FYS-STK4155 - Applied data analysis and machine learning

Project 3

Tommy Myrvik and Kristian Tuv
December, 2018

Github repository for the project:
https://github.com/Myrvixen/Fys-Stk4155-Project3

Contents

1 Abstract

We successfully reproduced the Neural Network, Logistic Regression and Decision Tree results of Yeh and Lien[1], which aims to predict credit card defaults, using gain charts and probability charts.
Efforts to outperform the initial Neural Network by using Random Forests and Support Vector Machines together with data preprocessing techniques normalization, standardization, oversampling, undersampling and SMOTE gave little to no increase in the accuracy of the model. As no techniques or learning algorithms seems to increase the accuracy we conclude that the features of the data set are likely not sufficient for predicting the probability of default of credit card holders.

2 Introduction

For a credit card issuing bank, predicting credit card payment default is naturally extremely critical for a successful business model. Finding accurate factors for why costumers are defaulting on their payment is not an easy job however. In this project we will try using a data set using credit related features of credit card holders in Taiwan in the early 2000 and compare our results with that of Yie and Lien[1] and hopefully outperform them.

2.1 Goals

Reproduce the result of Yie and Lien[1] for Neural Networks and Logistic Regression and see if we can find a way of improving the results by either pre processing the data or using Decision trees, Random Forests, Support Vector Machines.

2.2 Structure

We first go through the theory of Decision trees, Random Forests, Support Vector Machines and some Data Preprocessing. For the theory on Neural Networks and Logistic regression please see the the authors earlier projects on these subjects **Linear Regression** and **Neural Net**.
We go on to discuss the data set before going through our results and coclusions.

I Theory

As stated, we're going to use logistic regression, neural networks, classification trees and support vector machines when exploring the patterns and features of the credit card data. The former two methods has been used in earlier projects, thus for the theory behind these methods we'll refer to our reports on **Project1, Linear Regression** and **Project2, Neural Networks**. The latter two methods are new supervised machine learning methods for our studies, so we'll mainly focus on the theory behind these in this section.

3 Decision Trees

Decision trees is a widely used machine learning technique, which has many attractive properties, and can for many applications match the performance of a neural network. One of the main upsides of these methods are that they are very interpretable. While a neural network is more like a black box, where it's not easy to understands what happens in the processing in the hidden layers, the structure of a decision tree makes it easy to grasp what's actually going on.

Decision trees is normally split in two different classes, *regression trees* and *classification trees*. Their functions correspond to those of linear and logistic regression (used in Project 1 & 2), where the former tries to predict absolute values, and the latter focuses on classification of data. As the data set we're using in this project belongs to a classification problem we'll be using classification trees. The functionality of both kinds of trees are very similar however, so a description of one type will also give good insight in the other. There also exist a lot of different algorithms for creating the trees, but the general concept behind them are all the same. A few examples of the most popular ones are the ID3, CART and C4.5 algorithms, each with their pros and cons. The latter is however considered a superior version of the two other, fixing some issues the other two has in terms of flexibility in feature types (categorical/numerical), and overfitting. The method implemented in Scikit Learn is the CART algorithm, so we will also mention the characteristics of this method throughout the next sections. Below we'll discuss the inner workings of classification trees, and why they in many cases are viable options to other machine learning methods.

3.1 Classification Trees

Classification trees are based on the model finding which features, or which values of them, that split the data the best in terms of separating by their true labels. This is done by "asking questions" about the data like i.e. "is the value of this feature larger than 10?", which then will split the data into two groups, "True" or "False". This process is done recursively, so that both of the new groups will be asked new questions, which will further split the data. The splitting of each branch is performed until a stopping criteria is reached, which we will get to shortly. The resulting structure of the data processing is somewhat tree-like, giving reason for the name "decision tree". This tree analogy will also become apparent when introducing the names of different aspects of the decision tree. It's easiest to both explain and visualize a binary classification tree (aims to classify as one of two classes), so we'll stick to these types of trees in the coming explanations.

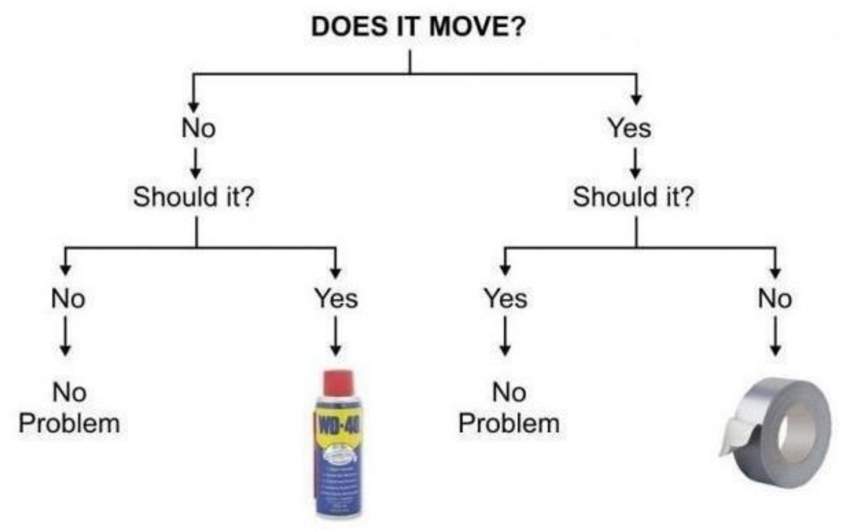

Figure 1: *A fun, little example of a classification tree. Can be made a binary tree by gathering the spray and tape into a common class "Fix Problem". Source: https://eight2late.wordpress.com/2016/02/16/a-gentle-introduction-to-decision-trees-using-r/.*

We have established that the splitting of the data is determined by asking questions, but how do we decide what question to ask, and how do we determine if it's a good question or not? The answer to the former is - we try everything! For each feature we store each different value which is found in the data. Many features are binary valued, meaning the example either has this feature, or it doesn't. This gives rise to *one* potential question, like i.e. "Is is sunny outside", which answers are either 'yes' (1) or 'no' (0). Other features can have several answers, for instance the color of a given fruit can be either "red", "yellow" or "green". These are three questions that can be asked in the form "is color == 'red'?". In case of numerical values we can store each value and ask "is feature >= value?", which also splits into two groups.

This is an area where the different algorithms differ. The CART algorithm *always* split in two branches at each step, where it picks the best *specific* question, and splits this into "True" and "False" groups. The ID3 and C4.5 algorithms however exhaust every possibility of each feature selected, and splits into the corresponding number of branches. If we take the example above of the color of a given fruit, these two algorithms will create three branches asking for each color, where each branch is split into True/False. CART will pick only one of the questions (from the pool of all possible questions of *all* features), meaning it can also ask questions about the same features at later stages.

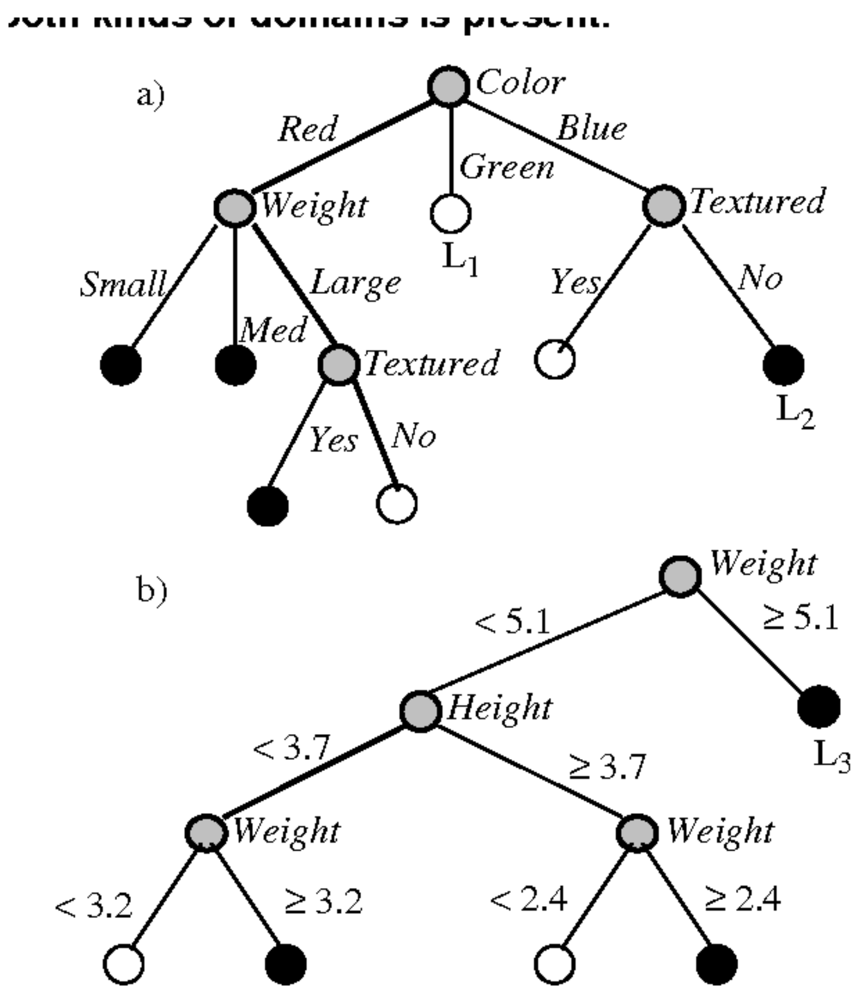

Figure 2: *Example of trees created by ID3 a), and CART b). ID3 explores each possibility of each feature at the same time, while CART picks one question at each feature. At the given depth, CART does not consider "Textured" as an important feature, while "Weight" is used several times.*

Now we need a measure on how good each of these accumulated questions are. In a decision tree we want *the best* (most impactful) questions to be asked *first*, in order to make the tree as efficient as possible. What we mean by this is that given a data set with examples labeled as 0 or 1, we want the question which separates the the 0s and the 1s the best, based on a question about one of their features. A popular measure of the quality of the split (used by CART as splitting criteria) is the *Gini impurity*, which is defined as:

$$G = \sum_{k=1}^{K} p_k(1-p_k) = \sum_{k=1}^{K} 1 - p_k^2 \tag{1}$$

where k runs over the labels of the data (only 0 and 1 in the binary case). p_k is the *probability* to find the label k within a group of the split, which is simply given as:

$$p_k = \frac{N_k}{\sum_i^K N_i} \tag{2}$$

where N_i is the number of samples with label i. This means that each True/False box has it's own Gini impurity, and the *total impurity* of the question will be the *weighted average* of the two boxes.

As seen from eq (1), a perfectly separated data set will give a Gini-score of 0, while the worst case (50/50 split of label 0 and 1) gives score 0.5. Hence, the question with the *lowest* Gini-score will be chosen as the question to ask at that step. This process is repeated for each branch at each step, building a tree from the top down. Each question, together with the data set to be split, is located at what is called a *node*. The node containing the *the first* question (the top of the tree) is called *the root node*.

Another measure of the "goodness" of a split is entropy, defined as

$$H = -\sum_{i=1}^{K} p_k \log_2 p_k \tag{3}$$

This is, in the same way as the Gini score, a measure of the impurity of the split. The question/feature giving the lowest entropy is in this case chosen as the question to ask.

The tree is continually being built until we at a node reach a stopping criteria. There are many different criteria being used, and they vary from algorithm from algorithm. One of these measure whether we *gain* any information by the splits being proposed. If the impurities of splits aren't *smaller* than the ones of the mother nodes, then there's no point in splitting the data further. A measure like this is called *information gain*, and is normally defined as the difference of the entropy of parent node, and the weighted sum of the entropy of the child nodes. If the information gain is 0 (or less), we will basically "override" the choice of question based on lowest Gini impurity, and instead stop further splitting.

Some algorithms (like CART) will keep splitting until either, the splits are completely *pure* (i.e. the resulting nodes only contain data belonging to one class), or there's only one sample left in each node (also considered as *pure*). That is, unless we put restrictions on them. If we allow the tree to split all the way down, it's easy for the model to overfit to the training data, creating a poorly generalized model. It's therefor normal to *prune* the tree, setting restrictions which result in smaller and simpler trees. When using CART, it's normal to set a maximum depth, i.e. the maximum number of splits the model is allowed to do before being forced to stop. One can also set a minimum on either, the number of samples needed to split further, or the number of samples in the resulting nodes. All these pruning methods does in many ways achieve the same thing however (i.e. preventing the trees to become too large), so it's usually sufficient to tune one or two of these hyperparameters.

These resulting "end nodes" are called *leaf nodes*, and this is where the actual classification will happen. After building the tree with all the available training data, we will have spread all training examples in all leaf nodes. In each leaf, we count up all training samples belonging to the different classes, and the class with the highest count will be the predicted class for a test sample ending up in that particular leaf. The confidence of the classification can also be calculated, which is simply the fraction of the training data with the given class.

There's a lot of advantages of using classification trees:

- As mentioned to the introduction to this section, trees are very easy to interpret, as they in many ways function like we humans do. Decisions are made directly through observations of the data features, by gradually eliminating the options that doesn't fit.
- We don't need to bother with normalization of the data, as each feature is treated separately.
- Models like CART are able to handle both numerical and categorical variables.

Pros however, are usually followed by cons, and that is also the case for decision trees:

- Trees are very prone to overfit to the training data, resulting in poorly generalized models. In this case careful tuning of hyperparameters may be necessary to prune the trees to a reasonable size.
- Decision trees are also prone to large variance, as small changes in the training sets may lead to completely different trees.
- Skewed data sets is likely to lead to biases trees towards the over represented class. As mentioned earlier, the classification in the leaves are determined by the dominant class. Naturally, the class with more samples is more likely to be dominant in the leaf nodes.

There are several methods to try to increase the robustness of decision trees, like *boosting* and *bagging*. A third one is *random forests*, which will be the subject of the next section.

3.2 Random Forests

A random forest is pretty much what it sounds like; a larger group of trees generated with some sort of randomness to them. The individual trees are created by following the same recipe as normal trees, but with some small tweaks.

Instead of building the trees from the training set as it is, a new data set is "bootstrapped" from it, and used to fit the model. So far, this also describes *bagging*, but a random forest has one additional stochastic element to it. In addition to bootstrapping the data sets, only a randomly selected subset of features are considered at each step of the building. This is done to counteract possible correlations of the features in the data set, as these features may not coexist in many trees. This will create an ensemble of different trees which all describe the data to some degree. The hope is that averaging the results from all trees in the forest will yield a well performing, low-variant result.

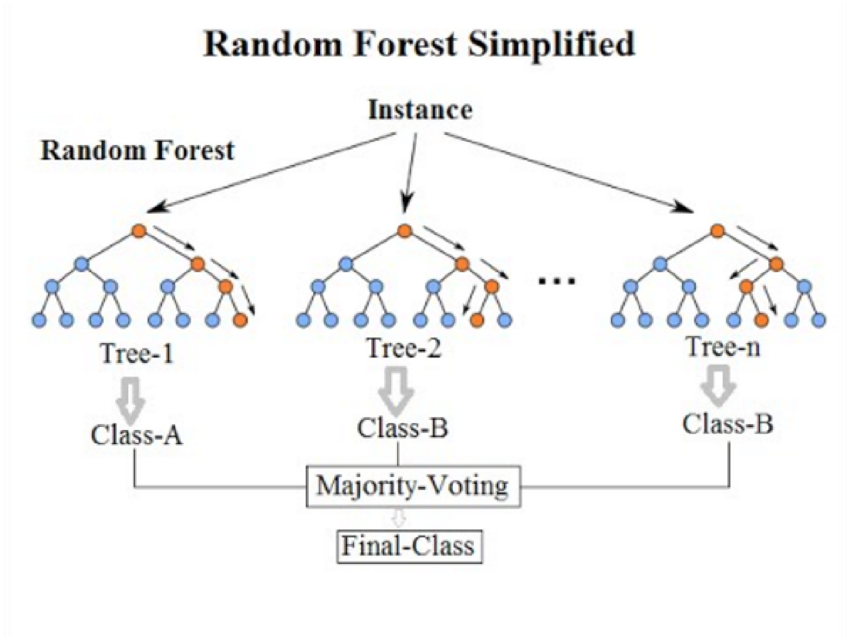

Figure 3: *A simple sketch of the workings of a random forest. A larger group of trees are created from bootstrapped data sets, and only subsets of the available features. For each new example, each tree "casts a vote" on which class it thinks the sample belongs to, and the most voted class will end up being the final result. Source: https://medium.com/@.*

The size of the subset of features is a hyperparameter that can be tweaked, but a good starting point for the size of the subset of features is $m = \sqrt{p}$, where p is the size of the full set [8]. As the forests consist of the same trees described in the previous section, the same hyperparameters can also be tuned in this case, where depth and minimum of samples may be the most prominent.

Once the forest is created, it can be used to classify new samples from a test set. Each sample is run through *every* tree in the forest, each giving it's own prediction. The predictions for each class are then counted up, and the class with the highest count is deemed "the winner". As the trees in the forest are likely to be quite different from each other, we will have a better reason to believe the "averaged" result of the forest rather than the result of a single classification tree. Hence, we expect the random forest to generally perform better. The "bootstrapped" nature of forests also makes them perform fairly well on sparse data sets.

The forests are still prone to performing quite bad when subjected to skewed data sets, as the bootstraping may cause the sparsely represented classes to appear even less, making them poorly represented in the forest overall. Using resampling techniques which aim to balance out the uneven data sets may help out the random forests to some degree.

4 Support Vector Machines

Rosenblatt's Perceptron Learning Algorithm[9] tries to find a separating hyperplane by minimizing the distance of misclassified points to the decision boundary. This algorithm is the starting point for the Support Vector Machine (SVM) algorithms and is valuable to understand in depth before venturing into the world of SVMs.

4.1 Rosenblatt Perceptron

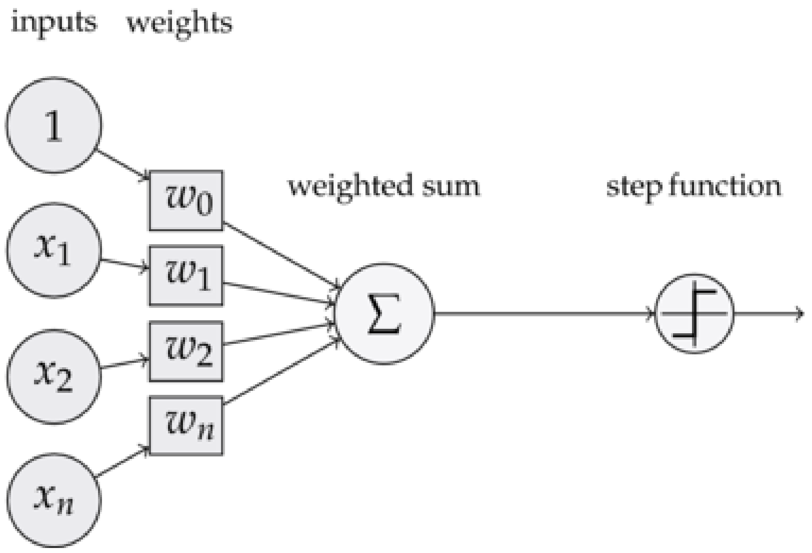

Figure 4: *A depiction of the Rosenblatt Perceptron learning algorithm. Source: Jessicayung.com*

We define a feature matrix

$$\mathbf{X} = [\mathbf{x_1} \quad \mathbf{x_2} \quad \mathbf{x_3} \quad ... \quad \mathbf{x_p}] \tag{4}$$

consisting of p feature vectors, each with n observations.
For the single perceptron algorithm shown in figure 4 our goal is to find a set of weights β, which when multiplied with a row of the feature matrix, creates a linear combination

of the features that correctly classify our predictions to the observed target values.
We can define this mathematically as

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} = 0 \tag{5}$$

where β_0 is the intercept. Written in a more compact way we have

$$f(\mathbf{x}) = \beta_0 + \beta^T \mathbf{x_i} = 0 \tag{6}$$

where $\mathbf{x}$ is a p-dimensional vector in consisting of p features. This is also the definition of a hyperplane or affine set L.
Now we need to use this result as a way of classifying our linear combinations. When working with binary target values, say $y_i = \pm 1$, we can define the following classification.

$$b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip} > 0 \rightarrow y_i = 1 \tag{7}$$

$$b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip} < 0 \rightarrow y_i = -1 \tag{8}$$

In other words we are trying to find a hyperplane in feature space which correctly separates our two classes.
Before we move on we will need the following properties of hyperplanes:

- For any two points $\mathbf{x}_1$ and $\mathbf{x}_2$ lying in L, $\beta^T(\mathbf{x_1} - \mathbf{x_2}) = 0$, and hence $\beta^* = \frac{\beta}{||\beta||}$ is a orthonormal vector to the surface L.
  This is easily seen when the intercept is zero as it equates to dot product of two orthogonal vectors which equals zero.
  This must also hold when adding the intercept as this equates to shifting the plane by a constant. The shifted plane is parallel to the original, hence the orthogonal vector is still orthogonal to the shifted plane.

- For any point $\mathbf{x}_0$ in L, $\beta^T \mathbf{x_0} = -\beta_0$.
  This follows naturally from the definition of f($\boldsymbol{x}$)

- The signed distance of any point $\mathbf{x}$ to L is given by

$$\beta^{*T}(\mathbf{x} - \mathbf{x}_0) = \frac{\beta^T \mathbf{x} + \beta_0}{||\beta||} = \frac{f(\mathbf{x})}{||f'(\mathbf{x})||} \tag{9}$$

  Hence f($\mathbf{x}$) is proportional to the signed distance from x to the hyperplane defined by f($\mathbf{x}$) = 0
  The signed distance is the distance to the boundary of L. This result follows from the first two properties and is what is used for minimizing the distance of the misclassified points to the decision boundary

As stated in the introduction we want to minimize the distance of the misclassified point to the decision boundary. Using our definition of y_i we know that if a response $y_i = 1$ is misclassified then $\beta_i^T + \beta_0 < 0$ and the opposite is true when misclassifying

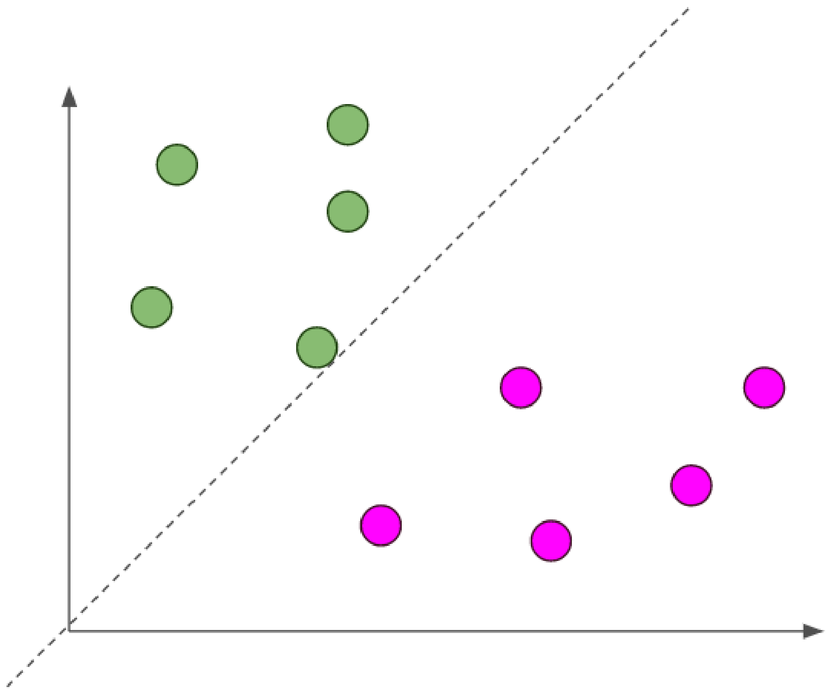

Figure 5: Depiction of the decision boundary between two classes. Source: Pythonmachinelearning.pro

$y_i = -1$.
We can now define a distance function using the result in equation 9

$$D(\beta, \beta_0) = -\sum_{i \in \mathcal{M}} y_i(\beta^T \mathbf{x}_i + \beta_0) \tag{10}$$

Where $\mathcal{M}$ is the set of misclassified points and $\beta^T \mathbf{x} + \beta_0 = 0$ is the decision boundary. The negative sign is for ensuring the distance will be positive for both misclassfications, as this definition will otherwise yield a negative result.
To minimize this equation we simply take the derivatives of D and use stochastic gradient descent for minimization (defined in project 2).
There are some notable problems with this algorithm:

- When the data is serperable, there are an infinite number of solutions, and which one we end up with depends on the starting conditions
- It can be show that if the data is linearly separable, the algorithm will converge in a finite number of steps. However the number of steps can be very large.
- When the data is not separable, the algorithm will not converge.

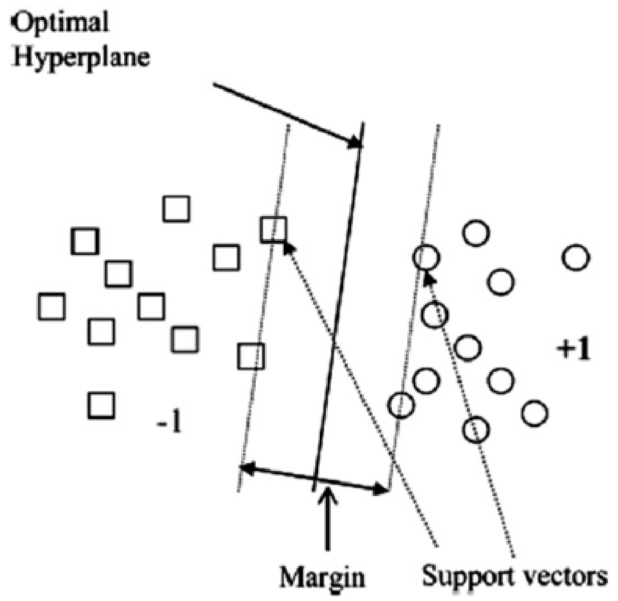

Figure 6: *The optimal hyperplane wich maximizes the distance between the classes in question. Source: Researchgate.net*

4.2 The Optimal Hyperplane

A solution to the first notable problem defined in the previous section is to maximize the distance to the closest point of each class. That is, finding the hyperplane which exactly splits the distance between the classes in half, as demonstrated in figure 6.

In equation 10 our goal was to find a hyperplane where no points were misclassified. However the distance between the correctly classified points and the hyperplane had no constraints, and the points could in practice be right next to or even on the hyperplane. This will perfectly classify a training set but there is a big chance it will not generalize very well.

Our goal now is the find the largest margin M which separates the two classes. We do this by solving the following optimization problem

$$\max_{\beta,\beta_0,||\beta||=1} M \tag{11}$$

$$\text{subject to } y_i(\beta^T \mathbf{x}_i + \beta_0) \geq M \tag{12}$$

The β in question is normalized as we see from the $||\beta|| = 1$ constraint. We can get rid of this constraint by simply multiplying the right hand side with the norm of β. Leaving us with the equation

$$y_i(\beta^T \mathbf{x}_i + \beta_0) \geq M||\beta|| \tag{13}$$

We can now arbitrarily set $M = 1/||\beta||$. We can do this because given a β and β_0 which satisfies the inequality, any positively scaled multiple of β and β_0 will also satisfy

the inequality.
We can now reformulate equation 11 as

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 \tag{14}$$

$$\text{subject to } y_i(\beta^T x_i + \beta_0) \geq 1, i = 1, 2, \ldots, N \tag{15}$$

Because the square root in the norm is redundant when we are optimizing, we choose to optimize the squared norm instead as this will yield prettier results when taking the derivative. The factor $\frac{1}{2}$ is also just a ”prettyfier” and does not change the result.
We already discussed in section 4.2 that the weight vector β is normal to the hyperplane, and when removing the normalization we now want to minimize this orthogonal vector because this equates to maximizing $M = 1/||\beta||$.
An optimization problem of the form of equation 24 is a convex optimization problem[5], which means we can solve it using the Lagrangian

$$L(\beta, \beta_0) = \frac{1}{2}||\beta||^2 - \sum_{i=1}^{N} \alpha_i[y_i(\beta^T x_i + \beta_0) - 1] \tag{16}$$

under the condition $\alpha_i > 0$, where α_i is called the Lagrange multipliers. This is dummy variables which is used as a help for calculating the variables we truly care about[3]
Setting the derivatives to zero we get

$$\frac{\partial \mathcal{L}}{\partial \beta_0} = 0 = -\sum_{i=1}^{N} \alpha_i y_i \tag{17}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = 0 = \beta - \sum_{i=1}^{N} \alpha_i x_i y_i \tag{18}$$

And by substituting into the original Lagrangian

$$L_W = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} \alpha_i \alpha_k y_i y_k x_i^T x_k \tag{19}$$

subject to $\alpha_i \geq 0$ as before and the new condition $\nabla L = 0$
This result is called the Wolfe duality[6] and the fact that we now have two constraints, one inequality constraint and one equality constraint means we now are subject to the Karush-Kuhn-Tucker conditions(KKT)[4] in the local optimum. KKT uses the constraint already in place, but adds the condition of complementary slackness

$$\alpha_i[y_i(\beta^T x_i + \beta_0) - 1] = 0 \quad \forall i \tag{20}$$

We do not concern our self with an exact solution of this problem, however we can extract some information by just looking at the conditions we have defined

- if $\alpha_i > 0$, then $y_i(\beta^T\mathbf{x} + \beta_0) = 1$, hence the vector $\mathbf{x}_i$ is situated on the boundary of the margin
- if $y_i(\beta^T\mathbf{x} + \beta_0) > 1$, $\alpha_i = 0$ and $\mathbf{x}$ is situated outside the boundaries of the margins

The points x which are closest to the margins are what we call the support vectors and we can use equation 18 and equation 20 for calculating β and β_0 giving us the following expressions

$$\beta = \sum_i \alpha_i y_i x_i \tag{21}$$

$$\beta_0 = \frac{1}{y_i} - \beta^T x_i \tag{22}$$

To find the expression of β we of course need to solve the Wolfe equation 19 and find the Lagrange multipliers, but assuming we are able to do this we can now classify our observations by using

$$G(x) = sign(\beta^T x_i + \beta_0) \tag{23}$$

4.3 Support Vector Classifier

In section 4.2 we defined the optimization problem of finding the maximum margin by

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 \tag{24}$$

$$\text{subject to } y_i(\beta^T x_i + \beta_0) \geq 1, i = 1, 2, \ldots, N \tag{25}$$

One assumption which was made when defining this optimization problem was that the classes in question was perfectly linearly separable. This is however rarely the case, as real features will have noise and errors, hence the classes might overlap. The solution to this problem is to introduce a slack parameter ξ which allows for a fixed number of points to be of the wrong side of the support vectors as shown in the right panel of figure 7.

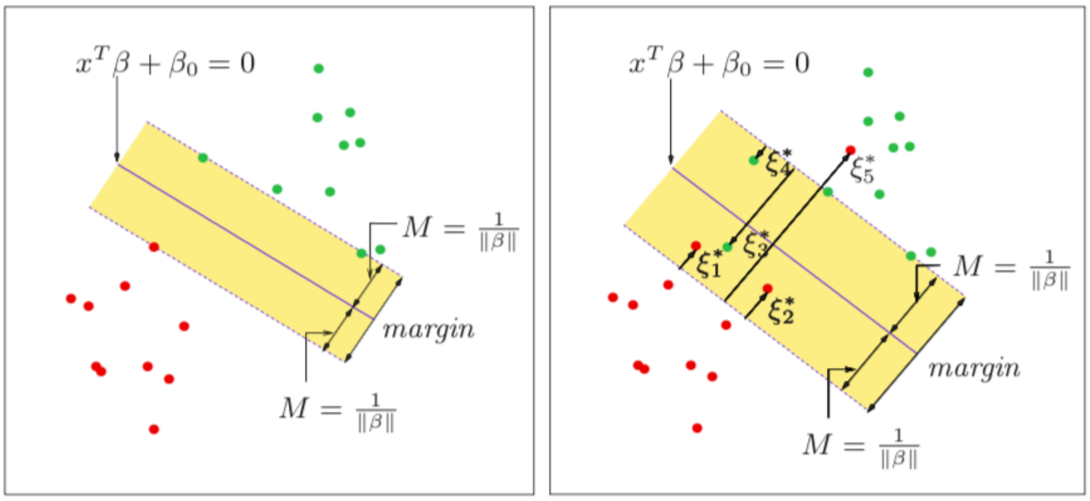

Figure 7: Left side: Depiction of the optimal hyperplane defined in section 4.2. The vectors situated on the margin is called the support vectors.
Right side: Introducing the slack parameter ξ which allows for a fixed number of points to be on the wrong side of the margin.
Source: The Elements of statistical learning, Hastie[7]

We define $\xi = (\xi_1, \xi_2, \ldots, \xi_N)$ and subtract it from the margin

$$y_i(\beta^T x_i + \beta_0) \geq M - \xi_i \; \forall i, \quad \xi_i \geq 0, \quad \sum \xi_i \leq c \tag{26}$$

or

$$y_i(\beta^T x_i + \beta_0) \geq M(1 - \xi_i) \; \forall i, \quad \xi_i \geq 0, \quad \sum \xi_i \leq c \tag{27}$$

Where ξ_i is the slack parameter and c is some predefined constant.
We prefer the latter equation as it still gives a convex optimization problem. We redefine M as we did in section 4.2, $M = 1/||\beta||$ which yields the following problem

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 \quad \text{subject to} \begin{cases} y_i(\beta^T x_i + \beta_0) \geq 1 - \xi_i & \forall i \\ \xi_i \geq 0, \quad \sum_i \xi_i \leq c \end{cases} \tag{28}$$

Or written in a different way

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \xi_i \tag{29}$$

$$\text{subject to } \xi_i \geq 0, \; y_i(\beta^T x_i + \beta_0) \geq 1 - \xi_i \; \forall i \tag{30}$$

Where C replaces the $\sum \xi_i \leq c$ constraint and is essentially a regularization hyperparameter defining how much error we tolerate in the algorithm. As we are minimizing the expression we see there will be a trade off between the minimal $||\beta||$ and the error made. With a large C, the algorithm will seek a solution with fewer errors, hence searching for a smaller margin. With a smaller C the tolerance of errors are larger and the margins

can increase.
This problem is then solved by using the Lagrangian and the Wolfe dual as done in section 4.2

$$L = \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i(\beta^T x_i + \beta_0) - (1-\xi_i)] - \sum_{i=1}^{N}\mu_i\xi_i \tag{31}$$

With derivatives

$$\beta = \sum_{i=1}^{N}\alpha_i y_i x_i \tag{32}$$

$$\sum_{i=1}^{N}\alpha_i y_i = 0 \tag{33}$$

$$\alpha_i = C - \mu_i,\ \forall i \tag{34}$$

Inserting the derivatives gives again the wolfe dual

$$L_W = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{N}\alpha_i\alpha_k y_i y_k x_i^T x_k \tag{35}$$

The KKT conditions gives

$$\alpha_i[y_i(\beta^T x_i + \beta_0) - (1-\xi_i)] = 0 \tag{36}$$

$$\mu_i\xi_i = 0 \tag{37}$$

$$y_i(\beta^T x_i + \beta_0) - (1-\xi_i) \geq 0 \tag{38}$$

We use the same techniques as in section 4.2 for finding β adn β_0, and given these solutions we can classify our observations by

$$G(x_i) = sign(\beta^T x_i + \beta_0) \tag{39}$$

This classifier is also referred to as a soft classifier, while the optimal hyperplane is called a hard classifier as it does not allow any points to cross the margin. As $C \to \infty$ these two classifiers will converge.

4.4 Kernels

In the sections on optimal hyperplanes and support vector classifiers we worked with data which was linearly separable. Now we take these algorithms a step further by using kernels for transforming our feature vectors x_i into higher dimensions. This extension of the previous algorithms is what constitutes the*Support Vector Machine* classifier.

In equation 19 (the Wolfe dual) we calculate the inner product between all the features.

$$\langle x_i, x_k \rangle \tag{40}$$

If the data is not linearly separable we transform the features into higher dimensions as seen in figure 8, using basis functions

$$\phi(x_i) \tag{41}$$

The problem is finding a the transformation function $\phi(x)$, however using Mercer's theorem defined in the text box we do not need to know the shape of $\phi(x)$, we only need to find a Kernel which satisfies the Mercer conditions.

$$K(x_i, x_k) = \langle \phi(x_i), \phi(x_k) \rangle \tag{42}$$

Poplular choices for the the Kernel is
dth-Degree polynomial:

$$K(x_i, x_k) = (1 + \langle x_i, x_k \rangle)^d \tag{43}$$

Radial basis functions:

$$K(x_i, x_k) = exp(-\gamma ||x_i - x_k||^2) \tag{44}$$

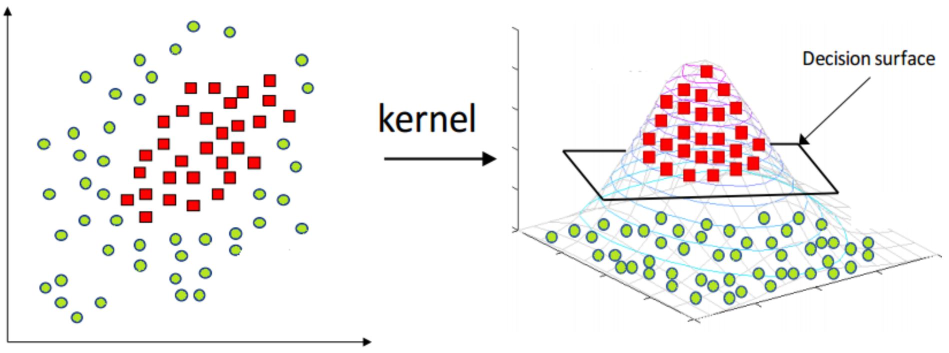

Figure 8: *Transformation of data in two dimensions into three dimensions which now makes it linearly separable. Source: Hackerearth.com*

Mercer's Theorem[10]
If a function K(a,b) is continuous and symmetric in its arguments, then there exists a function ϕ that maps a and b into another space s.t $K(a, b) = \phi(a)^T \cdot \phi(b)$.
This means we can use K as a kernel since ϕ exists, we just do not know what it is.
A note is that a Kernel does not necessarily have to respect Mercer's conditions to work as a kernel, but a function which respectes the conditions can definitively be used.

5 Data preprocessing

Machine Learning algorithms tend to learn more effectively if the inputs and targets are preprocessed before the network is trained. If the data points given have vastly different scales, preprocessing might be necessary for any learning to happen at all. Here we will look at some of the techniques used in this project.

5.1 Min-max normalization

Min-max normalization aims to scale all numerical values of a feature into a given range. For distance based learning algorithms like k-nearest neighbors this is obviously extremely important as it is normally evaluated by using the euclidean norm. When calculating the norm, having features in different scales will largely impact the result as the feature with the biggest scale will dominate.
In theory this is not a problem for a Multi Layered Perceptron as rescaling a feature is equivalent to adjusting the weight matrices of the network, however not rescaling the data can be risky as the network might use considerably longer time to reach a minimum. This effect is not hard to imagine when considering the initialization of the weight matrices is generally just set to random small numbers. If there is a large discrepancy between the values of two input features this is likely to lead to saturation for activation function like tanh or sigmoid, and exploding gradients or dead neurons for relu activations. This normalization technique is easily implemented using scikit-learn's MinMaxScaler.

$$\mathbf{x}' = \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})} \tag{45}$$

Where $\mathbf{x}'$ is the normalized feature vector and $\mathbf{x}$ is the original feature vector.
Most literature, when referring to normalization, are concerning a particular kind of min-max normalization, that is a normalization where the final interval is $[0, 1]$ which is also the definition we are using in this project.

5.2 Standardization

In some cases the min-max normalization is not useful or cannot be applied. If the minimum and maximum values of a feature is not know, using these values for normalization is obviously not possible. Also, if the feature data contains considerable outliers this can greatly bias the input data.
A better approach might be to use the standardization of the data.

$$\mathbf{x}' = \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sigma_{\mathbf{x}}} \tag{46}$$

Where $\mathbf{x}'$ is the standardized feature vector, $\mathbf{x}$ is the original feature vector, $\bar{\mathbf{x}}$ is the average of the feature vector and $\sigma_{\mathbf{x}}$ is the standard deviation. In practice the sample standard deviation is used for the calculation as the population standard deviation generally is not known.

5.3 Resampling

A great challenge in machine learning is how to handle skewness of data. Possible solutions are to randomly oversample with replacement from the underrepresented class, or to randomly undersample from the overrepresented class. A third option is to create new feature vectors using the SMOTE algorithm.

5.3.1 Synthetic Minority Over-sampling Technique (SMOTE)

The weakness of random under- and over sampling is that no new linear combinations of features are added to the data set, only random examples existing combinations are added.
With the SMOTE algorithm, developed by Chawla et al[11], we try to create new linear combinations f from the existing combinations.
We take the difference between the f-vector under consideration and its nearest neighbor. Multiply this difference by a random number between 0 and 1, and add it to the f-vector under consideration. This causes the selection of a random point along the line segment between two specific features.

Example with two feature vectors:
We imagine having two features with two entries $x_1 = (1, 3)$ and $x_2 = (2, 4)$, giving the f-vectors $f_1 = (1, 2)$ and $f_2 = (3, 4)$ in feature space. Because we only have two points, they naturally are each others nearest neighbors.
Following the smote algorithm we now find the distance between the nearest neighbors

$$f_2 - f_1 = (3, 4) - (1, 2) = (2, 2) \tag{47}$$

The new added f-vector now becomes

$$f_{new} = f_1 + r * (2, 2) = (1, 2) + r(2, 2) \tag{48}$$

Where r is a random number drawn from the unifrom distribution between 0 and 1.

6 Data evaluation

6.1 Cumulative gains chart

If the data set we're working with is skewed, a normal accuracy metric wouldn't be the optimal measure for model performance (assuming no resampling). If say 4/5 of the data has label 0 and the rest 1, a completely useless model which will guess label 0 no matter what the input data looks like will suddenly guess correctly 80% of the time. As this "bias" exists, it's difficult to tell whether the model has learned to separate the two classes or not. We should hence use other metrics, and one used by Yeh, Lien[1] was *cumulative gains chart*. This a type of *lift* chart, which measures the effectiveness of the model as a ratio between the results obtained by the model, and guessing randomly.

To make a gain chart, we have to pick *one* target, i.e. one type of label (0 or 1 in the binary case). A gain chart is usually generated for the label with lesser examples, being the "defaulted" (1) class in the case of the data set used in this project. The evaluation goes as follows.

- Firstly, fully train the model being used
- Use the trained model to calculate probabilities for each sample for the class in question, and sort these from highest to lowest
- Gradually pick out larger subsets of the **top** of the full set of sorted probabilities, i.e. top 10%, top 20%, etc.
- For each iteration in the previous step, count the number of samples in the subset which actually is labeled as the wanted class (target data), and give the result as a fraction of the total number of samples in the **full** data set with the same label.
- Plot the fraction of target data found as a function of the size of the subset for each iteration.
- Calculate the area between the resulting curve and the baseline (explained below). Also calculate the area between the optimal curve (also explained below) and the baseline, and the ratio between areas is the final evaluation of the model.

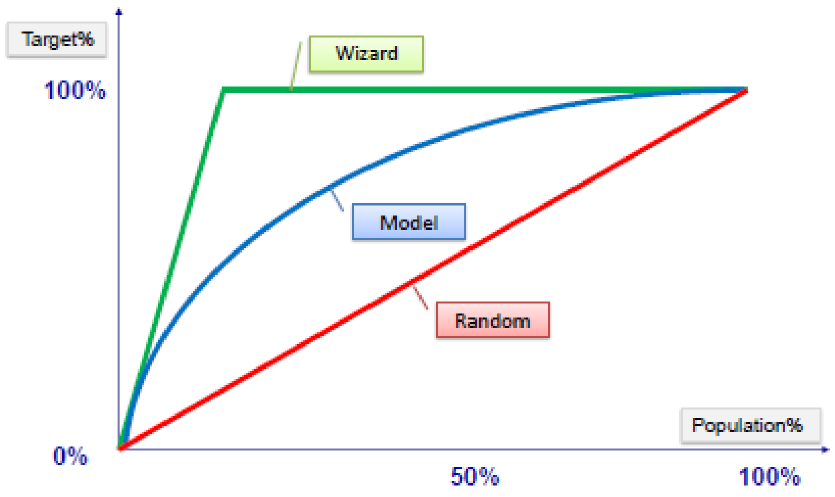

Figure 9: *Schematic plot of a gain chart. The red line is the "baseline", the blue the performance of out model, and the green one labeled "wizard" is the optimal performance. We can't speak for the creator, but we're guessing he/her believes there has to be some magic involved in order to classify every sample correctly (which we partly agree on). Source: https://www.saedsayad.com*

Figure 9 shows a simple concept plot of what a gain chart looks like. The red line labeled "random" is called the *baseline*, which shows the fractions of the found target data when we just randomly pick samples from the full data set. This is always assumed to be linear from 0% to 100%, no actual computations are done. The green line is the optimal performance of a model, which would find *all* target data in listed at the top of the sorted probability list. A model which has learned *anything* about the data will thus live between these two lines, i.e. inside the triangle drawn in Figure 9. The better the model, the higher *lift* the model curve will have, and the higher area ratio.

6.2 Predictive accuracy of probability

Another metric is one that evaluates the quality of the predicted probabilities directly. To make this assessment, we naturally need some actual probabilities to compare with. Problem is, the real probabilities aren't known, so we would need to find estimates for these instead. To find these estimates, Yeh, Lien[1] used the Sorting Smoothing Method (SSM), which gives estimated probabilities:

$$P_i = \frac{Y_{i-n} + Y_{i-n+1} + \cdots + Y_{i-1} + Y_i + Y_{i+1} + \cdots + Y_{i+n-1} + Y_{i+n}}{2n+1}. \tag{49}$$

This estimation assumes that the list of sorted (predicted) probabilities is in the cumulative gains estimation, but here Y_j is the *true* labels of the data. $Y_j = 1$ if j correspond to the class we're predicting, and $Y_j = 0$ otherwise. n is the number of neighboring labels (on each side) in the prediction list we're considering when calculating the probabilities. In the mentioned article they used n, so we will stick with this value throughout the project as well. We can then proceed to plot the estimated probabilities versus the predicted probabilities, and see how they match up. If they match exactly, $P_{i,estimated} = P_{i,predicted}$ for all i, create a straight line given by $y = x$ (slope 1 and intercept 0). Unless the model is perfect however, this will not be the case. We still want to see they relate linearly, hence the solution is to use linear regression to find the straight line that fits the best. The resulting line may in some cases be "perfect" however, so we need to use a metric like the R2-score to determine how well the line actually fits the data.

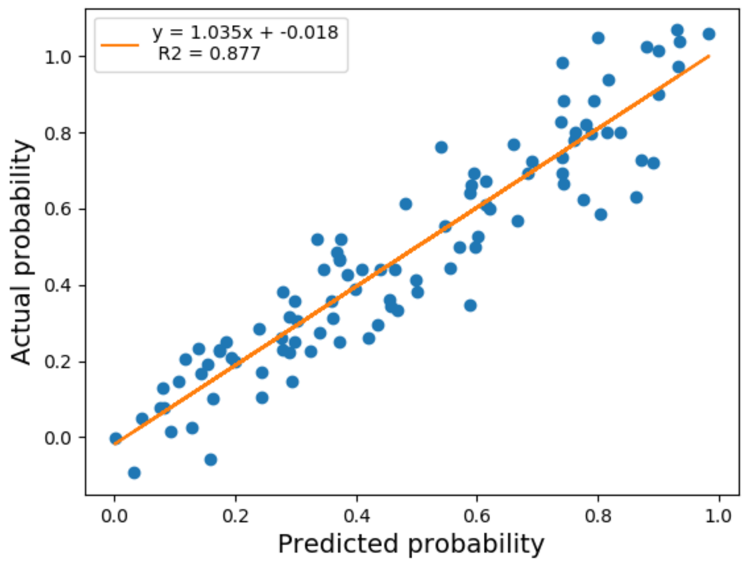

Figure 10: *An example of what a probability plot might look like. The dots show predicted versus estimated probabilities, while the line is the result of linear regression. The function y(x) for the line is shows, as well as the R2-score for this line.*

II Implementation and results

7 Implementation and data sets

7.1 Credit card data set

The data set we're going to study in this project is a set describing defaults of Taiwanese credit card client, based on a number of predictors like their sex, education and payment history. The data was produced from a period in the early 2000's where cash and credit card debt crisis. In order to increase market share, card-issuingbanks in Taiwan over-issued cash and credit cards to unqualified applicants.
The data set can be found here. An example of the data of a client is shown in Table 1.

X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14
50000	1	2	1	57	-1	0	-1	0	0	0	8617	5670	35835

X15	X16	X17	X18	X19	X20	X21	X22	X23	Y
20940	19146	19131	2000	36681	10000	9000	689	679	0

Table 1: *An excerpt from the data set. The different values correspond to:*
X1: Credit limit in NT dollars. It includes both the consumer's own credit and the credit given to family members.
X2: Gender (1 = male; 2 = female)
X3: Education (1 = graduate school; 2 = university,; 3 = high school; 4 = others)
X4: Marital status (1 = married; 2 = single; 3 = others)
X5: Age (years)
X6-X11: History of past payments from months leading up to September 2005
(-1 = pay duly; 1+ = payment delay for x months, -2 = not given by the data set creators but assumed to be "No loan")
X12-X17: Amount of bill statement (NT dollars) from the same months.
X18-X23: Amount of payment (NT dollars) from the same months
Y: Whether the person defaulted or not (1 = Yes; 0 = No)

What should strike machine learning enthusiast (like ourselves) first is the huge differences in ranges of the values of the predictors. Training on the data set as it is will not only require more time for the model to fit the large ranges, but will also introduce huge variances into it. It's therefor important to use normalization techniques described in Section 5 to make the data easy to digest for our models (except for decision trees, which we saw in Section 3, don't have this problem).

Another important aspect of this data set is that it's unbalanced/skewed, i.e. it has a lot more samples labeled as 0 (no default) than samples labeled as 1 (default). Counting up only yields 6636 defaults out of 30000 samples, only 22.12%. As mentioned in Section 6, we should use different metrics than the regular accuracy metric to measure the performance of our models. We can also try to use resampling techniques described in Section 5.3 to even out the difference, but any improvements in performance based on this heavily relies on the quality of the original data set.

8 Results

8.1 Logistic Regression & Neural Network

Firstly, we sought to reproduce the results given in Yeh, Lien[1] for logistic regression and neural networks. To optimize we used mini-batch stochastic gradient descent, as we in **Project 2** argued that this is the best and most robust of the normal gradient descent methods. We also argued that a batch-size between 10 and a couple hundred is a good size to start with, which also seems to be the consensus in the machine learning community.

As expected, logistic regression did not perform very well on the credit card data set, as was also shown in [1]. If we compare our results in Figure 11 and 12 with the corresponding plots in the article, we see that they are very similar. The similarities are also confirmed with the numerical results of the experiments, which are listed in Table 4.

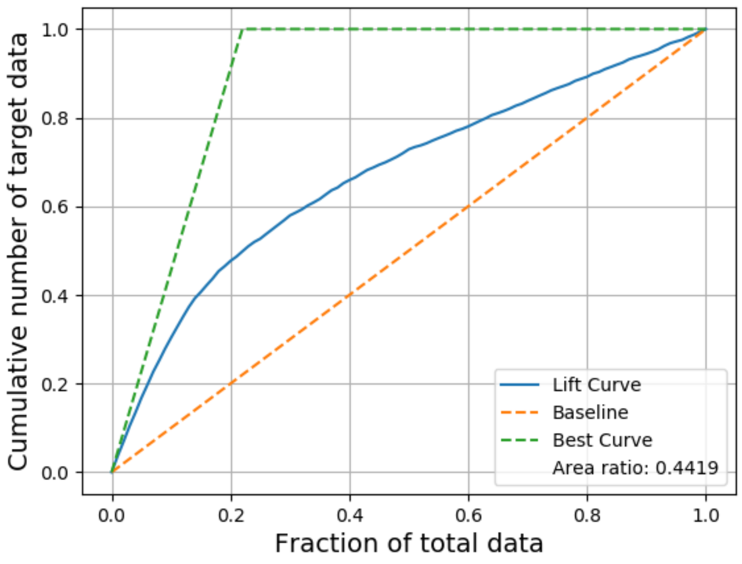

Figure 11: *Cumulative gains chart of the results from* ***logistic regression****.*

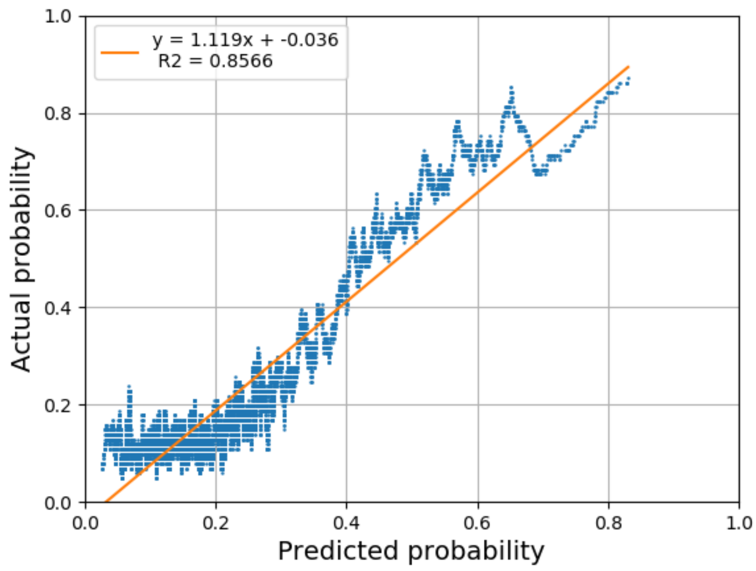

Figure 12: *Estimated probability vs predicted probability of the results from* ***logistic regression****.*

Authors	Error rate	Area ratio	R2-score	Reg. Coefficient	Reg. Intercept
Yeh, Lien(2009)[1]	0.18	0.44	0.794	1.233	-0.052
Myrvik, Tuv(2018)	0.191	0.442	0.857	1.119	-0.036

Table 2: *Comparison of the results presented in [1] and our own results for* ***logistic regression****. Error rate is the fraction of misclassifications by the model. All values are extracted from the performance on the validation/test set.*

We also found that regularization had no positive effect on the results, rather the opposite. Even the slightest regularization made the results worse than they would be for Ordinary Least Squares, meaning we omit using regularization at all for this data set. Examples of the examples of regularization can be found in log_regular.txt. We also found the same effect when using neural networks.

For neural networks we explored a lot of different architectures, in hopes of finding a certain type that is able to describe the data well. We ran tests for three different activation functions; tanh, sigmoid and relu, with a set of different numbers of hidden layers and nodes for each. Some of the best results is shown in Table 3.

Activation	H. layers	Nodes	Area Test	R2 test	Error rate
tanh	2	5	0.46843	0.95650	0.183
tanh	2	10	0.48612	0.95801	0.178
tanh	2	20	0.47938	0.94087	0.184
tanh	3	50	0.48931	0.95707	0.181
tanh	3	100	0.48857	0.96337	0.182
sigmoid	1	5	0.44705	0.86410	0.187
sigmoid	1	10	0.43648	0.88426	0.196
sigmoid	1	20	0.43760	0.84709	0.190
sigmoid	2	50	0.43020	0.85514	0.197
sigmoid	2	100	0.44161	0.84924	0.191
relu	1	5	0.43835	0.93106	0.186
relu	1	10	0.45868	0.93831	0.186
relu	1	20	0.46837	0.94844	0.182
relu	3	20	0.47273	0.95005	0.183
relu	2	100	0.47651	0.95984	0.181

Table 3: *The 5 best results for each activation function when considering number of hidden layers [1, 2, 3] and number of nodes [5, 10, 20, 50, 100]. Run for 2000 epochs with stochastic gradient descent. Running for more epochs will yield better results, but as convergence rate is also a factor, it's useful to explore the performance for fewer epochs. The tanh activation is shown to perform clearly better overall. The full set of results can be found in nn_architectures.txt.*

From the table, we see that in terms of activation functions, the tanh-function outperform the other two quite consistently. We also see that we don't need a very complex model in order to peak in performance. We get just as good results with 2 hidden layers with few nodes, as with 3 layers with many nodes. The lower number of weights we use, the less we risk introducing large variances into or model. From our results, a neural network with 2 hidden layers with 10 nodes in each seems like a good choice of architecture.

After deciding on all hyperparameters of our network, we trained our model for a larger number of epochs in order to (hopefully) find the global minimum, and the best description of the data as possible. We upped the number of epochs to 10000, and the results can be seen in Figure 13 & 14.

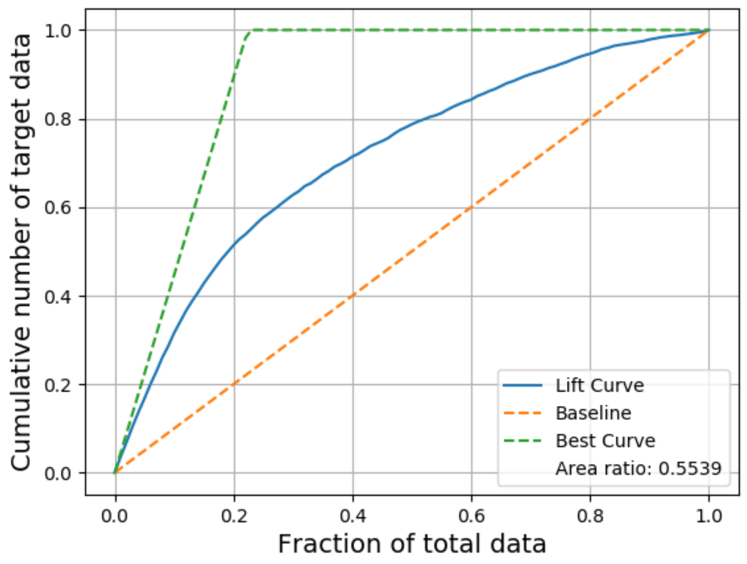

Figure 13: *Cumulative gains chart of the results from **neural network**.*

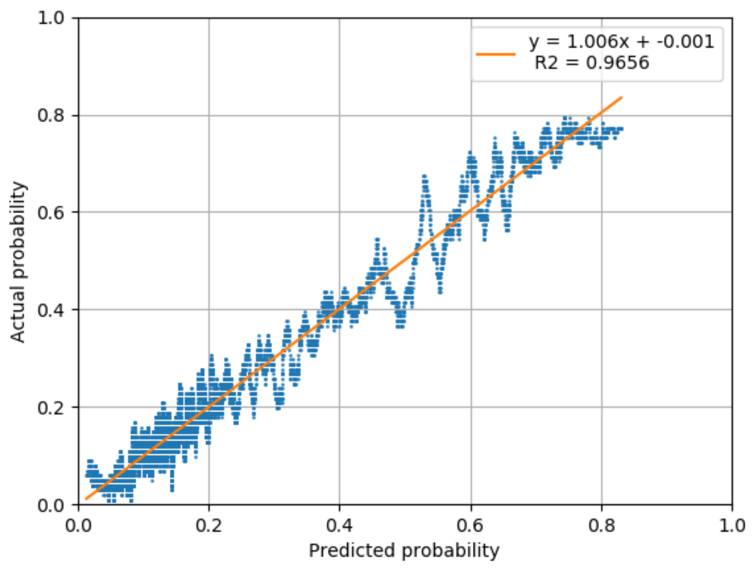

Figure 14: *Estimated probability vs predicted probability of the results from **neural network**.*

We see that the neural network performs a lot better than logistic regression, but still not impressive. This is neither expected, since the results we got match up almost perfectly by what was found by Yeh and Lien [1].

Authors	Error rate	Area ratio	R2-score	Reg. Coefficient	Reg. Intercept
Yeh, Lien(2009)[1]	0.17	0.54	0.965	0.998	0.0145
Myrvik, Tuv(2018)	0.168	0.554	0.966	1.006	-0.001

Table 4: *Comparison of the results presented in [1] and our own results for **neural network**. Error rate is the fraction of misclassifications by the model. All values are extracted from the performance on the validation/test set.*

8.2 Classification Tree & Random Forest

Next we explored how decision trees perform on the credit card data set, in forms of a single classification tree, and a random forest. For this, we used Scikit Learn's functions, and focused on tuning the hyperparameters of the trees in order to get the best result possible. As stated in Section 3, there are several parameters we can tune, but some of them end up doing more or less the same thing, i.e. pruning the trees. We have decided to use maximum depth as the main parameter, and minimum required samples to split as a small adjustment parameter for both methods. The outcomes of the analysis can be seen in Appendix A.1.

These plots show that a depth of 4-6 seems to be the optimal depth for a decision tree. Also setting minimum samples to around 200 seems to increase the performance another notch. Now running the decision tree with these hyperparameters on the final test set, we get the plots shown in Figure 15 and 16.

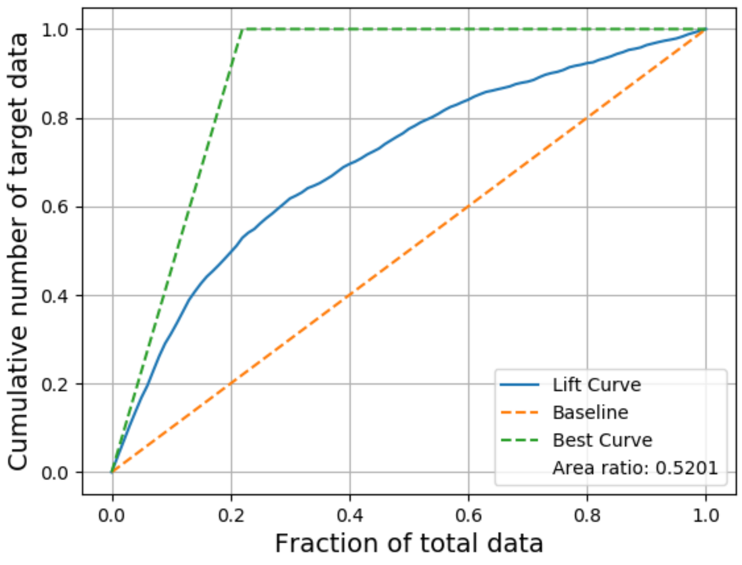

Figure 15: *Cumulative gains chart of the results from **decision tree**.*

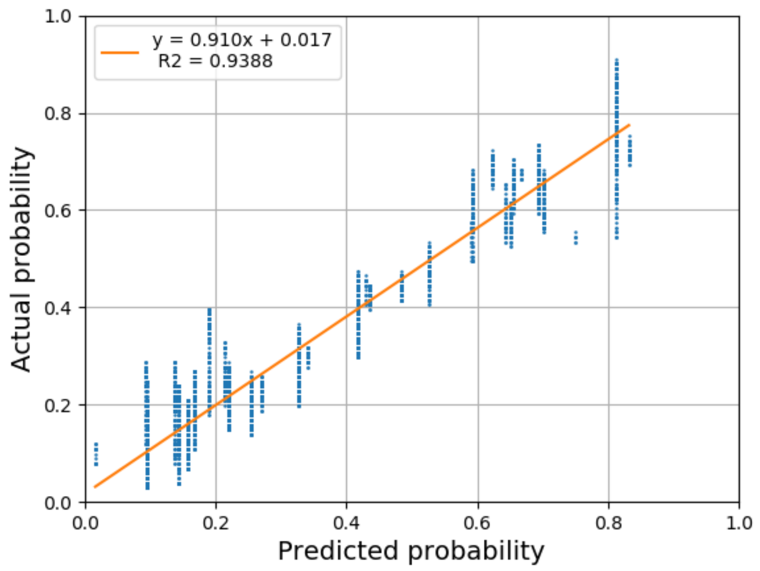

Figure 16: *Estimated probability vs predicted probability of the results from* ***decision tree****.*

These plots show that a single decision tree definitely performs better than logistic regression, but doesn't quite reach up to a neural network (although it's not very far away).

Yeh and Lien[1] also performed analysis on a classification tree, and comparison in Table 5 shows that we were able to reproduce the area ratio of the gains chart, but also that our model *massively* outperform theirs in terms of R2-score of predicted probability. As their description of the methods are quite sparse, we don't know what's eventually different in the models, but we're guessing it's due to less-than-optimal tuning of hyperparameters.

Authors	Error rate	Area ratio	R2-score	Reg. Coefficient	Reg. Intercept
Yeh, Lien(2009)[1]	0.17	0.536	0.278	1.111	-0.276
Myrvik, Tuv(2018)	0.177	0.52	0.939	0.910	0.017

Table 5: *Comparison of the results presented in [1] and our own results for* ***classification tree****. Error rate is the fraction of misclassifications by the model. All values are extracted from the performance on the validation/test set.*

We ran the same hyperparameter analysis on the random forest, and the resulting plots from this can also be seen in Appendix A.1. In this case, we see that a depth of 6-8 seems to be preferred, while minimum number of samples doesn't really alter the results much further. We do set the number of samples to 100 however, which seems to increase the performance a tiny percentage. Running the forest on the final test set yields the results shown in Figure 17 and 18.

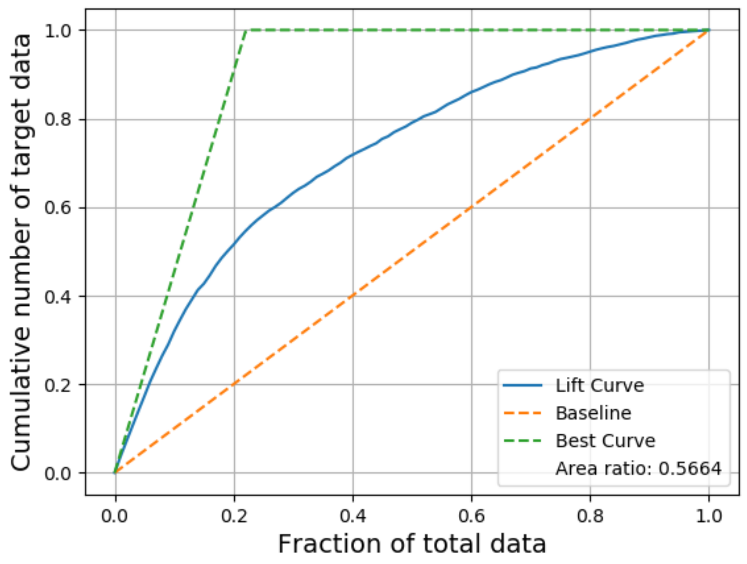

Figure 17: *Cumulative gains chart of the results from* ***random forest****.*

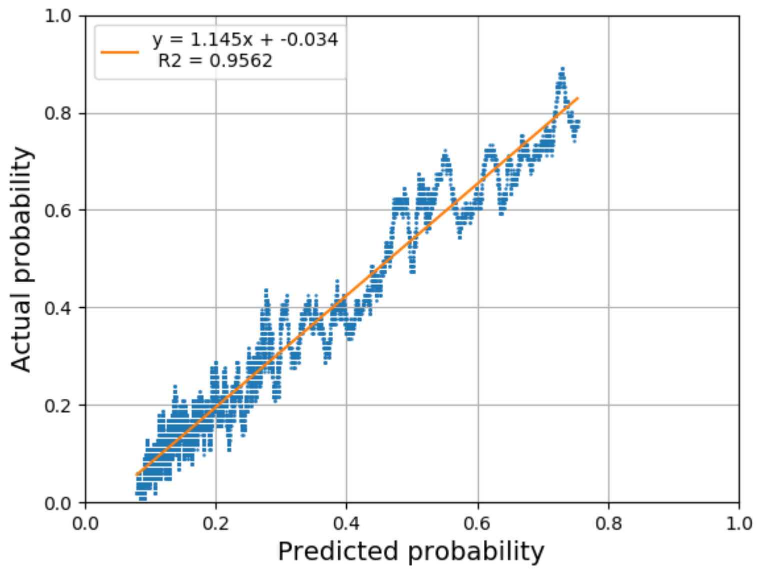

Figure 18: *Estimated probability vs predicted probability of the results from* ***random forest****.*

The random forests is shown to outperform the single decision tree (which is expected due to the increases complexity and flexibility), and also competes with the neural network as the top performer. The results do indeed fluctuate a little from run to run, but the general trend is that these to seems to perform at approximately the same level.

8.3 Support Vector Machine

The support vector machines algorithm was run with sci-kit learn for rbf, linear and polynomial kernels. In table 6 we see the ten best results when setting maximum iterations of the svm to 5000. All parameter combinations are mentioned in the table caption. As the C-parameter works as regularization, no additional regularization has been tested. One thing that is clear from table 6 is that the rbf kernel completely outperforms the others. We can be pretty sure that the linear svm is not a good choice for this data set as we can expect a lot of correlations of the features, however there might be some higher order polynomial kernel which would outperform the rbf. Given the results from all the other learning methods this does not seem very likely though. Another indicator that we are not likely to find any perfect model is the fact that there is not any clear tendencies for the gamma and C parameters and the combinations in these best models seem pretty random.

Gamma	C	Kernel	Area Test	Area Train	R_2 Test	R_2 Train
0.01	0.001	rbf	0.4366	0.4533	0.8285	0.8263
0.01	0.0001	rbf	0.4350	0.4498	0.9189	0.8943
0.01	0.1	rbf	0.4326	0.4608	0.8370	0.8225
0.001	10.0	rbf	0.4270	0.4621	0.8579	0.8508
0.001	1.0	rbf	0.4261	0.4464	0.8239	0.8181
0.001	0.1	rbf	0.4250	0.4416	0.8223	0.8180
0.1	1.0	rbf	0.4203	0.6583	0.9304	0.7611
0.1	0.1	rbf	0.4192	0.5877	0.8717	0.6367
0.1	0.001	rbf	0.4173	0.5126	0.7841	0.6498

Table 6: *Table of ten best results for max iterations of 5000 sorted by area of test set. Program was run for combinations of Kernels: Poly, RBF, Linear; Degrees (poly only): 1, 2, 3, 4, 5, 6; Gamma: 10, 1, 0.1, 0.01, 0.001; C: 100, 10, 1, 0.1, 0.001, 0.0001.*
It is clear that out of the possible combinations, the rbf kernel is completely dominating the results.

We then reran the five best results for 1 000 000 max iterations which yielded a slight increase in Area Test results. Interestingly the C and gamma combinations for the best model have all been shuffled compared to table 6. Which again substantiate our claim that which parameter combination giving the best results are not a given, and we could probably pick any of these models to be our best svm-model. The cummulative gains chart and probability chart for the best model in table 7 can be seen in figure 19 and 20

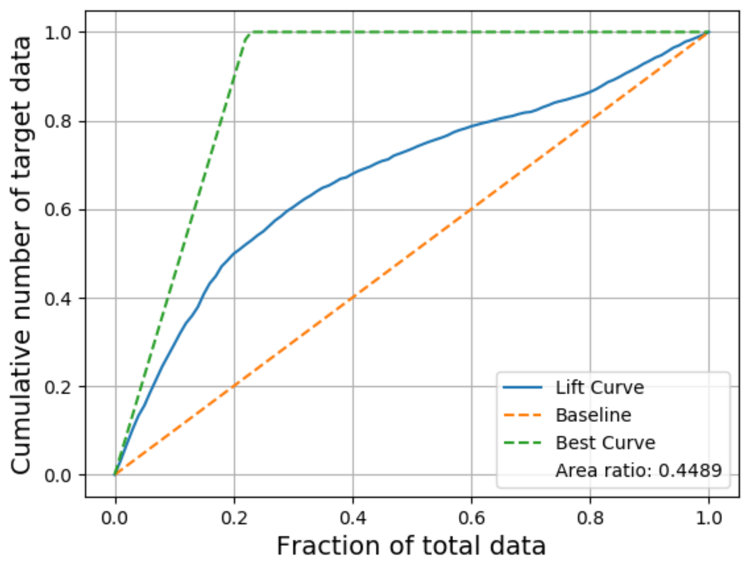

Figure 19: *Cumulative gains chart of the best results for Support vector machines*

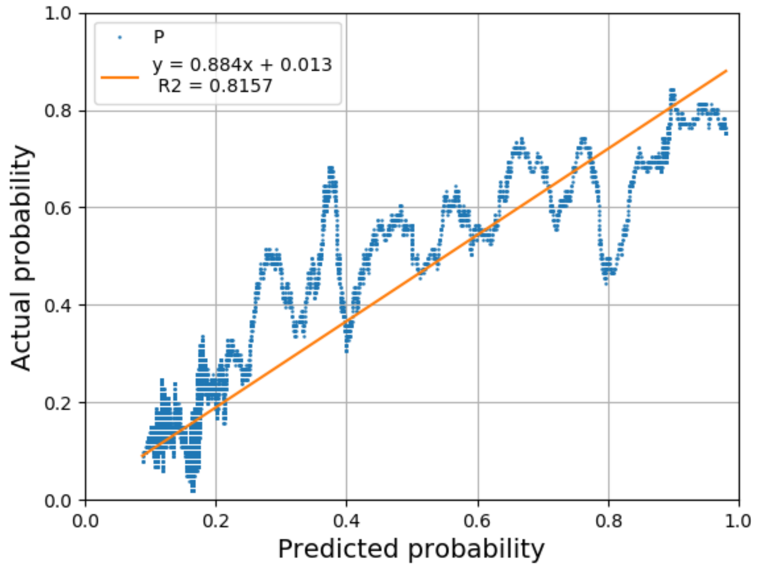

Figure 20: *Estimated probability vs predicted probability of the best results for Support Vector Machines*

Gamma	C	Kernel	Area Test	Area Train	R_2 Test	R_2 Train
0.01	0.1	rbf	0.4489	0.4550	0.8157	0.8093
0.001	10.0	rbf	0.4406	0.4510	0.8506	0.8190
0.01	0.001	rbf	0.4391	0.4442	0.8448	0.8213
0.001	1.0	rbf	0.4364	0.4394	0.8194	0.8092
0.01	0.0001	rbf	0.4311	0.4353	0.9089	0.8952

Table 7: *Rerun of the 5 best combinations in table 6. Now run for 1 000 000 max iterations. This is just the maximum number of iterations and most often the algorithm will terminate before reaching 1 000 000 iterations.*

8.4 Resampling

We reran the best results for data with undersampling, oversampling and SMOTE. All resampling techniques resampled until the ratio between class 1 and class 0 was 1.0.
The neural network was run for 10 000 epochs with the same architecture as the best result in section 8.1. We see in table 8 that the resampling only had a negative effect on the end results.
SVM reran the five best results from section 8.3. The best results for all three resampling techniqeus was with $\gamma = 0.01$ and $C = 0.1$. Unlike for neural networks the resampling had a positive effect, and especially when using the SMOTE algorithm. For classification trees and random forest we see that the resampling had little to no effect, certainly not enough to draw any conclusions on whether it's better or worse with or without these techniques.

One thing to note is that the error rate increases compared to what was found without resampling. This may at first raise a red flag in our heads, but this is actually an expected property. As we now have evened out the imbalance of labels in the data, we have made it so that the models no longer have a bias towards class 0 in terms of error rates. A randomly guessing model should now give about 50% accuracy, meaning that this metric is now actually viable, and the accuracy given might in fact be a realistic estimate of the quality of the model. We see that the models generally give accuracies between $70-80\%$, showing that the results indeed are not impressive. Now of course, the error rates in the models with or without resampling aren't directly comparable, but we still have the area ratio and R2 which are unaffected by the resampling.

Method	Sampling	Error	Area	R2	Reg.Coef	Reg. Int
Log.Reg	Under sampling	0.294	0.432	0.831	1.089	-0.042
Log.Reg	Over sampling	0.33	0.439	0.777	1.068	-0.035
Log.Reg	SMOTE	0.289	0.441	0.8631	0.0997	0.038
NN	Under sampling	0.2754	0.492	0.859	0.942	-0.030
NN	Over sampling	0.2908	0.473	0.857	0.981	-0.005
NN	SMOTE	0.2981	0.46	0.8533	0.976	0.009
Cls.Trees	Under sampling	0.244	0.49	0.87	0.739	-0.065
Cls.Trees	Over sampling	0.262	0.513	0.841	0.736	-0.095
Cls.Trees	SMOTE	0.251	0.486	0.79	0.739	-0.083
RF	Under sampling	0.21	0.547	0.936	0.913	-0.138
RF	Over sampling	0.23	0.56	0.92	0.946	-0.185
RF	SMOTE	0.20	0.543	0.936	0.974	-0.187
SVM	Under sampling	0.1885	0.447	0.856	0.980	-0.001
SVM	Over sampling	0.1917	0.444	0.828	0.999	-0.004
SVM	SMOTE	0.234	0.482	0.862	1.005	-0.004

Table 8: *We reran the best results for all learning methods using under sampling, over sampling and SMOTE on the data set. We see in the table that the impact of resampling generally did not improve the results.*

III Conclusions and future work

9 Summary and conclusion

We were able to reproduce the results of Yeh and Lien when using a Neural Network and Logistic regression, as well as outperforming some of their results on decision trees. Otherwise our results were almost identical, and after a lot of tuning of hyperparameters, we're confident that these results are indeed close to the thresholds of the maximum performance of the methods on this particular data. We also tested two other machine learning methods, namely random forests and support vector machines. While the random forest proved to performed just as well as the former champion, i.e the neural network, the support vector machines didn't seem to bring any new and deeper insight to the table. While it was able to perform better than logistic regression, it didn't quite reach up to the performances of neural networks and random forests. A summary of the best results for each method is shown in Table 9.

Method	Error rate	Area ratio	R2-score	Reg. Coef	Reg. Int	Resampling
Logistic regression	0.191	0.442	0.857	1.119	-0.036	No
Neural network	0.168	0.5539	0.9656	1.006	-0.001	No
Classification tree	0.177	0.52	0.939	0.910	0.017	No
Random Forest	0.165	0.566	0.956	1.145	-0.034	No
SVM	0.234	0.482	0.862	1.005	-0.004	SMOTE

Table 9: Summary of the best result for each method.

We can not rule out that there exists better models which will outperform our attempts, however it is likely that the data set is largely at fault for the bad results.
The fact that the data set is gathered from a period when Taiwanese banks over-issued credit cards to unqualified applicants should serve as a red flag to whether features in the data set is trustworthy or not. Features which obviously is part in determining if one would default or not is salary and total debt of the applicant. Normally these features might me inferred from the credit limits but if the banks was raising the limits to excess, this is no longer the case. If the credit limit has been set with honesty and caution, this is likely to be a very powerful feature but judging from our results it does not have the desired impact in this data set.
The features in the data set obviously has an impact on whether or not the applicant is likely to default as the models produced outperform the useless model mentioned in section 6.1. However we must conclude that the features of the data set is not powerful enough to comfortably predict the default probability and more features should be added for a better result.

10 Thoughts for future

The data set could be improved by feature engineering and manually cleaning the data set. This was done in **this study**, where they also used other machine learning methods like Gradient Boosting Machine and Stacking. According to the authors, these methods seemed to slightly improve the results, and perhaps delving deeper into these types of methods will yield even better results.

For investigating our claim that the features of the data set is not good enough, a solution would be to compare the results of this project with another data set of credit card defaults. This way we can investigate if there actually exist some underlying features of those who default, or if modeling this with high accuracy is a too complex and variable task.

IV Appendix

A.1 Additional plots

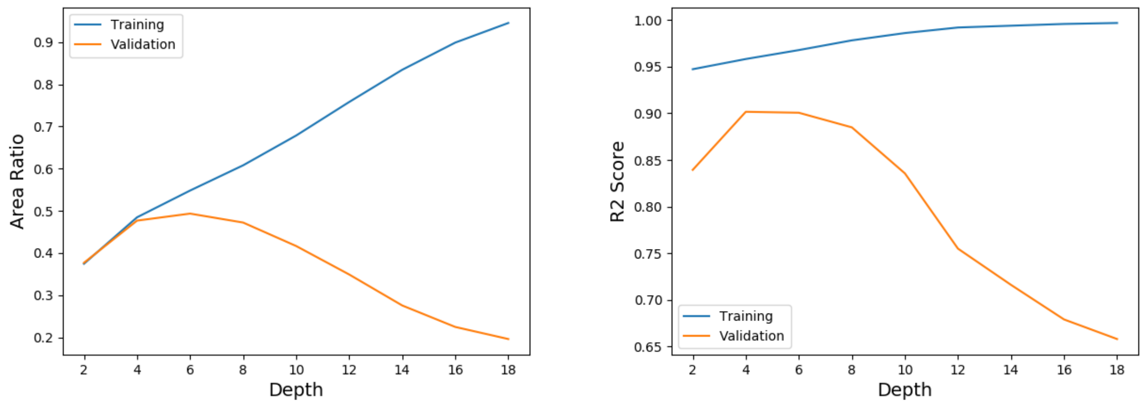

Figure 21: *Investigation of the performance of a* ***decision tree****, as function of maximum depth. The left plot shows the area of the cumulative gains chart, and the right one shows the R2 score of the probability estimations, both for training and validation sets. The plots shows that the model starts to overfit quite fast with increasing depths. The optimal depth seems to be at 4-6.*

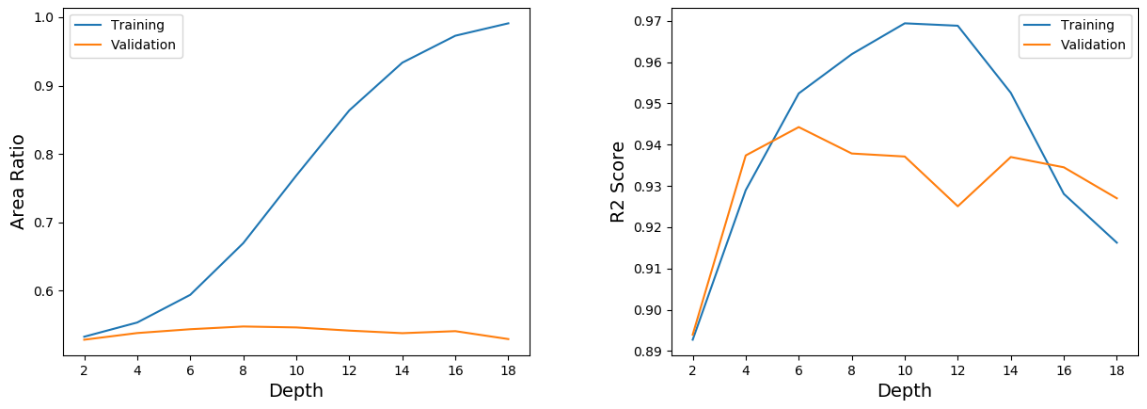

Figure 22: *Investigation of the performance of a* ***random forest****, as function of maximum depth. The left plot shows the area of the cumulative gains chart, and the right one shows the R2 score of the probability estimations, both for training and validation sets. The optimal depth seems to be at 6-8.*

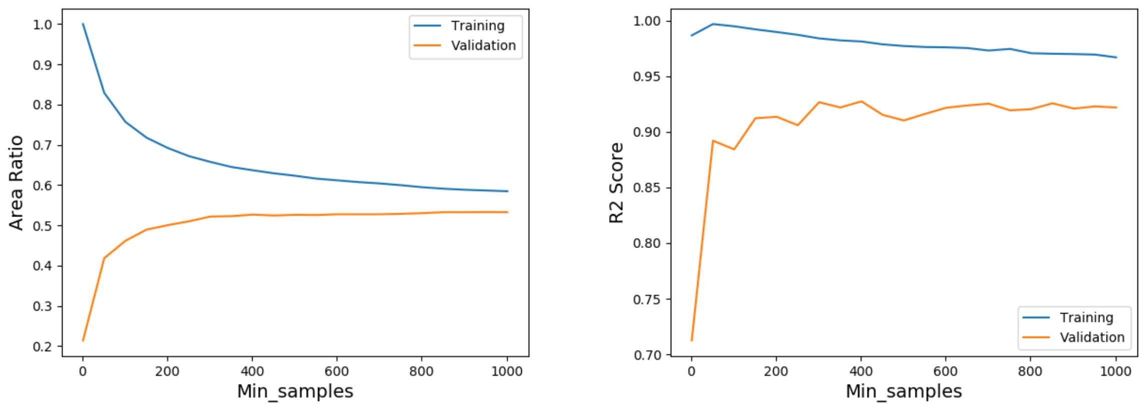

Figure 23: *Investigation of the performance of a* ***decision tree****, as function of minimum number of samples in order to perform a split, at max depth = 6. The left plot shows the area of the cumulative gains chart, and the right one shows the R2 score of the probability estimations, both for training and validation sets. We see that we don't gain much by setting this hyperparameter, but maybe a minimum of 100 is improving the performance a tiny bit.*

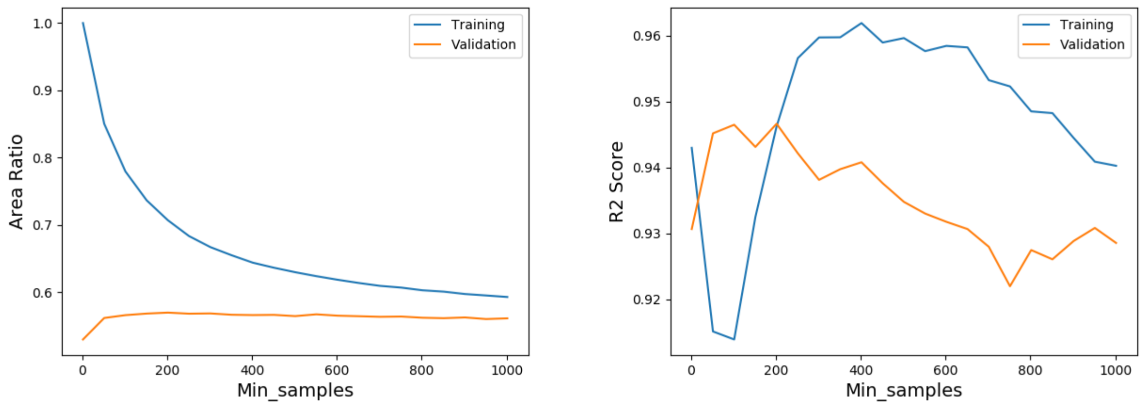

Figure 24: *Investigation of the performance of a* ***random forest****, as function of minimum number of samples in order to perform a split, at max depth = 6. The left plot shows the area of the cumulative gains chart, and the right one shows the R2 score of the probability estimations, both for training and validation sets. We see that we don't gain much by setting this hyperparameter, but maybe a minimum of 100 is improving the performance a tiny bit.*

V References

References

[1] "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients", I-Cheng Yeh, Che-hui Lien, 2009 https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi$_0$5$_c$*lassification*$_d$*atabases*/2.1 − *lesson/assets/datasets/DefaultCreditCardClients*$_y$*eh*$_2$009*.pdf*

[2] "Decision Trees in Machine Learning", Prashant Gupta, Towards Data Science, May 2017. https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052

[3] Lagrange multipliers, Harvard, 2009, Math 21A

[4] Karush-Kuhn-Tucker , Wikipedia, 11. Dec 2018

[5] Convex Optimization, Wikipedia, 11.Dec 2018.

[6] Wolfe Duality, Wikipedia, 11.Dec 2018.

[7] The Elements of Statistical Learning; Trevor Hastie, Robert Tibshirani, Jerome Friedman; Spring; Second Edition; 2009; ch. 12.2; p.418; Figure 12.1

[8] The Elements of Statistical Learning; Trevor Hastie, Robert Tibshirani, Jerome Friedman; Spring; Second Edition; 2009; ch. 15.3; p.592

[9] The Elements of Statistical Learning; Trevor Hastie, Robert Tibshirani, Jerome Friedman; Spring; Second Edition; 2009; p.130

[10] Hands-On Machine Learning with Scikit-Learn TensorFlow, Aurélien Géron, O'Reilly, 2017, ch. 5.9, p.165.

[11] SMOTE: Synthetic Minority Over-sampling Technique, Kevin W. Bowyer and Nitesh V. Chawla and Lawrence O. Hall and W. Philip Kegelmeyer, 2011

[12] Comparative study ID3, CART and C4.5 decision tree algorithm: A survey, S.Singh, P. Gupta, University of Delhi, July 2014.