FYS-STK4155 Project 2 - Classification and regression, neural networks

Joakim Flatby, Linus Ekstrøm

October 2019

https://github.com/jflatby/FYS-STK4155

Abstract

In this project we study logistic regression and multilayer perceptrons for classification purposes. Our study is of the default of credit card data set made available at the UC Irvine machine learning repository. We look into predicting whether a person will default on their credit based on various predictors from the data set. In the logistic regression analysis we obtain a maximum accuracy-score $A = 0.8160$ with learning $\eta = 0.1$ and $L2$-regularization $\lambda = 0.01$. We are more unsure of our results with the neural network. There has been a fair amount of confusion regarding the neural network detailed more in our concluding sections. Our analysis of the neural networks gives us the values $\eta = 0.0056$ and $\lambda = 0.0078$.

Contents

1 Introduction

In this project we will be dealing with both regression and classification problems. The analysis will be performed on a data set consisting of credit card default payments from Taiwan [1]. The data set consists of a variety of categories which we will make our predictions from. Some of them are: age, sex, marriage status, current available credit, etc. We start of with a logistic regression analysis of the data set. After this we use our own code for a multilayer perceptron artificial neural network in order to predict defaulting of credit card payments in terms of a binary classification problem. In other words, we train our network on a partition of the data and use the remainder of the data to test whether we achieve a higher than random prediction rate for people defaulting on their payments. For the analysis of our work we consider both the accuracy and the often used $F1$-score.

2 Theory

In this section we give a brief introduction to the necessary background knowledge for the analysis we perform in this project.

2.1 Logistic Regression

Logistic regression is a method in regression analysis commonly used when dealing with a data set that consists of discrete responses. We denote the number of possible values, or *classes* by K. This is possible to do for higher values of K, however in this project we will only consider the case of binary classification. That is, our data set will only have two possible responses $y_i \in \{0, 1\}$.

What logistic regression attempts to do is calculate the probability that the data point y_i will belong to one of the classes(automatically giving us the probability for the other class as well, since there are only two), based on the input $\mathbf{x}_i$. To achieve this we apply the commonly used sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and if the answer is above 0.5 our prediction is class 1. The regression model itself is calculated just as in linear regression

$$\hat{y} = \mathbf{X}\vec{\beta}$$

where $\vec{\beta} = (\beta_0, \beta_1, \dots, \beta_n)$ are the beta-coefficients of the model and $\mathbf{X}$ is the design matrix. As such, we use the sigmoid function to calculate the probability of $\vec{X}_i\vec{\beta}$ corresponding to a response in class $1(y_i = 1)$, given by

$$p(y_i = 1|\vec{X}_i\vec{\beta}) = \sigma(\vec{X}_i\vec{\beta})$$

and thus we also have the probability for class 0

$$p(y_i = 0|\vec{X}_i\vec{\beta}) = 1 - p(y_i = 1|\vec{X}_i\vec{\beta})$$

The model's prediction will be the value of most probable class.

2.1.1 Cross-entropy

To measure the performance of our classification model we use cross-entropy loss(or log loss). In binary classification we can calculate cross-entropy as

$$y\log(p) + (1 - y)\log(1 - p)$$

which can be rewritten to

$$-\Big((y(\vec{X}_i\vec{\beta}) - \log(1 + e^{\vec{X}_i\vec{\beta}})\Big)$$

and summing over all the training data examples we get

$$\mathcal{C}(\vec{\beta}) = -\sum_{i=0}^{n} y(\vec{X}_i\vec{\beta}) - \log(1 + e^{\vec{X}_i\vec{\beta}})$$

which is the cost function we use in our neural network.

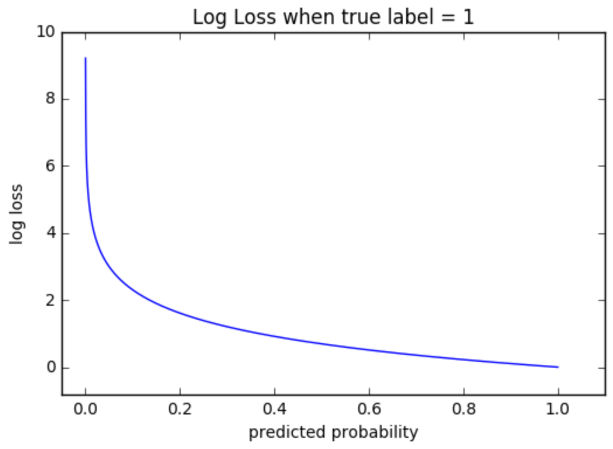

Figure 1: Cross-entropy loss(Log loss) visualized when 1 is the true value.

In Figure 1 we see the log loss plotted against the predicted probability, for the case where the true value is equal to 1. As we can see, the cross-entropy loss decreases as we approach the correct value, converging to 0 at the correct value(1) and increasing exponentially the further away from the correct prediction we get. What we want to achieve is to minimize this cost function, which is done by *gradient descent*, using the gradient of the cost function

$$\nabla C(\vec{\beta}_i) = -\hat{X}^T(\hat{y} - \hat{p})$$

As with project 1 both algorithms in this project can be altered to include the L_2-regularization which changes the expressions for the cost function and the cost function derivative. It boils down to adding a simple term to both functions, more details can be found in the lecture notes

2.2 Accuracy Calculations

To measure the performance of our calculations we use the *accuracy*-score and the $F1$-score. The accuracy-score is just the number of correctly guessed targets t_i divided by the total number of targets.

$$\text{accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y)}{n}$$

where I is the classifier function, 1 if $t_i = y$ and 0 otherwise. When the data set is biased as in the credit card data's case, a lot fewer people default on their debt than not, it is often necessary to use the so called $F1$-score. More information about the $F1$-score is available at [2]

2.3 Gradient Descent

Gradient descent is a method for finding the minimum of a function. It is a first-order iterative algorithm originally proposed by Augustin-Louis Cauchy in 1847 [3]. When dealing with machine learning and neural networks the function which is minimized is the cost function. Gradient descent requires the cost function to have a non-vanishing first order derivative with respect to the parameters β. In a multivariate situation the derivative is known as the gradient, and the crux of the gradient descent method is to iteratively 'walk' along the path with the steepest (*most negative*) gradient with respect to the β's.

$$\vec{\beta}_{i+1} = \vec{\beta}_i - \eta \nabla C(\vec{\beta}_i)$$

where η is referred to as the learning rate, which can be either fixed or adaptive. It is not uncommon, in machine learning problems, for the cost function to not be strictly convex making such an iterative process desirable.

2.3.1 Stochastic Gradient Descent

Stochastic gradient descent does the same thing as regular gradient descent, but instead of calculating the gradient on the entire dataset, you calculate it from a randomly selected subset of the data. Doing so can decrease the computational load, and therefore it is commonly applied when using big data sets. Each iteration will take less time to complete, however, it will also reduce the accuracy, or rather lower the convergence rate.

2.4 Artificial Neural Networks

An artificial neural network is a system for approximating a desired function output inspired, but not identical to, biological neural networks i.e. the human brain. These networks are able to *learn* by considering supplied examples of the function they are made to approximate. As stated in our introduction the neural network portion of this project will be focusing on the simplest type of artificial neural networks: *the multilayer perceptron.*

2.4.1 Multilayer Perceptron

A multilayer perceptron (MLP) is a class of *feedforward* neural network, often referred to as the most basic form of neural network. The absolute simplest case of a MLP consists of an input layer, a *hidden* layer and an output layer. Each layer consists of several nodes, or perceptrons. The perceptron was invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt [4]. The perceptron is function which outputs a single binary value from a set of binary input values.

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

where $\vec{x}$ is a vector of binary inputs, and $\vec{w}$ is a vector of weights with w_i corresponding to input x_i's weight. In addition there is an added bias b to the output of each perceptron. The bias is important as it shifts the decision boundary away from the origin and is independent of the input values.

For the network we are studying in this project we use more complicated versions of Rosenblatt's perceptron in our hidden layers. Where the Rosenblatt perceptron's output is a linear weighted sum of the inputs we use a non-linear *activation function* before sending the signal further through the network, the primary activation function studied in this project is one we already talked about in section (2.1): the Sigmoid function.

2.4.2 Making Predictions with the Neural Network

With a MLP setup as described above we now consider how such a network can be used as an universal function approximator. Going back to the simple perceptron, it is fairly obvious this has the limitation of begin a good classifier only when we have a standard binary model with linear boundaries between outcomes. Thus, the simple perceptron could be replaced by standard or logistic regression with the same results. Even multilayered perceptrons, with linear activation functions, can be shown to produce identical results to a simple perceptron with a larger hidden layer. However, the prediction ability for the multilayer perceptron we are studying in this report stems from the non-linear activation functions. These functions allow the network to function as a universal function approximator, given enough layers. This is known as the *universal approximation theorem* and is detailed in the following paper [5]. That is: the predictive power of artificial neural networks stems from the concept of superposition of non-linear functions. Now we come to how we use the neural network to make prediction, the *feedforward pass.* Below are the equations for what a neural network 'is'.

We denote the number of features, **F**, the number of hidden neurons **H** in layer l and **C** the number of predictive categories. For each input to our network we calculate

$$z_j^l = \sum_{i=1}^{\mathbf{F}} w_{ij} x_i + b_j^l$$

each of these sums are then passed through whatever activation function was chosen

$$a_j^l = f(z_j^l)$$

This is subsequently passed further along to either the next hidden layer $l+1$ or to the output layer **L** if the data has gotten to the end of the network

$$z_j^{\mathbf{L}/l+1} = \sum_{i=1}^{\mathbf{H}} w_{ij}^{\mathbf{L}/l+1} a_i^l + b_j^{\mathbf{L}/l+1}$$

Finally we calculate the output of neuron j in the output layer using an activation function suited for our problem. This procedure will for an input vector $\vec{x}$ produce a prediction in the form of the output of the output neurons. This is how a neural network can be used to arbitrarily approximate any function.

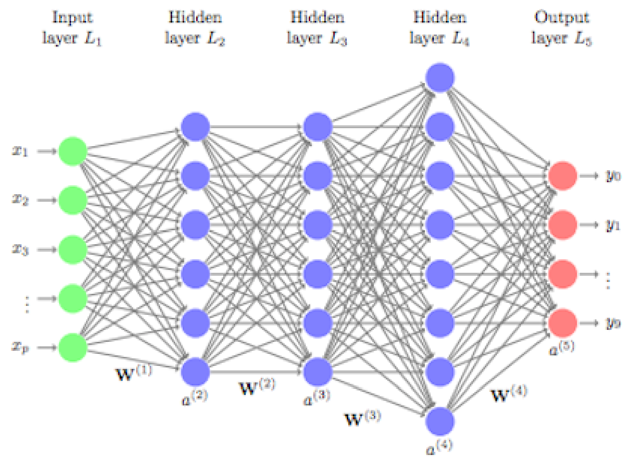

Figure 2: Flow of information in MLP, retrieved from [6]

Note how the prediction of the network is entirely up to the choices for the weights and biases.

2.4.3 Training the Neural Network

The previous section discussed what a neural network consists of and what it '*is*'. Now we move on to what actually lets it make predictions. We have to turn all the 'dials' for all the hidden weights and biases such that our network outputs our desired output. Doing this manually can be feasible for small cases, i.e XOR affair [8], however as the network complexity increases it quickly becomes infeasible. The algorithm which lets us turn and tune each 'dial' throughout the network is called *back-propagation*. Back-propagation is summed up by us calculating the difference in the networks output from the desired output, and then *propagating* this error backwards throughout our network in order to tune the weights and biases which have the greatest effect on the output error. The way of finding the way in which to tune the weights and biases is the aforementioned gradient descent method (2.3).

1. Calculate the error in the output layer according to $\delta_j^L = f'(z_j^L)\frac{\partial C}{\partial(a_j^L)}$
2. Compute the back propagate error for each subsequent layer $\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l)$
3. Update the biases and weights according to the gradient descent method $w_{jk}^l \leftarrow w_{jk} - \eta\delta_j^l a_k^{l-1}$ and $b_j \leftarrow b_j^l - \eta\delta_j^l$

where a_j^L is the activation of the j-th neuron, $f'(z_j^l)$ is the derivative of the activation function of the j-th neuron with respect to its input, and $\frac{\partial C}{\partial(a_j^L)}$ is the derivative of the cost function with respect to the activation a_j^L which in our case is equal to the output of the network. More info on the derivation can be found in the lecture notes [7]. *Note, this derivation requires the activation function to be the sigmoid function, for other activations a different derivation is required.*

3 Method

3.1 Data Handling

Before writing our neural network code we took a look at the credit card data we are using in this project. Some of the data showed to include inconsistencies, for example some entries in the education and marriage columns have values outside the discrete choices they are supposed to have. We completely removed these entries from the dataset.

3.1.1 Correlation Analysis

Next we plotted a correlation matrix of the data which is shown in Figure 3. As we can see, there are several columns in our data that have a decently strong correlation. In particular it seems like all the PAY, PAY_AMT and BILL_AMT-columns correlate locally within themselves. We can see this from the squares and rectangles forming in the correlation matrix. We decided to remove all except for one of each, and the correlation matrix from the resulting data set is shown in Figure 4.

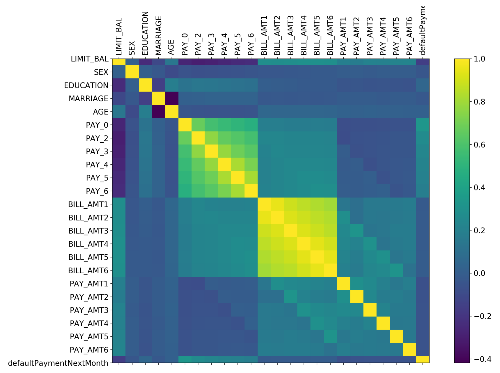

Figure 3: Credit card default data correlation matrix

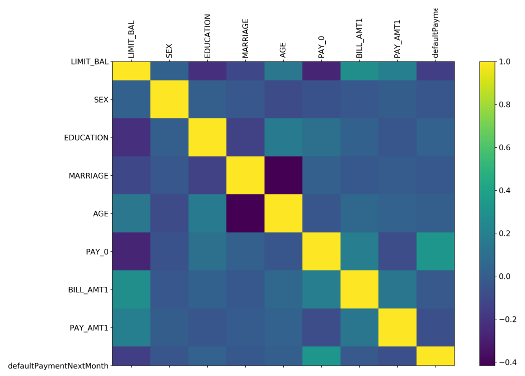

Figure 4: Filtered data correlation matrix

3.1.2 Re-scaling the Data

Before using our data into the logistic regression and neural network we made sure to rescale it using sklearn.preprocessing StandardScaler.

3.2 Logistic Regression

Our code for the logistic regression of the credit card data was implemented according to the equations in section (2.1). First we set up the design matrix according to the lecture notes on logistic regression [9]. Then we defined the cost function along with its derivative, both with and without the L_2-regularization term. With the gradient in place we made the method the gradient descent, which simply put loops over the desired number of iterations, calculates the gradient for each β-component and subtracts this from the existing β's. Once this was all set up we made a double loop with looped over the various learning rates and regularization values to generate the plot in appendix (A). We also generated the confusion matrices for various learning rates which can be seen in the logistic regression part of our results (4.1)

3.3 Multilayer Perceptron

In this section we elaborate on how we set up our code for the neural network.

3.3.1 Network Shape, Weights and Biases

We implemented our neural network as a class: NeuralNetwork which is initialized with a range of variables. We send in the training data and targets, the network shape (list), the learning rate η, the number of epochs and number of batches. In the init function we also call a secondary function which sets up the biases and weights. The way we do this is enumerate through the network shape from the first hidden layer and set the biases for each node equal to 0.1 and the weights to np.random.uniform with shape $(l, \text{network\_shape[i]})$. This ensures each neuron in the previous layer is *connected* to each neuron in the next layer with a weight. We made the neural network class work with any number of layers and nodes within each layer. The only constraints are on the input and output layers, these have to match with the number of features in the data and the desired mode of output respectively.

3.3.2 Feed-forward and Back-propagation

Next we implemented the methods for prediction and adjusting the weights and biases as described in sections (2.4.2, 2.4.3) respectively. For the feed-forward case we again enumerate through the network shape. We separate the cases for the first and last layer. On the first layer we input the data and not the activation from the last layer. For the last layer we use the soft-max function as the activation function instead of the sigmoid to obtain a probability for our outputs. In our back-propagation implementation we loop backwards through our layers calculating the layer error and use this to calculate the gradient for the weights and biases according to the equations in sections (2.4.3).

4 Results

In this section we present our results from the various algorithms we implemented as described in the previous section.

4.1 Logistic Regression

Below are our results for the logistic regression of the credit card data set with varying learning rates η. In addition we include the prediction accuracy as a function of the learning rates. We observe the accuracy generally tends downwards as the learning rate increases. Additionally, due to time constraints, we were not able to run the algorithm long enough for the gradient descent to converge in all the instances. This may have skewed the results slightly downwards.

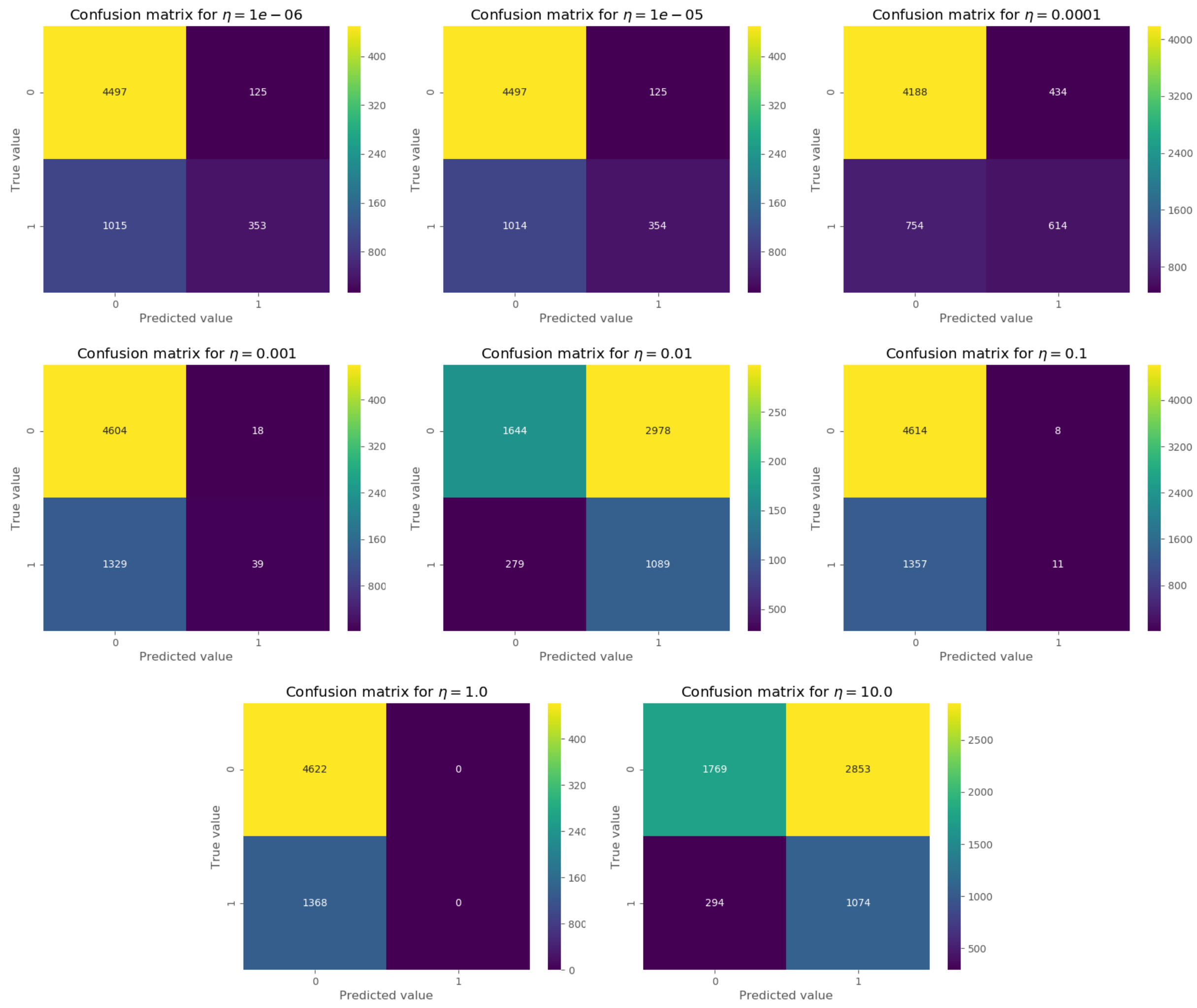

Figure 5: Confusion matrices for our logistic regression on the credit card default data set with $\eta \in (1e-6, 10)$

The following table contains the values for the accuracy for each learning rate in the figures above. The outlier values stem from an overflow error in the sigmoid function we were not able to fix in this project, this is definitely something we would address upon revisiting the project. Following this table we have included a heatmap of our grid-search for the best η and λ values. We have included a larger version of it in the appendix (11)

Accuracy	η
0.8097	1e-6
0.8098	1e-5
0.8017	1e-4
0.7751	1e-3
0.4562	1e-2
0.7721	1e-1
0.7716	1
0.4746	10

Table 1: Accuracy of the logistic regression predictions as a function of the learning rate.

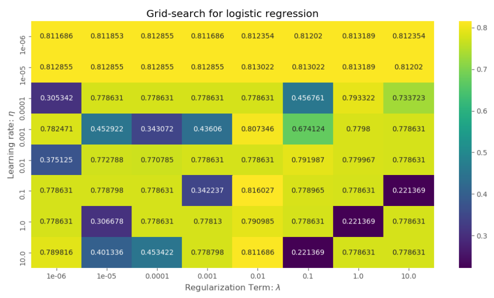

Figure 6: Grid-search for optimal η and λ values for the logistic regression.

4.2 Neural Network

Below are our results for the classification of the credit card data using our constructed neural network. The following grid-search was generated using two hidden layers each with 50 hidden neurons.

4.2.1 Grid Search over η and λ

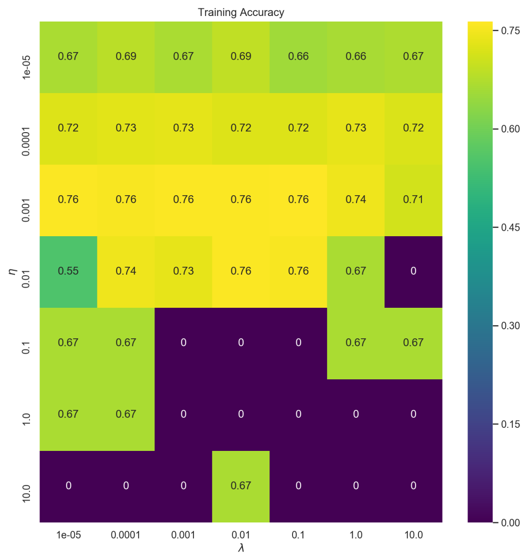

Figure 7: Training predictions from grid search using our neural network for several different values of η and λ. Using 30 epochs, 50 batches, and two hidden layers of 50 neurons each.

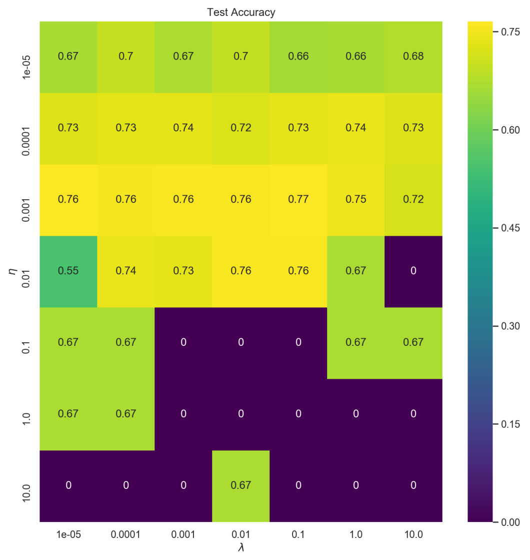

Figure 8: Test predictions from grid search using our neural network for several different values of η and λ. Using 30 epochs, 50 batches, and two hidden layers of 50 neurons each.

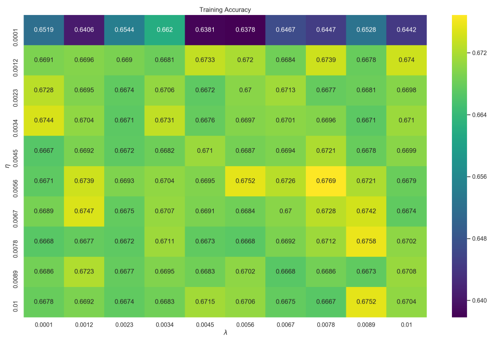

Figure 9: A more narrowed down grid search over η and λ. A bigger, more easier to read version can be found in the appendix.

4.2.2 Grid search over number of layers and number of neurons per layer

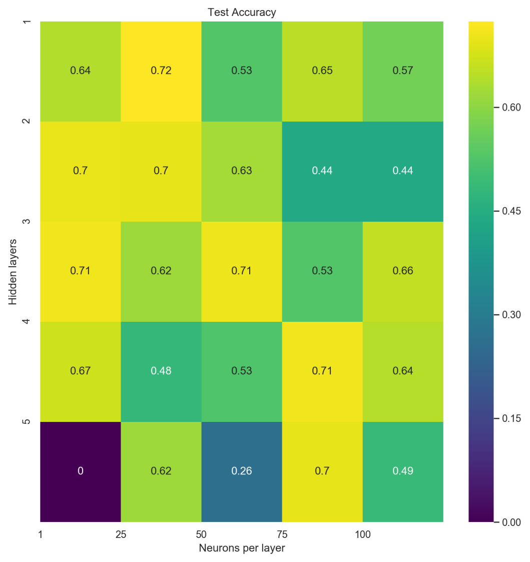

Figure 10: A grid search over the number of layers, as well as the number of neurons per layer, using the best parameters found from previous results.

5 Discussion

5.1 Logistic Regression

The results obtained from our logistic regression show a general trend of the accuracy worsening as the learning rate increases. This trend is not readily apparent as the regularization term increases. There are also some combinations with higher learning rates which yield similar results as the smaller ones. *note*: $\{\eta = 0.001,\ \lambda = 0.01\}$, $\{\eta = 0.1,\ \lambda = 0.01\}$, $\{\eta = 10,\ \lambda = 0.01\}$. Interesting to note that these values all lie along the same value for λ. The three 0.22-scores in the bottom right hand corner all stem from the effect of the combination of the learning rate and regularization term being equal to 1. We also note that the accuracy values in table (1) decrease as a function of the learning rate, but it decreases from a value which is not as high as the maximum value in the grid-search. This indicates that we obtain a greater precision for the regression when using the regularization term. I.E when performing regression it is beneficial to use a form of regularization to potentially reduce co-linearity in the data set.

5.2 Neural Network

After all the time we spent with this and different things we tried, we have still not managed to properly wrap our heads around the results from our neural network. Using over-sampling, we seem to be getting proper results, which actually improve when using the proper parameters, however it also seems to let the network perform fine even with ridiculously large or low paramteres. This is more thoroughly discussed in the next section.

As we can see in Fig. 10 we also tried to do a grid search over network shapes. Using the amount of layers as one variable and the amount of neurons per layer as the other. This result we can't really make much of.. The fact that the best network shape seems to be a single hidden layer with 25 neurons seems really strange. And the fact that it gets a score of 0.64 with a single neuron

5.3 Problems

After finishing our code, handling all array sizes and indexes correctly so that the code would run properly, we still had problems getting our network to make predictions properly. No matter our network shape, learning rate, λ or batch_size values, the network would predict 0 or 1 for all data points. Since a big part of the actual targets are zeros, this gave an accuracy score of around 72% and falsely led us to believe we had it working for a while.

After realizing this and doing some research, we tried using imblearn's SMOTE-function which is an oversampling technique often used to avoid this exact problem. After implementing SMOTE, our network finally started making predictions other than all 0 or all 1. At this point we thought our code was working again, getting plots for a F1 score grid search which are shown in the results section (4.2).

This seemed like it could be correct, as we finally got different predictions using different parameters. However, we quickly realized that the accuracy on the test set was the same, or sometimes even a little higher, than the accuracy from predicting on the training set, which makes little sense, and definitely wasn't a random occurence. The small changes also seemed like they could have just arised from rounding errors along the way. This seemed very strange, almost as if we were still only predicting 0 every time (especially since the accuracy was around 72% for the most part) however by looking at the confusion matrices and other output we could clearly tell that the network was predicting different values. We could also tell from the grid search plots (f.ex Fig. 7) that the accuracy wasn't always exactly the same value with different parameters, but it never went far away from 0.70.

This made us believe that there is something about the way SMOTE works that we don't fully understand. It also seems really strange that we suddenly get a well-defined score at learning rate = 10. Still, it didn't seem completely wrong, seeing that the highest accuracy scores were located around values for learning rate and lambda which seem to be about right, about 0.01 or 0.001. It does, however, seem strange that we get pretty close to the same accuracy score from using learning rate = 1.0. We eventually realized that even if we just had a single neuron in a single hidden layer, we would get about the same results. So something is definitely still wrong, or at least our understanding of the output our network provides is flawed.

Below is a case of our algorithm seemingly outperforming scikit-learn when using the SMOTE-upsamling technique. We find it strange because according to our accuracy score our network did not just guess all 1's or 0's considering getting a 73% accuracy is pretty good. *This was not just guessing randomly or all zeros / ones.*

Network Shape			
		Our Net	sklearn
8, 50, 50, 2	Accuracy	0.73	0.29
	F1-score	0.72	0.23
		Our Net	sklearn
8, 51, 50, 2	Accuracy	0.73	0.35
	F1-score	0.72	0.32

Table 2: Comparison of scikit-learn to our own algorithm.

6 Conclusions

In our own opinion we managed to obtain a decent result using logistic regression, however at the time of delivery we are still very unsure of our results regarding the neural network. From only predicting ones or zeroes to our network outperforming scikit-learn's implementation. We have spent an obscene amount of time looking for our mistake in the code. We have been through three or more iterations of our neural network. Comparing them to each other, finding an error in one and looking for the same in the other. We have also looked where it seemed one network was able to perform, be that scikit's implementation or our own. This has caused us to not be able to complete task d) because we did not feel we had a functioning network. This is something we would definitely look more into upon revisiting this project and something we are sorry we were not able to do. The variation of our results when tweaking the tiniest parameter, for example adding one neuron in one hidden layer completely changed our result in some occasion, has made us very wary of claiming something was ever right. We have learned a lot about the algorithms while working on this project and have definitely benefited from working on it. We understand more about the inner workings of both logistic regression and neural networks then before we started, although the learning process has been bittersweet.

A Logistic Regression Grid-Search

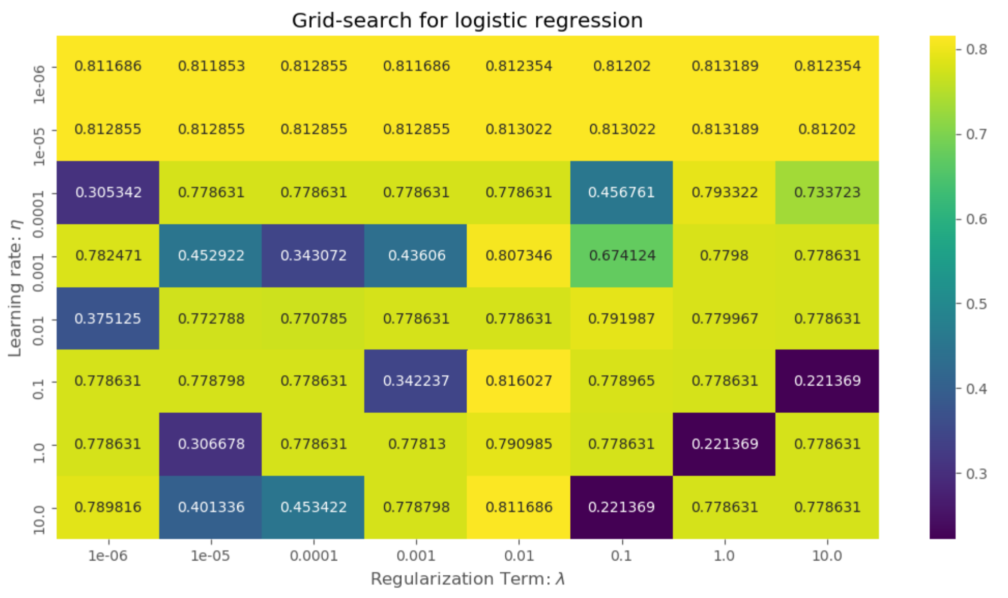

Figure 11: Larger variant of grid search for the logistic regression.

B Neural Network narrow Grid-Search

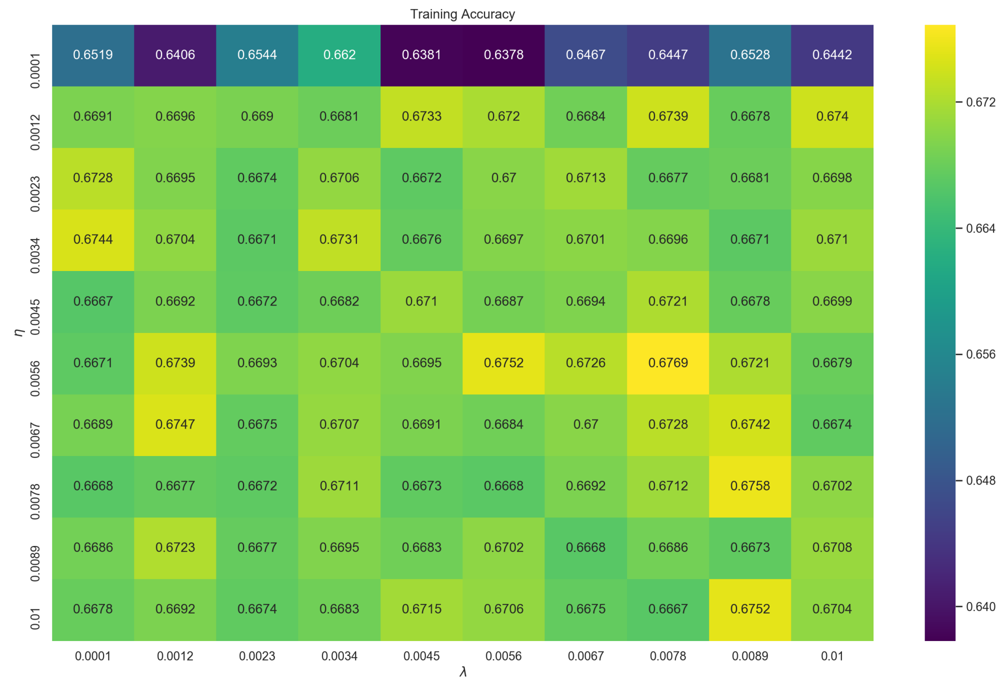

Figure 12: A more narrowed down grid search over η and λ

References

[1] Default of credit card data set. Last visited: 10/28/2019:
`https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients`

[2] Sasaki, Yutaka (Oct 26, 2007) The truth of the F-measure, School of Computer Science, University of Manchester

[3] Dimitri P. Bertsekas, Nonlinear Programming, Athena Scientific 1999, 2nd edition, pp. 187.

[4] Rosenblatt, Frank (1958), The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, pp. 386–408.

[5] Cybenko, George (1989), Approximartion by Superpositions of a Sigmoidal Function, University of Illinois, Mathematics of Control, Signals, and Systems, No. 2, pp 303-314.

[6] Feedforward Deep Learning Models. UC Business Analytics. Retrieved 15:26 November 1, 2019, `http://uc-r.github.io/feedforward_DNN`

[7] Hjorth-Jensen, Morten (Oct 4, 2019), Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning

[8] Minsky, Marvin and Papert, Seymour (1969), Perceptrons: an introduction to computational geometry

[9] Hjorth-Jensen, Morten (Oct 17, 2019), Data Analysis and Machine Learning: Logistic Regression