Classification and Regression: Neural Networks vs. Logistic/linear Regression

FYS-STK4155: Project 2

Trygve Leithe Svalheim
Kristian Joten Andersen
github.com/trygvels/FYS-STK4155

November 15, 2019

Abstract

In this project we look at classification an regression and compare methods for solving each problem. First we compare the classification ability of logistic regression vs. a feed forward neural network in determining probability of default for credit card clients. In the second part of the project we once again attempt to fit a polynomial to the Franke function, but this time through the application of a Neural network. We find that there are many issues with the credit card data, which has direct implications on our ability to do classification in a meaningful way. However, applying logistic regression, we manage to get an accuracy of 0.777 and an F_1-score of 0.783, which is improved upon by the neural network which gets an accuracy of 0.823 and a F_1-score of 0.801. Considering that the data set consists of 78.2% non-defaults, this accuracies are not much better than simply guessing 0 for all cases. For the polynomial regression problem we manage to tune the neural network to acquire an R^2-score of 0.947, which is slightly better than the simple ridge regression of our first project, which got an R^2-score of 0.943.

Contents

1 Introduction **2**

2 Data overview **2**
2.1 Credit Card data . . . 2
2.2 Franke Function . . . 4

3 Methods **4**
3.3 Logistic regression . . . 4
Gradient descent . . . 12
Stochastic Gradient descent . . . 13
3.4 Neural Networks . . . 14
The mathematical algorithm . . . 15
Activation functions . . . 16
The Softmax function . . . 17
Initialization . . . 17
Regularization . . . 17
3.5 Goodness of fit parameters . . . 18

4 Results and discussion **19**
4.6 Classification . . . 19
Logistic regression . . . 19
Neural Networks . . . 24
Optimization . . . 24
4.7 Regression . . . 26
Linear regression . . . 26
Neural Networks . . . 26
Optimization . . . 26

1. INTRODUCTION

In our technology based world with increasing gathering of data, especially online, there is a huge market for processing information. If an online marketplace is able to gather enough information on their costumers they might predict which products they are likely to buy and do targeted marketing of those products which may increase the revenue. In modern times it is more and more common to analyze data using what we call neural networks which in many cases are able to learn patterns themselves, with limited or no inputs from the users. Though this has become the popular thing to do there is no need to disregard "old", well tested methods and algorithms.

In this work we aim to see if the neural network algorithms are outperforming the old algorithms by testing them both in a classification problem versus logistic regression methods and on a terrain fitting problem versus a polynomial linear regression fit. In the classification problem we will work with information on credit card clients to predict whether or not they will default on their payments, and in the terrain fitting problem we will do a throwback to our first project by fitting a function known as the Franks function.

We start this report by presenting the different data set we have used in this work. Then we are going to present the different algorithms starting with classical logistic regression algorithms before we present the neural network algorithm. Lastly we will present the results from our analysis together with a discussion of the results.

2. DATA OVERVIEW

In this project we have worked with two different sets of data. The first is a data set containing information on credit card clients in Taiwan from April 2005 to September 2005. The data set comes from the UCI Machine Learning Repository Irvine, CA: University of California, School of Information and Computer Science [1] and has already been studied by Yeh, I. C. and Lien, C. H. [2]. The second data set is similar to the one we used in the first project, i.e. the Franke function [3]. We will now take a closer look at these two data sets.

2.1. Credit Card data

The data set of information on credit card clients from Taiwan [1] contain information on a total of 30000 clients from September 2005 going back to April 2005. The information gathered on the clients are the following: Credit limit, gender, education, marital status, age, payment history, history of bill statements, history of payment amount, and default payment in June 2005.

In order to avoid potential singularities, and to exclude data of little/no importance, we omit data from clients with no billing amount or payment amount history. Also, we take a look at the distribution of data regarding classification, e.g gender and education. Figure 1 shows the distribution of gender, marital status, highest education and age of the clients. In the distribution we see that some clients are missing some data values or have irregularities in the form of data values without descriptions [1, 2]. The number of client with irregularities are very small compared to the total number, thus we exclude these data points from our analysis as well. Furthermore, as the values of these classifications might not necessarily have a linear correlation with our target value, i.e. whether or not they default on the payment, it makes more sense to split them up in binary classes. This we do for education and marital status, as these are non-binary with uncertain linear correlation with our target value.

After excluding the data with irregularities as described above, and the data without payment/-billing amounts, we are left with 28121 clients. But, the highest number of irregularities/missing information we find in the payment history columns, shown in Fig. 2. The values '0' and '-2' are not described and make up most of the data set in these columns. Yeh, I. C. and Lien, C. H. [2] define the values of the payment history as "a measurement scale for the repayment status" with values as defined by: -1 = pay duly, 1 = payment delayed for one month, 2 = payment delayed for two months, ... 9 = payment delayed for nine months and above. This requires us to consider the data carefully. We can choose to exclude all client data containing either '0' or '-2' in any payment history data, thereby avoiding unknown data quantities, but this leaves us with a very reduced data set of 3883 clients.

Another possibility is to include additional data columns, like what is done when splitting a column

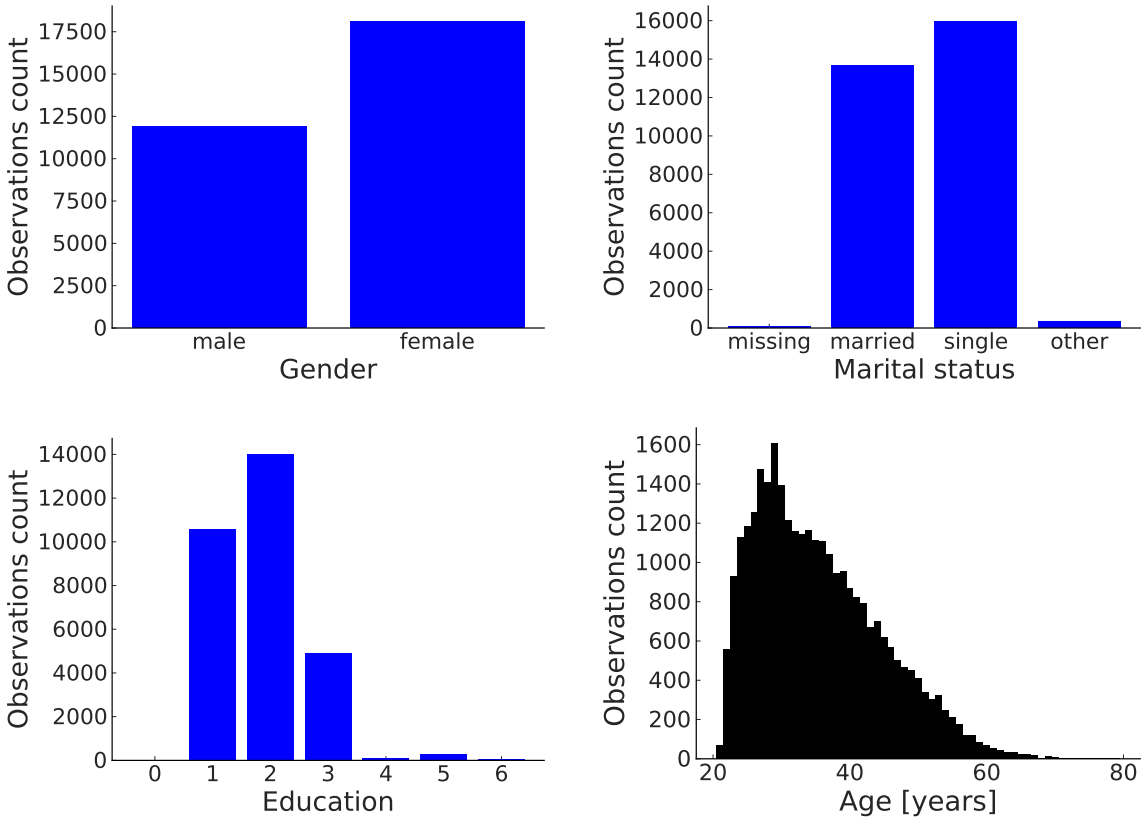

FIG. 1. Distribution of gender, marital status, highest education, and age of the complete data set. The x-axis indices of the education plot represents (1) graduate school, (2) university, (3) high school, and (4) others.

into binary classes/columns, but not changing the original columns. Instead we add binary columns for each payment history column corresponding to whether the values in that column are '0'/'-2' or not. This effectively adds 12 new columns (2 per payment history column) to our data set, which may prove costly with respect to computational time. The final setup of our data set is shown in Table I.

Before applying different data mining techniques on the data set it is preferable to take a look at the correlation between the different data columns. Calculating the Pearson correlation coefficient between each column in the (reduced) data set, the full correlation matrix is shown in Figs. 3 and 4 (5 and 6). From both the full and reduced data sets (see Figs. 3 and 5) it is clear that the heavily populated binary classes of education and marital status are heavily anti correlated, as is expected since no client can be positive (i.e. 1) in more than one column. Furthermore, we see that age and marital status are fairly correlated, where higher age favors the client to be married. Other significant correlated data are the groups of payment history and the billing amount, both internally and with each other (especially in the reduced set).

Additionally, in the full data set we have strong correlations internally in the flag groups of payment history, i.e. where the values in payment history are '0' or '-2' (pay_i_zero and pay_i_n2), but we also see that the two groups are anti-correlated, indicating that clients with '0' or '-2' in their payment history have a smaller probability of having the other value in their payment history than the average client. Another clear feature is the anti-correlation between the '-2' flag classes and the payment history, which is evident as a negative value in payment history (i.e. '-2') will give a positive value in the '-2' flag class.

Finally, for both the reduced and full data set, we see that the target value, i.e. whether or not the client defaulted on the payment in June, is mostly positively correlated with the payment history, indicating that if the client is behind on payment (higher value) the probability of defaulting on the payment. For the reduced data set we see this more clearly, where also the billing amount shows a fairly positive correlation. The most anti-correlated data is the credit limit, possibly indicating that higher credit is given to those who normally manage to pay back their debt.

2.2. Franke Function

The Franke function we know from the first project. It is defined on $x, y \in [0, 1]$ and is given as [3]

$$\begin{aligned} f_{\mathrm{F}}(x,y) = &\frac{3}{4}\exp\left\{-\frac{(9x-2)^2+(9y-2)^2}{4}\right\} \\ &+\frac{3}{4}\exp\left\{-\frac{(9x+1)^2}{49}-\frac{(9y+1)}{10}\right\} \\ &+\frac{1}{2}\exp\left\{-\frac{(9x-7)^2+(9y-3)^2}{4}\right\} \\ &-\frac{1}{5}\exp\left\{-(9x+4)^2-(9y-7)^2\right\}. \end{aligned} \tag{1}$$

In the first project [4, 5] we sampled this function on a $N \times N$ equidistant grid with values of $x, y \in [0, 1]$. We do the same here using a 20×20 grid and add Gaussian random noise with zero mean and a standard deviation of 0.05 to each sampled point on the Franke function. For information on how we apply this to a polynomial fit using linear regression, see the first project reports [4, 5]. How we use this for the neural network is explained in sections 3.4 and 4.7.

3. METHODS

3.3. Logistic regression

In classification problems, where the aim is to divide a set of data into two or more classes based on some independent variables $\hat{x_i}$ related to the data, it is not the same process as for linear regression where one aims to find a functional fit to (assumed) continuous variables y_i. This is because the values y_i take discrete values, corresponding to the class they belong to. Usually when one works with classification problems there are only two possible outcomes or classes, which is denoted as binary outcomes, true or false, positive or negative etc. The mathematical method of classifying data is called logistic regression, where we are interested in finding the probabilities of some data belonging to a given class.

In this project our goal is to use the data on the credit card clients described in Sec. 2.1 to model

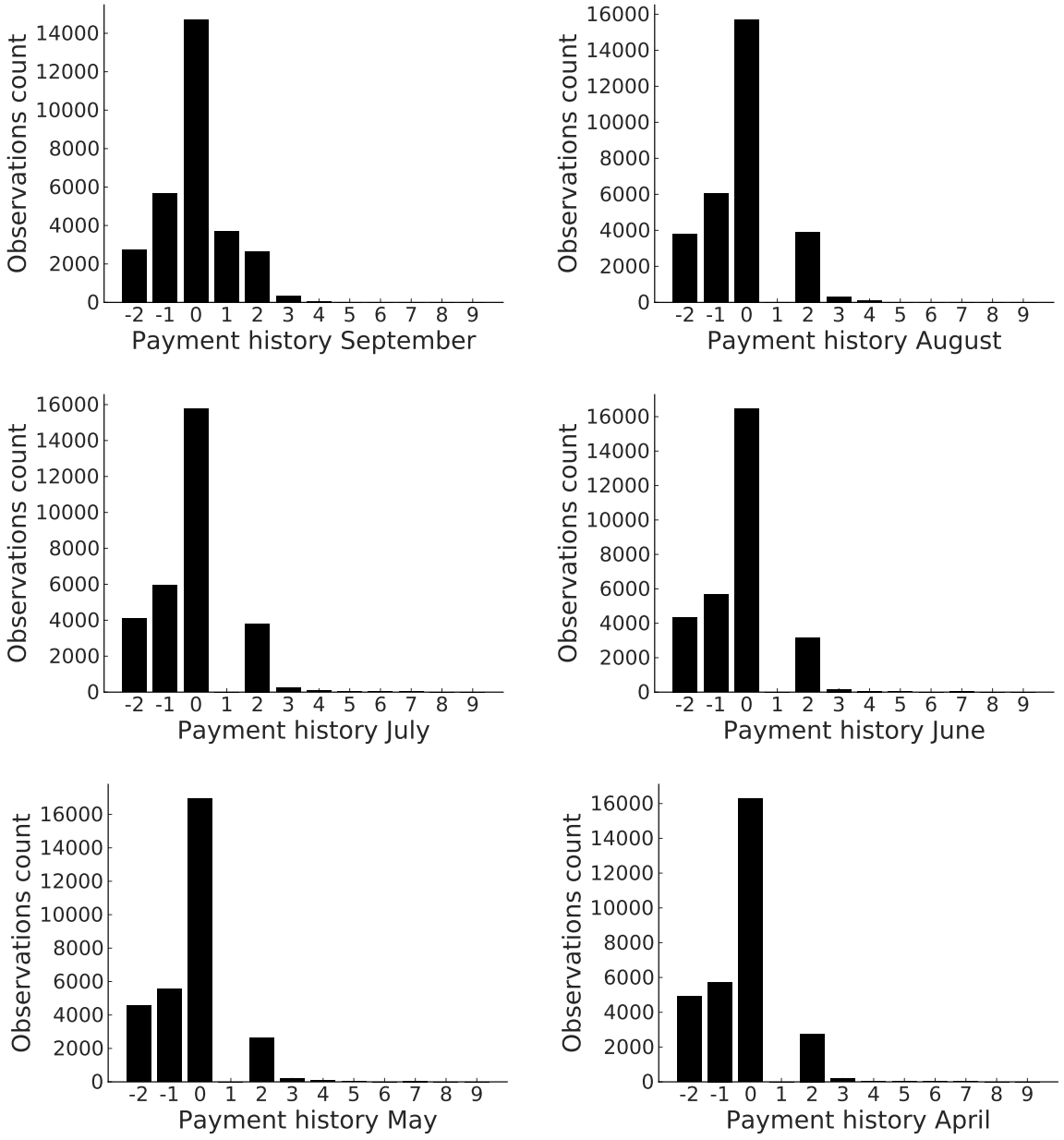

FIG. 2. Distribution of the payment history for the complete data set. We see that most of the data points take values that do not match any of the provided classifications from UCI [1] nor Yeh, I. C. and Lien, C. H. [2], i.e. the values '0' and '-2'.

TABLE I. Overview of the input data, their indexing and labeling, and their values.

Data type	Index	Specifics	Label	Value
Education	1	graduate school	edu_gs	{0, 1}
	2	university	edu_uni	{0, 1}
	3	high school	edu_hs	{0, 1}
	4	other	edu_o	{0, 1}
Marital status	5	married	marital_m	{0, 1}
	6	single	marital_s	{0, 1}
	7	other	marital_o	{0, 1}
Credit limit	8		credit	[0, ∞)
Gender	9	{male, female}	gender	{1, 2}
Age	10		age	[20, 80]
Payment history	11	September	pay_1	{-2, -1, 0, 1, 2, ... 9}
	12	August	pay_2	{-2, -1, 0, 1, 2, ... 9}
	13	July	pay_3	{-2, -1, 0, 1, 2, ... 9}
	14	June	pay_4	{-2, -1, 0, 1, 2, ... 9}
	15	May	pay_5	{-2, -1, 0, 1, 2, ... 9}
	16	April	pay_6	{-2, -1, 0, 1, 2, ... 9}
Bill statement amount	17	September	bill_amt1	[0, ∞)
	18	August	bill_amt2	[0, ∞)
	19	July	bill_amt3	[0, ∞)
	20	June	bill_amt4	[0, ∞)
	21	May	bill_amt5	[0, ∞)
	22	April	bill_amt6	[0, ∞)
Previous payment amount	23	September	pay_amt1	[0, ∞)
	24	August	pay_amt2	[0, ∞)
	25	July	pay_amt3	[0, ∞)
	26	June	pay_amt4	[0, ∞)
	27	May	pay_amt5	[0, ∞)
	28	April	pay_amt6	[0, ∞)
Payment history '0' flag	29	September	pay_1_zero	{0,1}
	30	August	pay_2_zero	{0,1}
	31	July	pay_3_zero	{0,1}
	32	June	pay_4_zero	{0,1}
	33	May	pay_5_zero	{0,1}
	34	April	pay_6_zero	{0,1}
Payment history '-2' flag	35	September	pay_1_n2	{0,1}
	36	August	pay_2_n2	{0,1}
	37	July	pay_3_n2	{0,1}
	38	June	pay_4_n2	{0,1}
	39	May	pay_5_n2	{0,1}
	40	April	pay_6_n2	{0,1}
Default payment in June	41 (29)	(If excluding irregular payment history data)	default	{0,1}

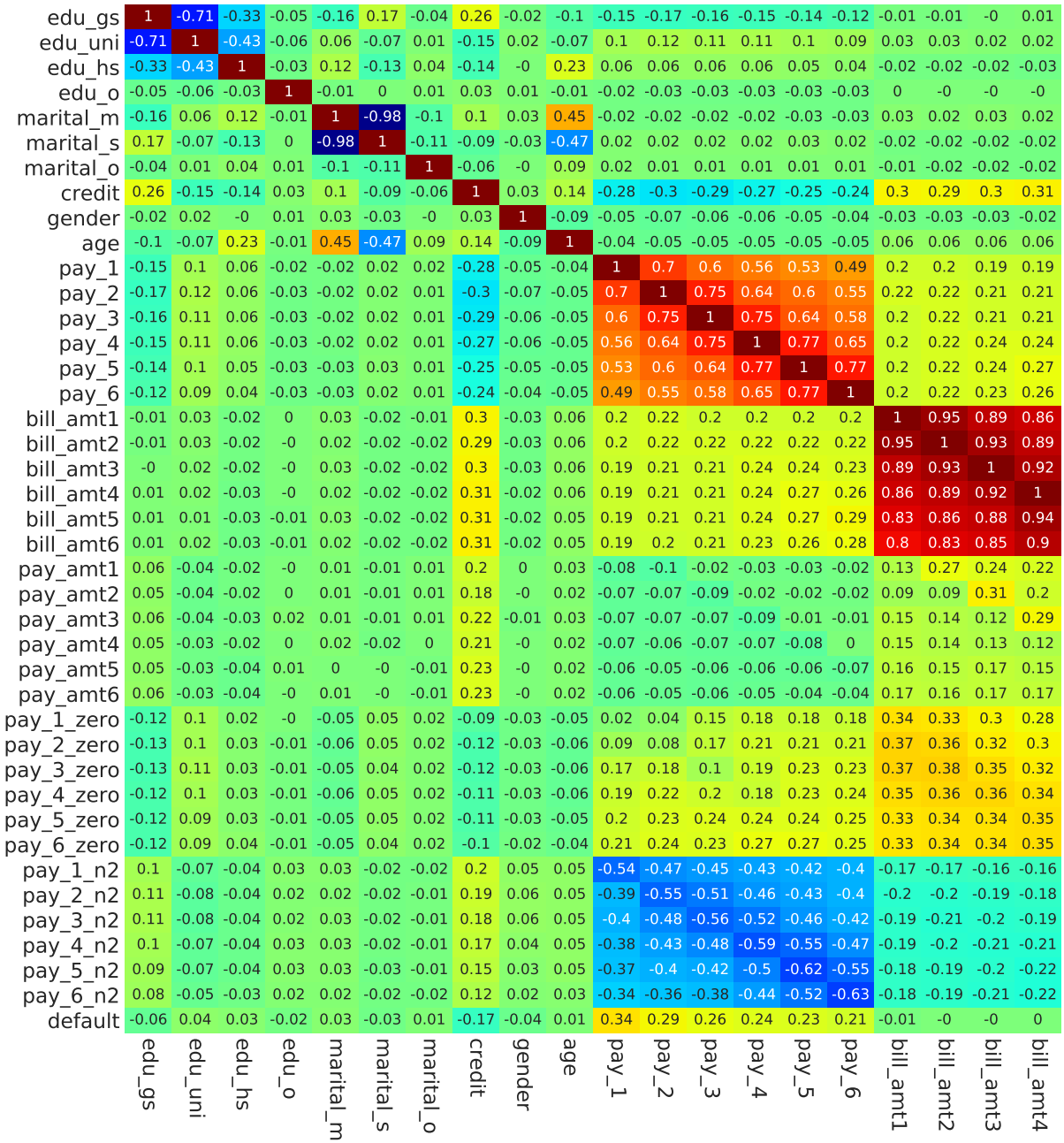

FIG. 3. First half of the correlation matrix of the full data set, the indices on the x- and y-axis are corresponding to the data types as shown in Table I.

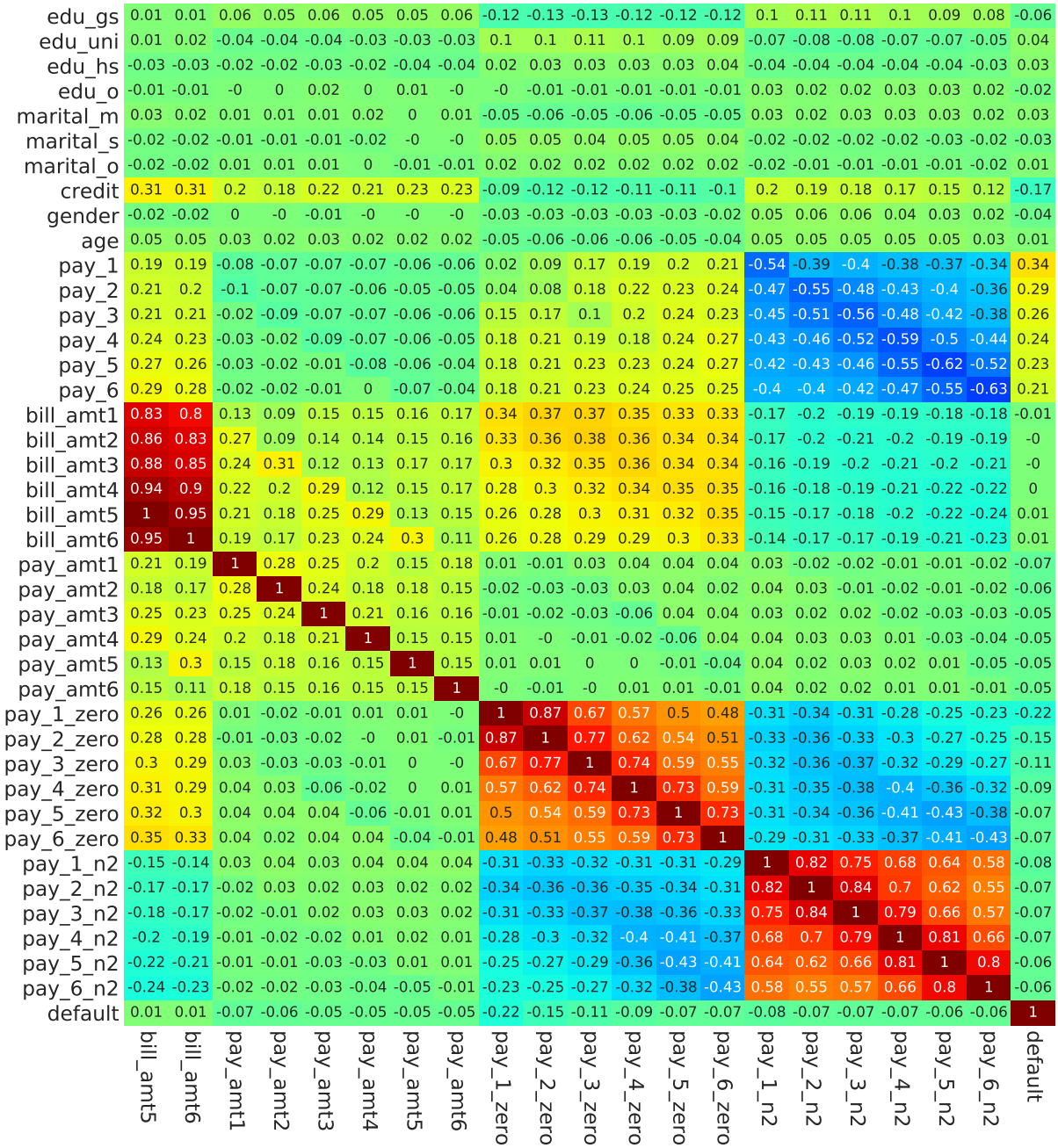

FIG. 4. Second half of the correlation matrix of the full data set, the indices on the x- and y-axis are corresponding to the data types as shown in Table I.

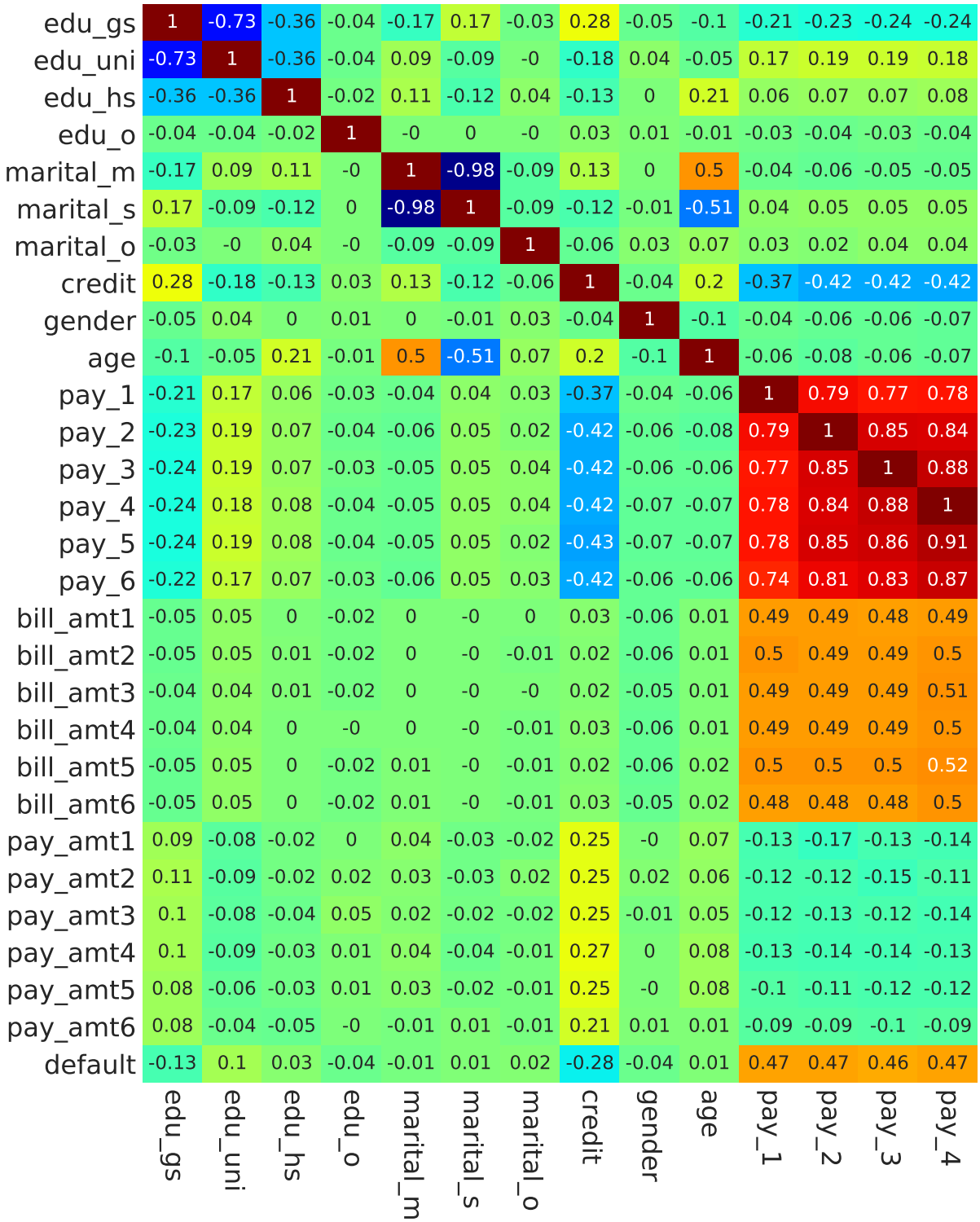

FIG. 5. First half of the correlation matrix of the reduced data set, the indices on the x- and y-axis are corresponding to the data types as shown in Table I.

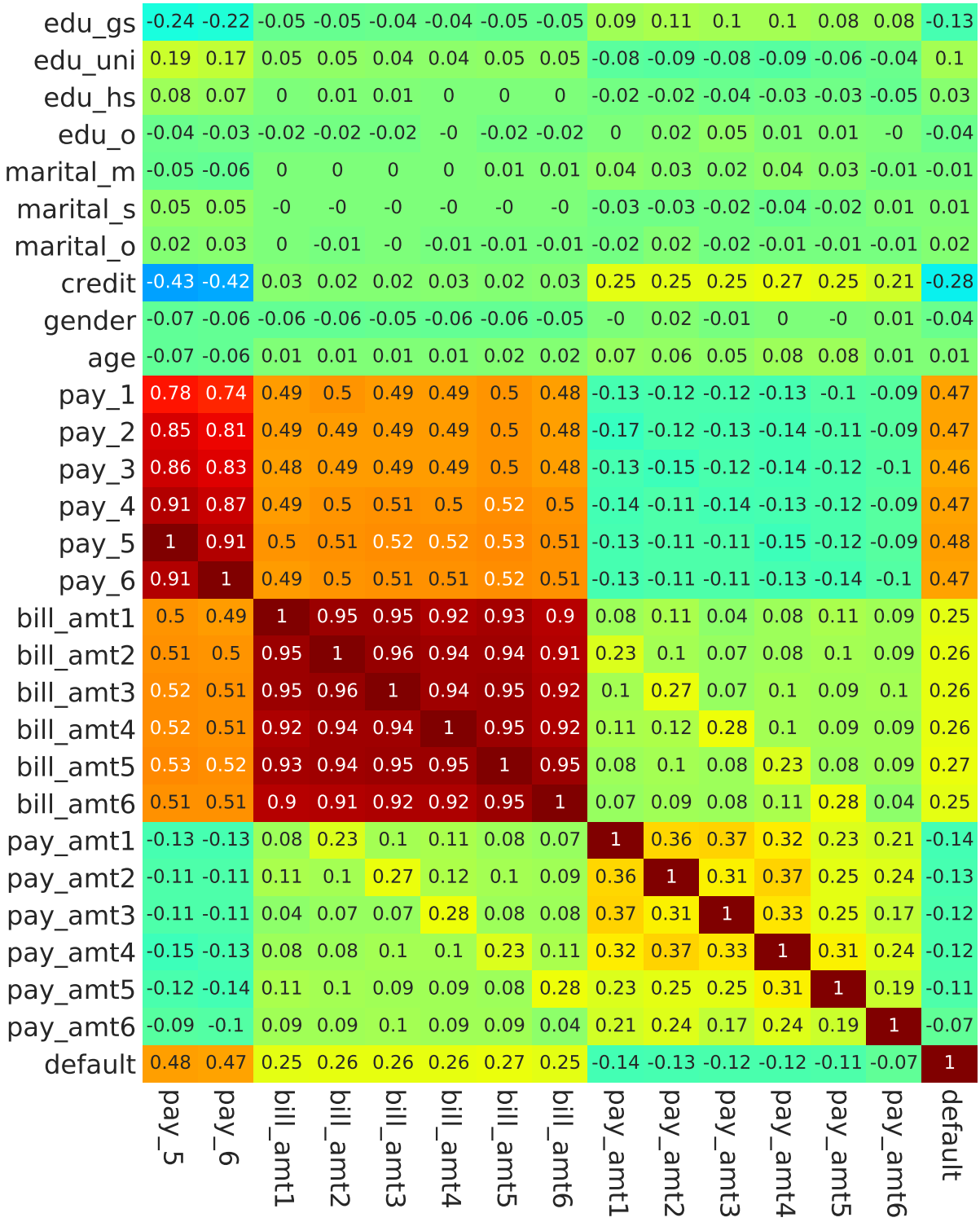

FIG. 6. Second half of the correlation matrix of the reduced data set, the indices on the x- and y-axis are corresponding to the data types as shown in Table I.

whether or not the clients default on their payment (positive outcome) or not. We have in other words a binary classification problem. All our independent data, i.e. the information on the clients, are put together in a data (or design) matrix $\hat{X}$ similar to what we do for linear regression, where x_{ij} is the data value of the j-th data type (e.g. age) to the i-th client (see Tab. I for all data types and possible values). If we were to use the same kind of estimation as in linear regression one would have

$$\hat{\tilde{y}} = \hat{X}^T\hat{\beta}, \tag{2}$$

where $\hat{\beta}$ is our fitting parameters, one for each column in $\hat{X}$, and $\hat{\tilde{y}}$ is the vector of predicted values. If $\hat{X}$ has the shape $n \times p$, n being the number of clients and p being the number of columns, then $\hat{\beta}$ has length p and $\hat{\tilde{y}}$ has length n. The problem here is that $\hat{\tilde{y}}$ doesn't take discrete values. Instead we can use the logit (or Sigmoid) function $p(t)$ defined [6]

$$p(t) = \frac{1}{1+\exp(-t)} = \frac{\exp(t)}{1+\exp(t)} \tag{3}$$

which has the property $p(-t) = 1 - p(t)$. We see that the function takes values between 0 and 1, and $p(0) = 0.5$. Inserting $\hat{X}^T\hat{\beta}$ for t we can use the logit function

$$p(y_i = 1|\hat{x}_i, \hat{\beta}) = p(\hat{x}_i \cdot \hat{\beta}), \tag{4}$$

where $\hat{x}_i$corresponds to the i-th row of $\hat{X}$, as a measure of the probability of the data belonging to the positive case (defaulting on payment). We immediately see that the probability of belonging to the negative case is then

$$\begin{aligned} p(y_i = 0|\hat{x}_i, \hat{\beta}) &= p(-\hat{x}_i \cdot \hat{\beta}) \\ &= 1 - p(\hat{x}_i \cdot \hat{\beta}) \\ &= 1 - p(y_i = 1|\hat{x}_i, \hat{\beta}). \end{aligned} \tag{5}$$

Let us denote our data set $\mathcal{D} = \{(y_i, \hat{x}_i)\}$, where as before y_i is the binary class ('default') and $\hat{x}_i$ is a vector of data on the i-th client. Using Eqs. (4) and (5) we can write the maximum likelihood function for the full data set, assuming the clients are independent, as a product of all the individual probabilities of a specific outcome y_i [6]

$$\begin{aligned} P(\mathcal{D}|\hat{\beta}) = \prod_{i=1}^{N} &\Big[p(y_i = 1|\hat{x}_i, \hat{\beta})\Big]^{y_i} \cdot \\ &\Big[1 - p(y_i = 1|\hat{x}_i, \hat{\beta})\Big]^{1-y_i}, \end{aligned} \tag{6}$$

where we see that the first bracket is the probability of finding $y_i = 1$ given that that is the case, while the second bracket is the probability of finding $y_i = 0$ if that is the case.

Our goal is to maximize the probability function, i.e. we follow the Maximum Likelihood Estimation principle in order to fit our model. To do this we take the partial derivative of Eq. (6) to find the (local) minimum and maximum points. As it is very hard to take the derivative of a large product like EQ. (6), we use the mathematical property that the logarithm of a function has maximum/minimum at the same values as the function itself, as the logarithm is a strictly growing and continuous function. Taking the logarithm we get a 'loss' function

$$\begin{aligned} \mathrm{C}(\hat{\beta}) = \sum_{i=1}^{N} &[\, y_i \log(p(y_i = 1|\hat{x}_i, \hat{\beta})) + \\ &(1 - y_i) \log(1 - p(y_i = 1|\hat{x}_i, \hat{\beta}))] \end{aligned} \tag{7}$$

Using Eq. (3) we may solve rewrite Eq. (7) to

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{N} [y_i\hat{x}_i \cdot \hat{\beta} - \log(1 + \exp(\hat{x}_i \cdot \hat{\beta}))]. \tag{8}$$

We would like to maximize this function, as the highest value indicates the maximum likelihood. Or, by defining our cost function as the negative of Eq. (8), our goal will be to minimize the following function for $\mathcal{C}(\hat{\beta})$, that we from now of refer to as the *cost function*

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{N} \Big[y_i\hat{x}_i \cdot \hat{\beta} - \log(1 + \exp(\hat{x}_i \cdot \hat{\beta}))\Big], \tag{9}$$

and is known in statistics as the *cross entropy*[6].

We see the similarities of the cost function with the one we used in the first project of this course [4, 5] where we used a version of the square error (see Sec. 3.5). Taking the partial derivative of Eq. (9) with respect to the j-th element in $\hat{\beta}$, we get

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_j} = -\sum_{i=1}^{N} \left(y_i x_{ij} - x_{ij}\frac{\exp(\hat{x}_i \cdot \hat{\beta})}{1 + \exp(\hat{x}_i \cdot \hat{\beta})}\right),$$

(10)

where x_{ij} is the data value of the j-th column in th i-th row of $\hat{X}$. If we were to define $\hat{y}$ as a vector of length N with elements y_i, and further define $\hat{p}$ as a vector of length N with elements of the fitted probabilities $p(y_i|\hat{x}_i, \hat{\beta}) = p(\hat{x}_i \cdot \hat{\beta})$ (right hand side (R.H.S.) using Eq. (3)) we may write Eq. (10)

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T(\hat{y} - \hat{p}). \tag{11}$$

Furthermore, by defining a diagonal matrix $\hat{W}$ of size $N \times N$ with elements $p(y_i 1\hat{x}_i, \hat{\beta})(1-p(y_i 1\hat{x}_i, \hat{\beta}))$ it may be shown (we will not do this here as we won't use this specifically) that the second partial derivative can be written [6]

$$\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta} \partial \hat{\beta}^T} = \hat{X}^T \hat{W} \hat{X}. \tag{12}$$

We also note that depending on the input data, the landscape of the cost function may have a mixture of very steep and flat directions. We therefore subtract the mean of each column of the design matrix in addition to normalizing the variance to unity. This way we reduce the differences in directions for the cost function, as well as eliminating the need of having an intercept term in $\hat{\beta}$. Lastly, this also makes the inclusion of a regularization term, like is done for Ridge/Lasso regression in our first project[4, 5]. A Ridge regularization term will add a $\lambda\hat{\beta}^2$ in the cost function, see Eq. (9), and $2\lambda\hat{\beta}$ term to the gradient, see Eq. (11).

Gradient descent

Now that we know how to compute the gradient of the cost function we need to use some method of finding the values of $\hat{\beta}$ that optimize the cost function. Let us first look at a simpler case. Assume x to be the optimizing parameter (like $\hat{\beta}$) and $f(x)$ to be the derivative of a cost function. By Taylor expanding around $f(s) = 0$ we get

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + ... \tag{13}$$

For small enough values and well-behaved functions, all terms beyond linear are not important, i.e.

$$f(x) + (s-x)f'(x) \approx 0, \tag{14}$$

so that

$$s = x - \frac{f(x)}{f'(x)} \tag{15}$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \tag{16}$$

where the last equation gives a possible algorithm for an iterative solution of s where k is the iteration number. If we now equate this to our own problem we see that we may write

$$\hat{\beta}_{k+1} = \hat{\beta}_k - \frac{\frac{\partial \mathcal{C}(\hat{\beta}_k)}{\partial \hat{\beta}_k}}{\frac{\partial^2 \mathcal{C}(\hat{\beta}_k)}{\partial \hat{\beta}_k \partial \hat{\beta}_k^T}}, \tag{17}$$

where we need only to use Eqs. (11) and (12). This would be fine if the cost function and its derivatives is behaving well, but that is not necessarily the case. Additionally, only for $\hat{\beta}$-values where the second derivative is positive we would expect the cost function to have a minimum, as a negative value gives a maximum and the iteration above goes towards the one specified by the second derivative. Another problem with this solver is that it requires the algorithm to compute the second derivative for each iteration, which might take a long time for a large design matrix $\hat{X}$. To solve this we rather introduce a learning rate (LR) parameter, here represented by λ, so that

$$\hat{\beta}_{k+1} = \hat{\beta}_k - \lambda \frac{\partial \mathcal{C}(\hat{\beta}_k)}{\partial \hat{\beta}_k}. \tag{18}$$

What the learning rate effectively does is to assume a value of the second derivative of the cost function, and since the larger the value of the first derivative the longer it is away from zero, the LR is a measure of how long steps one takes in $\hat{\beta}$ given the value of the first derivative. A large value of λ will make the values of beta change faster, but it might also lead into the values overshooting the minimum/maximum and go even further away in the opposite direction. A smaller λ will reduce this risk, but then again increase the number of iterations needed in order to reach the minimum/maximum. One case to make for the value of λ is that

if it is always smaller in absolute value than the true second derivative, then one never overshoot the minimum/maximum, but rather creep slowly towards it.

Another important choice of the LR is that we set it to be positive. This way, we assume the second derivative to be positive which is a necessity at any minimum. IF we start out close to a maximum where the second derivative truly is negative, then our choice of LR makes sure that we move away from the maximum, rather than towards it.

The algorithm we have presented in Eq. (18) is known as Gradient Decent (GD), where one from an initial value of $\hat{\beta}$ computes the gradient of the cost function and moves towards the (local) minimum. If one have no prior knowledge it is wise to use random values in order to sample the cost function plane better, and to avoid eventual plateaus. The only question is: when to stop? A common solution which we utilize here is to use the computed value of the gradient for each step, and if the norm of the gradient (i.e. it's total length) is smaller than some given threshold, τ, then we exit the search and the last value of $\hat{\beta}$ becomes our estimate.

There are three problems when using this algorithm, first the choice of LR as mentioned will be critical. Too large and we might never converge on a minimum, too low and it might take many iterations. Secondly, if the tolerance is small then the search may never converge, or use many iterations to do so. Lastly, given a small enough LR so that the steps are not too large, the algorithm moves towards the closest minimum along the greatest gradient and thus we may end up being stuck at a local minimum and never reach the global minimum.

To solve these problems we have done the following. Firstly, we have introduced a maximum number of iterations in order to not run forever, with an error message if reached. If so then reconsidering the LR or the tolerance is in order, so that one may reach convergence in a timely manner. This solution solves more or less the first problem, but it doesn't solve the last one, namely that we may end up at a local and not global minimum. The best way to solve this, using the limitations of the algorithm, is to run the algorithm several times starting with different values of $\hat{\beta}$ for each search and choose the $\hat{\beta}$ from the run with the lowest overall cost.

The final weakness of this algorithm is the size of the data set that one tries to fit. The larger the data set, the longer each search will take. We attempt to solve this problem together with the local minimum problem with another algorithm, Stochastic Gradient Decent.

Stochastic Gradient descent

Stochastic Gradient Decent (SGD) differs from GD by how one estimates the gradient of the cost function given the current fitting parameters, i.e. $\hat{\beta}$. SGD is well suited when the gradient can be written as a sum over contributions from independent data points, e.g.

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_j} = \sum_{i=1}^{N} f(x_i|y_i, \beta_j, \hat{\beta}), \tag{19}$$

where f is a function giving the contribution from each data point x_i. In our case we see that the gradient of the cost function can be written as a sum of a function enacting on (assumed) independent data, see Eq. (10). We can then divide our independent data set, i.e. the credit card clients in our case, into m different groups or batches, with a group/batch size of $M = N/m$. If there are no huge correlations internally in each batch or between the different batches, then the gradient of the whole data set can be approximated by the gradient of one of the batches. If for each time the gradient is computed one picks one of the batches at random, e.g. batch B_k, then the gradient in Eq. (10) is approximated

$$\begin{aligned}\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_j} &= -\sum_{i=1}^{N}\left(y_i x_{ij} - x_{ij}\frac{\exp(\hat{x}_i\cdot\hat{\beta})}{1+\exp(\hat{x}_i\cdot\hat{\beta})}\right) \\ &\approx -\sum_{i\in B_k}\left(y_i x_{ij} - x_{ij}\frac{\exp(\hat{x}_i\cdot\hat{\beta})}{1+\exp(\hat{x}_i\cdot\hat{\beta})}\right).\end{aligned} \tag{20}$$

There are two possible ways of creating batches. Either one create the batches before the search or one can more or less create the batches as one iterates by drawing M random data points. It is not recommended to only draw a single number and use the M following data points as the data might be ordered in some way so that the batch is not representative of the full data set. We have decided on

using fixed batches in our algorithm, as it is easier to implement and (slightly) faster to run.

The question with SGD like with GD is when to stop. One could use the same argument as for GD and stop when the gradient becomes smaller than a given value, but the aim of this SGD algorithm is not to find a local minimum, but rather the global minimum. In order to not getting trapped in a local minimum we utilize two of the parameters previously defined: LR and batch size. If the batch size is large, i.e. few batches, we don't gain very much speedup of the code. Additionally, the gradient will probably be much the same as for the full data set. By keeping the batch size relatively small, there is a greater probability that the gradient will change slightly between each step and thus not trapping the search at a local minimum. When it comes to the LR we can utilize another trick in order to "force" convergence. If we start out with a larger LR the $\hat{\beta}$-values will change fast and we take longer steps, then we slowly reduce the LR so that the steps are getting smaller, and the chance of stepping from one local area (e.g. a minimum) and into another due to a large LR gets increasingly smaller.

The timescale of the reduction we classify as epochs. In each epoch the gradient is computed and a step is taken m times, equal to the number of batches. The LR is set to decrease as

$$\text{LR} = \lambda_0 \frac{kN}{(k+i)N + jM}, \tag{21}$$

where again N is the total number of data points, M is the batch size, i is the epoch number, j is the step number of epoch i, and k is a parameter to tune the speed at which LR decreases (higher value means slower reduction of LR). By default we set $k = 1$. Alternatively the denominator can be replaced by $kN + jM$ where j would be the total step count.

At the end of each epoch the cost function is computed and the value as well as $\hat{\beta}$ are stored. When a specified number of epochs have been completed the $\hat{\beta}$-values corresponding to the lowest cost are returned. Furthermore, if one iterates several times over a set number of epochs, starting where the last iteration ended, resetting the LR and slowly decreasing LR, one can sample the $\hat{\beta}$-values even better. If one still wants to sample further then all the steps can be performed a set number of times and the best solution chosen.

It is clear that though this method has its strengths, like it being less likely to be trapped in a local minimum, it also has its weaknesses. As the cost function (of the full data set) has to be computed several times, this is computationally expensive. One are also more likely to jump around in the parameter space of $\hat{\beta}$ more than for GD and therefore more likely to have a larger variance in the results.

3.4. Neural Networks

Neural networks is one of the most popular and relatively new techniques in machine learning. The name stems from the fact that the method is inspired by biology, and the inter-connectivity of neurons in biological systems, such as our brain. In this project, we implement a supervised learning method. This differs from an unsupervised learning method in the way that we know which categories to predict. In our case, that is default or not default. For unsupervised learning methods, we might want to run our network on an arbitrary data set to discover important features across the data that are not obvious to the human eye, such as K-clustering, which clusters the K nearest neighboring points together in one category. This is useful when we have no labeled data set. Which is one of the biggest drawbacks of supervised learning, where one needs a large set of labeled data points, like a million images labeled "cool person" and pictures of me.

In both cases the neural network consists of multiple inter-connected layers, each layer consisting of a range of *nodes*. The baseline structure of a neural network consists three categories of layers. The first layer is the *input layer*, which takes the same structure as the number of input features in a data set, or for example the number of pixels in an image. The second set of layers is an arbitrary number of *hidden layers*, which takes the output from the input layers response to a set of initially random *weight* and *bias* factors, and passes it through an *activation gate* at each node. An activation gate is a non-linear function that transforms the data in some suitable way, for example truncating it between values of 1 and -1. This activation gate is crucial in the neural network as it prevents the network

from becoming a simple linear mapping from input to output. Lastly, we pass on the output of the last hidden layer through the output layer with another set of weights and biases. The last step then applies another activation function, which transforms the data into the desired shape, which for a classification problem, might be a probability that the input data belongs in a certain category. The value of the prediction made is then measured using a carefully chosen *cost function*. Using calculated cost start the process of *backpropagation*, which calculates the impact each node had on the final result, and adjusts each weight and bias accordingly. Once we reach the input layer, we run the feed-forward network once again until we reach a desired result.

A simple analogy is that of a sound mixing console full of nobs used to tune the audio of some band that just recorded their album. First, one turns all the nobs to a random setting, then one listens to the output, and if it sounds bad, one goes back, and changes each nob relative to how bad that particular nob made the music sound. Because that's exactly how music works.

In recent years, Neural networks has taken many different forms. One example is AlexNet which has shown the benefits of deep learning, which uses many hidden layers. This can be especially useful when considering hierarchical data, where one can find smaller patterns within a pattern. An example of this is an image of a person, where one pattern is the person, but sub-patterns are the eyes and mouth etc.

Furthermore, there exists a plethora of different compositions of neural networks. One common alternative to the dense neural network described above is *convolutional neural networks*, which is commonly used in image analysis. Convolutional neural nets uses smaller filter kernels which it convolves the image with, the sum of the response of which is the hidden layers in the network. The advantage of this is that it conserves spatial information because it does not process the full image at each node because the response at each pixel in the next layer is only the sum of the number of pixels confined within the filter kernel. Therefore, one can chose the size of each filter to extract features of varying size.

However, these methods are not necessary for the data in question here. That is because there is no inherent hierarchy in the features of our data set. Our binary description of whether or not the credit card client is male or female can not be decomposed into more descriptive features. However, the intricate relationship between features can be extracted through training a dense neural network.

The mathematical algorithm

In mathematical terms, the previously described feed forward neural network algorithm can be explained with a few simple equations. As with any type of categorization and regression, one wants to use some algorithm F to transform a data set X into a prediction y through $F(X) = y$. For a neural network with an arbitrary number of hidden layers, we express the response at a layer l as

$$z_k^l = \sum_{j=1}^{n^{l-1}} w_{jk}^l a_j^{l-1} + b_k^l \tag{22}$$

$$a_k^l = g(z_k^l) \tag{23}$$

where w is the weight, a^{l-1} is the activation from the last layer (Which for first hidden layer is simply the input data), b is the bias and the g is the activation function.

After the data has passed through the last layer and activation gate, we score the prediction with a chosen cost function. For the **classification** problem, we chose a cross-entropy cost function described in equation (9), but substitute the prediction $\hat{x}_i \cdot \hat{\beta}$ with our predicted probability from the last layer a^L,

$$\mathcal{C}(W^L) \equiv -\sum_{i=1}^{n} a^L \log(y) + (1-y)\log(1-a^L) \tag{24}$$

where y is the true class of the data set, in our case default or not. Furthermore, we chose the activation of the last layer to be the *softmax* function, in order to truncate the last response into probability values between 0 and 1, more on this in the next section.

Now that we can measure how bad our fit is, we calculate the derivative of the cost function with

respect to each weight and bias.

$$\frac{\partial \mathcal{C}(W^L)}{\partial w_{jk}^L} = \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial w_{jk}^L}\right]$$
$$= \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L}\right]$$
$$= \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} g'_L(z_j^L) a_k^{L-1} \tag{25}$$
$$\tag{26}$$

in equation 26 we see the last term is simply the activation of the last hidden layer, the second term is the derivative of the activation function for the output layer, and the first term is derivative of the cost function with respect to the output, which in our binary classification using cross-entropy is simply:

$$\frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} = \frac{a^L - y}{a^L(a^L - 1)} \tag{27}$$

Which, when combined with the derivative of the softmax function, which can be shown to equate

$$g'_{\texttt{softmax}}(z^L) = a^L(a^L - 1) \tag{28}$$

hence, for the classification case, we simply get

$$\frac{\partial \mathcal{C}(W^L)}{\partial w_{jk}^L} = (a^L - y)a^{L-1} \tag{29}$$

Furthermore, as we attempt to tackle the regression problem with neural networks, namely applying our neural network to the Franke function linear regression fit, we want to change our cost function. For a regression problem like this, the L2 distance is the natural choice, hence we want to calculate the derivative of the MSE with respect to the last activation. Fortunately for us, this is also very simple:

$$\frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\left[\frac{1}{2}\sum_{i=1}^{N}(a_j^L - y_j)^2\right] \tag{30}$$
$$= a_j^L - y_j. \tag{31}$$

and considering the fact that we don't truncate our data using the softmax function, but rather pass along the response to the weights as $g(x) = x$ which leads us to the same conclusion as equation 29.

However, we are only halfway done. In order to propagate the error to every weight in every hidden layer (for an arbitrary number of hidden layers), we need to calculate the effect of the cost in any layer l. By applying the chain rule we can express the derivatives in terms of each response as

$$\frac{\partial \mathcal{C}(W^L)}{\partial z_j^l} = \frac{\partial \mathcal{C}(W^L)}{\partial z_j^{l+1}}\frac{\partial z_j^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}. \tag{32}$$

If we calculate these derivatives, we get

$$\frac{\partial \mathcal{C}(W^L)}{\partial z_j^l} = (W^{l+1} \cdot \frac{\partial \mathcal{C}(W^L)}{\partial z_j^{l+1}}) g'_l(z_j^l), \tag{33}$$

where the second term is what we have proven earlier, assuming that l is the last hidden layer. This is often expressed as

$$\delta^l = \delta^{l+1} W^{l+1} a'(z^l). \tag{34}$$

So to implement this, we write

$$W^l \leftarrow = W^l - \eta a^{l-1T}\delta^l, \tag{35}$$
$$b^l \leftarrow = b^l - \eta\delta^l, \tag{36}$$

where η is the learning rate. This is one of many ways of updating the weights and biases, other algorithms are mentioned in the optimization section. As a final note, we did not derive the cost with respect to the biases, this is much easier to calculate, as the derivative with respect to the activation is one, but the derivation technique is the same.

Activation functions

An important aspect of neural networks is activation functions. These functions are carefully chosen depending on the type of data in question. The activation function is what gives Neural networks complexity, as their non-linear nature prevents the network from becoming a linear mapping from input to output, or a polynomial of degree one. In principle, one can use any non-linear function that is continuous, bounded and non-constant. Another important property of the activation function is that it needs to be differentiable, if not, we would not be able to backpropagate.

The most common activation function, especially for classification problems, is the *sigmoid* function

$$g(x) = \frac{1}{1 + e^{-x}}, \tag{37}$$

which truncates the input between 0 and 1. Historically, this activation function has been very common, but after the advent of deep neural nets, a major flaw was discovered, namely the *vanishing gradient problem*. The vanishing gradient problem is the problem of gradients going to zero as a result of the partial derivatives truncated to small values being multiplied l times, which for a high number of hidden layers goes to zero. This means the first layers learn a lot slower, and in the worst case scenario, not at all. For our purposes, using only one hidden layer, this is not an issue. However, for deep networks, or recurrent neural networks, this is an important issue one faces, and can be dealt with using gated recurrent nets or exploring different activation functions.

Another related activation function is the hyperbolic tangent function or *tanh*.

$$f_{\mathrm{t}}(x) = \tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}}. \tag{38}$$

which is a scaling of a sigmoid, centering it at zero, so that the function truncates values between -1 and 1. It has a steeper derivative so it updates gradients faster, and because it is centered on zero it is easier to optimize. However, calculating exponentials are more computationally expensive, and it still has the problem of vanishing gradients.

On of the most popular activation functions these days is the rectified linear units (ReLU), takes the input and returns all values above zero,

$$f_{\mathrm{ReLU}}(x) = \max(0, x). \tag{39}$$

It does not have vanishing gradient problem and is very computationally cheap. But on the other hand, a neuron which goes to zero has little chance of recovery, which results in the *dying ReLU* problem. Additionally, because it has no upper bound, ReLU also has the issue of exploding gradients, crashing the weight update for high learning rates.

Another variation of the relu is the is the exponential linear unit

$$f_{\mathrm{ELU}}(x; \alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\mathrm{e}^{x} - 1) & \text{if } x < 0. \end{cases} \tag{40}$$

which has been designed to perform better on classification problems.

The Softmax function

The softmax activation function is a special case often used in perceptions and classifiers. Because it transforms the output of the previous layer into values between 0 and 1, making a prediction in the form of a probability value. Mathematically, it is defined as

$$g(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}. \tag{41}$$

and also has the convenient derivative mentioned in equation 28.

Initialization

In order to avoid any issues related to the downsides of a particular activation function, we need to carefully chose the way we initialize our biases and weights. For example, if one chooses to set all weights to zero, the derivative with respect to the cost function will be the same for all weights, causing the system to essentially kill the hidden layer, reducing it to a simple linear mapping. Furthermore, if the weights are initialized too high or too low in a sigmoid layer, the values will be truncated to 0 or 1 and learning will be a lot slower. For this project, we have chosen to use He intialization [7] for the ReLU layers, and Xavier initialization [8] for sigmoid and tanh. Both these techniques simply multiply a set of randomly initialized weights with a scaling factor.

All biases can be set to zero without worry.

Regularization

As we have experienced in Project 1, regularizing the weights is highly important in order to combat over fitting the data. If we don't do this, overly complex networks might fit to noise, making our network less generalizable. We add a regularization term $\lambda \cdot L2$ to the cost function in order to penalize large values for the weights, similar to what was done in project 1. The addition of regularization on the cost function is propagated all the way to

the weight and bias update, so that we get

$$\frac{\partial \mathcal{C}(W^L)}{\partial w_{jk}^l} = \delta^l a_j^{l-1} + \lambda w_{jk}^l \tag{42}$$

$$W^l \leftarrow W^l - \eta({a^{l-1}}^T \delta^l + \lambda w_{jk}^l). \tag{43}$$

3.5. Goodness of fit parameters

In order to test how well our algorithms preform, we need to introduce some goodness of fit parameters. For the classification algorithms we introduce the following three: Area ratio, F1-score and accuracy. For the linear regression like fit of the Franks function, eq. (1), we use the same parameters as for the first project, namely mean square error (MSE) and the R-squared score (R2).

Area ratio (AR) is defined from the Lift plot of the data set, see Fig. 7. According to the given model, one sorts the data from most likely to least likely to belong to the target class. Then one plots the number of targets in the first n number of data points, i.e. a cumulative function of the models predictive power. If the model is the best fit, all target data are more likely, given the model, to belong to the target class than any of the data not belonging to the target class. The cumulative function will then rise linearly from the origin to (N_t,N_t), N_t being the number of targets in the data set. If one were to sort the data at random, one would end up drawing a straight line (a baseline) from the origin to (N_d,N_t), as the number of targets in the firs n data points are proportional to the relative number of targets in the full data set (i.e. N_t/N_d). The area ratio is then defined as

$$\text{AR} = \frac{\text{Area between the model curve and the baseline}}{\text{Area between the best fit curve and the baseline}}. \tag{44}$$

We note that the area ratio can take values $\text{AR} \in [-1, 1]$, and that negative values means that the model does worse than classifying target/non-target at random. This can be transformed into the also used goodness-of-fit parameter area under curve (AUC) which takes the same principles as AR, but plots target on the y-axis and non-targets on the x-axis. The AUC is further defined as the relative ratio under the whole curve so that by transformation

$$\text{AR} = 2(\text{AUC} - 0.5), \tag{45}$$

$$\text{AUC} = 0.5 + \text{AR}/2. \tag{46}$$

$$\tag{47}$$

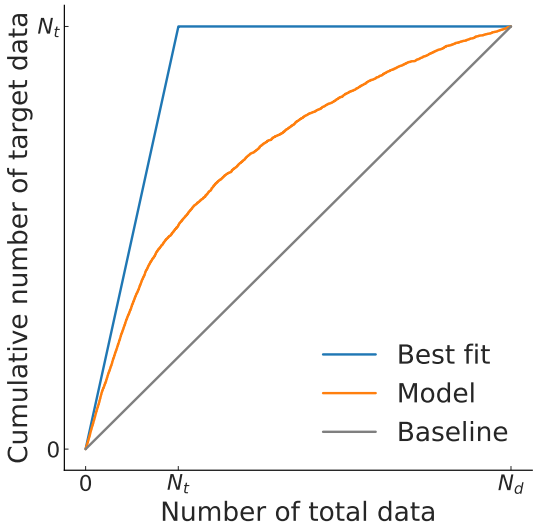

FIG. 7. Lift plot of a model to a data set. N_t is the number of targets in the data set and N_d is the total amount of data in the data set.

We notice that AR = 1 corresponds to AUC = 1, AR = 0 corresponds to AUC = 0.5, and AR = −1 corresponds to AUC = 0.

In a single binary classification problem we can define the data to either be *positive*, i.e. the target, or *negative*, while the predictions from our model we call *predicted positive* (PP) or *predicted negative* (PN) respectively. We can now divide the data into 4 groups: true positive = data that is both positive and predicted positive, true negative = data that is both negative and predicted negative, false positive = data that is both negative and predicted positive, and false negative = data that is both positive and predicted negative.

We now introduce two new terms, the true positive rate (TPR) and the positive predictive value (PPV), which are also known as *recall* and *precision* respectively. They are defined as

$$\text{TPR} = \frac{\sum \text{true positive}}{\sum \text{predicted positive}}, \tag{48}$$

$$\text{PPV} = \frac{\sum \text{true positive}}{\sum \text{positive}}, \tag{49}$$

and together they define the F_1-score for the *positive* target

$$F_1 = \frac{\text{TPR} \cdot \text{PPV}}{\text{TPR} + \text{PPV}}. \tag{50}$$

The *accuracy* (AC) of a model is defined as

$$\text{AC} = \frac{\sum \text{true positive} + \sum \text{true negative}}{\sum \text{total population}}, \quad (51)$$

which is the ratio of the total population the model correctly classifies.

All these parameters have their own strengths and weaknesses. The area ratio is a good measure if one wants to optimize the use of resources and only can act on parts of the data. In our case, if we were to follow up the clients, we would like to get as many clients who defaults on their payment as possible with the least amount of work. A high area ratio is therefore good for the model. However, a high area ratio doesn't necessarily mean that our model is accurate. If our model predicts all data to be target data, only that the true target data has slightly higher probability, we loose out on all the true negatives, giving a very low accuracy if the majority of the data is negative. This also affects the PPV and therefore also the F_1-score. As the accuracy looks at the whole data set, and the F_1-score looks at only one of the classes, the latter has more details. In our work, we would like our models to do well with both the positive and negative predictions, so a weighted average F_1-score for both the positive and negative (i.e. treat the negatives as positive and vice versa when computing the F_1-score for the negative data) with weights corresponding to the true number of each. Also, comparing the test and training data predictions will show us whether we over-fit or under-fit our model.

For the linear regression, or polynomial fitting using neural networks, we use the same two goodness-of-fit parameters as for our first project, namely the mean square error (MSE) and the R^2-value defined as

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2, \quad (52)$$

$$R^2 = 1 - \frac{\sum(y_i - \tilde{y}_i)^2}{\sum(y_i - \bar{y})^2}, \quad (53)$$

where y_i is the true value and $\tilde{y}_i$ is the predicted value of the i-th data point. $\bar{y}$ is the mean value of the true data. Assuming Gaussian noise with zero mean, reducing the MSE is equivalent to optimizing the likelihood of the fit. The nominator in the fraction of the R^2-score we recognize as the total square error between the model and the data, while the denominator is $N \cdot \text{Var}[y]$ or a sum of the variance of the true data. We see that if the MSE is much smaller than the internal variance in the true data then $R^2 \approx 1$, while if MSE $\approx$ Var$[y]$ gives a $R^2 \sim 0$. We see that $R^2 \in [1, -\infty)$, as there is no upper bound of the MSE, which we saw from the first project might be very large if we over-fit the model, i.e. sample the noise in the training data.

4. RESULTS AND DISCUSSION

4.6. Classification

We have presented the data on the credit card clients in Sec. 2.1 and the different algorithms for both logistic regression and neural networks in Sec. 3 used to classify data. We are now going to present the results from each algorithm, starting with classical logistic regression before looking into neural networks.

Logistic regression

In this section we present the results of the GD and SGD classification algorithms presented in sections and . As described in the referred sections, both algorithms require us to define a λ for the learning rate, which is proportional to the inverse second derivative of the cost function, see Eqs. (17) and (18). In the case for SGD the λ is the starting LR while for GD it is equal to the constant LR. This λ has to be tuned for each algorithm separately, and in the case of GD the most notable fitting restrictions are: (1) do we reach the tolerance level and (2) how long does it take?

Figure 8 shows a plot of the evolution of the computed cost for different values of λ for the GD algorithm on the reduced data set. Note that the values in the labels are not truly corresponding to the real λ-value. In order to accommodate different design matrices for the reduced and full data sets the actual value of the learning rate is LR $= \lambda/p$, where p is the number of columns in the design matrix $\hat{X}$. This is an attempt to reduce the step length if the gradient were to become large. As we see from the figure, the larger the LR the faster we converge. One could then think that we should choose $\lambda = 1$, but increasing it past that point results in

never reaching the tolerance, which has been set to $\tau = 0.01$. Also running over several searches shows that the standard deviation of the goodness-of-fit parameters stabilize at about $\lambda = 0.1$. We attempted to reduce the tolerance below 0.01 but it resulted in far too long runtime compared to the reduction in standard deviation, though further tuning could be done. Therefore, we chose the LR for GD was to be $\lambda = 0.1/p$ for further comparisons.

When it comes to the choice of the LR for the SGD algorithm, the combinations of the parameters are more finely tuned. If the LR is too large we might end up jumping around a lot. If the reduction of the learning rate is fast, and since we are only computing a set number of iterations, we might end up converging at a point far away from any minimum. Reducing the rate at which one reduce the LR helps avoid that, but then one would need more epochs in order to actually converge. Out initial thought was to use the same LR as for GD and use batch sizes of 100 and $k = 1$, see Eq, (21), but this ended up giving much larger deviations of the goodness-of-fit parameters than GD. Reducing the LR to 0.01 helped more, but the standard deviations was still one order of magnitude larger and the average about 1$sigma$ smaller. Plotting the evolution of the cost showed that the LR was reducing too fast, and the reduction parameter was changed to $k = 5$. Figure 9 show the evolution of the running computed costs at the end of each epoch for three LR values, spanning two consecutive iterations. One can see that a large LR takes you away from potential minimum values, while smaller values do not vary the cost as much (as we predicted). After running 50 searches of 10 iterations of 50 epochs, using these LRs and a reduction factor $k = 5$, we found that for the reduced data set the $\lambda_0 = 0.001$ gave the best and most consistent results.

When running the algorithms on the full data set we use the same parameters as for the reduced data set, with the exception to the number of epochs in SGD, which is set to 25 in order to reduce computation time. This was tested on the reduced set as well, and it showed only minor changes in the standard deviation of the fits. The algorithms performance on both the reduced and full data set can be seen in Tab. II which shows the mean and standard deviation of the goodness of fit parameters from 50 independent searches.

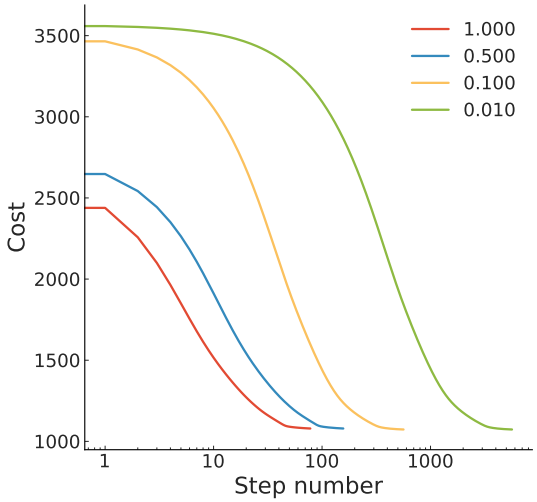

FIG. 8. Plot of the evolution of the computed cost when running GD on the reduced data set. The values in the labels correspond to the learning ratio parameter λ. The tolerance of these runs were 0.01.

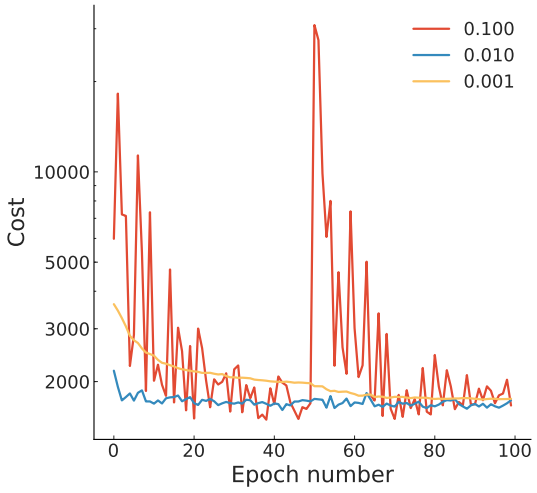

FIG. 9. Plot of the evolution of the computed cost 2 consecutive iterations of 50 epochs each when running SGD on the reduced data set. The values in the labels correspond to the maximum value of learning ratio parameter, i.e. λ_0. The SGD learning rate parameter k was set to 5, see Eq. (21).

The best values for any of the searches are given in Tab. III, though the values are not necessarily from the same search. Additionally, to see how well our algorithms perform compared to other logistic regression logarithms, we compare our results with the `LogisticRegression` algorithm of the `scikit-learn` python library. The `scikit-learn` algorithm only produced the same result if run several times, probably because it initiates on the same $\hat{\beta}$-values each time, so only that result is shown in the tables mentioned.

In table II we see that on average the GD algorithm performs slightly better than the SGD algorithm on the reduced data set, while on the full data set the SGD performs slightly better. In both cases the average value of the goodness-of-fit parameters are not matching the `scikit-learn` values, but in Tab. III we see that the best fit of both are rivaling the external algorithm, even surpassing it for the area ratio on the full data set for GD.

To additionally compare our results to the `scikit-learn` algorithm we plot the Lift plot, i.e. the cumulative prediction, of the solutions giving the highest area ratio. These can all be seen in Fig. 10. We also noticed there was greater correlation between the payment history and the target ('default' in Figs. 3-6) which might indicate that (at least some of) the data in the reduced set is easier to separate, thus giving a larger area ratio. We also notice a sharper bend in the model curve of the full data set and almost becoming linear. This would indicate that after a certain number of clients/data points we do no better than almost pure guessing. The similarities between the model curves of each data set is an indication that the algorithms are not very different for the same data sets.

Overall we see that our algorithms are slightly worse than the external one from `scikit-learn`, but with some more tuning of the parameters they might become better. Especially for SGD the biggest question is the batch size. We used 100 for both data sets, but 100 might be too small for the full data set with 14060 clients. Additionally, a change in batch size to a higher value reduces the number of steps per epoch (as our algorithm is set up) meaning we have to increase the number of epochs to sample enough of the $\hat{\beta}$-space. Increasing the number of epochs also require k to increase as well if one do not want to reduce LR too fast. One solution, that would make this more independent, would be to set the number of steps in each epoch to a fixed number.

Tables II and III show yet another feature, namely that while the area ratio of the full set is smaller than for the reduced data set, the other goodness-of-fit parameters are getting better for the full data set. This might be connected to the fact that the ratio between the number of targets (defaulting clients) and the total population is different between the two data sets. In the reduced data set we have 663 targets in a total of 1942 while in the full set we have 3063 targets out of 14061, giving a relative ratio of targets equal to 0.3414 and 0.2178 respectively. If we were to say that there are no targets in the data set, one would get an accuracy of 0.658 for the reduced data set and 0.782 for the full set. We see the accuracy is biased on the target-to-population ratio. Were we to compute the F_1-score we would get 0 for the target population, and for the non-target population we get 0.793 (reduced) and 0.878 (full). If we weight these with the true number in each group we get an average F_1-score of 0.524 (reduced) and 0.686 (full). We see that we are still getting higher values for the full data set, but the F_1-scores are lower than the accuracy. Therefore, we are more comfortable using the average F_1-score (i.e. the ones given in the tables) as a measurement of how well the algorithms perform than the accuracy.

One explanation of why the area ratio and the other goodness-of-fit parameters are changing differently is that the area ratio is not dependent on how many data points we classify correctly, it only depends on which data points we say are *most likely* to belong to the target population. If we were to say all or none of the data are targets, but the probabilities align so that the target data are still most likely to be targets, then we would still have an area ratio of 1.

The argument for which algorithm to choose, given that the algorithms otherwise perform similarly well, would be to compare the computation time needed by each algorithm. Given the values of the parameters in our algorithms, we see that the GD algorithm and the `scikit-learn` algorithm performs on the same order of computational time, while the SGD algorithm needs one order of magnitude more. If the algorithms perform equally on all other areas, then given our data set the GD al-

TABLE II. Overview of the average goodness-of-fit parameter values with standard deviation from each algorithm based on 50 searches on the reduced and full data set. The **scikit-learn** algorithm yielded only one result.

Algorithm	Area ratio (test data)	Accuracy (test data)	Accuracy (average)	F_1-score (test data)	F_1-score (average)
			Reduced data set		
GD	0.606 ± 0.005	0.764 ± 0.004	0.769 ± 0.004	0.765 ± 0.004	0.771 ± 0.003
SGD	0.603 ± 0.005	0.762 ± 0.003	0.766 ± 0.003	0.764 ± 0.003	0.769 ± 0.002
`scikit-learn`	0.620	0.772	0.779	0.769	0.777
			Full data set		
GD	0.517 ± 0.006	0.757 ± 0.006	0.759 ± 0.005	0.768 ± 0.004	0.770 ± 0.003
SGD	0.528 ± 0.005	0.760 ± 0.005	0.764 ± 0.004	0.772 ± 0.004	0.774 ± 0.003
`scikit-learn`	0.531	0.823	0.823	0.801	0.802

TABLE III. Overview of the best values of the goodness-of-fit parameters from each algorithm based on 50 searches on both the reduced and full data set. The **scikit-learn** algorithm yielded only one result. The values in parenthesis in area ratio is the area under curve (AUC).

Algorithm	Area ratio (AUC)	Accuracy (test data)	Accuracy (average)	F_1-score (test data)	F_1-score (average)
		Reduced data set			
GD	0.616 (0.808)	0.771	0.775	0.771	0.776
SGD	0.617 (0.809)	0.769	0.773	0.767	0.773
`scikit-learn`	0.620 (0.810)	0.772	0.779	0.769	0.777
		Full data set			
GD	0.532 (0.766)	0.777	0.777	0.783	0.783
SGD	0.530 (0.765)			0.777	0.779
`scikit-learn`	0.531 (0.766)	0.823	0.823	0.801	0.802

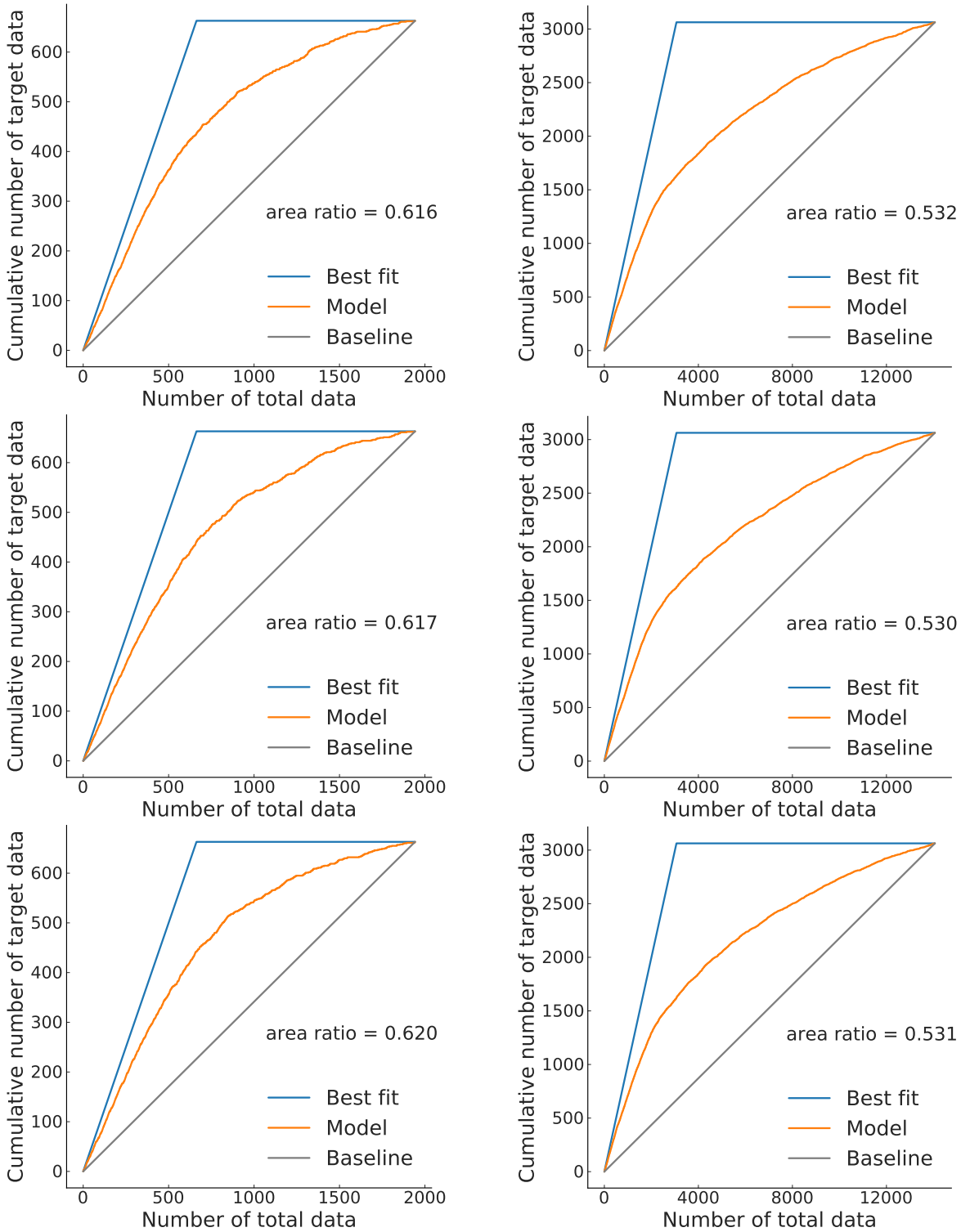

FIG. 10. Lift plot of the logistic regression searches yielding the best area ratio of the test data. The left plots are for the reduced data set, the right plots for the full data set. From the top: GD algorithm, SGD algorithm, `scikit-learn` algorithm

gorithm is to be preferred over the SGD algorithm.

A final note is that the structure of the cost function plane is hard to know, given that we have 28-40 dimensions. It would require very much to map out everything, but if there are many places on $\mathcal{C}(\hat{\beta})$ that are a local minimum we would expect the GD algorithm to perform worse than SGD. Now if the cost function surface were to be very flat this would no matter, as the cost at any point is fairly equal. We see quite large changes in the cost function so this is probably not the case in our case. Most likely $\mathcal{C}(\hat{\beta})$ has a defined global minimum, but it might have smaller fluctuations on the surface which creates the differences between each search. In such a case the GD algorithm would not be a problem and the SGD is not necessary. In addition, we know that several of the data columns are quite correlated or anti-correlated. GD and SGD algorithms tend to perform worse if the design matrix is very correlated [9], so a possibility in our case would be to remove some of the data types, i.e. columns, of the data set and see whether one is able to make a better fit by doing so.

Neural Networks

For this project we implemented a simple and efficient feed-forward neural network. First, we applied it to the credit card default data set. The network consists of an input layer, one hidden layer and an output layer. For the classification issue, we chose a Softmax output activation in combination with a binary cross-entropy loss function.

Optimization

The implemented neural network has many different hyper parameters that need to be tuned in order to get the best possible result. First and foremost is our choice of activation function for our hidden layer. From theory, both sigmoid and tanh is known to be well suited for these tasks, and because our network is very small, we will not be bothered by vanishing gradients. However, all available activations (Sigmoid, tanh, ReLU and ELU) will be considered. In addition to this, we want to fine tune the regularization parameter as well as the learning rate for each of the activations. Finally, we also look at the impact of hidden layer size and number over our parameter space.

Figure 11 show an example of the optimization on the credit card data using payment values of 0 and -2. The optimization calculates accuracy and area under curve as a function of epoch for a variety of parameters. For visualization purposes, a smaller parameter grid was chosen for these figures, although all previously mentioned activation functions, learning rate and regularization in the range 10^{-5} to 10 was considered. Additionally, a strict convergence criteria was applied, so that most updates stop after 11 epochs (strictly for visualization).

Figure 11 shows the neural network predicting credit card defaults with the sigmoid and relu activations using a hidden layer size of 12, which has shown to yield consistently good results. Here, all runs with more than an 0.822 accuracy score have been highlighted in the plot, and we see that the pink line, a relu activation with learning rate 0.01 and regularization parameter of 0.003 gave the best result; weighted F_1-score 0.801, accuracy 0.823 and auc of 0.658. We also note that several of these runs have obviously not converged, but full runs show the end result gives a worse score than the quick relu network mentioned.

We also note that the network is quite slow even though it only has one hidden layer. No specific measures have been taken to speed it up, one such improvement could be using a different optimizer than SGD, such as as ADAM, which uses adaptive learning rates for the weights. Additionally, optimizers such as momentum gradient descent could help us out of local minima, but is mainly a problem in data sets with large parameter space.

Next, in figure 12 we find that running the reduced credit card data set through the network gives higher area under curve scores. This is when payments that equal 0 and -2 have been removed. In this particular setup, a sigmoid activation is preferred with a learning rate of 0.001 and regularization of 0.1, still with a hidden layer size of 12.

In order to display the impact of hidden layer size on the accuracy and area under curve, we fixed the other parameters on the best fit sigmoid model found for the reduced set and ran with varying hidden layer size. As we can see in figure 13, the variance is very small for converged results over the different sizes.

The last result for the classification analysis can be seen in figure 14, which shows the change in

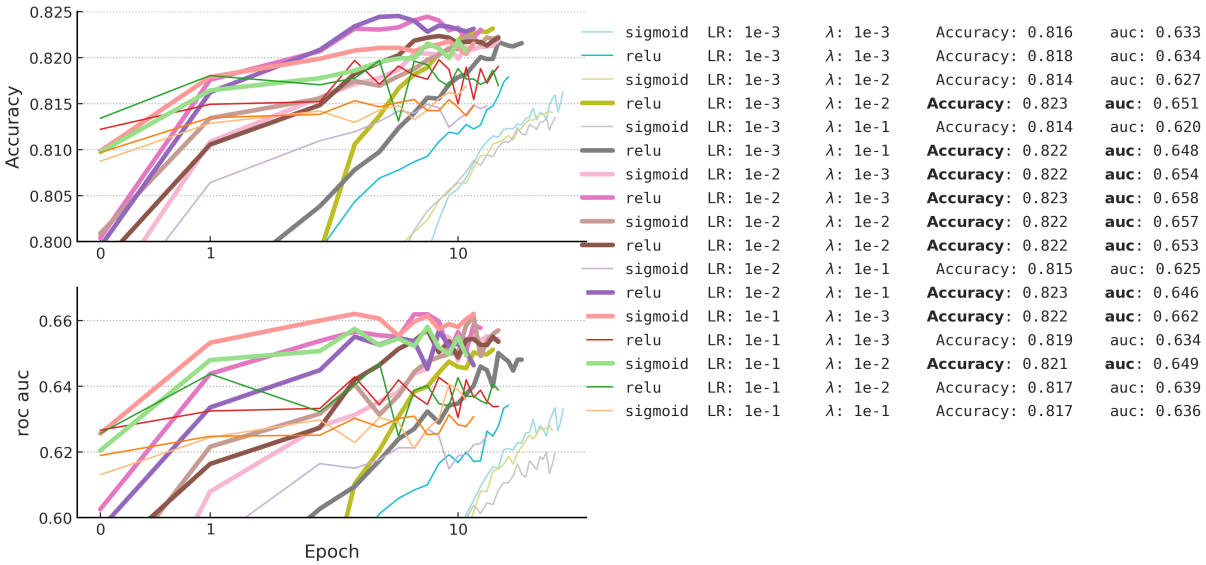

FIG. 11. Neural network classification on (almost) full credit card data. Accuracy and area under curve score over epoch with hidden layer size 12 and batch size 100. Runs with an accuracy score of more than 0.82 have been highlighted with thicker lines. Both `relu` and `sigmoid` give reasonable results but pink; relu with $\texttt{LR} = 1e-2$ and $\lambda = 1e-3$ gives the lowest accuracy and auc.

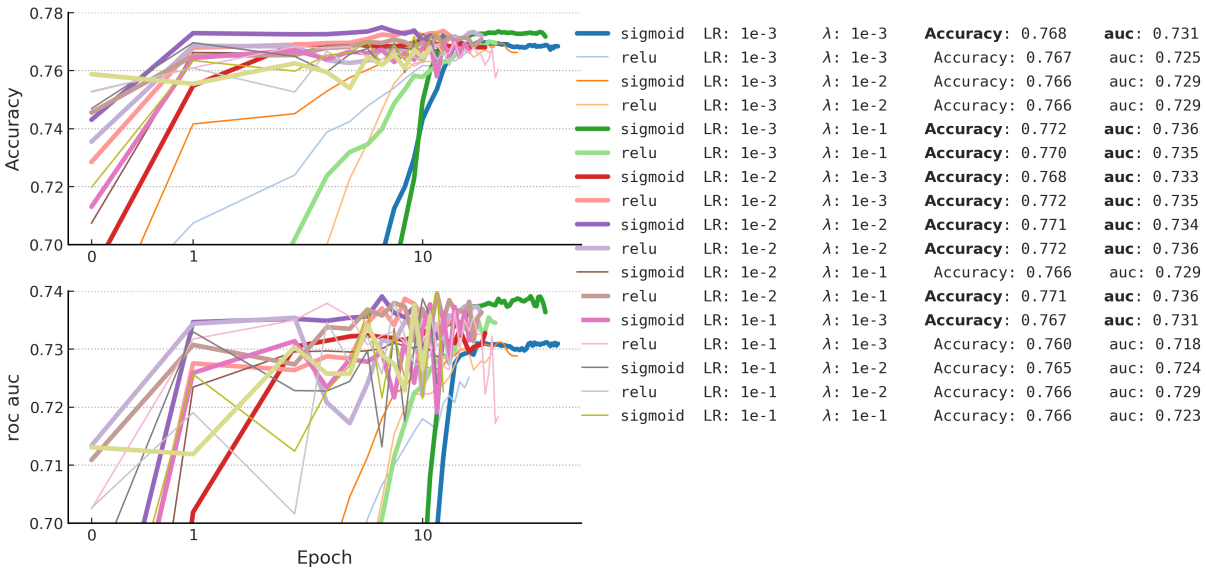

FIG. 12. Neural network classification on reduced credit card data. Accuracy and area under curve score over epoch with hidden layer size 12 and batch size 100. Runs with an area under curve score of more than 0.73 have been highlighted with thicker lines. Both `relu` and `sigmoid` give reasonable results but green; `sigmoid` with $\texttt{LR} = 1e-3$ and $\lambda = 1e-1$ gives the lowest accuracy and auc.

cost over epochs for varying activation functions, learning rates and regularizations. There is little variance in cost for the final epoch, as every run seems to converge nicely.

4.7. Regression

In this section we measure our ability to fit a polynomial to the Franke function of a 20×20 grid with a gaussian noise with standard deviation 0.05. To make a proper comparison, we run the regression using ridge regression in the code developed for project 1, before comparing with our simple feed forward neural network.

Linear regression

For comparison, we also want to run a linear regression on the same Franke function data. We therefore refer the reader to project 1 [4, 5] where the optimized ridge regression returned an R2 value of 0.943 with a regularization term $\lambda = 10^{-5}$.

Neural Networks

In order to make a neural network suited for a regression problem, we made some minor adjustments to the existing classification network. As we have shown in the methods chapter, the choice of cost function, whether cross entropy or MSE has shown give similar backpropagation algorithms. While we chose a softmax activation for the output and cross entropy for the cost of the classification network, we now use an identity activation $g(x) = x$ for the regression problem with an MSE cost function. The result is the same for both combinations, as has been deduced in the methods chapter.

Optimization

In order to find the best hyper parameters for this neural network we repeated the steps taken in the classification analysis. First, we run a large grid search over many parameters and check which perform the best, then we repeate the process with a smaller grid for visualization, using only the best parameters. After optimizing hyper parameters in figure 15 we see a strong preference for using ReLU in the hidden layer. All runs with an R2 larger than 0.8 are highlighted, and all of them use ReLU. It is also evident form this plot that not all runs are converged, the red line is still on a rising trajectory although the R2 is currently lower than the pink.

In this analysis, contrary to the classification case, figure 16 shows a higher variance in hidden neuron number score. It is clear that the network prefers larger hidden layers this time.

After optimizing over the full grid, the best fit was obtained using the neural network was applying the ReLU activation in the hidded layer along with a learning rate of 0.1, a regularization term $\lambda = 10^{-5}$ and a hidden layer size of 100, which resulted in an R2 of 0.947. This fit can be seen in figure 17.

In this project we have not implemented multilayer functionality. This is because after doing a comparison with scikit-learn, which can be seen in figure 18 and shows a network with two hidden layers, we are not able to beat the score of our single network. However, this has not been tested for a three and four-layer network, which may increase our score further. Although, deep neural networks work particularly well for larger data sets and hierarchical data structures.

5. CONCLUSION

After careful optimization of both our logistic regression implementation and our neural network (NN), we conclude that a simple feed forward neural network outperforms the logistic regression method, using both gradient decent (GD) and stochastic gradient descent(SGD), on accuracy, while on the measure of area under curve it is the opposite. Using logistic regression, we were able to get an accuracy of 0.777 and an area under curve of 0.766 for the full data set. For the reduced data set these values were 0.771 and 0.808 respectively. However, our neural network with a single hidden layer managed to get a maximum accuracy of 0.823 and for the reduced set an area under curve (AUC) score of 0.73. Looking at the weighted F_1-score we see that the neural network again is outperforming the logistic regression methods with a value of 0.801 (same as `scikit-learn`) for the full data set vs. 0.783 for the GD method. The increased area under curved for the reduced set might be caused by the stronger correlations in the payment history ("pay") category of the data set.

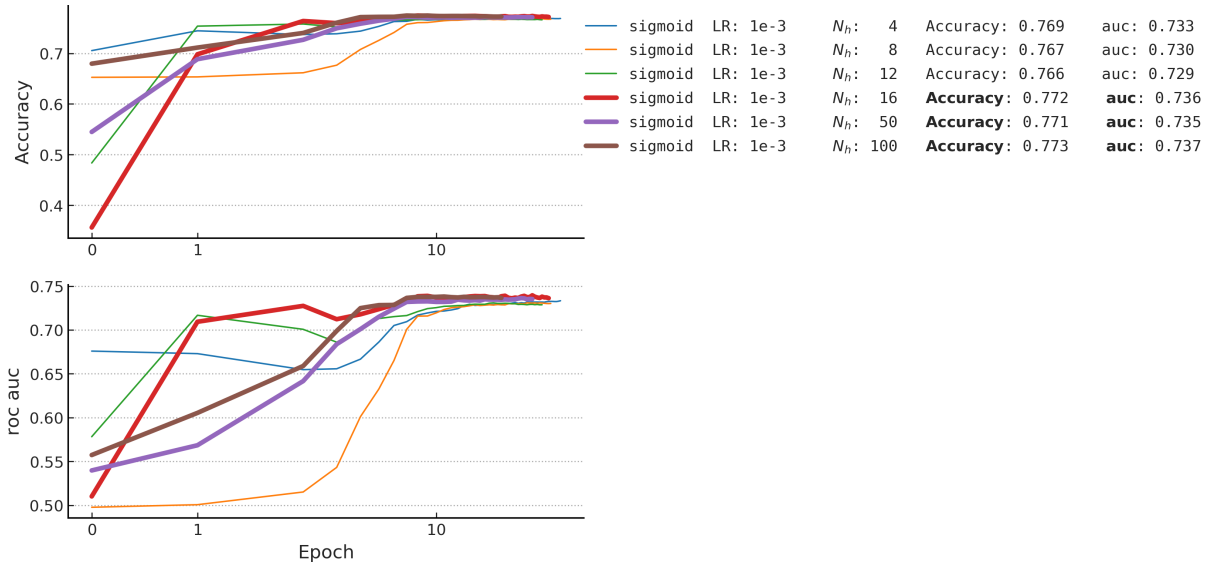

FIG. 13. Neural network classification on reduced credit card data. Accuracy and area under curve as a function of epoch with varying size of the hidden layer. After finding the best hyperparameters from figure 12, we ran with varying sizes of hidden layers. It turns out that the size we chose resulted in lowest scores, but with very little variance.

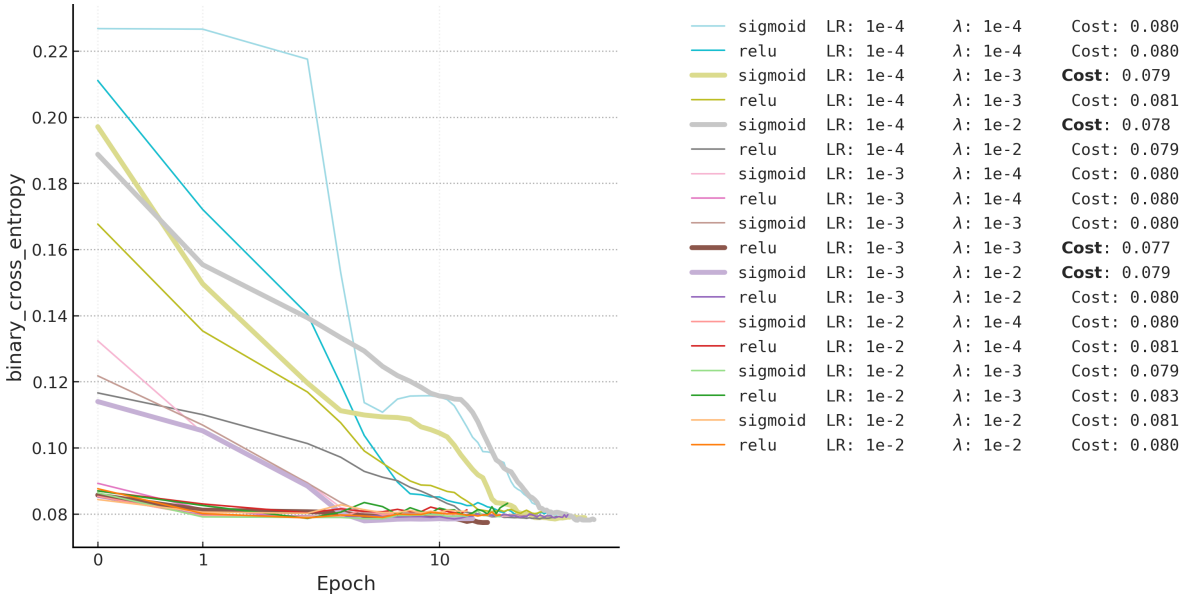

FIG. 14. Neural network classification cross entropy cost over epochs. Runs with cross entropy lower than 0.079 have been highlighted. Here, the system seems to prefer low learning rates and little regularization, but there is little variance across the board and every run seems to converge nicely.

For the regression analysis, the linear regression method (Ridge regression) gives similar results to that of the neural network, however, with optimalization, the neural network once again outperforms ridge regression with an R2 score of 0.947 vs. a score of 0.943 for ridge. The presented results have been compared with scikit-learn implementation, which yields similar results.

Possible improvements could be the implementation of an ADAM optimizer and further analysis with deeper networks and perhaps even better data-tuning. In the case of the logistic regression algorithms our algorithms could not match the algorithm from `scikit-learn`, but the neural network had better accuracy though a much smaller area under curve (or area ratio, AR). Why this is the case we do not know at the moment, and it would be the first thing to solve going forward.

On the classification problem, since the NN is outperforming on accuracy and F_1-score, while the logistic regression methods outperform NN on area ratio/area under curve, then there is no way of telling which one is the better. It would depend on what you are looking for. If you want the total pickture, then NN is the winner. If you want to get most return on money spent dealing with the clients, then GD or SGD is the one you should pick. What is quite interesting is that the AR/AUC for NN is so much smaller than for the logistic regression. Understanding why would be the next step in working with the algorithms.

REFERENCES

[1] UCI University of California Irvine. *Default of credit card clients Data Set*. 2016. URL: https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients (visited on 11/12/2019).

[2] Ivy Yeh and Che-Hui Lien. "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients". In: *Expert Systems with Applications* 36 (Mar. 2009), pp. 2473–2480. DOI: 10.1016/j.eswa.2007.12.020.

[3] Richard Franke. *A critical comparison of some methods for interpolation of scattered data*. Tech. rep. Monterey, California: Naval Postgraduate School., 1979.

[4] Trygve Leithe Svalheim. *Regression and Resampling methods*. Oct. 2019.

[5] Kristian Joten Andersen. *Project 1. FYS-STK4155 Data analysis and Machine Learning*. Oct. 2019.

[6] Morten Hjorth-Jensen. *Lectures Notes in FYS-STK4155. Data Analysis and Machine Learning: Logistic Regression*. Oct. 2019.

[7] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

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

[9] Morten Hjorth-Jensen. *Data Analysis and Machine Learning Lectures: Optimization and Gradient Methods*. Sept. 2019.

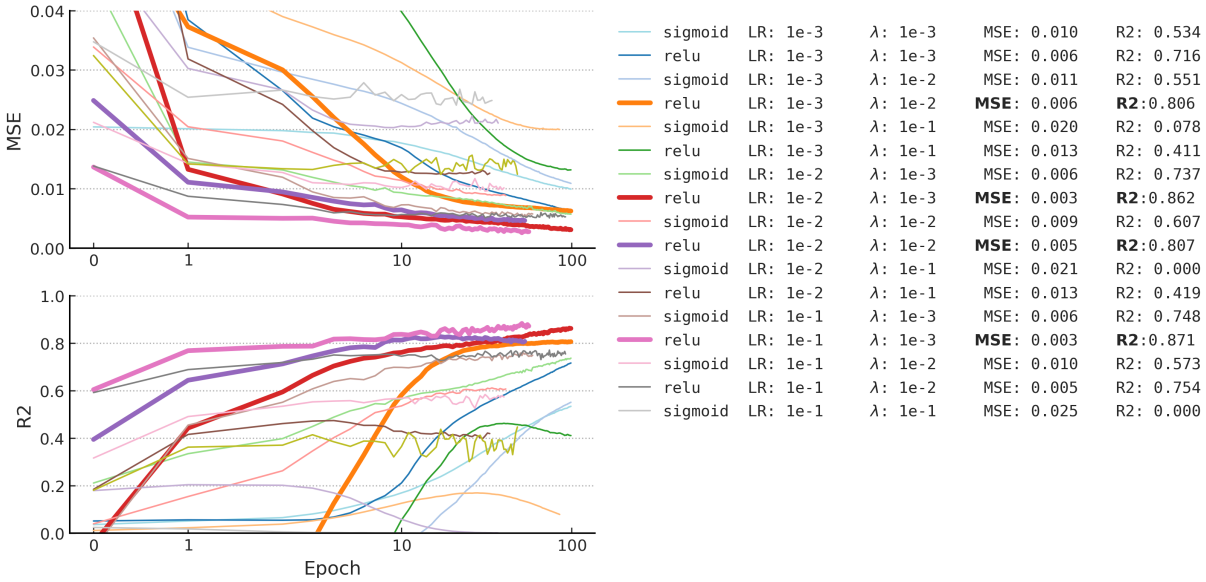

FIG. 15. Neural network regression on the Franke Function data with polynimial degree 5 and 20×20 grid withgaussian noise with standard deviation 0.05. MSE and R2 score over epoch with hidden layer size 12 and batch size 1. Runs with an R2 of more than 0.8 have been highlighted with thicker lines. `relu` is the preffered activation function and pink; $\texttt{LR} = 1e-1$ and $\lambda = 1e-3$ gives the lowest MSE and R2

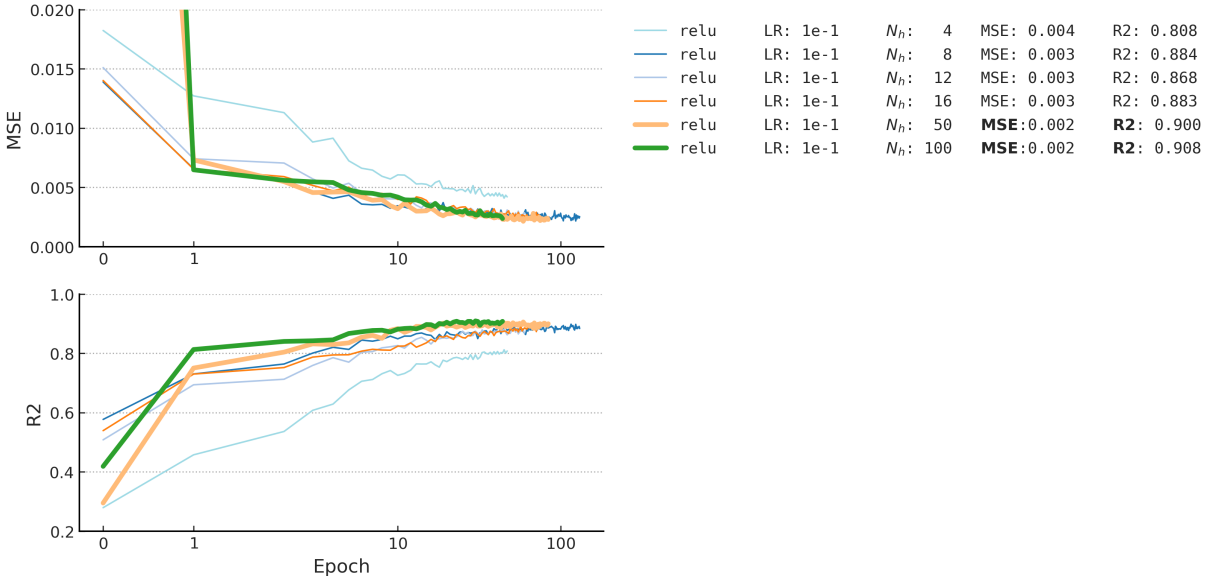

FIG. 16. Neural network regression on the Franke Function data with polynimial degree 5 and 20×20 grid with gaussian noise with standard deviation 0.05. MSE and R2 as a function of epoch with varying size of the hidden layer. After finding the best hyperparameters from figure 15, we ran with varying sizes of hidden layers. The optimal hidden layer size for these hyper parameters is 100.

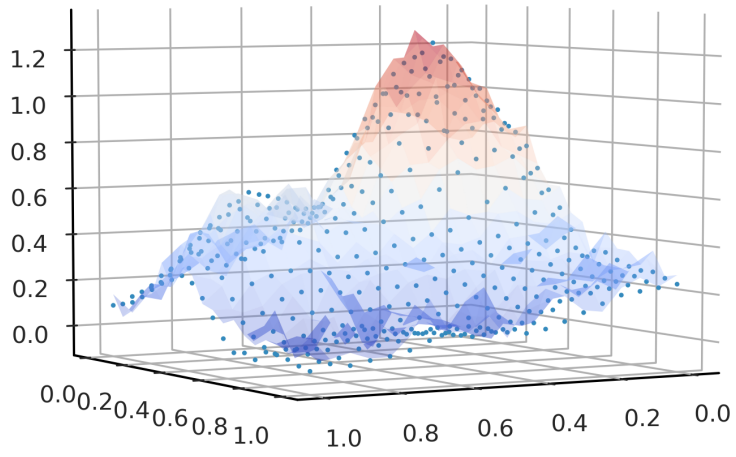

FIG. 17. Neural network regression on the Franke Function data with polynimial degree 5 and 20×20 grid with gaussian noise with standard deviation 0.05. With optimized parameters; `relu`, `LR` $= 1e-1$ and $\lambda = 1e-5$ $N_{hidden} = 100$ we get an R2 of 0.947.

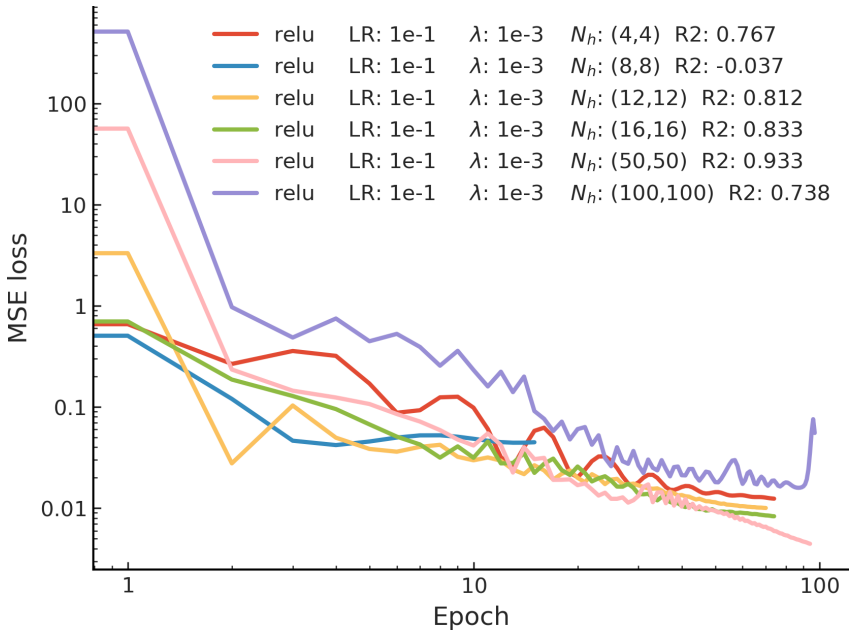

FIG. 18. Scikit learn multi-layer Neural Network regression on the Franke Function data with polynimial degree 5 and 20×20 grid with gaussian noise with standard deviation 0.05. After doing a grid search over hyperparameters this plot shows the best fit with varying layer size. With multiple layers we are not able to beat a single hidden layer.