# Project 3
# FYS-STK3155/4155

Magnus Bergkvam

December 18, 2023

## Contents

# 1 Abstract

In this project we explored how different models performed for a binary classification of credit-card fraud-data. For this we looked at three different types of models, one being a simple logistic regression, another one being neural networks and lastly gradient-boosted decision trees. What we found was that the logistic regression without any polynomial features managed 96.5% classification accuracy on the test-data, with this increasing to 99.7% when adding second order polynomial features. The neural network was only marginally able to outperform this with 99.8% final accuracy. The best performance by quite a was the gradient boosted decision tree, ending up with more than 99.9% accuracy on the test-data.

Overall this project has shown that for credit card fraud detection, logistic regression and neural networks, with some limitations applied, serve as quite good classifiers. However, neither of these methods surpassed gradient boosting in terms of accuracy, which at the moment seems to be one of the best classifiers for this data.

# 2 Introduction

Machine learning classifiers have many important uses, from spam classification of emails, to image classification in the medical sector and much more. Credit card fraud detection is one of many applications where machine learning binary-classification models are widely used [1]. Furthermore it is a case where having a good classifier is essential for the credit-card system to work in the first place. It is important that the vast majority of legitimate purchases are detected as legitimate, as the reliability of the credit cards is important if you want people to use your credit cards. For example, when going to the store and buying something you should not have to worry about the card being declined. At the same time it is perhaps even more important that the actual frauds are detected as such, as the credit-card company can then take action to block the payments and maybe even the card. This is both important for the user of the credit card and the issuer, as it can be a huge hurdle for the owner if fraud payments are not detected and go through normally, which is both uncomfortable and time-consuming. Additionally for the issuer, the more fraud payments they let go through, the more of these fraudulent payments they potentially have to cover themselves.

What is clear is that a very good classifier is needed in order to limit the situations discussed. We will here try to explore some different models for classifying fraud and see how good they perform. The dataset we will be using contains more than 550000 transactions, each with 28 anonymized features plus the amount spent, and a binary response representing fraud or not fraud. We will focus mainly on three types of models. The simplest one being logistic regression, then we will go back to neural networks as we found that they per-

formed very well in project 2 [2] and finally we will look at a decision tree, more specifically a gradient boosted decision-tree.

According to prior studies, various ensemble methods building upon decision trees seem to be currently perceived as the best classifiers, with especially gradient boosting and random forest classifiers performing very well, certainly better than a plain logistic regression model [3, s. Model Performance Conclusions]. One motivation behind using gradient-boosting methods here as well is to see if we can gain an even better classifier by tweaking some of the hyperparameters. They also are often a natural choice for binary classification scenarios due to their often good performance for this task. A justification for introducing a logistic regression model here is that logistic regression is easily extensible by creating new features, and we will see if doing some feature engineering will make the logistic regression model better, perhaps even to the point where it is competitive with the best ensemble methods. Using neural networks is also a natural choice in a lot of different scenarios, binary classification included as we saw in project 2, and perhaps especially in this case as we have a lot of data at hand.

We will start this report with the method, where we introduce more details in some of the models we will use, and go through the data a bit closer. Thereafter, we'll look at how the three different models perform, and from there conclude which of the three models performs the best for this data.

# 3 Method

## 3.1 The credit card fraud data

At the heart of all our experimentation is the credit-card fraud detection data from 2023 [4]. This is a dataset where each observation corresponds to one transaction. Luckily for our models, the dataset is quite rich on data with there being 568630 total observations/transactions. Furthermore this data is equally split among the two classes, with there being 284315 non-fraudulent and fraudulent transactions each. This is probably going to be a good thing for our models, as the models has then as much information of what constitutes a fraudulent transaction, as a non-fraudulent transaction.

The data is collected using transactions from European credit card holders in the year 2023 [4]. The fact that we only have data collected from Europeans does perhaps limit the performances of our models on non-european card holders. If for example in another part of the world a legitimate credit-card transaction typically looks different, or likewise with fraud, our models will probably struggle with predictions. Each transaction contains a unique identifier (labeled *id*), 28 anonymized features (labeled *V1* to *V28*), the transaction amount (labeled as *Amount*) and a categorical variable indicating whether we have a fraudulent transaction (labeled as *Class*). For *Class* we have that 1 constitutes fraud while 0 constitutes a legitimate transaction. We will obviously not use *id* for training

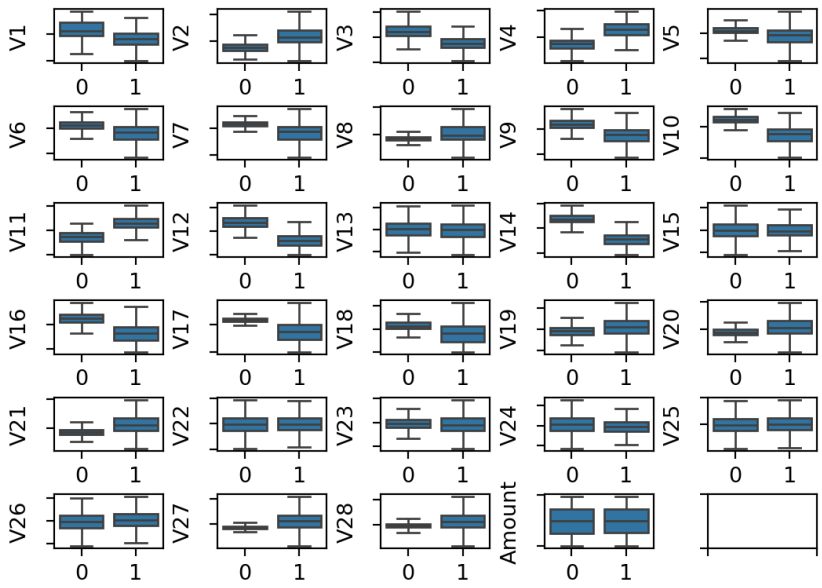


Figure 1: Shows a boxplot with each feature along the y-axis and the binary response on the x-axis. This plot was generated from a random sample of 10000 observations of the data. We see that for the anonymized features (bar 13 15 22 23 24 25 26) there seems to be a clear difference in distributions for the feature-values for the non-fraudulent and fraudulent transactions.

here, as this should not impact whether a transaction is fraudulent or not, but we will use all the other variables in our models, with *Class* as the response. Another thing which makes this data quite convenient is that all of the features are numerical, with there being no NA/NULL values at all, which makes the required preprocessing here very small (the only preprocessing we will do is scaling the features using scikit-learns StandardScaler [5]).

To see some dependencies of the response for each explanatory variable we can create a box-plot for each combination of feature and response value. Figure 1 shows exactly this. As we can see it looks like especially a few of the anonymized features are distributed considerably differently for the fraudulent transactions than the legitimate ones. This difference in distribution of features looks promising for our classifications.

### 3.2 Logistic regression

The first and most simple method we will explore is logistic regression. I will not go into detail on how this method works here, as this already has been presented in project 2 [2], but I will give some details about our application of this method. In our code we will use the scikit-learn python-library for creating our models. We will here simply do a train-test split and train a logistic regression on the training-set and evaluate the performance on a test-set. For the features we will first try a model with just the plain features as described previously in 3.1, and thereafter try to extend this with polynomial features of second degree polynomials (with interactions). We will not add a higher polynomial-degree than this due to the fact that we have 29 features already, and a high polynomial order will give us considerably more features which can both lead to overfitting and to problems with memory (keep in mind that the dataset here is quite big). We will also add some regularization for our models where we will try both a $l_1$ and $l_2$ penalty for the model with the plain features, and just a $l_2$ penalty for the model with second order polynomial features. For finding the appropriate regularization parameter we will here take use of cross-validation. Here we will simply use the LogisticRegressionCV in scikit-learn which does this job for us. This will do a simple grid-search for us, on which we will use a grid of 9 values between $10^{-8}$ and 1 on logarithmic scale (with base 10).

### 3.3 Neural Networks

For the neural networks in this project we will use a basic multi-layer perceptron, this time using only tensorflow. I will assume knowledge of how this method works as we already went through this in project 2. For more details on the neural networks feel free to check out the report of project 2 [2, s. 3.7]. Most of our experimentation here will go towards optimizing the various hyperparameters in the network. Our strategy here will be to split the data into a train, validation and test datasets, and then pick the hyperparameters based on what gives us the best performance on the validation set. Unlike logistic regression, we will not use cross-validation here. The reason for this is that it would make the programs even more computationally expensive. Luckily for us though we have a lot of data, with the two classes being quite evenly distributed, so using this approach over something like cross-validation should be unproblematic.

The main things we will be tweaking is the regularization parameter (we will limit ourselves to use $l_2$ regularization), the optimization method (with the corresponding learning-rate), the activation functions and the network size. For this we will use a greedy method, starting with determining the optimal optimization method, where we for each method test on a grid of learning-rates and regularization parameters and picking the best one, while keeping the activation function and network-size fixed (we will use 2 hidden layers with each 100 neurons and the relu activation function). Having chosen the method, we explore the different activation functions using our chosen method and the same hidden layer sizes, again testing with a grid of learning-rates and regularization

parameters. Lastly we will try different hidden layer sizes for the chosen method and activation function, but now testing on a grid of different regularization-parameters and keeping the best learning-rate previously determined. In other words, we will not search for the optimal learning-rate as well for each of the hidden layers, but we assume that this optimal learning-rate is more or less the same for the different sizes of hidden layers (the reason for taking this approach is to drastically reduce the computational cost). The output function will be fixed to be the sigmoid function in this case as we are dealing with binary classification, while the cost-function will be the binary cross-entropy, both just as in project 2 [2]. A thing to note is that while searching for the optimal hyper-parameters and optimization-methods we are testing a lot of different models, therefore we need each model to be fitted relatively quickly in order for this search to finish in a reasonable amount of time. Because of this we will not fit and test these models on the full dataset as that would simply take too much time. Instead we simply choose a sample of 20000 observations in each the training and validation set and choose the model which gives the best performance of this smaller train and validation-set. Another measure we take to limit the time it takes to fit the models is that we will limit the training to one epoch when doing the model selection. When we finally have chosen our model we train and test on the full dataset and allow as many epochs as we need for convergence.

The optimization methods we will try in this case is just as in project 2, i.e. AdaGrad, Adam, RMSProp and SGD. We have dropped trying out normal gradient descent, because of it's uncompetitive performance in project 2, as well as the fact that our dataset now will be huge, meaning that the amount of epochs we are willing to give in the learning-process is severely limited, meaning it should have even worse prerequisites in this case. All these are built directly into tensorflow so the amount we need to implement here is quite limited.

For the activation-functions we will try out the relu and sigmoid (which we already have explored in project 2), as well as some other activation functions built into tensorflow. Notably these are swish, tanh and elu. We will now go through these three as we have not explored them already. For more the details on the sigmoid and relu, see the project 2 entries on these [2, s. 3.8].

#### 3.3.1 tanh

The tanh is given by:

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} [6].$$

This looks much like the sigmoid function, in that is has the same s-shape, but is different in the sense that the sigmoid limits us between 0 and 1, while the tanh limits us between $-1$ and 1. This may be preferable in some situations.

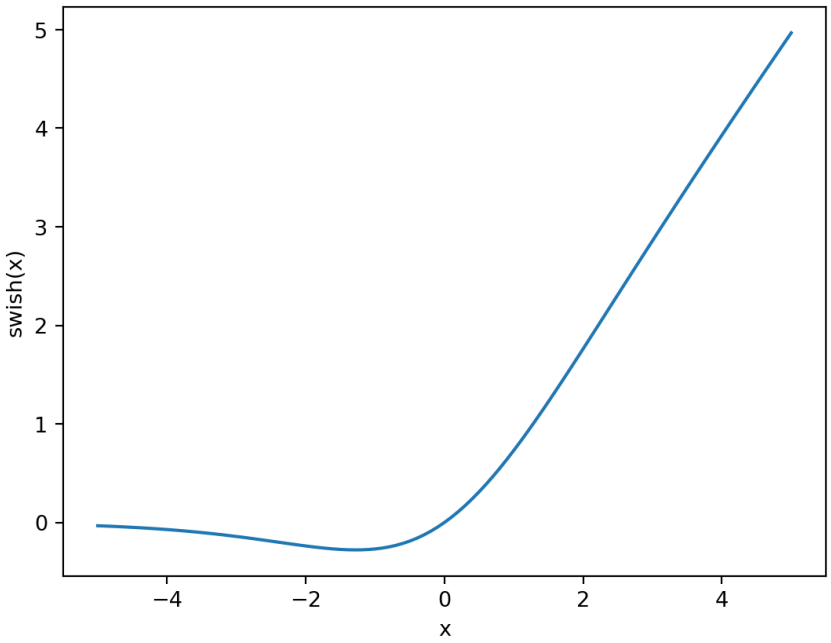


Figure 2: Shows how the swish-function behaves on $[-5, 5]$.

#### 3.3.2 swish

The swish activation-function is given by:

$$\text{swish}(x) = x \cdot \text{sigmoid}(x)[7].$$

Using our knowledge about the sigmoid-function it's clear that as $x$ grows, the sigmoid will approach 1 quite quickly. This means that for high values of $x$, the function behaves pretty much be the same as the relu. For negative values of $x$ we have that the sigmoid quickly approaches 0, much quicker than the linear part $x$ approaches $-\infty$, so the product will again quite quickly approach 0, again much like the relu. However for small values in absolute value we will get a shrinking in absolute value of the activations, as can be seen in figure 2. This does differentiate this from the ordinary relu. The swish activation-function often works better than the ReLU on deep models on challenging datasets [8], so we will explore if this is one of those cases.

#### 3.3.3 ELU

The exponential linear unit is given by:

$$\text{elu}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha \cdot (\exp(x) - 1) & \text{if } x < 0 \end{cases} \text{[9]}.$$

We will here just use $\alpha = 1$ (which is the default), but of course it is easy to test others as well. The ELU can lead to faster learning and better performance on neural nets with more than 5 layers in particular [10], which is less than what we will be using here, but we will try it anyway because including this takes essentially no code.

### 3.4 Boosting trees

This is a method we have not been through earlier, so we will go a little bit more into detail on this here. At the core of a boosting-tree is a normal classification-tree which we will use as a so-called weak classifier. We will first look a little bit into how this method works.

#### 3.4.1 Classification trees

A classification tree is a rather intuitive method. Like other decision trees they are typically divided into a root node, interior nodes and leaf nodes. The nodes are connected via branches, where the branch we move along with being dependent on a test on the node we are in, where the test has only two outcomes (for binary trees that is). Each of the leaf nodes contains the classification we should make [11]. To make predictions on a tree, we start at the root node and then move along the branch according to the test at the root-node until we come to a new node, where we do the same procedure until we end up at a leaf node, in which case we have our prediction.

For finding appropriate structure of the tree (that is the appropriate tests and nodes), perhaps the most popular algorithm is the CART algorithm. This uses what is called the Gini index, which we will denote $G$. Let $p_{m1}$ be the proportion of observations of credit card fraud (i.e. with $y_i = 1$) within a region $R_m$ and $p_{m0} = 1 - p_{m1}$ be the proportion of non-fraud cases within $R_m$. More formally we have:

$$p_{m1} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = 1).$$

The Gini index is then:

$$G = \sum_{i=0}^{1} (p_{mi}(1 - p_{mi})) = 2p_{m1}(1 - p_{m1}).$$

This Gini index gives us a metric of how good each split is. A lower Gini-index means that region $R_m$ contains a higher proportion of one class, which

means we have managed to find a criterion which splits the classes apart (which essentially is the end goal). The Cart algorithm is one way of optimizing with regards to this gini-factor. This algorithm splits the data into two subsets by doing a split on a single feature $k$ and a threshold $t_k$ [11, s. The CART algorithm for Classification]. This is done by searching for the $(k, t_k)$ which minimizes the cost:

$$C(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

[11, s. Classification tree, how to split nodes], where $m_{left}$ is the amount of observations in the left subregion $m_{right}$ is the amount of observations in the right subregion, $m$ is the amount of observations in the region we are doing the split on, and $G_{left}$ and $G_{right}$ is the Gini in left and right subregion. A lower value of $C(k, t_k)$ essentially means that the overall split results in regions more pure (i.e. less mixing of classes). Finding this exact $(k, t_k)$ may involve searching through all the features and then for each feature find the $t_k$ by searching through a lot of thresholds and then choosing the optimal pair $(k, t_k)$ which gives the lowest value for $C(k, t_k)$.

### 3.4.2 Gradient Boosting

While normal classification trees will be able to fit the training data well, they often will lead to overfitted trees if we fit the training-data too well, and many of the techniques to combat overfitting will often lead to trees not giving that good performance on new data anyway. However, they will often be able to pick up some underlying patterns in the data, more so than just making random predictions anyway. This enables its use as a weak learner for a boosting setting. In gradient boosting we combine many weak learners, adding them together to create a good learner. For gradient boosting, assume we have a cost-function function $C(f) = \sum_{i=0}^{n-1} L(y_i, f(x_i))$, where $f$ is our classifier. We then have the following algorithm [11, s. Gradient Boosting, algorithm].

- Initialize our estimate $f_0(\boldsymbol{x})$ (this will be our classification tree)
- for $m = 1, 2, \ldots, M$ do:
  - Compute the negative gradient vector $\boldsymbol{u}_m = -\frac{\partial C(\boldsymbol{y}, f)}{\partial f(\boldsymbol{x})}$ at $f(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x})$.
  - Fit the weak learner (again is this is a classification tree) to this negative gradient. Denote this as $h_m(\boldsymbol{u}_m, \boldsymbol{x})$.
  - Update the estimate $f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + h_m(\boldsymbol{u}_m, \boldsymbol{x})$.

We then will get a final classifier $f_M$. For our loss-function in a binary classification case we could for example use:

$$C(\boldsymbol{y}, f) = \sum_{i=0}^{n-1} \exp(-y_i(f_{m-1}(x_i) + \beta G(x_i))) = \sum_{i=0}^{n-1} w_i^m \exp(-y_i \beta G(x_i))$$

[11, s. Adaptive Boosting, AdaBoost], where $y_i$ is $-1$ if we do not have fraud and 1 if we do have fraud, $\beta$ is the weighted parameter which we want to optimize and $G$ is our weak classifier. The XGBoost (Extreme Gradient Boosting) python-library, builds upon the idea of gradient boosting and makes optimizations on top of this in many ways. Often gradient-boosted trees are very good for classification purposes. The two hyperparameters we will optimize here is going to be the amount of estimators and the maximum amount of allowed leaves. Here the amount of estimators is essentially the $M$ in the algorithm above, while the max leaves will be the maximum amount of leaves or regions we will allow for each of the weak classifiers.

# 4 Results

## 4.1 Logistic regression

In **lr_creditcard.py** in the project codes [12] we have tested three different models, two of whom are fitted with the ordinary features, with $l_1$ and $l_2$ regularization, and then a logistic regression-model with polynomial features. Table 1 shows the results for the three different models. Here we can see that the choice of regularization type does not give any meaningful difference in performance, but that the model with second degree polynomial features performs a lot better, with this getting 99.7% accuracy compared with 96.5% accuracies for the non-polynomial feature models. Figure 3 and 6 shows the confusion matrices for the models without polynomial features and with polynomial features respectively. Again here we see that the polynomial feature model gives better predictions throughout. For this model we get that the false-positive rate (fpr), which in this case means the rate of non-fraud cases classified as fraud, is $\frac{245}{70907+245} \approx 0.34\%$, while the false-negative rate (fnr), which in this case means the rate of fraud cases classified as non-fraud, is $\frac{140}{70866+140} \approx 0.20\%$. Furthermore figure 4 shows a ROC-curve for the models not using polynomial features. These ROC-curves look reasonably good, so we could adjust the classification threshold to minimize the false positives without getting too many false negatives and vice versa. If we look at the ROC-curve for the polynomial model, as shown in figure 5, we see that this is near perfect, which allows for the same possibility of adjusting the classification threshold to lower the false positives, for this model with even less of a hit in introducing false negatives and vice versa. The optimal regularization-parameter for the final model is $10^{-7}$.

## 4.2 Neural networks

The **nn_creditcard.py** program explores fitting neural networks to the creditcard data. Table 2 shows the accuracy on the validation-data when trained over some smaller training-dataset using one epoch. Here we see that the different methods give essentially the same results. The optimization method here therefore doesn't seem to matter all that much for the performance on

| Model | Test-accuracy |
|---|---|
| Plain features, $l_1$-regularization | 0.9653 |
| Plain features, $l_2$-regularization | 0.9654 |
| Polynomial-features, $l_2$-regularization | 0.9973 |

Table 1: The accuracy scores of the three different logistic regression models we have fitted, evaluated on the test-data.

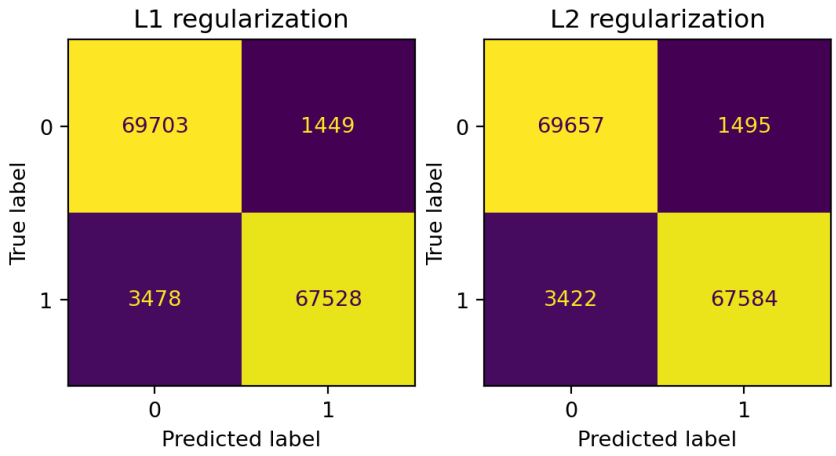


Figure 3: Shows the confusion matrices for a logistic-regression with $l_1$ and $l_2$ regularization, without using polynomial features, and with the regularization-parameter determined through cross-validation.

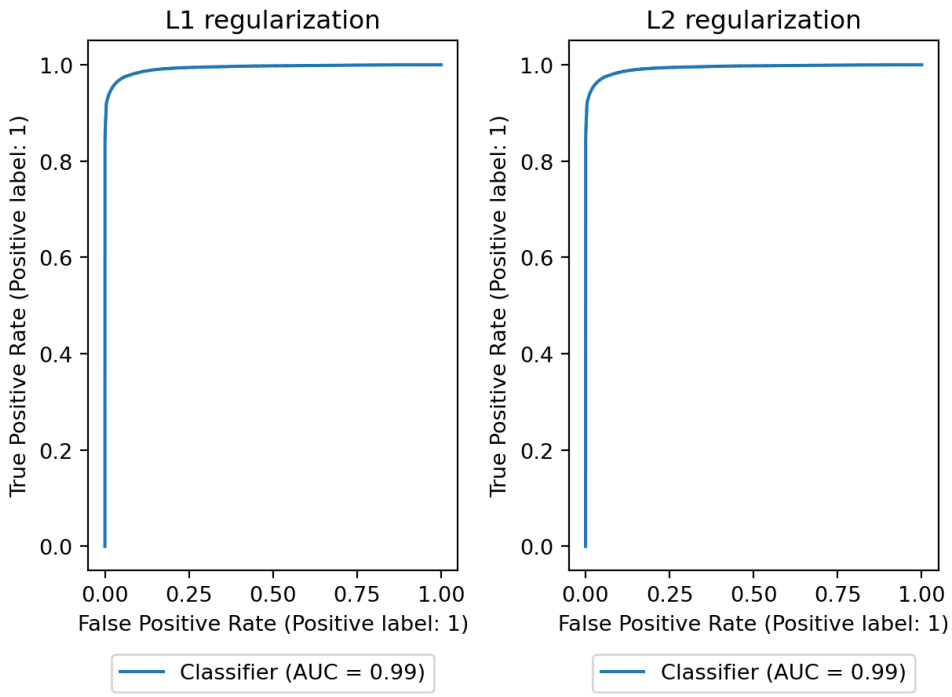


Figure 4: Shows the ROC-curve for a logistic-regression with $l_1$ and $l_2$ regularization, without using polynomial features, and with the regularization-parameter determined through cross-validation.

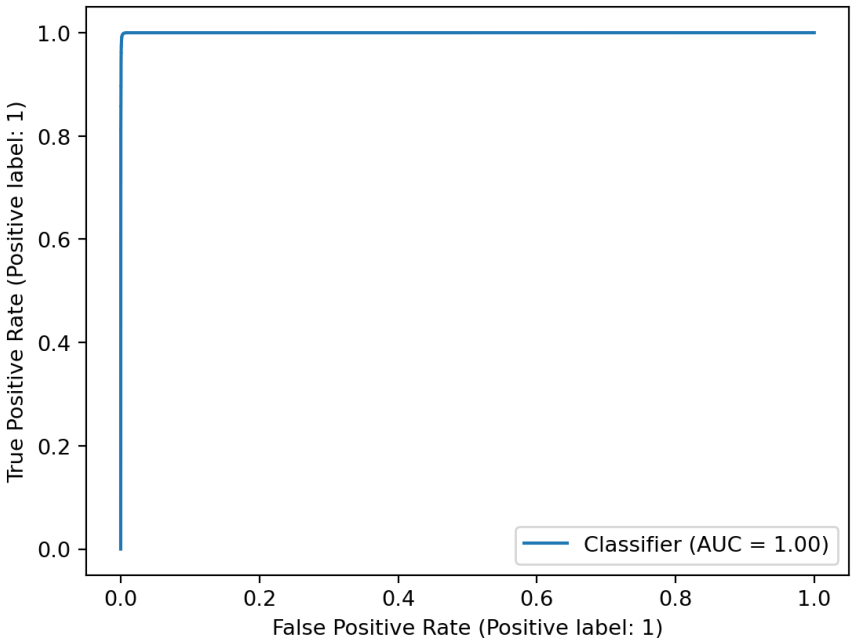


Figure 5: Shows the ROC-curve for a logistic-regression with polynomial features, and with the regularization-parameter determined through cross-validation. As we can see we get a AUC-score of 1.00, which is more or less perfect.

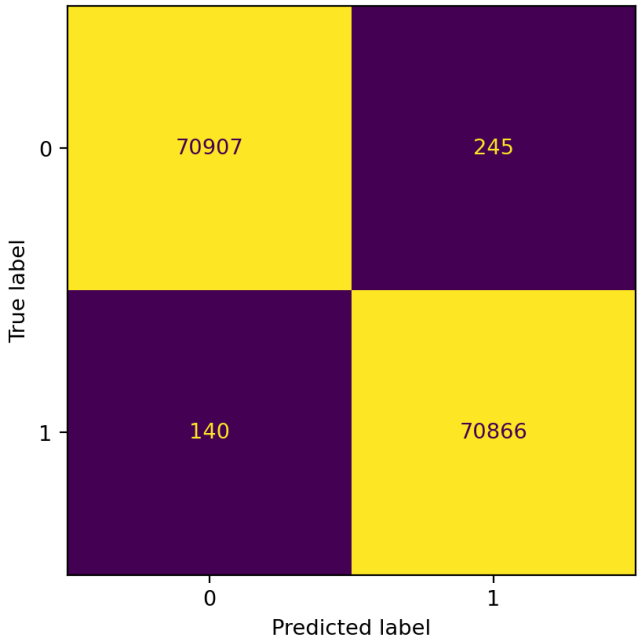


Figure 6: Shows the confusion matrix for a logistic-regression with $l_2$ regularization, with the regularization-parameter determined through cross-validation, and using polynomial features of degree 2.

| Optimization method | Validation-accuracy |
|---|---|
| AdaGrad | 0.9810 |
| Adam | 0.9802 |
| RMSProp | 0.9791 |
| SGD | 0.9779 |

Table 2: The validation-accuracies after training various neural networks using 10000 datapoints trained on 1 epoch, where we for each method have optimized the learning-rate and regularization parameters.

new data. Which optimizer we choose will therefore not matter too much. To see more about how the accuracy varies depending on the learning-rate and regularization-parameters, see figure 7. Going forward, we have picked RMSProp as our method of choice. Looking at the results this may seem odd as this did not give the absolute highest accuracy, but keep in mind that tensorflow is not built with complete determinism in mind, so the results will vary from each time you run the program, and on another occasion this performed the best. We also see from the heatmaps that overall the RMSProp and Adam seemed to give better performance for a wider variety of learning-rates and regularization parameters.

After choosing the optimizer we explored the different activation functions. Table 3 shows the accuracy on the validation-data for the different models. Here we see a more substantial difference, where relu did seem to perform the best by a little margin. Our model then becomes a RMSProp with relu activation functions, with the optimal regularization parameter occurring at $\lambda = 10^{-6}$ and the best learning-rate being $\eta = 10^{-\frac{5}{2}}$. Having found this we also try some different sizes of neural networks. Table 4 shows the accuracy on the validation-data for the different sizes of hidden layers. Here we see that some of the bigger networks do seem to give better results. However these models will be a lot more computationally expensive to train, which severely would limit the amount of epochs we would be willing to give it. Therefore for the final model we keep using the model with 2 hidden layers each of 100 neurons, as this also seemed to give quite good results. Figure 8 shows the confusion matrix for the final model. Here we see that we get 75 false positives and 195 false negatives, which overall is a bit better than the logistic model, but the amount of false negatives actually went up, which is also not ideal. The fpr then becomes $\frac{75}{71077+75} \approx 0.11\%$ while the fnr becomes $\frac{195}{70811+195} \approx 0.27\%$.

## 4.3 Gradient boosting

The last of our three methods was one based on gradient boosting using the XGBoost python-library. The code for our analysis is contained in **xgb_creditcard.py**. Here we only tweaked the number of estimators and the number of leaves. After optimizing these on a smaller subset of our training and validation data we get the amount of estimators giving the best results being 240 (giving an

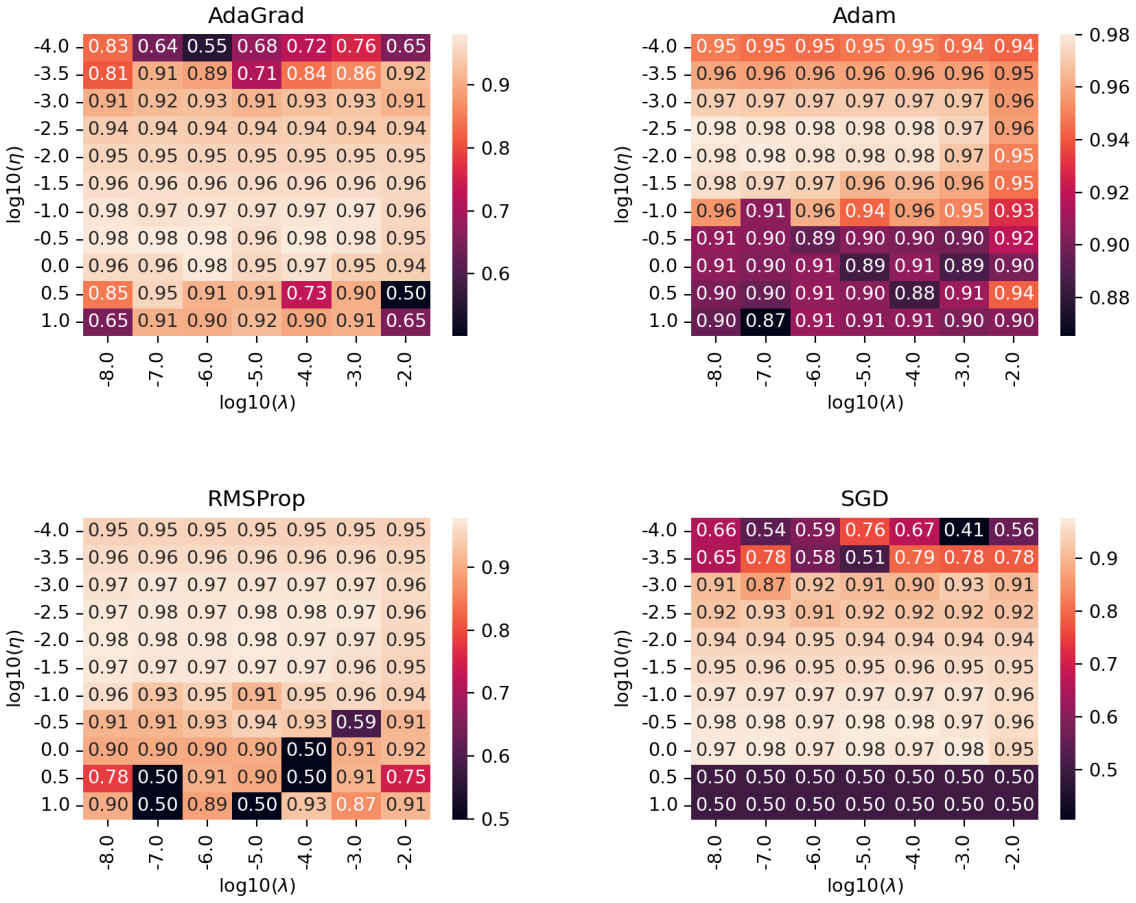


Figure 7: Shows heatmaps for the validation-accuracies for the different models, detailing how the performance varies depending on the learning-rate and regularization parameter.

| Activation function | Validation-accuracy |
|---|---|
| relu | 0.9789 |
| sigmoid | 0.9656 |
| swish | 0.9772 |
| tanh | 0.9685 |
| elu | 0.9731 |

Table 3: The validation-accuracies after training various neural networks on 10000 datapoints using 1 epoch, where we for each activation function have optimized the learning-rate and regularization parameter.

| Hidden layer sizes | Validation-accuracy |
|---|---|
| (10) | 0.9605 |
| (10, 10, 10) | 0.9649 |
| (100, 100) | 0.9781 |
| (50, 50) | 0.9749 |
| (100, 100, 100, 100) | 0.9820 |
| (10, 10, 10, 10) | 0.9676 |
| (100, 100, 100, 100, 100) | 0.9812 |
| (1000, 100, 10) | 0.9814 |
| (1000, 1000) | 0.9815 |
| (1000, 1000, 1000) | 0.9820 |
| (40, 100, 30) | 0.9789 |

Table 4: The test-accuracies after training various neural networks on 10000 datapoints using 1 epoch, where we for each layer size have optimized the regularization parameter.

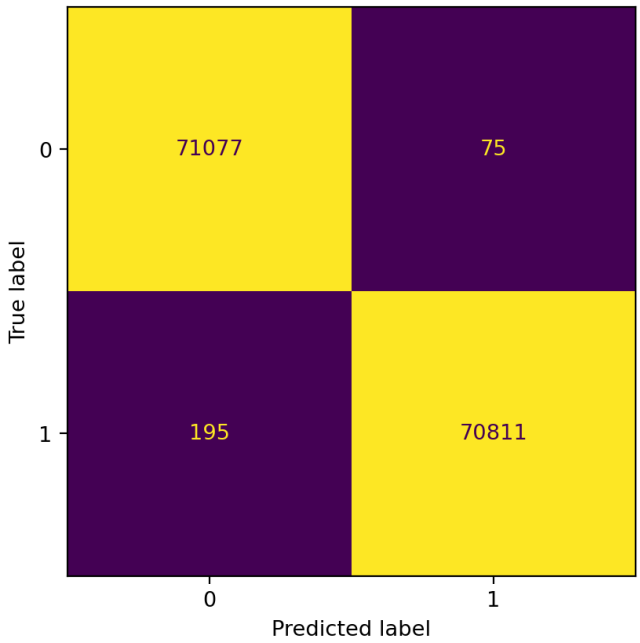


Figure 8: Shows the final confusion matrix of our chosen neural network trained on all the training-data.

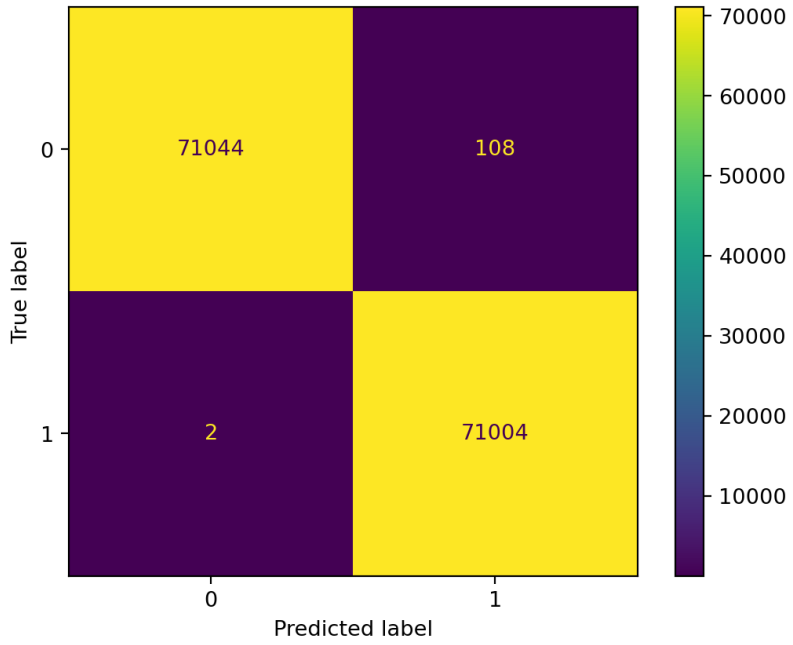


Figure 9: Shows the confusion matrix for a XGBoost gradient boosting model with 240 estimators and 7 as the max-leaves.

accuracy of 0.9912 with the default max leaves), and the best value of max-leaves being 7 (yielding an accuracy of 0.9923 with the 240 estimators). Figure 9 shows the confusion-matrix of the final model, trained on the full train-data and evaluated on the test-data. We see that we are able to classify all but 2 of the fraud-cases as fraud, while this time getting 108 false positives. This gives a fpr of $\frac{108}{71044+108} \approx 0.15\%$ and a fnr of $\frac{2}{71004+2} \approx 0.0028\%$. Figure 10 shows the importance-plot of each of the features. We can here see that a lot of the features which appears to have differing distributions for the fraud and non-fraud (looking at figure 1) cases appear to be some of the most important, which is not particularly surprising. Somewhat interestingly, *Amount* doesn't seem to be included in the model, which does check out with its distribution we saw from figure 1, but can still be a little surprising looking at the real-world context nonetheless. Finally for a visualization of one of the weak classifiers, see figure 11.

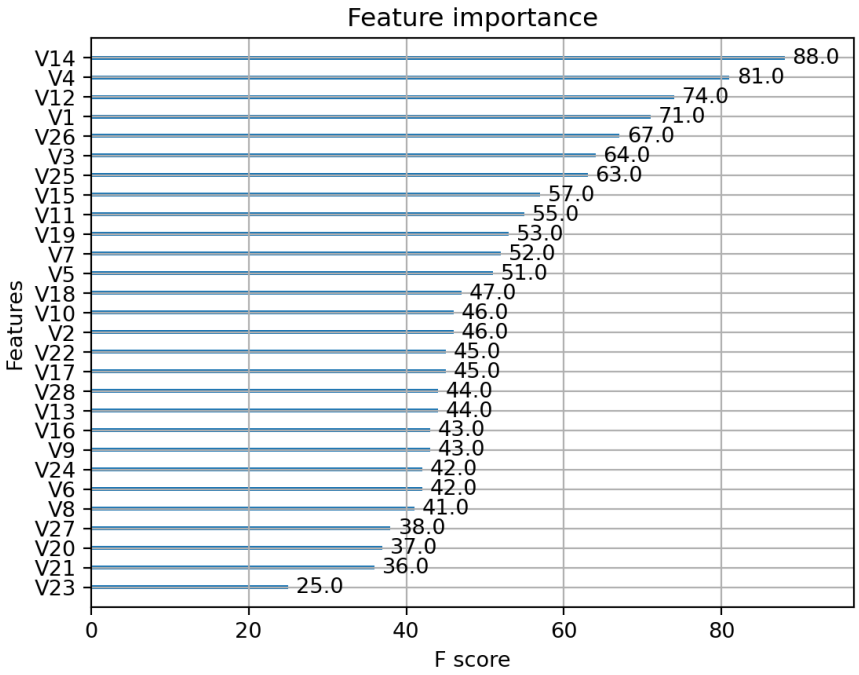


Figure 10: Shows the feature importance for our gradient boosting model.

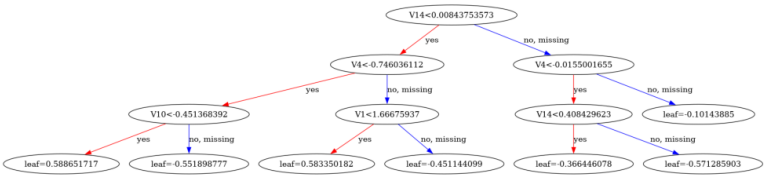


Figure 11: A tree plot for one of our weak classifiers.

| Model | Test-accuracy |
|---|---|
| Logistic regression (sklearn) | 0.9973 |
| Neural network (tensorflow) | 0.9981 |
| Gradient boosting (XGBoost) | 0.9992 |

Table 5: The accuracy scores on the test-data for our final models.

| Model | false-negative rate | false-positive rate |
|---|---|---|
| Logistic regression (sklearn) | 0.20% | 0.34% |
| Neural network (tensorflow) | 0.27% | 0.11% |
| Gradient boosting (XGBoost) | 0.15% | 0.0028 |

Table 6: The false-positive and false-negative rates gathered from on the test-data for our final models.

# 5 Analysis

After exploring the three different models and tweaking their respective hyperparameters we then, as already mentioned, trained the final models on all the training-data and evaluated the performance on all the test-data. Table 5 and 6 summarizes some key metrics from our final models. As we can see from this and other results already presented, the gradient boosted trees gave the best results for the overall accuracy. The gradient boosted tree was especially good at classifying the fraud-cases with 0.5 as the decision threshold, indicating that for classifying the fraud-cases especially, gradient boosting is considerably better than the two other methods.

One thing to keep in mind was that we have had to take some shortcuts here, especially when it came to neural networks, mostly due to the fact that neural networks can become very computationally expensive. We for example saw that some of the bigger neural network models seemed to perform better, so exploring bigger sized neural nets could change our results considerably, though such a model could take potentially hours to train properly even on powerful hardware, which is why I have chosen to not do this here. Another central shortcut we have used was when we determined the optimal hyperparamters. For selecting these, we trained a network allowing only one epoch, but this rests on the assumption that the methods doing the best after one epoch also does the best allowing for more epochs. This does not necessarily need to be the case, meaning that some of the optimal hyperparameters that were found need not to be optimal for the final model when we allow for more epochs. Here it would be better to allow for at least a couple more epochs so that the models get closer to convergence, this way the optimal hyperparameters may also be more reliable.

# 6 Conclusion

We see that the logistic regression model did come out the worst at $\approx 99.7\%$ accuracy, marginally beaten by the neural network at $\approx 99.8\%$, which again was marginally beaten by gradient boosting at $\approx 99.9\%$. Overall we can conclude that ensemble-methods like gradient boosting still seems to be the best classifiers for this data. For the gradient boosting we got the optimal model having 240 estimators and 7 as the maximum amount of leaves. For the neural network we saw that the choice of optimization-method did not greatly effect results, but that the choice of activation function having a greater effect, with the relu seemingly being the optimal choice here. Our optimal model became a RMSProp model with relu activation-function and 2 hidden layers of 100 nodes each, with $\lambda = 10^{-6}$ as regularization-term (using $l_2$ regularization) and a learning-rate of $\eta = 10^{-\frac{5}{2}}$. Do note though that for the optimal size of the network we saw some tendency that the bigger networks may perform better, but we had to cut some corners here due to limitations in computational power, hence our conclusions here does not necessarily hold for even bigger networks. Additionally we saw a large gain by adding polynomial features of second order to the logistic regression model. The optimal regularization-parameter for the logistic regression was $\lambda = 10^{-7}$, using $l_2$ regularization.

With this in mind some natural steps forwards would be to allow for considerably bigger neural nets, somewhere along the lines of 2 to 4 hidden layers each with 1000 nodes. It would also be interesting to do more advanced feature-engineering for the logistic regression model, as we could see that just adding polynomial features of second order did give us quite a boost in accuracy. We could for example explore higher order polynomial features or even non-polynomial transformations like for example log-transformations of features and so on. Perhaps we could also take use of some statistical techniques like forward or backward elimination. Both neural networks and logistic regression may therefore with the right features or size be able to equal, or maybe even outperform the gradient boosting method, but it is going to take more advanced models than a 2nd order polynomial feature logistic regression models or neural networks having 2 hidden layers with 100 nodes each.

# 7 Appendix

## 7.1 Project 2

This project in many ways build upon project 2. This can be found at: `https://github.com/magnouvean/ml-physics-projects/tree/main/project2/`

## 7.2 Project code (github)

All the python programs described in the report and with the full source code can be found at: `https://github.com/magnouvean/ml-physics-projects/tree/main/project3/python`

# References


[1] How major credit card networks protect customers against fraud. https://www.bankrate.com/finance/credit-cards/major-credit-card-networks-protect-against-fraud/, 2023.

[2] Project 2. https://github.com/magnouvean/ml-physics-projects/blob/main/project2/report/report_project2.pdf, 2023.

[3] Lucas Antoine. Credit card fraud detection - 100 https://www.kaggle.com/code/dreygaen/credit-card-fraud-detection-100-accuracy, 2023.

[4] Credit card fraud detection dataset 2023. https://www.kaggle.com/datasets/nelgiriyewithana/credit-card-fraud-detection-dataset-2023, 2023.

[5] sklearn.preprocessing.standardscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html, 2023.

[6] Wikipedia contributors. Hyperbolic functions — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Hyperbolic_functions&oldid=1185284795, 2023. [Online; accessed 1-December-2023].

[7] tf.keras.activations.swish. https://www.tensorflow.org/api_docs/python/tf/keras/activations/swish, 2023.

[8] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[9] tf.keras.activations.elu. https://www.tensorflow.org/api_docs/python/tf/keras/activations/elu, 2023.

[10] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2016.

[11] Week 46: Decision trees, ensemble methods and random forests. https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week46.html, 2023.

[12] Project 3. https://github.com/magnouvean/ml-physics-projects/tree/main/project3, 2023.