerned with finding the best hyper-parameters that allow a standard Multi Layer Perceptron to achieve the best performance on a credit risk dataset.

It is the first, among the examples discussed here, to use the famous “German dataset”. It consists of 1000 entries of people who applied for credit grants, 300 of which are classified as “bad” (insolvent) and 700 as “good”. The features consist of 24 numerical and categorical fields, representing various demographic and financial informations on the applicants (credit history, job, social status, family details, informations on the requested credit, etc).

Three MLP architectures are tried out: they all have 24 input neurons and one output neuron (whose value is interpreted as “good” if greater than 0,5, “bad” otherwise), but they vary in the dimension of the single hidden layer, which can be formed by either 18, 23, or 27 neurons. Each of these architectures is trained with nine different ratios between training and test data, starting from 1/9 and going up to 9/1.

The best results are provided by the network that has 23 neurons in its hidden layer and splits the data in 400 cases for the training set and 600 for the test set. The validation error of this network is 26,83%; no information is provided on the contribution of Type 1 and Type 2 errors to this total.

- Kim 2011

The author of this paper analyses the bankruptcies in a very specific sector: Korean hotels in the years ranging from 1995 to 2002. This is interesting as it can provide an insight on whether it can be easier for model that is focused on a specific industry to produce better results. The data collection procedure is also very interesting: first, the data from the accounts of 33 bankrupted hotels was collected; then more hotels are selected to be

included in the dataset, by choosing 33 more whose total assets sizes matched those of the bankrupted hotels.
The variables chosen to represent the financial situation of the considered hotels fall in five categories: liquidity (current ratio, quick ratio, and account receivable turnover), stability (debt to equity ratio and fixed assets to long-term capital ratio), profitability (profit margin ratio, ordinary income margin, return on equity (ROE), ordinary income to owners' equity ratio), activity (asset turnover, inventory turnover, and fixed asset turnover), and growth (growth in revenue, growth in assets, growth in ordinary income, growth in net income, and growth in owners' equity).
A Multi Layer Perceptron is used on this data, and the results show a 4,8% Type 1 error and 12,1% Type 2 error. This is consistent with the intuition that a gain in accuracy can be expected when studying firms in a single economic sector, as they usually "behave" in a similar way.

- Pacelli and Azzolini 2010

Pacelli and Azzolini used data consisting of 273 Italian firms which had a total turnover of less than 50 million euros and a workforce of less than 500 employees. This means that this dataset, just like the one that will be used in this study (discussed in Chapter 3), focuses on Small and Medium Enterprises. The chosen network is an MLP with 24 financial ratios as the input layer, then two hidden layer of 10 and 3 neurons respectively. The output is constituted by a single node.
This study presents an interesting peculiarity: instead of dividing the firms in two classes, "failed" and "healthy", like most studies do, they decided to rank them as either "safe", "vulnerable" or "at risk". This makes it harder to compare the results with other ones, but is an interesting methodology that, depending on the specific needs of who needs to use this kind of technique in real world applications, could be very useful. The results show an error of 65,2% for the "at risk" class and 15,8% for the "safe" class.

- Lee and Choi 2013

In this study the objective is to investigate whether the application of ANNs for the prediction of corporate failures yields different results when applied in different industries. For this reason, the authors collect three different datasets consisting of firms coming from three economic sectors: construction, retail, and manufacturing.

The total amount of firms in the three datasets amounts to 229 (75 in construction, 67 in retail, 88 in manufacturing), of which 91 are bankrupt. A total of 100 financial ratios are available for every firm, but the authors decide to use only a small amount of those (6 for construction and manufacturing, 4 for retail) by selecting the ones that based on t-tests and correlation analysis prove to be the best predictors of bankruptcies.

As expected, the results differ significantly between the three sectors: for construction, Type 1 error is 12% and Type 2 6%, for retail 21% and 7% respectively, and for manufacturing 7,9% and 10,5%.

- Zhao et al. 2015

There are three main aspects this study explores: proposing a procedure for the shuffling of the input data; finding the best size for the training, test, and validations sets; and exploring the change in accuracy obtained by varying the number of neurons in an ANN.

For the shuffling of the data (which come from the already mentioned German dataset) the authors propose a procedure that guarantees that each time a set is sampled from the database the percentage of failed and non-failed firms remains constant. This is particularly useful in the case of a relatively small dataset where pure and unaccounted randomness could cause huge variations in the share of failed firms in the test, validation and training sets. The proposed procedure guarantees that any good accuracy scores obtained is not due just to the fact that a “lucky” training set was used.

Every experiment is repeated three times, every time varying the sizes of the three sets. The three compositions tried were: 80% training set, 10% test and 10% validation; 90%, 5% and 5% respectively; 60%, 20% and 20%. The authors discuss the fact that the their networks seem to have the best performance when used with the 80% and 10% combination.

The ANN architecture used is an MLP with one hidden layer. The number of units in the hidden layer is varied from 6 to 39. For each size of the hidden layer twenty training sessions were run. The authors do not find any substantial difference in performance among the various sizes when comparing accuracies on the validation set, however the accuracy on the test set (which is the value used to determine when to stop the training of the network) seems to decrease with the increase in the size of the hidden layer.

Chapter 3

The Dataset

4.1

Guidelines for choosing a good dataset

Based on the observations made in the previous sections, some requirements must be laid out for the dataset we will choose. The choice of the dataset is in fact one of the most important when building a predictive model [Wenzelburger et al. 2013]. It is obvious that no model can predict relations and behaviors if those are not present in at least one case of the dataset, but this is not the only reason a good choice of the data is crucial in giving predictive power to a model: first of all, any bias or distortion of the phenomena that is present in the data will also be reflected in the predictions, thus lowering the overall accuracy; secondly, when attempting to train models such as Artificial Neural Networks, that rely on the identification of a local optimum to solve a problem (because the high dimensionality of their parameters and their complex interactions makes it impossible to compute the global solution), any fluctuation in the shape of the dataset is able to drastically change the shape of the space of possible solutions, which means that the final prediction could be much different.

For these reason, the dataset that will be used should ideally have the following characteristics:

- *Abundancy*

  The dataset needs to contain as many examples as possible. As stated in the previous chapters, the rise of Artificial Neural Networks techniques is also due to the availability of very large collections of data, in the order of magnitude of hundred of thousands or even millions of example cases. This is necessary not only to reduce the noise-to-information ratio, allowing faster and convergence to a better local solution [Wong and Sherrington 1993], but also to reduce the chances of overfitting the features. If a dataset does not contain at the very least some tens of thousands of entries, it can be better to use classic approaches such as regression that in these cases can often produce more consistent results.

- *Quality*

  The data must, of course, represent the real values as close as possible, and thus it must be collected in a careful manner by a trusted agent. Every field should be accompanied by an explanation of its meaning and the collecting procedure. Also, the amount of missing values in the fields should not be too big, as these can have destructive effects on the learning algorithm [Lopes and Ribeiro 2011] and their handling is in general not an easy task. If data is missing, reasons for that should be provided.

- *Generality*

  Unless the researcher intentionally aims at identifying a phenomenon in a confined and peculiar set of conditions, the dataset should contain cases that are not specific to any particular context or collecting method. The set of conditions presented should be broad enough that no hidden dataset-specific fixed effects are present, so that when the findings are applied to data coming from other sources, the accuracy can be close to the value computed on the training data.

  In many cases in credit risk research, the dataset comes from firms or people whose credit applications were accepted by the issuer (such is the case in the cited “German dataset”). This clearly creates a bias towards a model that instead of identifying the general indicator of the likelihood of a debtor to fail, identifies the features of debtors that looked good initially but then failed.

- *Conciseness*

  The features that form the data should represent the phenomenon in a concise way, avoiding to be either too generic or too specific. If the features are too generic it is unlikely that they represent meaningful informations, if they are too specific the training could be too slow or could get stuck too easily in local optima [Tuckova and Bores 1996].

  Also, if the model is required to output predictions rather than just a classification, great care must be taken in choosing which features to use so as to avoid introducing data that represents a consequence of the outcome rather than one of its determinants.

When choosing the right dataset there is another criterion that must be taken into consideration, at it is of course the objective of the research. The main reason behind this work is to investigate the possibility to build a tool that is useful to rate the credit risk of those firms, particularly medium to small sized ones, for which other traditional methods prove not to be so effective. This happens mainly because data on share prices or independent rating agencies is not available or difficult to obtain, making it impossible to apply popular models such as Merton’s [1974] and Vasicek’s [1977]. A data source that focuses on this kind of firms is then necessary.

The criteria outlined in this paragraph will be used throughout this chapter as a reference point to evaluate the choice of the dataset that was made for this study.

4.2
The chosen data source and its structure

The dataset that was chosen for the experiment comes from the *AIDA* database[6]. *AIDA* is a database which collects financial, commercial

6 www.bvdinfo.com/en-gb/our-products/data/national/aida

and anagraphic informations of Italian companies; in particular, it holds the digitalized balance sheets of more than one million firms and their legal status (whether they are still active, bankrupted, or else).
The database query used to extract the data was organized as follows:

- *At least five years of available accounts*

  Only firms that had at least five years of available accounts were selected. This grants the possibility to feed our algorithm with a dynamic view of each firm and not just a static picture of it, and will be essential for the training of the Recurrent Neural Network model that will be discussed in chapter 4.
- *Max total assets = 43.000.000€* _OR_ *max revenues from sales and services = 50.000.000€ and max employees = 250*

  This criterion was defined so that every firm in the dataset falls in the definition of Small and medium-sized enterprises (SMEs) as provided by the European Commission. This is useful to focus our research on those firms that would benefit the most from a non conventional model of credit risk assessment, as stated in the previous paragraph. The criterion had to be met in every year.

The total number of entries resulting from this query amounts to 872.558. A link to the file containing extracted firms in *.csv* format can be found in Appendix 1. Even though this list of firms will be refined and reduced in the following steps, it is already possible to see that the amount of accounts being worked on is pretty large, and this is in line with the abundancy principle defined above. This is, to the knowledge of the author, by far the largest dataset used in this kind of research. As discussed in Chapter 1, studies on Machine Learning applications to credit risk rarely use more than a couple of thousand firms as the input for their models, and many of them are actually limited to some hundreds or less.

The *AIDA* database allows to choose among dozens of different categories of data (the columns of the database, if we consider each firm to represent a row). A choice has to be made with respect to which among these categories will be downloaded: not only this choice is necessary to follow the conciseness principle and this way produce a dataset that will yield better results when applied to our training algorithm, but is also necessary from a practical point of view. In fact, the downloaded files with 52 columns in *.csv* format occupy 3,4GB of space in total and had to be downloaded in 211 batches of 4.400 entries each, which constitutes the maximum file size that the database script can handle (also, all the data processing was made on a computer with 4GB of RAM memory, and even though many computations were done in batches, this would have been highly impractical if the data size had been too large).

For this reason, instead of downloading data for every single account sheet section (the database provides entries for every category that the Italian law requires firms to fill), only the major indicators computed on the base of the balance sheet and the profit and loss account and already provided within the *AIDA* framework were selected. This should conserve most of the information contained in the database while also reducing its size and the noise in the data. Only two balance sheet fields are kept: *'Profit (loss) EUR'* and *'Total assets EUR'*, these are kept in the data in hope that they will help the network to have a sense of "scale" while comparing firms: by checking those two values the network may be able to estimate the total size of the profit and loss account and the balance sheet respectively (being able to tell a "very small firm" from a "medium firm"), and it is reasonable to suppose that the economic size of a firm is important in estimating its financial stability.

Besides these strictly economic features, some anagraphic and descriptive variables are also considered, those are: *'Number of employees'*, *'Accounting closing date Last avail. Yr'*, *'Tax code number'*, *'Trading* *address – Region'*, *'Legal status'*, *'Incorporation year'*, *'Last accounting closing date'*, *'Procedure/cessazione'*, *'Date of open procedure/cessazione'*, *'NACE Rev. 2'*.

A complete rundown and description of every feature will be given in the next paragraph. It is however worth mentioning that while some of these variables are kept just for database cleaning and data analysis reasons, some others will become explanatory variables in the final model. From now on the first group of them will be referred to as *“anagraphics”*, while the latter, together with the previously mentioned account indexes, will be called *“controls”*. Finally, *'Legal status'* and *'Procedure/cessazione'* will be used to compute the dependent variable of the model, which will be referred to also as “label” (Paragraph 4.4 will provide the details of this step).

4.3
Statistical overview of the data

In Table 1 the complete list of the fields that were downloaded from the database is presented, together with some basic statistical informations (the names are reported exactly as they are in the database, most are in English, some in Italian and some are mixed, for more informations check the *AIDA* website). Account indexes are calculated on the basis of the fields described in the Italian accounting law, a descriptive table of the formulas used to calculate this fields is available in the link provided in Appendix 1.

The data for each of the categories indicated in Table 1 is collected over a period of five years, so the statistics presented are computed considering the data for every year and every firm.

Table 1

Name	***Median***	***Mean***	***Standard Deviation***	***Maximum***	***Minimum***
'Tax code number'	-	-	-	-	-
'Legal status'	-	-	-	-	-
'Incorporation year'	-	-	-	*2017*	*1845*
'Last accounting closing date'	-	-	-	*2017*	*1994*
'Procedure/cessazione'	-	-	-	-	-
'Date of open procedure/cessazione'	-	-	-	-	-
'NACE Rev. 2'	-	-	-	-	-
'Profit (loss) EUR'	*1032.0*	*-6339.14*	*2946444.9*	*2.5512e9*	*-1.1846e9*
'Total assets EUR'	*523053.0*	*2707407.8*	*3.6923e7*	*2.1024e10*	*0*
'Total shareholder's funds EUR'	*65698.0*	*849603.9*	*1.8413e7*	*1.0108e10*	*-9.0713e8*
'Return on sales (ROS) %'	*3.34*	*2.53*	*12.40*	*30.0*	*-50.0*
'Return on asset (ROA) %'	*1.78*	*-1.67*	*39.53*	*996.36*	*-999.95*
'Return on equity (ROE) %'	*2.64*	*2.44*	*32.40*	*150*	*-150*
'Banks/turnover %'	*0.99*	*13.17*	*20.93*	*100*	*0*
'Liquidity ratio'	*0.92*	*1.34*	*1.53*	*10*	*0*
'Current ratio'	*1.2*	*1.69*	*1.62*	*10*	*0*
'Current liabilities/Tot ass. %'	*0.97*	*0.78*	*0.31*	*4.99*	*0*
'Long/med term liab/Tot ass. %'	*0.03*	*0.22*	*0.31*	*1*	*0*
'Tang. fixed ass./Share funds %'	*0.22*	*0.98*	*2.20*	*15*	*-10*
'Depr./Tang. fixed assets %'	*0.87*	*1.72*	*2.08*	*10.0*	*0*
'Leverage'	*3.49*	*15.60*	*160.87*	*9991.3*	*-1999.9*
'Coverage of fixed assets %'	*1.3*	*16.4*	*72.4*	*999.9*	*-49.9*
'Banks/Turnover (%) %'	*0.99*	*13.17*	*20.92*	*100*	*0*
'Cost of debt (%) %'	*5.41*	*6.36*	*4.52*	*20*	*0*
'Interest/Operating profit %'	*6.49*	*31.82*	*64.24*	*400*	*0*
'Interest/Turnover %'	*0.86*	*4.13*	*10.28*	*100*	*0*
'Share funds/Liabilities %'	*0.26*	*2.14*	*10.51*	*199.99*	*-19.96*
'Net Financial Position EUR'	*255*	*488191.1*	*5975234.1*	*2.2311e9*	*-3.3734e9*
'Debt/Equity ratio %'	*0.04*	*2.95*	*30.83*	*997.43*	*-998.98*
'Debt/EBITDA ratio %'	*0*	*0.58*	*50.00*	*999.73*	*-999.92*
'Total assets turnover (times)'	*0.59*	*0.81*	*0.88*	*5*	*0*
'Incidenza circolante operativo %'	*16.8*	*39.63*	*137.52*	*999.99*	*-999.98*
'Stocks/Turnover (days)'	*0*	*22.34*	*62.67*	*499.98*	*0*

'Durata media dei crediti al lordo IVA (days)'	*88.71*	*150.48*	*231.47*	*1999.99*	*0*
'Durata media dei debiti al lordo IVA (days)'	*95.52*	*121.79*	*108.46*	*500*	*0*
'Durata Ciclo Commerciale (days)'	*35.44*	*64.98*	*181.79*	*2296.88*	*-499.9*
'EBITDA EUR'	*17173.0*	*86895.84*	*745337.98*	*3.5718e8*	*-3.6346e8*
'EBITDA/Sales %'	*6.86*	*1.06*	*82.29*	*999.89*	*-999.99*
'Return on investment (ROI) %'	*3.02*	*4.03*	*10.59*	*30.0*	*-30.0*
'Number of employees'	*1*	*7.11*	*43.71*	*44695*	*0*
'Added value per employee'	*35540.0*	*44993.30*	*48094.83*	*499970.0*	*-50000.0*
'Staff Costs per employee'	*26170.0*	*27233.06*	*16746.59*	*100000*	*0*
'Turnover/Staff Costs'	*4.83*	*9.12*	*12.77*	*100*	*0*
'Net working capital EUR'	*36452.0*	*277643.3*	*7255399.2*	*4.0647e9*	*-5.1363e9*
'Gross profit EUR'	*134977.0*	*704869.7*	*2359459.5*	*5.8944e8*	*-5.9986e8*
'Net short term assets EUR'	*-22074.0*	*-465315.6*	*1.4976e7*	*3.335e9*	*-9.828e9*
'Share funds - Fixed assets EUR'	*7011.0*	*-288787.1*	*1.2546e7*	*1.8628e9*	*-9.3463e9*
'Revenues from sales and services th EUR'	*144.0*	*1401.29*	*5176.87*	*165114.0*	*-1274.0*
'Cash Flow EUR'	*8659.0*	*50993.4*	*2919754.5*	*2.5534e9*	*-1.184e9*

As appears to be evident from Table 1, the most extreme values of some fields were clipped in the original dataset. This means that when the original data was collected, any value that surpassed some threshold *X* was replaced with the value *X*. As an example, *'Staff Costs per employee'* does not have any values greater than 100.000€, so clearly any firm that had 120.000€ as a value for that variable had its account modified and that value substituted with 100.000€. This is also clear in its histogram (Graph 1) as it can be seen that the right tail of the distribution is suddenly truncated at that value.

Other categories that may have suffered clipping of their data include: *'Added value per employee'*, *'Return on investment (ROI) %'*, *'EBITDA/Vendite %'*, *'Durata media dei debiti al lordo IVA (days)'*, *'Debt/Equity ratio %'*, *'Debt/EBITDA ratio %'*, *'Total assets turnover (times)'*, *'Incidenza circolante operativo %'*, *'Stocks/Turnover (days)'*, *'Leverage', 'Coverage of fixed assets %', 'Return on equity (ROE) %', 'Return on asset (ROA) %'.* The histograms for some of these variables are presented in Graph 2. In the case of *'Return on investment (ROI) %'* and *'Cost of debt (%)'* the distortion caused by data clipping is evident in the graphs, while in the other cases it seems to be limited to just a few outliers, and so hopefully the information carried by those variables is likely not to be distorted too much by this clipping.

In the link available in Appendix 1, the complete set of histograms for every feature can be found, each histogram is divided in 40 bars and has a total number of cases equal to:

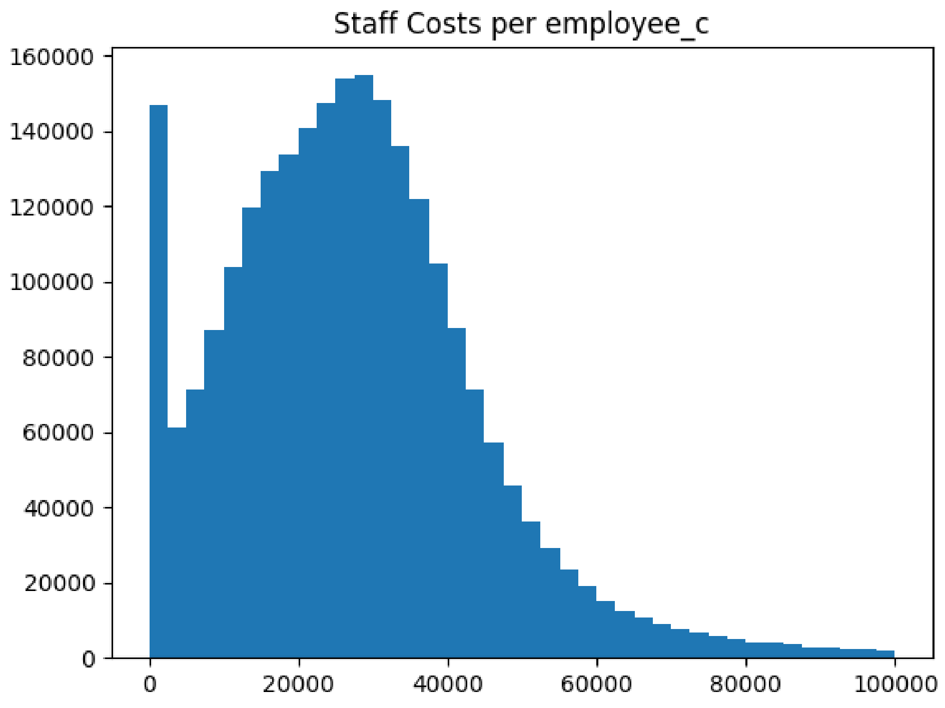
$$(number\ of\ firms) \times (account\ years\ collected) - (missing\ values\ of\ category)$$

The list of indexes and categories included in the dataset that was downloaded is surely broad and varied, but this makes sense if considered in light of three considerations:

First, it must be recalled that the objective of this work is to build a model which takes as few assumptions as possible on the nature of the studied phenomenon. The modelling of the phenomenon, needed to make predictions, will be self-built by the algorithm and not provided by the researcher. Making decisions on which variables to exclude by stating that they are not useful predictors is just as strong an assumption on the phenomenon as any other, and even if something has to be excluded in order to avoid introducing too much noise and making computations too hard, it is a good idea in general to keep the features that will be fed to the network as numerous as possible.

Secondly, even though feeding hundreds of categories to the network would be impractical and counter-productive, there is actually no reason to be excessively shy in the parameters selection: neural networks are regularly used with input sized that go up to hundreds of nodes, especially when working with video or image processing (see for example [Qiao et al.

Graph 1

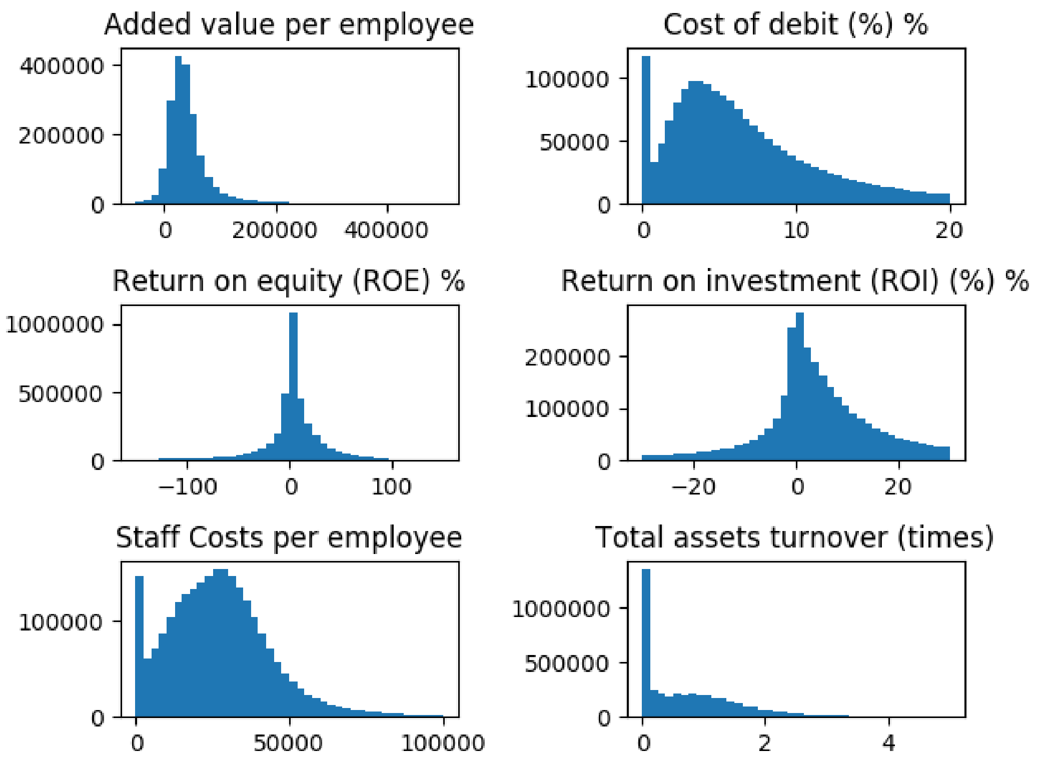


Histogram showin the distribution of the values of the variable 'Staff Costs per employee' in the complete dataset, before the removal of any row.

2017]). Of course, the nature of image recognition problems is very different from a financial analysis, but it is still indicative of some degrees of freedom when dealing with this particular issue. Also, contrary to techniques like Least Squares or other traditional estimators, neural networks are not so sensitive to high values of correlation in the explanatory variables [Wendemuth et al. 1993]. So there is no need to check the covariance matrix and prune inputs that present high covariance with each other.

Thirdly, the reasons and the paths that lead to the failure of a company are them too very broad, complex, varied, and interconnected. If the artificial neural network has to learn complex mechanisms, be able to make good

Graph 2



Histogram showin the distribution of the values of some of the mostsignificant variables in the complete dataset, before the removal of any row.

abstractions, and learn underlying relations in the data that are presented to it, the only way is to give a representation of the agents it has to analyze that is as comprehensive as possible.

4.4

Data cleaning procedure

As it is now, the dataset can not be fed directly into a neural network, and some work is necessary to refine the data and transform it in a format that is more suitable to be processed. Two main steps are required before the work can proceed any further:

- The general quality of the data must be assessed, and entries and variables with too many missing values must be dropped to avoid any problems that could hamper the training and the abstraction ability of the network.

- The data must be transformed in a format that allows it to be fed correctly to the network. This step also includes the definition of the labels variable, which must be created from the available informations.

In this section, the details on how these procedures were implemented will be provided. Also, the complete Python code that was used for this purpose can be found in Appendix 4.

First, every entry that had a missing value in *'NACE Rev. 2'* or *'Incorporation year'* was deleted. This is mainly due to the fact that the web script that downloaded the database created *.csv* files in which sometimes multiple rows were created for the same firm, as a consequence of some extra information contained in anagraphic variables that could not be contained in a single row. Deleting rows with those missing values grants the certainty that every row in the dataset corresponds to one and just one firm. A total of 600 rows were dropped this way, thus reducing the total amount to 871.958.

Then, the problem of missing values (also referred to as “NaNs”, which stands for “Not a Number”) was tackled. Many different approaches exist on how to handle missing values in a dataset [Kaiser 2014, Louzada et al. 2016], and there is no “one size fits all” solution. The method used in this study follows a four-folded approach: 1) columns with too many missing values are dropped, 2) rows with too many missing values are dropped too, 3) missing values are replaced with their means, 4) dummy variables are built to signal missing values. The details of each of these steps will now be discussed:

1) The number of missing values for every variable of those listed in Table 1 is checked (this count is available in Appendix 2). In this case, NaNs are counted year by year. For example, this means that the interest is not on how many values are missing in the variable *'Leverage'*, but rather how many missing values are in *'Leverage last available year'*, how many in *'Leverage last available year minus 1'*, and so on. After those are counted, columns that present more than 40% of missing values in at least one year are dropped. Also, if a variable has more than one year with 20% or more missing values, it is also dropped. Fifteen features were deleted, and they are: *'Return on sales (ROS) %', 'Banks/Turnover (%) %', 'Depr./Tang. fixed assets %', 'Cost of debt (%) %', 'Banks/turnover %', 'Interest/Operating profit %', 'Incidenza circolante operativo %', 'Stocks/Turnover (days)', 'Durata media dei crediti al lordo IVA (days)', 'Durata media dei debiti al lordo IVA (days)', 'Durata Ciclo Commerciale (days)', 'Return on investment (ROI) %', 'Added value per employee', 'Staff Costs per employee', 'Turnover/Staff Costs'.* Clearly, some of the values that were dropped are actually important indicators of the financial status of a firm (such as Return On Sales, Return On Investment, Cost of Debt, etc) and the model that will be tested will probably have a harder time in predicting failures if it cannot access these values. However if the results will prove to be good anyway, this will only pose in favor of the reliability of the resulting algorithm.

2) This time missing values are checked row by row. If in a row every single year of a variable is missing, and this happens for more than one variable, then that row is deleted. Also, if more than two years of a variable are missing, and this happens in more than five variables, then that row is deleted too. A total of 149.798 rows are deleted following this criterion. This step is especially important in prospect of the application of the Recurrent Neural Network model, which relies heavily on the temporal

structure of the data making it especially important to have consistent time-series values for as many variables as possible.

3) As neural networks can only be fed numerical values, any NaN that remains in the dataset must be replaced by a numerical placeholder. In this case, the placeholder used is simply the mean of the values of that field among all the other firms. Other more sophisticated values could have been computed, but respecting the principle that the least amount of assumptions should be made by the researcher when building an ANN model, the preferred choice was to use a simple placeholder, the mean, and instead build dummy variables.

4) It can be a good idea to let the ANN that will analyze this data have the possibility to "know" which values were missing in the original dataset and were substituted by their mean as explained in the previous point. For this reason, a set of dummy variables is created, each one of them corresponding to one of the fields of the dataset and taking value "1" if the corresponding field is blank, or "0" if it has a number. This means that the total number of features that will constitute the input of the network is doubled in size. This could cause the network to overfit the data, if there is correlation between having some missing values and being at risk of bankruptcy (this issue will be discussed further in the next paragraphs). However, this tradeoff is considered worthy as great care has already been taken in deleting rows and columns that presented too many missing values, and replacing NaNs with "fictional" values without giving our model a way to identify them, could be a cause of even greater distortions in the resulting predictions.

Now that the problem of assessing the quality of the data has been tackled, this data must be prepared for its processing:

The variable *'NACE Rev. 2'* is a four figures code that divides the firms by the economic sector in which they operate[7]. The first figure represents the most general economic activity, and every subsequent one specifies it more in detail. This variable is kept in our model as it is reasonable to assume that the economic sector has an impact on the chance of failure of a firm. In particular, only the first two figures are kept, as being too specific would probably not benefit the model. As this is a categorical value, it must be converted to a numerical form. One popular way to do this consists in the use of One Hot Encoding, which is a method that is from a practical and also theoretical point of view equivalent to building a set of dummy variables for every possible state of *'NACE Rev. 2'*. However, the set of possible values of the first two figures of this variable contains 73 elements, and so such a choice would add a level of complexity to the features that could be unnecessary. Instead, these values were kept as they are, as integer numbers, exploiting the fact that *NACE* codes are organized in such a way that keeps similar economic sectors grouped in numbers close to each other[8], and so it may me reasonable to use a numerical representation that keeps into account this measure of "distance" and does not erase it from the data.

The variables *'Tax code number'*, *'Incorporation year'*, and *'Last accounting closing date'* were not used as inputs for the model. *'Tax code number'* was helpful anyway as a unique identifier of each firm.

Now the dependent variable of the model must be built. To do that, all the possible values of *'Legal status'* and *'Procedure/cessazione'* must be

7 http://ec.europa.eu/eurostat/documents/3859598/5902521/KS-RA-07-015-EN.PDF

8 For example, "Extraction of crude petroleum and natural gas" has the NACE code "06", and the similar sector "Mining of metal ores" has the code "07". A totally different sector instead, like "Repair of computers and personal and household goods" has the very distant number "95" as its code.

considered. In Table 2 all the possible values of *'Legal status'* and their frequencies are listed.

Table 2

Legal Status	***Occurrences***
'Active'	*654065*
'In liquidation'	*81186*
'Bankruptcy'	*47071*
'Dissolved'	*37866*
'Dissolved (liquidation)'	*32376*
'Dissolved (merger)'	*14249*
'Active (default of payments)'	*4278*
'Dissolved (bankruptcy)'	*404*
'Dissolved (demerger)'	*251*
'Active (receivership)'	*212*

A new binary variable is built, *'failure'*, which indicates with "1" the firms that are considered to be failed. Every firm whose value in the *'Legal status'* variable is either *'Bankruptcy'*, *'Dissolved (bankruptcy)'*, *'Active (receivership)'*, or *'Active (default of payments)'* is marked as failed, while the classification of the other firms will be decided based on the value of *'Procedure/cessazione'* which better specifies whether the legal status of the firm can be considered a failure or not. The set of possible unique values of *'Procedure/cessazione'* contains 60 elements, and a conversion table (that can be found in Appendix 3) is built to decide whether any single value among those sixty indicates a failure or not.
It is important to stress the importance of this step: the results of the analysis carried out in this work will of course be highly dependent on the chosen procedure used to create the variable *'failure'*, which will be used to store the labels for our model. The main consequences of poorly chosen labels could either be a bias in the predictions, if a systematic error is

present in the labels, or a slower rate of convergence to a worse local optimum if the noise is centered on the real values [Sukhbaatar and Fergus 2014]. In the case studied here, the presence of a class of firms classified as *'In liquidation'* means that it is hard to be confident in their classification either way, as it is hard to tell whether those should be considered as failed by the model or not. The fact that the column *'Procedure/cessazione'* is checked before making this decision helps in partially reducing this problem, but a potential source of debate remains as to whether the presented procedure can be considered the best, or if better solutions exist.
By this criterion, the number of failures in the dataset amounts to 137.938, which constitutes 19,1% of the total.

The last transformation that was performed on the dataset is the normalization of the values, which is particularly useful to speed up the learning process of a neural network. The values of every year of every variable were collected and normalized to have zero mean and unit variance. The values of *'NACE Rev. 2'* were simply divided by 100 so that they all lie in the 0 to 1 interval.

4.5
Final analysis of the dataset

It could be interesting to know whether the probability of failing is the same between the firms that we excluded from the data and the initial dataset we had. In practical terms, this means exploring the possibility that there might be correlation between the number of NaN values in the account of a firm and its probability of failure. To do that, we set up a Welch t-test to check the null hypothesis that the average number of failures in the complete dataset is the same as the average number of failures in the set of firms that were excluded. The Welch t-test [Welch 1947] was chosen as it provides the best estimates when the two distributions have different numerosities and different variances (and, in a binomial setting like the one this experiment is based on, having different expected values also means having different variances, as the variance of a binomial distribution is $p(1-p)$ where *p* is the expected value). The formula for the Welch t-test is the following:

$$t=\frac{Fc-Fd}{\sqrt{\frac{s_c^2}{n_c}+\frac{s_d^2}{n_d}}}$$

Where

$Fc=\frac{186852}{871958}\approx 0{,}214$ is the share of failures in the complete dataset

$Fd=\frac{48914}{149798}\approx 0{,}327$ is the share of failures in the deleted rows

The test outputs a result of $t=-87{,}09$ corresponding to a p-value so small that the statistical package used (Scipy, a Python library) outputs a value of zero, leaving no doubt as to whether those two values are different at any level of statistical significance.

This results supports the choice of deleting those rows, as clearly the number of NaNs in a row can be a predictor of the likelihood of that firm to fail. Deleting this data means making it harder for the model to find informations to base its predictions upon, which will in turn make the results obtained even stronger if those will be considered satisfactory.

This can be considered a measure to remove outliers in the data, at least if we only consider outliers in the “missing values” dimension. It is often suggested that removing outliers is a good approach when preparing a neural network, and it makes sense that removing the cases that are “harder” to fit would result in a performance increase, but overdoing it might also remove important features that are present in the data, depending on the chosen cutoff, and would also introduce the risk of artificially increasing accuracy only because most of the variance in the features has been eliminated.

Something that is worth discussing, now that the final dataset has been selected, is whether the variables that are left in the final version of the dataset are similar to those chosen in other studies on the subject of corporate financial risk. This is the reason why the comparative graph in Table 3 was created. It shows the variables taken into consideration in four different studies on this subject (one of which, [Corazza, Funari, Gusso 2016], actually deals with a creditworthiness problem in a MURAME environment, a different problem than the one studied in this research, but it is still an interesting comparison as it was run by extracting data from the same database used here, *AIDA*) and whether the same variables, or ones that are similar, are also included in this analysis. As the table highlights, there is only partial overlap among the features used in the various studies.

Table 3

Variables used	***Corresponding or similar variables in the following work***
Altman, Sabato 2007	
Short term debt/equity book value	*'Debt/Equity ratio %' and 'Current liabilities/Tot ass. %'*
Cash/total assets	*'Cash Flow EUR' and 'Total assets EUR'*
EBITDA/total assets	*'EBITDA EUR' and 'Total assets EUR'*
Retained earnings/total assets	*'Return on asset (ROA) %'*
EBITDA/interest expenses	*'Interest/Operating profit %'*
Corazza, Funari, Gusso 2016	
Cost of debt: Financial costs/bank debts	*'Cost of debt (%) %'*
(Current assets – inventories)/current liabilities	*'Liquidity ratio'*
Return on equity (ROE): Net profit before tax/total equity	*'Return on equity (ROE) %'*
Return on sales (ROS): Net profit before tax/sales	*'Return on sales (ROS) %'*
EBITDA	*'EBITDA EUR'*
Total assets turnover: Sales/total assets	*'Total assets turnover (times)'*
R&D costs/total asset	-
Income tax/profit before taxes	-
Equity – equipment	-
Rate of increase of revenues from sales and services	-
Liabilities/total assets	*'Current liabilities/Tot ass. %'*
Cash/total assets	*'Cash Flow EUR' and 'Total assets EUR'*
Working capital/total assets	*'Net working capital EUR' and 'Total assets EUR'*
Intangible/total assets	-
EBIT/sales	*'EBITDA/Sales %'*
EBITDA/total assets	*'EBITDA EUR' and 'Total assets EUR'*
Retained earnings/total assets	-
Net income/sales	*'EBITDA/Sales %'*
Short term debt/equity	*'Debt/Equity ratio %'*

EBITDA/interest expenses	*'Interest/Operating profit %'*
Account payable/sales	-
Account receivable/liabilities	-
Sales/personnel costs	*'Turnover/Staff Costs'*
Angelini et al. 2008	
Cash flow/total debt	*'Cash Flow EUR' and ''Long/med term liab/Tot ass. %'*
Turnover/inventory	-
Current liability/turnover	*'Current liabilities/Tot ass.', 'Revenues from sales and serv.'*
Equity/total assets	*'Leverage'*
Financial costs/total debts	*'Cost of debt (%) %'*
Net working capital/total assets	*'Net working capital EUR' and 'Total assets EUR'*
Trade accounts receivables/turnover	-
Value added/total assets	*'Added value per employee'*
Utilized credit line/accorded credit line	-
Unsolved effects/under usual reserve effects	-
Unsolved effects/under usual reserve effects	-
Transpassing short-term/accorded credit line	-
Transpassing medium-long term/accorded credit line	-
Utilized credit line short-term/accorded credit line	-
Utilized credit line medium-long term/accorded credit line	-
Yu et al. 2008	
Sales	*'Revenues from sales and services th EUR'*
Profit before tax/capital employed	*'Return on investment (ROI) %'*
(earnings before tax and depreciation)/total liabilities	*'EBITDA EUR' and 'Total assets EUR'*
(current liabilities + long-term debt)/total assets	*'Current liabilities/Tot ass.', 'Long/med term liab/Tot ass.'*
Current liabilities/total assets	*'Current liabilities/Tot ass.'*
Current assets/current liabilities	*'Current ratio'*
(current assets - stock)/current liabilities	-
(current assets - current liabilities)/total assets	-
Days from account year end to the failing of annual report	-
Number of years the company has been operating	-
1 if changed auditor in previous three years, 0 otherwise	-
1 if company auditor is a Big6 auditor, 0 otherwise	-

It is now time to come back to the four principles that were laid out at the beginning of this chapter to try to see whether each one of those has been fulfilled, and if they were not, try to understand why. Let us go through each one of them one by one:

Abundancy: with 722.160 data points in this selection, there is little doubt as to whether the collected dataset constitutes a large enough sample. Actually, this is probably, to the knowledge of the author, by far the largest dataset ever used in a research on machine learning and credit scoring (for a very detailed and complete summary and meta-analysis on the topic, which includes a section on database selection, see [Louzada et al. 2016]). This is especially important as it allowed a high level of freedom in the process of data cleaning, by granting the opportunity to drop a lot of “dirty” data points, and because it will help in having a higher degree of flexibility when the neural networks will be trained, as will be discussed in Chapter 4.

Quality: the *AIDA* database is maintained by Bureau Van Dijk, a company that is part of the Moody's network and provides trustworthy informations used in many studies. The fields that are collected in the database are prepared according to EU directives and the Italian law on accounting, so they are easily comparable between sources and clearly defined. However, on the quality of the data one important issue was encountered: many fields presented a very large amount of missing data. Many fields actually had the majority of their data missing, and this made it necessary to drop fifteen of them from this analysis, as was discussed in paragraph 4.4. Luckily, the huge amount of firms in the dataset has reduced the impact of this problem, but still this led to the necessity to drop some variables, like *'Return on sales (ROS) %'* and *'Cost of debt (%) %'*, that at a first glance could have been considered very important for this model to work well.

Generality: this criterion has to be evaluated in conjunction with the general purpose of this study, which aims at giving a tool that is useful in particular for those firms that the European Commission calls "Small to Medium Enterprises". The chosen database has the advantage of being free from at least two sources of selection bias: it collects, in fact, every firm that meets the SME criterion regardless of whether they were granted some form of credit by a bank or an agency or if they applied for it (a problem that is present in all of the other studies on this subject). It, however, collects only Italian firms (a choice that was made for ease of access to the data and to avoid an over-complex data cleaning phase), and this is of course a potential source of bias that may be generated if the results if this study are applied to foreign firms. One more potential source of bias could be the selected time frame: in this case, the whole period ranging from 1989 to the present day was considered (the database collects firms whose last account closing dates goes back to at most 1994 (Table 1), and as the last 5 years of available accounts were used for this analysis, this makes the data go back to 1989), anyway, someone could argue that because of shifts in the Italian macroeconomic environment, or any other cultural or legal change, the reasons why firms go bankrupt today are different from twenty or more years ago. In this case, the choice to not introduce macroeconomic variables in the model and prioritize abundancy over generality was made, but for sure this is an aspect that must be investigated in the future.

C*onciseness:* this is probably the hardest criteria to evaluate in an objective way, but some things can still be said about it. As discussed earlier, the variables chosen for this model are computed directly on the accounts of the firms and contain some the most widely used indicator of the financial and economic situation of an enterprise. However, the fact that following the quality criterion many of these indexes had to be excluded hampers the representativeness of the ones that remain in a significant way. In this case, the researcher has to decide on where to position the dataset in a trade-off between conciseness and quality that could lead to a bad predictive ability of the model if the decision to go too far in any of the two directions is taken. This is a very delicate choice, and even just by looking at the informations of TABLE X (the comparative analysis of the variables) it can be easily understood that a consensus on how to tackle this issue is hard to find.

Before the discussion of the actual experiment begins in the next chapter, it is important to remark that a lot of arbitrary choices must be made when working on this kind of a problem. Many of those were made in the selection of the dataset, and many more will be made in the construction of the neural networks that will analyze the data. Of course, the result of the experiment will be dependent on the set of choices and assumptions that were made, and even if it would be outside the scope of this work to explore all the possible alternatives, it is important to read the rest of the work keeping in mind these considerations.

Chapter 4

The Experiments

5.1

Fine tuning ANNs by trial and error

It is now time to test all of the assumptions and ideas laid out in the previous chapters and see how they fare against the available data. In this chapter, two main categories of model will be tried out: a Multi Layer Perceptron (MLP), one of the oldest and most straightforward neural network architectures available; and a Recurrent Neural Network (RNN), a more recent development of the ANN technology that is particularly suited to deal with time-based problems.

The application of a neural network to a problem is inherently a trial and error procedure. A neural network is, as discussed in Chapter 1, just a complex system that empirically finds the best solution in a neighborhood of the location it finds itself in (the initial value of the weights and biases), and this location is determined by random chance. But this is not the only profile of randomness in an ANN that makes it so important to explore many different alternatives when training such a model. For this reason it is a good idea, before analyzing the results of the experiments, to list and discuss the most important among the so called "hyper-parameters" that must be empirically optimized by the researcher:

- *Network structure*

  Artificial Neural Networks come in many shapes and sizes, and while sometimes they can be used interchangeably, more often than not the choice of the right architecture is crucial to get good results from the experiment [Mitchell 1997]. A choice has to be made on a double level: first, a general family of architectures must be chosen. Popular choices are MLPs, RNNs, Convolutional Networks, Deep Belief Networks, all used depending on the task. Secondly, the architecture-specific parameters must be analyzed. In the case of MLPs, this means choosing the appropriate number of hidden layers and neurons per layers, in an RNN the number of steps the network must be unrolled on also needs to be defined, and so on as every architecture has its own quirks.

- *Initial location of weights and biases*

  At the core of the principles that regulate ANNs training, there is the update of the weights and biases that connect the neurons to one another. Obviously to have the possibility to update those parameters it is necessary to have them initialized with some kind of value. These values are generally extracted from a random distribution, bearing in mind that different initial parameters can lead to different final solutions [Hertz et al. 1991]. This holds true unless the gradient on the parameters space is monotonic, which for every application except the most trivial (like the case of linearly separable data) would be an incredibly strong and unlikely assumption.

- *Activation function*

  The activation function is what gives ANNs their incredible qualities in the analysis of non-linear data. A neural network which does not use an activation function is just an unnecessarily complex linear model. The three most popular choices in this regard are: sigmoid, hyperbolic tangent, and ReLU [Nair and Hinton 2010]. The latter is the one that research seems to have proven to be the best in almost the totality of cases [Schmidt-Hieber 2017] and so it is the only one used in this study.

- *Output shape*

  The output shape can be essentially one of two options: either you have one single output neuron, and base your predictions on its final value (particularly useful for regressions, but can be used in some cases for classification too), or you have one neuron per class you want to divide your data in, and use softmax to get the final result.
- *Loss function*

  A neural network would be totally blind to the objective of the research if it did not have a loss function: the loss function represents an analytical way to compute a value that estimates how close the network is to reaching its goal. The gradients that the optimizer uses to determine how to train the model are computed directly from the loss function.
- *Regularization and Dropout*

  Artificial Neural Networks, like every statistical tool, can suffer from overfitting. This is especially dangerous in ANNs as the high number of parameters used can often reduce the training process to the mere compiling of “lookup tables”. Two techniques have proven to be especially useful in reducing overfit: L2 regularization which aims at reducing the probability that any single parameter is too decisive for the final prediction by appropriately modifying the loss function; and Dropout [Srivastava 2014], a technique that randomly “turns off” neurons during the training phase, de facto training multiple ANNs at the same time and recombining them in the test phase.
- *Optimizer and learning rate*

  There are many options to chose from when deciding which optimizing algorithm to use. The very first algorithm used for this purpose was Stochastic Gradient Descent, but this has been proven to get stuck in saddle points and in general to have slow rates of convergence [Ruder 2016]. Other techniques have then been proposed, and one very common choice nowadays is the ADAM optimizer [Ba and Kingma 2015]. Once the optimizer is chosen, an appropriate learning rate (which represents the length of the step the optimizer takes at each epoch), must be identified.

- *Number of epochs, test and validation size*

  At some point, the training phase must be concluded and the results analyzed against an independent dataset. Obviously, a choice must be made on when to stop training (the total number of epochs to run), being careful that if this choice depends on the test set (for example, if the training is stopped when the error on the test set starts to increase) this may lead to a bias in the estimated accuracy, creating the necessity to have another independent series of data called "validation set" to compute the network's accuracy on. An appropriate size for the training, test and validation sets must be chosen.

Of course, this takes for granted that all the alternatives in the shape and collection of the input data have already been discussed, as was already done in Chapter 3 where the high amount of alternatives that arise in that problem too has been discussed.

Let us discuss in more detail how each one of those dimensions has been dealt with in this study:

Network structure: two main types of architecture were tried out, a Multi Layer Perceptron and a Recurrent Neural Network. The first one was used as a benchmark on the problem, a way to see how the most common architecture could fare against the collected data, and more in general as a first experiment to see whether ANNs could actually extract any meaningful informations from this problem. The input data for the MLP model only consists of the last available account year of every firms, as a consequence of the fact that MLPs are not very good at modelling time-series, and the fact that it could actually be interesting to see how good a model that only sees a static image of a firm can get. The RNN architecture was chosen because of the time-based nature of the problem that this study deals with. RNNs are in fact inherently able to model very well data that has an ordered structure (which means data that has a clear beginning and an end, which cannot be reversed), such as text or time-series, and for this reason it is surprising to find that, to the knowledge of the author, among all the studies that dealt with the analysis of financial risk, none of them ever used this architecture.

In the MLP case, two hidden layers were used, and various numerosities of the neurons in each layers were tried out. In the RNN case, one hidden layer was used, varying the number of recurrent units that compose it.

Initial location of weights and biases: in both experiments, the initial value of the weights and biases was chosen by extracting values from a normal distribution with zero mean and unit variance. This choice helps in making the dimension of the parameters of the network the same as that of the input data (which, as stated in Chapter 3, were normalized with the same moments), in hope that this can help with the efficency of the training phase [LeCun et al. 1998].

Activation function: as stated above, there is wide consensus among researchers on the fact that "Rectified Linear Unit" (ReLU) function provides in general better results than the sigmoid or hyperbolic tangent. This is the reason why the only activation function used in this study is ReLU. Alternatives and improvements of ReLU such as Leaky Relu[9] [Maas et al. 2013] exist, but it was decided for simplicity not to try them.

Output shape: as this study is concerned with a classification problem (each firm is either classified as "failed" or "not failed") the most appropriate output shape consists of two neurons, one for the "failed" class

9 An alternative version of ReLU that has a very small slope for values smaller than zero, instead of being constant. This means that the gradient of the function is never zero, a property that can be useful in certain situations.

and one for the “non-failed” class. When the neural network is applied to the data of a certain firm, if the output neuron associated to the “failed” class outputs a value higher than the one given by the other output neuron, than that firm is predicted to be failed (the opposite is true if the values are inverted). The alternative model in which just one node is used as an output is better suited for “regression” models, which means that they are required to produce a value rather than a classification.

Loss function: many kinds of loss functions can be defined for such a problem, but one of the most used is the so called “cross entropy” function, whose formulation is:

$$L = -\frac{1}{n}\sum_{i=0}^{n}[y_i \ln(p_i) + (1-y_i)\ln(1-p_i)]$$

where

n is the number of firms in the training set,

y_i is the label of the i^{th} firm as extracted from the column *'failure'* of the dataset,

p_i is the first of the two values from the softmax output computed by the network on the account of the i^{th} firm.

This choice was made as this function grants faster learning in the initial stages of training (a property discussed in [Mitchell 1997], together with the fact that this kind of function is in most cases the best choice when trying to estimate probability distributions), with respect to the also very popular Quadratic Loss, which computes the euclidean distance between the output of the network and the desired prediction.

Regularization and Dropout: Dropout was introduced in both the models, as it is a highly suggested practice and pretty much every study dealing with neural networks uses it to prevent overfitting. The dropout probability was set to 0,5; a common choice as typical values range from 0,5 to 0,6. L2 regularization was also introduced in the loss function of every model, as a strong problem of overfitting, even when Dropout was applied, was encountered. This will be evident in the results that will be presented in the next sections. The β value used for regularization (which represents the "strength" of the effect) was set to 0,01 in every model.

Optimizer and learning rate: in this case too is possible to find a consensus in the literature as to which optimizer is best for general purpose tasks, and it is ADAM optimizer. This is indeed the optimizer that is always used in this study. Even if at the base of the functioning of ADAM there is the auto-updating of the learning rate, it still needs a value to begin with. In this study the value of 0,00001 was used for the MLP network (as preliminary trials showed that a very small value was needed to give some stability to the accuracy) while for the RNN model every experiment was conducted with four different learning rate values (0,1; 0,01; 0,001; 0,00001).

Number of epochs, test and validation size: given the large dimension of the dataset, there is little concern on the risks that usually arise when deciding the size of the training set. Usually, the size of the training set must be decided considering that if it is too small the risk is that it does not contain enough meaningful examples and so the final accuracy may be not satisfactory, while on the other hand if the training set contains most of the data this could cause the test and validations sets to be biased for the very same reason. Also, if the test and validation sets are too small, the variance of the accuracy computed on the could bee very high, and so the results may be not so meaningful. In this study, however, the size of the test set and that of the validation set is chosen to be 10% of the total number of cases in the dataset. This means that when the complete dataset is used, 72.216 firms are present both in the test and in the validation set, granting some degree of consistency in the measurements. With "number of epochs" is indicated the number of times a backpropagation phase ("step") is completed on the whole of the training set. If the data has to be split in batches because it can not fit entirely into memory (in both the experiments a batch size of 5000 was used), then in each epoch the number of steps completed is:

$$\frac{\text{numerosity of the training set}}{\text{batch size}}$$

The number of epochs to train the networks for is set to 2000 for the MLP model, as preliminary experiments showed that results stabilized themselves around 1000 or 1500 epochs (this issue will be discussed more in depth in the next section). It is set to 500 epochs for the RNN as results for this model are more stable and the training of this network took longer.

In Table 4 the choices made for this experiment are summarized in a table that can be used as a guide to navigate the results presented in the next section.

Table 4

	MLP	**RNN**
network structure	*2 hidden layers, in three configurations:* *50, 50* *100, 25* *100, 100*	*1 hidden layer, four different numbers of hidden neurons:* *20* *50* *100* *200*
initial weights and biases	*~ N(0, 1)*	*~ N(0, 1)*
activation function	*ReLU*	*ReLU*
output shape	*2 softmax nodes*	*2 softmax nodes*
loss function	*cross entropy*	*cross entropy*
regularization	*L2*	*L2*
dropout	*50%*	*50%*
optimizer	*ADAM*	*ADAM*
learning rate	*0.00001*	*0.1* *0.01* *0.001* *0.0001*
number of epochs	*2000*	*500*
batch size	*5000*	*5000*
test and validation sets	*10%, 10%*	*10%, 10%*

5.2
Software and hardware details

All the code used in this study was written in *Python*, a popular programming language used in particular for its applications in data analysis. The code used to produce the results of the following paragraphs is available in Appendix 5, while the code used for the operations described in Chapter 3 is available in Appendix 4. The *Python* version used is 3.5.2, 64 bit.

The neural networks were built and trained using *TensorFlow*, a machine learning package developed by *Google*, which offers bindings to *Python* and other languages and has lately become the standard choice in the field of neural networks research. This package was particularly useful as it provided an easy way to train the models on a GPU, rather than on the CPU, through the use of *NVIDIA CUDA* toolkit, which sped up the computations significantly.

In Table 5 the details of the PC architecture on which the networks were run are presented.

Table 5

CPU	*Intel Core i7-4710HQ, 2.50GHz*
RAM	*4 GB*
GPU	*GeForce GTX 850M*
GPU-RAM	*2 GB*

5.3
Experiment 1: Multi Layer Perceptron

In the first experiment an MLP architecture was used. MLP is perhaps the most widespread architecture that implements the neural

networks paradigm, and will be used as a benchmark to see what kind of results can be expected from the dataset that is used in this study.
Also, as the structure of Multi-Layer Perceptrons is not very good at modelling time series data, the input for this network will be constituted only of the last available year of account for each firm. This will also be useful to highlight the improvement in accuracy that can be gained when introducing a time dimension to this problem.

Three different MLP configurations were tried out: 50 neurons on the first and second hidden layer, 100 neurons on the first and 25 on the second (in hope that the second could serve as a “convolution” over the features of the problem), and 100 neurons on both layers. It is important to try different dimensions for the hidden layers as there is a trade-off between wider layers that can learn many features and build complex decision rules, and smaller layers that are less prone to overfitting as they have less parameters to build lookup tables with.
Each of these configurations was trained four different times, every time with a new shuffling of the datasets and a new extraction of weights and biases. 2000 epochs were run, and for every session the result with the highest accuracy on the training set was picked to be shown in Table 6, together with the corresponding value in the training and validation sets. Each training session took around two hours, the best result on the validation set for every configuration is highlighted.

From the results of Table 6, it is clear that the network with 100 neurons on the first hidden layer and 25 on the second outperformed the others, both in terms of the mean accuracy and the best overall result.
It must be noted that this result is not due to a reduced overfitting with respect to the *100-100* network, as in both cases the difference between the training set accuracy and that of the test set ranges from 0,1% to 0,2%. This

supports the hypothesis that a second hidden layer smaller than the first one helps the network in making convolution that could represent some high-level abstractions useful in making its predictions.

Table 6

Hidden layers structure	***Training set accuracy %***	***Test set accuracy %***	***Validation set accuracy %***	***Mean validation accuracy %***
50, 50	*84,5*	*84,7*	*84,6*	*84,53*
	84,8	*84,8*	*84,8*	
	83,9	*83,8*	*83,7*	
	85,1	*85,1*	**85,0**	
100, 25	*85,1*	*85,2*	*85,2*	*85,27*
	85,2	*85,2*	*85,2*	
	85,3	*85,5*	**85,4**	
	85,2	*85,1*	*85,3*	
100, 100	*85,0*	*84,9*	*84,9*	*85,05*
	85,2	*85,4*	*85,1*	
	85,1	*84,9*	*84,9*	
	85,3	*85,2*	**85,3**	

In general, in this model the problem of overfitting is almost non existent, and sometimes the error computed on the test and validation sets is even smaller than the error on the training set. This is even more clear in Graph 3, where some examples were picked from the training of the networks, showing how the accuracy increased with the number of epochs: the separation between the line representing the training and test accuracy is visible, barely, only in some cases.
In Graph 3 is also evident how the performance of the network is unstable during training, especially in the first thousand of epochs. In particular, the wildest behaviour comes from the *100-25* structure. It would be interesting to know the exact reason of this phenomenon, as it could provide useful

insights on how the network is operating, but this study could not find any useful explanation.

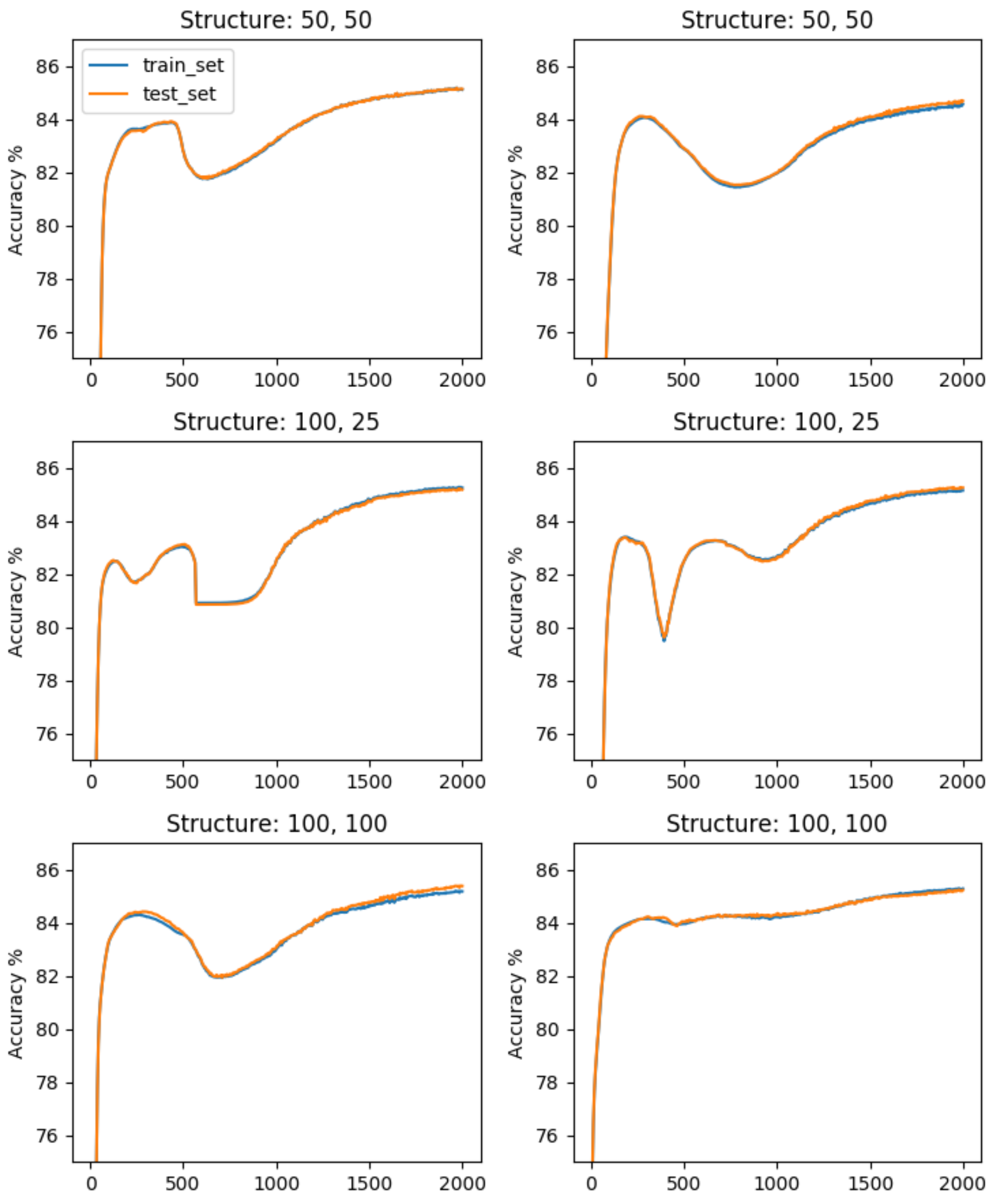
Before any assessment can be made on the quality of the results presented in Table 6, some more informations must be provided. The dataset used, in fact, is highly unbalanced: of the 722.160 firms in it, only 19,1% are failed. This means that an algorithm that simply predicts every firm to be not failed would get an accuracy score of 80.9%.
Under this light, the best overall result obtained by the MLP model (85,4%), while being for sure an improvement, does not seem very far from the results of a simple strategy like the one described above. To have a better understanding of this issue, Table 7 presents the error values of the best result of each structure and breaks them down into Type 1 and Type 2 error. Here, Type 1 error is the share of firms incorrectly classified among all the firms that failed, while Type 2 error is the is the number of "healthy" firms that were misclassified.

Table 7

	Validation accuracy ***%***	***Type 1 error*** ***%***	***Type 2 error*** ***%***
50, 50	*85,0*	*68,4*	*2,4*
100, 25	*85,4*	*64,8*	*2,7*
100, 100	*85,3*	*64,0*	*3,2*

Table 7 highlights the fact that the MLP adopts a strategy that is very similar to "always guess not-failed". Indeed, the error rate on the healthy firms is very low as generally the networks predicts everything as not-failed, while the error on the failed is around two thirds of the total. Even though such a result is a sizeable improvement over the 100% error rate expected from that simple strategy, this is still a very poor performance

Graph 3



Some examples from the training of different types of MLPs. On the horizontal axis the number of epochs the training was run for. On the vertical axis the value of the Train set and Test set accuracy.

especially in the field of credit risk, where any single Type 1 error has an economic impact that is generally much larger than errors committed in the Type 2 category.

In the next section a strategy to deal with this issue, that will produce better results, will be discussed.

5.4
Experiment 2: Recurrent Neural Network

For the second experiment, a RNN architecture was chosen. As discussed above, Recurrent Neural Networks are particularly suited to solve problems that are presented in a sequential way, like data that is collected on various subsequent years. In this case, the input is constituted of the whole dataset collected, with five different years of accounts for every firm. The input for the features is then a tensor (multi dimensional matrix) of shape $722160\times5\times55$, while the input labels are stored in a vector with 722.160 elements in it.

This experiment is divided in two phases: in the first one, the same exact dataset used for the MLP experiment will be used, and the differences in the results will be discussed; in the second a strategy to deal with the high Type 1 error will be implemented.

For the first phase, two recurrent architectures were tried out: one with 50 recurrent nodes in the hidden layer, another with 100 recurrent nodes in it. Both of them were trained a total of four times, two time with 0.01 learning rate and two times with a 0.001 learning rate. The best of the two results per each combination is reported in Table 8. Training took about forty minutes per network. The dataset was shuffled at the beginning of each training session in the same way as the previous experiment.

These results show a decisive improvement over the MLP model, both in the overall accuracy, which was increased by up to 2,7 percentage points, and in the Type 1 error, reduced by more than 20 percentage points. The accuracy on the healthy firms slightly worsened, but this should not cast a shadow on the fact that the overall result is much better, and most importantly now the majority of failed firms are classified correctly.

Table 8

Neurons	***Learning rate***	***Validation accuracy %***	***Type 1 error %***	***Type 2 error %***
50	*0.01*	*87,9*	*42,8*	*4,9*
	0.001	*88,1*	*42,8*	*4,7*
100	*0.01*	*87,9*	*42,1*	*4,9*
	0.001	*88,0*	*41,5*	*5,0*

This is not yet a result that is satisfactory enough to be used in any real world decision making, but it highlights one very important point: the Recurrent Neural Network with 5 years of accounting as input has a predictive ability that is much higher than that of the MLP model. This was expected, as it is a more complex structure to which more data were fed, but it suggests that future research should begin to investigate the possibilities of this model and its possible applications, as to the current date no study on this has been published.

Table 9

	Train accuracy %	***Test accuracy %***	***Overfitting (Train minus Test)***
50 neurons 0,01 learn. rate	*89,3*	*88,0*	*1,3*
50 neurons 0,001 learn. rate	*88,8*	*88,1*	*0,7*
100 neurons 0,01 learn. rate	*88,8*	*88,2*	*0,6*
100 neurons 0,001 learn. rate	*88,9*	*88,0*	*0,9*

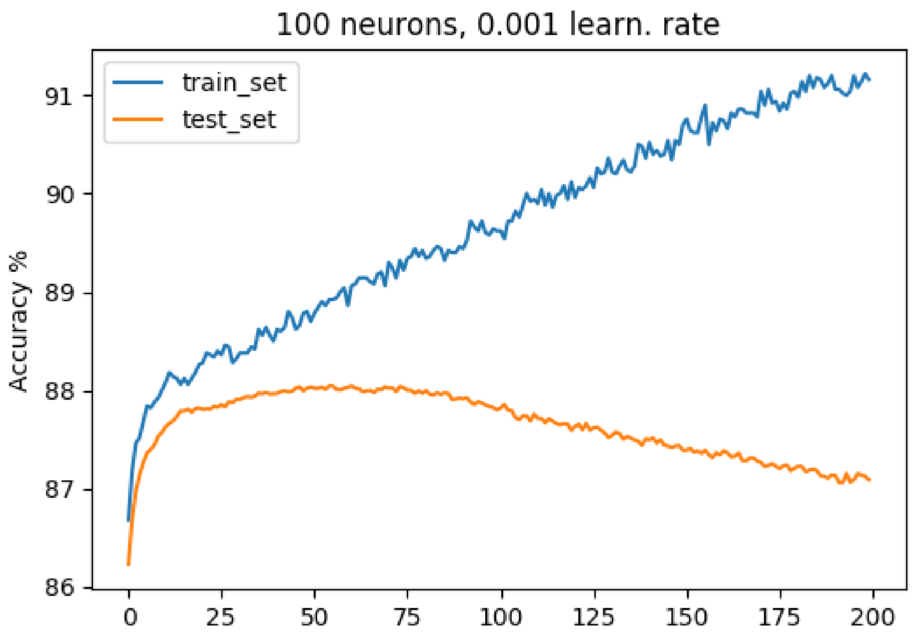
One important fact that emerged from this first exploration of the application of the RNN model is that in this case the problem of overfitting, which was practically non-existent in the MLP case, is clearly noticeable, as

Table 9 summarizes. It would be interesting to perform a careful analytical enquiry on the reasons of this phenomenon, but for now it can be hypothesized that the fact that the RNN network has a more complex structure, with more parameters to optimize, makes it easier to adapt the shape of its parameters to every single feature of the dataset, rather than having a more smooth surface that has an harder time accommodating outliers.

From Graph 4, which depicts the training results of one of the RNN networks with 100 neurons, the phenomenon of overfitting is even more evident: the training and test accuracy rise together up until about epoch 25, from that point the accuracy of the training set increases linearly, while that of the train set slows down, reaching its maximum around epoch 50 and then declining. This is a clear indicator of the fact that from epoch 50 onwards, the network is not learning anything useful anymore on the actual indicators of the possibility of a firm to fail, and it is instead just building a lookup table of the train set.

It is now time to present the result of the second phase of the RNN experiment. The objective of this phase is two-folded. First, try a method that reduces the imbalance between Type 1 and Type 2 error. Secondly, fine tune the network in detail, by exploring the space of the hyper-parameters as much as possible.
There are various strategies that could be adopted to solve the problems arising from an unbalanced dataset, in this case a very simple approach was adopted: the dataset was divided in two groups, the failed firms and the healty firms. The group containing the healthy firms was shuffled before each training session, and a number of entries that is equal to the number of failed firms in the dataset was extracted. Then, those were appended to the

Graph 4



Graph of the training phase of the RNN with 100 neurons and 0,001 learning rate. On the horizontal axis the number of epochs, on the vertical axis the accuracy.

list of failed firms and this new rebalanced dataset was shuffled again. The training, test and validation sets were then extracted from this new selection

of data. This procedure granted the elimination of the problem of class imbalance, and made each set of data be composed exactly of 50% failed and 50% not-failed firms.

Then, four different RNN structures were trained, with 20, 50, 100 and 200 neurons in the hidden layer. Each structure was trained with four different values for the learning rate: 0.1, 0.01, 0.001, 0.0001. Each combination of number of neurons and learning rate was trained for a total of ten times, which results in a total of 160 networks trained. Training took about forty minutes for the RNN with 20 neurons, and about a hour and a half for the RNN with 200 neurons.

For each network, the best result on the train set among the 200 epochs was picked, and the corresponding validation error was computed. Table 10 presents a summary of those results. The mean validation error is the average value of the ten networks for each combination, while the "best validation error" and the Type 1 and Type 2 errors are computed on the network with the lowest validation error among those ten.

Table 10

Hidden neurons	***Learning rate***	***Mean validation accuracy %***	***Best validation accuracy %***	***Type 1 error in the best network %***	***Type 2 error in the best network %***
20	*0.1*	*80,9*	*81,1*	*18,9*	*18,9*
	0.01	*81,7*	*82,0*	*18,1*	*17,8*
	0.001	*81,9*	*82,2*	*17,2*	*18,4*
	0.0001	*81,1*	*81,3*	*20,0*	*17,3*
50	*0.1*	*81,1*	*81,5*	*18,9*	*18,1*
	0.01	*82,0*	*82,3*	*18,1*	*17,4*
	0.001	*82,1*	***82,4***	*17,4*	*17,7*
	0.0001	*81,7*	*82,1*	*18,3*	*17,5*
100	*0.1*	*81,0*	*81,4*	*18,1*	*19,1*
	0.01	*81,9*	*82,1*	*18,2*	*17,6*
	0.001	*81,8*	*82,2*	*18,6*	*16,9*
	0.0001	*81,9*	***82,4***	*18,9*	*16,2*
200	*0.1*	*80,8*	*81,1*	*21,6*	*16,2*
	0.01	*81,9*	*82,1*	*16,4*	*19,5*
	0.001	*82,0*	*82,2*	*18,8*	*16,9*
	0.0001	*81,6*	*81,6*	*19,7*	*17,3*

The conclusion that can be drawn from the data presented in Table 10 is that the RNN network with the rebalanced dataset sacrifices Type 2 accuracy (from a best error of 4,7% to a best result of 16,2%) to have a large improvement on Type 1 error (from 41,5% to 16,4%). This is exactly the target that this experiment was aiming for: as the economic impact of a company that fails when it was granted credit is much larger than the impact of not granting credit to a company that would have repaid it, in a credit risk model the preference is strong for a better accuracy on the former kind of mistakes.

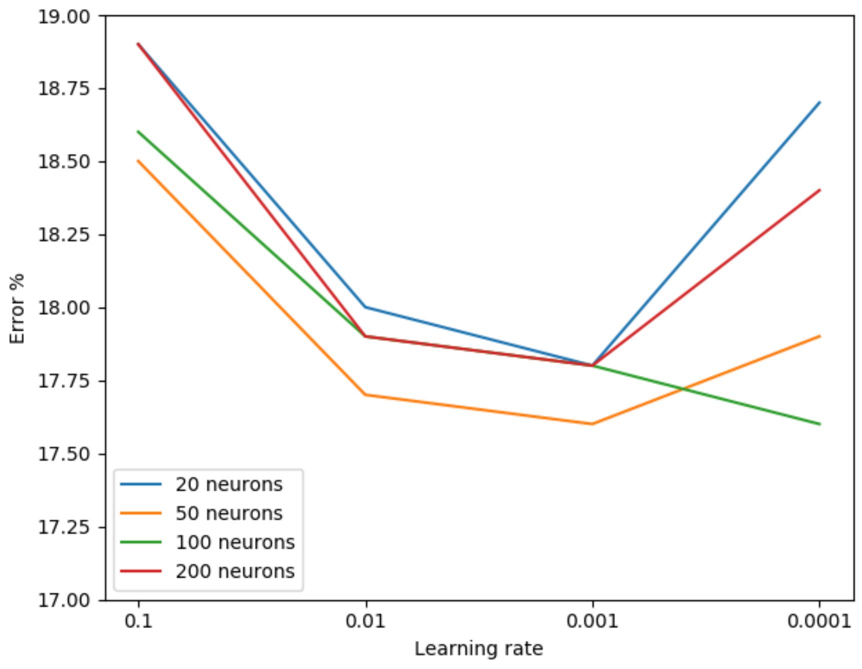
In these experiments, the ratio between Type 1 and Type 2 seems to depend strongly on the ratio between failed and healthy entries in the dataset, and this is very important as it means that it is possible to fine-tune the type of error based on the profile of the final user of the network predictions: if the user has a strong aversion towards the risk of granting credit he/she can use a network trained on a high percentage of failed firms, while other users with different needs can adjust this percentage accordingly.

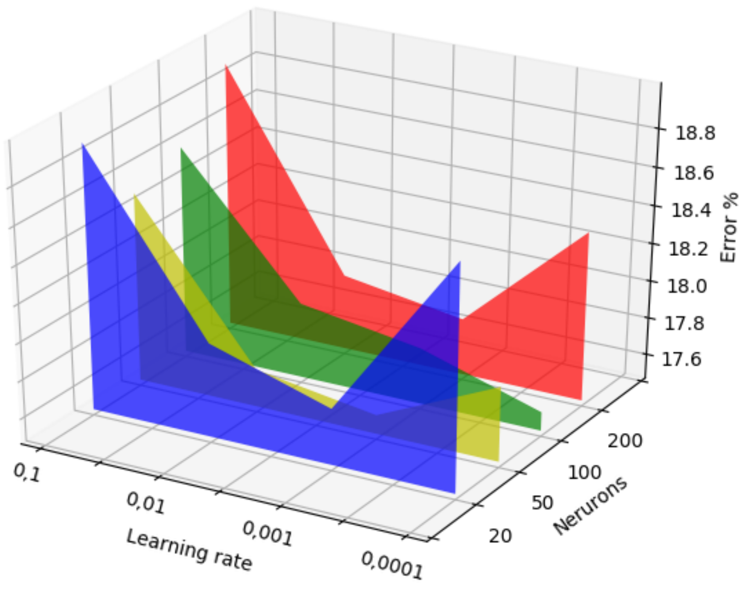
One more important result of this last set of experiments is a better understanding of dependence of the achieved accuracies on the hyper-parameters. Graph 5 depicts the relation between the parameters that were used and the accuracy obtained. Some trends can be identified:

For every configuration of the hidden layer except the one with 100 neurons, the 0,01 and 0,001 learning rates performed much better, with the 0,001 rate being the best in terms of accuracy achieved on the overall dataset. The only exception is the 100 neurons networks, in which the 0,0001 rate was the best, showing a kind of monotonic relation between the learning rate and the error.

The hidden layer configuration that seems to work better in almost every instance is the one with 50 neurons. This configuration is the absolute best on every learning rate except the 0,0001, in which the network with 100 hidden neuron outperforms it. The worst performance is the one of the 20 neurons architecture. These result highlight the fact that a right balance must be found between simpler networks with a lower number of neurons that can be trained in an easier way but may not be large enough to model complex relations, and larger networks that can accommodate complex “ideas” but are much harder to train and way more likely to overfitt.

Graph 5





In the top graph: the results of the RNN experiments with the rebalanced dataset. On the horizontal axis the learning rate, on the vertical axis the best error for each learning rate and number of neurons.
In the bottom graph: the same data and variables in a 3D representation.

The overall best result on the validation set (82,4% error) is shared between the network with 50 hidden neurons and 0,001 learning rate and the one with 100 hidden neurons and 0,0001 learning rate. This is not much different than the result on the unbalanced dataset (85,4% validation error),

but the two should not actually be compared as they come from input data that have a very different nature. As discussed multiple times in this work, in a credit risk setting the share between Type 1 and Type 2 error is way more important than the overall error rate. It is worth mentioning that while the 50 neurons network achieves its 82,4% result with 17,4% Type 1 error and 17,7% Type 2 error (pretty much identical values) the one with 100 hidden neurons has a higher Type 1 error (18,9%) both compared to the same quantity of the smaller network and to its own Type 2 error (16,2%).

5.5
Quality of the results

When comparing the accuracies scores obtained in this chapter to the ones found in literature, it may be worth considering that the previous chapter laid out very strict rules for the processing of the dataset, for example by reducing the overall features to the bare minimum and deciding not to proceed with the elimination of outliers from the data. The hope is that these tighter constraint give more “robustness” to the results obtained here, which means that even if the accuracy obtained is not the best in literature, the “value” carried by the result is nonetheless very high.

With this said, in Louzada et al. [2016] it is possible to find a summary of the best results obtained on the two most popular datasets in the field of credit risk, called the “Australian” and the “German” datasets [Bache, Lichman 2013]. Both of them collect data from people who applied for loans, the former containing 1000 instances while the latter 690 (about three hundred times less instances than the ones used in the RNN model studied in this chapter). On the Australian dataset, performances range from 81% to 98%, proving to be an easier dataset to train on compared to the German dataset where performances range from 72% to 85%. The model presented in this study then seems to align with the highest results of the models for the German dataset and the lowest among those of the Australian dataset.

Among the studies that used corporate failures as their dependent variable, Angelini et al. [2008] uses an extremely small dataset with 76 firms in it, and produces an overall error of 8,6%, with 0% Type 1 error and 13,3% Type 2 error. The small sample size and the 0% Type 1 error however cast a doubt on their reliability, as they could be the result of the reporting procedure that was used. Quoting the paper: “a large number of tests has been performed”, and the best of those test was reported. With such a small sample size those tests those error rates could have been achieved just as a result of random chance.

Pacelli and Azzolini [2010], on the other hand, used data consisting of 273 firms, but instead of classifying them in two classes, those were classified either as “Safe”, “Vulnerable”, or “At Risk”. The confusion matrix that results from this approach shows an error of 65,2% for the “At Risk” class and 15,8% for the “Safe” class, presenting the same problem of class imbalance encountered in the first stages of the experiments discussed in this chapter.

Conclusions

The field of machine learning applied to credit risk evaluations has seen a flourishing production of results that demonstrate its utility and in general the ability to predict insolvencies in a consistent and reliable way. However, it looks like there are still large margins of improvement, especially as we now live in a period of great discoveries in artificial intelligence and its applications. The implementation of the new paradigms and models that AI researchers all around the world are creating will boost the research in this field too. The hope is that this work has shed a light on the current status of the research and given some insights on possible new directions it could take.

The huge size of the dataset used is a unique feature among published researches in this field, which is often constrained to used at most some hundreds of data points. Notwithstanding this, the results that were obtained are consistent with those found in literature. This robustness in the results adds confidence in the fact that this kind of studies produces useful information when applied in real world scenarios.
The first architecture that was experimented (Multi Layer Perceptron) did not prove to be very effective, as it adopted a decision rule that was very similar to the trivial strategy of always guessing firms to be "healthy". This made it necessary to adapt the model and the dataset to the problem at hand, and this was done in two ways: first, a different and more complex

architecture was used as the learning model; secondly, the dataset was rebalanced through a random process.
The first of these two methods consisted in the implementation of a Recurrent Neural Network, a structure that is best suited to analyze time-series data, to which the last five years of accounts available for each firms were fed. This model, which is a novelty in the field of neural networks for credit risk evaluation, proved to be very effective and increased the accuracy score significantly.
The second method involved the reduction of the number of healthy firms in the dataset to eliminate class imbalance. The large number of data collected granted the possibility to operate this reduction without sacrificing the explanatory power of the model. This technique produced the exact results that were expected, making the network almost equally sensitive to Type 1 and Type 2 errors.

The final accuracy values that were obtained can be considered satisfactory, especially in light of the strict limitations that were imposed on the dataset: the fact that the study focused on SMEs, the deletion of many features and data based on missing values, and the fact that the choice to keep outliers in the collection was made, all contributed to impose conditions that likely made it much harder for the networks to make good predictions. Notwithstanding this, the final accuracy scores are in line with those found in literature, and this suggests that the use of new techniques like the ones tried out in this study could produce even better results in the future.

Bibliography

Altman, Sabato 2007 *Modelling Credit Risk for SMEs: evidence from the U.S. market,* Abacus, September 2007, Vol.43(3), 332-357

Angelini et al. 2008 A neural network approach for credit risk evaluation, The Quarterly Review of Economics and Finance 48 (2008) 733–755

Ba, Kingma 2015 ADAM: a method for stochastic optimization, arXiv preprint arXiv:1412.6980, 2015

Bache, Lichman 2013 UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml

Corazza, Funari, Gusso 2016 An evolutionary approach to preference disaggregation in a MURAME-based creditworthiness problem, Applied Soft Computing 29 (2015) 110–121

Cybenko 1989 Approximation by Superpositions of a Sigmoidal Function, Mathematics of Control Signals Systems, 1989, 2, 303-314

Hertz et al. 1991 Introduction to the theory of neural computation, Santa Fe Institute Series, 1991

Kaiser 2014 Dealing with missing values in data, Journal of Systems Integration 2014/1

Khashman 2010 Neural networks for credit risk evaluation: investigation of different neural models and learning schemes, Expert Systems with Applications 37, 2010, 6233–6239

Kim 2011 Prediction of hotel bankruptcy using support vector machine, artificial neural network, logistic regression, and multivariate discriminant analysis; , The Service Industries Journal, 31:3, 2011, 441-468

LeCun et al. 1989 Backpropagation applied to handwritten ZIP code recognition, Neural Computation 1, 1989, 541-551

LeCun et al. 1998 Efficient BackProp, Neural networks: Tricks of the trade. Springer, 1998. 9-50

Lee, Chao 2013 A multi-industry bankruptcy prediction model using back-propagation neural network and multivariate discriminant analysis, Expert Systems with Applications 40, 2013, 2941–2946

Lipton et al. 2015 A critical review of Recurrent Neural Networks for sequence learning, arXiv preprint arXiv:1506.00019, 2015

Lopes, Ribeiro 2011 A robust learning model for dealing with missing values in many-core architectures, Proceedings of the 10th International Conference on Adaptive and Natural Computing Algorithms – Part II (ICANNGA 2011), LNCS 6594, Springer-Verlag, 2011, 108–117

Louzada et al. 2016 Classification methods applied to credit scoring: Systematic review and overall comparison, Surveys in Operations Research and Management Science 21 (2016) 117–134

Maas et al. 2013 Rectifier Nonlinearities Improve Neural Network Acoustic Models, *Proc. icml*. Vol. 30. No. 1. 2013

Merton 1974 On the pricing of corporate debt: the risk structure of interest rates, The Journal of Finance, Vol. 29, No. 2, 1974, 449-470

Mitchell 1997 Machine Learning, McGraw-Hill, 1997

Nair, Hinton 2010 Rectified Linear Units improve restricted Boltzmann Machines, Proceedings of ICML-10, 2010

Okun 1962 Potential GNP: its measurement and significance, Cowles Foundation Paper 190, 1962

Pacelli, Azzolini 2010 An Artificial Neural Network approach for credit risk management, Journal of Intelligent Learning Systems and Applications, 2011, 3, 103-112

Qiao et al. 2017 Gradually updated Neural Networks for Large-Scale image recognition, arXiv preprint arXiv:1711.09280, 2017

Rosenblatt 1957 The Perceptron: a perceiving and recognizing automaton, report 85-460-1, Cornell Aeronautical Laboratory Inc., 1957

Ruder 2016 An overview of gradient descent optimization algorithms, arXiv preprint arXiv:1609.04747, 2016

Schmidhuber 2014 Deep learning in neural networks: an overview, Neural Networks 61, 2015, 85–117

Schmidt-Hieber 2017 Nonparametric regression using deep neural networks with ReLU activation function, arXiv preprint arXiv:1708.06633, 2017

Srivastava 2014 Dropout: a simple way to prevent Neural Networks from overfitting, Journal of Machine Learning Research 15, 2014, 1929-1958

Sukhbaatar, Fergus 2014 Learning from noisy labels with Deep Neural Networks, arXiv preprint arXiv:1406.2080, 2014

Tuckova, Bores 1996 Influence of the number of the features with the neural network function, Radioengineering Vol.5, No. 1, 1996

Vasicek 1977 An equilibrium characterization of the term structure, Journal of Financial Economics, 5, 1977, 177-188

Welch 1947 The generalization of Student's Problem when several different population variances are Involved, Biometrika, Vol. 34, No. 1/2, 1947, 28-35

Wendemuth et al. 1993 The effect of correlations in neural networks, Journal of Physics and Applied Math, 1993, 3165-3185.

Wenzelburger et al. 2013 Implications of dataset choice in comparative welfare state research, Journal of European Public Policy, 20:9, 1229-1250

Wong, Sherrington 1993 Neural networks optimally trained with noisy data, Physical Review, Vol.47 No.6, 1993

Yu et al. 2008 Credit risk assessment with a multistage neural network ensemble learning approach, Expert Systems with Applications 34 (2008) 1434–1444

Zhao et al. 2015 Investigation and improvement of multi-layer perceptron neural networks for credit scoring, Expert Systems with Applications 42, 2015, 3508–3516

Appendix 1

Links to the dataset

At the following address the data and the code used for this study are available:

https://drive.google.com/open?id=1VejrDXJ6AUIe1P13g5LFJfIBVd1tSNfH

The folder is organized as follows:

- Dissertation (main folder)
  - Dataset
    - cleaned_downloads
    - cleaned_revenues
    - graphs_images

    cleaner.py

    downloads_converter.py

    numpify_dataset.py

    reader.py

    indexes definition.odt
  - Model: RNN

    RNNtrain_and_use.py

    original_numpyfied_database.pickle
  - Model: standard MLP

    Network Use 0.2.py

    NetworkTraining.py

    numpyfied_database_year_0.pickle

Dataset contains the Python files used to convert the *.csv* files to Python compatible formats, and to perform all the transformations described in Chapter 3.

cleaned_download and cleaned_revenues contain the *.csv* files with every data downloaded from the *AIDA* database.

graphs_images contains the graphs in image format of the distribution of the values for every column of the downloaded database. The graphs whose filename ends in “_c” are those extracted from the complete dataset, before the deletion of any row. The graphs whose filename ends in “_d” are the ones computed after the rows were deleted as described in Chapter 3.

Model: RNN and Model: standard MLP contain the files necessary to run the experiments of Chapter 4, and their results.

indexes definition.odt contains the definition of the formulas used by the *AIDA* database to compute all of the balance sheet indexes that were used in this study as the explanatory variables.

Appendix 2

Missing values distribution

The following table presents the number of missing values for every year of every variable of those that were kept in the final version of the dataset.

Name	***Missing values***				
	Last Year	***Last year -1***	***Last year -2***	***Last year -3***	***Last year -4***
'Profit (loss) EUR'	67	120	746	820	1665
'Total assets EUR'	0	0	0	2	1
'Total shareholder's funds EUR'	1219	1409	1458	1485	1543
'Return on asset (ROA) %'	4158	1639	1023	792	778
'Return on equity (ROE) %'	166015	136699	121977	112239	103100
'Liquidity ratio'	62948	57516	55173	54019	56253
'Current ratio'	81155	76234	73250	70962	72253
'Current liabilities/Tot ass. %'	7817	6468	6783	7228	9109
'Long/med term liab/Tot ass. %'	7984	6719	7235	8092	9961
'Tang. fixed ass./Share funds %'	31192	33763	35886	37830	39136

'Leverage'	2203	2364	2391	2438	2583
'Coverage of fixed assets %'	130099	103569	87199	75451	67664
'Interest/Turnover %'	121762	117778	110249	105492	114289
'Share funds/Liabilities %'	183726	163201	153753	150575	159095
'Net Financial Position EUR'	309917	167777	161329	115836	107037
'Debt/Equity ratio %'	310569	168620	162095	116785	108113
'Debt/EBITDA ratio %'	317668	175423	167984	122069	115068
'Total assets turnover (times)'	11423	9998	10988	12780	13161
'EBITDA EUR'	1232	1421	1493	1601	1798
'EBITDA/Sales %'	127677	117568	108455	103002	111457
'Number of employees'	18252	19293	23461	38318	50767
'Net working capital EUR'	1234	1448	1728	1502	1549
'Gross profit EUR'	6659	7839	11956	18354	18806
'Net short term assets EUR'	285232	157864	152345	11277	94749
'Share funds - Fixed assets EUR'	144509	113479	96109	84383	73954
'Revenues from sales and services th EUR'	12592	125198	125701	125953	126739
'Cash Flow EUR'	1260	1452	1727	1628	1804

Appendix 3

Definition of the dependent variable

As explained in Chapter 3, to define the dependent variable the values of the column *'Procedure/cessazione'* had to be checked when the value of *'Legal status'* was not enough to determine whether a firm could be considered failed or healthy.

In the following table all the possible values of *'Procedure/cessazione'* are listed, together with the corresponding classification performed on the firms that presented that value.

Value	***Classification***
'Transfer to another province'	*not-failed*
'Reasons provided for in the articles of association'	*not-failed*
'Winding up without liquidation'	*failed*
'Closure due to liquidation'	*failed*
'Closure due to bankruptcy or liquidatione'	*failed*
'Removal ex officio'	*not-failed*
'Winding up'	*failed*
'Initiation of cancellation procedure'	*not-failed*
'Composition with creditors'	*failed*
'Winding up and liquidation'	*failed*
'Voluntary liquidation'	*not-failed*
'Winding up in advance without liquidation'	*failed*
'Winding up and placing into liquidation'	*failed*
'Extraordinary administration'	*failed*
'Cancellation ex officio following creation of Chamber of Commerce, Industry, Craft Trade and Agriculture for Fermo'	*not-failed*
'Court order of cancellation'	*failed*
'Debt restructuring agreements'	*failed*
'Conclusion of liquidation'	*failed*
'Approved by all partners'	*not-failed*

'Cancellation ex officio following creation of Chamber of Commerce, Industry, Craft Trade and Agriculture for Monza'	*not-failed*
'Court ordered seizure'	*failed*
'Cancellation due to communication of allocation plan'	*failed*
'Lease of company'	*not-failed*
'Demerger'	*not-failed*
'Contribution'	*failed*
'Cancellation ex officio of registration with register of companies'	*failed*
'Removed ex officio because already included in the register of firms and not transferred to the register of companies'	*not-failed*
'Failure to re-establish multiple partners'	*failed*
'Fulfilment of company object'	*not-failed*
'Transformation of legal status'	*not-failed*
'Cancelled ex officio pursuant to Italian Presidential Decree no. 247 of 23 July 2004'	*failed*
'Reason not specified'	*failed*
'Conclusion of bankruptcy procedures'	*failed*
'Cancellation from the register of companies'	*failed*
'Cancelled ex officio pursuant to Article 2490 of the Italian Civil Code'	*failed*
'Court ordered liquidation'	*failed*
'Duplication'	*not-failed*
'Bankruptcy'	*failed*
'Removal ex officio following report by register of companies for the registered office'	*failed*
'Other reasons'	*failed*
'Closure due to bankruptcy'	*failed*
'Merger by incorporation into another company'	*not-failed*
'Supervening failure to meet the prerequisites for a company'	*failed*
'Court ordered administration'	*failed*
'Impossibility of fulfilment of the company object'	*failed*
'Precautionary seizure of shares'	*failed*
'Closure of local branch'	*not-failed*
'Cessation of any business'	*failed*
'Following expiry of time limits'	*not-failed*
'Post-bankruptcy composition with creditors'	*failed*
'Transformation into a registered office'	*not-failed*
'Winding up by official order'	*failed*
'State of insolvency'	*failed*
'Removal ex officio, lack of tax code (Article 21 of Italian Presidential Decree no. 605 of 29 September 1973, as amended)'	*failed*

'Transfer of firma'	*not-failed*
'Merger by incorporation of new company'	*not-failed*
'Compulsory administrative liquidation'	*failed*
'Cessation of business within the province'	*not-failed*
'Liquidation'	*failed*
'Controlled administration'	*failed*

Appendix 4

Data processing code

In this section the complete Python code used to process the data is presented. Every source-file is introduced by a brief explanation of its purpose.

downloads_converter.py

This code converts the raw downloads as produced by the script of the *AIDA* database into *.csv* files.

```python
import operator
folder = 'revenues_downloads'
files = os.listdir(folder)
def names():
    print(files[:10])
    split_names = [operator.itemgetter(1, 3)(name.replace('.',
'_').split('_')) for name in files]
    split_names = [[int(x[0]), int(x[1])] for x in split_names]
    split_names = sorted(split_names, key=lambda x: x[0])
    print(split_names[:10])
    for i in range(1, len(split_names)):
        if split_names[i-1][1] + 1 != split_names[i][0]:
            print(i, split_names[i-1][1])
def convert1():
    for serial in range(len(files)):
        selected_file = folder + '/' + files[serial]
        output = []
        with open(selected_file, 'r') as f:
            for i, line in enumerate(f):
                if i == 0:
                    labels = line[2:]
                elif i % 2 == 0:
                    output.append(line)
        output_name = str(serial) + '.csv'
        with open(output_name, 'w', encoding='utf8') as out:
            for i, line in enumerate(output):
                out.write(line)
        print('serial', serial, 'done')
def convert2():
        for serial in range(len(files)):
                selected_file = folder + '/' + files[serial]
                with open(selected_file, 'rb') as source_file:
                        with open(str(serial), 'w+b') as dest_file:
                                contents = source_file.read()
                                dest_file.write(contents.decode('utf-16').encode('utf-8'))
                print(serial, 'done')
convert2()
```

reader.py

Used to transform the files from *.csv* to Pandas dataframes.

```
import pickle
import csv
# df = pd.concat([pd.read_csv(str(i), dtype={'Tax code number': str,
'NACE Rev. 2': str}) for i in range(212)],
#                 ignore_index=True, verify_integrity=True)
with open('cleaned_revenues/0', 'r') as f:
        reader = csv.reader(f)
        i = next(reader)
data_types = {key: str for key in i[:14]}
for key in i[14:]:
        data_types[key] = float
data_types_revenues = {key: str for key in i[:1]}
for key in i[14:-1]:
        data_types_revenues[key] = float
data_types_revenues[i[-1]] = str
deleting = []
for label in i:
        if 'Solvency' in label:
                deleting.append(label)
                print(deleting)
for i in range(212):
        filename = str(i)
        df = pd.read_csv('cleaned_downloads/{}'.format(filename),
                                    dtype=data_types, na_values=['', ' ', 'n.a.', 'n.s.'],
thousands='.', decimal=',')
        with open('dataframes/{}.pickle'.format(i), 'wb') as out:
                pickle.dump(df, out)
        print(i, 'done')
for i in range(6):
        filename = str(i)
        df = pd.read_csv('cleaned_revenues/{}'.format(filename),
                                    dtype=data_types_revenues, na_values=['', ' ',
'n.a.', 'n.s.'], thousands='.', decimal=',')
        with open('{}.pickle'.format(i), 'wb') as out:
                pickle.dump(df, out)
        print(i, 'done')
# joins the original dataframes with the new revenues
```

```
revenues_df = pd.concat([pd.read_csv('cleaned_revenues/{}'.format(j),
dtype=data_types_revenues,
                                                 na_values=['', ' ', 'n.a.', 'n.s.'],
thousands='.', decimal=',')
                                  for j in range(6)], ignore_index=True,
verify_integrity=True)
revenues_df.drop(labels=['Mark', 'Company name'], axis=1, inplace=True)
```

cleaner.py

This performs all the actions described in Chapter 3, except for the deletion of rows and columns.

```
import pickle
import csv
import collections
import numpy as np
class OrderedSet(collections.MutableSet):
    def __init__(self, iterable=None):
        self.end = end = []
        end += [None, end, end]        # sentinel node for doubly linked
list
        self.map = {}                  # key --> [key, prev, next]
        if iterable is not None:
            self |= iterable
    def __len__(self):
        return len(self.map)
    def __contains__(self, key):
        return key in self.map
    def add(self, key):
        if key not in self.map:
            end = self.end
            curr = end[1]
            curr[2] = end[1] = self.map[key] = [key, curr, end]
    def discard(self, key):
        if key in self.map:
            key, prev, next = self.map.pop(key)
            prev[2] = next
            next[1] = prev
    def __iter__(self):
        end = self.end
        curr = end[2]
        while curr is not end:
            yield curr[0]
            curr = curr[2]
    def __reversed__(self):
        end = self.end
        curr = end[1]
        while curr is not end:
            yield curr[0]
```

```
                curr = curr[1]
    def pop(self, last=True):
        if not self:
            raise KeyError('set is empty')
        key = self.end[1][0] if last else self.end[2][0]
        self.discard(key)
        return key
    def __repr__(self):
        if not self:
            return '%s()' % (self.__class__.__name__,)
        return '%s(%r)' % (self.__class__.__name__, list(self))
    def __eq__(self, other):
        if isinstance(other, OrderedSet):
            return len(self) == len(other) and list(self) == list(other)
        return set(self) == set(other)
# saves a variable containing all of the column names
with open('cleaned_downloads/0', 'r') as f:
    reader = csv.reader(f)
    columns = next(reader)
with open('cleaned_revenues/0', 'r') as f:
    reader = csv.reader(f)
    columns.extend(next(reader))
dropped_labels = ['Solvency ratio (%) %  Last avail. yr', 'Solvency ratio (%)
%  Year - 1',
                  'Solvency ratio (%) %  Year - 2', 'Solvency ratio (%) %
Year - 3', 'Mark',
                  'Solvency ratio (%) %  Year - 4', 'Previous company
name', 'Company name']
columns = list(OrderedSet([x for x in columns if x not in
dropped_labels]))
labels_to_drop = dropped_labels
categories = ['Accounting closing date Last avail. yr',
'Tax code number',
'Trading address - Region',
'Legal status',
'Incorporation year',
'No of available years',
'Last accounting closing date',
'Procedure/cessazione',
'Date of open procedure/cessazione',
'NACE Rev. 2',
'Profit (loss) EUR',
'Total assets EUR',
"Total shareholder's funds EUR",
'Return on sales (ROS) %',
'Return on asset (ROA) %',
'Return on equity (ROE) %',
'Banks/turnover %',
'Liquidity ratio',
'Current ratio',
'Current liabilities/Tot ass. %',
'Long/med term liab/Tot ass. %',
'Tang. fixed ass./Share funds %',
'Depr./Tang. fixed assets %',
'Leverage',
'Coverage of fixed assets %',
'Banks/Turnover (%) %',
'Cost of debit (%) %',
'Interest/Operating profit %',
'Interest/Turnover (%) %',
'Share funds/Liabilities %',
'Net Financial Position EUR',
'Debt/Equity ratio %',
'Debt/EBITDA ratio %',
'Total assets turnover (times)',
'Incidenza circolante operativo (%) %',
'Stocks/Turnover (days)',
'Durata media dei crediti al lordo IVA (days)',
'Durata media dei debiti al lordo IVA (days)',
'Durata Ciclo Commerciale (days)',
'EBITDA EUR',
'EBITDA/Vendite (%) %',
'Return on investment (ROI) (%) %',
'Number of employees',
'Added value per employee',
'Staff Costs per employee',
'Turnover/Staff Costs',
'Net working capital EUR',
'Gross profit EUR',
'Net short term assets EUR',
'Share funds - Fixed assets EUR',
'Cash Flow EUR',
'Revenues from sales and services']
is_bankrupt_by_status = [['Dissolved (demerger)', 0], ['Bankruptcy', 1],
['Dissolved (liquidation)', 0],
                                    ['Dissolved (bankruptcy)', 1], ['Dissolved
(merger)', 0], ['In liquidation', 0],
                                    ['Dissolved', 0], ['Active (receivership)', 1],
['Active', 0],
                                    ['Active (default of payments)', 1]]
is_bankrupt_by_procedure = [['Transfer to another province', 0],
['Reasons provided for in the articles of association', 0],
['Winding up without liquidation', 1],
['Closure due to liquidation', 1],
['Closure due to bankruptcy or liquidatione', 1],
['Removal ex officio', 0],
['Winding up', 1],
['Initiation of cancellation procedure', 0],
['Composition with creditors', 1],
['Winding up and liquidation', 1],
['Voluntary liquidation', 0],
['Winding up in advance without liquidation', 1],
['Winding up and placing into liquidation', 1],
['Extraordinary administration', 1],
['Cancellation ex officio following creation of Chamber of Commerce,
Industry, Craft Trade and Agriculture for Fermo', 0],
['Court order of cancellation', 1],
['Debt restructuring agreements', 1],
['Conclusion of liquidation', 1],
['Approved by all partners', 0],
['Cancellation ex officio following creation of Chamber of Commerce, 
Industry, Craft Trade and Agriculture for Monza', 0],
['Court ordered seizure', 1],
['Cancellation due to communication of allocation plan', 1],
['Lease of company', 0],
['Demerger', 0],
['Contribution', 1],
['Cancellation ex officio of registration with register of companies', 1],
['Removed ex officio because already included in the register of firms and 
not transferred to the register of companies', 0],
['Failure to re-establish multiple partners', 1],
['Fulfilment of company object', 0],
['Transformation of legal status', 0],
['Cancelled ex officio pursuant to Italian Presidential Decree no. 247 of 23 
July 2004', 1],
['Reason not specified', 1],
['Conclusion of bankruptcy procedures', 1],
['Cancellation from the register of companies', 1],
['Cancelled ex officio pursuant to Article 2490 of the Italian Civil Code', 1],
['Court ordered liquidation', 1],
['Duplication', 0],
['Bankruptcy', 1],
['Removal ex officio following report by register of companies for the 
registered office', 1],
['Other reasons', 1],
['Closure due to bankruptcy', 1],
['Merger by incorporation into another company', 0],
['Supervening failure to meet the prerequisites for a company', 1],
['Court ordered administration', 1],
['Impossibility of fulfilment of the company object', 1],
['Precautionary seizure of shares', 1],
['Closure of local branch', 0],
['Cessation of any business', 1],
['Following expiry of time limits', 0],
['Post-bankruptcy composition with creditors', 1],
['Transformation into a registered office', 0],
['Winding up by official order', 1],
['State of insolvency', 1],
['Removal ex officio, lack of tax code (Article 21 of Italian Presidential 
Decree no. 605 of 29 September 1973, as amended)', 1],
['Transfer of firma', 0],
['Merger by incorporation of new company', 0],
['Compulsory administrative liquidation', 1],
['Cessation of business within the province', 0],
['Liquidation', 1],
['Controlled administration', 1]]
#
----------------------------------------------------------------------------------------------------
--
```

```
#
----------------------------------------------------------------------------------------------------
--
# views the dataframe
with open('dataframes/{}.pickle'.format(91), 'rb') as inp:
    df = pickle.load(inp)
    print(df.iloc[0])
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
# drops rows where "Mark", "Nace" or "Incorporation" is NaN
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        # df.drop(df[pd.isnull(df.Mark)].index, inplace=True)
        df.drop(df[pd.isnull(df['NACE Rev. 2'])].index, inplace=True)
        df.drop(df[pd.isnull(df['Incorporation year'])].index, inplace=True)
    with open('dataframes/{}.pickle'.format(i), 'wb') as inp:
        pickle.dump(df, inp)
    print(i, 'done')
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
# deletes columns marked in the variable labels_to_drop
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        df.drop(labels=labels_to_drop, axis=1, inplace=True)
    with open('dataframes/{}.pickle'.format(i), 'wb') as inp:
        pickle.dump(df, inp)
    print(i, 'done')
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
# counts the number of firms in the dataframe
number_of_firms = 0
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        for firm in df['Tax code number']:
            number_of_firms += 1
    print('counting firms:', i, 'done')
print(number_of_firms)
```

```
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
# counts the number of NaN in every column of the database
missing_by_column = {key: 0 for key in columns}
for i in range(212):
      with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
            df = pickle.load(inp)
            for key in columns:
                  for value in df[key]:
                        if pd.isnull(value):
                              missing_by_column[key] += 1
      print('counting NaNs:', i, 'done')
missing_list = []
for key in missing_by_column:
      percentage = round((missing_by_column[key] /
number_of_firms)*100)
      missing_list.append([key, missing_by_column[key], percentage])
missing_list = sorted(missing_list, key=lambda x: x[1], reverse=True)
with open('missing_list.pickle', 'wb') as f:
      pickle.dump(missing_list, f)
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
# finds the categories in which for every year the percentage of NaN is
less than fallback_nan, or if in at
# most one year it is less than fallback_nan but higher than maximum_nan
def keep_category(category_to_check):
      maximum_nan = 20
      fallback_nan = 40
      years_to_check = category_to_check[1]
      keep = True
      dangerous_years = 0
      for year in years_to_check:
            if year[2] > fallback_nan:
                  keep = False
            elif year[2] > maximum_nan:
                  dangerous_years += 1
      if dangerous_years > 1:
            keep = False
      if 'rocedure/cessazione' in category_to_check[0]:
            keep = True
      return keep
nan_count_by_categories = [(x, []) for x in categories]
with open('missing_list.pickle', 'rb') as f:
      nan_count_by_columns = pickle.load(f)
for i in nan_count_by_columns:
    column = i[0]
    for j in nan_count_by_categories:
        category = j[0]
        if category in column:
            j[1].append(i)
# for x in missing_list_by_categories:
#     print(x)
print(len(nan_count_by_categories))
nan_count_by_categories_reduced = [x for x in nan_count_by_categories
if keep_category(x)]
print(len(nan_count_by_categories_reduced))
excluded = [x for x in nan_count_by_categories if x not in
nan_count_by_categories_reduced]
categories_to_exclude = [x[0] for x in excluded]
remaining_categories = [x[0] for x in nan_count_by_categories_reduced]
with open('categories_to_exclude.pickle', 'wb') as out:
    pickle.dump(categories_to_exclude, out)
# for x in excluded:
#     print(x)
for x in nan_count_by_categories_reduced:
    print(x)
print(remaining_categories)
#
-------------------------------------------------------------------------------------------------
--
#
-------------------------------------------------------------------------------------------------
--
# finds rows with more than 2 missing data per category in more than 5
categories and rows
# with more than 1 empty category
def delete_row(row):
    delete = False
    dangerous_categories = 0
    empty_categories = 0
    dangerous_categories_limit = 5
    empty_categories_limit = 1
    for category in remaining_categories:
        columns_to_check = [x for x in columns if category in x]
        nans = sum([1 for x in columns_to_check if pd.isnull(row[x])])
        if nans > 2:
            dangerous_categories += 1
        if nans > 4:
            empty_categories += 1
    if dangerous_categories > dangerous_categories_limit:
        delete = True
    if empty_categories > empty_categories_limit:
        delete = True
    return delete
rows_to_delete = []
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        for index, row in df.iterrows():
            if delete_row(row) is True:
                rows_to_delete.append(row['Tax code number'])
    print('calculating rows to delete: {} done'.format(i))
with open('rows_to_delete.pickle', 'wb') as out:
    pickle.dump(rows_to_delete, out)
with open('rows_to_delete.pickle', 'rb') as inp:
    rows_to_delete = pickle.load(inp)
print('number of rows to delete:', len(rows_to_delete))
#
--------------------------------------------------------------------------------------------------
--
#
--------------------------------------------------------------------------------------------------
--
# changes NACE to keep only the first 2 letters
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        df['NACE_first_2'] = df['NACE Rev. 2'].map(lambda x: str(x)[:2])
    with open('dataframes/{}.pickle'.format(i), 'wb') as out:
        pickle.dump(df, out)
    print(i, 'done')
#
--------------------------------------------------------------------------------------------------
--
#
--------------------------------------------------------------------------------------------------
--
# converts all dates to years
def convert_to_year(date):
    date_as_string = str(date)
    year_as_string = date_as_string[-4:]
    return year_as_string
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        for column in ['Accounting closing date Last avail. yr',
'Incorporation year',
                       'Last accounting closing date', 'Date of open
procedure/cessazione']:
            df[column] = df[column].apply(convert_to_year)
    with open('dataframes/{}.pickle'.format(i), 'wb') as out:
        pickle.dump(df, out)
    print('converting dates {} done'.format(i))
#
--------------------------------------------------------------------------------------------------
--
#
--------------------------------------------------------------------------------------------------
--
# counts the number of NACE codes
naces = []
```

```
for i in range(212):
      with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
            df = pickle.load(inp)
            new_naces = set(list(df['NACE_first_2']))
            naces = list(naces)
            naces.extend(new_naces)
            naces = set(naces)
      print(i)
print()
print(len(list(naces)))
print(list(naces)[:20])
#
---------------------------------------------------------------------------------------------------
--
#
---------------------------------------------------------------------------------------------------
--
# finds the possible values of 'Legal status' and 'Procedure/cessazione'
status = set([])
for i in range(212):
      with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
            df = pickle.load(inp)
            new_status = set(list(df['Legal status']))
            status.update(new_status)
      print(i)
status_specific = set([])
for i in range(212):
      with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
            df = pickle.load(inp)
            new_status = set(list(df['Procedure/cessazione'].dropna(axis=0,
how='all')))
            status_specific.update(new_status)
      print(i)
with open('procedures_cessazioni.pickle', 'wb') as inp:
      pickle.dump(status_specific, inp)
with open('procedures_cessazioni.pickle', 'rb') as inp:
      status_specific = pickle.load(inp)
print()
print(len(list(status)))
print(list(status))
print()
print(len(list(status_specific)))
print(list(status_specific))
#
---------------------------------------------------------------------------------------------------
--
#
---------------------------------------------------------------------------------------------------
--
# assigns to each firm either 0 (not failed) or 1 (failed) in the column
['failed']
failures_by_category = {status[0]: 0 for status in is_bankrupt_by_status}
def is_failed(row):
      failed_ = 0
```

```
    firm_status = row['Legal status']
    firm_procedure = row['Procedure/cessazione']
    for status in is_bankrupt_by_status:
        if status[0] == firm_status:
            failed_ = status[1]
    # checks if the variable 'Legal status' is already enough to say that
the firm is failed, if not checks the
    # value of row['Procedure/cessazione']
    if failed_ != 1:
        # if there is no procedure then the firm did not fail, otherwise
checks which kind of procedure
        # the firm underwent
        if not pd.isnull(row['Procedure/cessazione']):
            for procedure in is_bankrupt_by_procedure:
                if procedure[0] == firm_procedure:
                    failed_ = procedure[1]
    failures_by_category[firm_status] += failed_
    return failed_
number_of_failures = 0
number_of_failures_deleted_rows = 0
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        df['failed'] = 0
        for index, row in df.iterrows():
            failed = is_failed(row)
            if row['Tax code number'] in rows_to_delete:
                number_of_failures_deleted_rows += failed
            df.at[index, 'failed'] = failed
    with open('dataframes/{}.pickle'.format(i), 'wb') as out:
        pickle.dump(df, out)
    print(i, 'done')
    number_of_failures += sum(list(df['failed']))
print()
print(number_of_failures)
print(number_of_failures_deleted_rows)
print(failures_by_category)
#
-----------------------------------------------------------------------------------------------------------
--
#
-----------------------------------------------------------------------------------------------------------
--
# extracts the necessary information to graph the data (both for the
complete dataset and the reduced one)
numerical_categories = [category for category in categories[10:]]
non_numerical_categories = ['Accounting closing date Last avail. yr',
'Trading address - Region',
                                            'Legal status', 'Incorporation year',
'NACE_first_2']
columns.append('NACE_first_2')
for graphing_category in numerical_categories:
    # _c refers to the complete dataset, while _d to the dataset with the
deleted rows
    values_c = []
    values_d = []
    for i in range(212):
        with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
            df = pickle.load(inp)
            df_d = df[df['Tax code number'].isin(rows_to_delete)]
            for column in columns:
                if graphing_category in column:
                    values_c.extend(list(df[column].dropna(axis=0,
how='all')))
                    values_d.extend(list(df_d[column].dropna(axis=0,
how='all')))
        print('extracting graph of {}:'.format(graphing_category), i,
'done')
    values_c = np.array(values_c)
    values_d = np.array(values_d)
    mean_c = np.mean(values_c)
    mean_d = np.mean(values_d)
    median_c = np.median(values_c)
    median_d = np.median(values_d)
    std_c = np.std(values_c)
    std_d = np.std(values_d)
    max_c = np.max(values_c)
    max_d = np.max(values_d)
    min_c = np.min(values_c)
    min_d = np.min(values_d)
    graph_c = {'mean': mean_c,
               'median': median_c,
               'min': min_c,
               'max': max_c,
               'std': std_c,
               'values': values_c}
    graph_d = {'mean': mean_d,
               'median': median_d,
               'min': min_d,
               'max': max_d,
               'std': std_d,
               'values': values_d}
    with open('columns_graph/
{}_c.pickle'.format(graphing_category.replace('/', '-')), 'wb') as out:
        pickle.dump(graph_c, out)
    with open('columns_graph/
{}_d.pickle'.format(graphing_category.replace('/', '-')), 'wb') as out:
        pickle.dump(graph_d, out)
    print()
    print('graph_c of {}:'.format(graphing_category))
    for key in graph_c:
        if key != 'values':
            print('{0}: {1}'.format(key, graph_c[key]))
    print()
    print('graph_d of {}:'.format(graphing_category))
    for key in graph_d:
        if key != 'values':
            print('{0}: {1}'.format(key, graph_d[key]))
    print()
    print()
for graphing_category in non_numerical_categories:
    if 'Accounting' in graphing_category:
        # _c refers to the complete dataset, while _d to the dataset with
the deleted rows
        values_c = []
        values_d = []
        for i in range(212):
            with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
                df = pickle.load(inp)
                df_d = df[df['Tax code number'].isin(rows_to_delete)]
                for column in columns:
                    if graphing_category in column:
                        values_c.extend(list(df[column].dropna(axis=0,
how='all')))

values_d.extend(list(df_d[column].dropna(axis=0, how='all')))
            print('extracting graph of {}:'.format(graphing_category), i,
'done')
        values_c_set = list(set(values_c))
        values_d_set = list(set(values_d))
        graph_c = {value: values_c.count(value) for value in
values_c_set}
        graph_d = {value: values_d.count(value) for value in
values_d_set}
        with open('columns_graph/
{}_c.pickle'.format(graphing_category), 'wb') as out:
            pickle.dump(graph_c, out)
        with open('columns_graph/
{}_d.pickle'.format(graphing_category), 'wb') as out:
            pickle.dump(graph_d, out)
        # print()
        # print('graph_c of {}:'.format(graphing_category))
        # print(graph_c)
        # print('graph_d of {}:'.format(graphing_category))
        # print(graph_d)
        # print()
        years = []
        for key in graph_c:
            years.append(key)
        years = sorted(years)
        print(years)
#
----------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------
--
```

```
# finds rows marked as failed whose date of open procedure is equal or
minor than the date
# of last account available
with open('rows_to_delete.pickle', 'rb') as inp:
	rows_to_delete = pickle.load(inp)
rows_to_delete_by_closing_date = []
def delete_row(row):
	delete = False
	if row['failed'] == 1:
		if pd.notnull(row['Date of open procedure/cessazione']):
			if row['Date of open procedure/cessazione'] < row['Last
accounting closing date']:
				if row['Tax code number'] not in rows_to_delete:
					delete = True
	return delete
for i in range(212):
	with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
		df = pickle.load(inp)
		for index, row in df.iterrows():
			if delete_row(row) is True:
				rows_to_delete_by_closing_date.append(row['Tax code
number'])
	print('calculating rows to delete: {} done'.format(i))
print('number of rows to delete based on the closing date:
{}'.format(len(rows_to_delete_by_closing_date)))
with open('rows_to_delete_by_closing_date', 'wb') as f:
	pickle.dump(rows_to_delete_by_closing_date, f)
print()
print('all done')
```

numpify_dataset.py

This code deletes the rows and columns as described in Chapter 3, and then converts the Pandas dataframe to a Numpy array, which is the data format used as input by the neural networks.

```
import numpy as np
import collections
import pickle
import csv
class OrderedSet(collections.MutableSet):
	def __init__(self, iterable=None):
		self.end = end = []
		end += [None, end, end]		# sentinel node for doubly linked
list
		self.map = {}			# key --> [key, prev, next]
		if iterable is not None:
			self |= iterable
	def __len__(self):
        return len(self.map)
    def __contains__(self, key):
        return key in self.map
    def add(self, key):
        if key not in self.map:
            end = self.end
            curr = end[1]
            curr[2] = end[1] = self.map[key] = [key, curr, end]
    def discard(self, key):
        if key in self.map:
            key, prev, next = self.map.pop(key)
            prev[2] = next
            next[1] = prev
    def __iter__(self):
        end = self.end
        curr = end[2]
        while curr is not end:
            yield curr[0]
            curr = curr[2]
    def __reversed__(self):
        end = self.end
        curr = end[1]
        while curr is not end:
            yield curr[0]
            curr = curr[1]
    def pop(self, last=True):
        if not self:
            raise KeyError('set is empty')
        key = self.end[1][0] if last else self.end[2][0]
        self.discard(key)
        return key
    def __repr__(self):
        if not self:
            return '%s()' % (self.__class__.__name__,)
        return '%s(%r)' % (self.__class__.__name__, list(self))
    def __eq__(self, other):
        if isinstance(other, OrderedSet):
            return len(self) == len(other) and list(self) == list(other)
        return set(self) == set(other)
# saves a variable containing all of the column names
with open('cleaned_downloads/0', 'r') as f:
    reader = csv.reader(f)
    columns = next(reader)
manual_labels_to_drop = ['Accounting closing date Last avail. yr', 'Tax
code number', 'Trading address - Region',
                         'Legal status', 'Incorporation year', 'No of
available years',
                         'Last accounting closing date',
'Procedure/cessazione',
                         'Date of open procedure/cessazione', 'NACE
Rev. 2']
columns = list(OrderedSet([x for x in columns if x not in
manual_labels_to_drop]))
```

```
columns.append('NACE_first_2')
columns.append('failed')
categories = ['Accounting closing date Last avail. yr',
'Tax code number',
'Trading address - Region',
'Legal status',
'Incorporation year',
'No of available years',
'Last accounting closing date',
'Procedure/cessazione',
'Date of open procedure/cessazione',
'NACE Rev. 2',
'Profit (loss) EUR',
'Total assets EUR',
"Total shareholder's funds EUR",
'Return on sales (ROS) %',
'Return on asset (ROA) %',
'Return on equity (ROE) %',
'Banks/turnover %',
'Liquidity ratio',
'Current ratio',
'Current liabilities/Tot ass. %',
'Long/med term liab/Tot ass. %',
'Tang. fixed ass./Share funds %',
'Depr./Tang. fixed assets %',
'Leverage',
'Coverage of fixed assets %',
'Banks/Turnover (%) %',
'Cost of debit (%) %',
'Interest/Operating profit %',
'Interest/Turnover (%) %',
'Share funds/Liabilities %',
'Net Financial Position EUR',
'Debt/Equity ratio %',
'Debt/EBITDA ratio %',
'Total assets turnover (times)',
'Incidenza circolante operativo (%) %',
'Stocks/Turnover (days)',
'Durata media dei crediti al lordo IVA (days)',
'Durata media dei debiti al lordo IVA (days)',
'Durata Ciclo Commerciale (days)',
'EBITDA EUR',
'EBITDA/Vendite (%) %',
'Return on investment (ROI) (%) %',
'Number of employees',
'Added value per employee',
'Staff Costs per employee',
'Turnover/Staff Costs',
'Net working capital EUR',
'Gross profit EUR',
'Net short term assets EUR',
'Share funds - Fixed assets EUR',
'Cash Flow EUR',
'Revenues from sales and services']
```

```
# copy data from dataframes/
for i in range(212):
    with open('dataframes/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'wb') as out:
        pickle.dump(df, out)
    print('copying: {} done'.format(i))
# views the dataframe
with open('dataframes_to_numpy/{}.pickle'.format(91), 'rb') as inp:
    df = pickle.load(inp)
    print(df.iloc[0])
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
with open('rows_to_delete.pickle', 'rb') as inp:
    rows_to_delete = pickle.load(inp)
print('number of rows to delete:', len(rows_to_delete))
with open('rows_to_delete_by_closing_date', 'rb') as f:
    rows_to_delete_by_closing_date = pickle.load(f)
# deletes rows
for i in range(212):
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        df = df[~df['Tax code number'].isin(rows_to_delete)]
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'wb') as inp:
        pickle.dump(df, inp)
    print('deleting rows: {} done'.format(i))
# counts the number of firms in the dataframe
number_of_firms = 0
for i in range(212):
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        for firm in df['Tax code number']:
            number_of_firms += 1
    print('counting firms:', i, 'done')
print(number_of_firms)
#
----------------------------------------------------------------------------------------------------
--
#
----------------------------------------------------------------------------------------------------
--
# calculates columns to drop
with open('categories_to_exclude.pickle', 'rb') as out:
    categories_to_exclude = pickle.load(out)
print(categories_to_exclude)
labels_to_drop = manual_labels_to_drop
for column in columns:
    for category in categories_to_exclude:
        if category in column:
            labels_to_drop.append(column)
labels_to_drop = list(set(labels_to_drop))
# deletes columns marked in the variable labels_to_drop
for i in range(212):
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        df.drop(labels=labels_to_drop, axis=1, inplace=True)
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'wb') as inp:
        pickle.dump(df, inp)
    print('dropping columns: {} done'.format(i))
#
-----------------------------------------------------------------------------------------------
--
#
-----------------------------------------------------------------------------------------------
--
# divides NACE by 100
for i in range(212):
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'rb') as inp:
        df = pickle.load(inp)
        df['NACE_first_2'] = df['NACE_first_2'].apply(lambda x:
int(x)/100)
    with open('dataframes_to_numpy/{}.pickle'.format(i), 'wb') as out:
        pickle.dump(df, out)
    print('converting NACE: {} done'.format(i))
# views the dataframe
with open('dataframes_to_numpy/{}.pickle'.format(89), 'rb') as inp:
    df = pickle.load(inp)
    print(df.iloc[0])
#
-----------------------------------------------------------------------------------------------
--
#
-----------------------------------------------------------------------------------------------
--
# gets column names
with open('dataframes_to_numpy/{}.pickle'.format(89), 'rb') as inp:
    df = pickle.load(inp)
    columns = list(df.columns.values)
# for column in columns:
#     print(column)
# groups column names by year
year0_names = ['Year - 1', 'Year - 2', 'Year - 3', 'Year - 4']
year1_names = ['Last avail. yr', 'Year - 2', 'Year - 3', 'Year - 4']
year2_names = ['Last avail. yr', 'Year - 1', 'Year - 3', 'Year - 4']
year3_names = ['Last avail. yr', 'Year - 1', 'Year - 2', 'Year - 4']
year4_names = ['Last avail. yr', 'Year - 1', 'Year - 2', 'Year - 3']
year_names = [year0_names, year1_names, year2_names, year3_names,
year4_names]
year_columns = [[] for _ in range(5)]
for i in range(5):
    for column_name in columns:
        keep = True
        for name in year_names[i]:
```

```
                        if name in column_name:
                                keep = False
                if keep is True:
                        year_columns[i].append(column_name)
# for i in range(5):
#     print(year_columns[i])
#     print(len(year_columns[i]))
# groups column names by category
removing_words = ['Last avail. yr', 'Year - 1', 'Year - 2', 'Year - 3', 'Year - 4']
categories = []
for column in columns:
        category = column
        for word in removing_words:
                category = category.replace(' ' + word, '')
        categories.append(category)
categories = list(OrderedSet(categories))
print()
print(categories)
print(len(categories))
#
-------------------------------------------------------------------------------------------------------
--
#
-------------------------------------------------------------------------------------------------------
--
# gives all the columns in a category
def columns_of_category(cat):
        removing_words = ['Last avail. yr', 'Year - 1', 'Year - 2', 'Year - 3', 'Year
- 4']
        result = [cat + ' ' + word for word in removing_words]
        return result
# normalizes the values of the dataframe
for i in range(212):
        for category in categories:
                if category not in ['NACE_first_2', 'failed']:
                        converting_category = columns_of_category(category)
                        with open('dataframes_to_numpy/{}.pickle'.format(i), 'rb')
as inp:
                                df = pickle.load(inp)
                                df_section = df[converting_category]
                        a = np.array(df_section)
                        column_lenght = len(a)
                        # print(column_lenght)
                        # print(a[18])
                        a = np.ma.array(a, mask=np.isnan(a))  # Use a mask to
mark the NaNs
                        a_norm = a - np.mean(a)  # The sum function ignores the
masked values.
                        a_norm2 = a_norm / np.std(a)  # The std function ignores the
masked values.
                        dtype = [(column, 'float64') for column in
converting_category]
                        values = a_norm2
                index = ['Row' + str(i) for i in range(1, len(values) + 1)]
                # print(df.iloc[0])
                df_section = pd.DataFrame(values, index=index,
columns=converting_category, dtype='float64')
                df[columns_of_category(category)] =
df_section[columns_of_category(category)].values
                # print(df.iloc[0])
                with open('dataframes_to_numpy/{}.pickle'.format(i), 'wb')
as out:
                    pickle.dump(df, out)
    print('normalizing values: {} done'.format(i))
#
--------------------------------------------------------------------------------------------------
--
#
--------------------------------------------------------------------------------------------------
--
# gives all the columns for one year
def columns_of_year(year):
    year_suffix = ['Last avail. yr', 'Year - 1', 'Year - 2', 'Year - 3', 'Year - 4']
    yearly_columns = [category + ' ' + year_suffix[year] for category in
categories
                      if category not in ['NACE_first_2', 'failed']]
    result = ['failed', 'NACE_first_2']
    result.extend(yearly_columns)
    return result
# saves the database in a giant numpy array
firms = []
for file_num in range(1):
    with open('dataframes_to_numpy/{}.pickle'.format(file_num), 'rb') as
inp:
        df = pickle.load(inp)
        for index, row in df.iterrows():
            this_firm = []
            for year in range(5):
                this_year = []
                this_year_dummies = [0 for _ in range(27)]
                for i, cell_value in
enumerate(row[columns_of_year(year)]):
                    if pd.isnull(cell_value):
                        this_year_dummies[i - 2] = 1
                        this_year.append(0)
                    else:
                        this_year.append(cell_value)
                this_year.extend(this_year_dummies)
                this_year = np.array(this_year, dtype=np.float32)
                this_firm.append(this_year)
            this_firm = np.array(this_firm)
            firms.append(this_firm)
    print('numpyfying database: {} done'.format(file_num))
numpy_database = np.array(firms)
firms = []
for file_num in range(1, 212):
    with open('dataframes_to_numpy/{}.pickle'.format(file_num), 'rb') as
inp:
        df = pickle.load(inp)
        for index, row in df.iterrows():
            this_firm = []
            for year in range(5):
                this_year = []
                this_year_dummies = [0 for _ in range(27)]
                for i, cell_value in
enumerate(row[columns_of_year(year)]):
                    if pd.isnull(cell_value):
                        this_year_dummies[i - 2] = 1
                        this_year.append(0)
                    else:
                        this_year.append(cell_value)
                this_year.extend(this_year_dummies)
                this_year = np.array(this_year, dtype=np.float32)
                this_firm.append(this_year)
            this_firm = np.array(this_firm)
            firms.append(this_firm)
    print('numpyfying database: {} done'.format(file_num))
firms = np.array(firms)
numpy_database = np.append(numpy_database, firms, axis=0)
print('length of the numpy array: {}'.format(len(numpy_database)))
with open('original_numpyfied_database.pickle', 'wb') as out:
    pickle.dump(numpy_database, out)
#
-------------------------------------------------------------------------------------------------
--
#
-------------------------------------------------------------------------------------------------
--
with open('numpyfied_database.pickle', 'rb') as inp:
    numpy_database = pickle.load(inp)
print()
print('database loaded!!!')
print()
with open('numpyfied_database.pickle', 'wb') as out:
    pickle.dump(numpy_database, out)
print()
print('database shape:', numpy_database.shape)
print(numpy_database[0])
database_transposed = np.transpose(numpy_database, (1, 0, 2))
numpyfied_database_year_0 = database_transposed[0]
with open('numpyfied_database_year_0.pickle', 'wb') as out:
    pickle.dump(numpyfied_database_year_0, out)
```

Appendix 5

ANNs code

This section will present the source code of the programs used to run the experiments described in Chapter 4.

NetworkTraining.py

This is the code use for the training phase of the MLP model.

```python
from time import localtime, strftime
import tensorflow as tf
import numpy as np
import os
dataset = 'numpyfied_database_year_0.pickle'
print('loading dataset...')
with open(dataset, 'rb') as f:
      data = pickle.load(f)
# here I am using a label like: [1, 0] is 'non failed' and [0, 1] is 'failed', so the output
# of the network will consist in two neurons, one for each class. But does it make sense?
# wouldn't it be better to use just one neuron and if it is > 0.5 the firm failed?
def calc_label(firm):
      if int(firm[0]) == 0:
            label = [1, 0]
      else:
            label = [0, 1]
      return label
print('composing fetatures and labels...')
features_and_labels = [[x[1:], calc_label(x)] for x in data]
print('shuffling data...')
np.random.shuffle(features_and_labels)
print('dividing features from labels...')
features = np.array([x[0] for x in features_and_labels])
labels = np.array([x[1] for x in features_and_labels])
del data
del features_and_labels
tf_log = 'tf.log'
try:
      epoch = int(open(tf_log, 'r').read().split('\n')[-2]) + 1
      print('Starting epoch:', epoch)
except:
      epoch = 1
if epoch == 1:
      print('dividing train, test and validation sets...')
      train_size = int(len(labels) * 0.8)
      validate_size = int(len(labels) * 0.9)
      train_y = np.array(labels[:train_size])
      train_x = np.array(features[:train_size])
      test_y = np.array(labels[train_size:validate_size])
      test_x = np.array(features[train_size:validate_size])
      validation_y = np.array(labels[validate_size:])
      validation_x = np.array(features[validate_size:])
      test_set = list(zip(test_x, test_y))
      with open('test_set.pickle', 'wb') as f:
            pickle.dump(test_set, f)
      validation_set = list(zip(validation_x, validation_y))
      with open('validation_set.pickle', 'wb') as f:
            pickle.dump(validation_set, f)
      train_set = [train_x, train_y]
      with open('train_set.pickle', 'wb') as f:
            pickle.dump(train_set, f)
try:
      classes_n = len(labels[0])
except TypeError:
      classes_n = 1
nodes_per_layer = [100, 100]
hidden_layers_n = len(nodes_per_layer)
batch_size = 5000  # with 4GB of RAM don't go higher than 10000
epochs = 2000
print_step = 5
saving_step = 5
learn_r = 0.00001
network_structure = [classes_n, nodes_per_layer, hidden_layers_n,
len(features[0])]
with open('network_structure.pickle', 'wb') as f:
      pickle.dump(network_structure, f)
x = tf.placeholder('float', [None, len(features[0])])
y = tf.placeholder('float', [None, classes_n])
failures = tf.placeholder('float')
keep_prob = tf.placeholder(tf.float32)
current_epoch = tf.Variable(1)
layers = [{'weights': tf.Variable(tf.random_normal([len(features[0]),
nodes_per_layer[0]])),
                'biases': tf.Variable(tf.random_normal([nodes_per_layer[0]]))}]
for i in range(1, hidden_layers_n):
      layers.append({'weights':
tf.Variable(tf.random_normal([nodes_per_layer[i - 1], nodes_per_layer[i]])),
                        'biases':
tf.Variable(tf.random_normal([nodes_per_layer[i]]))})
output_layer = {'weights': tf.Variable(tf.random_normal([nodes_per_layer[-
1], classes_n])),
                    'biases': tf.Variable(tf.random_normal([classes_n]))}
def neural_network_model(data):
      l = []
      l.append(tf.add(tf.matmul(x, layers[0]['weights']), layers[0]['biases']))
      l[0] = tf.nn.relu(l[0])
      l[0] = tf.nn.dropout(l[0], keep_prob)
      for i in range(1, hidden_layers_n):
            l.append(tf.add(tf.matmul(l[i - 1], layers[i]['weights']), layers[i]
['biases']))
            l[i] = tf.nn.relu(l[i])
            l[i] = tf.nn.dropout(l[i], keep_prob)
      output = tf.add(tf.matmul(l[hidden_layers_n - 1],
output_layer['weights']), output_layer['biases'])
      return output
saver = tf.train.Saver()
# normal cost function
def train_neural_network(x, learn_rate, keep_probability):
      global train_x
      global train_y
      global test_x
      global test_y
      global train_set
      global test_set
      global validation_x
      global validation_y
      learning_rate = learn_rate
      keep = keep_probability
      prediction = neural_network_model(x)
      # this is the cost function that can be used when we know the label of
the data, so when we already knok the
      # rating class of the firms
      cost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction,
labels=y))
      # # based on (Likas 2000) a log loss is better to predict probabilities
when I have binary labels
      # cost = tf.losses.log_loss(predictions=prediction, labels=y,
epsilon=1e-8)
      # 0.0001 is usually a good value for the learning rate
      optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
epsilon=1e-8).minimize(cost)
      with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            try:
                  epoch = int(open(tf_log, 'r').read().split('\n')[-2]) + 1
                  print('Starting epoch:', epoch)
            except:
                  epoch = 1
            if epoch != 1:
                  with open('datetime.pickle', 'rb') as f:
                        folder_name = pickle.load(f)
                  with open(os.path.join(folder_name, 'graph.pickle'), 'rb') as
f:
                        graph = pickle.load(f)
                  with open('train_set.pickle', 'rb') as f:
                        train_set = pickle.load(f)
```

```
                    train_x = np.array(train_set[0])
                    train_y = np.array(train_set[1])
                    print('training set loaded')
               with open('test_set.pickle', 'rb') as f:
                    test_set = pickle.load(f)
                    test_x = np.array([x[0] for x in test_set])
                    test_y = np.array([x[1] for x in test_set])
                    print('test set loaded')
               saver.restore(sess, "model.ckpt")
          else:
               folder_name = strftime("%d-%m-%Y_%H:%M:%S",
localtime())
               graph = []
               if not os.path.exists(folder_name):
                    os.makedirs(folder_name)
               with open('datetime.pickle', 'wb') as f:
                    pickle.dump(folder_name, f)
               with open(os.path.join(folder_name, 'datetime.pickle'), 'wb')
as f:
                    pickle.dump(folder_name, f)
               with open(os.path.join(folder_name, 'test_set.pickle'), 'wb')
as f:
                    pickle.dump(test_set, f)
               with open(os.path.join(folder_name, 'validation_set.pickle'),
'wb') as f:
                    pickle.dump(validation_set, f)
               with open(os.path.join(folder_name, 'train_set.pickle'), 'wb')
as f:
                    pickle.dump(train_set, f)
               with open(os.path.join(folder_name,
'network_structure.pickle'), 'wb') as f:
                    pickle.dump(network_structure, f)
          print('Starting training...')
          while epoch <= epochs:
               epoch_loss = 1
               i = 0
               while i < len(train_x):
                    start = i
                    end = i + batch_size
                    batch_x = np.array(train_x[start:end])
                    batch_y = np.array(train_y[start:end])
                    _, c = sess.run([optimizer, cost], feed_dict={x: batch_x,
y: batch_y, keep_prob: keep})
                    epoch_loss += c
                    i += batch_size
               if (epoch + 1) % print_step == 0:
                    print('Epoch', epoch, 'out of',
                           '{} completed,'.format(epochs), 'loss:',
epoch_loss)
                    correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y,
1))
                    accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
```

```
                accuracy_number = accuracy.eval({x: test_x, y: test_y,
keep_prob: 1})
                accuracy_number_training_set = accuracy.eval({x:
train_x, y: train_y, keep_prob: 1})
                accuracy_number_validate_set = accuracy.eval({x:
validation_x, y: validation_y, keep_prob: 1})
                print('Train accuracy:', accuracy_number_training_set)
                print('Test accuracy:', accuracy_number)
                graph.append([epoch, accuracy_number_training_set,
accuracy_number, epoch_loss, accuracy_number_validate_set])
            with open(os.path.join(folder_name, 'graph.pickle'), 'wb') as
f:
                pickle.dump(graph, f)
            # 'folder' is the folder in which the program is running,
folder_name is the additional
            # folder in which the log and checkpoint files are saved
            if epoch == 1:
                folder = os.path.dirname(os.path.realpath(__file__))
                saver.save(sess, folder + "/model.ckpt")
            if (epoch + 1) % saving_step == 0:
                folder = os.path.dirname(os.path.realpath(__file__))
                epoch_folder = 'epoch_{}'.format(epoch)
                epoch_folder_name = os.path.join(folder_name,
epoch_folder)
                saver.save(sess, folder + "/model.ckpt")
                saver.save(sess, epoch_folder_name + "/model.ckpt")
                # print('Epoch', epoch, 'completed out of', epochs,
'loss:', epoch_loss)
                with open(tf_log, 'a') as f:
                    f.write(str(epoch) + '\n')
                with open(os.path.join(epoch_folder_name, tf_log), 'a')
as f:
                    f.write(str(epoch) + '\n')
            epoch += 1
if __name__ == '__main__':
    train_neural_network(x, learn_r, 0.5)
    tf.reset_default_graph()
```

<u>Network Use 0.2.py</u>

Code used for the testing of the MLP networks.

```
import numpy as np
import multiprocessing as multip
import pickle
import time
dataset = 'numpyfied_database_year_0.pickle'
with open(dataset, 'rb') as f:
    data = pickle.load(f)
def convert_label(data):
    ranks = ['not failed', 'failed']
```

```
      for i, value in enumerate(data):
            if value == 1:
                  return ranks[i]
with open('test_set.pickle', 'rb') as f:
      test_set = pickle.load(f)
test_x = [x[0] for x in test_set]
test_y = [convert_label(x[1]) for x in test_set]
with open('validation_set.pickle', 'rb') as f:
      validation_set = pickle.load(f)
validation_x = [x[0] for x in validation_set]
validation_y = [convert_label(x[1]) for x in validation_set]
with open('network_structure.pickle', 'rb') as f:
      network_structure = pickle.load(f)
classes_n = network_structure[0]
nodes_per_layer = network_structure[1]
hidden_layers_n = network_structure[2]
features_len = network_structure[3]
x = tf.placeholder('float', [None, features_len])
y = tf.placeholder('float', [None, classes_n])
current_epoch = tf.Variable(1)
layers = [{'weights': tf.Variable(tf.random_normal([features_len,
nodes_per_layer[0]])),
                'biases': tf.Variable(tf.random_normal([nodes_per_layer[0]]))}]
for i in range(1, hidden_layers_n):
      layers.append({'weights':
tf.Variable(tf.random_normal([nodes_per_layer[i - 1], nodes_per_layer[i]])),
                              'biases':
tf.Variable(tf.random_normal([nodes_per_layer[i]]))})
output_layer = {'weights': tf.Variable(tf.random_normal([nodes_per_layer[-
1], classes_n])),
                         'biases': tf.Variable(tf.random_normal([classes_n]))}
def neural_network_model(data):
      l = []
      l.append(tf.add(tf.matmul(x, layers[0]['weights']), layers[0]['biases']))
      l[0] = tf.nn.relu(l[0])
      for i in range(1, hidden_layers_n):
            l.append(tf.add(tf.matmul(l[i - 1], layers[i]['weights']), layers[i]
['biases']))
            l[i] = tf.nn.relu(l[i])
      output = tf.add(tf.matmul(l[hidden_layers_n - 1],
output_layer['weights']), output_layer['biases'])
      return output
saver = tf.train.Saver()
tf_log = 'tf.log'
def convert_prediction(value):
      predict = ''
      if value == 1:
            predict = 'failed'
      elif value == 0:
            predict = 'not failed'
      return predict
def use_neural_network(test_or_validation):
      if test_or_validation == 'test':
        set_x = test_x
        set_y = test_y
    elif test_or_validation == 'validation':
        set_x = validation_x
        set_y = validation_y
    prediction = neural_network_model(x)
    with tf.Session() as sess:
        for word in ['weights', 'biases']:
            output_layer[word].initializer.run()
            for variable in layers:
                variable[word].initializer.run()
        saver.restore(sess, "model.ckpt")
        predictions = sess.run(tf.argmax(prediction.eval(feed_dict={x:
set_x}), 1))
        predictions = np.array([convert_prediction(value) for value in
predictions])
        result = list(zip(set_y, predictions))
        return result
def is_correct(x):
    if x[0] == x[1]:
        return 1
    else:
        return 0
def test_set_errors():
    prediction = use_neural_network('test')
    print('\nCalculating errors in test set...')
    predictions_dict_type1 = {'not failed': [],
                              'failed': []}
    predictions_dict_type2 = {'not failed': [],
                              'failed': []}
    for elem in prediction:
        predictions_dict_type1[elem[0]].append(elem)
    for elem in prediction:
        predictions_dict_type2[elem[1]].append(elem)
    correct_guesses = sum(is_correct(x) for x in prediction)
    correct_ratio = correct_guesses / len(prediction)
    print('correct:', correct_ratio)
    stats_dict = {'not failed': {}, 'failed': {}}
    for key in stats_dict:
        # 'type1 err' is the number of elements belonging to that class
that are mis-classifies. 'type2 err' is the
        # number of elements classified in that class among all those
that were classifies wrong. 'type1 mistakes'
        # tells how many times an element belonging to that class is
incorrectly assigned to other classes.
        # 'type2 mistakes' measures which classes are most likely to be
misclassified with that one.
        stats_dict[key] = {'type1 err': 0, 'type1 mistakes': [], 'type2 err':
0, 'type2 mistakes': []}
        wrong_1 = sum(abs(is_correct(x) - 1) for x in
predictions_dict_type1[key])
        wrong_2 = sum(abs(is_correct(x) - 1) for x in
predictions_dict_type2[key])
```

```
        wrong_1_ratio = wrong_1 / len(predictions_dict_type1[key])
        wrong_2_ratio = wrong_2 / len(predictions_dict_type2[key])
        stats_dict[key]['type1 err'] = wrong_1_ratio
        stats_dict[key]['type2 err'] = wrong_2_ratio
    for key in stats_dict:
        print()
        print('CLASS {}:'.format(key))
        print('type 1 elements', len(predictions_dict_type1[key]))
        print('type 1 error:', stats_dict[key]['type1 err'])
        print('type 2 elements', len(predictions_dict_type2[key]))
        print('type 2 error:', stats_dict[key]['type2 err'])
def validation_set_errors():
    prediction = use_neural_network('validation')
    print()
    print('\nCalculating errors in validation set...')
    predictions_dict_type1 = {'not failed': [],
                              'failed': []}
    predictions_dict_type2 = {'A': [],
                              'B': [],
                              'C': [],
                              'D': [],
                              'E': [],
                              'F': [],
                              'Def': []}
    predictions_dict_type2 = {'not failed': [],
                              'failed': []}
    for elem in prediction:
        predictions_dict_type1[elem[0]].append(elem)
    for elem in prediction:
        predictions_dict_type2[elem[1]].append(elem)
    correct_guesses = sum(is_correct(x) for x in prediction)
    correct_ratio = correct_guesses / len(prediction)
    print('correct:', correct_ratio)
    # stats_dict = {'A': {}, 'B': {}, 'C': {}, 'D': {}, 'E': {}, 'F': {}, 'Def':
{}}
    stats_dict = {'not failed': {}, 'failed': {}}
    for key in stats_dict:
        # 'type1 err' is the number of elements belonging to that class
that are mis-classifies. 'type2 err' is the
        # number of elements classified in that class among all those
that were classifies wrong. 'type1 mistakes'
        # tells how many times an element belonging to that class is
incorrectly assigned to other classes.
        # 'type2 mistakes' measures which classes are most likely to be
misclassified with that one.
        stats_dict[key] = {'type1 err': 0, 'type1 mistakes': [], 'type2 err':
0, 'type2 mistakes': []}
        wrong_1 = sum(abs(is_correct(x) - 1) for x in
predictions_dict_type1[key])
        wrong_2 = sum(abs(is_correct(x) - 1) for x in
predictions_dict_type2[key])
        wrong_1_ratio = wrong_1 / len(predictions_dict_type1[key])
        wrong_2_ratio = wrong_2 / len(predictions_dict_type2[key])
```

```
            stats_dict[key]['type1 err'] = wrong_1_ratio
            stats_dict[key]['type2 err'] = wrong_2_ratio
      for key in stats_dict:
            print()
            print('CLASS {}:'.format(key))
            print('type 1 elements', len(predictions_dict_type1[key]))
            print('type 1 error:', stats_dict[key]['type1 err'])
            print('type 2 elements', len(predictions_dict_type2[key]))
            print('type 2 error:', stats_dict[key]['type2 err'])
if __name__ == '__main__':
      # print(prediction)
      test_set_errors()
      validation_set_errors()
```

RNNtrain_and_use.py

This code is used for both the training and the test phase of the RNN model.

```
# this code is a modified version of: http://monik.in/a-noobs-guide-to-
implementing-rnn-lstm-using-tensorflow/
from time import localtime, strftime
import pickle
import tensorflow as tf
import numpy as np
import os
def remove_old_files():
      files_to_remove = ['checkpoint', 'datetime.pickle', 'model.ckpt.data-
00000-of-00001',
                                    'model.ckpt.index', 'model.ckpt.meta', 'tf.log',
'network_structure.pickle',
                                    'test_set.pickle', 'train_set.pickle',
'validation_set.pickle']
      for file in files_to_remove:
            try:
                  os.remove(file)
            except FileNotFoundError:
                  print('file "{}" not found'.format(file))
# here I am using a label like: [1, 0] is 'non failed' and [0, 1] is 'failed', so
the output
# of the network will consist in two neurons, one for each class. But does it
make sense?
# wouldn't it be better to use just one neuron and if it is > 0.5 the firm
failed?
def calc_label(firm):
      if int(firm) == 0:
            label = [1, 0]
      else:
            label = [0, 1]
      return label
# this converts labels from data so that a data containing [1, 0, 0, 0, 0, 0,
0] becomes 'A',
# [0, 1, 0, 0, 0, 0, 0] becomes 'B' and so on. It is the inverse of
calc_label(x)
def convert_label(data):
      ranks = ['not failed', 'failed']
      for i, value in enumerate(data):
            if value == 1:
                  return ranks[i]
def convert_prediction(value):
      predict = ''
      if int(value) == 1:
            predict = 'failed'
      elif int(value) == 0:
            predict = 'not failed'
      return predict
def is_correct(x):
      if x[0] == x[1]:
            return 1
      else:
            return 0
def set_errors(test_or_validation_or_train):
      set_prediction = use_neural_network(test_or_validation_or_train)
      # element os set_predictions are like: [array([1, 0]), 'non failed']
      # type_1 error is the share of failed classified as 'non failed'
      # type_2 error is the share of non failed classified as 'failed'
      # error_share is the number of elements in type_1 over the number of
elements in type_2.
      non_failed = [x[1] for x in set_prediction if x[0][0]==1]
      failed = [x[1] for x in set_prediction if x[0][0]==0]
      type_1 = [x for x in failed if x=='not failed']
      type_2 = [x for x in non_failed if x=='failed']
      if len(non_failed) == 0:
            type_1_ratio = 0
      else:
            type_1_ratio = len(type_1) / len(failed)
      if len(failed) == 0:
            type_2_ratio = 0
      else:
            type_2_ratio = len(type_2) / len(non_failed)
      if len(type_2) == 0:
            error_share = 1
      else:
            error_share = len(type_1) / (len(type_2) + len(type_1))
      return type_1_ratio, type_2_ratio, error_share
def use_neural_network(test_or_validation_or_train):
      global prediction, sess
      if test_or_validation_or_train == 'test':
            set_x = test_x
            set_y = test_y
      elif test_or_validation_or_train == 'validation':
            set_x = validation_x
            set_y = validation_y
      else:
            set_x = train_x
```

```
        set_y = train_y
    set_prediction = prediction.eval(session=sess, feed_dict={data:
set_x, dropout: 0})
    set_prediction = tf.argmax(set_prediction, 1)
    set_prediction = np.array(set_prediction.eval(session=sess))
    predictions = [convert_prediction(x) for x in set_prediction]
    result = list(zip(set_y, predictions))
    return result
def create_RNN_model():
    global cell, val, state, last, weight, bias, prediction
    global cross_entropy, optimizer, minimize, mistakes, error
    if num_layers == 1:
        cell = tf.nn.rnn_cell.LSTMCell(num_hidden, state_is_tuple=True)
        cell = tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob=1.0 -
dropout)
    elif num_layers == 2:
        # cell = tf.nn.rnn_cell.LSTMCell(num_hidden, state_is_tuple=True)
        cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers,
state_is_tuple=True)
        cell = tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob=1.0 -
dropout)
    val, state = tf.nn.dynamic_rnn(cell, data, dtype=tf.float32)
    val = tf.transpose(val, [1, 0, 2])
    last = tf.gather(val, int(val.get_shape()[0]) - 1)
    prediction = tf.nn.softmax(tf.matmul(last, weight) + bias)
    regularizer = tf.nn.l2_loss(weight)  # L2 regularization
    cross_entropy = -tf.reduce_sum(target *
tf.log(tf.clip_by_value(prediction, 1e-10, 1.0)))
    cross_entropy = tf.reduce_mean(cross_entropy + 0.01 * regularizer)
# L2 regularization
    optimizer = tf.train.AdamOptimizer(learn_rate)
    minimize = optimizer.minimize(cross_entropy)
    mistakes = tf.not_equal(tf.argmax(target, 1), tf.argmax(prediction, 1))
    error = tf.reduce_mean(tf.cast(mistakes, tf.float32))
def training_and_using():
    global cell, val, state, last, weight, bias, prediction, sess
    global cross_entropy, optimizer, minimize, mistakes, error
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())
    graph = []
    num_of_batches = int(len(train_x) / batch_size)
    for epoch in range(epochs):
        ptr = 0
        cost = 0
        for j in range(num_of_batches):
            batch_x, batch_y = train_x[ptr:ptr + batch_size],
train_y[ptr:ptr + batch_size]
            ptr += batch_size
            sess.run(minimize, {data: batch_x, target: batch_y, dropout:
0.5, learn_rate: learning_rate})
            cost += cross_entropy.eval(session=sess, feed_dict={data:
batch_x, target: batch_y, dropout: 0.5})
            if j == num_of_batches - 1:
                    batch_train_error = error.eval(session=sess,
feed_dict={data: batch_x, target: batch_y, dropout: 0})
          print("Epoch:", str(epoch))
          if (epoch+1) % save_step == 0:
               incorrect = error.eval(session=sess, feed_dict={data: test_x,
target: test_y, dropout: 0})
               incorrect_validate = error.eval(session=sess,
feed_dict={data: validation_x, target: validation_y, dropout: 0})
               accuracy = 100 * (1 - incorrect)
               accuracy_validation = 100 * (1 - incorrect_validate)
               training_set_accuracy = (1 - np.mean(batch_train_error)) *
100
               print('Epoch {:2d} loss {:4.2f}'.format(epoch, cost))
               print('Epoch {:2d} train accuracy {:4.2f}%'.format(epoch,
training_set_accuracy))
               print('Epoch {:2d} test accuracy {:4.2f}%'.format(epoch,
accuracy))
               type_1_ratio, type_2_ratio, error_share =
set_errors('validation')
               print('type 1 error: {}%'.format(round(type_1_ratio*100, 1)))
               print('type 2 error: {}%'.format(round(type_2_ratio*100, 1)))
               print('type 1 over type 2: {}
%'.format(round(error_share*100, 1)))
               graph.append([epoch, training_set_accuracy, accuracy, cost,
type_1_ratio, type_2_ratio, error_share, accuracy_validation])
               # folder = os.path.dirname(os.path.realpath(__file__))
               # epoch_folder = 'epoch_{}'.format(i)
               # epoch_folder_name = os.path.join(folder_name,
epoch_folder)
               # saver.save(sess, epoch_folder_name + "/model.ckpt")
     with open(os.path.join(folder_name, 'graph.pickle'), 'wb') as f:
          pickle.dump(graph, f)
     # accuracy_list = use_neural_network('test')
     # print(accuracy_list[:100])
     # for i in accuracy_list[:100]:
     #     if i[1] == 'failed':
     #          print(i)
     sess.close()
def set_variables():
     global train_x, train_y, test_x, test_y, validation_y, validation_x
     global test_set, train_set, validation_set, folder_name,
network_structure
     dataset = 'original_numpyfied_database.pickle'
     print('loading dataset...')
     with open(dataset, 'rb') as f:
          input_data = pickle.load(f)
     input_data = np.array(input_data)
     np.random.shuffle(input_data)
     input_data = np.transpose(input_data, (1, 0, 2))
     input_data = input_data[:years_n]
     input_data = np.transpose(input_data, (1, 0, 2))
     input_data = np.transpose(input_data, (2, 1, 0))
     print('creating features and labels...')
```

```
    labels = input_data[0][0]
    labels = np.array([calc_label(x) for x in labels])
    features = input_data[1:]
    features = np.transpose(features, (2, 1, 0))
    del input_data
    print('rebalancing dataset...')
    features_and_labels = list(zip(features, labels))
    failed = np.array([x for x in features_and_labels if x[1][0]==0])
    non_failed = np.array([x for x in features_and_labels if x[1][0]==1])
    np.random.shuffle(non_failed)
    if rebalance_dataset:
        non_failed = non_failed[:len(failed)]
    features_and_labels = np.concatenate((non_failed, failed), axis=0)
    np.random.shuffle(features_and_labels)
    features = np.array([x[0] for x in features_and_labels])
    labels = np.array([x[1] for x in features_and_labels])
    del features_and_labels
    print('dividing test and validation sets...')
    train_size = int(len(labels) * 0.8)
    validate_size = int(len(labels) * 0.9)
    train_y = np.array(labels[:train_size])
    test_y = np.array(labels[train_size:validate_size])
    validation_y = np.array(labels[validate_size:])
    del labels
    train_x = np.array(features[:train_size])
    test_x = np.array(features[train_size:validate_size])
    validation_x = np.array(features[validate_size:])
    del features
    print(train_x.shape)
    print(train_y.shape)
    print(test_x.shape)
    print(test_y.shape)
    print(validation_x.shape)
    print(validation_y.shape)

    folder_name = strftime("%d-%m-%Y_%H:%M:%S", localtime())
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
    network_structure = [num_hidden, num_layers, batch_size, epochs,
save_step, rebalance_dataset]
    with open(os.path.join(folder_name, 'network_structure.pickle'), 'wb')
as f:
        pickle.dump(network_structure, f)
    test_set = list(zip(test_x, test_y))
    validation_set = list(zip(validation_x, validation_y))
    train_set = [train_x, train_y]
if __name__ == '__main__':
    for _ in range(10):
        num_hidden = 50  # number of hidden neurons per layer
        num_layers = 1  # number of hidden layers, either 1 or 2
        batch_size = 5000
        epochs = 200
        save_step = 1
        years_n = 5  # numbers of years in the balance to use
        rebalance_dataset = True
        learning_rate = 0.001
        weight = tf.Variable(tf.truncated_normal([num_hidden, 2]))
        bias = tf.Variable(tf.constant(0.1, shape=[2]))
        data = tf.placeholder(tf.float32, [None, years_n, 27])
        target = tf.placeholder(tf.float32, [None, 2])
        learn_rate = tf.placeholder(tf.float32)
        dropout = tf.placeholder(tf.float32)
        saver = tf.train.Saver()
        remove_old_files()
        set_variables()
        create_RNN_model()
        training_and_using()
        tf.reset_default_graph()
```