Classification methods

Project 2 on machine learning

Bjørn Inge Vik
Jolynde Vis

November 8, 2019

Abstract

We apply machine learning classification methods to predict probabilities of default among credit card users in Taiwan. This is based on data containing information about credit status, repayment history and certain private data variables like age and marital status. Two methods are tested: logistic regression and feed forward neural networks. At the expense of computational time, neural networks proved the better models, giving slightly higher accuracy and fewer false negatives (wrongly predicted defaults) than the logistic regression models. The best models achieved an accuracy on the test set of roughly 82.5%.

Separately, we apply neural network regressors to estimate Franke's function. Compared with linear regression methods investigated in a separate report, neural networks again perform slightly better. When fitting a $7th$ order polynomial, the best models achieve a cross validated mean squared error of 0.010, compared with 0.011 with OLS regression.

Contents

1 Introduction

In this report, we apply machine learning classification methods to predict probabilities of default among credit card users in Taiwan. Separately, we also investigate the suitability of neural network regressors to estimate Franke's function.

Credit card data
Credit card debt in Taiwan soared after the late 90's. By the end of 2005, as many as 700,000 customers were not able to repay their loans and the average debt size was roughly one million NTD [1]. The consequent socio-economic impact may have contributed to rising suicide rates. To expand their business, banks had in the years prior been aggressively targeting young people and lowering requirements for credit card approvals.

To evaluate risk of defaults in the customer base, one could look at customer transaction and repayment records and attempt to estimate probabilities of default. Yeh and Lien [2] looked at customer data gathered from banks in Taiwan in the time period April-September 2005 and tested several models to predict probabilities of defaults.

Using the same data set, we apply a logistic regression model and a neural network classifier with the aim of replicating or possibly improving the results. Whereas regression deals with continuous variables, classification in machine learning addresses the problem of predicting categorical variables. In the case of the credit card data, this enables us to identify customers at risk of defaulting.

Franke's function We apply a similar neural network, modified to output one continuous variable instead of class probabilities, to estimate Franke's function. This is compared with the results obtained with OLS, Ridge and LASSO regressions in Project 1 by one of us (Vis). Neural networks are much more complex models than general linear regression methods. They may be able to better fit data that is not well explained by a linear model, provided steps are taken to prevent overfitting.

This report is structured as follows. Section two describes the theory used for the different algorithms and analysis. Section three and four include some details on the code implementation and data preprocessing. The results of the analyses are presented in section five. In section 6 we end the report with a conclusion and a critical discussion of the models.

2 Theory

This section explains the theory and algorithms used in our analysis. Seeing that our classification data set has binary outcomes, the theories are discussed with this limitation.

2.1 Logistic regression

Logistic regression uses a logistic function to fit a binary outcome variable. It gives the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$, as given by

$$p(y_i = 1|x_i, \hat{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}),$$

where $\boldsymbol{x}$ contains the input features and $\boldsymbol{\beta}$ are the parameters of the model. The logistic model is fitted by using the maximum log-likelihood estimation, where the log turns the exponentials in the MLE (maximum likelihood estimation) into summations. The maximum log-likelihood is then given by

$$C(\beta) = \sum_{i=1}^{N} \{(y_i \beta^T x_i - log(1 + e^{\beta^T x_i})\}$$

Maximizing the logarithm of a function is the same as maximising the function itself. Therefore, by taking the negative of the maximum log-likelihood, we can define the cost/loss function for logistic regression, also known as the cross entropy:

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} (y_i(\beta_0 + \beta_1 x_i) - \log(1 + \exp(\beta_0 + \beta_1 x_i)) \tag{1}$$

The optimal parameters of the model are found by minimizing the cost function. To do so, we use gradient descent algorithm described below.

2.2 Gradient descent

To minimise the cost function as described above, we use two versions of gradient descent: standard and mini-batch. The gradient descent method is commonly used in machine learning algorithms to determine the parameters w that minimize the cost function F. It involves taking consecutive steps k in the parameter space in the opposite direction of the gradient of the cost function:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla F(\mathbf{w}_k),$$

When calculating the cost on the whole data set, this algorithm is referred to as (standard) gradient descent. The drawback is the computational cost of calculating the loss if the data

set is big. Convergence is reached faster if the loss is calculated on a subset on the data and a step is taken in a parameter space for each subset. The size of the subset is referred to as the batch size and a typical value may be 32 data points. This version of gradient descent is called mini-batch gradient descent, and it can also help prevent getting stuck in a local minimum as the gradient is evaluated on different samples each time. Local minima is not an issue for logistic regression, where the cost function is convex. In that case, any minimum is global. A function f is convex if

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \tag{2}$$

For $x_1, x_2 \in X$ and for $t \in [0, 1]$, given that $X \subset R^n$ is a convex set and $f : X \to R$ is continuous.

2.3 Neural network

A neural network consists of several layers with a different number of connected neurons, or nodes. The first layer is called the input layer and has the same number of nodes as there are features in the data set. The last layer is called the output layer and has one node per output value, in a binary case two (here: 0 or 1). The layers in between are called hidden layers, which contain an arbitrary number of nodes. In a feed forward network, all nodes in a layer are connected to all nodes in the two neighboring layers.

The input to the first layer is simply the features in the data set. The input to each node is then transformed to an output using a so called activation function. This is in turn fed into the nodes in the next layer, with the input to each node being a linear combination of the outputs of all the nodes in the first layer plus a bias number. The coefficients in the linear combination are typically called weights. This process continues and the prediction of the network is the output of the last layer.

The network in this report will be trained using the mini-batch gradient descent algorithm described above. This is done via a number of so called feed forward and backpropagation cycles. Feed forward refers to the process of letting the network take input and calculate predictions given the current weights. The cost is then calculated using a cost function. Then, gradients of the cost function with respect to the weights are calculated for the different layers, starting from the output layer and propagating back to the first layer of the network. For this process to function properly, the weights has to be initialized to small, non zero values.

The start of the feed forward process involves letting the nodes in the input layer correspond to the values of the features in the data set. The input to a node j in subsequent

layer l is then given by

$$z_j^l = \sum_{i=1}^{H} w_{ij}^l a_i^{l-1} + b_j^l. \tag{3}$$

and the output from the same node, which in turn feeds into nodes in the subsequent layers, is given by

$$a_j^l = f(a_j^l) \tag{4}$$

with f denoting the activation function. The output a_j^L from the last layer L is the prediction of the model.

During backpropagation, the weights are updated according to equations 5-9.

$$z_j^L = \sum_{i=1}^{H} w_{ij}^L a_i^l + b_j^L, \tag{5}$$

$$\delta_j^L = f'(z_j^L) \frac{\partial \mathcal{C}}{\partial (a_j^L)}, \tag{6}$$

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l), \tag{7}$$

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1}, \tag{8}$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l, \tag{9}$$

where C denotes the cost and b denotes the bias number. Neural networks can be used for both classification as well as regression.

2.3.1 Classification

The standard cost function applied to neural network classifiers is the categorical cross entropy, or the binary cross entropy in case of only two outcomes:

$$C_{ce} = (ylog(p) + (1y)log(1p)) \tag{10}$$

Neural networks are prone to overfitting, therefore we will also test models with L2 regularization. That involves adding the squared sum of the weights times a regularization factor λ to the cost function. Early stopping and dropout are other ways of combating overfitting, however these are not further discussed in this report.

We have tested networks with two different activation functions applied to the hidden layers:

$$ReLU(z) = max(0, z) \tag{11}$$

$$Sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{12}$$

On the output layer we use the softmax function, which is standard for classification problems. It is defined as

$$softmax(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}} \tag{13}$$

with i summing over the number of output classes K. Softmax activation on the last layer ensures that the output for each category sums to one and can be interpreted as probabilities.

2.3.2 Regression

When using a neural network for regression, the output layer shouldn't output probabilities for categories, but single values, meaning there is only one node in the output layer. We use mean squared error as the cost function, which is common for regression networks. L2 regularization is tested for regression as well. ReLU and sigmoid activation have been used on the hidden layers. As the output does not have to be interpreted as probabilities and should not be confined within a certain range, we do not apply an activation function on the output layer. The model complexity resulting from high numbers of interconnected nodes means that a neural network typically is able to fit more complex data than linear regression methods like OLS, Ridge and Lasso. This is especially interesting if a regression equation doesn't fit the data well.

2.4 Data preprocessing

In order to create a sensible machine learning model, we have to ensure that the data used as input represents the population. In order to do so we remove outliers, i.e. values that likely come from incorrectly entered or measured data. We also experiment with feature selection in order to remove redundant or irrelevant features and speed up training of the models.

2.4.1 Data cleaning

In scatterplots we can identify outliers and where they are located (either very high or very low values). To remove the outliers, we look at the inter quartile range (IQR). This is the difference between the 75th and 25th percentiles of the data. The 'rule of thumb' [3] is that a value is an outlier if:

- The value is larger than $Q3 + 1.5 * IQR$
- The value is less than $Q1 - 1.5 * IQR$

2.4.2 Feature selection

Using principal component analysis (PCA) we can reduce the dimensionality of our dataset. Reducing the dimensionality saves computational time. The algorithm works by creating orthogonal combinations of the original features. The dominant new feature (component) is the feature that has the largest variance. The other components are sorted from highest to lowest variance and included until one has covered a set percentage of the original variance in the data.

2.5 Model evaluation

The models are evaluated based on four scores; 1) the accuracy score, 2) the confusion matrix, and accordingly the number of false negatives, 3) the precision score, and 4) the area ratio.

2.5.1 Accuracy score

The accuracy score is the number of correct predictions divided by the number of total predictions. It indicates how well the model classifies. The accuracy score has some limitations when classes are unbalanced. Take the the binary case where class 0 makes up 90% of the data. With an accuracy of 90%, it is possible that either the model only predicts 0's or that the accuracy on the true 0's are 99% while the accuracy on true 1's are roughly 10%. Thus there is a need for multiple metrics to assess the models.

2.5.2 Confusion matrix

While the accuracy score assesses the model on share of corret predictions, we also assess the models on where their wrong predictions are located. Perhaps some types of errors are more costly. For the data set used here, on could imagine that the false negatives (FN) are a problem for the bank. False negatives means that the model falsely predicts a zero, i.e. that the customer will not default. This error could be important to minimize, depending on if any measures can be taken, as a default is costly for the bank. If the model falsely predicts a default, the model is just overly protective. A confusion matrix gives the distribution of the predictions. For the binary case it is defined as shown in figure 1. The false negatives are located in the lower left corner.

2.5.3 Precision score

The precision score is defined as the number of correctly predicted positives divided by the number of total predicted positives:

$$Precision(1) = \frac{TP}{(TP + FP)}, Precision(0) = \frac{TN}{(TN + FN)} \tag{14}$$

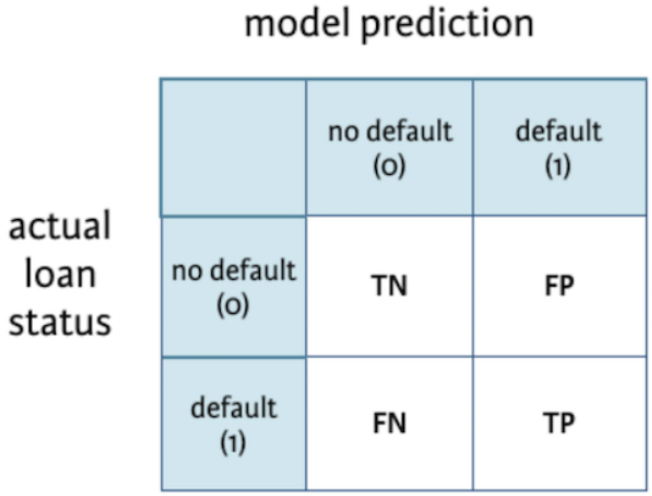


Figure 1: Confusion matrix layout

where TP stands for true negatives, TN for true negative and FP for false negative. Under the assumption that that in our case a FN is a bigger problem (it all depends on what the banks or regulators could use the predictions for), we rate the models on the precision score for defaults (0's).

2.5.4 Area ratio

As last measure we use the area ratio. The area ratio is based on the cumulative gains curve of the model. Cumulative gains curve is in our case the number of defaults as a function of number of predictions sorted from highest to lowest according to probability of default. For example, the cumulative gain after 10 predictions is the number of defaults among the 10 cases with the highest predicted probability of default. If the model is good, this number should be very close to 10, but the ratio between predictions and defaults will of course drop when cases with less likelihood of default are being included.The cumulative gains chart is shown in figure 2 and contains three curves; the diagonal baseline curve, the theoretically best curve and the model curve. The baseline represents a model with no predictive power and the theoretically best curve represents a perfect model that would always predict correct. So, the more the model curve approaches the theoretically best curve, the better the model. As a measure of this, we use the area ratio as specified by Yeh and Lien [2], which is defined as:

$$\text{area ratio} = \frac{\text{area between model curve and baseline curve}}{\text{area between theoretically best curve and baseline curve}} \tag{15}$$

An area ratio of 1.0 would describe a perfect model, i.e. the higher the area ratio, the

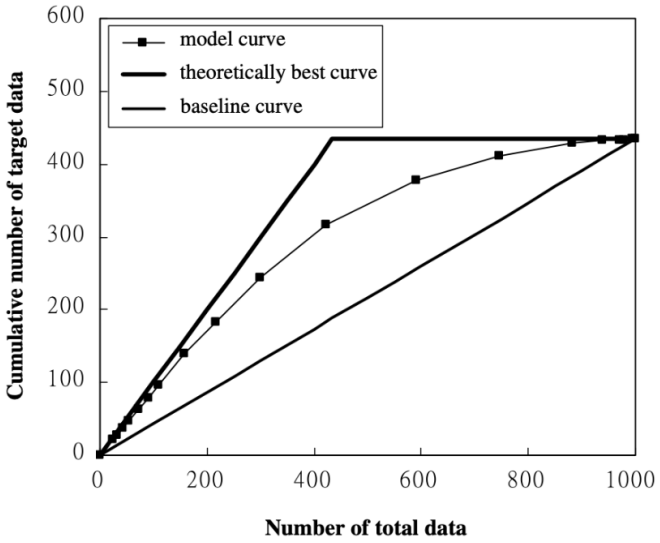


Figure 2: Cumulative gains chart

better the model. Area ratio is a good measure of a model's predictive power, especially when the classes are unbalanced. It is the preferred metric by Yeh and Lien [2].

2.6 Regression metrics

The regression models are evaluated on their MSE- and R2-score, same as for project 1, which are defined as

$$\text{MSE}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{16}$$

$$R^2(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = 1 - \frac{\sum_i (y_i - \tilde{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \quad \text{,where} \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{17}$$

3 Code implementation

As well as running models based on libraries from Scitkit-Learn and Keras/Tensorflow, we have constructed classes for logistic regression and neural networks.

The logistic regression method allows for free choice of the learning rate η, regularization λ and number of iterations to minimize the cost function. To fit the model we use normal (full sample) gradient descent, as convergence time is not a problem. The logistic regression model class is shown in its entirety in appendix 1.1.

The neural network implementations allow the user to specify the number of neurons in each layer, the L2 regularization λ, as well as the number of epochs and batch size for training, using mini-batch gradient descent. It supports ReLU and sigmoid activation functions for the hidden layers. For the neural network classifier model, softmax activation is used on the output, while the neural network regression model has no activation on the output. Our numerically stable implementation of the softmax function is shown in appendix 1.2. Our implementation of the backpropagation algorithm is shown in appendix 1.3.

We initialize the neural network weights in each layer according to a random normal distribution with sigma, with $\sigma = 1/\sqrt{(2/n)}$, where n is the number of neurons in the previous layer. This is called Kaiming initialization and proved to work better when using the ReLU activation function.

The full code can be found on Github:
https://github.com/bingovik/FYS-STK4155/tree/master/Project2

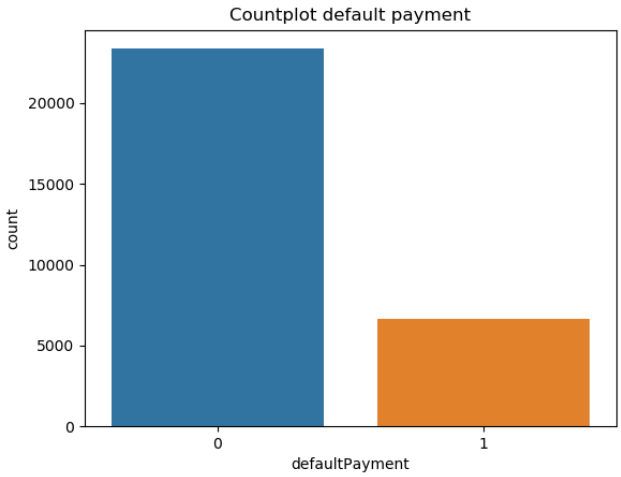


Figure 3: Countplot of the target variable 'default payment'

4 Data preprocessing

For this report we used the *'default of credit card clients Data Set'* from UCI, which contains data of credit card holders of an important bank in Taiwan. The outcome variable is default payment, where 0 = no and 1 = yes. The database contains 30.000 entries with 23 independent variables, which are defined in table 1.
The outcome variable contains 6636 entries that defaulted (= 1) and 23364 entries that did not default (= 0), see figure 3. This means that the dataset is biased, and the minority class makes up 22.1% of the data. Although not extreme, this is a challenge for classification, seeing that a model that only predicts 0's would be correct 77.9% of the time.

4.1 Data and preprocessing

We look at the variables, and see that we are dealing with both categorical and numerical variables.

Categorical variables
We start by looking at the categorical variables in the dataset. There appear to be a few undefined values. For the variable 'marital status', there are 49 entries with a value of 0, which is not defined. We delete these rows. The variable 'education' has the undefined values 0, 5 and 6 in the data set. Most of the values are in the defined four classes, and we delete the undefined entries for education as well. We elaborate on this and included some plots in appendix 2.

	Variables	
X1	Amount of given credit	(numerical) Dollars
X2	Gender	(categorical) 1 = male 2 = female
X3	Education	(categorical) 1 = graduate school 2 = university 3 = high school 4 = others
X4	Marital status	(categorical) 1 = married 2 = single 3 = others
X5	Age	(numerical) Years
X6-X11	History of past payment	(categorical, ordinal) -1 = pay duly 1 = payment delay for 1 month 2 = payment delay for 2 months etc. till 9 ¿
X12-X17	Amount of bill statement	(numerical) Dollars
X18-X23	Amount of previous payment	(numerical) Dollars

Table 1: Variables of the credit card dataset

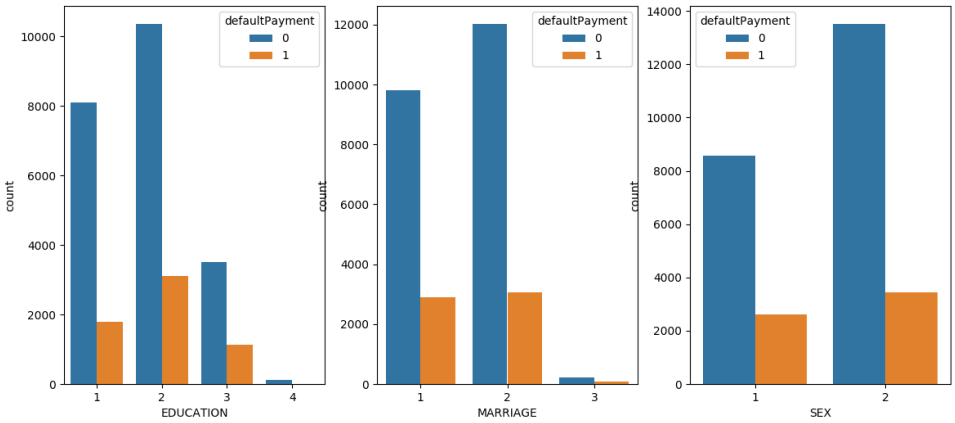


Figure 4: Countplots of 'education', 'marriage' and 'sex', seperated by whether the customer defaulted or not

The 'history of past payment' variable has some entries containing -2, 0. However, for this variable the majority of the entries are in 0, which means we cannot delete these entries without losing a significant amount of data. We will keep these values and make the assumption that 0 means that there was no bill to pay that month, i.e. 'nothing to pay', and that the -2 means that the bill was paid early, i.e. 'paid before'. We have plotted histograms of the variables after deleting the selected entries, see figure 4 and 5.

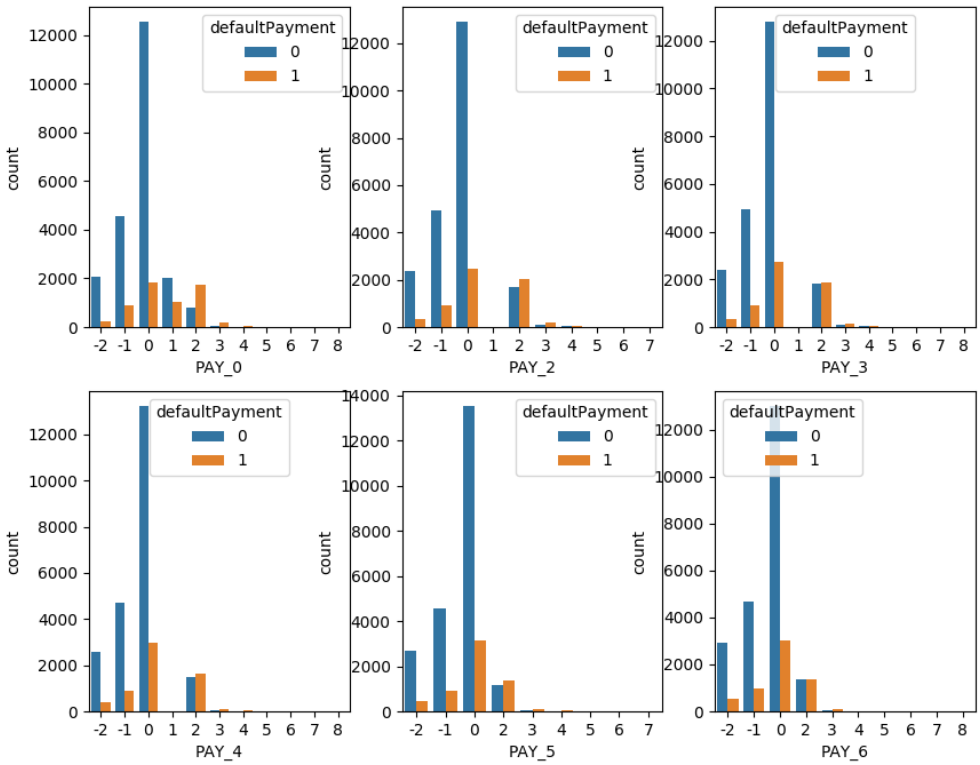


Figure 5: Countplots of 'history of past payment 1-6', separated by whether the customer defaulted or not

We see in figure 4 that for the variables education, marriage and sex the ratio between defaulting or not seems to be equal for the different categories. For the 'history of past payment' variable, in figure 5, it seems that for a higher payment delay (i.e. a higher category), the chances of defaulting on payment are higher, which intuitively makes sense.

Numerical variables

For the numerical variables we looked at the distribution of the variable and whether there are outliers. We start with taking out the entries that have only zero's for 'amount of bill statement' and 'amount of previous payment', seeing that if all these values are zero, they are customers of the bank but they don't have any payment history.

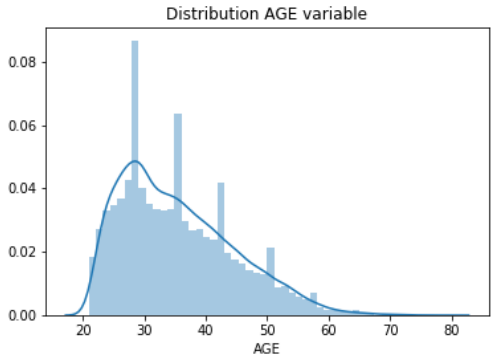


Figure 6: Distribution of the age variable

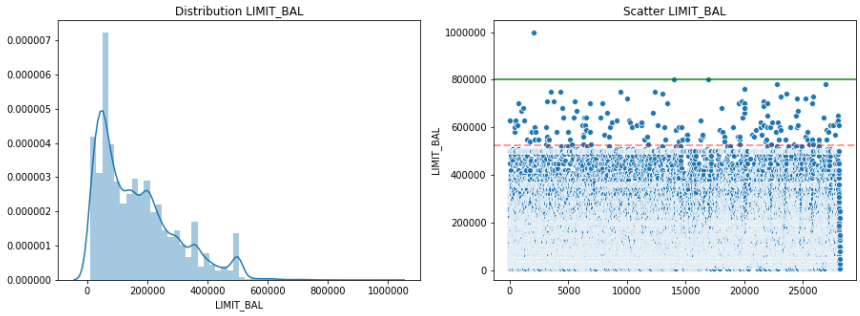


Figure 7: Distribution (left) and scatterplot (right) of the LIMIT _BAL variable

The age variable has values between 21 and 79, with a distribution as shown in figure 6. These values seem very reasonable, i.e. no outliers, and we keep all.
For the LIMIT_BAL variable, we observe in the distribution of the variable (figure 7, left) that it has a tail to the right. In the scatterplot we see that there are some positive outliers and that there are no negative values. In figure 7 (right) the dotted red line shows the 'rule of thumb' outlier removal, with Q3 + 1,5*IQR. Applying this rule would in this case delete too much of the data. We decide to use a higher value to keep the large values included, but delete the one we suspect are outliers. We delete the values above 800.000 (see the green line in the scatterplot).

Figure 8 shows the distribution of the BILL_AMT_variable. In the scatterplots (figure 9) we see that there are outliers in the data; there are some values that are lower than zero and a few very high values. Values below zero would mean a negative bill amount, which may or may not make sense. We observe again that the 'rule of thumb' (red line) would

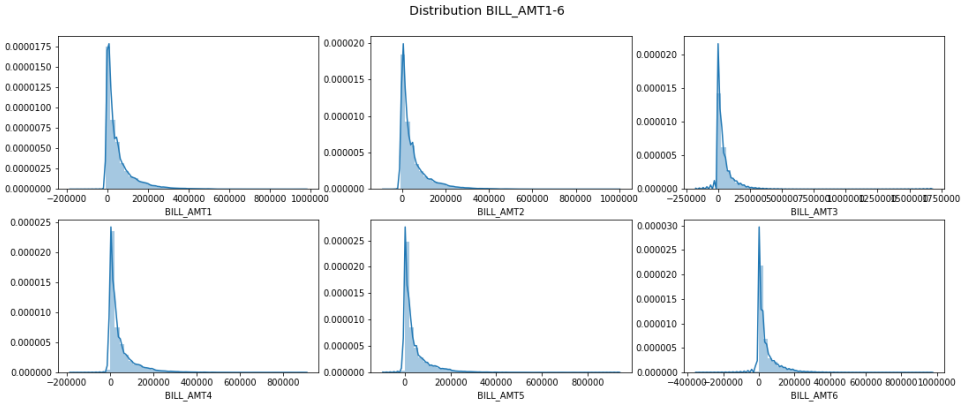


Figure 8: Distribution of the BILL_AMT_variables

delete too much data. Moreover, because it is based on the quartiles per variable, it is different for each variable. However, the variables are related to each other: they describe the same thing, it is just a different month for each variable. For this reason we have decided to treat them the same, and we remove the values that are below zero and the values that are higher than 700,000 (green line).

For the PAY_AMT_variable we see in figure 10 that there are some outliers that create a very long tail. In the scatterplots (figure 11) we see that there are some very high values, especially for PAY_AMT2 (of almost 1,750,000). There are no negative outliers here. We also observe again that the 'rule of thumb' is not working here (red line), because the low values have such a high density. Also, same as for the BILL_AMT variable, we want to treat these six variables equally. We remove the values that are above 400,000 (green line).

Appendix 2 shows additional plots of the variables after the outlier removal.

4.2 Feature selection

The correlation matrix in figure 12 gives an initial insight into which features are correlated with each other and the outcome variable the most. We see that defaultPayment has no strong correlations with any of the variables, which indicates that there is not one or a few variables that strongly determine the outcome, but rather (hopefully) all the variables combined. We also see that the six history of past payment (PAY_) variables are strongly related to each other, and this is the same for the six amount of bill statement (BILL_AMT_) variables. Remarkable is that the six amount of previous payment (PAY_AMT_) variables are not strongly correlated to each other. Besides these there are no strong correlations between the different variables.

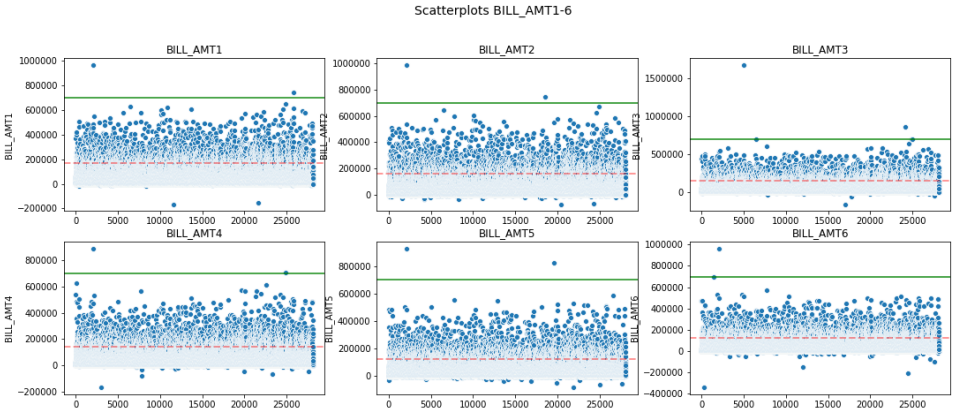


Figure 9: Scatterplots of the BILL_AMT_variables

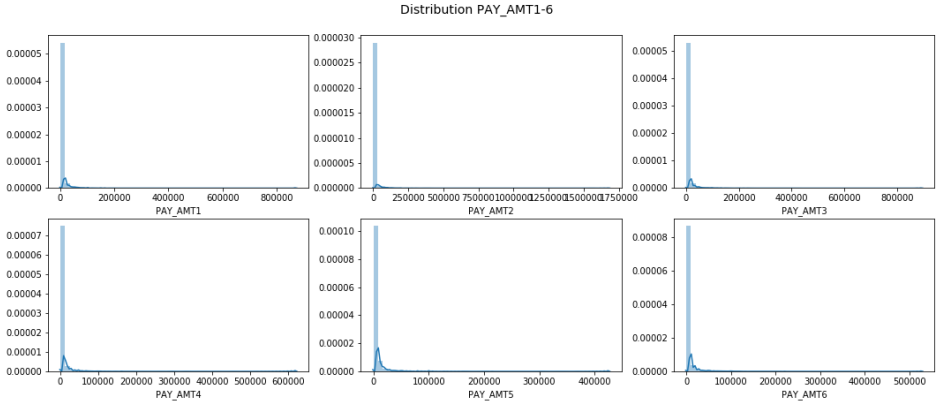


Figure 10: Distribution of the PAY_AMT_variables

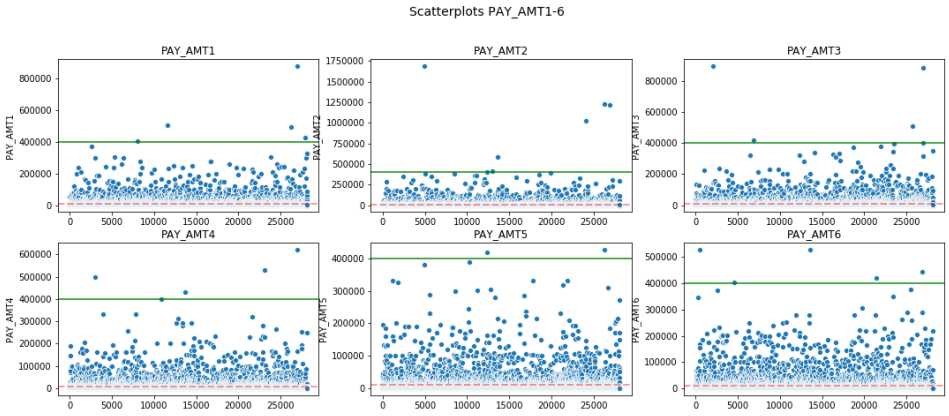


Figure 11: Scatterplots of the PAY_AMT_variables

We now apply the Principal Component Analysis to the design matrix. Figure 13 shows the loss of variance versus the number of principal components. When specifying that we want to retain 97% variance, we get a new design matrix with 19 principal components. We will compare the model scores on the reduced dimensionality data set with the model scores on the data set containing all components.

5 Analysis

To optimize all models we used a grid search with 5-fold cross validation. This was implemented using Scikit-learn's method GridSearchCV.

5.1 Classification of credit card customers

We applied our logistic regression model and neural network model to predict probabilities of default in the credit card data. We use a Scikit-Learn's LogisticRegression model and a Keras/Tensorflow's neural network to compare the results and validate our models. All models were cross validated using the training data only (consisting of 80% of the entire data set shuffled), and thereafter refitted to the entire training set with metrics calculated for the separate test set (20% of the data).

5.1.1 Logistic regression

To find the optimal parameters for the logistic regression classifier we use a grid search. The search is ran over the parameter grid as given in table 2.

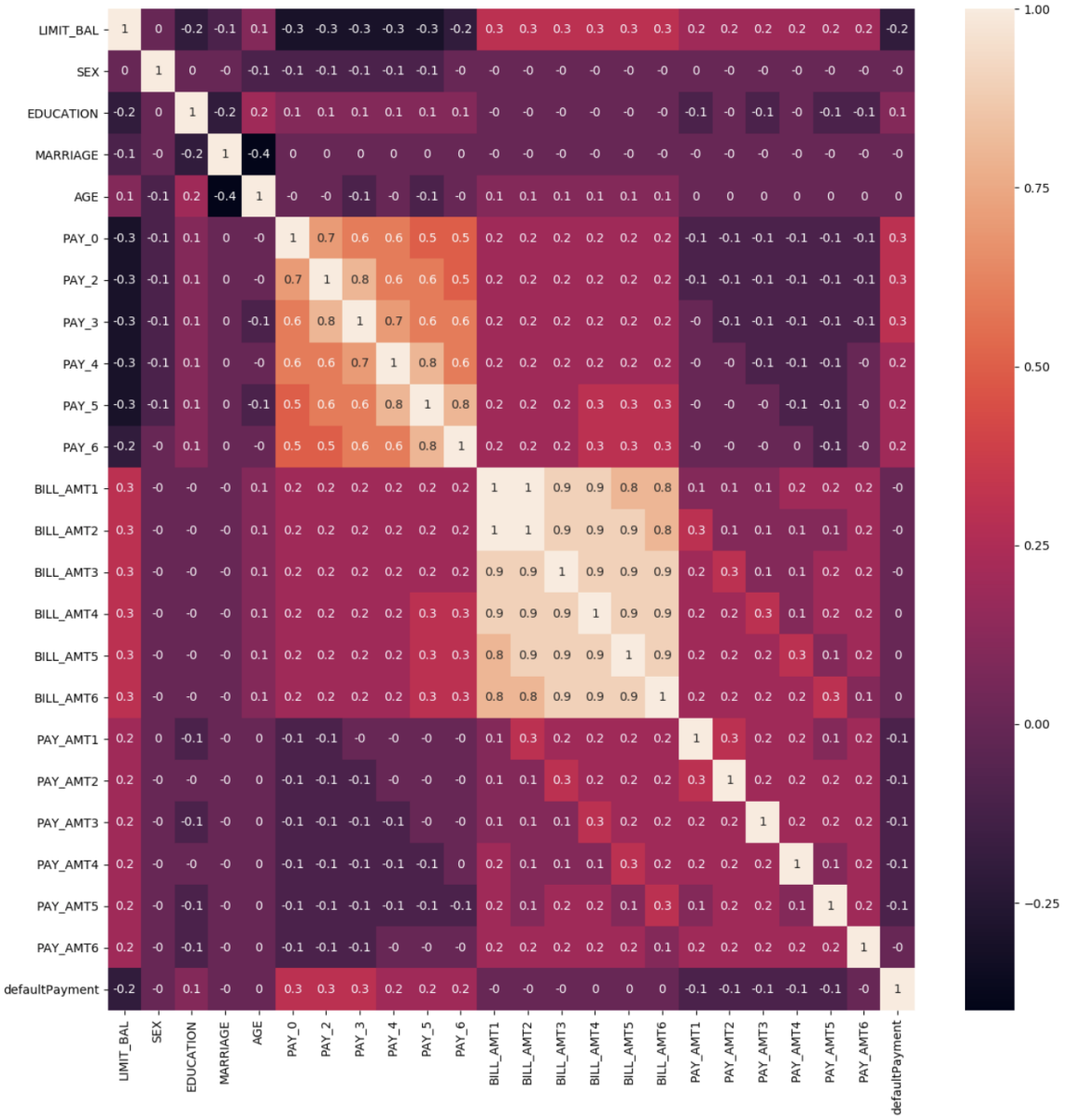


Figure 12: Correlation matrix for the credit card data. Variables BILL AMT and PAY are strongly correlated as expected. No feature is strongly correlated with the target variable defaultPayment

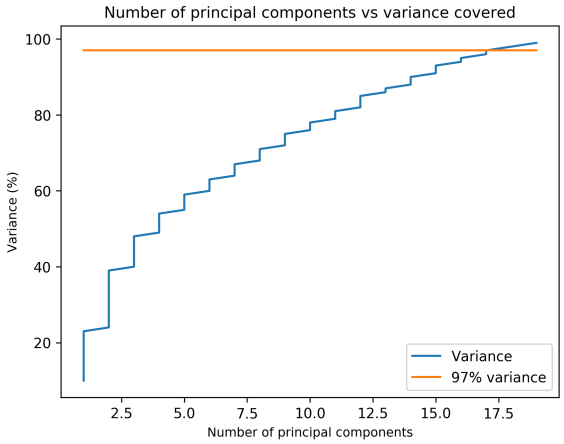


Figure 13: Principal component analysis of the credit card data.

From the grid search results in figure 14 we see that an η (learning rate) of 1 gives the best accuracy scores, or an η of 0.1 with a high number of iterations. A very low or very high value for η makes the model perform notably worse, as it then does either not have enough time to converge or it jumps around to much near the minimum, if coming close at all.The value for λ (regularization) does not have a large effect, only when λ goes up to a 100 the model seems to start performing worse. Overfitting is not a problem as there are "only" 26 features and logistic regression is a relatively simple classifier. This is also seen from the learning chart of the best model refitted to the entire training data (figure 15).

The Scikit-Learn logistic regression model does not use the gradient descent and no learning rate is specified. It is therefore optimized with a different grid search, which can be found in appendix 3.1. The Scikit-Learn model works with an inverse of regularization strength C ($1/\lambda$), where a smaller number specifies a stronger regularization. For the Scikit-Learn model, a higher value for λ gives a better result ($\lambda = 10$ or $\lambda = 100$). This means that the Scikit-Learn model also gives better results for a smaller regularization. The analysis is in appendix 3.2. Figure 16 shows the cumulative gains charts for both logistic regression models. The charts look very similar, with the model line is pretty far lifted from the baseline. We will come back to the cumulative gain charts and evaluate the models later on, together with the neural network classifiers.

Parameter	Description	Values
'lambda'	L2 regularization	0 0.1 1.0 10
'eta'	Learning rate	0.01 0.1 1.0
'max_iter'	Number of iterations	100 500 1000

Table 2: Parameter grid for GridSearchCV for logistic regression

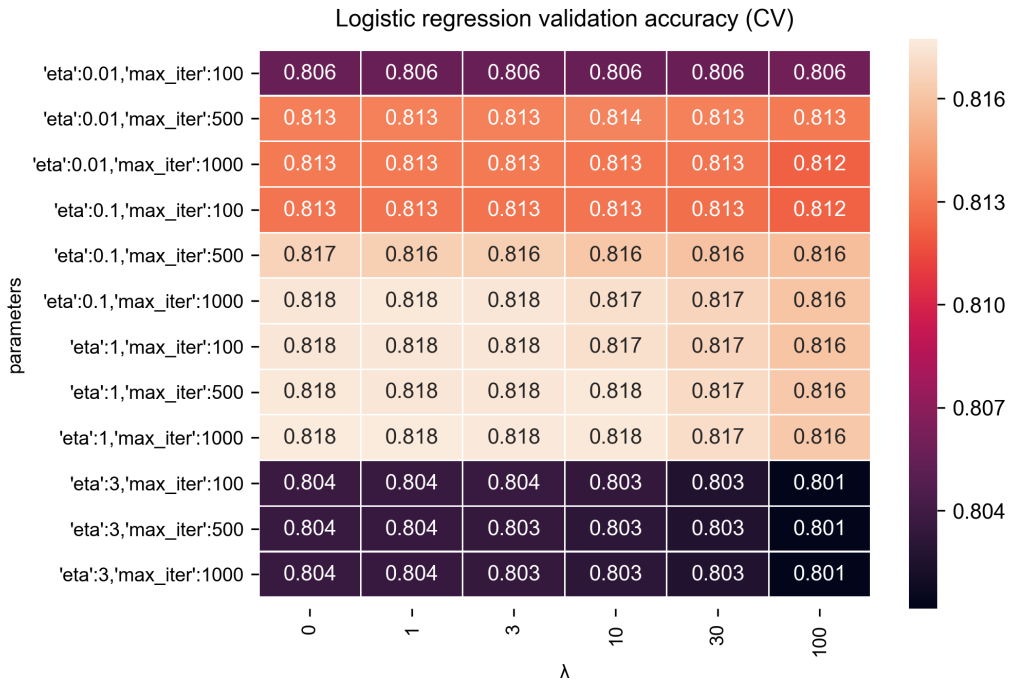


Figure 14: Heatmap of the accuracy scores for the logistic regression model. The scores are evaluated on validation sets within the training data, using 5-fold cross validation.

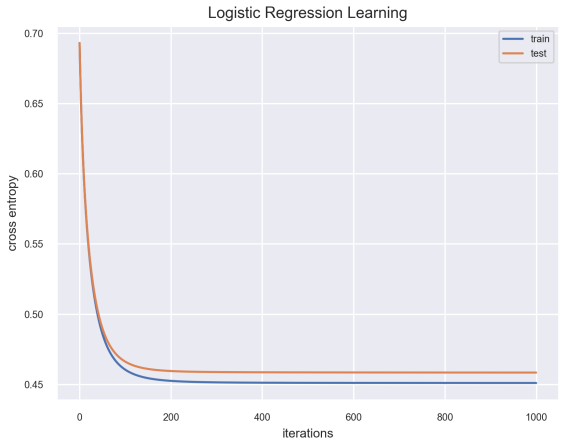
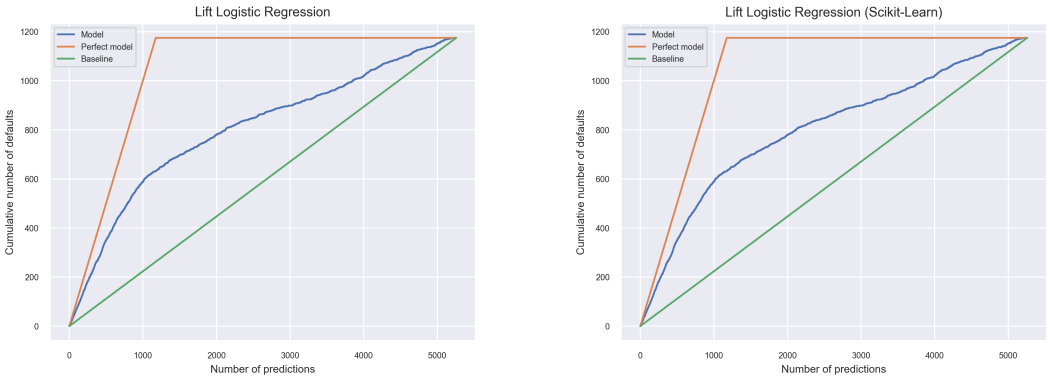


Figure 15: Learning during training for our implementation of logistic regression.



(a) Our logistic regression model

(b) Scikit-Learn regression model

Figure 16: Cumulative gains chart logistic regression

Parameter	Description	Values
'layer_sizes'	The size of the nodes in the hidden layers	[10] [10] [50] [50,20] [30, 20, 10]
'activation_functions	Activation function in the hidden layers	'sigmoid' 'relu'
'lambda'	L2 regularization	0 0.01 0.03 0.1 0.3
'epochs'	Number of total forward-backward passes	10 40

Table 3: Parameter grid for GridSearchCV for the neural network

5.1.2 Neural network implementation

The hyperparameters space that was tested for the neural network classifiers are shown in table 3. In order to narrow the search space we used a fixed learning rate η of 0.005 and a batch size of 128, as this seemed to work well in initial runs. The resulting accuracies on the validation sets are shown as a heatmap in figure 17.

Astonishingly, when comparing figure 17 with figure 14, we see that all the network models obtain higher accuracies than the best logistic regression models. However, the range of results is small and the best models have an accuracy of 0.825, less than one percent higher than the logistic regression models. Several models, with varying number of nodes, training epochs and regularization achieve this accuracy, a testament to the adaptability of neural networks. Technically, the highest score was somewhat surprisingly a model with only 10 nodes in a single hidden layer. As expected, 10 epochs seems low when training the two most complex models. The accuracy scores in the first column ($\lambda = 0$) seem to be the lowest overall, which shows that some regularization appears to be beneficial. As best model we continue with a neural network with 10 neurons, a λ of 0.01 and the fixed learning rate of 0.005, trained for 30 epochs.

A learning chart displaying accuracy of the best model refitted to the entire training data is shown in figure 18. There are no clear signs of overfitting; the difference between values on the training set and test set is small and it is not obvious that the test accuracy is

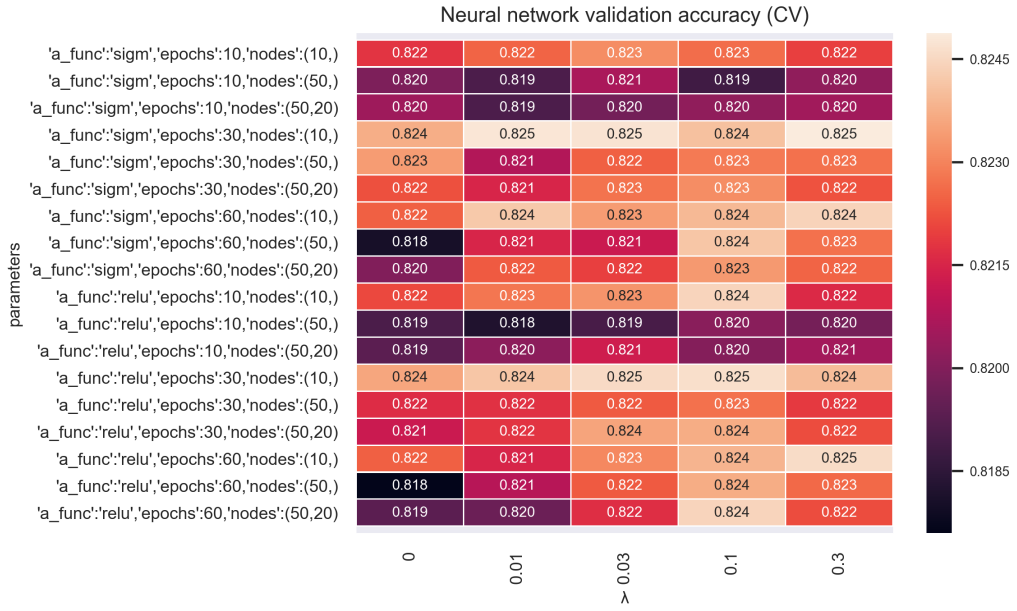


Figure 17: Heatmap of the accuracy scores for our neural network implementation. The scores are evaluated on validation sets within the training data, using 5-fold cross validation.

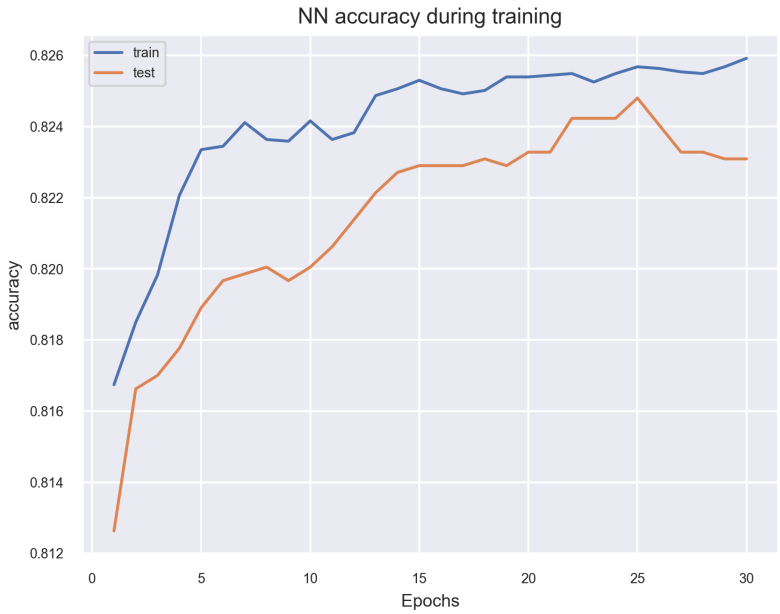


Figure 18: Learning during training for our neural network implementation.

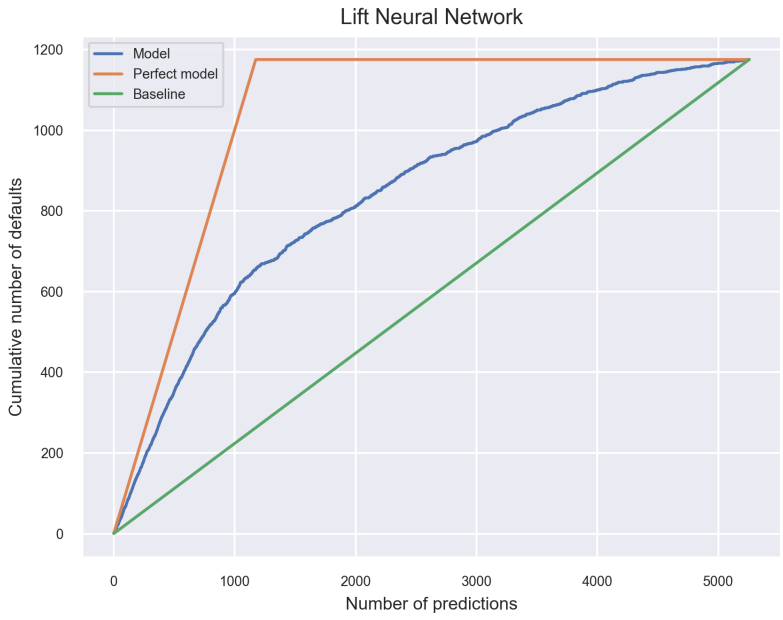


Figure 19: Cumulative gains chart for our neural network implementation.

falling towards the end of training. Figure 19 shows the cumulative gains chart.

5.1.3 Neural network Keras/Tensorflow

A batch size of 32, which is the default, was used for all Keras models. We have also used the default settings for the Adam optimizer, which involves an adaptive learning rate and momentum. The resulting accuracies on the validation sets are shown as a heatmap in figure 20.

It is immediately obvious that regularization $\lambda \geq 0.01$ does not go well for trials using the sigmoid activation function. This could have something to do with the fact that the Adam optimizer does not do well in combination with L2 regularization [4].

For the ReLU models the regularization does seem to work, but too strong regularization ($\lambda = 0.1$) is also not good. It seems that stronger regularization requires longer training and even 60 epochs is not enough. So, combining the Adam optimizer with the ReLU activation function makes regularization possible and even better ($\lambda = 0.01$), but a longer training time is needed. We could not figure out why the Sigmoid activation is such a bad combination with L2-regularization.

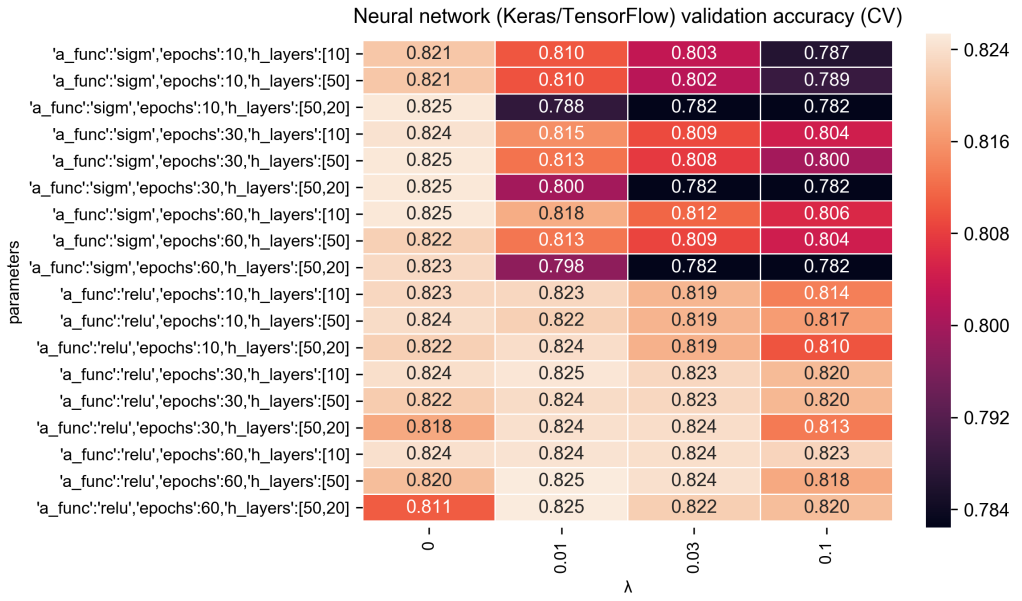


Figure 20: Heatmap of the accuracy scores for the Keras/Tensorflow neural network. The scores are evaluated on validation sets within the training data, using 5-fold cross validation.

As for our self-implemented network, many models architectures perform equally well, with the best accuracies at 0.825. The highest scoring model during this grid search was a 50 node single layer model with $\lambda = 0.01$ that was trained for 60 epochs.

5.1.4 Classification model evaluation

The comparison between optimized models in table 4 shows that neural networks are the better models. They outperform logistic regression on the area ratio metric and give slightly higher accuracy and precision scores. Their superior area ratio is reflected in the confusion matrix, with fewer false negatives, meaning that the cumulative gains curve stays at a higher level towards the tail end. The cost is a somewhat higher rate of false positives. Our implementation has higher accuracy on the test set compared to the Keras/Tensorflow implementation, but this is most likely a coincidence, as it was different for different runs. The exact status of the weights at the moment training is stopped can influence the result, as seen in figure 18. This depend on the learning rate and batch size. Cross validated data in figures 17 and 20 (heatmaps) show that the models perform equally well.

These results were obtained on a data set *without* dimensionality reduction (PCA), but with outliers (according to our definition) removed. In appendix 4 there are two other tables with results, one for the dataset with PCA applied and one on the original dataset,

Model	**Accuracy score**	**Confusion matrix**	**Precision score** (for zero)	**Area ratio**
'Logistic regression ($\lambda = 0$, $\eta = 1.0$, max_iter = 1000)	0.814	[3942 140] [839 336]	0.82	0.469
'Logistic regression Scikit-Learn (C = 1, max_iter = 1000)	0.814	[3942 140] [839 336]	0.83	0.469
'Neural network (activation = 'sigmoid', batch_size = 128, epochs = 30, $\eta = 0.005$, $\lambda = 0.01$, n_hidden_neurons = 10)	0.823	[3856 226] [704 471]	0.85	0.562
'Neural network Keras (activation = 'ReLU', batch_size = 32, epochs = 60, $\lambda = 0.01$, n_hidden_neurons = 50)	0.818	[3871 211] [745 430]	0.84	0.545

Table 4: Classification model scores

i.e. with outliers included and no PCA. We observed that results with or without outliers and PCA did not significantly impact the metrics.

Our implementation of logistic regression and Scikit-Learn's implementation perform virtually identically, suggesting the true minimum is found by both algorithms. Technically, the number of iterations was smaller for the best Scikit-Learn model compared with the registered best self-implementation, but also the latter converged before 100 iterations was reached with $\eta = 1.0$.

5.2 Regression estimate of Franke's function

We used a similar network as for the classification problem, but with a different cost function and no activation function as described in the theory section. We also test a Keras/Tensorflow model, applying 5-fold cross validation on all results.

When comparing the neural network regressor with the linear regression methods, we use the optimal polynomial degree found in project one, i.e. a polynomial degree of 7. Of course we use the same number of data points (100 x 100) and noise $\sigma = 0.1$.

Parameter	Description	Values
'layer_sizes'	The size of the nodes in the hidden layers	[10] [100] [100, 20] [128, 64, 32]
'activation_functions	Activation function in the hidden layers	'sigmoid' 'relu'
'batch_size'	Subset of the data, the number is the number of parts the dataset is divided into	16 32 64

Table 5: Parameter grid for GridSearchCV for the neural network regressors

For the neural network regression we start by finding the best values for η (learning rate) and λ (regularization) parameters using a network with (100,20) nodes trained for 150 epochs with a batch size of 16. ReLu activation was applied to the hidden layers. We compare the MSE for different values for η and λ, seen in figure 21.

We see that the lowest MSE is 0.105 for $\lambda = 10^{-5}$ and $\eta = 0.01$. In project 1 we saw that for the same data, OLS regression performed the best, but the Ridge results were almost the same, possibly slightly better for a fine tuned λ. In the case of the neural network regressor, λ is the l2-regularization, same as Ridge (which also uses l2-regularization). In project 1 we found an optimal value of λ = 1e-7 for Ridge. If regularization was beneficial for linear regression, it should also have a place in the neural network, which is more complex and prone to overfitting. We see that the MSE-scores for low values for both λ and η are notably better than the MSE-scores for high values of λ and η.

While being aware that our optimal λ and η where adjusted using the model specified above, we continue by applying GridSearchCV to find the good values for the layer sizes, the activation function and the number of epochs. This was run on only the training data (consisting of 80% of the entire data set shuffled), and thereafter refitted to the entire training set with metrics calculated for the separate test set (20% of the data). The parameter grid is defined in table 5. The gridsearch is ran on both our neural network as well as the Keras/Tensorflow neural network. The best parameters are listed in table 7).

Table 6 and 7 show the results from project 1 and for the neural networks with optimized parameters, respectively. We can see that only after several decimals is there a difference between the neural networks. Keras/Tensorflow is slightly better than our network. The difference is so small that it is hard to read too much into it, but one factor could be that

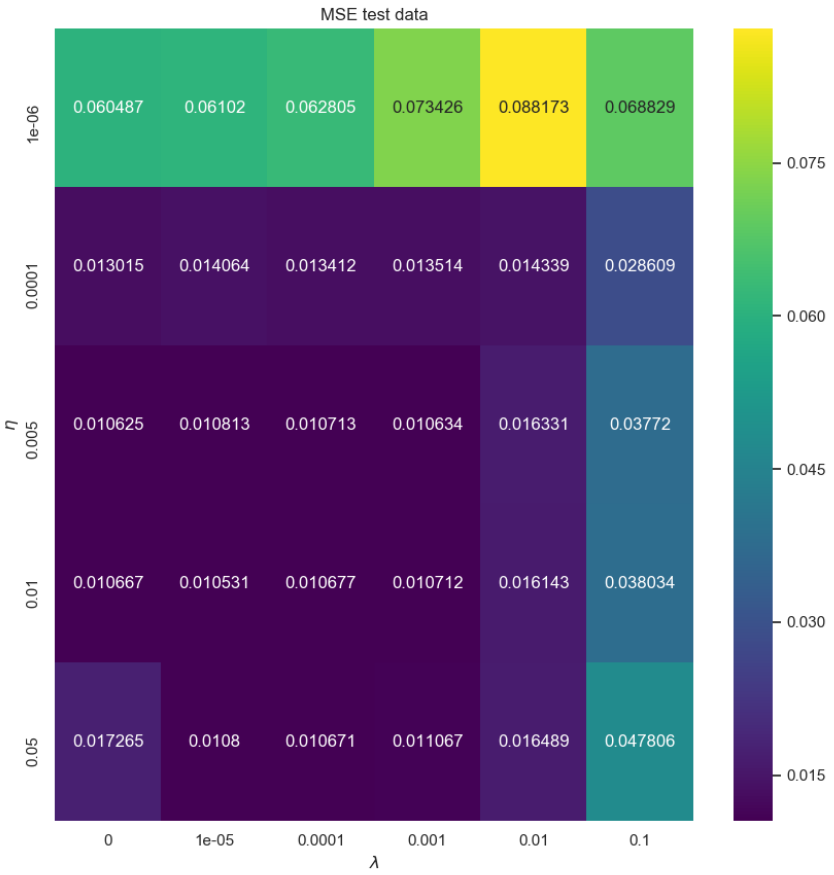


Figure 21: Neural network regression, MSE on the test data for different values for η and λ, using a network with (100,20) hidden nodes trained for 150 epochs with a batch size of 16. ReLu activation was applied to the hidden layers.

	MSE-score	**R2-score**	**MSE-score** Real data, no cv	**R2-score** Real data, no cv
OLS	0.0108 (+/- 0.000)	0.884(+/- 0.007)	0.001	0.993
Ridge (λ = 10e-7)	0.0118 (+/- 0.000)	0.873 (+/- 0.004)	0.001	0.993
LASSO $\lambda = 10e-7$	0.030 (+/- 0.000)	0.683 (+/- 0.009)	0.019	0.772

Table 6: OLS, Ridge and LASSO regression scores using a 7th-degree polynomial and 5-fold cross-validation (project 1).

	Parameters	**MSE-score**	**R2-score**
Neural Network	layer_size = (100,20) epochs = 150 batch_size = 16 η = 0.01 λ = 0 activation_function = 'relu'	0.0104 (+/- 0.000)	0.876 (+/- 0.004)
Neural Network Keras	layer_size = (100,20) epochs = 150 batch_size = 16 η = 0.01 λ = 0 activation_function = 'relu'	0.0103 (+/- 0.001)	0.877 (+/- 0.005)
Optimized Neural Network	layer_size = (128,64,32) epochs = 150 batch_size = 32 η = 0.01 λ = 1e-5 activation_function = 'relu'	0.010 (+/- 0.001)	0.886 (+/- 0.007)
Optimized Neural Network Keras	layer_size = (128,64,32) epochs = 150 batch_size = 32 η = 0.01 λ = 1e-5 activation_function = 'relu'	0.010 (+/- 0.001)	0.887 (+/- 0.006)

Table 7: Neural network (our own and Keras) regression scores

Keras uses an adaptive optimizer, while ours uses a mini-batch gradient descent with a fixed learning rate.

We observe that we are able to get slightly better predictions with a neural network regressor than with an OLS, Ridge or LASSO regression model. However, the whole parameter optimization costs a lot of computational power and thus time (this code with the grid-search and cross-validation takes about 30 minutes already). When taking into account the small improvement, it may not be worth it. Neural network regression for this data could be considered an overkill, because Franke's function already lends itself pretty well linear regression using sufficiently high order polynomials.

6 Conclusion

For predicting probabilities of default among credit card users, neural networks proved to be the best models. As evident from the grid search, a wide range of model architectures and hyperparameters can give good results on this data set. Our own implementation of the neural network gave results on par with the Keras/Tensorflow libraries, which is encouraging. The best models gives an accuracy of 0.825 cross validated and an area ratio of 0.56, which is comparable to the results from Yeh et al. [2] (error rate 0.17 and area ratio 0.54 for neural network). Strangely, Yeh et al. report a lower error rate on the validation data than on the training data for all but one of their models. Being the simpler and faster model, logistic regression also performs quite well on this data set. Although scoring lower accuracy an area ratio, it actually has fewer false positives. This, of course, comes as the expense of more false negatives. Logistic regression comes across as a more optimistic model in this sense, i.e. tending to give customers the benefit of the doubt and classify them as not likely defaulting.

In our opinion, feature reduction with PCA is not necessary on this data set. It speeds up learning somewhat as there are fewer weights in the network. But there are "only" 26 original features (including categorized variables) in the data set and it is clear what they represent. Intuitively, most of them could have an impact on whether the customer defaults or not. Also, unless very complex models are fitted, overfitting is not a big problem for this data set.

Theoretically, the fact that we exclude some outliers from the calculations should improve our metrics and bias our results favorably when compared to Yeh et al. However, we observe very little difference if whether we include them or not. Afterall, we remove only a small number of data points compared to the size of the set.

Compared with the linear regression methods investigated in a separate report, neural networks are slightly better at estimating Franke's function. When fitting a $7th$ order polynomial, the best models achieve a cross validated mean squared error of 0.010, compared with 0.011 with OLS regression. A very big improvement from OLS was not expected as a $7th$ order polynomial already allows for quite detailed fits even with a linear model. Again, our implementation and Keras/Tensorflow yield very similar results. Our approach on the regression problem was to first fix the optimal learning rate and regularization for one specific model architecture. But of course, the optimal parameters will not be exactly same for the other architectures tested in the grid search afterwards. Some more experimentation with parameters could possibly yield an even better results, but probably only marginally.

7 References

References

[1] J.-C. T. Chih Hsiung Chang, Heidi H. Chang, "A study on the coping strategy of financial supervisory organization under information asymmetry: Case study of taiwan's credit card market," *Universal Journal of Management*, vol. 5(9), pp. 429–436, 2017.

[2] J.-C. T. Chih Hsiung Chang, Heidi H. Chang, "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients," *Expert Systems with Applications*, vol. 36, pp. 2473–2480, 2009.

[3] J. Han and M. Kamber, *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients.* 2006.

[4] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

8 Appendix

.1 Appendix 1

1.1 Code logistic regression

classes_classification.py — ~/Documents/GitHub/FYS-STK4155_1

classes_classification.py

```python
class logisticRegression(BaseEstimator, ClassifierMixin):
    #inherits properties from Scitkit-learn objects to be able to perform gridSearchCV
    def __init__(self, _lambda = 0, eta = 0.1, max_iter = 1000):
        self._lambda = _lambda; self.eta = eta; self.max_iter = max_iter

    def fit(self, X, y):
        _lambda = self._lambda; eta = self.eta
        X = np.hstack((np.ones(X.shape[0])[:,None], X))
        self.beta = np.zeros(X.shape[1])[:,None]
        n = X.shape[0]

        #run full gradient descent
        for i in range(self.max_iter):
            y_pred = sigmoid(X@self.beta)
            grad = X.T@(y_pred - y)/n + np.vstack((0,_lambda*self.beta[1:]/n))
            self.beta = self.beta - eta*grad

    def fit_track_learning(self, X, y, X_test, y_test, plot = False, savefig = False, filename = ''):
        _lambda = self._lambda; eta = self.eta
        X = np.hstack((np.ones(X.shape[0])[:,None], X))
        X_test = np.hstack((np.ones(X_test.shape[0])[:,None], X_test))
        self.beta = np.zeros(X.shape[1])[:,None]
        n = X.shape[0]
        cross_entropy_train = np.zeros(self.max_iter)
        cross_entropy_test = np.zeros(self.max_iter)

        #run full gradient descent
        for i in range(self.max_iter):
            y_pred = sigmoid(X@self.beta)
            y_pred_test = sigmoid(X_test@self.beta)
            cross_entropy_train[i] = categorical_cross_entropy(y_pred,y)
            cross_entropy_test[i] = categorical_cross_entropy(y_pred_test,y_test)
            grad = X.T@(y_pred - y)/n + np.vstack((0,_lambda*self.beta[1:]/n))
            self.beta = self.beta - eta*grad
        if plot:
            plot_several(np.repeat(np.arange(self.max_iter)[:,None], 2, axis=1),
                np.hstack((cross_entropy_train[:,None],cross_entropy_test[:,None])),
                ['r-', 'b-'], ['train', 'test'],
                'iterations', 'cross entropy', 'Logistic Regression Learning',
                savefig = savefig, figname = filename)
        return cross_entropy_train[i], cross_entropy_test[i]

    def get_proba(self, X):
        X = np.hstack((np.ones(X.shape[0])[:,None], X))
        y_pred = sigmoid(X@self.beta)
        return y_pred

    def predict(self, X):
        y_pred = self.get_proba(X)
        y_pred_outcome = np.zeros(y_pred.shape)
        y_pred_outcome[y_pred>=0.5] = 1
        return y_pred_outcome
```

classes_classification.py 1:1 LF UTF-8 Python master Fetch GitHub Git (6)

1.2 Code softmax function

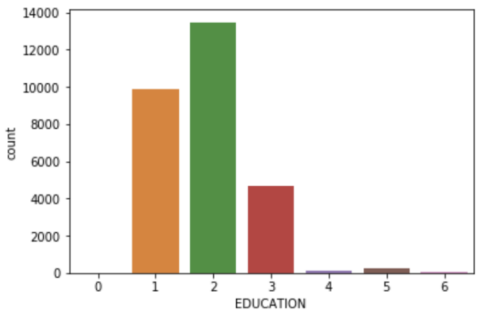
```python
def softmax(x):
    #made numerically stable by subtracting max values
    exp_term = np.exp(x - np.amax(x,axis = 1, keepdims = True))
    return exp_term / np.sum(exp_term, axis=1, keepdims=True)
```

1.3 Code backpropagation algorithm

```python
    def backpropagation(self):

        #initialize error and gradient lists
        error = [0]*(self.n_hidden_layers + 1)
        w_grad = [0]*(self.n_hidden_layers + 1)
        bias_grad = [0]*(self.n_hidden_layers + 1)

        #calculating error and gradients for the output layer
        error[self.n_hidden_layers] = self.probabilities - self.Y_data #always the case if using cross entropy (and softmax last layer?)
        w_grad[self.n_hidden_layers] = self.a[self.n_hidden_layers-1].T@error[self.n_hidden_layers] + self.lmbd * self.w[self.n_hidden_layers]
        bias_grad[self.n_hidden_layers] = np.sum(error[self.n_hidden_layers], axis = 0)

        #looping back through hidden layers
        for i in range(self.n_hidden_layers-1,0,-1):
            error[i] = error[i+1]@self.w[i+1].T * self.activation_z_derivative(i)
            w_grad[i] = self.a[i-1].T@error[i] + self.lmbd * self.w[i]
            bias_grad[i] = np.sum(error[i], axis = 0)

        #calculating error and gradients for the first hidden layer
        error[0] = error[1]@self.w[1].T * self.activation_z_derivative(0)
        w_grad[0] = self.X_data.T@error[0] + self.lmbd * self.w[0]
        bias_grad[0] = np.sum(error[0], axis = 0)

        #updating weights and bias
        self.w = list(map(add, self.w, [-i*self.eta for i in w_grad]))
        self.bias = list(map(add, self.bias, [-i*self.eta for i in bias_grad]))
```

1.4 Running classification models

```
bingo@bingo-G56JK:~/python/uio_ml/FYS-STK4155/Project2$ python3 data_classification.py
Using TensorFlow backend.
/home/bingo/.local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:516: F
  np_resource = np.dtype([("resource", np.ubyte, 1)])
  ------------------------------------
  -
  -
Value count payment default: 0    23364
1     6636
Name: defaultPayment, dtype: int64
Accuracy neural network test: 0.818528
```

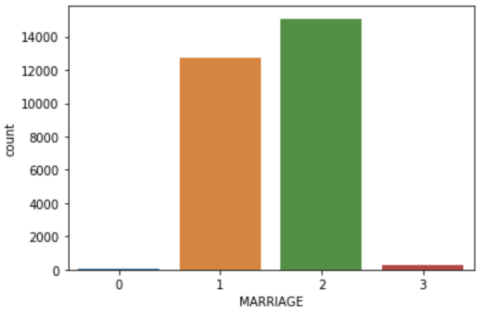


(a) Countplot of the education variable

```
df['EDUCATION'].value_counts()
```

```
2    13477
1     9893
3     4684
5      266
4      116
6       48
0       13
Name: EDUCATION, dtype: int64
```

(b) Number of values per category education variable



(a) Countplot of the marriage variable

```
df['MARRIAGE'].value_counts()
```

```
2    15086
1    12726
3      309
0       49
Name: MARRIAGE, dtype: int64
```

(b) Number of values per category marriage variable

.2 Appendix 2

2.1 Categorical variables

The education variable

Figure 22a and 22b show the original education values, with a few values for 0, 5 and 6.

The marriage variable

Figure 23a and 23b show the original values for marriage, where there are 49 values for 0.

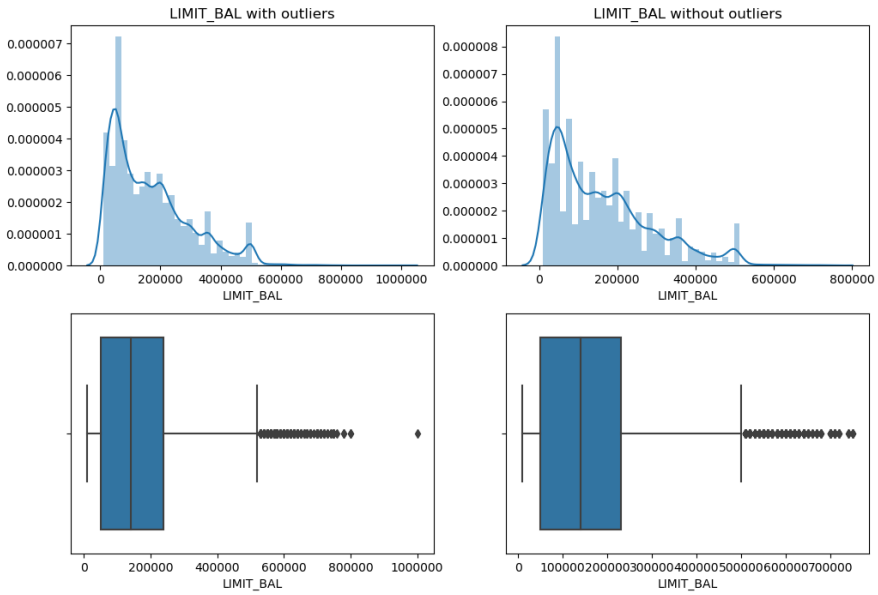


Figure 24: Distribution and boxplot of the LIMIT_BAL variable with outliers (left) and without outlier (right)

2.2 Numerical variables

The LIMIT_BAL variable

In figure 24 we see that after removing the outliers in the LIMIT_BAL variable both the distribution and the barplot has improved. The barplot still shows outliers, this is because we did not use the 'rule of thumb' but only removed very high values.

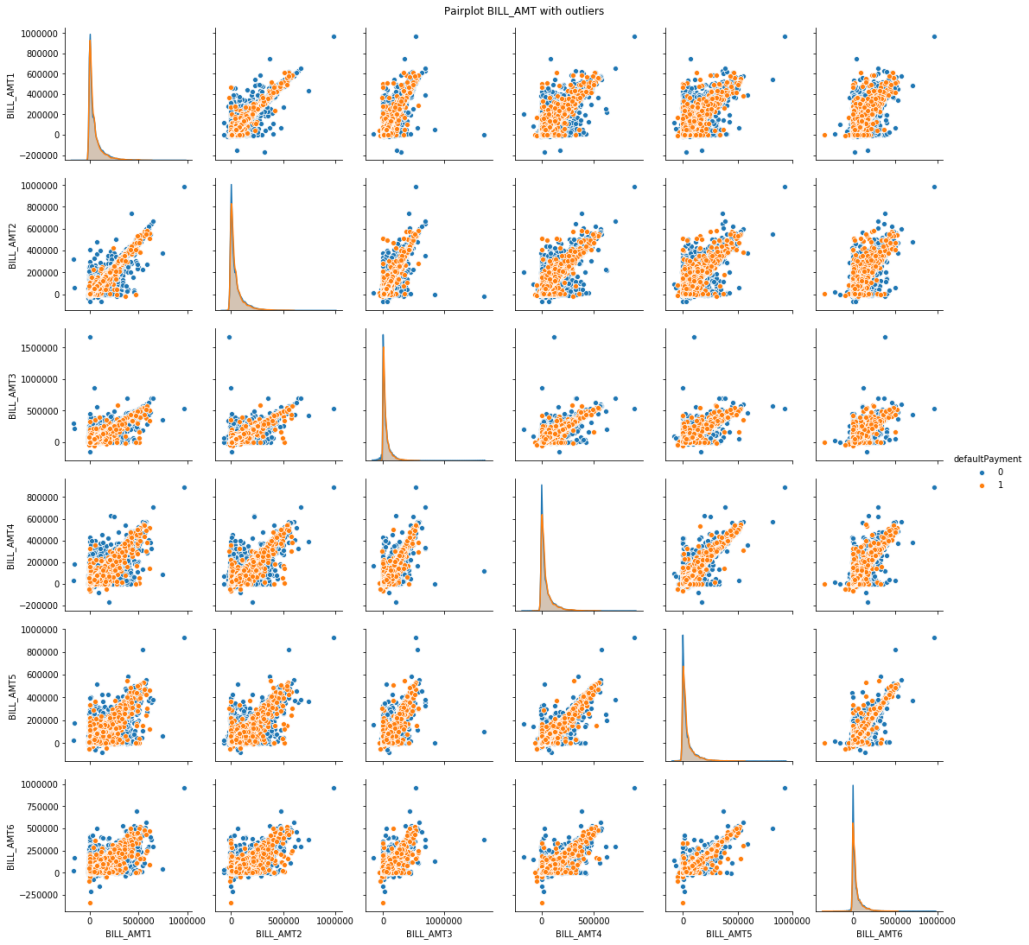


Figure 25: BILL_AMT pairplot with outliers

The BILL_AMT variable

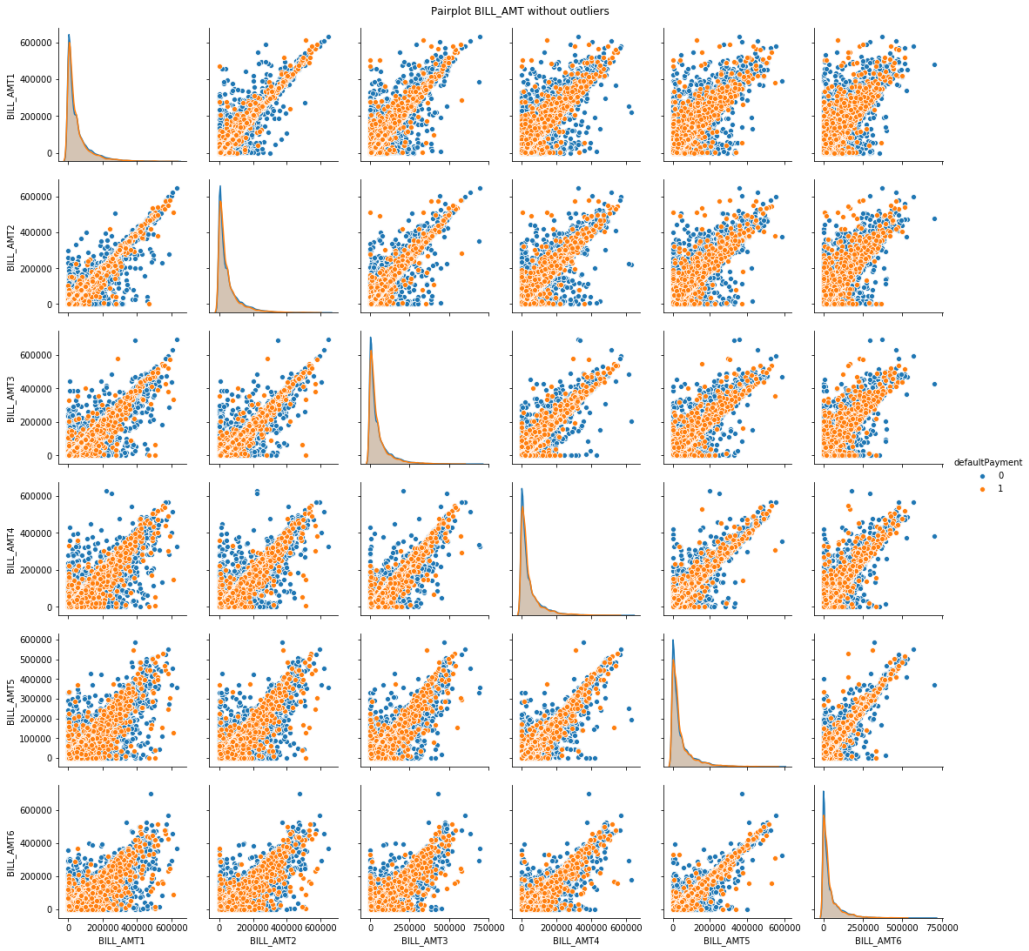


Figure 26: BILL_AMT pairplot without outliers

We see that both the distribution and the scatterplots have improved after removing the outliers.

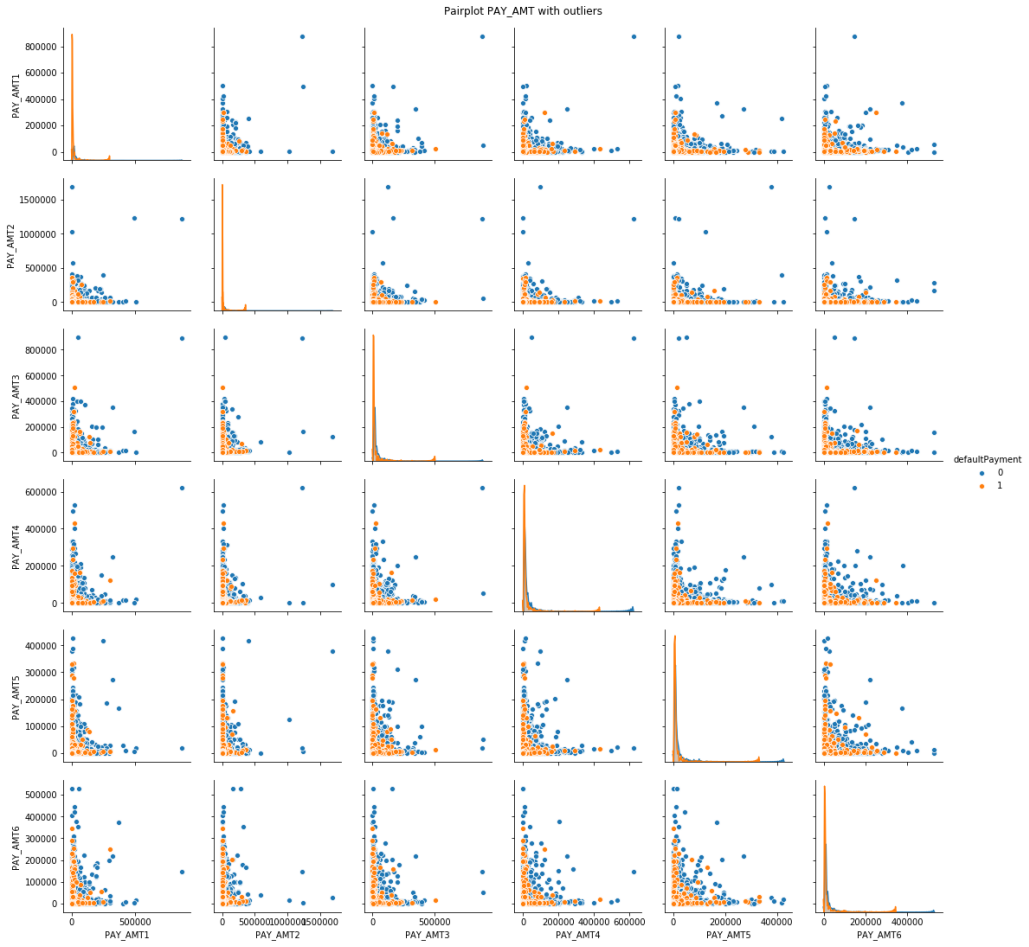


Figure 27: PAY_AMT pairplot with outliers

The PAY_AMT variable

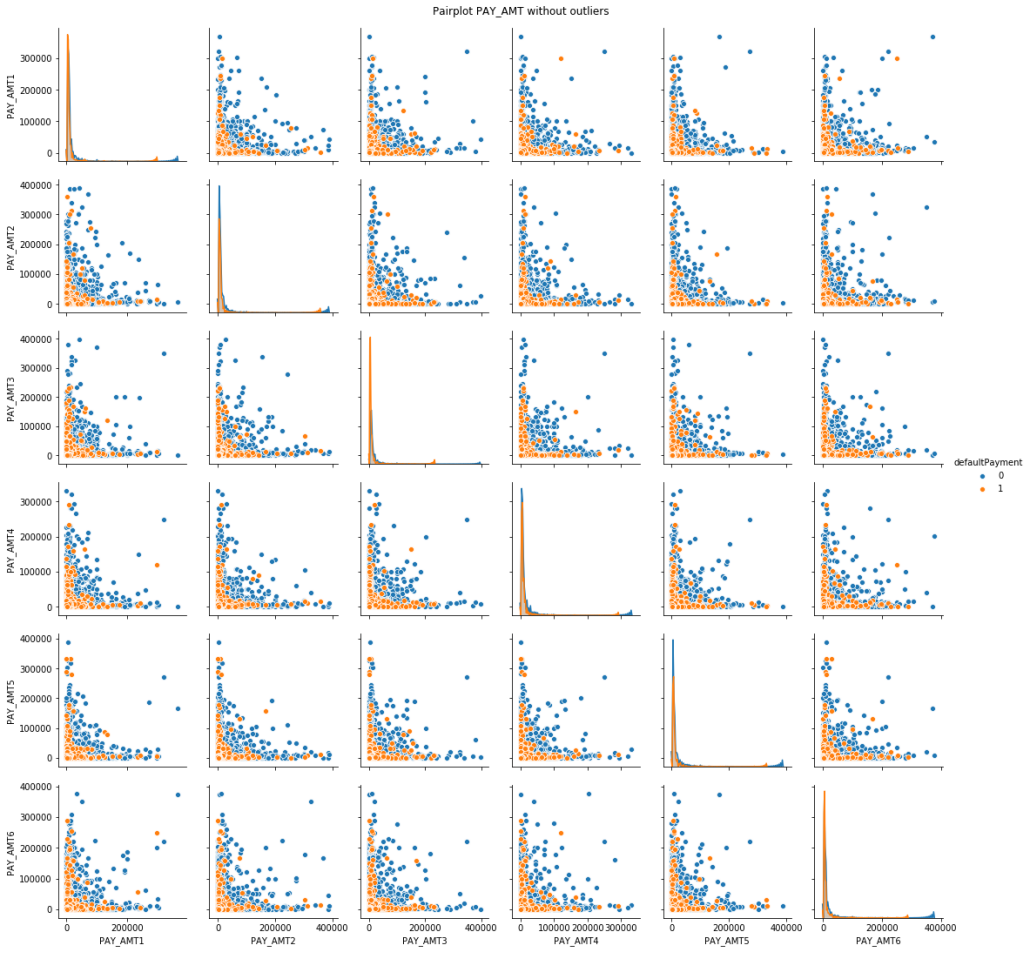


Figure 28: PAY_AMT pairplot without outliers

We see that both the distribution and the scatterplots have improved after removing the outliers.

Parameter	Description	Values
'lambda'	L2 regularization	0.01 0.1 1.0 10 100
'max_iter'	Number of iterations	100 500 1000

Table 8: Parameter grid for GridSearchCV for Scikit-Learn logistic regression

.3 Appendix 3

3.1 Gridsearch Scikit-Learn logistic regression mode

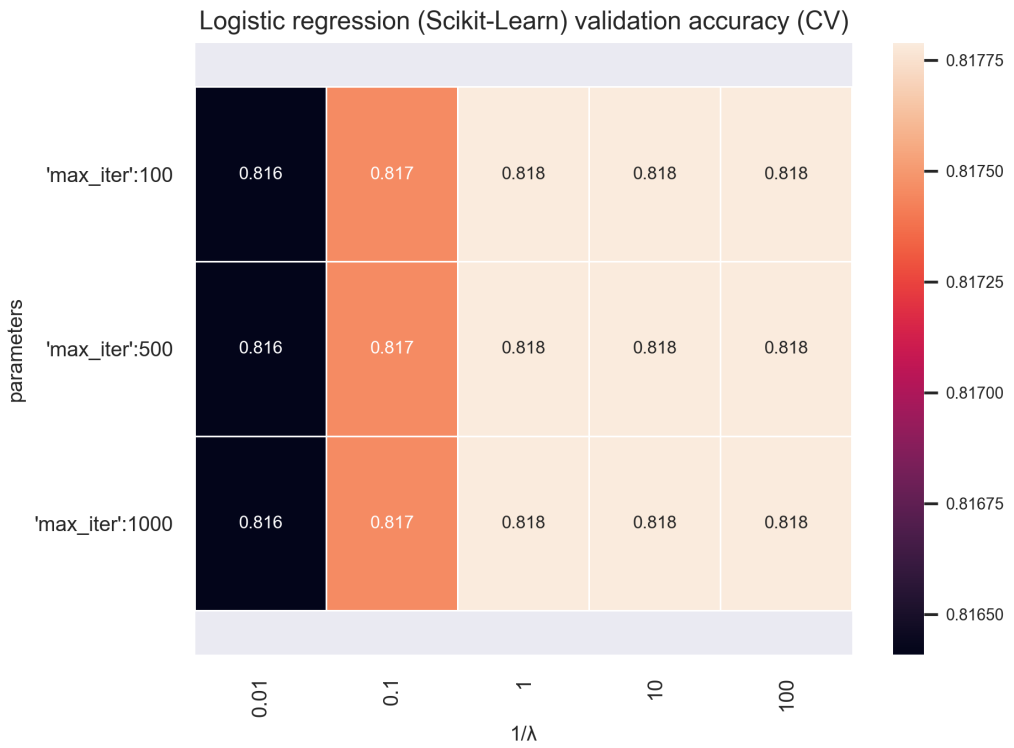


Figure 29: PAY_AMT pairplot with outliers

3.2 Results gridsearch Scikit-Learn logistic regression model

Model	**Accuracy score**	**Confusion matrix**	**Precision score** (for zero)	**Area ratio**
'Logistic regression ($\lambda = 0$, $\eta = 1.0$, max_iter = 1000)	0.817	[4258 133] [898 336]	0.83	0.458
'Logistic regression Scikit-Learn (C = 1, max_iter = 100)	0.817	[4259 132] [897 337]	0.83	0.458
'Neural network (activation = 'sigmoid', batch_size = 32, epochs = 30, $\eta = 0.005$, $\lambda = 0.01$, n_hidden_neurons = 10)	0.818	[4236 155] [866 368]	0.83	0.529
'Neural network Keras (activation = 'ReLU', batch_size = 32, epochs = 60, $\lambda = 0.01$, n_hidden_neurons = 50)	0.824	[4141 250] [742 492]	0.85	0.543

Table 9: Model scores for the dataset with outliers and no PCA

.4 Appendix 4

4.1 Model evaluation

On the dataset with outliers and no PCA.

Model	**Accuracy score**	**Confusion matrix**	**Precision score** (for zero)	**Area ratio**
'Logistic regression ($\lambda = 3$, $\eta = 1.0$, max_iter = 500)	0.810	[3936 146] [853 322]	0.82	0.475
'Logistic regression Scikit-Learn (C = 1, max_iter = 100)	0.810	[3936 146] [852 323]	0.83	0.475
'Neural network (activation = 'sigmoid', batch_size = 32, epochs = 60, $\eta = 0.005$, $\lambda = 0.03$, n_hidden_neurons = 10)	0.820	[3848 234] [711 464]	0.84	0.564
'Neural network Keras (activation = 'ReLU', batch_size = 32, epochs = 10, $\lambda = 0.01$, n_hidden_neurons = (50,20))	0.820	[3857 225] [720 455]	0.84	0.550

Table 10: Model scores for the dataset with outliers and no PCA

4.2 Model evaluation

On the dataset with the outliers removed and dimensionality reduction included (PCA).