Corporate Credit Rating using Deep Learning with Genetic Algorithms

FIE453 - Term Paper

Norwegian School of Economics
Birk Carlenius, Eivind K. Døvik, Johannes K. Kolberg, Kristin Waage & Benjamin Aanes

November 13, 2017

Executive Summary

Corporate credit rating is a complex and expensive process where automation may yield significant benefits. Beyond simple cost reduction, such automation would also give suggestions for ratings from a purely objective perspective compared to the subjectivity of credit rating agencies. In this paper, we therefore investigate the extent to which deep learning can be used to predict corporate credit ratings. As part of designing a neural network, we attempt to leverage a genetic algorithm to generate promising multilayer perceptron architectures for predicting credit ratings using Norwegian Corporate Accounts data. The resulting deep learning model performance is then compared to two benchmarks: simply guessing last year's rating, and a model based on Breiman's Random Forest algorithm (Breiman, 2001). We find that our genetic algorithm converges on a general architecture after 5-7 generations, and beats simple guessing. Manually adjusting the architecture and tuning hyperparameters improves performance further, but this final deep learning model is still dominated by the random forest model. Though we believe this demonstrates that deep learning is a feasible approach to (partially) automating corporate credit ratings, we must conclude that random forest is preferable as it is both much simpler and considerably more accurate.

Contents

Introduction

Credit rating is the analysis of credit risk and addresses an issuer's overall capacity and willingness to meet its financial obligations (Hájek & Olej, 2016). Payment delays, negotiations for reduced payback, and a failure to respect specific clauses can all be reasons for why obligations are not met by the issuer (Balios, Thomadakis, & Tsipouri, 2016). As an inability to pay back will eventually result in the issuer defaulting on its obligations, the credit rating can also be viewed as the default risk of an issuer (Balios et al., 2016; Hájek, 2012).

In this paper, we investigate to what extent deep learning can be used to predict corporate credit ratings. To answer this research question, we compare the performance of a deep learning model to that of a random forest model. According to Nanni and Lumini (2009), random forest can be an efficient technique for credit rating predictions. We also benchmark against the accuracy of guessing last year's credit rating and the credit rating mode. The framework of our deep learning model is a deep feedforward network, or multilayer perceptron (MLP), which is evaluated to be a good method for credit rating (Nanni & Lumini, 2009). Furthermore, to identify the design of the deep learning architectures, we use genetic algorithms.

The paper is organised as follows. The rest of chapter one ('Introduction') discusses benefits of using machine learning to predict credit ratings, in addition to the costs of misclassifications. Chapter two ('Theory') presents the theoretical framework for the random forest and deep learning model. Chapter three ('Data and Variables') describes the data cleaning process and the final dataset used in the analyses. In chapter four ('Benchmark Implementation'), the random forest model is implemented and evaluated, in addition to the benchmarking approaches of simple guessing last year's rating and the rating mode. Then, the deep learning model is implemented in chapter five ('Deep Learning Implementation'), first with manually constructed architectures and second using the genetic algorithm to identify the architecture design. Finally, chapter six ('Conclusion') sums up the paper's main findings and give suggestions for further studies.

Benefits of Machine Learning in Corporate Credit Rating

Credit rating is both a complex and expensive process, first of all because of the many different parameters that must be taken into account when performing the assessment (Hajek & Michalak, 2013; Hájek & Olej, 2016). For example, Bisnode D&B AS includes financial variables, ability to pay, operational data, and information on company background such as ownership type and credit history in their rating model (Bisnode, 2016). In addition to, or because of, the high degree of complexity, ratings must be performed by domain experts which further drives up the cost of assessment (Hajek & Michalak, 2013).

Many studies have been dedicated to exploring potential models for automating the credit rating processes. Due to the complexity and high degree of subjectivity in ratings, this is a challenging task. However, several machine learning methods applied in other studies have yielded promising

results. So, why is this an interesting topic to study? What are some of the benefits that can be achieved by automating credit ratings through machine learning?

First, we believe there is a potential for improvements in quality. Machine learning algorithms can train on extensive amounts of quantitative and qualitative data, and possibly discover patterns and risk factors that the human brain might overlook when performing ratings manually. In addition, employing machine learning algorithms will ensure objective ratings. Variation in credit ratings can for example be explained by differences in the optimism of the analysts performing the ratings (Fracassi, Petry, & Tate, 2016). Still, the argument that appears to be the most frequent in literature, is that of cost and time savings. The value of extensive amounts of collected data and specialised knowledge possessed by rating companies can justify the high costs of performing ratings (Balios et al., 2016; Hájek & Olej, 2011). However, the time-consuming and expensive nature of the process is such that ratings can mainly be afforded by larger and more established companies, while smaller companies "are simply below the radar of credit rating" (Balios et al., 2016). This can create a discriminatory and self-reinforcing situation where companies that can afford ratings, can more easily gain access to financing – and thus have better prerequisites for improving ratings and receiving even better access to financing. Smaller companies in great need of financing, on the other hand, can be left out of this self-reinforcing loop.

The use of machine learning techniques to perform credit ratings can bring down the costs and time requirements of the process significantly, making ratings more affordable and accessible for all types of companies. This has benefits not only on the microeconomic, but also the macroeconomic level; as pointed out in Balios et al. (2016), extending the proportion of companies with access to ratings can yield benefits in terms of better credit decisions and more stable financial institutions, which again result in more efficient allocation of both private and public resources – and ultimately higher potentials for economic growth.

The Cost of Misclassification

Credit ratings provide investors and lenders with crucial information regarding the risk associated with an investment in, or loan to, corporations. This information is further reflected in e.g. expected returns and interest rates, and plays an important role in the allocation of capital. Incorrect ratings thus lead to less efficient markets (Terovitis, 2017).

When assessing the performance of our models, we therefore consider the misclassification rate. That is, the extent to which our models predict a rating other than that given by Bisnode D&BNorway AS. In that regard, it is important to be aware that not all classification errors are of the same importance. As pointed out by Hajek and Michalak (2013), , classifying a company belonging to a bad credit class (e.g. rating 'C') as a company with strong creditworthiness ('AAA' rating) is a more serious mistake than giving the rating 'AAA' to a company that in reality belongs to the 'AA' rating class. In general, classifying a company that is not creditworthy

as being creditworthy is the type of mistake our models must avoid to as large an extent as possible. In addition to evaluating the misclassification rate, we will therefore also evaluate to what extent our models can correctly distinguish between the two categories **creditworthy** and **not creditworthy**.

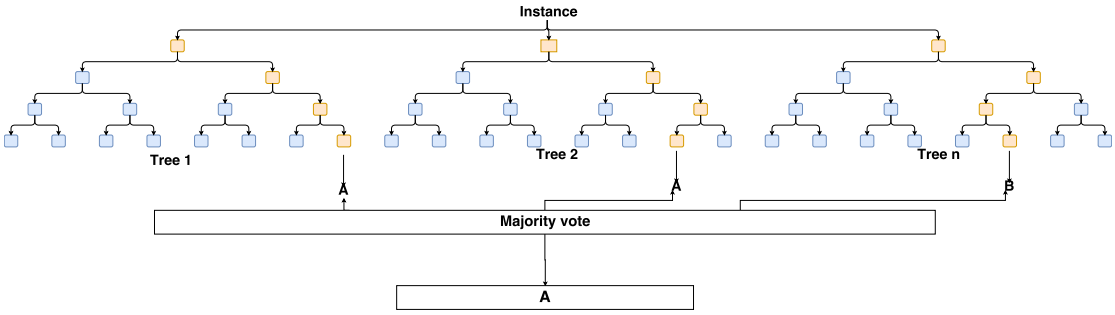
Theory

While this paper's main focus is on developing and applying a deep learning model to predict credit ratings, we develop and apply a random forest model as a benchmark. This chapter starts by giving a brief explanation of how random forest models work, before providing a comprehensive explanation of the theory behind deep learning models.

Random Forest

Tree-based models for classification involve taking the predictor space and stratifying it into smaller regions. The rules behind the splitting process can then be summarised in a tree (Hastie & Tibshirani, 2014). Random forest models grow multiple trees; that is, it is an ensemble method. The ensemble of trees are then combined and each tree cast their vote in predicting the outcome. The consensus prediction is then decided by simple majority-rule (see simplified illustration in figure 1).

Figure 1: Random Forest Simplified



In random forest models, the tuning parameter is the parameter `mtry`. That is, the best `mtry` is the `mtry` that gives the highest prediction accuracy. Random forest models grow multiple trees, and each tree has many splits. At each split, random forest picks a number m randomly selected predictors from the predictor space p (hence $m \leq p$). This is different from tree bagging models, where all predictors are included at each split. Selecting a number of random predictors at each split decorrelates the trees, improving the prediction error when averaging the ensemble of trees (Hastie & Tibshirani, 2014).

Deep Learning Theory: Deep Feedforward Neural Networks

This section explains the form and function of deep feedforward networks, cost functions, output functions, activiation functions, the architecture of the networks, the concept of backpropagation, and the regularization technique dropout.

Form and Function

Deep feedforward network, or multilayer perceptrons (MLPs), are frameworks for one of the most commonly used deep learning models, well known for robustness and core simplicity (Goodfellow, Bengio, & Courville, 2016). The object of the model is to approximate a function f^* by mapping $\upsilon = f(\chi; \theta)$, and learning the optimal χ. Characteristic for the feedforward network is the no-feedback connections between the linked functions. The model can be depicted as flow of data in one direction; the links between nodes are one-way roads and the layer structure prevents outputs from being sent backwards to be used as input for earlier functions (layers).

The layers define the architectural design of the model, each expressed by interlinked functions connected in chain. An object function $f(\chi)$ approximating f^* can consists of infinite amount of layers $f(2)...f(n)$ so that $f(\chi) = f_n \dots (f_3(f_2(f_1(\chi))))$. Each function $f(n)$ represents a layer, and the design of the functions are closely linked to the form of the object function; linear output functions which attempts to approximate non-linear functions need at least one non-linear layer or activation function (Goodfellow et al., 2016).

The behavior of the output function is clearly defined as an approximation to a function f^*, while the other functions' behaviors are more prone to cause confusion, as their goal is to facilitate the output function's performance. The applied learning algorithm dictates how each layer is used to approximate f^*, which in practice operates as a black box. Hence, the layers are called hidden layers (Touretzky & Pomerleau, 1989).

Furthermore, the hidden layers are often vector-valued, and each hidden layer consists of units which act in parallel to represent a vector-to-scalar-function. The number of layers defines the depth of the network, while the number of units defines the width. Layers where each neuron is connected to all neurons in the following layer are called dense layers. Other design options for a network include layers partially connected to the next layer, where filters are applied to the weights of the links between neurons to assemble convolutions. Literature suggest convolutional networks are preferred over dense layers when there are easily separable properties for specific subsets of the input data, such as for image recognition and document classification (Johnson & Zhang, 2014).

A fully connected deep feedforward network requires a defined cost function, the form of output values, an optimizer and an activation function. In addition, number of layers (greater than one) and units must be selected for the model to take form of a network and finally a training procedure defined, to dictate the learning algorithms' behavior.

Cost Functions

The choice of cost function for the neural network is largely related to the type of problem; whether it is regression or classification which is the object, and to some degree preference of the user. A common approach for models which assign a distribution $\psi(\upsilon|\chi;\theta)$ is to use the principle of maximum likelihood (Goodfellow et al., 2016), which means the cross-entropy between predicted values and respective correct values defines the cost function. A mathematical fundament for maximum likelihood-derived functions can be denoted

$$\jmath(\theta) = -E_{\chi,\upsilon\sim\hat{\psi}_{train}} log\, \psi_{model}(\upsilon|\chi) \tag{1}$$

The cost function varies in form between models, where each case is represented by a specific pmodel. Differing versions of the model may include terms which are not dependent on the underlying parameters, but those values can often be discarded, as they are constant with regard to the model parameters. An advantage regarding the properties of cross-entropy is the inherent robustness against saturation of gradients caused by activation functions. The logarithm prevents, to some degree, cases where exponential negative values otherwise would have caused the gradient to be saturated. This has strong impact on the effectiveness of learning algorithms where gradient descent is a prevalent method of optimizing.

For models where probabilistic interpretation are necessary, categorical cross-entropy are used in combination with a softmax function. In cases where the correct values are single classes, the values are one-hot encoded to facilitate a measure of the difference between the correct label and the assigned probabilities for the estimated vector from the softmax (Goldberg, 2016).

$$Ł_{categorical\, crossentropy}(\upsilon,\chi) = log(\upsilon_l) \tag{2}$$

Where l is the class label, υ is a vector of correct values and χ is a vector of predicted values, which represents the conditional class membership distribution,

$$\hat{\xi}_i = (\upsilon = i|\chi) \tag{3}$$

Thus, as a result of the effectiveness in dealing with potentially saturating learning algorithms and a way of quantifying residuals between probabilistic vectors and one-hot encoded labels, categorical cross-entropy stands out as the premier cost function for a deep feedforward neural network aiming to predict classes.

Output Functions

The output function defines the form of the final output from the network. The choice of output function is closely connected to the cost function, as illustrated with cross-entropies tandem with the softmax function. For networks where the object is to classify labels, a softmax function can be applied to represent n different labels' probability distributions. The name stems from the

fact that the function is continuous and differentiable, rendering it a "soft" version of argmax (Géron, 2017).

The general case of n possible labels is represented by a vector v, where each vector element takes a value between 0 and 1, and the elements sum to 1. In this case, each element is given by:

$$\hat{\iota}_i = P(\upsilon = i|\chi) \tag{4}$$

The softmax function takes an input, κ, and squashes it to generate ι, :

$$SoftMax(\kappa_i) = \frac{exp(\kappa_i)}{\sum_j exp(\kappa_j)} \tag{5}$$

where

$$\kappa = \mathbf{W}^T\mathbf{h} + \mathbf{b} \tag{6}$$

and

$$\mathbf{h} = f(\chi; \theta). \tag{7}$$

are the initial linear transformation from the final output of the model, and the output from the final layer of the network, respectively. A network with cross-entropy as the cost function will lead the model to learn weights that drive the softmax function to predict the percentage of counts of the corresponding correct values, mathematically expressed by:

$$SoftMax\left(\kappa(\chi;\theta)\right)_i = \frac{\sum_{j=i}^{m} 1_{\upsilon^{(j)}=i,\chi^{(j)}=\chi}}{\sum_{j=i}^{m} 1_{\chi^{(j)}=\chi}} \tag{8}$$

In theory, this approach is capable of perfectly replicating the correct values, as long as the model can represent the training data. However, imperfect learning algorithms and limited computational power prevents perfect replications to happen in practice (Goodfellow et al., 2016).

Activation Functions

Activation functions or hidden units are functions which weigh input from neurons in the previous layer and calculate an output that is sent forward to the neurons in the following layer. In other words, they receive a vector of inputs, x, compute an affine transformation,

$$\kappa = \mathbf{W}^T\chi + \mathbf{b} \tag{9}$$

and finally apply an element-wise nonlinear function, $\zeta = g(\kappa)$, before forwarding the output ξ (Goodfellow et al., 2016). Recitified linear units (Glorot, Bordes, & Bengio, 2011) is a version of hidden units which returns either a positive value or zero, expressed by the following relationship,

$$ReLU(\chi) = max(0, \kappa) = \begin{cases} 0 & \kappa < 0 \\ \kappa & otherwise \end{cases} \tag{10}$$

and are typically used in combination with an affine transformation:

$$\zeta = h(\mathbf{W}^T\chi + \mathbf{b}) = h(\kappa) \tag{11}$$

Strenghts of the recitfied linear units comprise both their simplicity due to similarities with linear functions and their inherent large derivatives, caused by only returning positive values when activated. They are proven to be more effective than other activation function due to their resistance against saturation and easier requirements for computational power, especially for networks consisting of multiple hidden layers. A rule of thumb to guide the selection of hidden units, according to Goldberg (2016), is that:

$$ReLU > tanh > sigmoid^1 \tag{12}$$

for feedforward neural networks.

Architecture

The architecture of the network is the defined by the number of layers and neurons, how they are connected, and whether they are connected or not. A common structure is to link the layers together in a chain, where the first layer is given by:

$$\eta = \vartheta^{(1)}(\mathbf{W}^{(1)T}\chi + b^{(1)}) \tag{13}$$

and the second:

$$\eta^{(2)} = \vartheta^{(2)}(\mathbf{W}^{(2)T}\eta^{(1)} + b^{(2)}) \tag{14}$$

There are no formal guidelines for how to construct an optimal architecture, other than by trial and error. Some neural networks are deep, with many layers, others perform better with increasing width, whereas some networks' optimal architectures are both wide and deep. Theoretically, one hidden layer is sufficient for a model to approximate any Boreal measurable function from one finite dimensional space to another (Goodfellow et al., 2016). However, in practice that may require infeasible amounts of computational power.

According to Barron (1993), a network consisting of two layers with n neurons in the first layer and 2^n neurons in the second layer is an example of a sufficiently large network, able to approximate any function. Nevertheless, even though the universal approximation theorem states that any Boreal measurable function can be replicated with an adequately large, finite number of neurons, there are no assurance that the optimizer can learn said function. There are mainly two reasons for this; the learning algorithm may fail to find the correct weights for the connections between neurons, while also being prone to overfit the training set, resulting in flawed learning (Goldberg, 2016).

[1] tanh and simoid are other common activation functions.

Backpropagation and Gradient Descent

Backpropagation (Rumelhart, Hinton, Williams, et al., 1988) is the process of computing the gradient of the cost function with regards to each parameter in the model. A general example of how backpropagation works can be expressed by the chain rule of calculus in an n-dimensional space, where multiplying the Jacobian matrix $\frac{\partial \upsilon}{\partial \chi}$ with a gradient $\nabla_\upsilon \kappa$ returns the gradient for χ. Consider the basic chain rule of calculus first,

$$\frac{\mathbf{d}\kappa}{\mathbf{d}\chi} = \frac{\mathbf{d}\kappa}{\mathbf{d}\upsilon}\frac{\mathbf{d}\upsilon}{\mathbf{d}\chi} \tag{15}$$

where $\upsilon = f(\chi)$ and $\kappa = \mathbf{g}(\upsilon)$.

Then generalise it for any $\chi \in \mathbb{R}^{\mathrm{m}}$, $\upsilon \in \mathbb{R}^{\mathrm{n}}$, where $\mathbb{R}^{\mathrm{m}} \to \mathbb{R}^{\mathrm{n}}$, and $\mathbf{g}$ maps from $\mathbb{R}^{\mathrm{n}} \to \mathbb{R}$:

$$\nabla_\chi \kappa = \frac{\mathbf{d}\upsilon}{\mathbf{d}\chi}^T \nabla_\upsilon \kappa \tag{16}$$

where $\frac{\mathbf{d}\upsilon}{\mathbf{d}\chi}$ is the $n \times m$ Jacobian matrix of $\mathbf{g}$. The backpropagation performs the Jacobian-gradient product for each data point or convolution of weights in the network, and feeds it to the learning algorithm.

The most common learning algorithm for a deep feedforward neural network is the stochastic gradient descent or a modified version of it (Bottou, 2012; LeCun, Bottou, Orr, & Müller, 2012). The learning algorithm takes into consideration a function f parameterized by θ, desired pairs of input and output, and tries to minimize the selected cost function by altering the θs. Mathematically,

$$\mathbf{min} \sum_{j=i}^{m} (f(\chi_i; \theta), \upsilon_i) \tag{17}$$

by changing the parameters θ in the direction of the respective gradients. The weights are then changed by a scale defined by the learning rate, λ.

Stochastic gradient descent is a modified version of gradient descent where only subsets of parameters are tuned for each iteration, thus drastically reducing the needed computational power for performing an iteration. Additions to vanilla stochastic gradient descent include momentum, learning decay and Nesterov momentum. Momentum defines to which degree the weights change in the direction of the gradient: high momentum leads to greater changes than lower momentum. Learning decay decreases the learning rate for each iteration by a fixed amount. The Nesterov momentum is an alteration of the order in which the gradients are calculated, where a change of parameter is based on the previous gradient at first and then rectified in the direction of the current gradient. Theory suggests that Nesterov momentum is an effective tool to prevent an optimizer from getting stuck in a local minima or saddle point (Nesterov, 1983). All these hyper-parameters, as well as the initial learning rate, change the behaviour of the learning algorithm and can yield increasing accuracy if tuned correctly.

Another, more advanced, version of stochastic gradient descent is the optimizer Adaptive Moment Estimation (ADAM). ADAM introduces concepts of moment and adaptive learning rates to decrease the chance for the algorithm to get stuck in local minima and improve relevant learning (Kingma & Ba, 2014). Adaptive learning rates refer to random changes in learning rates from parameter to parameter in the network, with the intention of discovering nuances otherwise covered by dominant neurons. ADAM also considers the hyperparameters beta 1, beta 2, and epsilon, which can be manually tuned. The betas define the exponential decay for first and second-moment estimates, whereas the epsilon prevents division by zero in the learning process.

Dropout

Dropout is a regularization technique which aims to reduce overfitting by preventing the learning algorithm from becoming too dependent on the training set (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). This is achieved by ignoring a fixed percent of randomly selected neurons for each layer, per iteration. Empirical studies suggest that dropout improves performance of deep neural networks (Krizhevsky, Sutskever, & Hinton, 2012), especially when combined with Rectified Linear Units (ReLU) as activation functions (Dahl, Sainath, & Hinton, 2013).

Data and Variables

With data for almost 1.3M company observations over the years 2000–2009, SNF's[2] and NHH's Norwegian Corporate Account (NCA) dataset is the basis for our analysis. In total, there are 170 variables in the dataset containing data about the companies' income statements and balance sheets, some generated variables, company information, sector information, and accounting items not included in the companies' income statements or balance sheets. Our main learner will be the Bisnode D&B Norway AS's internal credit rating variable.

Before the dataset can be used in our analyses, there are some pre-processing steps that must be carried out. First, companies without credit ratings must be excluded from the dataset. Then, additional variables that may be relevant for credit rating must be generated. Finally, there are some other pre-processing steps remaining related to the removal of (assumable) irrelevant variables and missing values, and the incorporation of multilevel factors and standardisation of the variables for use in the deep learning model. Appendix B contains the code used for creating the "mother" data-set and feature data.

[2]Centre For Applied Research at NHH

Credit Ratings in the NCA Dataset

The variable *ratingkode* contains Bisnode D&B Norway AS's internal credit ratings, shown in Table 1. There are five categories of ratings, ranging from 'C' (= credit not advisable) to 'AAA' (= strong creditworthiness), in addition to the two categories not rated and bankrupt/dissolved/liquidated. Companies from 2005 and onwards have been assigned a rating in the dataset. We thus exclude all observations before 2005. We also exclude companies that have not been rated, and companies that are bankrupt as indicated by the variable *konkaar*. Furthermore, to reduce some of the imbalance in the dataset with respect to the frequency of each rating, we have chosen to put rating 'B' – credit against security – and rating 'C' – credit not advisable – together to one category of companies that are not creditworthy. The third column in Table 1 summarizes how each rating category is treated in the rest of this paper.

Table 1: Bisnode D&B Norway AS's Internal Credit Ratings

Rating category	**Explanation**	**In this paper**
0	Not rated	Excluded
1 - C	Credit not advisable	C/B – not creditworthy
2 - B	Credit against security	C/B – not creditworthy
3 - A	Creditworthy	A
4 - AA	Good creditworthiness	A
5 - AAA	Strong creditworthiness	AAA
9	Bankrupt/dissolved/liquidated	Excluded

Generating Variables Relevant for Credit Rating

To identify variables relevant for corporate credit rating, we conducted an extensive literature review. Appendix D presents a summary of the variables used in some of the studies for credit rating prediction that we consulted (Balios et al., 2016; Doumpos, Niklis, Zopounidis, & Andriosopoulos, 2015; Hajek & Michalak, 2013; Hwang, Chung, & Chu, 2010; Hájek & Olej, 2016), in addition to financial variables used in a Bisnode D&B business report for a rated company (Bisnode, 2016).

The NCA dataset provides information about the companies' financial results mainly in level form. Based on the variables identified from the literature review, we also generated 23 new variables, mainly ratios, from the NCA data. Table 2 gives an overview of these generated variables. As the table shows, our generated variables can be grouped into five key categories for assessing financial performance: profitability, activity, asset structure, liquidity and leverage. In addition to the variables in Table 2, we calculated about 20 other variables such as the interest coverage, quick ratio and current ratio; however, due to the number of positive or negative

infinite values being relatively high, we decided to leave them out.[3] We consider the variables in Table 2 to still be highly representative of profitability, activity, asset structure, liquidity and leverage.

Furthermore, when it comes to company size, we consider the original NCA variables such as sales revenue, equity and total capital/total assets to be representative. We also generated a few additional variables that we consider relevant. Those are company age and dummies for 1) whether the company has listed shares, bonds, commercial papers or options, 2) whether the company has an auditing firm, and 3) whether the company has an accounting firm. Age is considered relevant, for example, as recently created companies cannot achieve an 'AAA' rating regardless of performance in their first years of operation. The stock exchange dummy is included as listed companies tend to have better ratings than the overall sample, with very few not creditworthy ratings. We also experimented with including market data from Oslo Stock Exchange as several studies point out how the market value of companies is relevant in predicting credit ratings (see Appendix D). However, the inclusion of market data did not improve our results. This can be explained by the fact that only a small proportion of companies with credit ratings in the NCA dataset have market data. Thus, we chose to only include the stock exchange dummy to distinguish between listed and non-listed companies. Finally, the auditing and accounting dummies are included since we hypothesize that companies employing auditing and accounting firms might also tend to achieve higher ratings than those without.

[3]For example, when it comes to the interest rate ratios, many companies have no interest expenses and thus ratios such as the interest coverage result in infinity values as the denominator, interest expenses, equals zero.

Table 2: Generated Variables

Category	**Variable**
Profitability	Earnings Before Interest and Taxes (EBIT)
	Return on Assets (ROA)
	Return on Equity (ROE)
	Return on Capital (ROC)
	Retained Earnings to Total Assets
Activity	Sales to Total Assets
	Sales to Net Worth
	Operating Revenue to Total Assets
	Working Capital
Asset Structure	Working Capital to Total Assets
	Fixed Assets to Total Assets
	Intangible Assets to Total Assets
	Total Assets to Equity
	Log of Total Assets
Liquidity	Cash Flow
	Non-Cash Working Capital
Leverage	Total Debt to Total Capital
	Short-Term Debt to Total Capital
	Long-Term Debt to Total Capital
Additional Variables	Company Age
	Stock Exchange Dummy
	Auditing Dummy
	Accounting Dummy

Dealing with Time

Bisnode D&B uses three years of historical data in their ratings (Bisnode, 2016). In literature, we have found time periods from $t-1$ (Doumpos et al. 2015) to t–3 (Hwang, Chung and Chu, 2010) being the basis for rating predictions. In this paper, we intend to treat the data as time agnostic to increase our sample size. If we were to use historical data, we would lose observations as a number of companies are not registered in the NCA dataset in previous years – at least not with the same organisational number. This could be because they did not exist before. However, we also found examples of companies founded more than 30 years back in time that lack historical data, e.g. before 2005. When choosing to use same-year data, we assume that the financial data for the companies has not changed radically – so that this year's data gives a relatively representative picture of the company's financial situation at the time the rating was made.

Furthermore, we do not include historical ratings in our model either in case those exist. Our objective is to develop a model that is capable of rating any company - regardless of previous rating history. This is also in accordance with literature we have consulted, where previous ratings were excluded from the set of variables used (see Appendix D).

Dealing with Missing Values

There are a number of not available (NA) entries in the NCA dataset. We use two methods for dealing with NAs in the dataset. Before generating new variables, we exclude variables that contain a relatively high proportion of NAs and that are assumed to be irrelevant for the credit rating analysis, from the dataset.[4] Then, we remove rows with remaining NAs using the 'na.omit' function in R. This gives us a clean data set before we generate new variables.

When generating new variables, we get some positively or negatively infinite (Inf) entries in the data set. This happens in cases where we divide by a denominator that equals zero. To deal with these Inf entries, we set them to zero. The logic behind this method is as follows. Every variable is interpreted as an interaction term consisting of the variable itself and a dummy indicating whether each observation (company) has data available for the given variable. If the company has data available, the value of the entry is returned (dummy = 1); otherwise the entry is set to zero (dummy = 0).

One-Hot Encoding

We have a few factor variables with more than two levels.[5] In the random forest model, we leave them as is. For the deep learning model, on the other hand, we one-hot encode these factors. That is, for each factor level we add a binary variable indicating whether the factor level is true (value = 1) or false (value = 0). Due to the one-hot encoding of multilevel factor variables, there will be a slightly higher number of variables in the deep learning model than in the random forest model.

Standardisation Procedure

We standardise the variables before they are used in the deep learning model. For each observation, this is done by subtracting the mean and dividing by the standard deviation. Please note that we standardise the variables before any NAs are set to zero to exclude the NAs from the calculations of variable mean and standard deviation. One-hot encoded multilevel factors and dummy variables are not standardised.

[4] Those variables are: *brkod*2, *brkod*3, *brtxt*2, *styrehon*, *daglonn*, *revhon*, *ansatte*, *bransjek_*02, *bransjek_*07, *bransjegr_*02, *bransjegr_*07, *bransjet_*02, *bransjet_*07, *postnr*, *kommnr* and *poststed*.

[5] Those variables are: *eierstruktur* and *sector*.

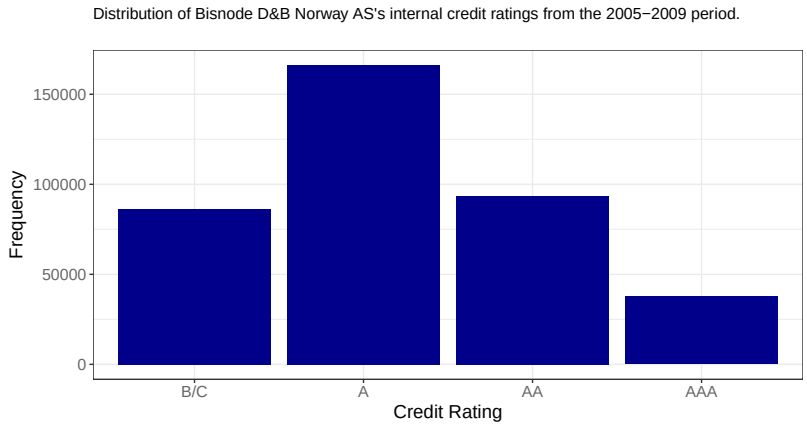
Final Dataset

The pre-processing tasks described in this chapter leave us with a data frame consisting of 168 variables, being used in our random forest application, and a matrix consisting of 184 variables used; that is, 168 variables plus 16 one-hot encoded variables for the multilevel factors, which we use in our deep learning application.

The distribution of credit ratings for the approximately 384,000 companies in our final dataset is shown in Figure 2. Rating 'A' is the most frequent and more than 40 per cent of the firms have this rating. The least frequent rating is 'AAA', only achieved by 10 per cent of the firms in the dataset. The remaining 50 per cent of the firms are almost evenly distributed between rating 'AA' and rating 'B'/'C'.

Although we reduce some of the imbalance in the dataset by putting together rating 'B' and 'C' to one category (not creditworthy), it is important to keep in mind for the analysis that there is still imbalance in the dataset – especially with respect to rating 'AAA'.[6]

Figure 2: Credit Rating Distribution



Training, Validation and Test Split

Both our benchmark and deep learning models depend on the fine-tuning of hyperparameters. Because of this, we decide on a three-way split of our data: training data fed to the algorithms, validation data as a point of reference for tuning, and final test data, which is only introduced to both models' final forms. Since we have a satisfactory amount of rows, we decide on a 70/15/15 split for training, validation and test data respectively. These sets are created by assigning a randomized sample of observations to each.

[6]In our analysis, we have chosen to keep 'AAA' as a separate category. However, one could have considered merging the top two ratings – 'AA' and 'AAA' – to the category "good creditworthiness" to reduce imbalance further.

Benchmark Implementation

To assess the performance of our deep learning model, we present two benchmarks. While the first approach is simply based on guessing either last year's value as this year's value or guessing the most common value, the second approach is random forest.

Benchmark Approach 1: Simple Guessing

Guessing last year's credit rating as this year's credit rating may seem like a natural starting point for benchmarking, and we decided to add this strategy as one of our benchmark approaches. This is done by including a lagged variable $(t-1)$ for credit rating to the observations in our dataset. Comparing the rating in year t to that in year $t-1$ for the observations where we have data on ratings in both years, we find that we would be correct in 67.88% of cases.

Instead of guessing last year's rating, an alternative approach is **guessing the credit rating mode as this year's credit rating**. The distribution in Figure 2 shows that the most common rating is 'A'. 'A' is the rating in about $166,000$ instances out of about $384,000$, so guessing 'A' for each instance results in a prediction accuracy of 43.23%.

Benchmark Approach 2: Random Forest

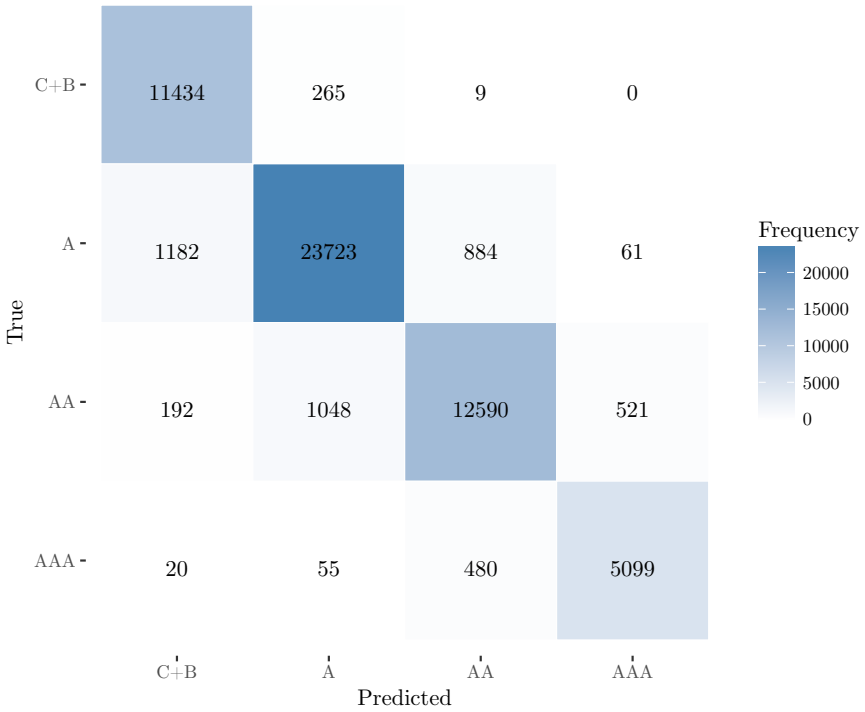
Our second benchmark is a result of running various random forest models. We started off loading our data on to Amazon Web Services where we trained and tuned a random forest model on a subset of our training set, utilising parallel processing for increased speed. The tuning process yielded an optimal `mtry` of 84 and a test set accuracy of 90.31%.

Seeking to improve on our first model, we ran several models where we steadily increased the number of trees while keeping the value for `mtry` constant. We ran models with 32, 64, 128, and 512 trees, and saw only marginal improvements in prediction accuracy. Our best random forest model has 512 trees and a out-of-sample prediction accuracy of 91.81%.

The distribution of predictions are given in Figure 3. The figure shows that the most frequent predicted credit rating for each true rating, is the true rating. Furthermore, the random forest model has zero errors when it comes to predicting credit rating 'AAA' when the true rating is 'B' or 'C'.

The model thus succeeds well with respect to not assigning the top rating ('AAA') to a company that belongs to the not creditworthy class. When looking at the binary classification problem - creditworthy (ratings 'AAA', 'AA' or 'A') or not creditworthy (ratings 'B' or 'C') - our random forest model performs even better. In this case, our model makes correct predictions in 97.03% of cases.

Figure 3: Final Random Forest Model



In only 20 cases does our model make the mistake of predicting 'B' or 'C' when the true rating is 'AAA'. We view this as an acceptable error rate, and indeed an economically significant result with reference to our allocation of capital-discussion in the introduction. Our model does, however, predict 'not creditworthy' (that is, 'B' or 'C') for 1182 companies that are actually rated 'A'. This is an economically significant error not in favour of our model, as it is likely that these companies would suffer in the marketplace from a poor rating when, in fact, they should enjoy the benefits of being 'creditworthy' ('A'). Nevertheless, our random forest model performs well, and it is clear that our deep learning model is up against tough competition.

Benchmarks Summarised

We have presented prediction accuracies of two benchmark approaches: simply guessing on a value and a random forest model. These estimates will be important in the process of assessing the performance of our deep learning model. While the first two estimates are based on simple guesses, our random forest model is both a better model in terms of prediction accuracy and in terms of richness. Our guessing-based estimates do not include any estimates of *how wrong* a guess is when a guess is wrong, whereas all guesses the random forest model makes are laid out in the confusion matrix in figure 3. For reference, we have summarised our benchmarks in table 3.

Table 3: Benchmark Prediction Accuracies

	Guessing last year's rating	Guessing the mode	Random forest
Prediction accuracy	67.88%	43.23%	91.81%

Deep Learning Implementation

Our initial approach to building a deep learning model was doing it manually, meaning we adjusted the network architecture and hyperparameters simultaneously. To simplify this process we started with the ADAM optimizer, thereby reducing the number of hyperparameters we needed to tune - at least initially. This left us with the number of epochs, batch size, and network architecture. The latter includes a wide range of choices, such as:

- The number of hidden layers
- The type of each hidden layer (dense, convolutional, LSTM, etc.)
- The number of nodes per individual hidden layer, and the activation function
- The weight initialization and any weight limits for the nodes in each hidden layer
- Regularization for each layer (e.g. L2 norm and/or Dropout)

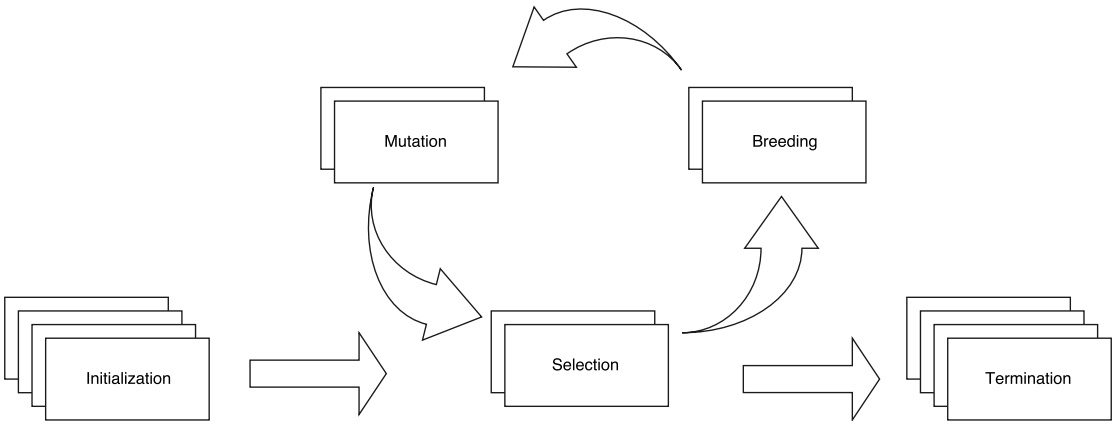
Due to the non-additive nature of neural network design, adjusting all these factors in tandem is a complicated endeavour. For instance, if two hidden layers perform well, adding a third may completely reshape the underlying dynamic in the two previous layers. Changing many of the other factor may have the same effect, so to a certain extent it is an approach of starting from scratch each iteration and testing a full network composition. As previously mentioned we applied some design constraints, which made the job a little easier. After hundreds of iterations of manual designs we were certainly getting somewhere, but there was a certain element of guesswork involved.

Genetic Algorithm

As an alternative to designing network architectures manually, we attempted to use a genetic algorithm to automate some of thedesign choices. Put simply, genetic algorithms optimize attempts to solve some well-defined problem (Larson, 2009). In the case of deep learning, one approach to designing a network could be to try thousands of combinations of hidden layers, nodes, layer type, regularization, and so on, and pick the one that performs best. For simple problems with simpler models this may be feasible, but training and testing a deep learning network is usually time-consuming, such that doing it repeatedly thousands of times quickly makes the computational cost very high.

Genetic algorithms work similarly, but rather than naively continuing to try completely random combinations, the algorithm observes the traits of high-performers and uses these insights to limit the option space for following iterations. At the high level, a genetic algorithm creates some initial population of pseudo-randomized entities, measures their performance, selects some portion of top performers for survival, then uses their traits to create new entities to refill the population - usually with some added randomized traits to leapfrog performance and escape local minima. The performance of each entity in this partially new population is measured, and the process repeated a given number of times or until some stopping condition is met. In a nutshell, a genetic algorithm simulates natural selection or selective breeding, hence analogous terms like *children* and *parents* are frequently used.

Figure 4: Conceptually How a Genetic Algorithm Works



In our case, this meant initializing a set of 20 deep learning networks with pseudo-randomized architectures. These were then trained separately on the same training data, and their performance measured on the same validation data. The 40% top performers were selected for survival, as well as around 10% of the other networks - in case some performed poorly due to a genetic quirk that did not work well with the rest of that network design, but may work better in another configuration. Two and two parent networks were then selected and used in combination to create two child networks, 20% of which were given some random mutation. This was repeated until the remaining 11-12 population slots were refilled with new children. The resulting new population was then trained on the same training data, the performance of each measured on the validation data, and the whole process repeated for a total of 15 generations.

Genetic Approach to the Problem

Though there is a wealth of literature on genetic algorithms in general, the implementations are usually greatly simplified, such as that of Larson (2009). With significant inspiration from the work of Harvey (2017), we developed a custom algorithm in Python (Appendix C). Contrary to

other implementations we found, such as that of Harvey, we desired a separate choice of number of neurons for each individual hidden layer, rather than simply evolving the number of hidden layers and a single number of neurons per hidden layer. Consequently, though the underlying structure bears close resemblance to Harvey's, we had to make significant changes to facilitate the desired genetic process - such as adding Layer objects and completely re-engineering the genetic mechanisms.

As such, our needs involved a greatly increased number of variables to evolve; in addition to the number of hidden layers, for each such layer we set the number of neurons and amount of dropout regularization as variable. There are of course a plethora of other deep learning architecture design choices, so in order to make our implementation practically feasible we had to restrict the space of genetic options. We therefore limited the algorithm to feedforward dense networks with ReLU activation functions, and required it to use the ADAM optimizer, as we can expect relatively good results without needing to adjust its hyperparameters. After some experimentation, we also fixed the number of epochs at 50 and batch size at 2048, as this provided a good balance between computational cost and model convergence.

The algorithm was consequently able to vary the number of hidden layers, the number of neurons for each separate hidden layer, and the amount of dropout regularization (if any) applied to each individual layer. However, if these values - and especially the latter two - were essentially continuous, we would need a very large number of iterations to realistically sample a wide variety of the option space. For instance, a continuous range of potential neuron counts between 10 and 10,000 would necessarily favour wide layers when using uniform sampling. For each variable, we therefore defined a discrete and limited set of alternatives, so that we could reasonably sample most combinations with limited iterations:

$$\begin{aligned} layers &\in [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] \\ nodes_i &\in [8, 24, 32, 64, 128, 256, 512, 750, 1024, 1500, 2048, 3000, 4096] \\ dropout_i &\in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7] \end{aligned} \tag{18}$$

where $layers$ is defined at the network level, whereas $nodes_i$ and $dropout_i$ are defined per layer i.

Implementing the Algorithm

Even with these design constraints, running such a genetic algorithm is very computationally expensive. For our approach to even be feasible within the limited time available, we therefore implemented our custom algorithm in Python so that we could run it on a Graphics Processing Unit (GPU). Though originally designed and built for running games, GPUs are fortuitously perfect for the matrix multiplications involved in training deep neural networks. Compared to Central Processing Units (CPUs) with their few complex cores and single-threaded optimization, GPUs have hundreds of simpler cores with concurrent threads that are much better at solving smaller problems in parallel (Shaikh, 2017). In our case, training our deep learning models on a

GeForce GTX 980Ti gave a more than ten-fold increase in execution speed compared to running it on a laptop CPU [7]. With these tools, we could feasibly run our genetic algorithm within a reasonable amount of time. Even so, 15 generations with a population of 20 networks trained for 50 epochs on batches of size 2048 still took nearly 10 hours, despite various simplifying design constraints.

Our algorithm, found in appendix C, was an object-oriented implementation where we defined a Network class, Layer class, and Optimizer class. A *Layer* is a simple object that represents a hidden layer and contains only two properties: the number of nodes (or neurons) in that layer, and the amount of dropout regularization applied (if any). A *Network* object represents a complete deep learning network model; it has an array of functions and properties, but the most important are its accuracy measured on the validation data, and its architecture - a list of Layer objects where the first immediately follows the visible input layer, and the last immediately precedes the visible output layer. An *Optimizer* object contains the genetic properties such as the share of top performers selected for survival, the number of relatively poorly performing networks to randomly select for survival, and each child network's chance of a mutation. It also contains the core functions of the genetic algorithm, such as creating the initial randomized population, evaluating network performances, breeding children, mutating children, and the overarching function for evolving a population that performs those aforementioned steps. An array of object-agnostic functions are also defined to handle various supporting tasks, such as training and validating a *Network*, logging top architectures, and running the full algorithm.

As such, each of the 15 generations is a list of 20 *Network* objects, each itself containing a list of *Layer* objects defining its architecture. The Python file also includes a range of helper functions for plotting and evaluating final results, as well as the code for the final deep learning model we built manually based on the findings from running the genetic algorithm. The key functionality of the algorithm to note is how parent networks are combined to breed child networks, and how mutations are introduced. Considering each network has an individual number of hidden layers and individual combinations of nodes and dropout rates for each of these, there is a challenge of combining genetic material between parent networks in a way that preserves the traits that made each network successful. Our approach was therefore to iterate over each hidden layer in the parents, and select one complete *Layer* object to add to the child's architecture. For instance, when selecting the child's first hidden layer, the algorithm randomly picks the full first hidden layer of either the father or mother network, and does the same for the second hidden layer, and so forth. When the number of hidden layers in the parents differ, the algorithm will eventually find itself picking randomly between a *Layer* object and nothing (since the other parent has no corresponding hidden layer). We specifically allowed for this, such that a child network's depth will be between the depths of its deepest and shallowest parents. If a child receives a mutation, this simply means replacing one of its hidden layers with a completely randomized layer design

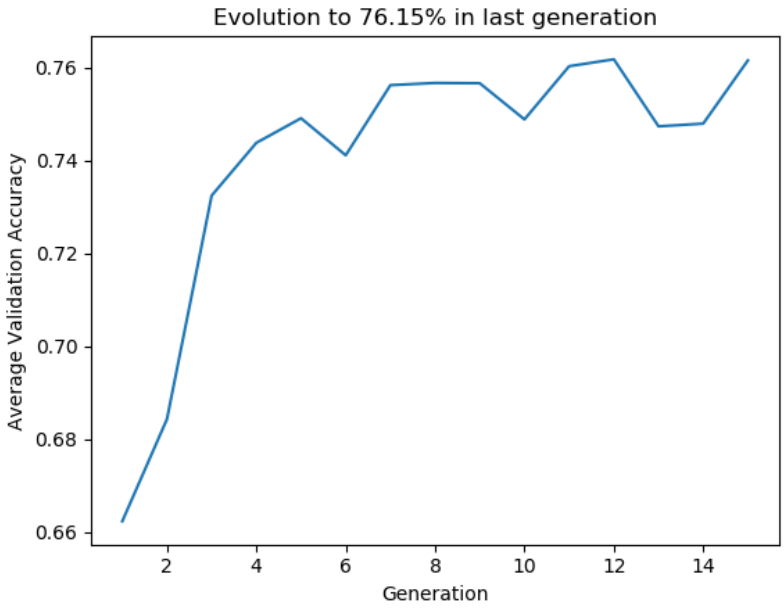
[7] Specifically, our algorithm was implemented in Python 3.6.3, using the Keras framework version 2.0.6. For Keras to work with our GPU, we had to use the GPU-version of Tensorflow 1.1.0 with NVIDIA's CUDA Toolkit 8.0 and cuDNN 6.0

(i.e. combination of number of nodes and dropout portion).

Algorithm Output

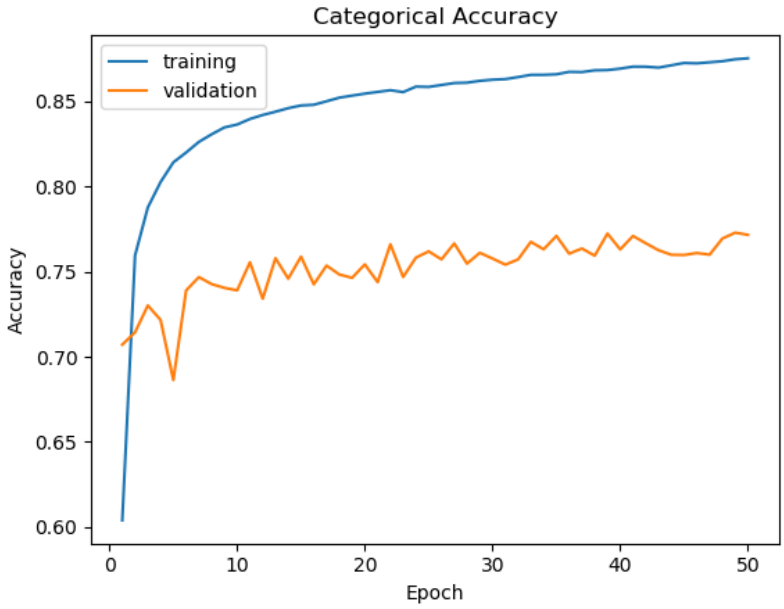
The average categorical accuracy over all networks in each generation are shown in Figure 5, where we see a clear improvement over time. The dips in some generations are to be expected as some random mutations yield very poor performance, or even some children of successful parent architectures may combine features in a way that simply performs poorly.

Figure 5: Evolution of Generational Average Validation Accuracy



As Figure 5 shows, the algorithm starts to converge around peak performance by generation 7 or 8. Further iterations of evolution only yield marginal improvements, with some generations seeing poorer performance due to the aforementioned reasons. At the end of 15 generations, the average network accuracy on validation data is 76.15%. It is especially interesting that no network have at this point exceeded 78% accuracy, meaning the generation has converged on a shared set of traits that generally yield the best performance - within the aforementioned constraints we have applied. Figure 6 below shows the performance of the best network at the end of the last generation, measured on the training and validation data sets.

Figure 6: Training and Validation Accuracy of Genetic Top Performer



As expected, the training accuracy continues to improve as the network is trained through additional epochs. As with the genetic algorithm, we here use 50 epochs, 2,048 in batch size, and the default ADAM optimizer. Interestingly, the validation accuracy almost reaches its zenith by the end of the first epoch. Throughout further epochs the validation accuracy continues to increase slightly on average, but this slight upwards trend contains a lot of peaks and troughs. This is likely caused by noise in the data, which makes it harder to determine whether validation accuracy has in fact begun to decline by the 50th epoch - in other words whether we have begun to significantly overfit.

Figure 7: Genetic Top Performer Validation Data Confusion Matrix

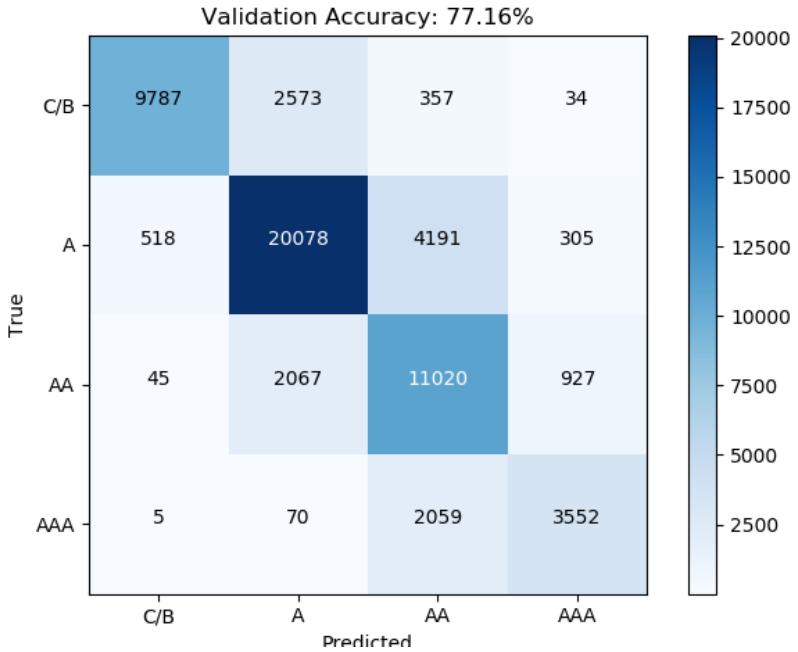


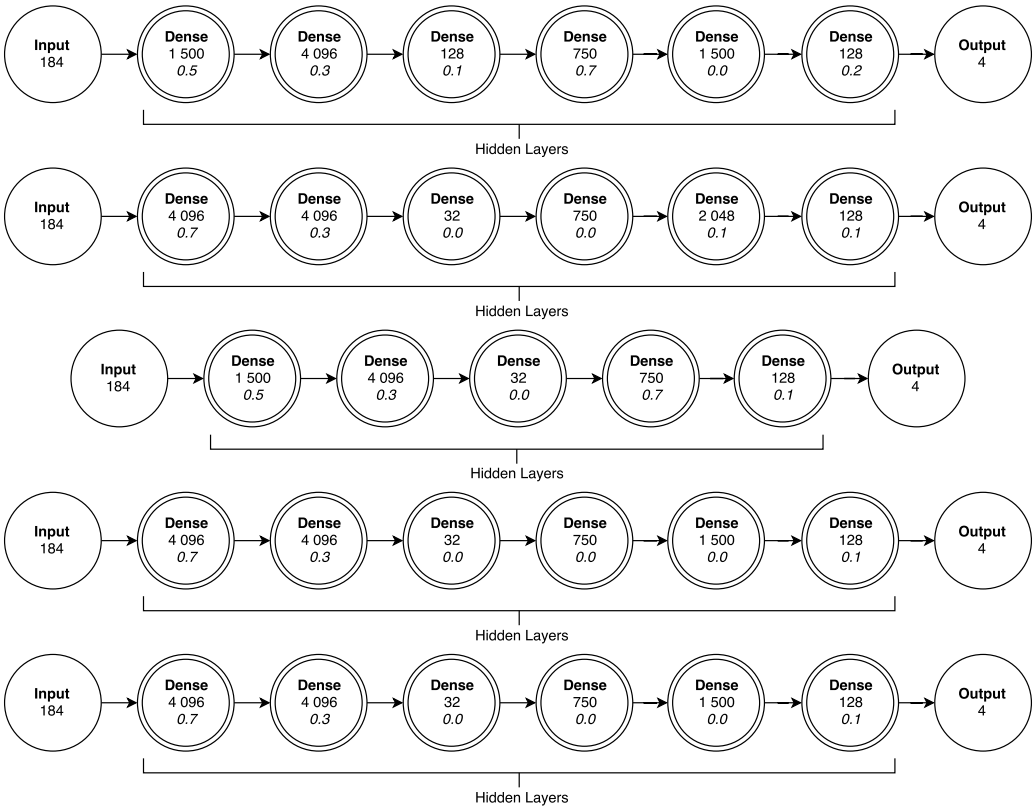
Figure 7 above shows the confusion matrix on the validation data set for the same model at the end of its 50 epochs. Total categorical accuracy is at 77.16%, and the model is generally good at separating between creditworthy and non-creditworthy companies - i.e. 'C/B' versus 'A', 'AA', or 'AAA'. From an economical perspective, the worst mistake the model can likely make is misclassifying what should be a 'C' or 'B' rating as something creditworthy, and the model in fact makes that mistake in 23.2% of 'C/B' cases. Luckily, 86.8% of these are at least no worse than giving an 'A' rating. Only 0.27% of 'C/B' rated companies are incorrectly given a 'AAA' rating. In fact, as Figure 7 shows, what the model struggles most with is dividing between 'AA' and 'AAA' ratings. When the company is in fact 'AA' the model gets it exactly right in 78.38% of cases - and in fact gives a non-creditworthy rating in only 0.32% of cases. However, when the true rating is 'AAA', the model only gets it exactly right in 62.47% of cases, and incorrectly rates it as 'AA' 36.21% of the time.

However, the argument can be made that this is really the least serious mistake from an economic perspective. Misclassifying a 'AAA' rating as 'AA' or vice versa is far from as big a mistake as giving a creditworthy rating to a non-creditworthy company, or vice versa. Additionally, one could expect the subjectivity of the person giving the rating to matter most in the distinction between 'AA' and 'AAA'. This is indeed what the model suggests, as this is where it struggles most to make the distinction based on the data available to it. Using Figure 7 and considering it a binary problem of classifying whether a company is creditworthy or not, the model achieves a notable 93.87% accuracy - and this is without even having tuned the model provided by the genetic algorithm.

Combining the Winning Traits

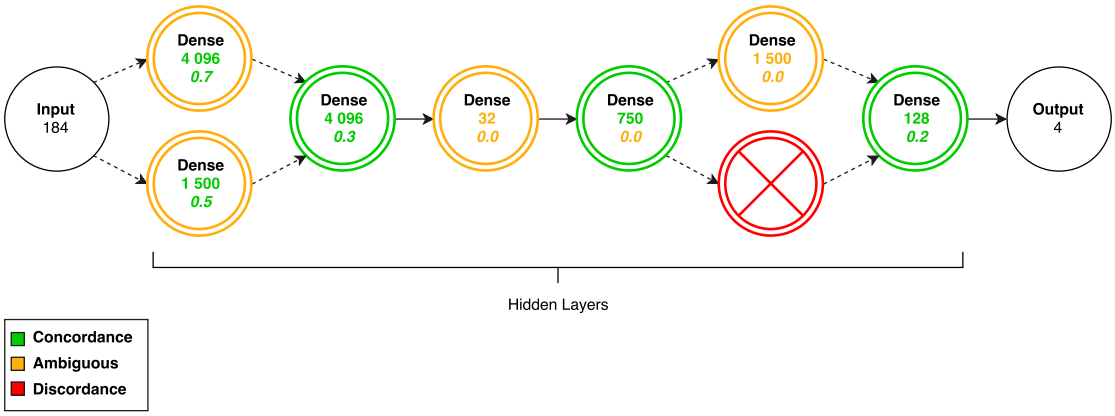
Our goal is, however, to build the best deep learning model that we can for predicting individual credit ratings, albeit treating 'C' and 'B' as simply not creditworthy, hence the model can likely be improved further. The results of running the genetic algorithm suggests that it is a very useful starting point for a rough design of the network architecture. Figure 8 below therefore shows the architectures of the top five networks at the end of the 15 generations, so that we can build on these in developing our own model.

Figure 8: Top 5 Network Architectures after 15 Generations



Interestingly, four of the five architectures have six hidden layers, and one has five. Recall that we allowed the algorithm to choose between 2 and 12 hidden layers. Furthermore, several of the top architectures share traits, with some being common among all five. Attempting to map these commonalities, we compiled Figure 9 below. A *trait* refers to either a complete layer design, the number of neurons in a given layer, or the dropout regularization amount for a given layer. In an attempt to map shared traits between the top five networks, we compiled the below Figure 9. Traits marked green are shared by all five, yellow are shared by some, and red are major deviations.

Figure 9: Top 5 Genetic Architectures Combined



As Figure 9 shows, the five genetically derived networks share many traits. One could argue this is the result of "inbreeding" due to the same initial top performers becoming frequent parents, but inspection of the other fifteen networks in the final population refute this, as does conceptually the relatively high rate of random mutations.

In summary:

- The first hidden layer is wide with a high dropout rate; three networks have 4,096 nodes and 70% dropout, while two have 500 nodes and 50% dropout.
- All second hidden layers contain 4,096 nodes with a dropout of 30%.
- All third hidden layers are narrow; all but one have 32 nodes and no dropout, with the exception being 128 nodes and 10% dropout.
- All fourth hidden layers contain 750 nodes, where all but one have no dropout.
- The fifth hidden layer is more varied; three networks have 1,500 nodes with no dropout, one has 2,048 nodes with 10% dropout, and one drops the layer entirely. This latter network is more sparse and yet only in third place, which could make it preferable as it essentially sacrifices no performance for this sparsity.
- All final hidden layers contain 128 nodes with a dropout of 20

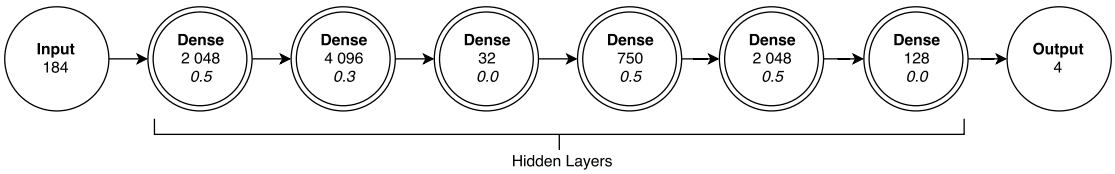
The Final Model

Based on the output of the genetic algorithm, we have a much more refined pool of architecture options from which to choose, as seen in Figure 9. Based on the shared traits of the top five genetic networks, we built a final model by iteratively testing various network designs and hyperparameter values.

After dozens of iterations, we arrived at a final model. Compared to Figure 9, we slightly adjusted

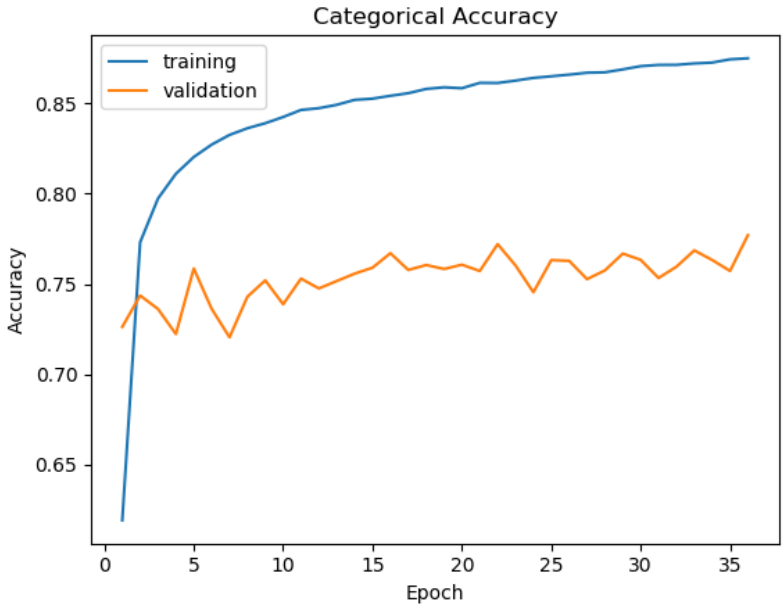
the number of nodes and hidden layers, introduced dropout to layers with more than 128 neurons, and manually adjusted the ADAM optimizer parameters. We experienced increasing accuracy when enlarging the fifth hidden layer, allowing for more neurons and an overall larger neural network. Interestingly, selecting a number of neurons for the first hidden layer which were closer to the lower bound suggestion from the genetic algorithm proved to be superior to any greater number of neurons. Including dropout for all large layers while removing dropout for the sixth layer, containing only 128 neurons, also enhanced model performance. Furthermore, no alteration of the models' depth, except for keeping the fifth layer, improved the validation set accuracy, hence revealing the architecture in Figure 10 below as optimal.

Figure 10: Architecture of Final Model



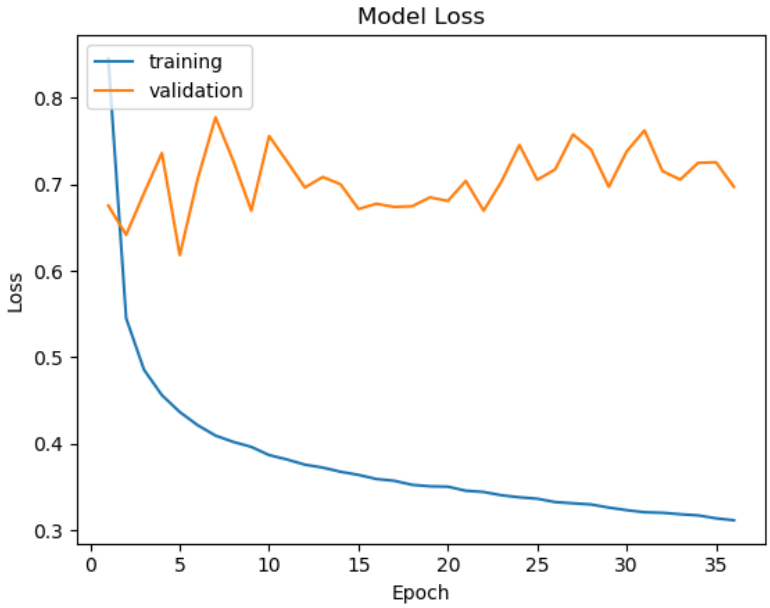
With the network architecture in place, optimizing the ADAM optimizer parameters included attempts at adjusting the algorithm's learning rate, the first- and second-moment learning decay, and the epsilon. Neither changing the learning decay, nor increasing the epsilon in accordance with recommendations by Brownlee (2017) yielded any notable model improvements. However, tweaking the initial learning rate by increasing it with an order of magnitude increased the accuracy substantially. Ultimately, validation accuracy peaked with the architectural design in figure 10 and the initial learning rate of ADAM set to 0.2, resulting in a categorical test accuracy of 77.91%.

Figure 11: Training and Validation Accuracy of Final Model



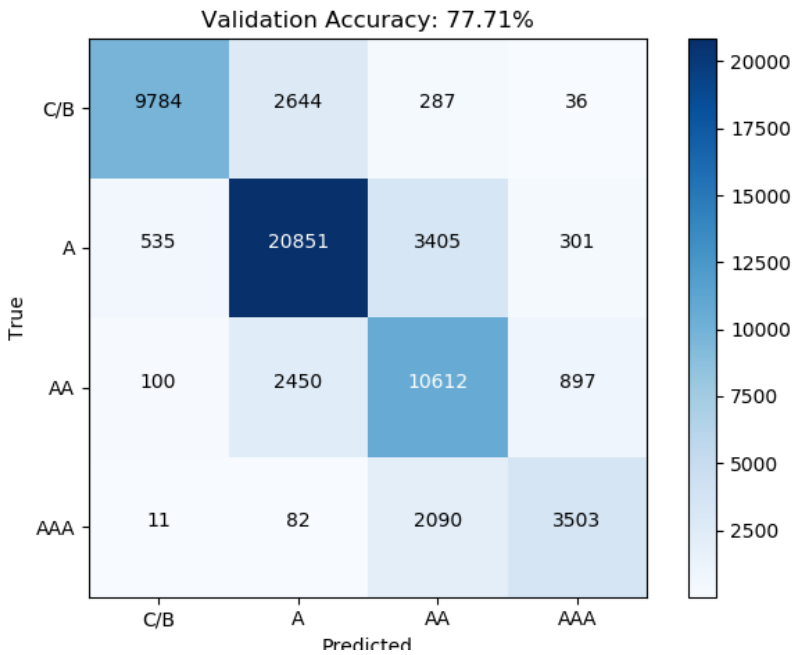
The training accuracy in Figure 11 above is similar to that of the genetic algorithm's top performer in Figure 6, with a high initial validation accuracy accompanied by a slowly upwards-trending validation accuracy. As expected, the training data continues to increase steadily - albeit asymptotically - through additional epochs. It is hard to say whether the curves converge or not, due to the noisy nature of the validation accuracy, but experience tells us that the validation curve will stabilize around its current level while the learning algorithm continues to overfit the training set. Looking at the training and validation loss in Figure 12 below, we get a similar picture; training loss continues to fall, but validation loss might potentially have a slight upwards trend, meaning we are overfitting the data and should rather stop where we are. Though we run training for 50 epochs, we had included a stopping condition on the training accuracy, such that the algorithm stopped after 36 epochs.

Figure 12: Training and Validation Loss of Final Model



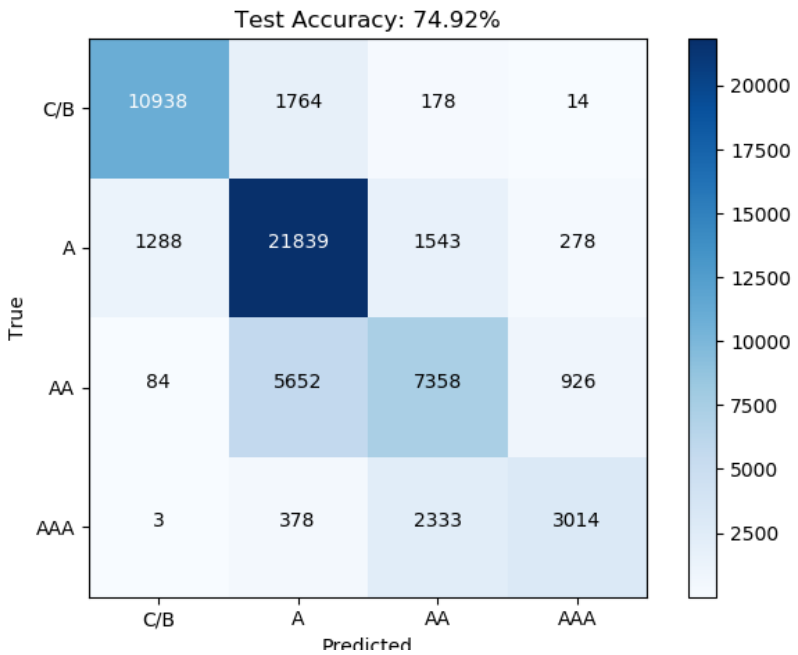
After training is complete, we get the confusion matrix on validation data in Figure 13. Comparing this to the corresponding matrix for the genetic algorithm's top performer in Figure 7, we see a slight increase in accuracy on validation data. On the one hand this suggests that the genetic algorithm works very well in identifying successful architectures. On the other hand, it also means that we can still achieve better results through manual fine-tuning of both architecture and hyperparameters. Interestingly, the source of the slight gain in accuracy seems to stem from the algorithm being quite a bit better at separating between 'A' and 'AA' ratings, but at the cost of more mistakes between 'B/C' and 'A' - although disproportionately so, hence the gain in accuracy.

Figure 13: Final Model Validation Data Confusion Matrix



The real test of the model is however its performance on the test data set, shown in Figure 14. Since our 'validation' and 'test' sets in our case were really two out-of-sample datasets, one would perhaps expect equal model performance on both - with some random variation. Performance on test data is barely two percentage points lower, but this could partially be due to indirect overfitting on validation data; the genetic algorithm trains, evaluates and picks networks for survival based on performance on the validation data. As such, we would in fact expect test accuracy to be a little lower as the deep learning architecture is, potentially, tailored to the validation data. Correspondingly, whereas the model's categorical accuracy is 77.71% on the validation data, it is 74.92% on the test data.

Figure 14: Final Model Test Data Confusion Matrix



Although the 74.92% performance is lower than on the validation data, the confusion matrix in Figure 14 arguably shows preferential patterns of misclassification; from an economic point of view, the impact of misclassifying a not credit worthy firm as being credit worthy - or vice versa - is higher than giving an incorrect rating within the correct such group. In that regard, the model is excellent at dividing between the two, achieving 94.22% accuracy if the classification problem is considered binary, compared to the 93.87% accuracy of the genetic top model on the validation data. The portion of true 'C/B' firms falsely identified as being creditworthy is 15.2%, whereas the portion of creditworthy firms falsely given a 'C/B' rating is only 3.08%. The discrepancy between the two directions can partially be attributed to the imbalanced data, as the model has far fewer 'C/B' observations from which to infer their defining characteristics.

Compared to the genetic top performing model, this final adjusted model is therefore more accurate on the whole, and particularly where the economic impact is largest. In fact, 76.93% of errors are within the creditworthy group, not between credit worthy and not creditworthy ratings. From a practical point of view, this is also where one could reasonably expect a greater element of subjectivity on the side of the credit rating agency. Identifying nuances between the different creditworthy ratings - especially the higher ones - may include a greater element of personal judgment calls, information which the data could in that case not accurately convey.

Conclusion

Deep learning appears to be a feasible approach to predicting corporate credit ratings. Actually setting a rating must, as of today, be performed by a professional, but this paper shows that deep learning is a feasible tool for providing an initial objective suggestion. Towards this end, the genetic algorithm does a great job of identifying a set of traits for successful network architectures, even with a relatively small population and low number of generations. In fact, manual improvements beyond what the genetic algorithm identifies yields improvements in performance, but only at the level of single-digit percentage points. Based on the architectures of the genetic algorithm's top five networks, as measured on the validation data, we manually designed a network architecture. We also adjusted other hyperparameters like ADAM's learning rate and decay, and arrived at a relatively accurate model for predicting credit scores 'C/B', 'A', 'AA', and 'AAA'.

On this multi-class problem, the final deep learning model achieves 74.92% categorical accuracy on the test data. This is lower than the 77.71% accuracy on validation data, which may in part be because the genetic algorithm selects networks for survival based on validation data accuracy, such that we may be indirectly overfitting network architecture to that data. Interestingly, 76.93% of errors are between the different creditworthy ratings, and 86.83% of true 'C/B' companies are given the correct rating. If, instead, the problem is to simply divide firms into creditworthy and not credit worthy groups, the final deep learning model is 94.22% accurate on the test data. As such, the model largely avoids errors with the highest practical economic impact, namely giving a creditworthy rating to a firm that isn't.

However, the random forest benchmark does significantly better. It achieves 91.51% accuracy when dividing into individual credit ratings, and a staggering 97.03% accuracy on the binary problem between creditworthy or not. It also gives the correct rating to 97.38% of true 'C/B' firms, essentially avoiding the problem of giving a creditworthy rating to unworthy firms entirely. As such, though the deep learning model may arguably be feasible for credit rating, it clearly loses to random forest.

It is worth mentioning that other deep learning approaches may work better. As such, we would like to suggest some guidelines for further research. In our time-agnostic model, we do not incorporate time series or a company's previous ratings. For instance, by considering last year's rating or trends in debt-to-equity ratios or profitability, one may conceivably boost prediction accuracy considerably. Finally, introducing architectures different than multilayer perceptrons could also capture nuances in the data that our model is unable to. Notable suggestions could be Convolution or a form of Recurrence, such as Long-Short-Term Memory (LSTM).

References

Balios, D., Thomadakis, S., & Tsipouri, L. (2016). Credit rating model development: An ordered analysis based on accounting data. *Credit rating model development: An ordered analysis based on accounting data*, *38*, 122–136.

Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, *39*(3), 930–945.

Bisnode. (2016). D&B business report for Signicat AS. Retrieved from `https://www.signicat.com/wp-content/financial-reports/bisnode-credit-rating-EN.pdf`

Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade* (pp. 421–436). Springer.

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Dahl, G. E., Sainath, T. N., & Hinton, G. E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on* (pp. 8609–8613).

Doumpos, M., Niklis, D., Zopounidis, C., & Andriosopoulos, K. (2015). Combining accounting data and a structural model for predicting credit ratings: Empirical evidence from european listed firms. *Journal of Banking Finance*, *50*, 599–607.

Fracassi, C., Petry, S., & Tate, G. (2016). Does rating analyst subjectivity affect corporate debt pricing? *Journal of Financial Economics*, *120*, 514–538.

Géron, A. (2017). *Hands-on machine learning with scikit-learn and tensorflow: concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Sebastopol.

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 315–323).

Goldberg, Y. (2016). A primer on neural network models for natural language processing. *J. Artif. Intell. Res.(JAIR)*, *57*, 345–420.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* MIT press.

Hajek, P., & Michalak, K. (2013). Feature selection in corporate credit rating prediction. *Knowledge-Based Systems*, *51*, 72–84.

Harvey, M. (2017). *Let's evolve a neural network with a genetic algorithm.* `https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164`. (Accessed: 2017-10-31)

Hastie, T., & Tibshirani, R. (2014). *Tree-based methods.* `https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearning/asset/trees.pdf`. (Accessed: 2017-11-01)

Hwang, R.-C., Chung, H., & Chu, C. (2010). Predicting issuer credit ratings using a semiparametric method. *Journal of Empirical Finance*, *17*, 120–137.

Hájek, P. (2012). Credit rating analysis using adaptive fuzzy rule-based systems: an industry-specific approach.

Hájek, P., & Olej, V. (2011). Credit rating modelling by kernel-based approaches with supervised and semi-supervised learning. *Neural Comput & Applic*, *20*, 761–773.

Hájek, P., & Olej, V. (2016). Predicting firms' credit ratings using ensembles of artificial immune systems and machine learning - an over-sampling approach.

Johnson, R., & Zhang, T. (2014). Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*.

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

Larson, W. (2009). *Genetic algorithms: Cool name damn simple.* `https://lethain.com/genetic-algorithms-cool-name-damn-simple/`. (Accessed: 2017-10-31)

LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9–48). Springer.

Nanni, L., & Lumini, A. (2009). An experimental comparison of ensemble of classifiers for bankruptcy prediction and credit scoring. *Expert Systems with Applications*, *36*, 3028–3033.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady an ussr* (Vol. 269, pp. 543–547).

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Shaikh, F. (2017). *Why are gpus necessary for training deep learning models?* `https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/`. (Accessed: 2017-11-07)

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, *15*(1), 1929–1958.

Terovitis, S. (2017). *The impact of credit ratings on capital markets.* `https://www2.warwick.ac.uk/fac/soc/economics/staff/sterovitis/cra_spyros_terovitis.pdf`. (Accessed: 2017-11-04)

Touretzky, D. S., & Pomerleau, D. A. (1989). What's hidden in the hidden layers. *Byte*, *14*(8), 227–233.

List of Tables

List of Figures

Appendix A: List of Abbreviations

Table 4: List of Abbreviations

Abbreviation	**Explanation**
ADAM	Adaptive Moment Estimation
AWS	Amazon Web Service
CPU	Central Processing Unit
GPU	Graphics Processing Unit
Inf	Infinity entry in R-studio
LSTM	Long-Short-Term Memory
MLP	Multilayer perceptron
NA	Not available entry in R-studio
NCA	Norwegian Corporate Account
NHH	Norwegian School of Economics
ReLU	Rectified Linear Unit
RF	Random forest
SNF	Centre For Applied Research at NHH

Appendix B: R Code

Creating Mother Data

Creating Mother data for feature building.

```
rm(list = ls())
# Creating Data Directory
suppressWarnings(dir.create("data/"))

print("Loading nca-dataset")
```

```
## [1] "Loading nca-dataset"
```

```
suppressWarnings(load("nca.Rdata"))
# Look at the data:
paste("Dimensions of NCA data-set", dim(nca))
# Create the credit rating dataset We need data from 2005 and
# onwards:
nca.cr <- nca[nca$aar > 2004, ]
# Getting an understanding of the amount of missing data The
# complete.cases() function tells us which rows have no
# missing values:
print(paste("Rows with missing values", round(mean(complete.cases(nca.cr)),
    5), "%"))
# We can use the sapply() function to count the number of
# missing in each row.  We sort and round it to three digits
# to make it more convenient:
missing.magnitude <- round(sort(sapply(nca.cr, function(x) mean(is.na(x)))),
    digits = 3)
# Remove brkod2, brkod3 and brtxt2:
nca.cr$brkod3 <- NULL
nca.cr$brkod2 <- NULL
nca.cr$brtxt2 <- NULL
# Calculate betskattesats and betskattes using formulas from
# the working paper:
nca.cr$betskattes <- nca.cr$betsk/nca.cr$resfs
nca.cr$betskattesats <- nca.cr$sumskatt/nca.cr$resfs
# When it comes to styrehon, daglonn, revhon and ansatte,
# they are all missing 10% of the time.  We exclude them:
nca.cr$styrehon <- NULL
nca.cr$daglonn <- NULL
```

```
nca.cr$revhon <- NULL
nca.cr$ansatte <- NULL
# Calculate ebitdamarg using formula from the working paper:
nca.cr$ebitdamarg <- nca.cr$ebitda/nca.cr$totinn
# Finally, we have the industry codes.  We remove them as we
# also have the sector variable:
nca.cr$bransjek_02 <- NULL
nca.cr$bransjek_07 <- NULL
nca.cr$bransjegr_02 <- NULL
nca.cr$bransjegr_07 <- NULL
nca.cr$bransjet_02 <- NULL
nca.cr$bransjet_07 <- NULL
# Remove the location data:
nca.cr$postnr <- NULL
nca.cr$kommnr <- NULL
nca.cr$poststed <- NULL
# Transform regorg and revorg to dummy variables:
nca.cr$regorg <- as.numeric(nca.cr$regorg)
nca.cr$revorg <- as.numeric(nca.cr$revorg)
nca.cr$regorg[is.na(nca.cr$regorg)] <- 0
nca.cr$revorg[is.na(nca.cr$revorg)] <- 0
nca.cr$regorg <- ifelse(nca.cr$regorg > 0, 1, 0)
nca.cr$revorg <- ifelse(nca.cr$revorg > 0, 1, 0)
# Fix some other binary variables with NAs:
nca.cr$utbpay[is.na(nca.cr$utbpay)] <- 0
nca.cr$stled_skift[is.na(nca.cr$stled_skift)] <- 0
nca.cr$dagl_skift[is.na(nca.cr$dagl_skift)] <- 0
# Fix the Oslo Stock Exchange dummy:
nca.cr$ose_dummy <- nca.cr$bors_aks + nca.cr$bors_obl
nca.cr$ose_dummy <- nca.cr$ose_dummy + nca.cr$bors_sert
nca.cr$ose_dummy <- nca.cr$ose_dummy + nca.cr$bors_opt
nca.cr$ose_dummy <- ifelse(nca.cr$ose_dummy > 0, 1, 0)
nca.cr$bors_aks <- NULL
nca.cr$bors_obl <- NULL
nca.cr$bors_sert <- NULL
nca.cr$bors_opt <- NULL
print("Numeric factors")

## [1] "Numeric factors"

# Fix multilevel factor variables:
```

```
levels(nca.cr$eierstruktur) <- 0:8
nca.cr$eierstruktur <- as.numeric(nca.cr$eierstruktur) - 1
nca.cr$eierstruktur[is.na(nca.cr$eierstruktur)] <- 8
levels(nca.cr$sector) <- 0:9
nca.cr$sector <- as.numeric(nca.cr$sector) - 1
nca.cr$sector[is.na(nca.cr$sector)] <- 9
# Remove the variable 'konkaar' as we're predicting ex ante:
nca.cr$konkaar <- NULL
## We remove all NAs from this point before we add
## transformations of the variables:
print("Omitting initial NA's")
```

```
## [1] "Omitting initial NA's"
```

```
nca.cr <- na.omit(nca.cr)
## Creating Company Age variable:
nca.cr$comp_age <- nca.cr$aar - nca.cr$stiftaar
nca.cr$stiftaar <- NULL
print("Step 2: Creating features...")
```

```
## [1] "Step 2: Creating features..."
```

```
# Net income (NI):
nca.cr$ni <- nca.cr$ordres
nca.cr$ordres <- NULL
# Equity (EQ):
nca.cr$eq <- nca.cr$ek
nca.cr$ek <- NULL
# Total assets (TA):
nca.cr$ta <- nca.cr$sumeiend
nca.cr$sumeiend <- NULL
# Total debt (TD):
nca.cr$td <- nca.cr$gjeld
nca.cr$gjeld <- NULL
# Total capital (TC):
nca.cr$tc <- nca.cr$sumgjek
nca.cr$sumgjek <- NULL
# Sales:
nca.cr$sales <- nca.cr$salgsinn
nca.cr$salgsinn <- NULL
# Retained earnings:
nca.cr$re <- nca.cr$opptjek
nca.cr$opptjek <- NULL
```

```r
# Depreciation:
nca.cr$dep <- nca.cr$avskr
nca.cr$avskr <- NULL
# Tax rate:
nca.cr$etr <- nca.cr$betskattesats
nca.cr$betskattesats <- NULL
# Dividends:
nca.cr$div <- nca.cr$utb
nca.cr$utb <- NULL
# Interest (Int):
nca.cr$int <- nca.cr$rentekost
nca.cr$rentekost <- NULL
print("Profitability")
```

```
## [1] "Profitability"
```

```r
# EBIT:
nca.cr$ebit <- nca.cr$ebitda - nca.cr$dep - nca.cr$nedskr
# Return on total assets (ROA):
nca.cr$roa <- nca.cr$ni/nca.cr$ta
# Return on equity (ROE):
nca.cr$roe <- nca.cr$ni/nca.cr$eq
# Return on capital (ROC):
nca.cr$roc <- (nca.cr$ni - nca.cr$div)/nca.cr$tc
# Retained earnings to total assets:
nca.cr$re_ta <- nca.cr$re/nca.cr$ta
print("Activity")
```

```
## [1] "Activity"
```

```r
# Sales to net worth:
nca.cr$sales_nw <- nca.cr$sales/nca.cr$eq
# Sales to total assets (asset turnover):
nca.cr$sales_ta <- nca.cr$sales/nca.cr$ta
# Operating revenue (or) to total assets:
nca.cr$or_ta <- nca.cr$totinn/nca.cr$ta
# Working capital (WC):
nca.cr$wc <- nca.cr$oml - nca.cr$kgjeld
print("Asset structure")
```

```
## [1] "Asset structure"
```

```r
# Working capital to total assets:
nca.cr$wc_ta <- nca.cr$wc/nca.cr$ta
```

```r
# Fixed assets to total assets:
nca.cr$fa_ta <- nca.cr$anl/nca.cr$ta
# Intangible assets to total assets:
nca.cr$ia_ta <- nca.cr$immeiend/nca.cr$ta
# Total assets (TA) to equity (EQ):
nca.cr$ta_eq <- nca.cr$ta/nca.cr$eq
# log of total assets (log(TA)):
nca.cr$log_ta <- log(nca.cr$ta)
print("Liquidity")
```

```
## [1] "Liquidity"
```

```r
# Cash flow (CF) - using method on page 25 in the NCA working
# paper:
nca.cr$cf <- nca.cr$invest + nca.cr$cash
# Non-cash working capital (NCWC):
nca.cr$ncwc <- nca.cr$wc - nca.cr$cash
print("Leverage")
```

```
## [1] "Leverage"
```

```r
# (Book) debt to total capital:
nca.cr$td_tc <- nca.cr$td/nca.cr$tc
# Long-term debt (LD) to total capital (TC):
nca.cr$ld_tc <- nca.cr$lgjeld/nca.cr$tc
# Short-term debt (SD) to total capital (TC):
nca.cr$sd_tc <- nca.cr$kgjeld/nca.cr$tc
## Infs and NaNs to NA
print("Infs and NaNs to NA")
```

```
## [1] "Infs and NaNs to NA"
```

```r
is.na(nca.cr) <- sapply(nca.cr, is.infinite)
is.na(nca.cr) <- sapply(nca.cr, is.nan)
nca.cr$ratingkode <- factor(nca.cr$ratingkode)
nca.cr$creditworthy <- ifelse(nca.cr$ratingkode == "AAA", 3,
    ifelse(nca.cr$ratingkode == "AA", 2, ifelse(nca.cr$ratingkode ==
        "A", 1, ifelse(nca.cr$ratingkode %in% c("B", "C"), 0,
        NA))))
nca.cr$creditworthy <- as.numeric(nca.cr$creditworthy)
nca.cr$ratingkode <- NULL
## Get rid of observations with no credit-rate
no.rating <- which(is.na(nca.cr$creditworthy))
nca.cr <- nca.cr[-no.rating, ]
```

```
## Save
print("Saving file")
```

```
## [1] "Saving file"
```

```
save(nca.cr, file = "data/Mother.Rdata")
```

Creating Feature Data Frame & Matrix

```
rm(list = ls())
setwd("/Users/BenjaminAanes/")
# Grep mother file
files <- paste0("data/", list.files("data")[grep("Mother",

                                        list.files("data"))])
# Random Forest
# load file
print("loading file")
```

```
## [1] "loading file"
```

```
suppressWarnings(load(files[1]))
# Removing organization number, name and current year
nca.cr <- nca.cr[, -c(1:3)]
print("removing rows with more than 20% NAs")
```

```
## [1] "removing rows with more than 20% NAs"
```

```
print("Sampling data")
```

```
## [1] "Sampling data"
```

```
#Set seed for sampling for reproducability
set.seed(3333)
# Pick random sample from data-population
scaled.sample <- nca.cr[sample(1:nrow(nca.cr)), ]
# Get nrows
x <- nrow(scaled.sample)
# Training set 70\& split
train <- floor(x * 0.7)
# Validation set 15\% split
valid <- train + floor(x * 0.15)
# Test set: Remaining 15\%
# Create split vector
nums <- c(0, train, valid, test)
```

```
# Create list
split <- list()
print("train/test split")

## [1] "train/test split"

# Split data set and print interval
for(i in 1:3){
  split[[i]] <- scaled.sample[(nums[i] + 1):nums[i+1], ]
  print(paste("Interval between", nums[i]+1, ":", nums[i+1]))
}
# For each data set, set NA to zero
split.df <- split
for(i in 1:3){
  split.df[[i]][is.na(split.df[[i]])] <- 0
}
#"Saving split data frame and split list"
suppressWarnings(save(split, file = paste("data/feature.Rdata")))
suppressWarnings(save(split.df, file = paste("data/feature.RF.Rdata")))
split.df.std <- list()
# Isolating binary, factor and normalized variables(askj_hhi)
onehots <- which(colnames(nca.cr) %in% c("eierstruktur", "aksj_hhi",
                                "stled_skift", "dagl_skift",
                               "ose_dummy", "revorg", "revorg-skift",
                             "regorg", "regorg_skift", "ifrs", "sector"))
# Standardize train/test split data frame
# For each data set get input and output variables, isolate
# binary, factor, standardize inputs w/o binary/factor
# ignoring all NA's to create "interaction terms",
# combine binaries/factors and standardized inputs
# set remaining NAs to zero for interaction effect
# put into list

for(i in 1:length(split)){
  x <- split[[i]][ , -ncol(split[[i]])]
  creditworthy <- split[[i]][ , ncol(split[[i]])]
  onehots.hold <- x[, onehots]
  x.z <- apply(x[, -onehots],
                    MARGIN = 2, function(v)
                            (v - mean(v, na.rm = TRUE))/sd(v, na.rm = TRUE))
   x.z <- as.data.frame(x.z)
```

```
                                        #bind together
    final.x <- data.frame(x.z, onehots.hold)
    final.x[is.na(final.x)] <- 0
    final.x <- data.frame(final.x, creditworthy)
 split.df.std[[i]] <- final.x
}
# Saving standardized RF dataset
suppressWarnings(save(split.df.std,
                      file = paste0("data/feature.RF.std.Rdata")))
library(keras)
# Deep learning feature matrix
# Standardizing matrix and one-hot encoding factors
# Same as above, but isolate factors and one hot before combining
split.k <- list()
for(i in 1:length(split)){
  split.k[[i]] <- as.matrix(split[[i]])
}
  x <- list()
  y <- list()
  onehots.hold <- list()
  bins <- list()
  factors <- list()
  onehot <- list()
  x.z <- list()
  final.x <- list()
  y.onehot <- list()
  print("Finding factors/dummies")

## [1] "Finding factors/dummies"

  onehots <- which(colnames(split[[1]]) %in% c("eierstruktur", "aksj_hhi",
                                        "stled_skift", "dagl_skift",
                                        "ose_dummy", "revorg",
                                        "revorg_skift", "regorg",
                                        "regorg_skift", "ifrs", "sector"))
  for(i in 1:length(split.k)){
    # split.k inputs
    x[[i]] <- split.k[[i]][ , -ncol(split.k[[i]])]
    #split.k outputs
    y[[i]] <- split.k[[i]][ , ncol(split.k[[i]])]
    #split.k by type
```

```
    onehots.hold[[i]] <- x[[i]][, onehots]
    onehots.hold
    # split.k bins
    bins[[i]] <- onehots.hold[[i]][, c(2:8, 10)]
    #split.k factors
    factors[[i]] <- onehots.hold[[i]][, c(1, 9)]
    #one hot encode factors
    onehot[[i]] <- cbind(to_categorical(factors[[i]][, 1]),
                         to_categorical(factors[[i]][, 2]))
    # standardize not onehots without NAs
    x.z[[i]] <- apply(x[[i]][, -onehots],
                      MARGIN = 2, function(v)
                        (v - mean(v, na.rm = TRUE))/sd(v, na.rm = TRUE))
    #bind together
    final.x[[i]] <- cbind(x.z[[i]], bins[[i]], onehot[[i]])
    #kick NAs to zero
    # kick NAs to zero in level nca.cr
    final.x[[i]][is.na(final.x[[i]])] <- 0
    y.onehot[[i]] <- to_categorical(y[[i]])
  }
# Create file names and file paths
text <- c("Training Matrix", "Validation Matrix", "Test Matrix")
files <- c("train.csv", "valid.csv", "test.csv")
# Saving to .csv
for(i in 1:length(split.k)){
    print(paste("Writing", text[i], "to csv"))
    suppressWarnings(write.csv(cbind(final.x[[i]],
                                y.onehot[[i]]), row.names = FALSE,
                                    sep = ";", file = files[i]))
}

## [1] "Writing Training Matrix to csv"

# Show each data-set distribution
## Keras data fixings
# Distribution
for(i in 1:length(split.k)){
  print("Unique values")
  print(unique(split.k[[i]][, ncol(split.k[[i]])]))
  print("Distribution between groups")
  print(table(split.k[[i]][, ncol(split.k[[i]])]))
```

```
                                /sum(table(split.k[[i]][,
                                ncol(split.k[[i]])))))
  print(table(split.k[[1]]))
}
## [1] "Unique values"
```

Guessing last year's rating as this year's rating

```
## Guessing last year as this year (with ratings B and C merged)

# Clean
rm(list=ls())

# Load data and packages
suppressWarnings(load('../nca.Rdata'))
library(plyr)

# Please no NA's in my ratingkodes
completeFun <- function(data, desiredCols) {
  completeVec <- complete.cases(data[, desiredCols])
  return(data[completeVec, ])
}

nca.complete <- completeFun(nca, 'ratingkode')
# Off you go, 'Ikke ratet' and 'Konk/avv/likv'
levels(nca.complete$ratingkode)
nca.complete <- nca.complete[!nca.complete$ratingkode == "Ikke ratet", ]
nca.complete <- nca.complete[!nca.complete$ratingkode == "Konk/avv/likv", ]
# Need only year, id, ratings
h <- data.frame(nca.complete$aar, nca.complete$orgnr, nca.complete$ratingkode)
names(h) <- c('aar', 'orgnr', 'ratingkode')
# Sort data for ddply
#h£ratingkode <- as.factor(h£ratingkode)
h <- h[with(h, order(orgnr, aar)), ]
h$creditworthy <- ifelse(h$ratingkode == "AAA", 3,
                         ifelse(h$ratingkode == "AA", 2,
                                ifelse(h$ratingkode == "A", 1,
                                       ifelse(h$ratingkode %in% c("B", "C"), 0, NA))))
head(h)
```

```
# Lagged creditworthy
h <- ddply(h, .(orgnr), transform,
                      creditworthy.t_1 =

                        c(NA, creditworthy[-length(creditworthy)] )
)
# Remove induced NA's
h <- na.omit(h)
# By subtracting t from t-1, should get 0 if same rating
h$pred <- h$creditworthy.t_1 - h$creditworthy
# Generate indicator such that if prediction is correct, "accurate" = 1, else "accurate" = 0
h$accurate <- ifelse(h$pred == 0, 1, 0)
# Now calculate
table(h$accurate)
# Accuracy
305661/(144668+305661)*100
```

```
## [1] 67.87504
```

```
## 67.88% accuracy
```

Random Forest

Random Forest model parallel process script for use on AWS RStudio server.

```
######################################## BENCHMARK: RANDOM FOREST #


rm(list = ls())
setwd("/home/rstudio/")
# Load libraries
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
library(doMC)
library(doSNOW)
library(e1071)
library(extrafont)
library(foreach)
library(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package:  'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin

library(ggplot2)
library(ggthemes)
loadfonts()
# Parallel processing
cores <- 32
registerDoMC(cores = cores)
registerDoSNOW(makeCluster(32, type = "SOCK"))
# Load data
suppressWarnings(load("feature.Rf.Rdata"))
# Training and test set from list
training.data <- split.df[[1]]
test.data <- split.df[[3]]
# Generating smaller train and test sets
training.tiny <- training.data[sample(nrow(training.data), 40000),
    ]
testing.tiny <- test.data[sample(nrow(test.data), 10000), ]
# Finding optimal mtry by tuning on small training set
rf <- train(as.factor(creditworthy) ~ ., data = training.tiny,
    method = "rf")
## Evaluate results
pred.rf <- predict(rf, newdata = testing.tiny)
confusionMatrix(testing.tiny$creditworthy, pred.rf)
### Test set accuracy: 90.31% Optimal mtry: 84
mtry <- 84


# Random forest models with optimal mtry #

# Run on full set with optimal mtry

# 32 trees
model.32 <- foreach(ntree = rep(1, cores), .combine = combine,
    .packages = "randomForest") %dopar% randomForest(as.factor(creditworthy) ~
```

```
    ., data = training.data, ntree = ntree, mtry = mtry, importance = TRUE)
## Warning:  executing %dopar% sequentially:  no parallel backend registered
pred.model.32 <- predict(model.32, newdata = test.data)
confusionMatrix(test.data$creditworthy, pred.model.32)
## Accuracy: 91.51%

# 64 trees
model.64 <- foreach(ntree = rep(2, cores), .combine = combine,
    .packages = "randomForest") %dopar% randomForest(as.factor(creditworthy) ~
    ., data = training.data, ntree = ntree, mtry = mtry, importance = TRUE)
pred.model.64 <- predict(model.64, newdata = test.data)
confusionMatrix(test.data$creditworthy, pred.model.64)
## Accuracy: 91.6%

# 128 trees
model.128 <- foreach(ntree = rep(4, cores), .combine = combine,
    .packages = "randomForest") %dopar% randomForest(as.factor(creditworthy) ~
    ., data = training.data, ntree = ntree, mtry = mtry, importance = TRUE)
pred.model.128 <- predict(model.128, newdata = test.data)
confusionMatrix(test.data$creditworthy, pred.model.128)
## Accuracy: 91.74%

# 512 trees
model.512 <- foreach(ntree = rep(16, cores), .combine = combine,
    .packages = "randomForest") %dopar% randomForest(as.factor(creditworthy) ~
    ., data = training.data, ntree = ntree, mtry = mtry, importance = TRUE)
pred.model.512 <- predict(model.512, newdata = test.data)
confusionMatrix(test.data$creditworthy, pred.model.512)
## Accuracy: 91.8%

save(model.512, file = "model512.Rda")
```

Plotting best Random Forest Model

```
##################################### # Plotting best model # #

rm(list = ls())
setwd("/Users/Eivind/.ssh/aws")
# Load libraries
```

```
library(caret)
library(extrafont)
library(ggplot2)
library(e1071)
library(randomForest)

loadfonts()
# Load data
suppressWarnings(load("feature.Rf.Rdata"))
# Load best model from AWS
suppressWarnings(load("/Users/Eivind/.ssh/aws/model512.Rda"))
# Test set
test.data <- split.df[[3]]
# Generate predictions and confusion matrix
pred <- predict(model.512, newdata = test.data)
confusion <- confusionMatrix(test.data$creditworthy, pred)  # 91.81% accuracy
conf.frame <- as.data.frame(confusion$table)
conf.frame$Reference.Rev <- with(conf.frame, factor(Reference,
    levels = (rev(levels(Reference)))))
pdf("model512_cm.pdf", width = 6, height = 5)
ggplot(data = conf.frame, mapping = aes(x = Prediction, y = Reference.Rev)) +
    xlab("Predicted") + ylab("True") + geom_tile(aes(fill = Freq),
    colour = "white") + geom_text(aes(label = sprintf("%1.0f",
    Freq)), vjust = 1, family = "CM Roman") + scale_fill_gradient(low = "white",
    high = "steelblue", name = "Frequency") + theme_bw() + scale_x_discrete(labels = c(`0` = "C+B",
    `1` = "A", `2` = "AA", `3` = "AAA")) + scale_y_discrete(labels = c(`0` = "C+B",
    `1` = "A", `2` = "AA", `3` = "AAA")) + theme_tufte(base_family = "CM Roman")
dev.off()

## pdf
##   2

embed_fonts("model512_cm.pdf", outfile = "model512_cm_embed.pdf")
```

Appendix C: Genetic Algorithm in Python

```
# == IMPORTING LIBRARIES ==
# Load Keras
import keras
import tensorflow as tf
import pandas as pd
import numpy as np
import pylab as pl
import matplotlib.pyplot as plt
import itertools
import random
import logging

from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.constraints import maxnorm
from keras.optimizers import SGD, Nadam, Adam
from keras.utils import plot_model, to_categorical
from keras.callbacks import EarlyStopping
from keras import backend
from keras import regularizers

from sklearn import preprocessing
from sklearn.metrics import confusion_matrix

from functools import reduce
from operator import add
from tqdm import tqdm

# Version 1.1.0 means it's using the GPU
print('Tensorflow:', tf.__version__)

# Necessary to get GPU memory to work
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.Session(config=config)

print('Keras:', keras.__version__)

# == LOADING DATA ==
```

```
# Data has already been separated into train, validation, and test sets
print('Loading train data...')
d_train = np.genfromtxt('4_train.csv', delimiter=',', skip_header=1)
d_train = np.array(d_train, dtype=np.float32)
# All indices in dataset
all_indices = np.array(range(d_train.shape[1]))
print('Done.')

print('Loading validation data...')
d_valid = np.genfromtxt('4_valid.csv', delimiter=',', skip_header=1)
d_valid = np.array(d_valid, dtype=np.float32)
print('Done.')

print('Loading test data...')
d_test = np.genfromtxt('4_test.csv', delimiter=',', skip_header=1)
d_test = np.array(d_test, dtype=np.float32)
print('Done.')

# Number of one-hot encoded labels (i.e. 1 label with 5 classes = 5)
num_cats = 4

# Splits a given data set into inputs, labels -->
# assuming latter on the far-right side, and one-hot encoded
def data_reshape(data, cats=num_cats):
    return data[:, 0:-cats], data[:, -cats:]

train_inputs, train_labels = data_reshape(d_train)
print('Training:', train_inputs.shape, train_labels.shape)

valid_inputs, valid_labels = data_reshape(d_valid)
print('Validation:', valid_inputs.shape, valid_labels.shape)
test_inputs, test_labels = data_reshape(d_test)
print('Test:', test_inputs.shape, test_labels.shape)

# Number of features
num_inputs = train_inputs.shape[1]

# == DEFINING THE GENETIC ALGORITHM ==
# Architecture parameters the algorithm can pick from
param_choices = {
    'nodes': [8, 24, 32, 64, 128, 256, 512, 750, 1024, 1500, 2048, 3000, 4096],
```

```
    'dropout': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7],
    'layers': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
}

# Batch and epoch, albeit with early stopping enabled on training accuracy
batch_size = 2048
epochs = 50

# Genetic algorithm parameters
# Portion of best performers that survive
o_retain = 0.4
# Portion of "worse" performers to randomly select for survival, to capture genetic diversity
o_select = 0.1
# Chance for each child to mutate
o_mutate = 0.2
optimizer = 'adam'
# Number of networks per generation
population_size = 20
generations = 15

# A Layer object represents a single hidden layer for some network,
# and contains the number of nodes, and amount of dropout
class Layer():
    def __init__(self):
        self.nodes = random.choice(param_choices['nodes'])
        self.dropout = random.choice(param_choices['dropout'])

# A Network object represents a full deep learning model, consisting chiefly of a list of
# Layers that comprise its architecture
class Network():
    # Constructor
    def __init__(self):
        # Network validation performance: always initiates to zero (untrained)
        self.accuracy = 0.0
        # Network design, i.e. list of Layer objects
        self.architecture = []

    # Create a random network; used to initialize first generation
    def create_random(self):
        # Pick a random number of hidden layers
        layers = random.choice(param_choices['layers'])
```

```
122             # Populate each hidden layer with a Layer object (random dropout and number of nodes)
123             for i in range(layers):
124                 # Create a random layer
125                 layer = Layer()
126                 # Append layer to network
127                 self.architecture.append(layer)
128 
129         # Train the network
130         def train(self):
131             # No point running train and validation again, i.o.w. if network is a surviving parent
132             if self.accuracy == 0.0:
133                 self.accuracy = train_and_score(self.architecture)
134 
135         # Create a network from a list of layers, rather than randomly
136         def create_set(self, architecture):
137             self.architecture = architecture
138 
139         # Prints a network architecture, both to a log file and to the console
140         def print_network(self):
141             logging.info("Network accuracy: %.2f%%" % (self.accuracy * 100))
142             print("Network accuracy: %.2f%%" % (self.accuracy * 100))
143             for layer in self.architecture:
144                 logging.info("Nodes: %d / dropout: %.2f" % (layer.nodes, layer.dropout))
145 
146 # An Optimizer object controls the initialization and evolution of generations,
147 # and is really just for handling
148 class Optimizer():
149     def __init__(self, retain=0.4, random_select=0.1, mutate_chance=0.2):
150         # Portion of networks to retain for use as parents for next generation
151         self.retain = retain
152         # Probability of a worse-performing network being randomly selected for survival
153         self.random_select = random_select
154         # Probability of each child receiving a mutation
155         self.mutate_chance = mutate_chance
156 
157     # Create a population of random networks
158     def create_population(self, count):
159         # Population is a list of Networks
160         pop = []
161         for _ in range(count):
162             # Initialize a new network object
```

```
163             network = Network()
164             # Populate the network with random layers
165             network.create_random()
166             # Add the new random network to the population
167             pop.append(network)
168         return pop
169
170     @staticmethod
171     # Get fitness of network, i.e. its accuracy
172     def fitness(network):
173         return network.accuracy
174
175     # Create two children from two parents
176     def breed(self, mother, father):
177         children = []
178         for _ in range(2):
179             # Create empty list, i.e. empty "starting" child network
180             child = []
181
182             # Backfill parents with None up to max layer, to easily create a
183             # pseudo-randomized child
184             while len(mother.architecture) < max(param_choices['layers']):
185                 mother.architecture.append(None)
186             while len(father.architecture) < max(param_choices['layers']):
187                 father.architecture.append(None)
188
189             # Populate child layers; for each hidden layer in mother and/or father Network...
190             for i in range(max(param_choices['layers'])):
191                 # ... randomly select the Layer from either, and add it to the child
192                 # If one parent has no Layer in this iteration,
193                 # the choice is between a Layer and None
194                 child.append(random.choice([mother.architecture[i], father.architecture[i]]))
195
196             # Remove None from parents
197             mother.architecture = list(filter(None, mother.architecture))
198             father.architecture = list(filter(None, father.architecture))
199
200             # Remove None from child, also so it can be mutated easily
201             child = list(filter(None, child))
202
203             # Create a Network object by feeding the resulting list of Layer objects
```

```
204             ## Child accuracy defaults to 0.0, so will be trained later
205             network = Network()
206             network.create_set(child)
207 
208             # Randomly mutate some children; each child draws a number, if below the
209             # Optimizer threshold, it mutates
210             if self.mutate_chance > random.random():
211                 # Return the next random floating point number in the range [0.0, 1.0).
212                 network = self.mutate(network)
213                 # No need to reset accuracy, since child is already a new Network,
214                 # and therefore has accuracy = 0.0
215             # Add the child Network to the list of children, whether or not it was mutated
216             children.append(network)
217         return children
218 
219     # Mutate a network: replace a randomly selected Layer by a new random Layer
220     def mutate(self, network):
221         # Pick a random Layer to mutate
222         mutation = random.choice(range(0, len(network.architecture)))
223         # Create a new random Layer
224         new_layer = Layer()
225         # Replace the randomly selected Layer with the new randomly built Layer
226         network.architecture[mutation] = new_layer
227         return network
228 
229     # Evolve a population of networks
230     def evolve(self, pop):
231         # Get scores for each network
232         graded = [(self.fitness(network), network) for network in pop]
233 
234         # Sort the scores descending so we can pick the x best networks
235         # Keep only the network objects, sorted by performance
236         graded = [x[1] for x in sorted(graded, key=lambda x: x[0], reverse=True)]
237         # Get the number of networks to keep for the next generation
238         retain_num = int(len(graded) * self.retain)
239 
240         # Extract the retain_num number best-performing networks for use as parents
241         parents = graded[:retain_num]
242 
243         # Randomly keep a few of the worse-performing networks, to catch some random quirks
244         for net in graded[retain_num:]:
```

```
245             if self.random_select > random.random():
246                 parents.append(net)
247
248         # Fill remaining spots in new population with new children
249         children = []
250         # While next population is not yet completely refilled
251         while len(children) < len(pop) - len(parents):
252             # Get indices of a random mom and dad
253             f = random.randint(0, len(parents) - 1)
254             m = random.randint(0, len(parents) - 1)
255
256             # No breeding with yourself, for obvious reasons
257             if f != m:
258                 # Get the Network objects based on index in list of parents
259                 father = parents[f]
260                 mother = parents[m]
261
262                 # Breed to create a new child; returns a list of two networks
263                 babies = self.breed(mother, father)
264
265                 # Add the children to the population, making sure it doesn't grow too large
266                 for baby in babies:
267                     if len(children) < len(pop) - len(parents):
268                         children.append(baby)
269
270         # Add children to parents to create resulting new population
271         parents.extend(children)
272         return parents
273
274 # = Helper functions =
275
276 # Returns compiled Keras model, provided an architecture (list of Layers)
277 def compile_architecture(architecture):
278     # Create new Sequential Keras model
279     model = Sequential()
280     # Iterate over all hidden Layers in architecture
281     # For each Layer, extract number of nodes and create a Keras layer with ReLU activation
282     for i in range(len(architecture)):
283         if i == 0:
284             # First hidden layer, so must include number of input variables as
285             # preceding visible layer
```

```
286             model.add(Dense(architecture[i].nodes, activation='relu', input_dim=num_inputs))
287         else:
288             model.add(Dense(architecture[i].nodes, activation='relu'))
289         # Add Dropout to the layer; accepts 0.0, i.e. no Dropout
290         model.add(Dropout(architecture[i].dropout))
291     # Add visible output layer
292     model.add(Dense(num_cats, activation='softmax'))
293 
294     # Compile the model, specifying Cross-Entropy as loss, ADAM as optimizer,
295     # and categorical accuracy as metric
296     model.compile(
297         loss='categorical_crossentropy',
298         optimizer=optimizer,
299         metrics=['categorical_accuracy']
300     )
301     return model
302 
303 def train_and_score(architecture):
304     # Compile Keras model
305     model = compile_architecture(architecture)
306     # Fit on training data
307     model.fit(
308         x=train_inputs,
309         y=train_labels,
310         batch_size=batch_size,
311         epochs=epochs,
312         validation_data=(valid_inputs, valid_labels),
313         verbose=0,
314         callbacks=[EarlyStopping(monitor='categorical_accuracy', min_delta=0.005,
315                    patience=3, verbose=0, mode='auto')]
316     )
317     # Calculate score on validation data
318     score = model.evaluate(valid_inputs, valid_labels, verbose=0)
319     # 0 is loss, 1 is accuracy
320     accuracy = score[1]
321     # Clear TensorFlow memory, since we don't need to store the full TensorFlow
322     # model and all its history
323     backend.clear_session()
324     # Clear the Python state, for the same reason
325     tf.reset_default_graph()
326 
```

```
327     return accuracy
328
329 # Train a list of networks
330 def train_networks(networks):
331     # Create progress bar, as this will take some time
332     pbar = tqdm(total=len(networks))
333     for network in networks:
334         pbar.update(1)
335         # Train each provided network
336         network.train()
337     pbar.close()
338
339 # Calculate average accuracy of a list of networks (i.e. of a population)
340 def get_average_accuracy(networks):
341     total_accuracy = 0
342     for network in networks:
343         total_accuracy += network.accuracy
344     return total_accuracy / len(networks)
345
346 # Function to print architectures of a list of networks
347 def print_networks(networks):
348     logging.info('-' * 80)
349     for network in networks:
350         network.print_network()
351
352 # Generate a network with the above genetic algorithm(s)
353 def generate(generations):
354     # Create Optimizer object to handle execution
355     optimizer = Optimizer(retain=o_retain, random_select=o_select, mutate_chance=o_mutate)
356     # Initialize a randomized starting population
357     networks = optimizer.create_population(population_size)
358     # List of networks and performances, i.e. each element in history is a list of
359     # generations (each a list of Networks, i.e. a population)
360     history = []
361
362     # Evolve the generation
363     for i in range(generations):
364         logging.info("***Doing generation %d of %d***" %
365                      (i + 1, generations))
366         print("\nGeneration:", i + 1)
367
```

```
368         # Train and get accuracy for population of networks
369         train_networks(networks)
370 
371         # Get average accuracy for this population for logging
372         average_accuracy = get_average_accuracy(networks)
373 
374         # Store generation performances and architectures
375         history.append(networks)
376 
377         # Print out the average accuracy each generation.
378         logging.info("Generation average: %.2f%%" % (average_accuracy * 100))
379 
380         # Print out top 3 networks in case it crashes later
381         top = sorted(networks, key=lambda x: x.accuracy, reverse=True)
382         print_networks(top[0:3])
383         logging.info('-' * 80)
384 
385         # Evolve each generation (except the last)
386         if i != generations - 1:
387             # Evolve the population; select top performers (+ some others), breed, mutate
388             networks = optimizer.evolve(networks)
389 
390     # Sort our final population by network performances
391     networks = sorted(networks, key=lambda x: x.accuracy, reverse=True)
392 
393     # Print out the top 5 networks
394     print_networks(networks[:5])
395 
396     return history
397 
398 # == RUNNING THE GENETIC ALGORITHHM ==
399 
400 # Setup logging
401 logging.basicConfig(
402     format='%(asctime)s - %(levelname)s - %(message)s',
403     datefmt='%m/%d/%Y %I:%M:%S %p',
404     level=logging.DEBUG,
405     filename='log.txt'
406 )
407 
408 logging.info("***Evolving %d generations with population %d***" % (generations, population_size))
```

```
409 logging.info("***Batches = %d, Epochs = %d***" % (batch_size, epochs))
410
411 # List of lists of networks; list of generations, each a list of networks
412 evolution = generate(generations)
413
414 # == EXTRACTING AND PLOTTING EVOLUTION AND BEST NETWORKS ==
415
416 # Function to re-train a given Network on the training data,
417 # score performance on validation and test, and build plots
418 def run_survivor(network, b=batch_size, e=epochs, v=0):
419     # Compile the Keras model
420     model = compile_architecture(network.architecture)
421     print('Fitting on training data...')
422     history = model.fit(
423         x=train_inputs,
424         y=train_labels,
425         batch_size=b,
426         epochs=e,
427         validation_data=(valid_inputs, valid_labels),
428         verbose=v
429     )
430     print('Evaluating performance on validation and test data...')
431     perf_valid = model.evaluate(valid_inputs, valid_labels, verbose=0)
432     perf_test = model.evaluate(test_inputs, test_labels, verbose=0)
433     print('Calculating validation and test predictions...')
434
435     print('Plotting...')
436     plot_model_accuracy(history, 1)
437     plot_model_loss(history, 2)
438     plot_model_predictions(model, valid_inputs, valid_labels,
439                            'Validation Accuracy: %2.2f%%' % (100*perf_valid[1]), 3)
440     plot_model_predictions(model, test_inputs, test_labels,
441                            'Test Accuracy: %2.2f%%' % (100*perf_test[1]), 4)
442
443 # Plots the model accuracy on training and validation data
444 def plot_model_accuracy(history, figs):
445     plt.figure(figs)
446     plt.plot(range(1,len(history.history['categorical_accuracy'])+1),
447              history.history['categorical_accuracy'])
448     plt.plot(range(1,len(history.history['val_categorical_accuracy'])+1),
449              history.history['val_categorical_accuracy'])
```

```
450     plt.title('Categorical Accuracy')
451     plt.ylabel('Accuracy')
452     plt.xlabel('Epoch')
453     plt.legend(['training', 'validation'], loc='upper left')
454     plt.show()
455 
456 # Plots model loss on training and validation data
457 def plot_model_loss(history, figs):
458     plt.figure(figs)
459     plt.plot(range(1,len(history.history['loss'])+1), history.history['loss'])
460     plt.plot(range(1,len(history.history['val_loss'])+1), history.history['val_loss'])
461     plt.title('Model Loss')
462     plt.ylabel('Loss')
463     plt.xlabel('Epoch')
464     plt.legend(['training', 'validation'], loc='upper left')
465     plt.show()
466 
467 # Plots confusion matrices for validation and test data
468 def plot_model_predictions(model, inputs, labels, title, figs):
469     probs = model.predict(inputs)
470 
471     preds = np.zeros_like(probs)
472     preds[np.arange(len(probs)), probs.argmax(1)] = 1
473 
474     class_predictions = preds.argmax(1)
475     class_true = labels.argmax(1)
476 
477     # Hardcoded list of labels
478     cats = ['C/B', 'A', 'AA', 'AAA']
479 
480     cm = confusion_matrix(class_true, class_predictions)
481 
482     plt.figure(figs)
483     plt.imshow(cm, interpolation='nearest', cmap=plt.get_cmap('Blues'))
484     plt.title(title)
485     plt.colorbar()
486     tick_marks = np.arange(len(cats))
487     plt.xticks(tick_marks, cats)
488     plt.yticks(tick_marks, cats)
489     thresh = cm.max() / 2.
490     for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
```

```
491         plt.text(j, i, format(cm[i, j], 'd'),
492                  horizontalalignment="center",
493                  color="white" if cm[i, j] > thresh else "black")
494     plt.tight_layout()
495     plt.xlabel('Predicted')
496     plt.ylabel('True')
497     plt.show()
498 
499 # Train and plot performance of top 5 genetic Networks
500 # 1st
501 run_survivor(evolution[generations - 1][0], b=batch_size, e=epochs, v=1)
502 
503 # 2nd
504 run_survivor(evolution[generations - 1][1], b=batch_size, e=epochs, v=1)
505 
506 # 3rd
507 run_survivor(evolution[generations - 1][2], b=batch_size, e=epochs, v=1)
508 
509 # 4th
510 run_survivor(evolution[generations - 1][3], b=batch_size, e=epochs, v=1)
511 
512 # 5th
513 run_survivor(evolution[generations - 1][4], b=batch_size, e=epochs, v=1)
514 
515 # Plot evolution of generational average validation accuracy
516 def plot_evolution(evolution, figs):
517     # For list of lists in evolution (itself a list)
518     generations = range(1, len(evolution)+1)
519     performances = []
520 
521     for generation in evolution:
522         average_accuracy = get_average_accuracy(generation)
523         performances.append(average_accuracy)
524 
525     plt.figure(figs)
526     plt.plot(generations, performances)
527     plt.title('Evolution to %.2f%% in last generation' % (performances[len(evolution)-1] * 100))
528     plt.ylabel('Average Validation Accuracy')
529     plt.xlabel('Generation')
530     plt.show()
531 
```

```
plot_evolution(evolution, 5)

# == DESIGNING OWN MODEL ==
# Close plots
plt.close()
plt.close()
plt.close()
plt.close()
plt.close()

backend.clear_session()
tf.reset_default_graph()

# Function to compile and run a given Keras model, to make iterative tests easier
def run_model(model, optimizer='adam', b=128, e=10, v=0):
    model.compile(optimizer=optimizer, loss='categorical_crossentropy',
                  metrics=['categorical_accuracy'])
    history = model.fit(
        x=train_inputs,
        y=train_labels,
        batch_size=b,
        epochs=e,
        validation_data=(valid_inputs, valid_labels),
        verbose=v,
        callbacks=[EarlyStopping(monitor='categorical_accuracy', min_delta=0.005,
                  patience=5, verbose=0, mode='auto')]
    )
    print('Evaluating performance on validation and test data...')
    perf_valid = model.evaluate(valid_inputs, valid_labels, verbose=0)
    perf_test = model.evaluate(test_inputs, test_labels, verbose=0)
    print('Calculating validation and test predictions...')

    print('Plotting...')
    plot_model_accuracy(history, 1)
    plot_model_loss(history, 2)
    plot_model_predictions(model, valid_inputs, valid_labels,
                           'Validation Accuracy: %2.2f%%' % (100*perf_valid[1]), 3)
    plot_model_predictions(model, test_inputs, test_labels,
                           'Test Accuracy: %2.2f%%' % (100*perf_test[1]), 4)

    return history
```

```
573
574 # Create manual model
575 model = Sequential()
576 model.add(Dense(2048, input_dim=num_inputs, activation='relu', input_shape=(num_inputs, )))
577 model.add(Dropout(0.5))
578 model.add(Dense(4096, activation='relu'))
579 model.add(Dropout(0.3))
580 model.add(Dense(32, activation='relu'))
581 model.add(Dropout(0.0))
582 model.add(Dense(750, activation='relu'))
583 model.add(Dropout(0.5))
584 model.add(Dense(2048, activation='relu'))
585 model.add(Dropout(0.5))
586 model.add(Dense(128, activation='relu'))
587 model.add(Dropout(0.0))
588 model.add(Dense(num_cats, activation='softmax'))
589
590 #sgd = SGD(lr=0.01, momentum=0.7, decay=1e-6, nesterov=True)
591 #nadam = Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, schedule_decay=0.004)
592 adam = Adam(lr=0.0001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=1e-5)
593
594 # Run model and get performance plots
595 hist = run_model(model, optimizer=adam, b=128, e=500, v=1)
```

Appendix D: Literature Review on Variables for Corporate Credit Rating

Table 5: Summary of Literature Review on Corporate Credit Rating Variables

Authors	**Variables Included in Credit Rating Predictions**
Balios, Thomadakis and Tsipouri (2016)	*Liquidity* (current ratio, sales to accounts payable) *Capital structure* (equity to total assets) *Profitability* (net profit margin, consecutive losses) *Operation cost* (operating expenses to net sales)
Hwang, Chung and Chu (2010)	*Market-driven* (monthly return, equity value, volatility of return, KMV-Merton default probability) *Financial leverage* (total assets to equity, long-term debt to capital, total debt to capital, short-term debt to capital, total debt to EBITDA) *Coverage* (EBIT to interest expenses, EBITDA to interest expenses) *Cash flow* (net income from operations, interest expenses, total cash and equivalent) *Profitability* (operating margin, return on capital, return on equity, return on assets, retain earnings to total assets) *Liquidity* (current ratio, quick asset ratio, cash ratio) *Industry indicators*
Doumpos, Niklis, Zopounidis and Andriosopoulos (2015)	*Profitability* (return on assets) *Coverage* (EBIT to interest expenses) *Solvency* (equity to total assets) *Leverage* (equity to long-term debt) *Size* (log of market capitalization) *Country risk indicator*
Hájek and Olej (2016)	*Financial indicators* (enterprise value, cash, revenues, earnings per share, return on equity, price to book value, enterprise value to earnings, price to earnings per share, market debt to total capital, high/low stock price, dividend yield, payout ratio, standard deviation of stock price) *Sentiment indicators* (frequency of negative terms, frequency of positive terms, frequency of uncertainty terms, frequency of litigious terms, frequency of strong modal terms, frequency of weak modal terms) *Investment grade/non-investment grade*
Hájek and Michalak (2013)	*Size of company* (total assets, total capital, sales, 12-month trailing sales, cash flow, equity, enterprise value, capital expenditures, size class, market capitalization, trading volume,

	no. of shares outstanding) *Corporate reputation* (shares held by mutual funds, shares held by insiders) *Profitability ratios* (EBIT, earnings after taxes, net income, 12-month trailing net income, net margin, operating margin, return on assets, return on equity, return on capital, EBITDA, enterprise value to EBITDA, high/low stock price, enterprise value to EBIT, retained earnings to total assets) *Activity ratios* (enterprise value to sales, growth in non-cash working capital, enterprise value to trailing sales, sales to net worth, sales to total assets, operating revenue to total assets, working capital to sales, cash to sales, non-cash working capital to sales) *Asset structure* (fixed assets to total assets, intangible assets to total assets, working capital to total assets, depreciation) *Business situation* (effective tax rate, growth in sales last year, expected growth in sales over next 5 years, SG&A expenditures) *Liquidity ratios* (current ratio, cash ratio, cash to firm value, cash, non-cash working capital) *Leverage* (book value to equity, book debt to total capital, enterprise value to total capital, enterprise value to book value, market cap. to total debt, total debt, cash flow to total debt, market debt to equity, market debt to total capital, net gearing, market debt to EBITDA) *Market value* (3-year stock price variation, beta regression coefficient, value line beta, the correlation of stock returns with market index, dividends, dividends to stock price, earnings per share, growth in earnings per share over 5 years, expected growth in earnings per share over next 5 years, stock price to cash flow, stock price to earnings, 12-month trailing stock price to earnings, forward stock price to earnings, stock price to earnings to earnings per share growth, price to book value ratio, retained earnings, reinvestment rate, pay-out ratio, stock price to sales, stock price)
Bisnode D&B business report – rating of Signicat AS (2016)	*Profitability* (profit margin, interest coverage, return on total capital, return on equity) *Solvency* (equity ratio, loss buffer) *Cash flow* (current ratio, quick ratio, liquid assets in % of turnover, average storage time) *Financing* (long-term stock (inventory) financing, cost of external capital) (Please note that also other factors are taken into account in Bisnode D&B AS's rating model in addition to financial data, such as payment history and ownership structure.)