BACHELOR'S THESIS IN ECONOMICS 2017

CORPORATE BANKRUPTCY PREDICTION USING MACHINE LEARNING TECHNIQUES

BJÖRN MATTSSON & OLOF STEINERT
supervised by ANDREAS DZEMSKI

Abstract

Estimating the risk of corporate bankruptcies is of large importance to creditors and investors. For this reason bankruptcy prediction constitutes an important area of research. In recent years artificial intelligence and machine learning methods have achieved promising results in corporate bankruptcy prediction settings. Therefore, in this study, three machine learning algorithms, namely random forest, gradient boosting and an artificial neural network were used to predict corporate bankruptcies. Polish companies between 2000 and 2013 were studied and the predictions were based on 64 different financial ratios.

The obtained results are in line with previously published findings. It is shown that a very good predictive performance can be achieved with the machine learning models. The reason for the impressive predictive performance is analysed and it is found that the missing values in the data set play an important role. It is observed that prediction models with surprisingly good performance could be achieved from only information about the missing values of the data and with the financial information excluded.

UNIVERSITY OF GOTHENBURG
SCHOOL OF BUSINESS, ECONOMICS AND LAW

Department of Economics
UNIVERSITY OF GOTHENBURG
SCHOOL OF BUSINESS ECONOMICS AND LAW
Gothenburg, Sweden 2017

Corporate Bankruptcy Prediction using Machine Learning Techniques
BJÖRN MATTSSON
OLOF STEINERT

Supervisor: Andreas Dzemski
Bachelor's Thesis 2017
Department of Economics
UNIVERSITY OF GOTHENBURG
SCHOOL OF BUSINESS ECONOMICS AND LAW
Box 100, S-405 30 Gothenburg
Telephone +46 31-786 0000

Typeset in LaTeX
Gothenburg, Sweden 2017

Acknowledgements

We want to express our sincere gratitude to our supervisor Andreas Dzemski for helping us throughout the process, and offering valuable guidance in how to make machine learning interesting for economists. We also want to thank Tim Falk and Alexander Piauger for offering valuable feedback during the opposition.

Björn Mattsson & Olof Steinert, London & Stockholm, August 2017

Contents

1

Introduction

1.1 Motivation

Estimating the risk of corporate bankruptcies is of large importance to creditors and investors. There are large indirect and direct costs associated with financial distress and bankruptcies (Altman, 1984) and for this reason bankruptcy prediction has for a long period of time constituted an extensive area of research (Linden et al., 2015). Corporate bankruptcies can have serious effects both locally and globally (De Haas and Van Horen, 2012), employees, investors, customers, suppliers and their financiers are all affected when a company disappears (Engström, 2002). In some cases a corporate bankruptcy can cause an entire industry to suffer (Yang et al., 2015).

Until recently the dominating methods for predicting corporate bankruptcies have been based on statistical modelling, however, lately models based on machine learning have been proposed (Linden et al., 2015). Recently, machine learning models have successfully been used for many classification and regression problems and these models have often outperform traditional classification methods (Krizhevsky et al., 2012). The purpose of bankruptcy prediction is to assess the financial health status and future perspectives of a company. For a given period of time this problem can be modelled as a two-class classification problem (Zięba et al., 2016). Companies either survive the given time period, or go bankrupt during it. The problem is to predict which of these two possible outcomes that is the more probable one.

Financial reporting laws and public expectations on transparency means that there is a lot of data available concerning companies' financial status (Bredart, 2014). The large amount of data makes the area well suited for sophisticated data intensive computation methods (Qiu et al., 2016).

1.2 Problem formulation

The intention of the study is to illustrate and investigate how machine learning can be exploited in the field of economics. More specifically, the aim is to study how machine

learning methods can be used to predict corporate bankruptcies.

The findings of the Zięba et al. (2016) serve as starting point for this survey. Zięba et al. suggests a machine learning approach for the problem of predicting corporate bankruptcies and reports impressive predictive performance. In their study they use financial ratios to address bankruptcy prediction of polish companies in the manufacturing sector between 2000 and 2013.

The intention of this study is to *reproduce*, *improve* and *extend* the work of Zięba et al. (2016). The first objective is to reproduce the impressive predictive performance, as a second step we want to investigate if we can achieve better performance for a neural network than reported by the authors. In the paper Zięba et al. reports very poor performance for a neural network based classifier. This is surprising since neural networks lately have proved to be very successful in classification tasks in general and also performed well in bankruptcy prediction settings (Bredart, 2014). For this reason another objective of the project is propose a neural network based classifier with improved performance on the data set. The intention is to identify key success factors for the performance of neural networks in the context of corporate bankruptcy prediction. Lastly, we want to extend the research scope by analysing underlying reasons for the outstanding performance reported for some of the algorithms used by Zięba et al. (2016). It seems unlikely that is possible to make such accurate predictions and for this reason we explore the importance of the missing values in the data set and the implication of their unbalanced distribution.

In summary, the results presented by Zięba et al. (2016) were reproduced. In addition, a neural network with significantly better performance than previously reported was built. The neural network based model was almost on par with the best performing tree-based models. Finally, it was found that missing values in the data set play an important role, they carry almost all information that is useful for the predictions. This is, of course, problematic and raises questions about the validity of the data set of polish companies first used by Zięba et al. (2016).

1.3 Scope

During this study corporate bankruptcy prediction using machine learning methods have been studied. To facilitate comparison the same data set that were used in the context of bankruptcy prediction by Zięba et al. (2016) was studied. Such an approach facilitated validation and benchmarking of results.

It should be underlined that the intention of the study was not to suggest an ideal prediction framework or to achieve an optimal prediction algorithm since such objectives would make the project more about engineering than about economics.

Moreover, it should be pointed out that no attempts to add any new information to the

data set have been made. It is likely, or at least possible, that new information retrieved from other data sources would be helpful. For example inflation, growth and competition indices would possibly carry useful information, however, such work is beyond the scope of this study.

The study addresses bankruptcy prediction of polish companies in the manufacturing sector between 2000 and 2013, but the methods used should be considered externally valid and applicable to other sectors and countries as well. An aspect that naturally would have been interesting to examine when investigating bankruptcy prediction models is how well they generalise to data collected at other time periods. However, the exploited data set did not allow for such analysis since the individual observations were not labelled with dates.

1.4 Related work

The first documented attempts of bankruptcy predictions were carried out by Patrick (1932). At that time no statistical model was used, the predictions were based on Patrick's own interpretations of the financial ratios and the trends he could discern. First in the 1960s statistical models and hypothesis testing were used for bankruptcy prediction. The work was initiated by Beaver (1966) and two years later Altman (1968) proposed the use of multiple discriminant analysis for bankruptcy prediction. This work was trend-setting and in the years that followed Altman's ideas inspired many others.

Another turning point in the field was the initiation of the generalised linear models (Ohlson, 1980). Generalised linear models have some advantages, firstly, they allow for analysis of the certainty of the predictions and, secondly, it is possible to analyse the effect of each predictor individually.

In recent years machine learning methods have at several times been successfully used to predict corporate bankruptcies. Neural networks trained with back propagation are the most common method for this type of problem (Tsai and Wu, 2008). In a study of small and medium-sized Belgian companies it was shown that relatively good results could be achieved by using only a small number of easily accessible financial ratios as inputs to an artificial neural network (Bredart, 2014).

The study of the Belgian companies was not the first of its kind. It was inspired by, among others, Shah and Murtaza (2000) who used a neural network to predict bankruptcies among US companies between 1992 and 1994 and Becerra et al. (2005) that studied British corporate bankruptcies between 1997 and 2000 using a similar method.

Lately, some attention has also been devoted to ensemble classifiers. It has by Alfaro et al. (2008) and Zięba et al. (2016) been shown that ensemble classifiers can successfully be applied to bankruptcy prediction and significantly outperform other methods.

2

Theory

This chapter aims to give a theoretical background to some technical concepts that are important for the project. The chapter gives a general introduction to machine learning and explains how machine learning techniques can be used for bankruptcy predictions. It also describes how the performance of the prediction models can be evaluated. The machine learning algorithms decision trees, random forest and gradient boosting are introduced briefly, whereas to artificial neural networks a more detailed introduction is given. Lastly, some simple techniques for data pre-processing are described.

2.1 Introduction to machine learning

Machine learning is a field that focuses on drawing conclusions from large amounts of data. This is done by letting a model find structures and relationships in the data. By presenting a machine learning model with samples from a data set the model *learns* learns to represent hidden relationships and patterns in the data. This process is commonly referred to as training. After training the algorithm is supposed to be able to generalise what it has learned to new, unseen, samples (Friedman et al., 2001). The ability of machine learning models to generalise to new data points is what engineers and researchers that choose to use machine learning methods are most interested in.

The field of machine learning combines computation and statistics (Bishop, 2006). Computational power is needed in the training procedure, which sometimes can take weeks even on computers built especially for the purpose. Statistics is used to derive the mathematics behind the machine learning models. It can be argued that the main reasons for the increased attention to machine learning during recent years are the increased processing power of modern computers and the large amounts of data easily accessible in today's digital society (Salvador et al., 2017).

2.1.1 Machine learning problems

The machine learning field can be divided into three different sub-fields: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning is about finding a predictive model that maps inputs to outputs (Hastie et al., 2009). To achieve this the learning algorithm is presented with examples of input and output pairs. The goal of the learning process is to produce a function that gives the desired output for inputs that have not been presented to the algorithm during training. It is achieved by fitting internal parameters, often referred to as θ, to the data it tries to approximate. The optimal set of parameters found by the algorithm is often denoted $\hat{\theta}$. The focus in supervised learning is to find a model that as good as possible maps unseen inputs to outputs. This is clearly similar to well known techniques as e.g. ordinary least squares. We want to use the data to make forecasts about future data points. In machine learning the models are generally very flexible and the internal parameters can thus recreate many different types of patterns in the data. However, the downside of this is that the values of the fitted parameters cannot easily be analysed with statistical rigour, and they typically lack structural or causal interpretation. This in contrast with ordinary least squares, where the fitted parameters, $\hat{\beta}$, can be used to understand causal relationships in the data.

Supervised learning can further be broken down into classification and regression problems. In classification problems the data points belong to two or more classes, and the problem is to assigns unseen inputs to the correct class. In regression, on the other hand, the outputs are continuous rather than discrete and cannot be summarised into classes.

In unsupervised learning no labels are given to the learning algorithm. The problem is to find a descriptive model of the input (Friedman et al., 2001). An example of a possible unsupervised learning problem is customer segmentation. In such a problem we want to assign customers to different categories, which we prior to using the model do not know which they are. The customer segments can later can be used for targeted marketing.

Lastly, reinforcement learning is about learning a policy: what actions are most beneficial depending on the circumstances. In reinforcement learning the learning algorithm is not presented any supervised output but instead a delayed reward. Instead of starting with a pre-defined data set, the model itself collects the data. Reinforcement learning is often used in a game like framework, e.g. a robot in a maze. Reinforcement learning recently became widely popular when Mnih et al. (2015) presented and algorithm that could learn to play many Atari games with super-human performance.

This thesis considers only supervised learning. Corporate bankruptcy prediction is treated as a two-class classification problem. A company either belongs to the class of companies that faces bankruptcy or to the class that still operates after a given time period.

2.1.2 Machine learning in economics

Machine learning and artificial intelligence have in many different fields proved to be very useful, for example the techniques has revolutionised the computer vision field (Sharif Razavian et al., 2014). However, the role of machine learning in economics has so far been limited. In economics structural models are instead dominating. The reason for this is likely that in economics it is often more important to be able to draw statistical significant conclusions about causal effects from the model than having a model with as good predictive performance as possible. With that said there is still large potential for machine learning in economics: there is much data available and a need to understand it. Many problems in the field of economics can also be regarded in the terms of classification or regression (McNelis, 2005). Furthermore linear models cannot fully exploit the information in the data since the underlying relationship often is not linear but non-linear (Clements et al., 2004). Neural networks are able to approximate any continuous function and therefore they could be expected to provide more effective models in such applications. Kuan and White (1994) provides a good overview of how artificial neural networks can be used in econometrics.

From these advantages and disadvantages it is clear that machine learning is suitable for problems where what is of interest not necessarily is to understand the governing mechanisms, but rather to have a model with as good predictive performance as possible. Predicting inflation is an important but difficult task faced by economists and central banks. Historically, this has been done with different types of autoregressive models. Lately, artificial neural networks, or simply neural networks, have gained more attention as prediction models. Nowadays there is a number of different types of neural network models that are commonly used in the field (Choudhary and Haider, 2012). Gupta and Kashyap (2015) provides a pedagogical description of how neural networks can be used to predict inflation.

Neural networks are extensively used in finance for predicting stock prices (Anderson and McNeill, 1992). There is also a a number of attempts to predict the currency exchange rates using machine learning methods (Verkooijen, 1996). However, in macroeconomics, the use of neural networks or even machine learning methods in general are not very common yet.

Bankruptcy prediction is another field of economics which is compatible with machine learning. To a bank that lends money to small companies the possibility to use the model to draw statistical significant conclusions about the underlying behaviour is of limited interest. On the other hand, having a model with good predictive power is of highest importance.

2.1.3 Model selection

In machine learning the data set is typically dived into a training and a test set. The training set is used by the model to discover and unveil hidden patterns and relationships in the data. The test set is used to measure the strength and utility of the trained model (Bishop, 2006). A loss or cost function is typically defined and the training could then be seen as a optimisation problem. The idea is that the cost function quantifies the error that the model makes in its predictions of the desired output (Nielsen, 2015).

Overfitting occurs when the model is more complex than the data relationship. The model loses its predictive power when it has started to learn random noise in the data. The opposite to overfitting is underfitting and this means that the model does not fully describe the underlying relationship between the inputs and the outputs (Wahde, 2008).

The problems of over- and underfitting is summarised by the bias–variance tradeoff. The bias and the variance could be seen as two different sources of error. The error due to variance is the error caused by the sensitivity of the algorithm to small fluctuations in the training set in the model building process (overfitting) and the error due to bias is the error caused by incorrect or insufficient assumptions of the model (underfitting) (Von Luxburg and Schölkopf, 2008).

To avoid overfitting a validation set is sometimes used in addition to the training and test sets. Then the errors of predictions on both the training and validation set can be monitored during training. However, no feedback is given to the algorithm regarding the performance of the predictive model on the validation set. The information about the performance on the validation set is used to retrieve the model that best generalises after training (Wahde, 2008). The test set is finally used as a final check to see that the model generalises well. Figure 2.1 shows how overfitting can be avoided by using a validation set.

2.1.3.1 Cross validation

Another method which is commonly used to avoid overfitting is cross validation (Arlot et al., 2010). Instead of statically dividing the data set into one part which is used for training and another which is used for testing (and potentially also a validation set) a dynamic division is used. The data set is divided into x equally large subsets, $x - 1$ of those are used for training and the last subsets is used for testing. This procedure is repeated x times, with each of the x subsets acting as test set once each. In this way we get x measurements of the scoring metric that we are concerned with, and we can calculate both the mean and standard error of this scoring metric. This makes overfitting improbable, since it will stop us from choosing a model that performs well for only a particular division of the data set. Instead,the model that works best on many different

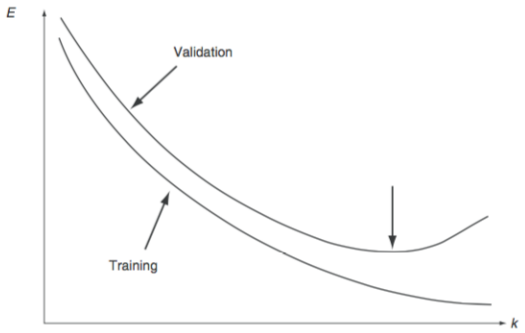

Figure 2.1: The figure illustrates the overfitting phenomenon. The number of training epochs are plotted on the x-axis and the training error on the y-axis. As the training progresses, the training error is gradually reduced. The validation error also drops in the beginning, however, after some time, the validation error begins to grow as the model starts fitting the noise in the data. When this happens it is time to stop training.

divisions is chosen. However, cross validation requires that the model is trained x times on the data subsets and this increases the required computation time. For this reason cross validation is in some cases unfeasible (Blockeel and Struyf, 2002).

2.2 Measurement metrics

To measure the performance of the machine learning algorithm we need to use of some performance metric. There are many different performance metrics to choose from. Depending on the problem and the purpose different metrics are suitable. For example, different performance metrics are generally used in regression and classification. Since we are only concerned with classification problems we will only present metrics that are commonly used for this type of task.

2.2.1 Accuracy

Accuracy is probably the metric that is easiest to understand. It is defined as:

$$\text{Acc} = \frac{t}{N},$$

where t is the number of samples which were correctly classified and N is the total number of samples (including both correctly and incorrectly classified examples) (Bishop, 2006). Accuracy is easy to understand as it tells us how often the classifier is correct. For example, an accuracy of 0.8 means that in 80% of the cases the classifier gives the correct predictions. However, it does not take into account cases where the classifier is incorrect. It might be that the classifier is very good at doing predictions when our sample belongs

Table 2.1: The figure illustrates how a confusion matrix is constructed. Each classified data point ends up in one of the four cells in the matrix, which cell is dependent on what class that is predicted and what class the data point actually belongs to.

	Predicted class 1	Predicted class 0
Actual class 1 (P)	True positive (TP)	False negative (FN)
Actual class 0 (N)	False positive (FP)	True negative (TN)

to one of the classes, but very poor when the sample belongs to the other class. Also, depending on the data set a trivial classifier can achieve good accuracy. Let us, for example, assume that we have a machine that predicts whether or not a company will go bankrupt in a certain time period. If 80% of the companies in the study still operates after the given time period a machine that predicts "still operating" for all companies (including the ones that actually face bankruptcy) would be correct in 80% of the cases, and would thus have an accuracy of 0.8. This would of course be a completely useless machine, however, the reported accuracy can still appear to be quite impressive.

2.2.2 Confusion matrix

To better analyse the performance of our classifier we can visualise the performance of the classifier in something that is called a confusion matrix, showed in Table 2.1. In a confusion matrix the number of true positives, false positives, true negatives and false negatives are listed (Sing et al., 2005). The machine discussed in the previous paragraph would get many true positives (companies that still operates and also are predicted do so), but unfortunately also many false positives (companies that face bankruptcy, but the machine predicts to be still operating). It will get no true negatives or false negatives since it never predicts that a company will face bankruptcy.

2.2.3 Sensitivity and specificity

From the confusion matrix many informative metrics can be derived. Two such metrics are the sensitivity and the specificity, also called the true positive rate (TPR) and the false positive rate (FPR) (Sing et al., 2005). They are specified as:

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN},$$
$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}.$$

Measured with these metrics the machine in the example above would get $TPR = 1$ which is very good, but also $FPR = 1$ which signifies poor performance.

Different classifiers will have different values of true positive rates and false positive rates. These two numbers are typically a good way of measuring desired performance of a classifier. It is important to understand the application when analysing these numbers. In the example above, it would, in some applications, be more important with a low false positive rate than a high true positive rate, but in others the other way around. For example, if a bank seeks to invest in companies typically a high true positive rate is more important; a high true positive rate means that few companies that will end up bankrupt are missed. Whereas for a governmental organisation that wants to discover companies that are likely to go bankrupt, to offer them some sort of costly support, it is more important with a low false positive rate to avoid wasting time and resources on companies that would survive even without the extra help.

2.2.4 Receiver operating characteristic

Most machine learning algorithms does not return a binary prediction e.g. (*still operating* or *bankrupt*), but rather a probability. To calculate the true positive rate and false positive rate from the returned probabilities we have to chose a threshold, which we use to decide which class we consider a sample to belong to. For example all companies that the machine think will face bankruptcy by more than 70% probability we will consider as bankrupt. As we increase this threshold both the true positive rate and the false positive rate will decrease. If we set the threshold to 100% we will have $TPR = FPR = 0$ and if we set the threshold to 0% we will get $TPR = FPR = 1$. In between these two endpoints we have a space of opportunity upon which we can decide how we want our classifier to perform. The curve which connects these two values are called a Receiver operating characteristic (ROC) curve. In Figure 2.2 we can see an example of such a ROC curve for one of the algorithms which have been developed in this study. The dashed line represents what we would get by random guessing, and the blue solid line represents what our algorithm returns. Values closer to the upper left corner are better since they mean a high true positive rate as well as a low false positive rate (Bishop, 2006) .

2.2.5 Area under curve

From just looking at a graph it is difficult to compare the performance of two different algorithms. To overcome this a metric called Area under curve (AUC) has been derived. It is, as the name suggests, defined as the area under the ROC curve. An algorithm that guesses randomly will obtain a value of 0.5, since the area under a triangle with legs of length one is precisely 0.5. The AUC value for the algorithm in Figure 2.2 is 0.852. AUC can be interpreted as how much opportunities our algorithms give us for different scenarios, where potentially different measures are of importance.

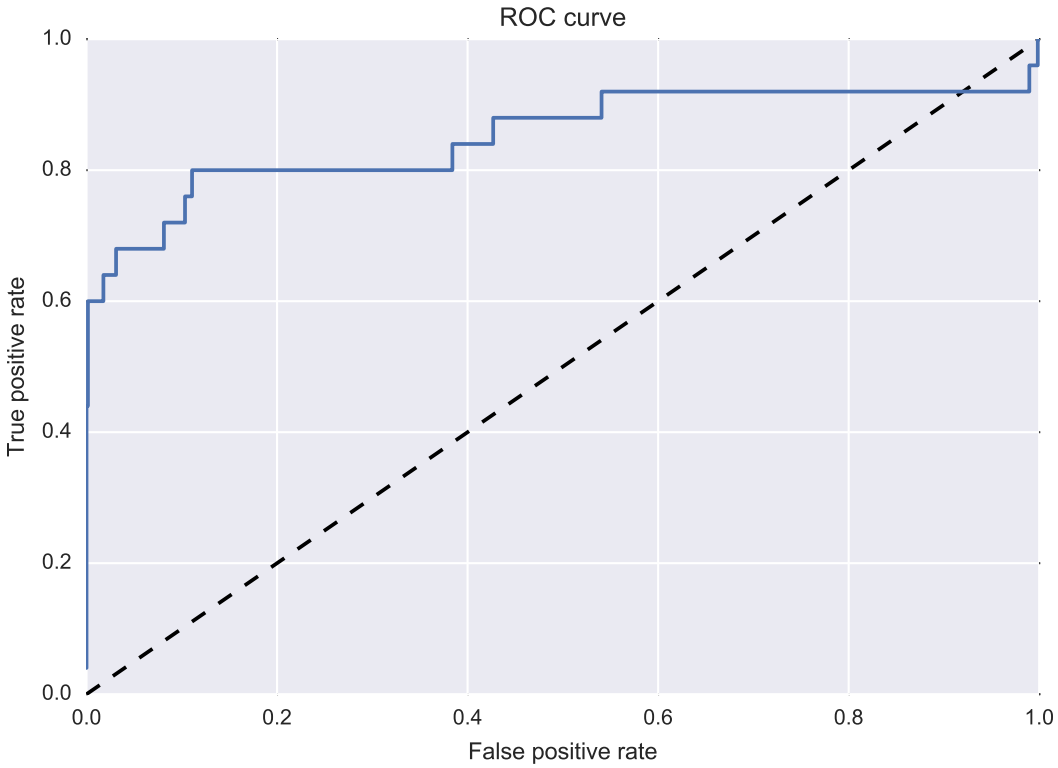

Figure 2.2: Receiver operating characteristic generated for one of the experiments that were run in this study.

2.2.6 Cross-entropy

Cross entropy is a loss function used in machine learning and classification problems (LeCun et al., 2015). A loss function or cost function is used to quantify how how well a machine learning model performs its intended task; a quality-of-fit measure. In the context of classification the loss function defines the penalty for an incorrect classification of an observation. By e.g. taking the sum over the complete training set the operational performance of a model can be quantified as a real number.

For a two class classification problem the cross entropy is defined as:

$$C = -\frac{1}{N}\sum_{n=1}^{N}[y_n \ln \hat{y}_n + (1 - y_n) \ln (1 - \hat{y}_n)] \tag{2.1}$$

where N is the total number of observations, $\hat{y}$ is the output of the machine learning model and y is the desired one-hot encoded output (Nielsen, 2015). The function is strictly positive and when $\hat{y}$ is close to y the the penalty is close to zero. The cross entropy is particularly suitable for neural networks since it solves the problem of learning slowdown (Nielsen, 2015). The cross entropy can be recognised as the negative of the averaged log-likelihood over the data set. And minimising the loss therefore equals finding the maximum likelihood estimate (MLE).

2.3 Machine learning models

2.3.1 Decision trees

Tree-based learning algorithms are common in supervised learning, in Bishop (2006) a good introduction to tree-based methods is given. A decision tree is essentially a set of splitting rules that can be summarised in a tree structure. The different split points are called nodes and at each node a decision is applied. A decision rule could for example be "Is the height of the object greater than 2 meters". Starting from what is called the root node the tree is built top down. The data set is then divided into different bins by splitting the data into smaller and smaller subsets. At each decision point the data is branched into subsets.

The training data is used to construct the tree. Then, in order to reach a decision, you just keep applying decisions until you reach a terminal node. The intermediate nodes are often referred to as internal nodes and terminal nodes are called leaves. The class is determined by the empirical knowledge of what classes other data points from the training data that ended up in the same leaf belonged to. Decision trees are practical since they, unlike many other classification algorithms, can handle both numerical and categorical data.

The splitting rules are found by searching through the space of all possible splitting attributes. The approach is greedy in the sense that at each node the best splitting attribute is chosen. In other words attributes are not selected with the best final model in mind. The aim is to partition the data into subsets that, to at least some extent, contain similar data points. Ideally the data is split into homogeneous subsets.

Different criterion to determine the best split can be used. A commonly used criterion is to choose the split that maximises the information gain. The information gain is a measure of the reduction of uncertainty after a split. To measure uncertainty entropy is used. Formally the entropy of a state S is defined as:[1]

$$H(S) = -\sum_{i=1}^{K} p(i) \log p(i) \tag{2.2}$$

where K is the number of branches and p the probability of each branch (Jaynes, 1957).

From the definition it could be seen that a homogeneous sample has the entropy zero and an equally dived sample has the entropy one.

[1]Entropy is a theoretical measure of information content, or unpredictability, of a state S. Learning the outcome of a state with high entropy gives us more information, than learning the outcome of a state with low entropy. We want to store as much information as possible in the Decision trees, therefore the entropy measure is suitable.

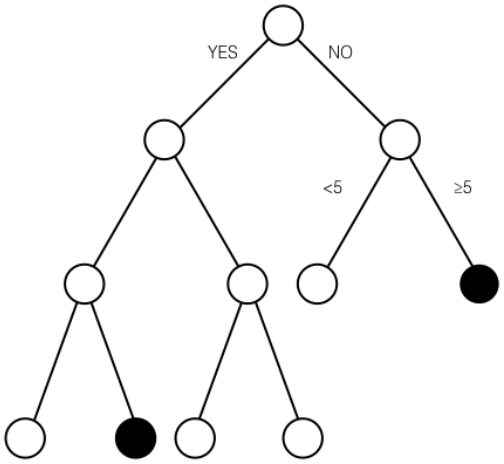

Figure 2.3: A very simple decision tree. At each node (circle) a splitting rule is applied. For example at the root node a feature that either can be "YES" or "NO" determines the split.

2.3.2 Random forest

Random forests is an ensemble learning method. The algorithm was first proposed by Ho (1995). Multiple weak learners are combined to form a final model. The weak learners are in the case of a random forest decision trees, but a weak learner is simply a model that performs better than chance. A random forest is essentially a forest of decision trees trained on randomly sampled training data. The main idea is that by combining multiple decision trees better predictive performance can be obtained than from any of the decision trees alone. The random forest algorithm can be used both for classification and regression tasks.

The random forest training algorithm is based on bagging, which is short for bootstrap aggregating. Each decision tree is trained on a data set randomly sampled from the training data with replacement. Such an approach means that each tree learner is shown a different subset of the training data and that same observation can be chosen more than once in a sample (Breiman, 1996).

Typically, deep decision trees have very high variance and low bias, which means that they tend to overfit. However, the variance can be reduced by including many trees trained on different data sets in the model. This is what is done when building a random forest model. Each decision tree is trained on a subset of the data points in the training set and the final prediction is decided by a vote. This way the variance is decreased at the expense of a small increase in bias.

The performance of an ensemble is typically improved if the ensemble is built up by a diverse and uncorrelated set of weak learners; the submodels should in a sense com-

plement each other. Several identical models are no better than one and in the random forest algorithm a diverse ensembles is achieved by showing different training sets to the different submodels (or decision trees). However, further diversification is achieved by introduction a randomness to the node splitting. In contrast to previously described approach for decision trees only a random subset of the features are considered at each node when searching for the best split.

2.3.3 Gradient boosting

Gradient boosting is another very powerful classification algorithm. The idea behind the algorithm is to sequentially construct models that improve performance on data points where previous models performed badly (Friedman, 2001). The shortcomings of previous models are found using gradient descent. The residuals $y - F_m$ corresponds to the negative gradient of the squared loss function $\frac{1}{2}(y - F)^2$. In the expressions F is a weak learner and y the actual output.

Just like random forest the gradient boosting algorithm is an ensemble method. However, the model building process is very different. Random forest could be seen as a parallel method where several weak learners (decision trees) are trained independently on different subsets of the training data, gradient boosting on the other hand, is sequential and the final model is formed by iteratively adding models. Starting from a weak learner and some loss function the final model is built incrementally by forming a weighted sum of models, where the models built later in the series compensate for previous models' shortcomings.

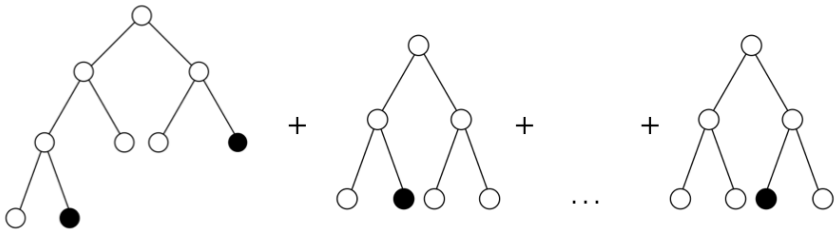

Figure 2.4: A schematic figure of the model building process. Weak learners compensating for previous models shortcomings are sequentially summed to finally form an ensemble of models.

2.3.4 Artificial neural networks

An artificial neural network is a computational model loosely inspired by biological nervous systems. It is basically a network structure built up by a large number of intercon-

nected processing units. Artificial neural network are often simply referred to as neural networks. A good overview of neural networks was presented by LeCun et al. in (2015). Nielsen (2015) has pedagogically described the technique in detail.

The neurons or nodes of the network can simply be seen information-processing units. In neural networks these processing units are typically arranged in layers. Signals are passed to the input layers and then they travel through the network to the output layer. Layers between the input and output layers are called hidden layers.

Neural networks can be seen as very complex functions that for a given input X generates an output Y. The parameters of the function are called weights. The number of layers defines the depth of the network and the number of nodes in a layer is the width of that layer. In addition to these parameters there are so called weights of each interconnection and activation functions that maps the weighted input signal to an output activation. One of the most important features of a neural network is the non-linearity (Bishop, 2006). Neural networks can approximate non-linear relations that typically cannot be reproduced by linear models. The non-linearity is introduced by non-linear activation functions.

Given a set of inputs $x_1, x_2, \ldots, x_N$ a weighted sum of the inputs is formed as

$$a_j = \sum_i^N w_{i,j} \cdot x_i + b_j \tag{2.3}$$

where $w_{i,j}$ are the network weights, x_i the inputs and b_j the bias term. Here a_j is the activation, which is the signal that activates the neuron. From the activation the output is computed by applying the activation function to the activation.

2.3.4.1 Feed forward neural networks

The feed forward neural network (FFNN) is the most simple type. The network has only forward pointing connections and the nodes are structured in layers. In Figure 2.5 a schematic drawing of interconnected nodes in a FFNN is shown.

2.3.4.2 Activation functions

The activation function is what makes neural networks to non-linear classifiers. Until recently the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.4}$$

and the hyperbolic tangent function

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1, \tag{2.5}$$

have been the most commonly used activation functions. The most important difference between these two functions is the output range. As can bee seen in Figure 2.6 the sigmoid

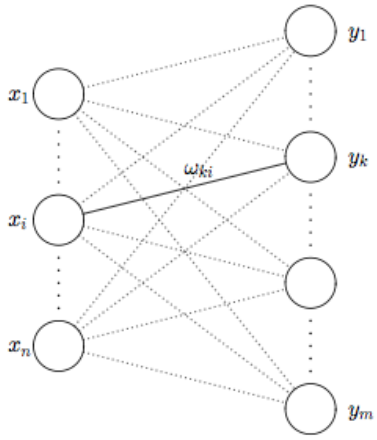

Figure 2.5: A schematic drawing of a feed forward neural network. Two fully connected layers x and y are displayed. In the drawing w denotes the network weights.

function returns a value in the range [0,1] and the hyperbolic tangent function a value in the range [-1,1]. Lately, another activation function called the rectifier is mostly used. The rectifier is in the context of neural networks defined by

$$g(z) = \max\{0,z\} \tag{2.6}$$

and has range $[0,\infty]$ (Hahnloser et al., 2000). A unit in a neural network implementing the rectifier is called a rectified linear unit (ReLU) and for this reason the activation type often is referred to as simply ReLu. In Figure 2.6 the three mentioned activation functions are drawn in the interval [-5,5].

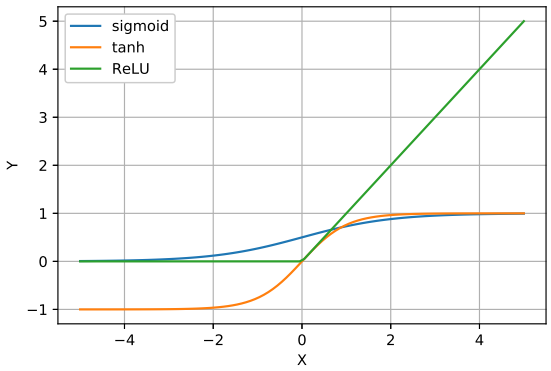

Figure 2.6: The graph shows the sigmoid, hyperbolic tangent and the rectifier drawn in the interval [-5,5].

The major advantage of the rectifier compared to both the sigmoid and hyperbolic tangent functions is that it overcomes the problem of vanishing gradients. The gradients

of these functions quickly vanishes as x grows. The gradient of the rectifier does not do this:

$$\frac{\mathrm{d}}{\mathrm{d}x}g(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{2.7}$$

2.3.4.3 Stochastic gradient descent

Training a neural network means finding the set of network weights that best reproduces a given set of data. There are several algorithms that can be used to solve this optimisation problem. Stochastic gradient descent is one of the most popular and successful techniques (Bottou, 1991).

Gradient descent is an optimisation algorithm where an optimum is found by incrementally taking steps in the direction of the negative gradient of the cost function. One gradient descent step is described by:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}^{(\tau)}, \mathbf{d}), \tag{2.8}$$

where η denotes the step length (sometimes referred to as learning rate), E the cost function, $\mathbf{w}$ i this case the set of weights and $\mathbf{d}$ the data used to calculate the cost and τ defines the current time-step.

In the gradient descent algorithm the cost function is computed for the complete training set. Hence, the weights will only be updated once for every one pass over the complete training set. When working with very large data set such a method becomes very slow and inefficient. Stochastic gradient descent is a method that seeks to solve this problem. It is a stochastic approximation of the gradient descent method that is used to reduce the required number of mathematical operations to optimise the model. The method reduces the computational cost of each update by approximating the gradient from a stochastic sample subset. This way the weights are updated more frequently. Instead of one update per epoch there is one update per batch (a subset of examples taken from the data set). This efficiency is fundamental in large-scale machine learning problems. The algorithm can be summarised as:

Algorithm 1. *Stochastic gradient descent*

1. *Input to algorithm: data set D, cost function E, starting weights $\mathbf{w}^{(0)}$, learning rate η, max number epochs I, number of batches J*
2. *Set $\tau \leftarrow 0$*
3. ***for** each epoch i of a total of I:*
   (a) *Randomly divide the training data D into J batches: $d_1, d_2, \ldots, d_J$*
   (b) ***for** each batch of data d_j:*
      i. *Calculate the cost function $E(\mathbf{w}^{(\tau)}, d_j)$*
      ii. *Update the weights according to $\mathbf{w}^{(\tau+1)} \leftarrow \mathbf{w}^{(\tau)} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}^{(\tau)}, d_j)$*

iii. Update the time step $\tau \leftarrow \tau + 1$

4. *Return the trained neural network with parameters* $\mathbf{w}^{(\tau)}$

2.3.4.4 Weight initialisation

Before starting the training process the set of weights needs to be given initial values. There are several different methods for this, but what almost all have in common is that the weights are initialised at random. It is very important to break the symmetry, which is done effectively by assigning initial values at random.

To illustrate the importance of the initialisation image that all weights are initialised to the same value. At the first hidden layer the signal is

$$a_i = \sum_{i}^{N} w_{i,j} \cdot x_i + b_j \tag{2.9}$$

If then all weights, $w_{i,j}$ have the same value then all units will get the exact same input signal. The activation will be the same for all units independent of the input. A network where all units are the same will not be able to learn anything.

A common method is to draw the weights from a normal distribution with zero mean and standard deviation of 0.1 or 0.01. The final result should not be dependent on the starting values, however, the time it takes to reach convergence may be affected. The biases are often initialised to 0 or a small value. [2]

2.3.4.5 Regularisation

Neural networks can consist of a huge amount of parameters. This imposes the danger that the neural network uses these parameters to overfit the data, as discussed in Section 2.1.3. In that section we discussed how to detect overfitting. In this section we will briefly present two techniques that are widely used to reduce the risk of overfitting.

The first one is called weight decay (Bishop, 2006). To the cost function $E(\mathbf{w}^{(\tau)}, \mathbf{d})$ we add a second term which is called a regularisation term $E_{l2} = \frac{\beta}{2}(\mathbf{w}^{(\tau)})^2$, where β is a constant that can be decided by the user. This cost term penalises large values of the weights $\mathbf{w}^{(\tau)}$. In total the cost function now becomes $E_T = E(\mathbf{w}^{(\tau)}, \mathbf{d}) + \frac{\beta}{2}(\mathbf{w}^{(\tau)})^2$. It is now this cost that we differentiate in the backpropagation step (described in Section 2.3.4.3).

A second method used to avoid overfitting is dropout (Hinton et al., 2012). Dropout is applied to layers in neural networks. For each training step all the nodes in a layer is multiplied by a random variable D_p (drawn independently for each node) that is defined

[2] For more advanced neural networks Glorot and Bengio (2010) have suggested a widely used methods for weight initialising.

as:

$$D_p = \begin{cases} 1/p & \text{with probability } p, \\ 0 & \text{with probability } 1-p. \end{cases} \tag{2.10}$$

p is the probability of keeping each node in the network.

Both these methods prevents any particular node in the network to have too much impact on the prediction. The weight decay do this by preventing any of the weights to be too large, and the dropout achieves the same result by preventing the output to rely on single inputs. Thus the network distributes the responsibility of the prediction between several nodes, which prevents the network to fit the noise of the input.

2.3.4.6 Oversampling

Text-book examples of neural networks often makes many assumptions about the data that does not hold in most real-world applications. One such assumption is that there exists more or less equally many examples in the two classes. For the data set that was used in this thesis this was not the case. This particular data set contains much more negative samples (companies that survive the time period) than positive examples (companies that go bankrupt), which leads to that each batch that is fed to the algorithm (as described in Section 2.3.4.3) contains more negative than positive examples. One consequence of this can be that the model only learns the distribution of the two classes in the data set, and not the patterns in the data which it is supposed to learn.

To avoid this, one technique that can be used is oversampling of the rare class, which means that at each training time step e.g. equally many examples of the two classes are shown to the classifier. This makes the algorithm focus on the patterns in the data without being distracted by the uneven class distribution.

2.4 Data pre-processing

An important part of machine learning is data pre-processing. It can in many cases be an even more challenging task than building the machine learning model itself. In this thesis two data pre-processing techniques were used: normalisation of the data and missing value imputation.

2.4.1 Normalising

Normalisation of the data is done by for each feature x_i replace it by y_i calculated as:

$$y_i = \frac{x_i - \bar{x}}{s}, \tag{2.11}$$

where $\bar{x}$ and s are the estimated mean and standard deviation computed on the training data set.

2.4.2 Missing value imputation

Few real-world data set are complete. Most of them contain missing data points. Missing values are problematic and how to best handle these could easily be the topic of a thesis in itself. In machine learning the most important consideration is that when dealing with these missing values we do not want to loose any important information. The imputation can therefore typically be divided into two separate processes. Firstly, a method is needed to replace the missing value with a valid value (Donders et al., 2006). Otherwise the algorithm does not know how to handle the observations with missing values. Secondly, a process can optionally be used to give the algorithm the information about which values were missing. For example a new feature representing the number of features that were missing in an observation could be added. How we dealt with missing values in this study is discussed in Section 4.1.2. Arial Arial Arial

$$testtext \tag{2.12}$$

3

Data

The data set used in this study was extracted and first used by Zięba et al. (2016)[1]. In this chapter we will describe this data set.

3.1 Data set description

The data set used in this study consisted of financial information about polish companies in the manufacturing sector. The data set contained information about both bankrupt companies and still operating ones. Many polish companies in the manufacturing sector went bankrupt in this sector after 2004 (Zięba et al., 2016).

The financial information were extracted from the database Emerging Markets Information Service (EMIS) (Zięba et al., 2016). The financial indicators describing the health of the bankrupt companies were collected during 2007-2013 and the information about the still operating companies was gathered between 2000-2012. For simplicity, we will refer to the companies that went bankrupt as to belonging to *Class 1* and the surviving companies as to belonging to *Class 0*.

The data is divided into five different subsets, which are described in Table 3.1. The subsets are created to allow for different lengths of forecasting period. The task in each of them is to predict whether or not a company goes bankrupt within five, four, three, two and one years respectively based on information that could retrieved from 64 different financial indicators (regressor variables or features). All five data sets are, as is expected also in real life, heavily imbalanced. There are much fewer bankrupt companies compared to still operating ones.

64 different financial indicators are used to describe the financial status of the companies. Almost all of the features are financial ratios constructed from information found in the companies' economic statements. The use of ratios should mean that the predictors are not too heavily correlated with the size of the companies. A complete list of the features is found in Table 3.2.

[1]The data is available online on:
https://archive.ics.uci.edu/ml/datasets/Polish+companies+bankruptcy+data

Table 3.1: Description of the data pertaining to the five different classification tasks present in this data set.

Data set	Features from	Bankruptcy after	No. bankrupt	No. not bankrupt	Sum
1stYear	1st year	5 years	271	6,756	7,027
2ndYear	2nd year	4 years	400	9,773	10,173
3rdYear	3rd year	3 years	495	10,008	10,503
4thYear	4th year	2 years	515	9,277	9,792
5thYear	5th year	1 years	410	5,500	5,910

3.2 Missing values

As in many real world data set some observations had missing data values for some of the variables. The missing value frequencies in the different task are relatively low about 1-3%. As can be seen in Figure 3.1 it is also rare that observations have large numbers of missing values. Thus, at a fist look it does not seem that very much attention needs to be paid to the missing values. However, when looking closer some problematic facts are revealed. Below some important aspects of the data set are discussed. There is no discernible difference between the distribution of missing values between the different classification tasks. For this reason we will here only present graphs and numbers from one classification task, namely task "1stYear".

Firstly, there is a difference in how the missing values are distributed between the different features. The features X11, X21, X27 and X37 have the highest missing value frequency. These four features represent (X11) the sum of gross profit, extraordinary items, financial expenses divided by total assets, (X21) sales current year divided by last year, (X27) profit on operating activities divided by financial expenses and (X37) the difference between current assets and the inventories divided by the long-term liabilities respectively. In Table 3.3 the missing value frequencies for these features are reported for each of the two classes. Figure 3.1 shows histograms of the number of missing values per feature for the remaining features. By combining the information in Table 3.3 and Figure 3.1 it can be understood that the missing value frequencies for X11, X21, X27 and X37 are much higher than for the rest of the features.

The fact that there is a discrepancy in how the missing values are distributed between the two classes is another important aspect of the the data set. In general missing values are more frequent for companies that go bankrupt (2.4%) of than for the companies that survive (1.3%). In Figure 3.2 we can see the difference in distribution of the number of missing values per company. The shapes of the two histograms are fundamentally different. Most still operating companies have zero or one missing value, whereas most of the

Table 3.2: Description of the 64 features available in this data set.

ID	Description	ID	Description
X1	net profit / total assets	X33	operating expenses / short-term liabilities
X2	total liabilities / total assets	X34	operating expenses / total liabilities
X3	working capital / total assets	X35	profit on sales / total assets
X4	current assets / short-term liabilities	X36	total sales / total assets
X5	[(cash + short-term securities + receivables - short-term liabilities) / (operating expenses - depreciation)] * 365	X37	(current assets - inventories) / long-term liabilities
X6	retained earnings / total assets	X38	constant capital / total assets
X7	EBIT / total assets	X39	profit on sales / sales
X8	book value of equity / total liabilities	X40	(current assets - inventory - receivables) / short-term liabilities
X9	sales / total assets	X41	total liabilities / ((profit on operating activities + depreciation) * (12/365))
X10	equity / total assets	X42	profit on operating activities / sales
X11	(gross profit + extraordinary items + financial expenses) / total assets	X43	rotation receivables + inventory turnover in days
X12	gross profit / short-term liabilities	X44	(receivables * 365) / sales
X13	(gross profit + depreciation) / sales	X45	net profit / inventory
X14	(gross profit + interest) / total assets	X46	(current assets - inventory) / short-term liabilities
X15	(total liabilities * 365) / (gross profit + depreciation)	X47	(inventory * 365) / cost of products sold
X16	(gross profit + depreciation) / total liabilities	X48	EBITDA (profit on operating activities - depreciation) / total assets
X17	total assets / total liabilities	X49	EBITDA (profit on operating activities - depreciation) / sales
X18	gross profit / total assets	X50	current assets / total liabilities
X19	gross profit / sales	X51	short-term liabilities / total assets
X20	(inventory * 365) / sales	X52	(short-term liabilities * 365) / cost of products sold)
X21	sales (n) / sales (n-1)	X53	equity / fixed assets
X22	profit on operating activities / total assets	X54	constant capital / fixed assets
X23	net profit / sales	X55	working capital
X24	gross profit (in 3 years) / total assets	X56	(sales - cost of products sold) / sales
X25	(equity - share capital) / total assets	X57	(current assets - inventory - short-term liabilities) / (sales - gross profit - depreciation)
X26	(net profit + depreciation) / total liabilities	X58	total costs /total sales
X27	profit on operating activities / financial expenses	X59	long-term liabilities / equity
X28	working capital / fixed assets	X60	sales / inventory
X29	logarithm of total assets	X61	sales / receivables
X30	(total liabilities - cash) / sales	X62	(short-term liabilities *365) / sales
X31	(gross profit + interest) / sales	X63	sales / short-term liabilities
X32	(current liabilities * 365) / cost of products sold	X64	sales / fixed assets

Table 3.3: Share missing values for four features separated by class (companies that went bankrupt pertain to Class 1). In the table X11 represent the sum of gross profit, extraordinary items, financial expenses divided by total assets, X21 the sales current year divided by last year, X27 the profit on operating activities divided by financial expenses and X37 the difference between current assets and the inventories divided by the long-term liabilities.

Feature	Missing value (Class 0)	Missing value (Class 1)
X11	0.04%	13.3%
X21	22.4%	40.2%
X27	2.8%	44.3%
X37	39.0%	38.7%

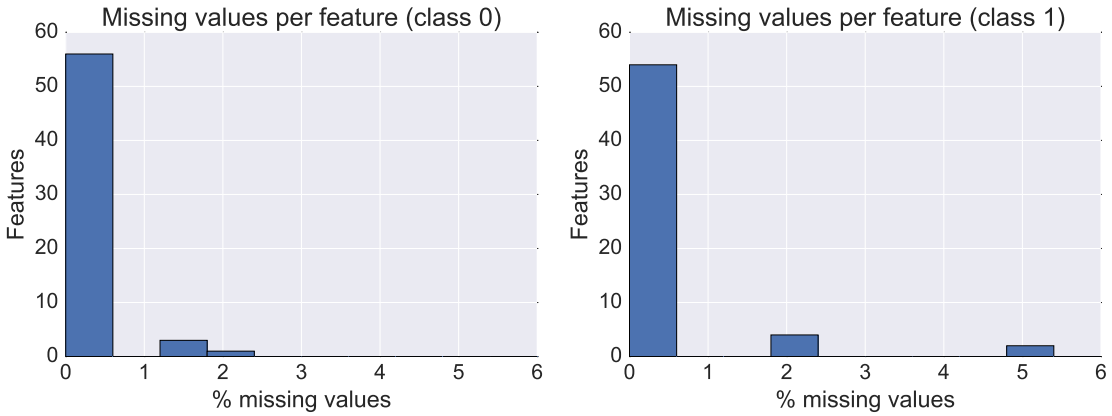

Figure 3.1: Histograms showing the number of missing values per feature for the two classes. Features X11, X21, X27 and X37 have been excluded from the histogram. Class 0 and 1 signifies companies that survived and went bankrupt respectively.

companies that went bankrupt have one or two missing values. Overall the distribution of missing values for companies that went bankrupt differs much from that of the companies that survived. In addition, from Table 3.1 it is clear that the shares of missing values for some features sometimes are significantly different between the two classes.

Since there are such large differences between how the missing values are distributed between the two classes we can conclude that how we leverage this information vastly will affect the performance of the classifier. The implications of the missing values will be discussed in more detail in Section 5.2.

Figure 3.2: Histograms showing the distributions of missing values per company. The distributions have been separated by class, where Class 0 and 1 signifies companies that survived and went bankrupt respectively. Companies with more than 10 missing values were very few and have been excluded from the histogram to improve readability.

4

Methods

This chapter describes the experimental design used to generate the results. Parts of the chapter is of rather technical nature. It could therefore be helpful to read it together with Chapter 2.

4.1 Experimental design

The goal of this study was to investigate how different machine learning techniques can be used for corporate bankruptcy predictions. An experimental bench implemented in the programming language Python was designed. In Appendix A the most important parts of the code is available[1]. The idea of the design was to share infrastructure such as data set splitting, data pre-processing and result generation between the classifiers that was experimented with. This approach let us easily compare different classifiers and reduced the required number of lines of code. In the following sub-sections the parts that were developed will be described.

4.1.1 Splitting up the data

As described in Section 2.1.3 it is important to divide the data set into one training set and one test set to avoid overfitting. This data split was built-in as a part of the experimental bench. The set-up allowed for both single run experiments and cross-validation (described in Section 2.1.3.1). To quickly get feedback about resulting performance when experimenting with different algorithms and hyperparameters (parameters that are set before the training process is initiated) single runs were mostly used, however, when generating the final results, cross-validation was used to boost the results and make them comparable with Zięba et al. (2016).

[1]The code is also available online together with some instructions: `https://github.com/bamattsson/neural-bankruptcy`

4.1.2 Data pre-processing

In Section 2.4 some data pre-processing techniques are described. The effect of these techniques were explored and analysed in order to improve the predictive performance of the algorithms. The multilayer perceptron worked better with normalised data, therefore the data was always normalised before being passed to that algorithm. Normalising the data did not have a big effect on the performance of the random forest and gradient boosting algorithm, however, we still normalised the data before passing it to these algorithms as well.

Missing values showed to be important and the effects of them were examined by creating the possibility to add extra synthetic features describing the nature of the missing values. Adding new features describing the missing values showed to be helpful and was not examined in the study carried out by Zięba et al. (2016). Two strategies were investigated. The first strategy was to add an extra feature representing the total number of missing values in an observation and the second strategy was to add dummy variables for each of the features with the value 1 if the corresponding feature was missing and 0 otherwise.

4.1.3 Classifiers

Random forest and gradient boosting classifiers from the Scikit-learn package (Pedregosa et al., 2011) were used. See Section 2.3.2 and 2.3.3 for an introduction to these algorithms. A wrapper was built around the algorithms to make them fit well into the experimental bench. These two algorithms were used to verify the implementation and ensure that results comparable to the ones presented by Zięba et al. (2016) could be generated. From the comparison it could be concluded that the experimental bench was designed and implemented properly and did not contain any flaws or bugs.

The multilayer perceptron that was used was implemented from scratch using the neural-network library Tensorflow (Abadi et al., 2015). It was designed so that it could take advantage of the techniques discussed in Section 2.3.4.

4.2 Hyperparameter exploration

Hyperparameters values typically have large influence on the performance and the behaviour of a machine learning algorithms. Therefore the hyperparameters of the implemented algorithms were to some extent tuned. For the random forest algorithm 100 trees were used, the quality of the split was maximised with respect to the information gain and no bootstrap samples were used. For the gradient boosting classifier default parameter values were used. 100 trees were used, and cross-entropy was used as objective function.

The multilayer perceptron was implemented in Tensorflow. The most successful implementation used two hidden layers of dimension 100 and 50, respectively. As activation function on hidden nodes the rectified linear unit function was used. Regularisation was a key to make the model work well, which was done by using dropout with a probability of 50% on all hidden units. Besides that, weight decay with the l2-norm on all weights, and a regularisation constant with value 0.001 was used.

All three algorithms had best performance when adding extra feature columns to the data set containing dummy variables describing the missing values.

4.2.1 Evaluation of predictive performance

After the classifier had been trained on the training set the test set was used to evaluate the performance of the classifier. On the test set the the Receiver operating characteristic (ROC) curve as well as the area under the curve (AUC) measure were calculated. This was done for each cross-validation sub-sample. When using cross-validation the individual AUC values obtained were not reported but rather its average and standard error of the mean.

5

Results and discussion

This chapter presents the findings of the project. Firstly, the performance of the three different classification algorithms is compared. Next, the performance of the neural network is discussed in a little more detail. This network significantly outperforms the multilayer perceptron used in (Zięba et al., 2016). Thereafter, an analysis of the importance of missing values in the data set is presented. It is shown that surprisingly good performance can be obtained from data only containing information about the missing values of each observation. Lastly, the usefulness and implications of our findings are discussed.

5.1 Algorithm comparison

In Table 5.1 we see a comparison of the three algorithms that were used in this study. Each row in the matrix corresponds to one algorithm, and each column corresponds to one prediction task, the values that are shown are mean and standard deviation of the AUC score. We can see that the gradient boosting algorithm (GB) performed best, followed by the random forest algorithm (RF) and the multilayer perceptron (MLP). They all get fairly similar results, the largest difference is for "2ndYear" where the average score for the gradient boosting algorithm is 0.07 higher than the score of for the multilayer perceptron. The order in which they perform is equal to what Zięba et al. reported, although they report significantly larger performance differences between the algorithms. The reason for why we get a smaller differences will be discussed in Section 5.1.1.

Table 5.1: AUC results generated from the three different classification methods that have been tested. The algorithms shown are Multilayer perceptron (MLP), Randon forest (RF) and Gradient boosting (GB)

Algorithm	1stYear	2ndYear	3rdYear	4thYear	5thYear
	mean (std)	mean (std)	mean (std)	mean (std)	mean (std)
MLP	0.92 (0.03)	0.83 (0.02)	0.87 (0.04)	0.89 (0.03)	0.91 (0.02)
RF	0.92 (0.04)	0.88 (0.04)	0.89 (0.02)	0.91 (0.02)	0.93 (0.02)
GB	0.94 (0.03)	0.90 (0.02)	0.91 (0.02)	0.92 (0.02)	0.95 (0.01)

It is not surprising that the gradient boosting algorithm comes out on top as this algorithm typically performs well on classification tasks similar to the ones of the study. Decision tree based algorithms are invariant for any scaling of the inputs and therefore such algorithms work well even if the values of the features vary significantly in size as in this data set.

The gradient boosting algorithm achieves comparable or slightly worse result than the gradient boosting algorithm used in (Zięba et al., 2016) on all the five classification tasks. The random forest algorithm achieves slightly worse performance than the gradient boosting algorithm, but it achieves better performance than what was reported for the algorithm in (Zięba et al., 2016). Overall we can see that there indeed is a difference between how the gradient boosting algorithm and random forest algorithm performs for this type of classification.

5.1.1 Performance of multilayer perceptron

As we can see in Table 5.1 the performance of the multilayer perceptron was constantly worse than the performance of the two other methods. The difference was, however, considerably smaller than what was reported in (Zięba et al., 2016). The improvement of the multilayer perceptron is between 0.20 and 0.38. It should of course at this stage be pointed out that their multilayer perceptron only uses one hidden layer, whereas ours use two hidden layers. However, this is far from enough to explain such a big discrepancy in performance.

Generally, good results are not achieved when using out-of-the-box implementation of machine learning algorithms. Some algorithms, like random forest and gradient boosting, tend to be more robust, but neural networks are known to be difficult to tune to good performance. Among other things they require special care and considerations when pre-processing the data and often extensive hyperparameter tuning. This fact makes comparisons between different algorithms hard; the performance is very much influenced by how much time was spent optimising the algorithm. Zięba et al. (2016) achieve best performance the most sophisticated algorithm implementation, a highly optimised and extended version of the gradient boosting algorithm. Most of the other algorithms on the other hand are out-of-the-box implementations taken directly from Weka data mining tool[1], which makes the comparison not completely fair. In this thesis we have instead spent most time fine-tuning the Neural network and used out-of-the-box implementations of the gradient boosting and random forest algorithm. Consequently, this lead us to report results which are rather different from what was reported in by Zięba et al. (2016). One conclusion is therefore that one has to be extremely careful when interpreting compar-

[1] http://www.cs.waikato.ac.nz/ml/weka

isons between different machine learning algorithms since the time spent on optimising the performance cannot be neglected.

However, in the end, the same conclusion as reported by Zięba et al. (2016) persists: tree-based algorithms tend to outperform neural networks on this classification task. This is in-line with the experience from the rest of the research community. Neural networks are good at feature extraction from unstructured data, but when the data is already structured into informative features, the best bet is typically to use a tree-based algorithm.

5.2 Importance of missing values

As we saw in Section 3 there is a large discrepancy in the distribution of missing values between the two different classes. In other words, information about the missing values should be very helpful in the classification. We wanted to explore the effect of the missing values more thoroughly than was done by Zięba et al. (2016).

By imputeing missing values as a very large or small values, i.e. placing the missing values very far away from all the other values, tree-based methods can easily leverage the information. It is more difficult for the multilayer perceptron to leverage this information, therefore we created new synthetic features as described in Section 4.1.2. In Table 5.2 we can see how the results differ for the multilayer perceptron when we have no such synthetic features added (*None*), when the total number of missing values is added as a synthetic feature (*Sum*) and finally when dummy variables for each possible missing value are added as synthetic features (*1-hot*). In the table we can see that the neural network performs much better when it can base its predictions on the synthetic features. For "1stYear"-task the neural network reaches an average score of 0.77 when no synthetic features are added, which can be compared to an average score of 0.92 when dummy variable encoding of the missing values is added. We can thus conclude that much information about the class labels of the examples in the data set is contained in the synthetic features, i.e. the missing values.

To further explore the importance of the missing values in the data set we also tried to do prediction only based on information about the missing values in the data. Before passing the data to the algorithm we threw away all feature columns except the ones which had been synthetically generated from the missing values. In other words, no financial information was used. In Table 5.3 we can see the results from this analysis with the random forest algorithm used as classifier. Even with only synthetic features we get good performance. With only information about the missing values good prediction of whether a company will go bankrupt can be made. In Figure 5.1 the ROC curves of three different classifiers trained are shown. The classifiers are trained with (1) all the data and one-hot encoding of missing values, (2) only one-hot encoding of missing values and (3) single

Table 5.2: AUC for the multilayer perceptron with different synthetic features created from the missing values. *None* signifies no synthetic features, *sum* corresponds to one synthetic feature representing the total number of missing values and *1-hot* a dummy variable representation of the missing values.

Synthetic feature	1stYear mean (std)	2ndYear mean (std)	3rdYear mean (std)	4thYear mean (std)	5thYear mean (std)
None	0.77 (0.02)	0.72 (0.02)	0.78 (0.03)	0.77 (0.02)	0.84 (0.03)
Sum	0.83 (0.03)	0.76 (0.02)	0.82 (0.03)	0.83 (0.03)	0.87 (0.02)
1-hot	0.92 (0.03)	0.83 (0.02)	0.87 (0.04)	0.89 (0.03)	0.91 (0.02)

Table 5.3: Comparison between performance of the random forest algorithm depending on what features were used. *Standard & 1-hot* means that both the original features and the one-hot encoding of missing values were used, *1-hot* signifies that only one-hot encoding of missing value was provided and *sum* means that only the total number of missing values in the observation was used as feature.

Features	1stYear mean (std)	2ndYear mean (std)	3rdYear mean (std)	4thYear mean (std)	5thYear mean (std)
Standard & 1-hot	0.92 (0.04)	0.88 (0.04)	0.89 (0.02)	0.91 (0.02)	0.93 (0.02)
Only 1-hot	0.85 (0.06)	0.73 (0.06)	0.74 (0.05)	0.76 (0.05)	0.76 (0.03)
Only sum	0.72 (0.05)	0.64 (0.06)	0.65 (0.04)	0.67 (0.03)	0.69 (0.04)

predictor describing the total number of missing values in each observation. The ROC curves (from which AUC scores are calculated) paints a similar picture, the algorithm performs well even with only access to the synthetic features.

The performance of the algorithm on data only containing information about what features that were missing is alarming. The result suggest that one can determine the health of a company solely based on what financial information that is available. To some extent this can be logical, the pressure of financial distress may cause firms to fail to meet reporting standards. If there is a relation between what information that is available and the type of firm (small/medium/large etc.) a possible explanation would be that some types of firms are more likely to face bankruptcy than others. Such a relation would need to be very strong if to explain the performance on the data set of only missing value information.

Another more problematic reason that one inevitably comes to think of is the fact that the information about bankrupt and successful companies were collected differently (described in Section 3). It is tempting to think that this might be the reason for the

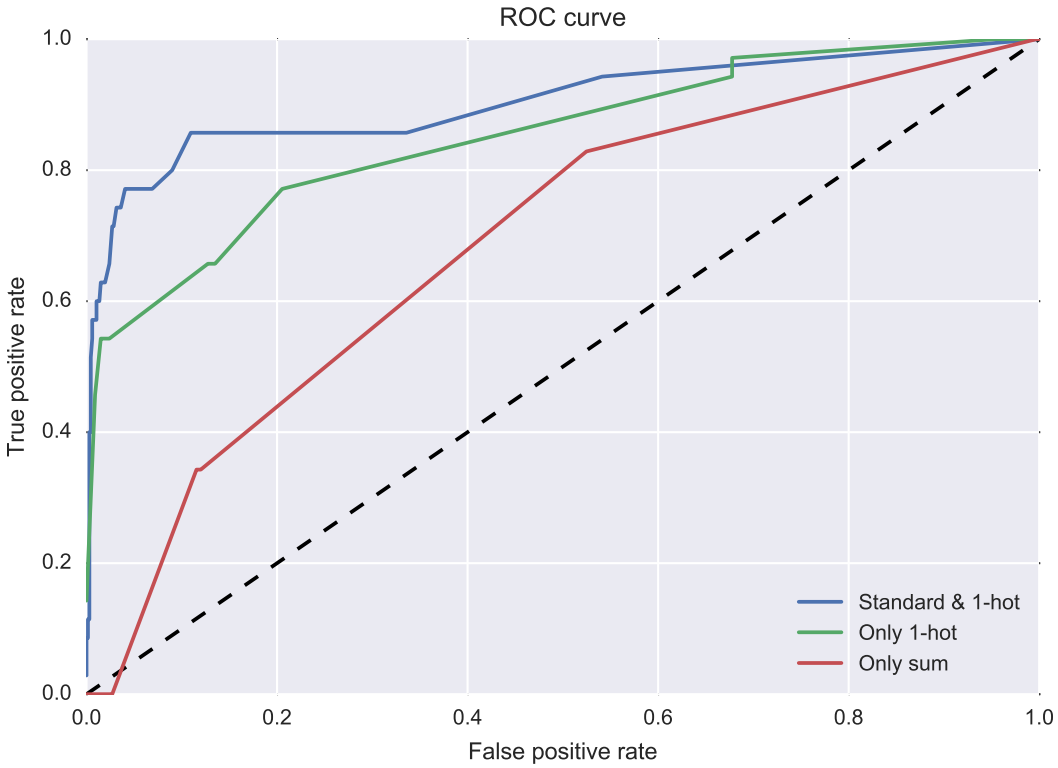

Figure 5.1: The figure shows the ROC curves for three different classifiers trained on (1) all the data and one-hot encoding of missing values, (2) only one-hot encoding of missing values and (3) single predictor describing the total number of missing values in each observation.

predictive power of the missing values. The missing values might indicate from where and when the data was collected and in this way give away information of whether or not the firm was successful. In such a case the classification problem is reduced to predicting where the data comes from rather than whether or not a bankruptcy is imminent. In any case the missing values are not missing at random and any imputation by removing features or examples with random values would destroy information.

5.3 Industry usage

As described in the first chapter the predictive power of a model is not the only thing that is of interest for an economist. From a scientific point of view it is often more interesting if a model can "explain" underlying reasons for behaviours observed in the data. This is most often difficult to do with machine learning techniques. Neural networks in particular are very difficult to interpret. Tree-based methods are better for this purpose since they can be used to analyse the importance of different features in the prediction and this can

give valuable insights in the data.

From a more pragmatical point of view of a bank that lends money to companies what is most interesting is the quality of the predictions. However, rather than choosing an algorithm with low area under curve value they care about choosing an algorithm that can identify as many of the companies that are likely to go bankrupt as possible so that they can avoid lending money to those companies. However, they do not want the algorithm to give to many false warnings either, since they then would miss business opportunities. To facilitate this decision the receiver operating characteristic presented in Section 2.2.4 can be used. In Figure 5.2 we see one of the receiver operating characteristic curves generated in this thesis and with what probability threshold values the different points are reached.

Lets assume that the bank that is planning to use this classifier wants to detect 90% of the companies that will go bankrupt they will have to use a probability threshold of around 30%. In other words, all the companies which the model is more than 30% certain of will file for bankruptcy will be marked by the model. We can also from this graph see that this will lead to a false positive rate slightly higher than 0.2. This means that the model also marks 20% of the companies that actually will survive.

Lets now instead assume that a public organisation that helps companies that are on the brink of bankruptcy want to use this classifier. Since the organisation has limited resources they cannot afford spending time and money on companies that would survive even without their help. Therefore they would, instead of aiming for a high true positive rate favour a low false positive rate. If they want only maximum 10% of the companies they help to be able to survive without their help we can from Figure 5.2 see that they would have to choose a probability threshold of around 60%. This results in a true positive rate of around 70%, i.e. they miss 30% of the companies that goes bankrupt.

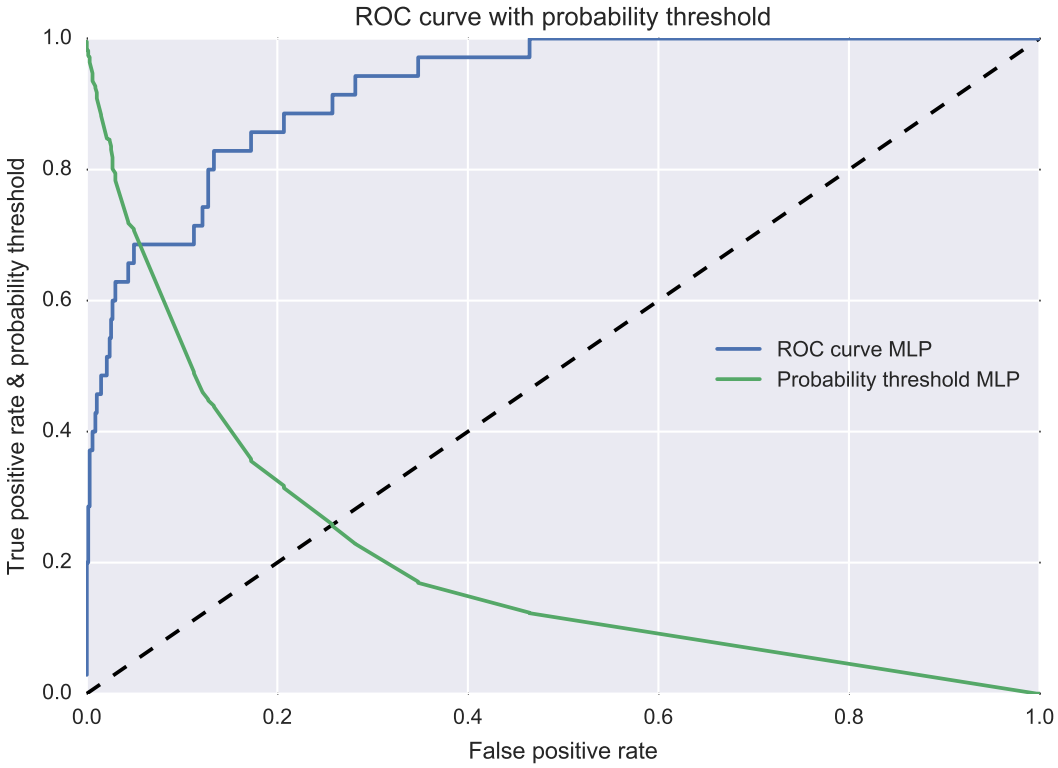

Figure 5.2: The figure shows the ROC curve of the multilayer perceptron on the "1stYear" data set. It also shows for what threshold the different points on the curve are reached.

6

Conclusion

The intention of the study was to illustrate how machine learning can be exploited in the field of economics. To summarise the thesis corporate bankruptcy prediction of polish companies between 2000 and 2013 has been studied. Three different classifications algorithm have been tested and compared. The predictive performance of the random forest and gradient boosting model is in line with the results that Zięba et al. (2016) previously have reported and the performance of our implemented neural network significantly outperforms the reported results for the neural network based model.

The predictive performance achieved on the data set is very good. One reason for the superb performance is the information contained in the missing values. Models based on information about the missing values of the data with the financial information left out could be trained to a surprisingly good performance. Possibly this could suggest a connection between financial reporting and the risk of bankruptcy, however, it could also be the case that the distribution of missing values between the successful and unsuccessful companies differs due to the fact that the financial information was collected from two different distributions.

Future interesting research topics in the development of algorithms for bankruptcy predictions could be to incorporate new types of data. For example unstructured data, such as texts in annual reports, could be of interest. For this purpose neural networks could add much value, as they are good at leveraging this type of data. Apply described algorithms to another data set would also be of interest to better understand the validity of the results. Especially since the importance of missing values in the data set introduced by Zięba et al. (2016) casts doubt on its validity.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Esteban Alfaro, Noelia García, Matías Gámez, and David Elizondo. Bankruptcy forecasting: An empirical comparison of adaboost and neural networks. *Decision Support Systems*, 45(1):110–122, 2008.

Edward I Altman. Financial ratios, discriminant analysis and the prediction of corporate bankruptcy. *The journal of finance*, 23(4):589–609, 1968.

Edward I Altman. A further empirical investigation of the bankruptcy cost question. *The Journal of Finance*, 39(4):1067–1089, 1984.

Dave Anderson and George McNeill. Artificial neural networks technology. *Kaman Sciences Corporation*, 258(6):1–83, 1992.

Sylvain Arlot, Alain Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010.

William H Beaver. Financial ratios as predictors of failure. *Journal of accounting research*, pages 71–111, 1966.

Victor M Becerra, Roberto KH Galvão, and Magda Abou-Seada. Neural and wavelet network models for financial distress classification. *Data Mining and Knowledge Discovery*, 11(1):35–55, 2005.

Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

Hendrik Blockeel and Jan Struyf. Efficient algorithms for decision tree cross-validation. *Journal of Machine Learning Research*, 3(Dec):621–650, 2002.

Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8), 1991.

Xavier Bredart. Bankruptcy prediction model using neural networks. *Accounting and Finance Research*, 3(2):124, 2014.

Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

M Ali Choudhary and Adnan Haider. Neural network models for inflation forecasting: an appraisal. *Applied Economics*, 44(20):2631–2635, 2012.

Michael P Clements, Philip Hans Franses, and Norman R Swanson. Forecasting economic and financial time-series with non-linear models. *International Journal of Forecasting*, 20(2):169–183, 2004.

Ralph De Haas and Neeltje Van Horen. International shock transmission after the lehman brothers collapse: Evidence from syndicated lending. *The American Economic Review*, 102(3):231–237, 2012.

A Rogier T Donders, Geert JMG van der Heijden, Theo Stijnen, and Karel GM Moons. A gentle introduction to imputation of missing values. *Journal of clinical epidemiology*, 59(10):1087–1091, 2006.

Stefan Engström. Kan nyckeltal påvisa framtida betalningsoförmåga? *Balans*, 3:36–45, 2002.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

Sanjeev Gupta and Sachin Kashyap. Forecasting inflation in g-7 countries: an application of artificial neural network. *Foresight*, 17(1):63–73, 2015.

Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, 2000.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Overview of supervised learning. In *The elements of statistical learning*, pages 9–41. Springer, 2009.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL `http://arxiv.org/abs/1207.0580`.

Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995.

Edwin T Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4): 620, 1957.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Chung-Ming Kuan and Halbert White. Artificial neural networks: an econometric perspective. *Econometric reviews*, 13(1):1–91, 1994.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Heikki Linden et al. Synthesis of research studies examining prediction of bankruptcy. 2015.

Paul D McNelis. *Neural networks in finance: gaining predictive edge in the market*. Academic Press, 2005.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518 (7540):529–533, 2015.

Michael A Nielsen. Neural networks and deep learning, 2015.

James A Ohlson. Financial ratios and the probabilistic prediction of bankruptcy. *Journal of accounting research*, pages 109–131, 1980.

P Patrick. A comparison of ratios of successful industrial enterprises with those of failed firms. *Certified Public Accountant*, 2:598–605, 1932.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau,

M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Junfei Qiu, Qihui Wu, Guoru Ding, Yuhua Xu, and Shuo Feng. A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing*, 2016(1):1–16, 2016.

Jaime Salvador, Zoila Ruiz, and Jose Garcia-Rodriguez. Big data infrastructure: A survey. In *International Work-Conference on the Interplay Between Natural and Artificial Computation*, pages 249–258. Springer, 2017.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

Jaymeen R Shah and Mirza B Murtaza. A neural network based clustering procedure for bankruptcy prediction. *American Business Review*, 18(2):80, 2000.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

Tobias Sing, Oliver Sander, Niko Beerenwinkel, and Thomas Lengauer. Rocr: visualizing classifier performance in r. *Bioinformatics*, 21(20):3940–3941, 2005.

Chih-Fong Tsai and Jhen-Wei Wu. Using neural network ensembles for bankruptcy prediction and credit scoring. *Expert systems with applications*, 34(4):2639–2649, 2008.

William Verkooijen. A neural network approach to long-run exchange rate prediction. *Computational Economics*, 9(1):51–65, 1996.

Ulrike Von Luxburg and Bernhard Schölkopf. Statistical learning theory: models, concepts, and results. *arXiv preprint arXiv:0810.4752*, 2008.

Mattias Wahde. *Biologically inspired optimization methods: an introduction*. WIT press, 2008.

S Alex Yang, John R Birge, and Rodney P Parker. The supply chain effects of bankruptcy. *Management Science*, 61(10):2320–2338, 2015.

Maciej Zięba, Sebastian K Tomczak, and Jakub M Tomczak. Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction. *Expert Systems with Applications*, 58:93–101, 2016.

A

Appendix: Programming code

Here we include the most important parts of the code used to generate the results in this thesis. The entire code, together with instructions on how to execute it, can be found online at `https://github.com/bamattsson/neural-bankruptcy`.

Listing A.1: run.py

```
import sys
import os
import time
import pickle
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics, model_selection
from utils import split_dataset, load_dataset, load_yaml_and_save
from algorithms import (RandomGuessAlgorithm, RandomForestAlgorithm,
        GradientBoostingAlgorithm, MultilayerPerceptron)
from data_processors import Imputer, Processor


def main(yaml_path='./config.yml', run_name=None):

    # Create output directory where experiment is saved
    if run_name is None:
        run_name = time.strftime('%Y%m%d-%H%M', time.localtime())
    run_path = os.path.join('./output', run_name)
    if not os.path.exists(run_path):
        os.makedirs(run_path)

    config = load_yaml_and_save(yaml_path, run_path)

    # Do the specified experiments
    np.random.seed(config['experiment']['np_random_seed'])
    for year in config['experiment']['years']:
        do_experiment_for_one_year(run_path, year, config)
    # Show figures if any have been generated
    if config['analysis']['plot_roc']:
        plt.show()


def do_experiment_for_one_year(run_path, year, config):
    """Performs the specified experiments for one year."""
    X, Y = load_dataset(year, shuffle=config['experiment']['shuffle_data'])
    if config['experiment']['type'] == 'single':
        X_train, Y_train, X_test, Y_test = split_dataset(X, Y,
                config['experiment']['test_share'])
        results = perform_one_experiment(X_train, Y_train, X_test, Y_test,
                config)
    elif config['experiment']['type'] == 'cv':
        results = perform_cv_runs(X, Y, config)

    results_path = os.path.join(run_path, 'results_year{}.pkl'.format(year))
```

```
    with open(results_path, 'wb') as f:
        pickle.dump(results, f)
    show_results(results, year, **config['analysis'])


def perform_one_experiment(X_train, Y_train, X_test, Y_test, config):
    """Performs one experiment with a given data set and generates results."""
    # Prepare data
    processor = Processor(**config['processor_params'])
    X_train = processor.fit_transform(X_train)
    X_test = processor.transform(X_test)
    imputer = Imputer(**config['imputer_params'])
    X_train = imputer.fit_transform(X_train)
    X_test = imputer.transform(X_test)

    # Creates the algorithm object
    algorithm_name = config['experiment']['algorithm']
    if algorithm_name == 'random_guess':
        algorithm = RandomGuessAlgorithm(**config['algo_params'])
    elif algorithm_name == 'rf':
        algorithm = RandomForestAlgorithm(**config['algo_params'])
    elif algorithm_name == 'multilayer_perceptron':
        algorithm = MultilayerPerceptron(n_input=X_train.shape[1],
                **config['algo_params'])
    elif algorithm_name == 'gradient_boosting':
        algorithm = GradientBoostingAlgorithm(**config['algo_params'])
    else:
        raise NotImplementedError('Algorithm␣{}␣is␣not␣an␣available␣option'
                .format(algorithm_name))

    # Perform experiment
    results = dict()
    results['fit_info'] = algorithm.fit(X_train, Y_train)
    pred_proba = algorithm.predict_proba(X_test)
    pred = np.argmax(pred_proba, axis=1)

    # Calculate and save results
    results['log_loss'] = metrics.log_loss(Y_test, pred_proba[:, 1])
    results['accuracy'] = metrics.accuracy_score(Y_test, pred)
    results['recall'] = metrics.recall_score(Y_test, pred, labels=[0, 1])
    results['precision'] = metrics.precision_score(Y_test, pred, labels=[0, 1])
    fpr, tpr, thresholds = metrics.roc_curve(Y_test, pred_proba[:, 1])
    results['roc_curve'] = {'fpr': fpr, 'tpr': tpr, 'thresholds': thresholds}
    results['roc_auc'] = metrics.auc(fpr, tpr)
    results['classification_report'] = metrics.classification_report(Y_test,
            pred, labels=[0, 1])
```

```
    return results


def perform_cv_runs(X, Y, config):
    """Performs cv test for one year."""
    # split up in cv and perform runs
    cv_splitter = model_selection.KFold(n_splits=config['experiment']
            ['n_folds'])
    result_list = []
    for train_index, test_index in cv_splitter.split(X):
        X_train, X_test = X[train_index], X[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]
        results_one_experiment = perform_one_experiment(X_train, Y_train,
                X_test, Y_test, config)
        result_list.append(results_one_experiment)

    # Loop through all the generated results and save to one place
    results = dict()
    for metric in result_list[0]:
        # Roc curve comes in a dict and are treated seperately
        if type(result_list[0][metric]) == dict:
            results[metric] = dict()
            for submetric in result_list[0][metric]:
                arrays = [result_from_cv[metric][submetric] for result_from_cv
                        in result_list]
                results[metric][submetric] = arrays
        # Float values are saved as a list of all values but also as mean and std
        elif isinstance(result_list[0][metric], float):
            results[metric] = dict()
            results[metric]['values'] = np.array([result_from_cv[metric] for
                result_from_cv in result_list])
            results[metric]['mean'] = results[metric]['values'].mean()
            results[metric]['std'] = results[metric]['values'].std()
        # Other values are saved in lists
        else:
            value = [result_from_cv[metric] for result_from_cv in result_list]
            results[metric] = value

    return results


def show_results(results, year, print_results=[], plot_roc=False):
    """Print and plot the results that have been generated."""
    # Print results
    if (len(print_results) > 0):
```

```
        print('\nResults for year {}:'.format(year))
        for metric in print_results:
            if type(results[metric]) == dict:
                print('{}={:.2f}, std={:.2f}'.format(metric,
                    results[metric]['mean'], results[metric]['std']))
            elif type(results[metric]) == str:
                print(results[metric])
            elif isinstance(results[metric], float):
                print('{}={:.2f}'.format(metric, results[metric]))
            else:
                print('{} cannot be printed.'.format(metric))

    # Plot results
    if plot_roc:
        plt.figure(year)
        plt.title('roc curve year {}'.format(year))
        plt.plot((0, 1), (0, 1), ls='--', c='k')
        if type(results['roc_curve']['fpr']) == list:
            # A CV run with multiple arrays
            for fpr, tpr in zip(results['roc_curve']['fpr'],
                    results['roc_curve']['tpr']):
                plt.plot(fpr, tpr)
        else:
            # Not a CV run
            plt.plot(results['roc_curve']['fpr'], results['roc_curve']['tpr'])
        plt.xlabel('False positive rate')
        plt.ylabel('True positive rate')


if __name__ == '__main__':
    try:
        yaml_path = sys.argv[1]
    except IndexError as e:
        print('You have to specify the config.yaml to use as `python run.py '
            'example_config.yaml`')
        print('Exiting.')
        sys.exit()
    main(yaml_path=yaml_path)
```

Listing A.2: algorithms/algorithm.py

```
from abc import ABCMeta, abstractmethod
import numpy as np


class Algorithm(metaclass=ABCMeta):
    """Abstract algorithm class."""

    @abstractmethod
    def fit(self, samples, labels):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
            labels (np.ndarray): y data, shape (n_samples)
        Returns:
            fit_info (dict): information from the fit for later analysis
        """
        return None

    @abstractmethod
    def predict_proba(self, samples):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
        Returns:
            proba (np.ndarray): Probability of belonging to a particular class,
                shape (n_samples, n_classes)
        """
        proba = np.zeros((samples.shape[0], 2))
        return proba

    def predict(self, samples):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
        Returns:
            predict (np.ndarray): Predicted class, shape (n_samples)
        """
        predict_proba = self.predict_proba(samples)
        predict = np.argmax(predict_proba, axis=1)
        return predict
```

Listing A.3: algorithms/multilayer_perceptron.py

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from .algorithm import Algorithm
from utils import split_dataset


class MultilayerPerceptron(Algorithm):

    def __init__(self, n_input, n_hidden, dropout_keep_prob, l2_reg_factor,
            dev_share, num_epochs, batch_size, batch_iterator_type,
            evaluate_every_n_steps, plot_training, tf_seed):
        # Structure of model
        self.n_input = n_input
        self.n_hidden = n_hidden
        self.dropout_keep_prob = dropout_keep_prob
        self.l2_reg_factor = l2_reg_factor
        self.n_class = 2
        # Training parameters
        self.dev_share = dev_share
        self.num_epochs = num_epochs
        self.batch_size = batch_size
        self.batch_iterator_type = batch_iterator_type
        self.evaluate_every_n_steps = evaluate_every_n_steps
        self.plot_training = plot_training

        # Create TF graph and session
        tf.reset_default_graph()
        tf.set_random_seed(tf_seed)
        self.graph_nodes = self._get_graph()

        self.sess = tf.Session()
        self.sess.run([tf.global_variables_initializer()])
        # TODO: add early stopping with tf.train.Saver

    def fit(self, samples, labels):
        """Train the model with the samples and lables provided according to
        the parameters of the model."""

        # Split into train and dev
        x_train, y_train, x_dev, y_dev = split_dataset(samples, labels,
                self.dev_share)

        # Create batch iterator
```

```
        if self.batch_iterator_type == 'normal':
            batch_iter = _batch_iter
        elif self.batch_iterator_type == 'oversample':
            batch_iter = _oversampling_batch_iter
        else:
            raise ValueError('{} is not a valid batch_iterator_type'.format(
                self.batch_iterator_type))

        # Train model
        train_batch_nr = []
        train_loss_val = []
        dev_batch_nr = []
        dev_loss_val = []
        for i, (x, y) in enumerate(batch_iter(x_train, y_train,
                self.num_epochs, self.batch_size)):
            # Train
            feed_dict = {
                    self.graph_nodes['x_input']: x,
                    self.graph_nodes['y_input']: y,
                    self.graph_nodes['dropout_keep_prob']:
                            self.dropout_keep_prob
                    }
            _, loss_val = self.sess.run([self.graph_nodes['optimize'],
                self.graph_nodes['loss']], feed_dict=feed_dict)
            train_batch_nr.append(i)
            train_loss_val.append(loss_val)
            if i % self.evaluate_every_n_steps == 0:
                feed_dict = {
                        self.graph_nodes['x_input']: x_dev,
                        self.graph_nodes['y_input']: y_dev,
                        self.graph_nodes['dropout_keep_prob']: 1.
                        }
                loss_val = self.sess.run(self.graph_nodes['loss'],
                        feed_dict=feed_dict)
                dev_batch_nr.append(i)
                dev_loss_val.append(loss_val)

        if self.plot_training:
            plt.plot(train_batch_nr, train_loss_val)
            plt.plot(dev_batch_nr, dev_loss_val)
            plt.show()

    def predict_proba(self, samples):
        """Make probability predictions with the trained model."""
        # Small model -> no need to loop over the samples
        feed_dict = {
```

```
            self.graph_nodes['x_input']: samples,
            self.graph_nodes['dropout_keep_prob']: 1.
            }
        proba = self.sess.run(self.graph_nodes['proba'], feed_dict=feed_dict)
        return proba

    def _get_graph(self):
        # Create placeholders for input and dropout_prob
        x_input = tf.placeholder(tf.float32, shape=(None, self.n_input))
        y_input = tf.placeholder(tf.int32, shape=(None))
        dropout_keep_prob = tf.placeholder(tf.float32)

        # Variables
        # Build the fully connected layers
        neurons = x_input
        l2_norm = tf.constant(0.)
        for i in range(len(self.n_hidden) + 1):
            input_dim = self.n_input if i == 0 else self.n_hidden[i - 1]
            output_dim = self.n_class if i == len(self.n_hidden) \
                    else self.n_hidden[i]
            layer_name = i + 1 if i < len(self.n_hidden) else 'out'
            # Create weights
            W = tf.Variable(tf.truncated_normal([input_dim, output_dim],
                    stddev=0.1), name='W_{}_layer'.format(layer_name))
            b = tf.Variable(0.1 * np.ones(output_dim, dtype=np.float32),
                    name='b_{}_layer'.format(layer_name))
            l2_norm += tf.nn.l2_loss(W)
            # Connect nodes
            neurons = tf.add(tf.matmul(neurons, W), b)
            if i < len(self.n_hidden):  # True if not last (output) layer
                neurons = tf.nn.dropout(neurons, dropout_keep_prob)
                neurons = tf.nn.relu(neurons)  # TODO: make this optional

        logits = neurons
        proba = tf.nn.softmax(logits)

        # Loss and Accuracy
        loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
            labels=y_input, logits=logits))
        regularized_loss = loss + self.l2_reg_factor * l2_norm
        correct_predictions = tf.equal(tf.cast(tf.argmax(logits, 1), tf.int32),
                y_input)
        accuracy = tf.reduce_mean(tf.cast(correct_predictions, 'float'))

        # Train operation
        # TODO: add so that we could change learning rate?
```

```
        optimize = tf.train.AdamOptimizer().minimize(regularized_loss)

        # Save important nodes to dict and return
        graph = {'x_input': x_input,
                 'y_input': y_input,
                 'dropout_keep_prob': dropout_keep_prob,
                 'proba': proba,
                 'loss': loss,
                 'accuracy': accuracy,
                 'optimize': optimize}

        return graph


def _oversampling_batch_iter(samples, labels, num_epochs, batch_size):
    """Batch iterator that oversamples the rare class so that both classes
    become equally frequent."""
    pos_examples = (labels == 1)
    pos_samples, pos_labels = samples[pos_examples], labels[pos_examples]
    neg_examples = np.logical_not(pos_examples)
    neg_samples, neg_labels = samples[neg_examples], labels[neg_examples]

    neg_batch_size = np.floor(batch_size / 2)
    pos_batch_size = np.ceil(batch_size / 2)

    neg_batch_iter = _batch_iter(neg_samples, neg_labels, num_epochs,
            neg_batch_size)
    pos_batch_iter = _batch_iter(pos_samples, pos_labels, num_epochs,
            pos_batch_size)

    for neg_batch, pos_batch in zip(neg_batch_iter, pos_batch_iter):
        neg_batch_samples, neg_batch_labels = neg_batch
        pos_batch_samples, pos_batch_labels = pos_batch
        batch_samples = np.concatenate((neg_batch_samples, pos_batch_samples),
                axis=0)
        batch_labels = np.concatenate((neg_batch_labels, pos_batch_labels),
                axis=0)
        yield batch_samples, batch_labels


def _batch_iter(samples, labels, num_epochs, batch_size):
    """A batch iterator that generates batches from the data."""
    data_size = len(labels)
    batch_num = 0
    while (batch_num) * batch_size // data_size < num_epochs:
        start_index = int(batch_num * batch_size % data_size)
```

```
end_index = int((batch_num + 1) * batch_size % data_size)
if start_index < end_index:
    samples_batch = samples[start_index:end_index]
    labels_batch = labels[start_index:end_index]
else:
    samples_batch = np.concatenate((samples[start_index:],
        samples[:end_index]))
    labels_batch = np.concatenate((labels[start_index:],
        labels[:end_index]))
yield samples_batch, labels_batch
batch_num += 1
```

Listing A.4: algorithms/gradient_boosting.py

```
from .algorithm import Algorithm
from sklearn.ensemble import GradientBoostingClassifier


class GradientBoostingAlgorithm(Algorithm):

    def __init__(self, loss='deviance', learning_rate=0.1, n_estimators=100,
            subsample=1.0, criterion='friedman_mse', min_samples_split=2,
            min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3,
            min_impurity_split=1e-07, init=None, random_state=None,
            max_features=None, verbose=0, max_leaf_nodes=None,
            warm_start=False, presort='auto'):

        self.clf = GradientBoostingClassifier(loss=loss,
                learning_rate=learning_rate, n_estimators=n_estimators,
                subsample=subsample, criterion=criterion,
                min_samples_split=min_samples_split,
                min_samples_leaf=min_samples_leaf,
                min_weight_fraction_leaf=min_weight_fraction_leaf,
                max_depth=max_depth, min_impurity_split=min_impurity_split,
                init=init, random_state=random_state, max_features=max_features,
                verbose=verbose, max_leaf_nodes=max_leaf_nodes,
                warm_start=warm_start, presort=presort)

    def fit(self, samples, labels):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
            labels (np.ndarray): y data, shape (n_samples)
        Returns:
            fit_info (dict): information from the fit for later analysis
        """
        self.clf.fit(samples, labels)

    def predict_proba(self, samples):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
        Returns:
            proba (np.ndarray): Probability of belonging to a particular class,
                shape (n_samples, n_classes)
        """
        return self.clf.predict_proba(samples)
```

Listing A.5: algorithms/random_forest.py

```
from .algorithm import Algorithm
from sklearn.ensemble import RandomForestClassifier


class RandomForestAlgorithm(Algorithm):

    def __init__(self, n_estimators=10, criterion='gini', min_samples_split=2,
            min_samples_leaf=1, min_weight_fraction_leaf=0.0, warm_start=False,
            bootstrap=True, oob_score=False, max_features='auto',
            max_depth=None, max_leaf_nodes=None, min_impurity_split=1e-07,
            class_weight=None):

        self.clf = RandomForestClassifier(n_estimators=n_estimators,
                criterion=criterion, min_samples_split=min_samples_split,
                min_samples_leaf=min_samples_leaf,
                min_weight_fraction_leaf=min_weight_fraction_leaf,
                warm_start=warm_start, bootstrap=bootstrap,
                oob_score=oob_score, max_features=max_features,
                max_depth=max_depth, max_leaf_nodes=max_leaf_nodes,
                min_impurity_split=min_impurity_split,
                class_weight=class_weight)

    def fit(self, samples, labels):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
            labels (np.ndarray): y data, shape (n_samples)
        Returns:
            fit_info (dict): information from the fit for later analysis
        """
        self.clf.fit(samples, labels)

    def predict_proba(self, samples):
        """
        Args:
            samples (np.ndarray): X data, shape (n_samples, n_features)
        Returns:
            proba (np.ndarray): Probability of belonging to a particular class,
                shape (n_samples, n_classes)
        """
        return self.clf.predict_proba(samples)
```

Listing A.6: data_processors.py

```
from abc import ABCMeta, abstractmethod
import numpy as np


class DataProcessor(metaclass=ABCMeta):
    """Abstract data processor class."""

    @abstractmethod
    def fit(self, data):
        """Fits the internal DataProcessor values to the data."""
        raise NotImplementedError

    @abstractmethod
    def transform(self, data):
        """Transforms the data with the DataProcessor."""
        raise NotImplementedError

    def fit_transform(self, data):
        self.fit(data)
        return self.transform(data)


class Imputer(DataProcessor):
    def __init__(self, strategy, new_features=False, only_nan_data=False):
        """Initialize object.

        Args:
            strategy (str): strategy to follow when imputing. Available:
                'mean', 'min'
            new_features (str): whether we should create new features depending
                from the information from missing values. False creates no new
                features, 'sum' creates one new and '1-hot' creates multiple
                new features.
        """
        self.strategy = strategy
        self.new_features = new_features
        self.only_nan_data = only_nan_data
        if (self.new_features == False and self.only_nan_data == True):
            raise ValueError('`new_features` equal to {} and `only_nan_data` '
                'equal to {} is not a valid parameter combination'.format(
                    self.new_features, self.only_nan_data))

    def fit(self, data):
        if self.strategy == 'mean':
            self.imputing_values = np.nanmean(data, axis=0)
```

```
        elif self.strategy == 'min':
            self.imputing_values = np.nanmin(data, axis=0)
        else:
            raise ValueError(
                    '{} is not a valid value for `strategy`'.format(
                        self.strategy))
        if self.new_features == '1-hot':
            self.contains_nan = np.any(np.isnan(data), axis=0)

    def transform(self, data):
        # Add new features from nan values
        if not self.new_features:
            extra_features = np.zeros([len(data), 0])
        elif self.new_features == 'sum':
            extra_features = np.isnan(data).sum(axis=1)[:, None]
        elif self.new_features == '1-hot':
            extra_features = np.isnan(data)[:, self.contains_nan]
            extra_features = np.atleast_2d(extra_features)
        else:
            raise ValueError(
                    '{} is not a valid value for `new_features`'.format(
                        self.new_features))
        # Imputes nan values
        data = np.copy(data)
        for i in range(len(data)):
            isnan = np.isnan(data[i])
            data[i, isnan] = self.imputing_values[isnan]

        if self.only_nan_data:
            out_data = extra_features
        else:
            out_data = np.concatenate((data, extra_features), axis=1)
        return out_data


class Processor(DataProcessor):

    def __init__(self, normalize, max_nan_share):
        self.normalize = normalize
        self.max_nan_share = max_nan_share

    def fit(self, data):
        self.features_to_drop = np.zeros([len(data), 0])
        if self.normalize:
            self.mean = np.nanmean(data, axis=0)
            self.std = np.nanstd(data, axis=0)
```

```
        if self.max_nan_share < 1.0:
            nan_frequency = np.isnan(data).sum(axis=0) / len(data)
            self.features_to_drop = np.where(nan_frequency >
                    self.max_nan_share)[0]

    def transform(self, data):
        data = np.copy(data)
        if self.normalize:
            data = (data - self.mean) / self.std
        if len(self.features_to_drop) > 0:
            data = np.delete(data, self.features_to_drop, axis=1)
        return data
```