Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datateknik
2020 | LIU-IDA/LITH-EX-A--20/055--SE

Predicting Risk of Delays in Postal Deliveries with Neural Networks and Gradient Boosting Machines

Predicering av risk för förseningar av leveranser med neurala nätverk och gradient boosting machines

Matilda Söderholm

Supervisor : Héctor Rodríguez Déniz
Examiner : Jose Peña

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

This thesis conducts a study on a data set from the Swedish and Danish postal service Postnord, comparing an artificial neural network (ANN) and a gradient boosting machine (GBM) for predicting delays in package deliveries. The models are evaluated based on F1-score for the important class which represents the data points that are delayed and needed to be identified. The GBM is already implemented and tuned using grid search by Postnord, the ANN is tuned using sequential model based optimization with the tree Parzen estimator function. Furthermore, it is trained using dynamic resampling to handle the imbalanced data set. Even with several measures implemented to handle the class imbalance, the ANN performs poorly when tested on unseen data, unlike the GBM. The GBM has high precision (84%) and decent recall (24%), which produces a F1-score of 0.38. The ANN has high recall (62%) but extremely low precision (5%) which gives a F1-score of 0.08, indicating that it is biased to predict sample as delayed when it is in time. The GBM has a natural handling of class imbalance unlike the ANN, and even with measures taken to improve the ANN and its handling of class imbalance, GBM performs better.

Acknowledgments

I want to thank my supervisor for their dedication, guidance and patience throughout this thesis. I want thank Bontouch for welcoming me into their office and helping me with connections and data sources. Finally, I want to thank my family and friends for their love and support.

Contents

List of Figures

List of Tables

1 Introduction

With the rise of machine learning usage and automation in recent years, new areas to apply data analysis appear frequently. As the main requirement for making machine learning possible is collection of parseable data, machine learning is both helpful and applicable in a wide collection of contexts. A company which has taken great steps of digitalization is PostNord AB, the Swedish and Danish state-owned mail delivery company. With the vast amount of data collected from the millions of packages sent, PostNord truly has ideal circumstances for using machine and deep learning in the logistic processes, both to improve the processes but also allow greater transparency and insights in them.

1.1 Motivation and Aim

Lately the e-commerce business in Sweden has increased with a growth of 16 % during 2017, and is thought to continue increasing; the yearly turn over of Swedish e-commerce in 2017 was 67 billion SEK and the estimated turn over of 2018 is 77 billion SEK[1]. With this growth, package deliveries increase as well. PostNord delivered 97.7 million packages in Sweden in 2017, of which 94.3 % were delivered in a timely manner.[2] Hence 5.6 million packages were delayed, and there is no recorded data of how many packages were delivered earlier than estimated. Since customers in general want a more flexible and transparent delivery process when receiving a package, a more accurate time estimation of arrival is of high importance, especially if the delivery is to be delayed. The aim of this thesis is therefore to implement a machine learning model that sufficiently and accurately can predict delays in postal deliveries based on historical data.

1.2 Research Questions

PostNord is today using a gradient boosting machine to predict if a delivery will be delayed or not, based on meta data collected in the delivery chain. The deliveries in this data is restricted to tracked parcels, which is mostly packages. Gradient boosting machines are naturally well suited for imbalanced data sets, which the data is in this context since a small minority of the packages are delayed. This model will suffice as benchmark in developing an artificial neural network for the same task. Hence, the research questions are:

- Can a neural network be sufficiently used for detecting delays in the PostNord package delivery service with a highly imbalanced data set?
- If so, can it compete with the currently implemented gradient boosting machine?

[1]PostNord in cooperation with Svensk Digital Handel and HUI research. *E-barometern*. 2017. URL: `https://www.postnord.se/siteassets/pdf/rapporter/e-barometern-arsrapport-2017.pdf` (visited on 04/04/2019).

[2]PostNord AB. *Ars- och hallbarhetsredovisning*. 2017.

1.3 Background

This thesis study is conducted at the company Bontouch AB. Bontouch has a B2B business model, where they produce mobile applications for other companies. At the moment they are in cooperation with PostNord, SJ, Swish, SEB, Coca Cola, White Lines among many other companies. Bontouch is specialized in mobile applications, and provides these for clients whose products benefit from a complementing application. Bontouch develops the mobile application for PostNord end customers, where packages can be tracked and an estimation of arrival time is given. PostNord[3] is the leading company providing postal and logistic services in the Nordic countries. Their products and services are used for distribution, e-commerce and logistics by both corporations and individuals.

[3]*PostNord AB*. URL: `https://www.postnord.se` (visited on 04/04/2019).

2 Theory

This section will present relevant information about the PostNord delivery process as well as the machine learning models examined in this study.

2.1 PostNord Delivery Process

PostNord delivers within Sweden, Denmark, Norway and Finland. Each delivery of PostNord goes through their delivery process. When a delivery is registered by the sender, also called consignor, a route to the reciever, also called consignee, is planned. The route is automatically planned by an internal API and database solution which has all optimized routes calculated. This contains a database of all sorting hubs, drop off hubs and pick up hubs and how each hub can be connected to the others through a chain of an arbitrary number of sorting hubs. The route contains all hubs the delivery will go through and at what expected time it will arrive at each hub. The time between two hub timestamps is taken from a set table with standard travel times between all neighboring hubs. After the route is planned, the package is sent accordingly. By using this API, the estimated time of arrival at the pick up hub can be retrieved for a delivery.

Delivery Data

At each hub a delivery reaches it is scanned and therefore an event is added to the meta data of the delivery. Each event throughout delivery make a record in the database as each of these events include information about the postal distribution hub which it is checked into, the event type and other delivery meta data. These events therefore work as checkpoints along the geographical journey of the delivery. Apart from PostNords official system for planning and delivering packages, each country in the region has its own local system for scanning packages. Details of the meta data is presented in section 3.1.

Risk For Delay

In the data set used in this project, aproximately 4% of all trackable deliveries are delayed within PostNord's services based on the initially estimated arrival time. To cope with this, PostNord have developed a gradient boosting machine model, further explained in section 2.3, to detect late deliveries based on single events in the delivery chain. For each event in a delivery, the model will predict whether or not the delivery will arrive in time or not to the consignee.

2.2 Supervised Learning

Supervised learning is an approach within machine learning where there is a labeled target of each training data example, and is widely used in many different machine learning problems both for classification, where the target is discrete, and regression, where the target is continuous [4, p. 9]. In supervised learning, the model is trained iteratively on each training data example and its parameters are updated continuously often using a loss function. The loss function evaluates the error of each prediction and is dependent on the model parameters. Said parameters can vary in form based on the model and implementation, for many models they are referred to as weights. The goal of the training is to minimize the loss function by iteratively change the weights [5, p. 41][4, p. 18].

As the loss function is used in training to evaluate the error of a prediction, choosing the type of loss function is very dependent on the problem formulation. Usually, just using the difference as loss function is not enough as it doesn't represent different errors in the same way; the loss function must provide a way to compare different errors equally in order to minimize the loss function effectively. Hence the loss function can manipulate the error to make it more comparable, such as Mean Square Error, Root Mean Square Error and Mean Absolute Percentage Error. For classification problems where the model outputs probabilities of a sample being in the different classes, the cross-entropy function is a popular choice, often referred to as log-loss in the machine learning community. Cross entropy is the representation of the differences of two probability distributions, as presented in equation 2.1 where p_k is the actual class indicator of a sample being class k and $\hat{p}_k$ is the probability predicted for class k by the model for all classes $k \in K$. In binary classification with only two classes, the error measure can be simplified into equation 2.2 [4, p. 309].

$$H(p, \hat{p}) = -\sum_{k \in K} p_k log(\hat{p}_k) \tag{2.1}$$

$$\mathscr{E} = -(p\ log(\hat{p}) - (1-p) log(1-\hat{p})) \tag{2.2}$$

In figure 2.1 the log loss is shown for a prediction where the correct class is $k = 1$. Hence, for predictions close to the correct class log loss slowly decreases as $\hat{p}$ approaches 1, while more heavily penalizing errors the closer to the wrong class, 0, they are. How the loss function is minimized is further elaborated for each discussed model in sections 2.3 and 2.4.

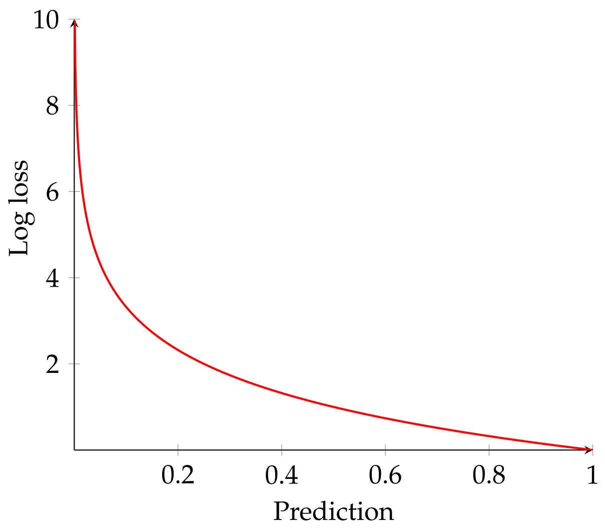

Figure 2.1: The log loss function for class 1.

2.3 Gradient Boosting Machines

Gradient boosting [4] is the term describing several weak learners used sequentially to build one strong learner. Each weak learner is trained on the data set by mapping each input to an output, and returns a weighted version of the data, which the next weak learner uses as input. Gradient boosting has proved to be a very successful and therefore widely used machine learning method. It is an adaptive basis function model, and the model $G(x)$ is trained using the following formula:

$$G(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{2.3}$$

The variable weight arrays $\alpha_m, m \in M$ are assigned after each fitting of a model G_m. The arrays are of the same length as the training data set, $|N|$. Each weight $w_n, n \in N$ represents how much a sample (x_n, y_n) should affect the models prediction. All weights in α_1 are set to $\frac{1}{N}$ as all input samples should be of the same importance for the first iteration. For each following iteration m, the weights are based on the output of the model in iteration $m-1$, $G_{m-1}(x)$. For all samples (x_n, y_n) that were missclassified in iteration $m-1$, the pertaining weight w_n is increased, while weights pertaining to correctly classified samples are decreased. This part of the algorithm makes it less sensitive to imbalance among classes in the training data, as it naturally weighs samples based on the prediction history.

The gradient boosting machine, or GBM, can be implemented as a forward stagewise additive model, meaning that an addition at each step of the model does not require any changes in parameters or coefficients in previous steps. The general algorithm for gradient boosting of a model f_M of M submodels includes the steps in snippet 2.1. Here β is the expansion coefficient and $b(x;\gamma)$ is some additive expansion function dependent on variable x and parameters γ [4, pp. 337–361].

Listing 2.1: General algorithm for GBM

```
$f_0 = 0$
For each model $m \in M$
        Compute $(\beta_m, \gamma_m) = \underset{\beta,\gamma}{argmin} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i;\gamma))$
        Assign $f_m(x) = f_{m-1}(x) + \beta_m b(x;\gamma_m)$
output $f_m(x)$
```

Decision Trees for Gradient Boosting

Gradient boosting can be implemented using decision trees. Decision trees sequentially divide the variable space into disjoint regions R_j based on some threshold γ for each node in the tree; the parameter γ in snippet 2.1 is hence the partitions of the regions in tree implementations of GBM. Given a variable combination x, the tree is traversed downwards based on which region R_j that x belongs to. With a constant γ_j pertaining to each R_j, a tree with J branches can hence be described as in equation 2.4 and a collection of these trees is described in equation 2.5, where $\Theta = \{R_j, \gamma_j\}$ for $j = 1, 2, .., J$.

$$T(x;\Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_j) \tag{2.4}$$

$$f_M(x) = \sum_{m=1}^{M} T(x;\Theta) \tag{2.5}$$

To calculate the additive expansion function, Θ needs to be assigned. Hence, in each iteration the sum of some loss function $L(y_i, \gamma_i)$ for each branch and each input sample needs to be minimized over Θ to find $\hat{\Theta}$.

$$\hat{\Theta} = \underset{\Theta}{\text{argmin}} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, \gamma_i) \tag{2.6}$$

Calculating a value for R_j is the most complex and difficult part, which γ_j is dependent on. A common approach is to estimate R_j by using a greedy top-down partitioning algorithm. The collection of trees in equation 2.5 is implemented as an forward stage wise model according to snippet 2.1 by adding $T(x_i;\Theta)$ as the additive expansion function. The parameter Θ_m can be found by minimizing the function over it, as shown in equation 2.7.

$$\hat{\Theta}_m = \underset{\Theta_m}{\operatorname{argmin}} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, f_{m-1}(x_i) + T(x_i;\Theta_m)) \tag{2.7}$$

The gradient boosting applies to the gradient of the loss function, as defined in equation 2.8.

$$-g_{im} = \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}, \text{ where } f = f_{m-1} \tag{2.8}$$

The gradient of the loss function is used to stagewise fit the model, when using cross entropy as loss function it represents the residual when fitting each tree. The probabilities for all classes K needs to be defined to use the gradient, and the logistic definition for class k defined in equation 2.9 is used.

$$p_k(x) = \frac{e^{f_{k(m-1)}(x)}}{\sum_{l=1}^{K} e^{f_{l(m-1)}(x)}} \tag{2.9}$$

The steps of the GBM algorithm are summarized in snippet 2.2. There are two main hyperparameters which are to be tuned; the number of trees, M, and the size of each tree, J. The negative gradient of the loss function is used as the residual, r, which is the deviance of the k^{th} class $y_k - p_k$ when cross entropy is used [4, pp. 337–361].

Listing 2.2: GBM algorithm for classification with cross entropy as loss function

```
$f_{k0} = 0$ for all classes $k \in K$
For m=1 to M:
        Assign $p_k = \frac{e^{f_k(x)}}{\sum_{l=1}^{K} e^{f_l(x)}}$ for all $k \in K$
        For each class k in K:
                For each training sample $i$:
                        Calculate $r_{ik} = y_{ik} - p_{ik}$
                Fit a regression tree to targets $r_{ik}$ resulting in
                regions $R_{jm}$ for $j = 1,2,...,J$
                For each $j$:
                        Calculate $\gamma_{jkm} = \underset{\gamma_{jm}}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma_{jm})$
                Update $f_{km}(x) = f_{k,m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jkm} I(x \in R)$
output $f_k(x) = f_{km}(x)$ for all $k \in K$
```

2.4 Artificial Neural Networks

Artificial neural networks, or ANNs, are a type of machine learning model and as the name suggests, the model consists of artificial neurons which are interconnected. The links between the neurons, which in a human biological neural network would be the synapses, transfers the signal from one neuron to another. Just as a GBM, the ANN is trained by mapping training data inputs to correct output resulting in a complex collection of simple functions that are supposed to describe the data as correctly as possibly. As the input data flows through the network, at each neuron it is manipulated using activation functions, weights and biases, all pertaining to the neurons. These components are further explained below.

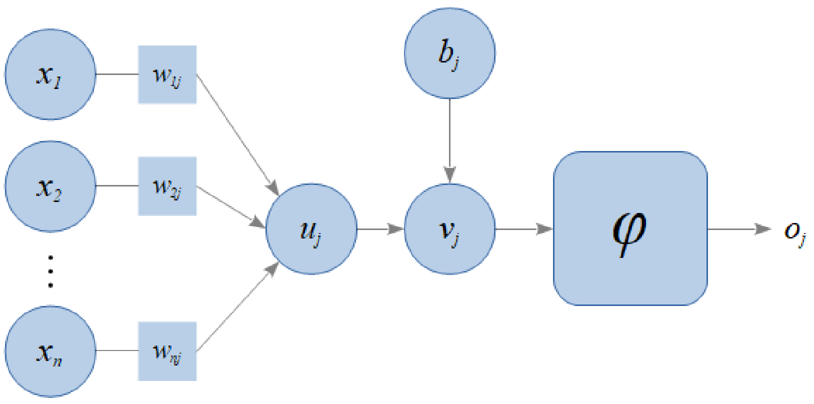

Figure 2.2: Model of an artificial neuron

Synaptic Weights The links, or synapses, between neurons carry different weights. A signal sent from neuron x_i to neuron x_j is enhanced by a factor of the connecting links weight, w_{ij}. In this case, neuron x_i is the predecessor of neuron x_j. The synaptic weight can both be of positive and negative value. The weights are updated through out the training, and are the components that affect which neurons should affect the output and how much [6, pp. 10–14].

Adder The adder is the summation function for all inputs of one neuron. If a neuron has inputs from neurons $x_i \in X$ all with respective weights of w_{ij}, the overall input to neuron j can be described by 2.10 where o_i is the output of neuron x_i. Hence, the input of a neuron consists of the summarized output of the preceding neurons multiplied with their respective weights [6, pp. 10–14].

$$u_j = \sum_{i}^{X} o_i w_{ij} \tag{2.10}$$

Bias A bias of some sort, b_j, is usually added to the summation of the adder to make the network more flexible. A bias can be added to any layer, and is set to a constant to provide a constant value for a neutron from which the ANN can learn. It can be compared to the constant c in a linear function $y = mx + c$, which shifts the function by simply adding a constant value. The shifting is important as it makes the network robust to bad weight initialization[5, pp. 139, 227].

Activation Function Each neuron has an activation function, φ, that transforms the output of the neuron. The input of the activation functions is the sum of the adders output and the bias, called the activation potential, $u_j + b_j = v_j$. A representation of the activation function is shown in equation 2.11. The activation function produces the output of the neuron, o_j. There are several kinds of activation functions, further explained in section 2.5 [6, pp. 10–14].

$$\varphi(u_j + b_j) = \varphi(v_j) = o_j \tag{2.11}$$

These components and their structure in a neuron is shown in figure 2.2, where $x_1, x_2..x_n$ are the inputs to the neuron. A neural network can be trained in different rates. The training data can be fed to the network all at once, in batches or using online training. Batches are very useful when there are memory restrictions for big data sets, but they also affect the rate of the learning, convergence and performance of the network which is further developed in section 2.5.

2.5 Feed Forward Networks

A common type of neural network is the feedforward neural network, FFNN for short, where activation from each node is only passed forward to neurons in the next layer. The Multi Layer Perceptron, referred to as MLP, is a simple yet powerful feed forward model which is depicted in figure 2.3.

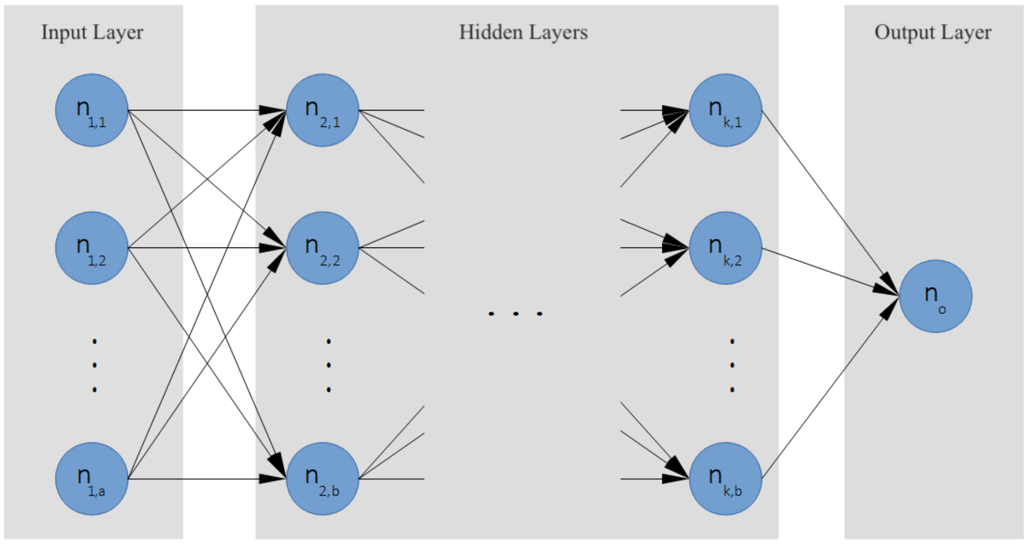

Figure 2.3: A multilayer perceptron

The MLP consists of an input layer, an arbitrary number of hidden layers (based on the problem) and an output layer. This example shows a model with a input neurons, b neurons per hidden layer and $k-1$ number of hidden layers.

Training with ANNs

There are several different types of learning for ANNs, although supervised learning will be further developed. Here, the correct output for each delivery entry is one of the classes "delayed" and "in_time" which the network is trying to predict, denoted y. When training an ANN, the synaptic weights of the network are updated in iterations by minimizing the loss function, denoted as L. To do this, the error of a prediction for the whole network needs to be defined. This is shown in equation 2.12, where e is the error, y is the actual correct output defined in the training set entry, and o the predicted output. As this is a binary classification problem, y will be a probability which can be translated to which class is predicted [6, pp. 51, 84–87].

$$e(n) = y(n) - o(n) \tag{2.12}$$

The error is hence the difference of correct output and the networks output. The error is fed back through the whole ANN to enable it to shift weights based on the chosen loss function, as seen in figure 2.4 [7, p. 151][8][6, p. 84].

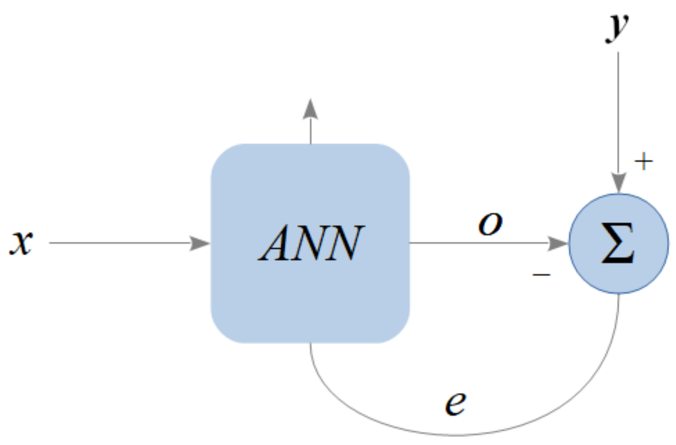

Figure 2.4: Error-correction learning process

Loss Function Minimization

Based on the definition of the error $e_j(n)$ of a neuron j for training example n, as in equation 2.12, the loss, L, needs to be defined. The loss function calculates the loss with the error as input parameter. The minimization of the loss function is then iterated during the training process for each training example, and after each batch the weight vector is updated based on the optimized loss function [6, p. 161].

The loss function is minimized with regards to the set of weights θ for N training data points, hence the goal is to minimize $L(\theta)$. Important to note is that a global minimum is not desired, as this would lead to overfitting of the model. Overfitting and how to avoid it is further elaborated in section 2.6. The minimization of $L(\theta)$ is shown in equation 2.13 for cross entropy.

$$L(\theta) = -\sum_{i=1}^{N} p(x_i)\, log(\hat{p}(x_i)) - (1 - p(x_i))log(1 - \hat{p}(x_i) \tag{2.13}$$

Since a neural network has an, often high, arbitrary number of weights, θ is often high-dimensional. An algebraic minimization of the calculation of $L(\theta)$ is impossible in terms of time and computational power, as the loss function needs to be minimized several times during training. Instead, a good estimate of a minimized loss function can be achieved by calculating the gradient of the loss function based on the networks weights at each iteration. Calculating the gradient of the loss with regards to the weights requires a calculation of each partial gradient of the loss function, $\frac{\partial L}{\partial w}$. This would result in a computational complexity dependent on the dimension of θ. To solve this problem, the back propagation algorithm was introduced [4, p. 395].

Backpropagation

Backpropagation is a commonly used algorithm which calculates the gradient of the loss function when training an ANN to update the weights. Instead of requiring traversing of the neural network for each partial derivative and hence for each weight, the backpropagation algorithm traverses through the network once, and then once again backwards which results in a feasible computational complexity.

Backpropagation utilizes the chain rule to calculate the derivative of the ANN. For each weight w_{ij} pertaining to a synapse between nodes i and j, the partial derivative for training sample n and the chain rule expansion is shown in equation 2.14. The chain rule is applied twice, since node output o_j is a function of node input u_j, which is a function of weights w_{ij}. Note that the bias is not included in this example, as it does not make any difference the method of calculation. Hence the added output of the preceding nodes u_j is referred to as the input. Furthermore, as this is pertaining to each training example x_n, $L(x_n)$ is denoted as L_n.

$$\frac{\partial L_n}{\partial w_{ij}} = \frac{\partial L_n}{\partial o_j}\frac{\partial o_j}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}} \tag{2.14}$$

It is then simpler to calculate each partial derivative. The partial derivative of the node input u_j over weight w_{ij} can be simplified by using the definition of u_j in equations 2.10 as below:

$$\frac{\partial u_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_i^X o_i w_{ij} + b_j = \frac{\partial(o_i w_{ij} + b_j)}{\partial w_{ij}} = o_i \tag{2.15}$$

The local gradient of the loss function over the node input u_j is denoted δ_j, and the definition of this can be used for deriving equation 2.16.

$$\delta_j = \frac{\partial L_n}{\partial u_j} = \frac{\partial L_n}{\partial o_j}\frac{\partial o_j}{\partial u_j} \Rightarrow \frac{\partial L_n}{\partial w_{ij}} = \delta_j o_i \tag{2.16}$$

δ can be calculated recursively, since L is a function of the errors from all neurons k between the current node j and the output node, and the actual output node as explained in equation 2.17. The term can be expressed as only dependent on a previous node k, to make it truly recursive. Given equations 2.10 and 2.16, this expression can be simplified to equation 2.18.

$$\delta_j = \frac{\partial L_n}{\partial u_j} = \sum_{k \in K} \frac{\partial L_n}{\partial u_k}\frac{\partial u_k}{\partial u_j} \tag{2.17}$$

$$\delta_j = \sum_{k \in K} \delta_k \frac{\partial u_k}{\partial u_j} = \varphi'(u_j) \sum_{k \in K} \delta_k w_{kj} \tag{2.18}$$

Since the δ values are already known for the output neurons, back propagation is easy to perform using these equations. The gradient of the loss function, L, with regards to each weight w_{ij} is then calculated using equation 2.16 and 2.18. The latter equation is rewritten to use the gradient of the activation function φ, which is further explained in section 2.5 [5, pp. 241–245].

Optimization

When the gradients have been calculated using back propagation, these are used as input to the optimization function of the neural network which then updates the weights. A commonly used optimization function is Adam, introduced by Kingma and Ba in 2014 [9]. It is popular due to how fast it is as well as how easy it is to implement. Adam stands for Adaptive Momentum Estimation, and adapts the learning rate through out the training process based on the moving average of the gradient, m_t and the squared gradient, v_t, at each time step t which represent one batch. Hence, all operations in the following equations are performed elementwise for each sample in the batch of training examples. Equations 2.19 and 2.20 show the calculation of the moment estimations.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.19}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.20}$$

Some variables are needed to be set depending on implementations; the learning rate α, the exponential decay rates for the moment estimates, β_1 and β_2 respectively, and the constant to avoid division by 0, ϵ. For the first time step, $t = 0$, both moment estimation values are initiated to vectors of zeros. Hence, the moving averages will be biased to zero in initial calculations, which needs to be corrected. Equation 2.21 describes the bias correction which is done to each value.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \ \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.21}$$

The update function of Adam is presented in equation 2.22, where θ_t is the updated weights to be applied after time step t.

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \tag{2.22}$$

Batch Learning

A natural incentive to divide training into batches is a limited amount of memory, but it can also improve the performance of the model in terms of accuracy. When using gradient based optimization techniques, such as Adam, batch learning is imperative as using the gradient of the whole data set often results in poorer generalizing ability of the model. Instead, mini-batches ranging from size 32-512 is commonly used. Smaller batches results in higher time complexity regarding calculations, but will also make the network converge faster [10].

Activation Functions

There are many different activation functions used when implementing a neural network, however some have come to be the most commonly used ones and these are presented in this section.

Linear Activation

Linear activation, also called the identity function, is described in equation 2.23. It simply passes the input of a neuron through as output without any manipulation other than multiplying with the pertaining weight. Hence, the number of hidden layers is insignificant as any structure will perform as a single layer network. Linear activation is commonly used for linear regression, but is not capable to handle more complex problems. To create more complex neural network models, non-linear activation functions are necessary to map inputs to outputs for more variant and non-linear data.

Furthermore, since a linear function has a static gradient, a network with this activation will not be suitable when using backpropagation in training. Even if backpropagation is used to calculate the gradient of the loss, the gradient of the activation function is used in this calculation, recall equation 2.18.

$$\varphi(v) = v \tag{2.23}$$

Tanh Activation

The tanh function is described in equation 2.24.

$$\varphi(v) = \tanh(v) = \frac{sinh(v)}{cosh(v)} = \frac{e^v - e^{-v}}{e^v + e^{-v}} \tag{2.24}$$

The non-linear function ranges from -1 to 1 and is centered around zero, as shown in figure 2.5. This means that extreme values, both negative and positive, will be mapped into similar values close to -1 and 1 respectively. The derivate, $1 - tanh^2(v)$, is helpful when the network is trained though backpropagation as it requires little calculation. As the function centers at the origin, it can produce exponentially decreasing gradients when training.

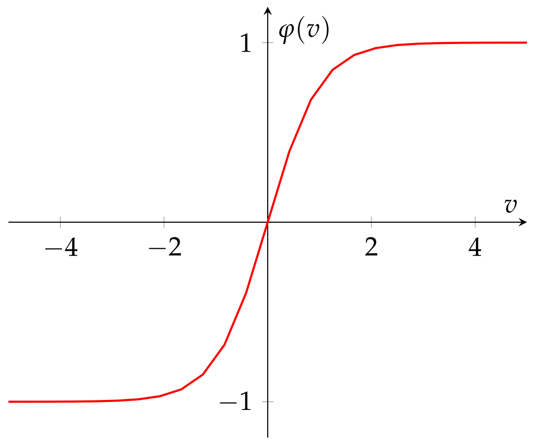

Figure 2.5: The tanh function

Rectified Linear Unit

Rectified Linear Unit, ReLU for short, is currently the most commonly used activation function of hidden layers of neural networks. It is described in equation 2.25 and figure 2.6.

$$\varphi(v) = max(0, v) \tag{2.25}$$

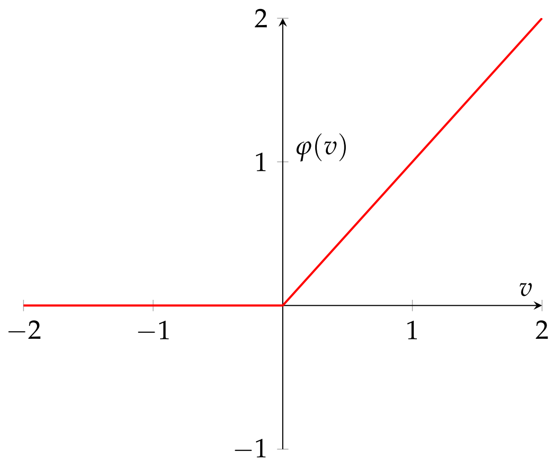

Figure 2.6: The ReLU activation function

As it is a combination of two linear functions, it has a static gradient of 0 for negative values, and 1 for positive values. This aspect of the ReLU function makes it a fitting activation for several reasons. The static gradient avoids the vanishing gradient problem and makes calculations simpler when training. The vanishing gradient problem occurs when changes in updates are so small that the gradient of the loss function approaches 0 when training. The two possible gradients of the ReLU function prohibits this from happening. Furthermore, it provides a sparsity of the data flowing through the network as all negative inputs are outputted as 0. The partly-linear nature of the ReLU can however lead to the activations blowing up to big values as it has a range of $0 < \varphi < \infty$.

Sigmoid Logistic

The range of the ReLU function also makes it inappropriate for classification problems. For the output layer of a classifier, a probabilistic function is a more common choice, outputting the probability of the input being a certain class. For binary classification problems, the Sigmoid function is popular. The function is described in equation 2.26.

$$\varphi(v) = \frac{1}{1 + e^{-v}} \tag{2.26}$$

A sigmoid curve has a characteristic horizontal S-shape. A commonly used version is the logistic function described in equation 2.26 ranges from 0 to 1. Figure 2.7 shows the sigmoid logistic function [6, pp. 10–14].

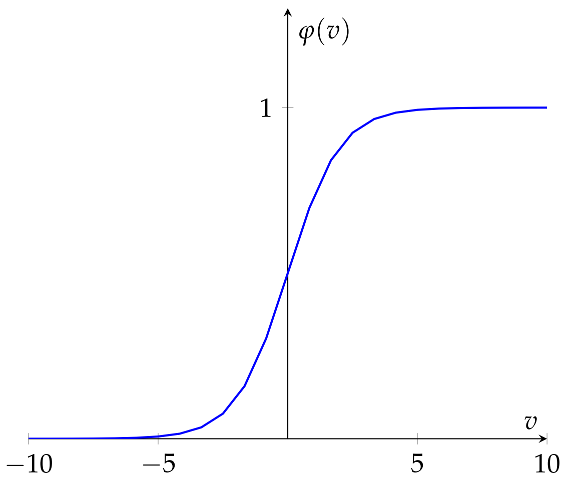

Figure 2.7: The sigmoid logistic function

2.6 Regularization

Finding a global minimum of the loss function would give the perfect model, however it would be optimal for the training data and probably not perform well when introduced to new data. When a model is trained to a global minimum, or too close to the global minimum, it will overfit to the training data. To overcome this, different regularization methods are commonly used. Regularization creates noise in the data, to make the model more robust and less likely to overfit on the training data and therefore better when encountering data not seen before.

Dropout

To create noise, dropout can be implemented. Dropout means that some inputs are ignored in order to make the model more robust, and differs in implementation depending on the model. In neural networks, a dropout layer is implemented to set a random group of inputs to 0 so they are not to influence the next activation, given a rate which decides what percentage of the inputs should be set to 0 [11]. In GBM, dropout is implemented per tree [12, p. 13].

Early Stopping

When training a machine learning model on a data set, after some iterations the model can overfit, leading to the training accuracy improving but the validation accuracy declining. This is easily prevented by using early stopping, which stops the training given some criteria. A common criteria is when validation accuracy begins to decrease, and the stopping can be implemented. This is an easily implemented method, but can have the effect of stopping the model training too early, not letting it converge properly [5, pp. 259–260].

This regularization method can be implemented with a patience parameter, which decides how many iterations the model's performance can decrease before stopping to make sure any fluctuating results does not stop the training too early, as well as stopping tolerance, which decides by what value the performance metric has to improve each iteration.

Data Shuffling

A common approach used when using a SGD optimization algorithm is shuffling the data, preventing the model from learning a pattern of the order of the presentation of training samples to the model. It is common to shuffle the data between epochs [13][14]. It is however sufficient to only shuffle the original training set once before training for very big data sets, as it will still break any natural order of the original data set that might affect training and is less time consuming [7, p. 277].

Batch Normalization

Batch normalization is a method to speed up training in neural networks by normalizing, scaling and shifting each batch during training, and is primarily to be used for mini-batches. Besides improving the training time, it also adds noise to the data at each layer, giving it some regularizing properties. It normalizes the input to each layer so that the mean and the standard deviation of the batch is close to 0 and 1 respectively, by subtracting the batch mean from the batch values and then dividing them with the batch standard deviation. Each value in a batch is normalized as explained in equation 2.27, where $\mathcal{B} = \{x_1, ..., x_m\}$ is the batch of data samples, $\mu_{\mathcal{B}}$ is the mean of batch $\mathcal{B}$, $\sigma^2_{\mathcal{B}}$ is the variance and ϵ is a constant to add numerical stability to the calculation, like the bias in neural networks. The output of the batch normalization layer is then calculated by scaling and shifting x_i as explained in equation 2.28, where γ and β are parameters to scale and shift the value. These are updated as the network is trained, and are needed since the normalization can restrict the activation of each layer, which can impact the network negatively; each activation function has a set range for a reason. Hence, the output of the batch normalization layer is $\hat{\mathcal{B}} = \{\hat{y}_i, ..., \hat{y}_m\}$.

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} - \epsilon}} \tag{2.27}$$

$$\hat{y}_i = \gamma \hat{x}_i + \beta \tag{2.28}$$

The stable distribution of the values at each batch leads to faster training. Batch normalization also adds noise to each hidden layer which has a regularizing effect [15].

2.7 Hyperparameter Optimization

Machine learning models, neural networks particularly, often require much time spent on tuning hyperparameters where both manual methods and algorithms are commonly used. However, finding a model with the optimal selection of hyperparameters is not a possibility when implementing a neural network of this sort, as the search space is too big. Hyperparameters include all parameters were a value is chosen and set before training, such as learning rate, network depth, number of layers, batch size and many more. Many of the parameters are dependent on each other and therefore should be tuned simultaneously.

Grid Search

Grid search is a commonly used hyperparameter optimization technique. It is easy to implement and does not require any prior knowledge or experience of statistics or optimization. It is an exhaustive methods, where the search space is set up to be a grid for decided range of the hyperparameters. These points are equally distributed over a decided range. For each point in the grid, the objective function is evaluated with pertaining hyperparameters, hence the time complexity is linear to the size of the grid. Due to its exhaustive nature grid search is often too time consuming for models with many hyperparameters and big search spaces, and a limitation of grid points might be necessary [16].

Random Search

Random search deals with the time complexity problem of grid search. Instead of exploring every point in the search space, a new point for every iteration is chosen at random which explains the name of the algorithm. It outperforms grid search as it is found to yield a better hyperparameter set in the same number of iterations as it can search a vaster search space, it can however result in biased results as the randomly chosen values can be nowhere representative of a near optimal solution [17].

Sequential Model-Based Optimization

To solve the problem of the time complexity of grid search and possible bias of random search, the optimization method Sequential Model-Based Optimization, SMBO for short, is commonly used. Instead of either doing an exhaustive search or choose where next to evaluate the model, referred to as the objective function, in the search space by random, this algorithm approaches the problem by using a less time-costly surrogate function to decide which set of hyperparameters to try next in each iteration. Even if the surrogate function adds to the time complexity, the over all result improves as the optimization does not need as many iterations and searches the space based on the surrogate function but is not limited by a smaller or sparser search space.

In SMBO, the surrogate function approximates the optimal set of hyperparameters, x^*, based on all previous iterations which have yielded a tuple of a set of hyperparameters and the pertaining loss for this, called the observed value and denoted as y. This selection x^* is then used as hyperparameters for the next iteration when the objective function is evaluated. The value of x^* for the next iteration is found by optimizing the Expected Improvement, EI, over x using the chosen surrogate function. The definition of EI is shown in equation 2.29, where x is the hyperparameter set, y^* is some threshold for the value of the objective function $f(x)$ and y is the actual value of the objective function, $f(x) = y$ [18].

$$EI_{y^*}(x) = \int_{-\infty}^{\infty} max(y^* - y, 0) p(y|x) dy \tag{2.29}$$

There are several different surrogate functions to apply to SMBO, a common one is tree Parzen estimator, TPE, which is suitable for bigger search spaces with time constraints [18].

Tree Parzen Estimator

The surrogate function tree Parzen estimator generates a candidate x^* for optimizing the EI described in equation 2.29 by estimating both $p(y|x)$ and $p(y)$. For some quantile q, threshold y^* is chosen so that q of the previously observed values of y fulfills the statement $p(y < y^*) = q$. Furthermore, y^* is chosen to be larger than the best observed value of the objective functions, $f(x)$, to be able to form two densities over the search space X where the density $l(x)$, contains all x that fulfills $f(x) < y^*$, and one, $g(x)$, containing all x that fulfills $f(x) > y^*$. Hence, all x where the pertaining objective function yields a value lesser than the threshold are in $l(x)$ and all x pertaining to an objective function yielding a value greater value than the threshold are in $g(x)$. From this the definition of $p(x|y)$ can be defined as in equation 2.30.

$$p(x|y) = \begin{cases} l(x), & \text{if } y < y^* \\ g(x), & \text{if } y \geq y^* \end{cases} \tag{2.30}$$

When maximizing the EI in equation 2.29, the fact that $y < y^*$ enables three shortcuts in the optimization:

- $max(y^* - y, 0) = y^* - y$
- $p(y)$ does not need to be calculated since $p(y < y^*) = q$
- The integral span is limited to $-\infty \leq y \leq y^*$

The use of these conditions in the optimization is shown in equation 2.31

$$\begin{gathered} EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy = \int_{-\infty}^{y^*} (y^* - y) \frac{p(x|y)p(y)}{p(x)} dy \\ \Big/ p(y \leq y^*) = q,\, p(x) = p(x|y)p(y) = ql(x) + (1-q)g(x) \Big/ \\ = \frac{qy^* l(x) - l(x) \int_{-\infty}^{y^*} p(y) dy}{ql(x) + (1-q)g(x)} \propto \left(q + \frac{g(x)}{l(x)}(1-q)\right)^{-1} \end{gathered} \tag{2.31}$$

To conclude, $EI_{y^*}(x) \propto \left(q + \frac{g(x)}{l(x)}(1-q)\right)^{-1}$, meaning that to maximize the EI, the term $\frac{g(x)}{l(x)}$ can simply be minimized. This means that points in $l(x)$ should have as high probability as possible, and the opposite for $g(x)$. Hence, TPE draws candidates from $l(x)$ an evaluates them on how the candidate maximizes $\frac{l(x)}{g(x)}$ and then outputs the candidate x^* which yields the best value. The hyperparameter set x^* is then used as hyperparameters for the next iteration for evaluation of the objective function [18].

2.8 Classification of Imbalanced Data Sets

Imbalanced data sets are a widely researched problem when applying machine learning algorithms to industrial problems. Outlier detection is an example of both supervised and unsupervised learning with imbalanced data. Chandola, Banerjee, and Kumar [19] present a survey of commonly used classifiers for outlier detection, showing that neural networks are often used for the application on both textual and image data sets. The application includes detection of fraud and intrusion, both types of outlier detection.

Sampling

A common approach to handle class imbalance is sampling of the training data. When a large data set is available, under sampling is a valid method. When under sampling is used, the majority class is reduced to the same size as the minority class. If the data set is smaller, over sampling can be more adequate. Instead of reducing the majority class, the minority class is expanded. This can be done by either making copies or augmented copies of the minority class, until it is the same size as the majority class.

Another method especially applicable in iterative machine learning models is dynamic sampling, where a new sample of the training data is drawn between iterations. Vannucci, Colla, Vannocci, and Reyneri purpose a resampling algorithm that both takes previous sampling history and current results of the model into account [20]. It is an iterative undersampling algorithm that draws all minority class data points and draws a sample of the majority class based on a calculated index of the trained machine learning models classification performance index and a ratio dependent on how many times the data points has been sampled before. The classification performance index m_i is shown in equation 2.32, where $UnfDet$ is the ratio of correctly detected minority class data points, FA is the false alarms ratio, $Corr$ is the overall accuracy, and γ as well as μ are empirically set variables. m_i is limited to $m_i \in [0,1]$.

$$m_i = \frac{\gamma UnfDet - FA}{Corr + \mu FA} \tag{2.32}$$

The probability for each of the data points of being drawn in a sample is then calculated using equation 2.33, where N is the number of resamplings done for the training, S_i is the number of times the data point evaluated has been sampled, and S_j is the number of times it has been sampled for each resampling iteration. Hence, the probability will be higher for samples that have never been sampled, as well as samples that have been used in a former resampling where the trained model produced a higher classification performance index. The sampling is not random, but based on former samplings that have performed well.

$$p_i = 0.5\frac{N - S_i}{\sum_{j=1}^{N} N - S_j} + 0.5m_i \tag{2.33}$$

3 Method

This chapter is dedicated to explain and motivate the choice of method.

3.1 Data Set

Since the data set provided was extracted for machine learning, it is already cleaned and formatted in a feature matrix. Feature selection was performed by implementing a GBM model one feature at a time and measuring the effect on the model's performance.

Features

The following features are available in the data set, some of which have been engineered into new features. All features are listed in table 3.1, and how they are derived if they are engineered. In this table, i represent the current event of a package, and 0 the first. All zip codes are reduced to their country and first two digits, as this is found to be as sufficient as the whole number by PostNord. Totally 17 of the features where used, as the EventTime and IETA where removed because of redundancy, and the ATA is used to calculate the Target. The excluded features are highlighted below. The target used in training is defined as in equation 3.1

$$Target = \begin{cases} delayed, & \text{if } ATA > IETA \\ in\_time, & else \end{cases} \tag{3.1}$$

3.2 Overall Framework

With the pre-existing conditions considered, the overall process of this thesis includes the following steps:

1. Sample data into training, validation and testing data
2. Preprocess data and cross validate neural network
3. Train and test models
4. Compare and evaluate both models

Note that the data received is already cleaned and readable by the GBM. Therefore, the data does not need to be cleaned and only needs to be preprocessed for the neural network. Since the GBM has already been tuned and crossvalidated using grid search, cross validation will only be done on the neural network.

Table 3.1: Features of the data set.

	Feature	Derivation
Continuous	EventTime, Unix time stamp	
	ATA, Actual time of arrival	
	IETA, Initially estimated time of arrival	
	IETTA, Current time to estimated arrival	$IETA - EventTime$
	DTA, Distance to arrival	Measured "as the crow flies"
	IEVTA, Required velocity from current location	$DTA/IETTA$
	ElapSec, Elapsed time from first event	$EventTime_i - EventTime_0$
	ElapFrac, Fraction of elapsed time	$ElapSec/IETTA$
	Location Longitude	
	Location Latitude	
Discrete	LocalEventKey, label for sorting event	
	ConsignorZipCode	
	ConsigneeZipCode	
	ServiceCode, type of service by PostNord	
	LocationKey, label for sorting hub	
	LocationZipCode	
	Hour, in intervals of 3	
	Day of the week	
	EventCode, label for current event	
	SourceSystem, name of local system used	

3.3 Neural Network

The workflow for implementing and tuning the neural network before comparing it to the GBM includes the following steps:

1. Preprocess data
2. Undersample and shuffle training data
3. Optimize neural network with cross-validation
4. Train the optimized model with dynamic resampling
5. Output best found model based on F1-score for validation data
6. Run model on test data set

Data Preprocessing and Splitting

The python library Sci-kit Learn[1] is used for cleaning and preprocessing of the data. It provides several different tools for normalization and sampling which are useful when handling this type of data. Even if the data has been cleaned and somewhat preprocessed by PostNord, neural networks can be sensitive to large ranges in continuous variables and are limited to numerical data input which causes this need. All normalizing and scaling is based on the entire data set, and if online training would be implemented on the model in the future the preprocessing could fail on never-seen records. However, the PostNord system has a limited set of categorical variables and all continuous variables have been feature engineered to not depend on external factors. An example of a continuous feature affected is the time stamp. This feature has been engineered into new features based on the first event of the delivery chain instead, to not expose the model of never seen values, as seen in table 3.1.

Encoding and Scaling

Each categorical feature is label encoded based on the entire data set into plain integers, which in turn can be handled by the input layers of the neural network. For all feature which are non-binary, an embedding layer is required for the input to not have the network make decisions about a record based on its label encoded class. For example, if a categorical feature has been encoded into classes $1, 2, 3$, the network would view the classes as dependent on each other based on numerical comparisons, $1 < 2, 2 < 3$ and such. An embedding layer is a substitute for one-hot encoding which is demands much memory, and maps a sparse one-hot vector to a dense vector with weights that are updated during training, just like the other weights in the neural network. The dense vectors are then used as a look up table for the neural network to map a categorical input to a numerical value. [22]

All data for each continuous feature is scaled according to equation 3.2, where X is an array of all records for the feature. This scales all records to values within the range $(0, 1)$, where the minimum value and maximum value of all records in a feature would be represented by 0 and 1 respectively.

$$X_{scaled} = \frac{X - min(X)}{max(X) - min(X)} \tag{3.2}$$

Framework

The model is implemented in Python, frameworks and components used when implementing the neural network is shown in figure 3.1. The backend used for the ANN is the library TensorFlow[2]. Originally developed by Google's Machine Intelligence research organization, TensorFlow is now an open source computation library used for a wide variety of tasks, machine learning among one of the most common ones. TensorFlow computes on CPU or GPU through dataflow graphs in heterogeneous environments and at large scale, making it optimal for machine learning. All data is formatted in tensors, and when using batch learning one batch consists of one tensor within the network.

As a complement to TensorFlow, the open source high-level neural networks API Keras[3] is used. Keras is written in Python and offers the user easily accessed methods for modeling a variety of different neural networks and specify hyperparameters.

A server running 64-bit Ubuntu 18.04 is used during modelling with a Nvidia GM200 GeForce GTX 980 Ti as GPU, Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz as CPU and 16 GB RAM.

[1]David Cournapeau. "Sci-kit Learn". In: *Machine Learning in Python* (2015). URL: `https://scikit-learn.org/` (visited on 04/04/2019).

[2]"TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. vol. 16. 2016, pp. 265–283. URL: `https://www.tensorflow.org/` (visited on 04/04/2019).

[3]François Chollet et al. *Keras*. 2015. URL: `https://keras.io` (visited on 04/04/2019).

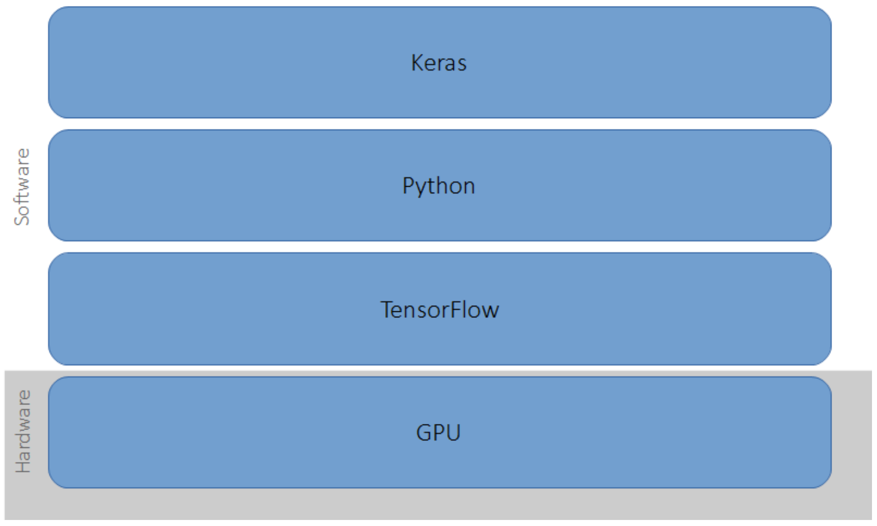

Figure 3.1: Stack of components

Implementation Structure

An overview of the preprocessing steps and neural network is presented in figure 3.2.

The categorical data is label encoded and the continuous data is scaled. All categorical data is first handled by respective embedding layers to handle their numerical values. These are then concatenated with the continuous input, pertaining to the input layer. The input layer is then connected to an arbitrary number of hidden layers, which all have ReLu as activation and a arbitrary percentage of dropout. The last of the hidden layers then connects to the output layers, which is activated by the Sigmoid function for a final binary classification, outputting the prediction. The model is trained by using binary cross entropy as loss function, as this is a fitting loss for binary classification tasks as discussed in section 2.5. To speed up training, batch normalization, as explained in section 2.6, is implemented between the hidden layers before the activation. The structure of each hidden layer is presented in figure 3.3.

Hyperparameter Selection

As discussed in section 2.7, there are several methods of hyperparameter selection. While grid and random search are commonly used, using SMBO has recently emerged as a better solution for big search spaces. Grid search is computationally demanding, and would result in a high time complexity. Random search solves the problem of high time complexity, but is however more likely to produce biased results based on the sampling. SMBO requires more implementation and understanding of optimization, however since there are tools that ease this, it is deemed the best option.

SMBO can be performed with different surrogate functions. There are several comparisons on surrogate functions, often based on the Convex and MRBI data sets [24][18][25]. Even if these provide an insight to the effect of the different surrogate functions, they are applied on a linguistic problem which can not be used as motivation in this thesis. The choice will instead be motivated by ease of implementation and available tools that are fitting. The choice of tool for SMBO is based on the following criteria:

- Open source
- Library available in Python 3.x

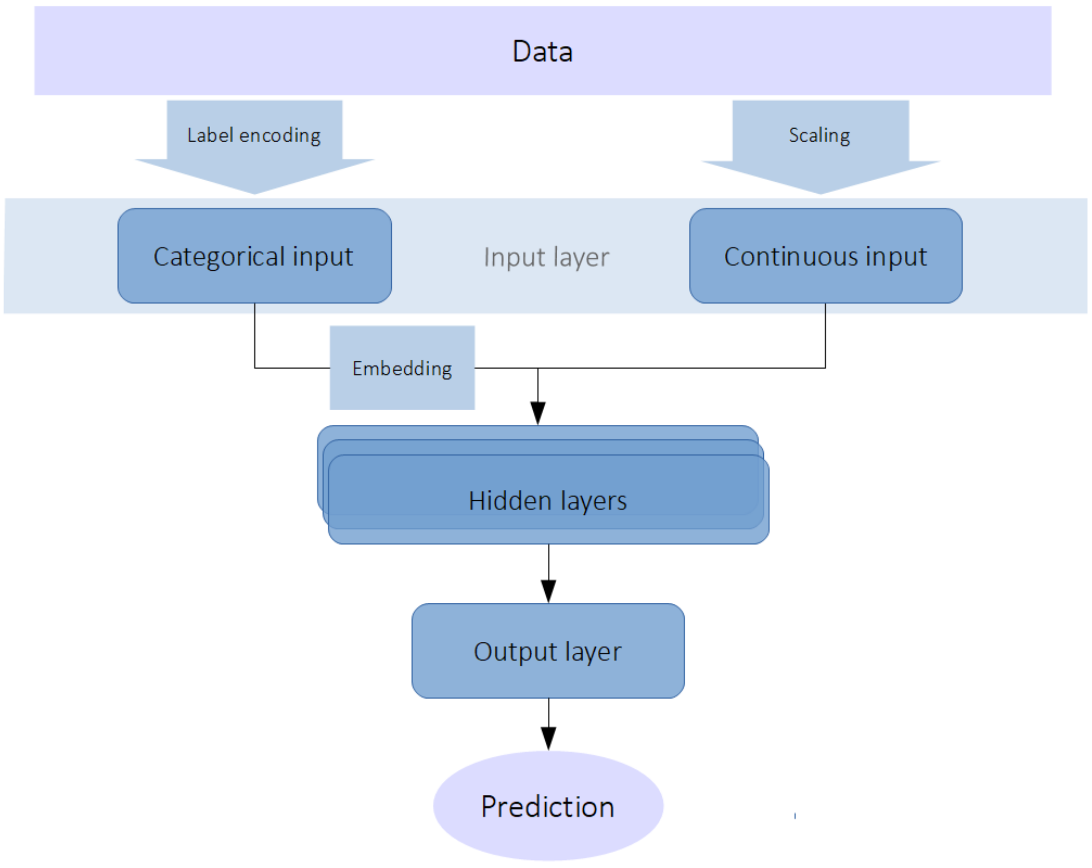

Figure 3.2: Overview of the implementation process

- Well documented

The tool found fitting for this thesis is HyperOpt, a python library [26]. It is open source, requires little implementation and offers optimization using either TPE or random forest as surrogate function. TPE is chosen for this thesis.

Search Space

The hyperparameters to be tuned are listed below, hence the optimization will include all possible combinations of these.

- Number of hidden layers
- Neurons per layer
- Learning rate
- Drop out (Regularization)

Hyperparameters to be tuned and their range is presented in table 3.2 for each optimization step.

Due to time constraint, many hyperparameters are not included in the hyperparameter optimizations. For weight initialization, He initialization [27] is used for the hidden layers and Glorot initialization [28] is used output layer respectively. The batch size is set to 2^{10}; as

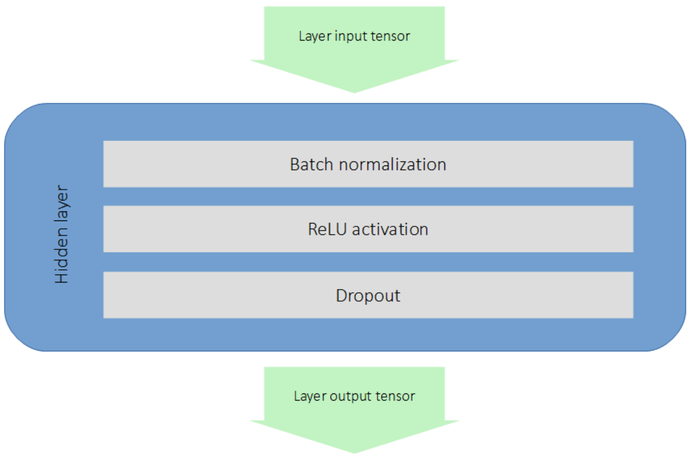

Figure 3.3: Layer structure

Table 3.2: Search space definition

Hyperparameter	Search space
Layer count	$\{x \in \mathbf{N} : 2 < x < 6\}$
Node count	$\{2^x : x \in \mathbf{N}, 3 < x < 10\}$
Learning rate	$\{x \in \mathbf{Z} : 10^{-5} < x < 10^{-1}\}$
Dropout	$\{x \in \mathbf{Z} : 0 < x < 0.7\}$

small as possible. With a smaller batch size, training would take too long. The training is done with early stopping with patience of 10 epochs.

For each epoch, the model is evaluated based on the validation accuracy. Preferably F1-score would be used to reflect the imbalanced data set, unfortunately this metric is not available in the TensorFlow implementation during training. Instead, an undersampled subset of the validation data is used with classes balanced, and the undersampled validation data accuracy is used as the metric for early stopping.

Dynamic Resampling

Since undersampling can lead to bias in the model, training is also done using dynamic sampling. Dynamic sampling allows the model to train on more data then in the undersampled set without affecting the class balance, since a subset of the majority class is drawn continuously during training. The dynamic resampling is done using the method presented in section 2.8. It will be done using static values for number of blocks, 20, and number of epochs per block, 5. Each block represents a new drawing of samples, so for each drawn set of samples the model will train for 5 epochs. However, the class balance will be tried in different variations as shown in table 3.3, where the total percentage of majority class, in this case packages in time, is listed. Since the resampling requires a static balance between classes during each training session, in cases where an excessive number of samples are drawn from either class, samples are discarded at random from the class in question to balance the resampled data set. For the index m_i presented in equation 2.32, the values for accuracy, false alarm rate and detection rate for the run with the maximum m_i is used for each data point D_i. m_i is initialized to 0 for all data points. The experiment is run with $\gamma = 0.7$ and $\mu = 0.3$ as in the original paper.[20]

Table 3.3: Parameter search space of dynamic resampling.

Blocks B	Iterations N	Percentage of majority class
20	5	50%
20	5	66.7%
20	5	25%
20	5	20%
20	5	12.5%
20	5	10%

3.4 Gradient Boosting Machine

The GBM is implemented using the AI framework H2O[4] with the same parameters as PostNords GBM and run on a different machine with 64 GB RAM and Threadripper 1920X CPU. The grid search performed by PostNord yielded values for hyperparameters as presented in table 3.4.

Training is done on the whole training set, without any static or dynamic under sampling as in the ANN. This is due to the GBM's natural weighing of samples during training. It is capable of handling an imbalanced data set, so undersampling is therefore deemed unnecessary. Hence, the GBM trains on more data than the ANN per iteration. However, since dynamic sampling is used for the ANN, it trains on more of the data than when undersampling, just distributed over different blocks of epochs.

Table 3.4: Hyperparameters for GBM.

Hyperparameter	Value
Maximum number of trees	30
Early stopping patience	2
Learning rate	0.2
Maximum depth of each tree	10
Dropout (Portion of columns used)	0.7
Dropout (Portion of rows used)	0.7
Patience (for early stopping)	2
Stopping tolerance (for early stopping)	0.01

3.5 Comparison of Models

Evaluation Metrics

Due to the imbalanced nature of the data where the important class is the minor one, metrics that reflect this aspect of the model is needed in the evaluation.

[4]Cliff Click, Michal Malohlava, Arno Candel, Hank Roark, and Viraj Parmar. *Gradient boosting machine with H2O*. 2017.

Precision and Recall

Precision and recall are common metrics used when evaluating classification models for detection of a certain important class. Recall represents how many samples of the important class was discovered by the model of all the samples in the class, while precision represents the accuracy of predictions for that certain class. In this application, the important class is "delayed". Equation 3.3 and 3.4 shows the formulas for precision and recall, *True positives* is the number of correctly identified data points of the important class, *False positives* is the number of data points incorrectly identified as important and *False negatives* is the number of data points incorrectly identified as not important.

$$Precision = \frac{True\ positives}{True\ positives + False\ positives} \tag{3.3}$$

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives} \tag{3.4}$$

F1-score

The F1 score is the harmonic mean of the recall and precision, and works as a combined measure of both these metrics.

$$F1 = 2\frac{recall \cdot precision}{recall + precision} \tag{3.5}$$

3.6 Limitations

When proceeding in a process of knowledge discovery in a database there are several issues that might have to be addressed in the research. Some of these are relevant to the data set in this research: [29]

1. Vast datasets
2. Overfitting
3. Time

Training demands much time due to the vast data set, even after undersampling, and especially for the ANN. Due to time constraint, this will limit the amount of hyperparameter tuning can be done. The optimization search space has been limited both horizontally (number of parameters to tune) and vertically (search space of each parameter). Further more, early stopping is used in the optimization to both decrease time consumption and overfitting, which will decrease overall optimization time. It can however stop training on models too early, meaning that they could have been better than presented in the results.

4 Results and Discussion

This section will present the results from the hyperparameter optimization, dynamic resampling and performance test on the test data, as well as discussion of the results, method, limitations and future work.

4.1 Hyperparameter Tuning of ANN

The hyperparameter optimization was run for 210 iterations, the results are presented in figure 4.1 and 4.2. The model which output the highest F1-score is marked with red circles and lines respectively.

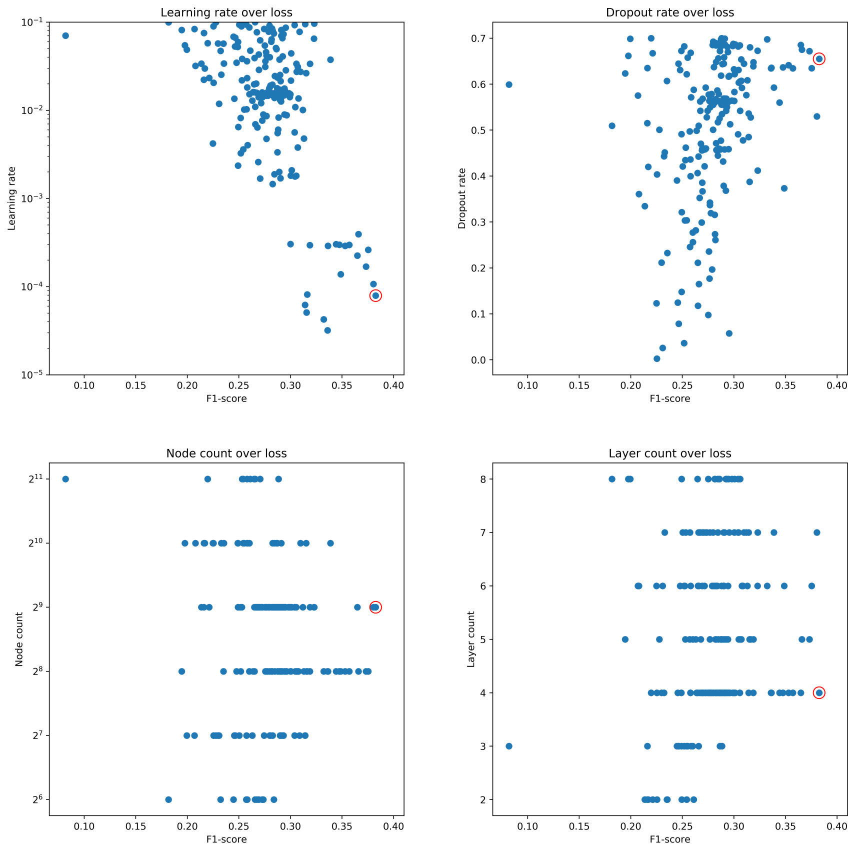

Figure 4.1: Values of hyperparameters during optimization plotted against resulting models F1-score.

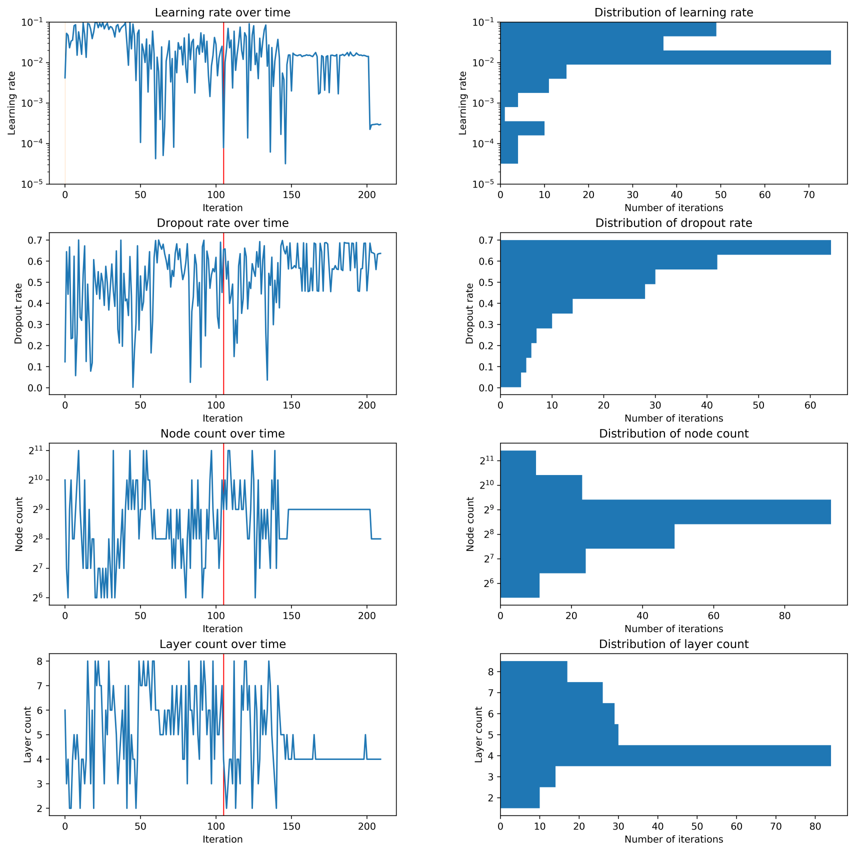

Figure 4.2: Values of hyperparameters during optimization plotted against iteration of optimization.

As seen in figure 4.3, iteration 105 yielded the model with highest F1-score. The chosen hyperparameter values and performance metrics for this model is listed in table 4.1. The iteration with highest F1-score is marked with a red line.

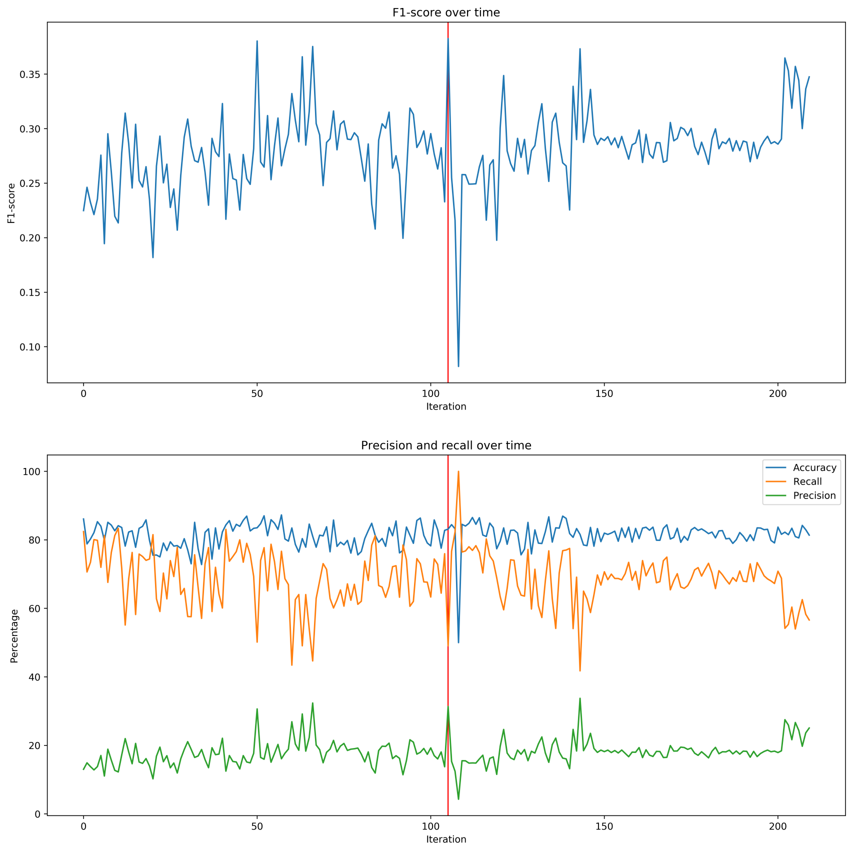

Figure 4.3: Metrics during optimization plotted against iteration

Table 4.1: Hyperparameter values for optimized model

Hyperparameter	Value
Layer count	4
Node count	2^9
Learning rate	$7.91081 \cdot 10^{-5}$
Dropout	0.65510

Dynamic Resampling

To improve the models generalization ability, dynamic sampling was applied to training instead of only undersampling. It was implemented with the six different class distributions as listed in table 3.3. Training the ANN with dynamic resampling on re-initialized weights led to an overall increase of the training accuracy but overfitting after the first resampling. The overfitting hence happened in the second block, but was persistent for the rest of each experiment. Since each resampling contain all samples from the minority class, but a resampled set of samples from the majority class, the model recognized all minority class samples but was biased towards this class. The model got 100% recall, since it predicted every validation sample to be of the minority class. Hence, the original model with weights from the optimization is used in the evaluation and comparison of the ANN and the GBM.

4.2 Evaluation of Models

Before running the models on the test data, a comparison of training loss can be made. The metrics for training of each model is presented in table 4.2 and the loss during training per epoch for the ANN and tree for the GBM is shown in figure 4.4. As explained in the method section, both training and validation data is undersampled for the ANN to handle the class imbalance, but not for the GBM, meaning all values for ANN are based on undersampled data.

Table 4.2: Training metrics for both models.

Metric	ANN	GBM
F1-score	0.38265	0.36909
Recall	49.095%	23.660%
Precision	31.150%	83.876%
Training accuracy	83.144%	96.544%
Validation accuracy	72.236%	96.571 %

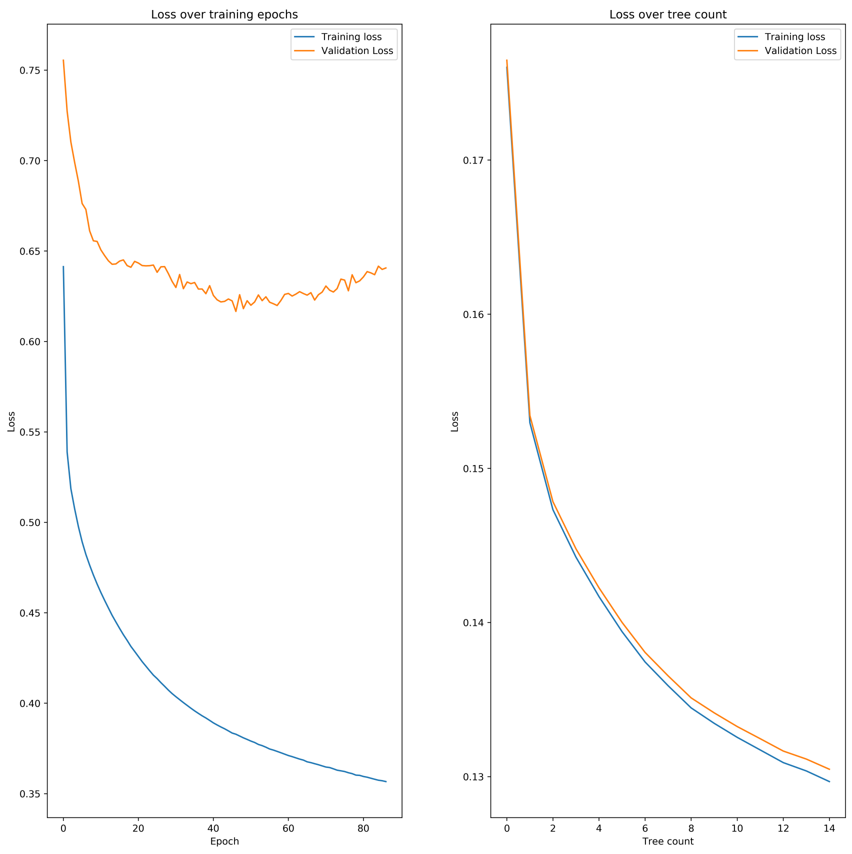

Figure 4.4: Metrics during optimization plotted against epoch for the ANN (left) and against tree for the GBM (right)

As seen in table 4.2, the ANN outperforms the GBM in recall and F1-score when run on validation data. However, the precision is poor, and the GBM has a much higher value. This means that the ANN is classifying data as "delayed" too easily; even if it identifies many of them, two thirds of the "delayed" classifications are wrong when validating. Worth noting here is that undersampling was only done on the ANN and not the GBM. Since the GBM trained on more data, it may be a reason for it having better precision than the ANN as it has seen more data of the negative class.

Both models are run on the same set of test data, the results are presented in table 4.3. The best result for each column is highlighted, the GBM outperforms the ANN in all metrics except recall.

Table 4.3: Metric values for test data.

Model	Accuracy	Recall	Precision	F1-score
ANN	42.773912926%	61.537153179%	4.494736293%	0.08377566595
GBM	96.566851991%	24.043585255%	83.373150374%	0.373235965

The ANN's low F1-score can be further demonstrated with its ROC-curve, which shows the true positive rate and false positive rate over different thresholds of the classification, which was defaulted to 0.5. This is shown in figure 4.5, which shows that the ANN is only slightly better than random guessing represented by the dotted line.

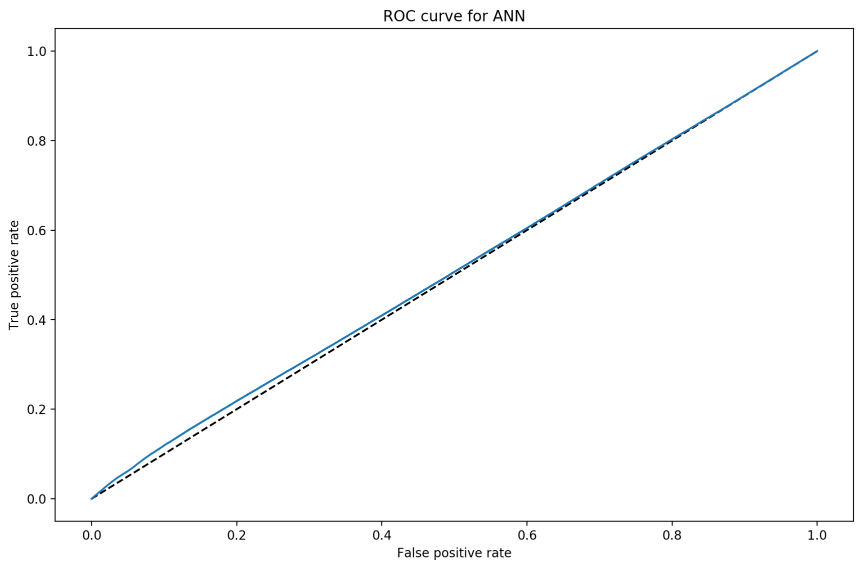

Figure 4.5: ROC of the ANN, where TPR and FPR are plotted against difference thesholds.

For the testing data, the ANN's has higher recall, but much lower precision. The extremely low F1-score shows that the model is overfitted, and is more inclined to classify data as "delayed". Since the delayed class is so rare, it accumulates in the low precision which lowers the F1-score significantly. The low F1-score proves that the ANN is not well suited for imbalanced data.

The GBM maintains a high precision and moderate recall when run on the testing data. Unlike the ANN, it is not overfit to the validation data and its performance is not affected much when introduced to new data, making it robust. Even with SMBO and dynamic resampling, the artifical neural network can not compete with the gradient boosting machine on this problem. A reason for this could be that the ANN does not train on as much data as the GBM per iteration. However, the dynamic resampling did not help at all but simply made it overfit to the training data, which implies that more data might not have been the solution to ANN's poor performance.

Due to GBM's natural handling of imbalance through the weighing of incorrectly classified data points, it outperforms the artifical neural network in F1-score as well as in time and implementation complexity, even with undersampled training and validation data for the ANN. However, neural networks are easily trained using batches which requires less memory. GBM's can be trained in distributed settings, but this requires a more complex implementation.

4.3 Discussion of Method

There are several aspects of the method which needs to be further explained and discussed, pertaining to the handling of class imbalance and hyperparameter optimization.

Class Imbalance

Of all the commonly used methods of handling class imbalance, undersampling was deemed to be the best fit for this study. The subset of the majority class was randomly sampled, using a method to make the subset as representative of the whole data set as possible would probably improve the model. Instead of sampling to make the two classes balanced, class weights could have been used in the implementation of the neural network, which would balance how much importance each training data point has to the training to make up for the imbalance. This would however require the model to train on the entire training data set for each iteration which is very time consuming.

The GBM being trained on the entire data set can be a factor in it outperforming the ANN. One can argue that the test is not fair as the GBM is exposed to more data, however since the ANN is also trained using dynamic sampling where it is exposed to more data, without any difference in performance, more data did not seem to improve the ANN. This could however be due to an unfitting implementation of the dynamic resampling. An investigation into how much undersampled data would have affected the GBM is something that would be interesting in this context, but was not done due to time limitation.

Hyperparameter Optimization

Even with a well performing optimization algorithm, there are several factors that could have affected the resulting hyperparameter selection negatively. Due to time constraint, the search space was limited. Tuning more than the chosen hyperparameters would probably improve the result; exploration of different batch sizes, types of regularization techniques, optimizers, initializations and such could have improved the results. The batch size is codependent on the learning rate and tuning both of these in the same optimization could give better results but would also take much longer time. The set batch size 2^{10} is chosen to have a balance between having the network converge as fast as possible, without having the training take too long. Since batch size and learning rate are dependent on each other, the batch size could not be decreased for the final model either even if the general case is that smaller batches produce better results.

The hyperparameter optimization was done with early stopping based on accuracy of a undersampled version of the validation data set. This was implemented both as a regularization technique, but also to save time as the optimization requires many iterations; early stopping does not allow the optimization to waste time on models that are clearly overfitted. However, early stopping can also be too restrictive as a model might converge after being stuck on a plateau of similar or decreasing validation accuracy values.

4.4 Handling of Limitations

During this study there was several limitations which affected the method choices and therefore the results, most of them directly or indirectly caused by the limited time span of the project as well as hardware resources in comparison to the size of the data set. For data preprocessing, compromises where necessary for sampling due to both time and hardware memory limitations. The optimization would have been more thorough and could have yielded a model closer to the global optimum if there was more time or if several machines where available for distributed optimization. A simple solution to this could have been to limit the data set to a subset based on for example a geographical area, such as country. This would give a smaller and a less variant data set, which is easier to analyze. However, this would result in a model only suitable for that geographical region.

The data set contains delivery events collected from June 2017 to September 2017. The model is also validated and tested on subsets from this period, so it wont affect the performance of the model in this study. It will however not perform as well with new data; and does not take into account seasonal changes.

4.5 Future Work

GBM's might be a better fit when working with imbalanced data for many problem formulations, but because of the flexibility and complexity of neural networks, there could be situations where a neural network with the right complements and preprocessing would outperform the GBM. Further research on neural networks for imbalanced data sets is vital before concluding the limitations they bring.

In the area of machine learning the problem is dependent on the available data, and a conclusion drawn based on one problem can rarely be applied to another problem if the data differs. Hence, hyperparameter optimization is imperative, especially with neural networks where the structure is flexible and can differ a lot in architecture and configuration. SMBO is a quite new approach in machine learning, further investigation of how it compares with classic optimization techniques could be useful as well as how the choice of surrogate function matters.

5 Conclusion

Even with measures to handle the class imbalance such as under sampling and dynamic resampling, the GBM outperformed the ANN. An ANN might have been sufficient, but due to the time consuming training of it and therefore hyperparameter optimization, a fitting ANN could not be identified in this study. It is clear that the GBM's natural handling of imbalance leverages the model against an ANN, regardless of the methods used to improve the ANN.

Using different measures to handle the class imbalance, such as class weights, might be a better solution as it allows the network to train on the entire data set. This would make the comparison between the two models more fair. More time and/or hardware resources would enable more thorough hyperparameter optimization, which could affect the outcome. Most importantly, the extra time and resources could enable the optimization to be run without early stopping, letting the network train longer for each optimization iteration would probably give better results overall for every iteration.

Bibliography

[1] PostNord in cooperation with Svensk Digital Handel and HUI research. *E-barometern*. 2017. URL: `https://www.postnord.se/siteassets/pdf/rapporter/e-barometern-arsrapport-2017.pdf` (visited on 04/04/2019).

[2] PostNord AB. *Ars- och hallbarhetsredovisning*. 2017.

[3] *PostNord AB*. URL: `https://www.postnord.se` (visited on 04/04/2019).

[4] Trevor Hastie, Robert Tibshirani, and JH Friedman. *The elements of statistical learning: data mining, inference, and prediction*. 2009.

[5] Christopher Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[6] Simon Haykin. *Neural networks: A comprehensive foundation*. Vol. 2. Pearson, 1999, pp. 1–23.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] Michael A Nielsen. *Neural networks and deep learning*. Determination Press, 2015.

[9] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations* (2014).

[10] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. "On large-batch training for deep learning: Generalization gap and sharp minima". In: *International Conference on Learning Representations* (2017).

[11] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[12] Cliff Click, Michal Malohlava, Arno Candel, Hank Roark, and Viraj Parmar. *Gradient boosting machine with H2O*. 2017.

[13] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. "Convergence analysis of distributed stochastic gradient descent with shuffling". In: *Neurocomputing* (2019).

[14] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016.

[15] Sergey Ioffe and Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. 2015.

[16] Zelda B Zabinsky. *Stochastic adaptive search for global optimization*. Vol. 72. Springer Science & Business Media, 2013.

[17] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[18] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems*. 2011, pp. 2546–2554.

[19] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Outlier detection: A survey". In: *ACM Computing Surveys* (2007).

[20] Marco Vannucci, Valentina Colla, Marco Vannocci, and Leonardo M Reyneri. "Dynamic resampling method for classification of sensitive problems and uneven datasets". In: *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Springer. 2012, pp. 78–87.

[21] David Cournapeau. "Sci-kit Learn". In: *Machine Learning in Python* (2015). URL: `https://scikit-learn.org/` (visited on 04/04/2019).

[22] "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283. URL: `https://www.tensorflow.org/` (visited on 04/04/2019).

[23] François Chollet et al. *Keras*. 2015. URL: `https://keras.io` (visited on 04/04/2019).

[24] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. "Towards an empirical foundation for assessing bayesian optimization of hyperparameters". In: *NIPS workshop on Bayesian Optimization in Theory and Practice*. Vol. 10. 2013, p. 3.

[25] Frank Hutter, Jörg Lücke, and Lars Schmidt-Thieme. "Beyond manual tuning of hyperparameters". In: *KI-Künstliche Intelligenz* 29.4 (2015), pp. 329–337.

[26] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. "Hyperopt: a python library for model selection and hyperparameter optimization". In: *Computational Science & Discovery* 8.1 (2015).

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[28] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[29] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. "The KDD process for extracting useful knowledge from volumes of data". In: *Communications of the ACM* 39.11 (1996), pp. 27–34.