Project 2 on Machine Learning - Classification and Regression

Thomas Haaland
Jie Hou
Huiying Zhang
Martin Funderud Gimse
https://github.uio.no/thomhaa/Fys-Stk4155.git

Nov 2019

Abstract

In this project we use logistic regression and neural networks on the credit card data to predict the binary case of default/no default and we will also predict terrain data with regression and neural networks. In classification problems neural networks can achieve a higher accuracy than logistic regression. For different solvers, the Newton-Raphson method has the best performance when it is stable while normal the gradient descent is the worst. Stochastic Gradient descent is somewhere in between. We also explore the different cost functions MSE and cross-Entropy and found that MSE often performed better than cross entropy. We also see how PCA can retain the most important features in the dataset while reducing the dimensionality, thereby saving computation time. In the Regression problem, we compare neural networks to linear regression from project 1 to predict terrain data. Our neural network does not perform as well as ridge regression from project 1 and SKlearns NN outperforms both our Neural Network and ridge regression.

1 Introduction

The main content in this project is to use both classication and regression methods, including the regression algorithms studied in project 1, to solve the credit card problem.

We will first use logistic regression on the binary case of the credit card data using our self made code to classify the binary case of default/no default and the compare our own results with those obtained using tensorFlow and Keras. In order to find the optimal parameters, we use the Newton-Raphson method to find the best learning rate and combine with a stochastic gradient descent with minibatches, trying to get the best results. Then, we use accuracy score to measure the performance of our model for classication. Finally, compare our outputs with those from tensorFlow.

Then we try the more advanced method, neural networks that can model more advanced relationships between the predictors and the output classes. We use both our self-made neural network and neural networks from TensorFlow and compare the performance. Finally, we compare the results obtained from both networks on credit card data to discuss and evaluate the differences of the results we get. In order to find the optimal weights and biases, what we do in this part is to write our Feed-Forward Neural Network code to implement the back propagation algorithm. We use different kinds of activation functions, including the Sigmoid function, Relu, Linear function, Tanh and the Softmax function with Cross Entropy and MSE, trying to find the best combination of them. Then, we test our results using TensorFlow and compare the results with those from logistic regression. Finally, we analyse

the different kinds of regularization methods, Ridge Regression and Lasso Regression, and learning rates we use to find the optimal accuracy score. We will also use Principle Component Analysis (PCA) to pick up the main characteristics of the dataset and do the same analysis, in order to test whether we could still get high accuracy when we reduce the dimensions.

Then we will use the neural network we have developed to make predictions on the terrain data. And compare this method to the linear regression methods that we used in project 1.

After that, we will summarize the evaluation of all the algorithms, including advantages and disadvantages, suitable situations for each method.

2 Theoretical Background

2.1 Logistic Regression

Logistic Regression is often used when the dependent variable is categorical, a logit function which represent the likelihood for a given event is used to determine the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$. The reason for this is that logistic functions returns values in a range such as $\{0, 1\}$, while linear regression methods can return any value.

2.1.1 Classification

In case of classification problem, what we try to predict is whether the result belongs to a certain class (True or False). For instance, whether a tumor is malignant or benign; whether an email is a spam or not spam. So we divide these results which are dependent variables into negative class and positive class respectively. That means $y_i = \{0, 1\}$, where 0 represents a negative class and 1 represents a positive class. If we use Linear regression to solve this problem, the output could be much bigger than 1, or much smaller than 0. This is because a linear model treats the classes as numbers 0 and 1, and fits the best line which minimizes the distances between the points and the line, rather than output probabilities. Thus, we need a new model whose results varies between 0 and 1. That's where the logistic regression comes in.

2.1.2 Sigmoid function

Instead of fitting a straight line, the logistic regression model uses the logistic function to squeeze the output of a linear equation between 0 and 1. The Logistic regression model is as follows:

$$h_w(x) = g(w^T x) \tag{1}$$

where x is the feature vector, and w is a vector with weights corresponding to each feature. $g(w^T x)$ is the Logistic function. The logistic function, also called the Sigmoid function, is defined as:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

By combining these two equations, we could get the hypothesis representation of Logistic Regression as follows:

$$h_w(x) = \frac{1}{1 + e^{-w^T x}} \tag{3}$$

Then we fit the parameters to our data and then use the parameters we get to calculate the estimated probability that $y = 1$, $h_w(x) = P(y = 1|x; w)$. For example, the problem of tumor ,for a given x, if we get the result $h_w(x) = 0.8$, it means that there's 80% chance that the tumor is malignant.

2.1.3 Cost function

From previous introduction, if we know that there are two possibilities $y = 1$ and $y = 0$, then we have:

$$P(y = 1|x; w) = h_w(x) \tag{4}$$

$$\begin{aligned} P(y = 0|x; w) &= 1 - P(y = 1|x; w) \\ &= 1 - h_w(x) \end{aligned} \tag{5}$$

Combine these two, we have:

$$P(y|x; w) = (h_w(x))^y (1 - h_w(x))^{1-y} \tag{6}$$

To facilitate the calculation, take the logarithm of both sides and get the following equation:

$$\begin{aligned} log(P(y|x; w)) &= ylog(h_w(x)) \\ &+ (1 - y)log(1 - h_w(x)) = L(w) \end{aligned} \tag{7}$$

Use the Maximum likelihood estimation, we have:

$$L(w) = \prod_{i=1}^{m} (h_w(x^i))^{y^i} (1 - h_w(x^i))^{1-y^i} \quad (8)$$

Divide by m to control the scaling of the cost function:

$$\mathcal{C}(w) = -\frac{1}{m} \left[\sum_{i=1}^{m} y^i \log h_w(x^i) + (1 - y^i) \log(1 - h_w(x^i)) \right] \quad (9)$$

2.1.4 Raphson Newton descent

In numerical analysis, Newton's method is a method for finding successively better approximations to the roots (or zeroes) of a real-valued function. Generally, Raphson Descent for finding a minimum is done by

$$\hat{\theta}_i = \hat{\theta}_{i-1} - \left(\frac{\partial^2 \mathcal{C}(\hat{\theta}_{i-1})}{\partial \hat{\theta}_{i-1} \partial \hat{\theta}_{i-1}^T} \right)^{-1} \frac{\partial \mathcal{C}(\hat{\theta}_{i-1})}{\partial \hat{\theta}_{i-1}} \quad (10)$$

The inverse term $\frac{\partial^2 C\theta}{\partial \theta^2}$ does not necessarily exist, since it might not be possible to invert it, and it can be expensive to calculate numerically. To improve stability and computational speed we can replace the inverted jacobian with some parameter η called the learning rate. When doing so we instead refer to this type of descent as gradient descent.

2.1.5 Gradient Descent

Gradient Descent is a very useful tool which can be used in almost all the machine learning algorithms. It is an iterative method used to find the values of parameters of a function that minimizes the cost function. A gradient is the slope of a function, the more gradient, the steeper the slope. In the above part, we have already got the Cost function. What we need to do now is to use the Gradient descent to find the parameters that minimizes the Cost function. The algorithm is as follows:

$$w_{ij} = w_{ij} - \eta \frac{\partial}{\partial w_{ij}} \mathcal{C}(w) \quad (11)$$

Where η is the learning rate, which is usually between 0 and 1, and it can control the size of each step, and if we go too far, maybe we get close to the perfect point, and then we cross the next step, it may not converge; If we go too slow, we will have to iterate many times before it converge. w_{ij} is the weights. According to the basic calculus theory, the gradient is a vector, and the direction of the gradient is the direction in which the function grows the fastest, if we want to find the optimal solution, you need to find the direction in which the Cost function decays the fastest. Then what we need to do is to adjust the parameters along the opposite direction of the gradient and that is the above function.

Then we simplify $\frac{\partial}{\partial w_j}\mathcal{C}(w)$ as follows:

$$\begin{aligned}
&\frac{\partial}{\partial w_j}\mathcal{C}(w) \\
&= \frac{\partial}{\partial w_j}\left[-\frac{1}{m}\sum_{i=1}^{m}\Big[-y^i log(1+e^{-w^T x^i}) \right. \\
&\qquad \left. - (1-y^i) log(1+e^{w^T x^i})\Big]\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[-y^i\frac{-x_j^i e^{-w^T x^i}}{1+e^{-w^T x^i}} \right. \\
&\qquad \left. - (1-y^i)\frac{x_j^i e^{w^T x^i}}{1+e^{w^T x^i}}\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\frac{y^i x_j^i - x_j^i e^{w^T x^i} + y^i x_j^i e^{w^T x^i}}{1+e^{w^T x^i}} \qquad (12) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\frac{y^i(1+e^{w^T x^i}) - x_j^i e^{w^T x^i}}{1+e^{w^T x^i}} \\
&= -\frac{1}{m}\sum_{i=1}^{m}(y^i - \frac{e^{w^T x^i}}{1+e^{w^T x^i}})x_j^i \\
&= -\frac{1}{m}\sum_{i=1}^{m}(y^i - \frac{1}{1+e^{-}w^T x^i})x_j^i \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^i - h_w(x^i)\right]x_j^i \\
&= \frac{1}{m}\sum_{i=1}^{m}\left[h_w(x^i) - y^i\right]x_j^i
\end{aligned}$$

Each parameter are updated over and over again, updating it with equation (12), which is to subtract the learning rate η by itself and multiply it by the following differential term until we get the minimum solution of our cost function.

3 Neural Networks

3.1 Basic Idea

Artificial neural networks are a set of algorithms which are modeled after the human brain. The input data is passed through the nodes that contain activation functions before they are passed on to the nest layer nodes and we eventually use the output from the output nodes.

3.2 Initialising weights and biases

Because functions like tanh and the sigmoid are very flat for input values much larger and much smaller than zero they have very small gradients for these values. This causes the gradient descent methods that we use to only make very small updates to the weights in the network. To avoid very large values in the input layer it is important to normalize the input values of the features. In addition, we also need to initialise our weights in a clever way depending on the number of nodes in different layers. The method we use is Kaiming initialisation [2]. In the Kaiming method we assume we use ReLu as our hidden layers. We initialise the weights by picking a number randomly from the normal distribution with mean 0 and variance 1. We then multiply these weights with the standard deviation in the output layer as a result of the cumulative effect of the weights, which proves to be

$$\mathrm{Var}(\hat{a}_{out}) = \sqrt{\frac{N_{Inputs}}{2}}. \qquad (13)$$

This method prevents uncontrolled weight growth which can lead to overflow numerically or overfitting. However, another problem is weights dying off (weights where the derivative is 0, preventing learning). This in turn can be prevented by properly centering and normalising the data. In the case of the biases we simply choose a small number, in our case we chose 0.0001.

3.3 Activation functions

Activation functions defines the output of a node when an input is given, in order to fit non-linear functions we need to introduce some non-linearity to the neural network with non-linear activation functions like the sigmoid function. Logistic Sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}} \qquad (14)$$

Hyperbolic tangent function:

$$f(x) = \tanh x \tag{15}$$

Linear function:

$$f(x) = w^T x \tag{16}$$

ReLu:

$$f(x) = max\{x, 0\} \tag{17}$$

It gives an output x if x is positive and gives 0 otherwise.

3.4 SoftMax function

In the above two-class classifier problem, we have used the Sigmoid function to map the input $-\theta^T x$ to the interval of (0,1) to get the probability of a specific category. When it comes to multi-class problem, we need to use SoftMax funtion to normalize the output value to a probability value. For a given feature vector x, if we want to figure out the probability for each of these categories i, $P(y = i|x; \theta)$, then the results of our hypothesis function would be a C dimensional vector whose sum of the vector elements is 1, represents estimated probability values of these C types. So for each sample, the probability that it belongs to category k is:

$$a_k^L = \frac{e^{z_k^L}}{\sum_{n=1}^{C} e^{z_n^L}} \tag{18}$$

where $\forall i \in 1...C$ Take the derivative of above equation, if $k = j$ we have:

$$\begin{aligned}\frac{\partial a_j}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{e^{z_j}}{\sum_{n=1}^{C} e^{z_n}}\right) \\ &= \frac{(e^{z_j})\prime \cdot \sum_{n=1}^{C} e^{z_n} - e^{z_j} e^{z_j}}{(\sum_{n=1}^{C} e^{z_n})^2} \\ &= \frac{e^{z_j}}{\sum_{n=1}^{C} e^{z_n}} - \frac{e^{z_j}}{\sum_{n=1}^{C} e^{z_n}} \frac{e^{z_j}}{\sum_{n=1}^{C} e^{z_n}} \\ &= a_j(1 - a_j)\end{aligned} \tag{19}$$

If k ≠ j, we have

$$\begin{aligned}\frac{\partial a_k}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{e^{z_k}}{\sum_{n=1}^{C} e^{z_n}}\right) \\ &= \frac{0 \cdot \sum_{n=1}^{C} e^{z_n} - e^{z_k} e^{z_k}}{(\sum_{n=1}^{C} e^{z_n})^2} \\ &= -\frac{e^{z_k}}{\sum_{n=1}^{C} e^{z_n}} \frac{e^{z_k}}{\sum_{n=1}^{C} e^{z_n}} \\ &= -a_k a_j\end{aligned} \tag{20}$$

Putting equations above together, we get:

$$\frac{\partial a_k}{\partial z_j} = a_j(\delta_{kj} - a_k) \tag{21}$$

3.5 Non-linear functions

Non-linear functions are used extensively as activation functions. The function Rectified Linear Unit (ReLu) is of particular interest. This function is simply given by:

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x < 0 \end{cases} \tag{22}$$

with a simple derivative

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x < 0. \end{cases} \tag{23}$$

Using activation functions which are nonlinear allows us to capture features which cannot otherwise be captured using only linear functions, such as is the case in linear regression.

3.6 Feed forward algorithm

Feed forward neural networks first take the input data, multiply them with weights and add biases and then pass it to a "neurons" or "nodes" in the first layer. Every neuron in the layer has an activation function. In this project, all the nodes in the same layer have the same activation function and we also have fully connected layers which means that every node in each layer passes its output to every node in the next layer. The connections only pass/feed forward and do not form cycles. When the output is passed to a node in the next layer, it is again multiplied by a weight and a bias is added. This continues through as many hidden layers as we have in the

neural network until we reach the output layer. In classification problems we have the same number of nodes in the last layer as the number of classes, so that each class corresponds to one output node. We then want the output to be normalized so that the output of all the output nodes equals to one, so that we can interpret the single output of one of the nodes as the probability of the sample that we put into the neural network to be that specific class.

As shown in the following figure, the leftmost layer in this network is called the input layer, and the rightmost layer is the output layer. The middle layer is called a hidden layer.

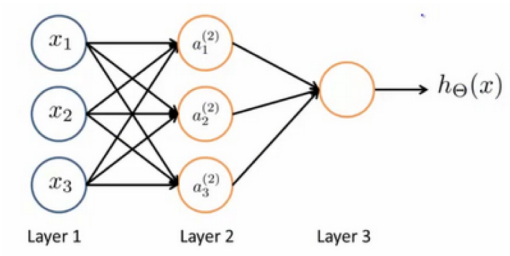


Figure 1: Generic Network with Connections [3]

Where a_i^j is the activation of unit i in layer j, w^j is a matrix of weights controlling function mapping from layer j to layer to layer j+1 and b_j is a vector of biases in layer j
Dimension: the matrix with the number of activation units at the j+1 layer as the number of rows and the number of activation units at the j layer as the number of columns. For example,θ^1 means the weights of matrix controlling function mapping from layer 1 to layer 2. θ^1 is a matrix of 3×3 Dimensions.

Thus, from the above figure, the activation units and the output are shown as follows:

$$y_i^l = f(z_i^l) = f(\sum_{j=1}^{M} w_{ij}^l x_j + b_i^l) \tag{24}$$

$$a_2^1 = f(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1) \tag{25}$$

$$a_2^2 = f(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2) \tag{26}$$

$$a_2^3 = f(w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3) \tag{27}$$

$$\begin{aligned} h_w(x) &= a_1^3 \\ &= f(w_{11}^2 a_1^2 + w_{12}^2 a_2^2 + w_{13}^2 a_3^2 + b) \end{aligned} \tag{28}$$

where w_{ab}^j are the b_{th} parameters in layer j when we calculate the a_{th} parameters.

3.7 Cost function

To quantify the results we get, we need to define a cost function like what we did in the logistic regression

$$\mathcal{C}(w,b) = -\frac{1}{2m}\sum_x ||y(x) - a||^2 \tag{29}$$

where, w are the weights of the network, b are the biases, m is the total number of training inputs, a is the vector of outputs from the network when x is the input, and the sum is over all training inputs x. Similarly, we use maximum likelihood estimation and then we get :

$$\begin{aligned} \mathcal{C}(w) = -\frac{1}{m}\Bigg[\sum_{i=1}^{m} y^i \log h_w(x^i) \\ + (1-y^i)\log(1-h_w(x^i))\Bigg] \end{aligned} \tag{30}$$

3.8 Cost function regularization

In linear regression we can use normalization as a means of scaling the principal components with least variance away, leaving the components which contributes the most. Likewise we can use regularization to do the same for logistic regression and neural networks. There are primarily two kinds which are used: L1 and L2 regularization. L1 and L2 respectively correspond to LASSO and Ridge regression. When

we use Regularization we add it to the cost function as a factor. For L1:

$$\mathcal{C}(\mathbf{W})_{L_1} = \mathcal{C}(\mathbf{W}) + \lambda||\mathbf{W}||_1 \tag{31}$$

For L2:

$$\mathcal{C}(\mathbf{W})_{L_2} = \mathcal{C}(\mathbf{W}) + \lambda||\mathbf{W}||_2^2 \tag{32}$$

So, if we use add L2 regularization into our Cost function, we will get:

$$\begin{aligned}\mathcal{C}(w) = -\frac{1}{m}\Bigg[\sum_{i=1}^{m}\sum_{k=1}^{k} y^i \log h_w(x^i) \\ + (1-y^i)\log(1-h_w(x^i))\Bigg] \\ + \frac{1}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_l+1}(w_{ji}^l)^2\end{aligned} \tag{33}$$

This equation is for a whole batch of m input samples. The first term is the normal cross-entropy loss function. The second term is a regularization term. It increases the cost function to "punish" large values for weights. It's called L2 weight-decay (L2 means it's using the square value, L1 means absolute value). This is done to prevent overfitting, makes it more difficult for the network to learn noise from input samples.

The summation symbol is just another way of writing a for-loop so what the three summations means for each layer l and for every pair of connecting units i and j, take the square of the weight from j to i, then add everything together. The lambda parameter is used to control how much regularization we want, a large lambda value means that we want our network to be very regularized, a small lambda means a small amount of regularization.

3.9 Stochastic Gradient Descent

Stochastic Gradient Descent is especially useful when handling big data sets, by reducing the computational burden. The process is linked with a random probability, hence, a few samples are selected randomly instead of the whole dataset for each iteration. Generally, machine learning considers the problem of minimizing an objective function that has the form of a sum:

$$C(w,(x_i,y_i)) = \frac{1}{2}\sum_{i=1}^{m}(h_w(x^i)-y^i)^2 \tag{34}$$

where $C(w,(x_i,y_i))$ is the cost function with the i-th data in input dataset. To minimize the above function, we often use the gradient descent method to solve this problem, which is what we did previously. However, evaluating the sum-gradient often require expensive evaluations of the gradients from all summed functions, which require expensive computations thus we use the Stochastic Gradient Descent method:

$$w = w_j - \eta(h_w(x^i)-y^i)x_j^i \tag{35}$$

where η is the learning rate, which is usually between 0 and 1. The stochastic gradient descent algorithm updates the parameter after each calculation, without the need to sum up all the training sets first. So SGD can be a lot faster than normal gradient descent. Stochastic gradient descent is also less likely to get stuck in local minimum compared to normal gradient descent.

3.10 Backpropagation

The Backpropagation algorithm is used to adjust the weights and bias of the neural network. Going backwards from the prediction we get from the last layer in the network. Based on the prediction $\mathbf{y}^L$ from the last layer we will get a value for the cost function. What we want now is to find out how the cost function behaves depending on the weights corresponding to the last layer. From the chain rule we get:

$$\frac{\partial\mathcal{C}(\hat{W^L})}{\partial w_{jk}^L} = f'(\hat{z}^L)\circ\frac{\partial\mathcal{C}}{\partial(\hat{a}^L)}\hat{a}_k^{L-1}. \tag{36}$$

where L denotes that we are looking at the last layer. If we then define δ as

$$\hat{\delta}^L = f'(\hat{z}^L) \circ \frac{\partial \mathcal{C}}{\partial(\hat{a}^L)} \tag{37}$$

we will get:

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1} \tag{38}$$

for the biases we get:

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L} \tag{39}$$

Replacing the L for the last layer with a general l for a hidden layer we get:

$$\begin{aligned} \delta_j^l &= \frac{\partial \mathcal{C}}{\partial z_j^l} \\ &= \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} \end{aligned} \tag{40}$$

Using the chain rule on the last partial derivative we get:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{41}$$

With these equations we can then after using our cost function in the last layer calculate the derivative of the cost function with respect to all the weights and biases in our neural network, which we will use to in our gradient descent method to update the all the weights and biases.

3.11 Cross Entropy

Cross-entropy is a measure which is commonly used in machine learning, building upon entropy and generally calculating the difference between two probability distributions. The cross-entropy function between two probability distributions, p and q, can be written as $H(p, q)$.

$$H(p, q) = \sum_{x \in \mathcal{X}} p(x) log(q(x)) \tag{42}$$

Where p is the target distribution and q is the approximation of the target distribution, both follow a distribution with parameter p from 0 to 1, $X \sim B(1, p)$. In order to get get our cost function in neural network we need to use maximum likelihood estimation:

$$p(t|x) = \prod_{i=1}^{C} p(t_i|x)^{t_i} = \prod_{i=1}^{C} y_i^{t_i} \tag{43}$$

Where $p(t|x)$ is the maximum likelihood estimate, t_i are the labels of the dataset. Due to the problem that continuous multiplication may lead the final result to approach to 0, the likelihood function is generally taken as the negative number of logarithm and that is the minimum logarithm likelihood function:

$$-\log p(t|x) = -\log \prod_{i=1}^{C} y_i^{t_i} = -\sum_{i=1}^{C} t_i \log(y_i) \tag{44}$$

Thus the log-liklihood cost function becomes:

$$\mathcal{C} = -\sum_i t_i \log a_i \tag{45}$$

For finding the weight and biases, we use:

$$w_{ij}^l = w_{ij}^l - \eta \frac{\partial}{\partial w_{ij}} \mathcal{C}(\hat{W}) \tag{46}$$

$$b_j^l = b_j^l - \eta \frac{\partial}{\partial b_j} \mathcal{C}(\hat{b}) \tag{47}$$

$$\frac{\partial \mathcal{C}}{\partial w_{ij}} = \sum_{kl} \frac{\partial \mathcal{C}(\hat{W})}{\partial a_k}\frac{\partial a_k}{\partial z_l}\frac{\partial z_l}{\partial w_{ij}} \tag{48}$$

$$\frac{\partial \mathcal{C}}{\partial b_j} = \sum_{kl} \frac{\partial \mathcal{C}(\hat{b})}{\partial a_k}\frac{\partial a_k}{\partial z_l}\frac{\partial z_l}{\partial b_j} \tag{49}$$

$\frac{\partial \mathcal{C}(\hat{W})}{\partial a_k}\frac{\partial a_k}{\partial z_l}$ and $\frac{\partial \mathcal{C}(\hat{b})}{\partial a_k}\frac{\partial a_k}{\partial z_l}$ are the same, we put in the log-likelihood function:

$$\begin{aligned}
&= \sum_k \frac{\partial}{\partial a_k}\left[-\sum_i t_i \log a_i\right]\frac{\partial a_k}{\partial z_l} \\
&= \sum_k -\frac{t_k}{a_k}\frac{\partial a_k}{\partial z_l} \\
&= -\sum_k \frac{t_k}{a_k}\frac{\partial}{\partial z_l}\left(\frac{e^{z_k}}{\sum_{n=1}^{C} e^{z_n}}\right) \\
&= -\sum_k \frac{t_k}{a_k} a_l(\delta_{kl} - a_k) \\
&= -\frac{t_l}{a_l}a_l(1-a_l) - \sum_{k\neq l}\frac{t_k}{a_k}a_l(-a_k) \qquad (50) \\
&= -t_l(1-a_l) + \sum_{k\neq l}\frac{t_k}{a_k}a_l a_k \\
&= -t_l + t_l a_l + \sum_{k\neq l} t_k a_l \\
&= -t_l + \sum_k t_k a_l \\
&= -t_l + a_l \sum_k t_k \\
&= a_l - t_l
\end{aligned}$$

Inserting result we got from (50) into equation (48) and (49), we get:

$$\begin{aligned}
(a_l - t_l)\frac{\partial z_l}{\partial w_{ij}} &= (a_l - t_l)\frac{\partial(w_{lk}a_k + b_l)}{\partial w_{ij}} \\
&= (a_l - t_l)\delta_{ik}\delta jl a_k \qquad (51) \\
&= (a_j - t_j)a_i
\end{aligned}$$

$$\begin{aligned}
(a_l - t_l)\frac{\partial z_l}{\partial b_j} &= (a_l - t_l)\frac{\partial(w_{lk}a_k + b_l)}{\partial b_j} \\
&= (a_l - t_l)\delta_{lj} \qquad (52) \\
&= (a_j - t_j)
\end{aligned}$$

This is the backwards propagation algorithm for softMax and cross-entropy.

3.12 MSE as cost function

For regression problems it is common to use the mean square error as a cost function. Repeating the steps we did for cross-entropy and softMax for classification we can do the same for MSE and an activation function for the final layer which returns its input:

$$f(x) = x. \qquad (53)$$

Now considering the δ_k^L we get:

$$\begin{aligned}
\delta_l^L &= \frac{\partial \mathcal{C}(\hat{W})}{\partial a_k}\frac{\partial a_k}{\partial z_l} \qquad (54) \\
&= 0.5\frac{\partial}{\partial a_k}\left(\mathbf{y} - \mathbf{a}^L\right)^2 \delta_{kl} \qquad (55) \\
&= (y_l - a_l^L) \qquad (56)
\end{aligned}$$

Which incidentally is exactly the same as equation 50.

4 Principal Component Analysis

Principal Component Analysis(PCA) is the most popular dimensionality reduction algorithm, it is a statistical procedure used to emphasize variation and bring out strong patterns in a dataset. It identifies the hyperplane that lies closest to the data and then it projects the data onto it [1]. When using this method it is important to first center the dataset. Generally, PCA seeks a linear combination of variables such that eigenvectors with the least variance is removed from the dataset. The eigenvector with the largest eigenvalue represents the direction in which the dataset has the largest variance. Analytical methods, algorithms and implementation

5 Preprocessing of the dataset

5.1 Data pruning

The first step of preprocessing the data is to remove the data samples that have non permissible values. Like the categorical variable X2(SEX) that can only take the values 1 and 2. If we get a blank space or another value than this we drop the whole data sample, instead we could have made a third category other and just put all entries that are not allowed into that category. Some features may also have defined boundaries like the credit limit X1 that must be a number greater than 0.

5.2 Column scaling

Scaling of the data is necessary when variables span different ranges, variables that are measured at different scales do not contribute equally to the analysis. There are many different ways of scaling the data, here we use standardization. The data are scaled so that the results has a zero mean and unit variance. For every column that needs scaling, the mean μ is subtracted and then the column is divided by the standard deviation σ.

$$Y = \frac{X - \mu}{\sigma} \tag{57}$$

Where Y is the standardized data and X is the original data.

5.3 One-Hot Encoding

In our dataset, we have some categorical data, for example, column X2 which represents Sex. However, machine learning algorithms require all input variables and output variables to be numeric, it does not work well on labeled data directly. To prevent this, we use One-Hot encoding. Column X2, X3, X4, X6, X7, X8, X9, X10 classifies a person in different way, column X2 which is the gender are presented with 1 = male, 2 = female.

$$\begin{bmatrix} Male & 1 \\ Female & 2 \\ Female & 2 \\ Male & 1 \\ Female & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} Male & [1,0] \\ Female & [0,1] \\ Female & [0,1] \\ Male & [1,0] \\ Female & [0,1] \end{bmatrix}$$

From the above example dataset, the left side is the original data, the right side is the data after it is one-hot encoded, it creates an array reserving one element for each category.

6 Logistic Regression

When implementing the logistic regression we started out with using cross entropy as the cost function. Using this cost function we then attempt to fit a sigmoid function to give a probability distribution which then are used to make a guess (choose an outcome such as 0 and 1, based on the probability at any point) which are then used as a guide for how well the fit are.

For linear regression it is possible to find an analytical solution to the problem, where the optimal solution is given in one operation. For logistic regression, however, such a straightforward solution is only possible for two classes [4], there are no solution for the general case, so we need to use some gradient solver to iterate our way to the global minimum. For doing so we implemented three different solvers with different strengths, Newton Raphson, gradient descent and stochastic descent. The most mathematically involved method is the Newton Raphson method from equation 10. The implementation of the first derivative is

```
dC = -X.T @ (y-p)
```

Listing 1: Python code for first derivative of cost function in Newton Raphson

and the second derivative

```
w = p * (1 - p)
for i in range(number_of_inputs):
    for j in range(
            number_of_classes):
        mat1 = X.T[i][:]
        mat2 = w * X.T[:][j]
        Jacobian[i][j] =
            mat1 @ mat2
ddC = Jacobian
```

Listing 2: Python code for the second derivative of the cost function.

Assuming the inverse exists it will select the direction and step size to make optimally sized steps towards the minimum. However, this method can be costly to calculate and we are not ensured the inverse jacobian exists. The code for the resulting Newton Raphson is

```
increment = inv(ddC) @ dC
beta = beta - increment
```

Listing 3: Python code for the implementation of Newton Raphson

To avoid the troubles of an expensive or unstable jacobian we can instead use the standard Gradient Descent method implemented as

```
    beta = beta - eta * dC
```

Listing 4: Python code for Gradient Descent.

Finally, we implemented a stochastic gradient descent method. This method has a better chance of avoiding local minima and especially for larger datasets should be faster since it only looks at a portion of the dataset at a time.

```
    for i in range(
    number_of_minibatches):
        indices = np.random.choice(
                number_of_datapoints,

    number_of_points_per_set)
        dC = dC_func(X[indices], p[
    indices], y[indices])
        ddC = ddC_func(X[indices], p[
    indices])
        increment = ddC @ dC
        beta = beta - increment
```

Listing 5: Python code for implementation of stochastic gradient descent

This implementation uses the Newton Raphson method. We can instead use the standard Gradient Descent method and replace the Jacobian with a learning rate.

6.1 Implementing gradient descent solver

When implementing the gradient descent solver, we refer to equation (11). The gradient descent procedure is performed on weights and biases to approach the minimum. The gradient descent solver is implemented by:

```
for each in epoch:
    for layer in layer ListBackwards:
        delta = dCdz
        layer.W -= eta * dot(a[layer
    -1], delta)
        layer.beta -= eta * np.mean(
    delta)
```

Listing 6: Python code for Gradient descent

The derivation of the cost function $dCdz$ is described below.

When implementing gradient descent with mini-batches we selected a set of random indices and fed these fewer samples into a loop.

```
    for each in epoch:
        data_indices = range(n_inputs)
        for mini in n_minibatches:
            chosen_datapoints = random
    .choice(data_indices)
            run_backpropagation()
            run_feed_forward()
```

Listing 7: Python code for Gradient descent with minibatches

7 Feed forward neural network

The feed forward neural network is implemented over a skeleton of layers. We initialise the network by giving the network a list of layers with information about the number of nodes, layer type and activation functions. This initialization is

```
    model = hBNN.homebrew([
        hBNN.layers(
            features = x,
            layerType="featureLayer"),
        hBNN.layers(
            outputNodes=nNodes1,
            activation=
    activationFunction1),
        hBNN.layers(
            outputNodes=nNodes2,
            activation=
    activationFunction2,
        ...,
        hBNN.layers(
            outputNodes=nOutputs,
            activation=
    outputActivation)])
    model.compile(
        loss = CostFunction,
        optimizer="gradientDescent",
        regularization="L2", _lambda=
    lambda,
        weightNormalizing=True)
```

Listing 8: Python code for initialising and compiling te neural network.

Once the network is initialised we can fit the model to some data. We do this using the fit function which needs the training target, the

learning rate, the number of epochs to train for and the method of backpropagation.

```
    model.fit(
        dataTrain,
        learningRate=learningRate,
        epochs=numberOfEpochs,
        backpropagation="normal")
```

Listing 9: Python code for fitting the neural network.

Once all of this is done we can call the history method to see how the network has performed during training or we can use the predict method to let the network perform a prediction on a dataset. Underlying all of this is the feed forward and backpropagation algorithms. The feed forward algorithm is simply implemented by

```
for Layer in layerList:
    z = dot(a[Layer-1], Layer.W) +
    Layer.b
    a[Layer] = f(z)
```

Listing 10: Python code for FFNN

which runs through all the different layers and adjusts the output a of each layer. We implemented the backpropagation algorithm for the activation functions ReLu, sigmoid, softmax and tanh are as follows:

```
    f = 1/(1+np.exp(-z))
```

Listing 11: Python code for sigmoid

```
    b = np.exp(z - np.max(z))
    a = np.divide(b, (np.sum(b, axis=1,
      keepdims=True)))
```

Listing 12: Python code for SoftMax

```
    z[a <=0] = 0
    z[a > 0] = 1
```

Listing 13: Python code for Relu

```
    return np.tanh(z)
```

Listing 14: Python code for tanh

To gain flexibility we made the derivative of the cost function with regards to z by using elementwise autograd on the cost function directly. The two kinds of cost functions are MSE and crossEntropy. The derivative of the cost function in the last layer are given as

```
    dCdz = egrad(C, 0)
    delta = dCdz(z, f, t)
```

Listing 15: Backpropagation

Alternatively we also have the option of initiating the last layer delta as

```
    delta = (t - a)
```

Listing 16: delta for MSE and linear regression and cross entropy and softmax.

but this is limited to the particular cases of MSE and linear regression and cross entropy and softmax. If we use anything different we would need to change this bit of code. Due to the increased flexibility of the elementwise autograd functionality, we preferred that solution.

7.1 Regression analysis using neural networks – terrain data

When using the neural network for regression analysis, we changed the cost function to be the MSE. Additionally we don't necessarily want to limit the possible output to be any particular distribution. We therefore use a simple identity function $f(x) = x$ as the activation function for the last layer. We had to reintroduce the design matrix which is used for linear regression.

The network we ended up with was constructed from two hidden layers with 64 nodes and with the ReLu activation functions. The last layer used no activation function and the MSE cost function.

8 Results and Discussions

8.1 Regression problem

Neural networks are well suited to perform linear regression. The basic algorithms are similar when the cost function are the mean

squared error and we omit using an activation function in the output layer. We will study how well the network works for linear regression and the effect several of the variable parameters have on the network. We will try the regression method on data of some terrain from over Norway. Finally, we will compare the results from the network with the results from a previous linear regression method.

8.1.1 Linear Regression on terrain data

We used the Neural Network for regression using terrain data over Norway. When we did this we first chose a part of the terrain data on which we will perform the regression. We also needed to split the data into a training, validation and test set. We did this by choosing half the points along the x-axis and y-axis respectively to be the train set, and further split the remaining half into a test and validation set. The train set is used for training, the validation set is used to consider the model as it is being trained, while the test set is used at the end as a last check for how well our model is doing.

We then performed a grid search to try and determine the most promising learning rate and regularization factor, using L2 regularization. The best learning rate is around 0.1 for both stochastic regression and normal gradient descent, while the regularization factor is around 0.05 for both. From figure 3d we see that the validation score is worse than the training score for low training rates in particular, indicating the model is under trained.

After finding optimal training rate and regularization factor we used these values to produce a final model using more epochs to produce an optimal fit. From figure 3 we see that the stochastic gradient descent require a lower training rate to reach a good value for the cost function. Likewise from figure 4 and 5 we notice a tendency for the gradient descent iterator to get stuck on a plateau, while the stochastic gradient descent quickly reach close to a minimum with a comparable value for the cost function. This is expected behaviour to a degree in that the gradient descent solver is vulnerable to get stuck in local minima without any easy way to get back out to converge on the global minima. The stochastic solver gets around this problem by 'smoothing' local minima away by choosing fewer points randomly on the whole data set. Furthermore stochastic regression achieves a higher R2 score than gradient descent while avoiding a tendency for over training we can start to notice in the gradient descent plots.

From figure 6 we can see the neural network have created a new image based on the test data set. While the prediction image roughly resembles the test data set there is still struggles to capture details.

The final cost for the test sets are seen in table 1 and taken together with figure 4 and 5 it seems the stochastic gradient descent finds the global minima faster and more reliably than the normal gradient descent solver.

	GD	SGD
MSE	0.0373	0.0381

Table 1: (Gradient descent (GD) vs Stochastic Gradient Descent (SGD).

Comparing the figures 31a and 31b, we see that Stochastic Gradient Descent method has a better performance than Normal Gradient Descent – Stochastic Gradient Descent method can get a higher R^2 score than Normal Gradient Descent method and save computation time at the same time. In order to get a higher R^2 scores, Normal Gradient Descent need over thousand epochs while Stochastic Gradient Descent method need only 50.

8.1.2 Comparison between linear regression and neural network.

When testing a segment of the terrain data using stochastic gradient descent, 200 epochs,

0.02 learning rate and regularization of 0.02 we got an MSE of 0.017 and R2 score of 0.85. Compared to ridge regression, where we ended up with MSE of 0.008 and R2 score of 0.9905, Ridge regression did better than our network.

But sklearn NN with 5 layers with 100, 80, 60, 40, 20 nodes performed even better with a mse=0.002028 on test data and an R2score=0.9910. From this it seems that neural networks perform well and can pick up details and features regression cannot, but that our implementation are lacking.

8.2 Classification problem

When doing classification, a regression method which has been used for a while is the logistic regression method. This method use cross entropy as a cost function and relies on the logistic function to return values between 0 and 1. This approach is transferable to neural networks, which makes the network suited to classification problems. We will first look at classification with logistic regression and then move on to classification using neural networks and compare the two approaches. The data we will study in this part is the Taiwan credit card dataset [5].

8.2.1 The Taiwan credit card dataset

This dataset employed a binary variable, default payment (Yes = 1, No = 0), as the response variable, and used the following 23 variables as explanatory variables: [5]

- X1: Amount of the given credit (NT dollar): it includes both the individual consumer credit and his/her family (supplementary) credit.
- X2: Gender (1 = male; 2 = female).
- X3: Education (1 = graduate school; 2 = university; 3 = high school; 4 = others).
- X4: Marital status (1 = married; 2 = single; 3 = others).
- X5: Age (year).
- X6 - X11: History of past payment. We tracked the past monthly payment records (from April to September, 2005) as follows: X6 = the repayment status in September, 2005; X7 = the repayment status in August, 2005; . . .;X11 = the repayment status in April, 2005. The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; . . .; 8 = payment delay for eight months; 9 = payment delay for nine months and above.
- X12-X17: Amount of bill statement (NT dollar). X12 = amount of bill statement in September, 2005; X13 = amount of bill statement in August, 2005; . . .; X17 = amount of bill statement in April, 2005.
- X18-X23: Amount of previous payment (NT dollar). X18 = amount paid in September, 2005; X19 = amount paid in August, 2005; . . .;X23 = amount paid in April, 2005.

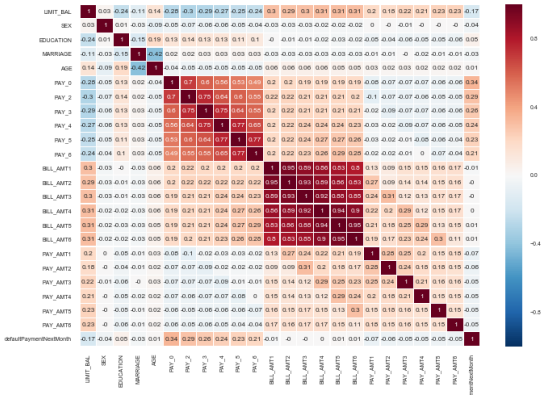


Figure 8: correlation

In figure 8(an enlarged figure added in appendix), we can clearly see the correlation between each variables. In the bottom line,

we can see that features pay_0 and pay_2 to pay_6 are the most relevant ones to default payment next month.
To study the dataset, we did different manipulations to the dataset and used tensorFlow to get some intuition about the dataset and reliable results.

Modification	Accuracy	Loss
Original dataset	0.824	0.424
With Normalized features	0.826	0.425
Only columns PAY_x	0.774	0.536
Using PCA	0.786	0.491

Table 2: Comparison of different manipulation to dataset using tensorFlow, PAY_x = pay_0 and pay_2 to pay_6

From the above table we could see that Normalization have effect on improving Accuracy but is not that obvious in this case. PCA method could remain the main characters of data set and save the computation at the same time. The most related features indeed make the main contribution to the data.

8.2.2 Classification with Logistic Regression on credit card data

For the classification problem we implemented code for logistic regression. We implemented the Newton-Raphson, gradient descent and stochastic gradient descent method. We compared these methods with an SKLearn classifier, a straight one guess, a straight zero guess and a 50/50 guess. We ran the regression methods with learning rate 0.005 and for 10 epochs. This resulted in the prediction rates seen in table below:

Method	Prediction
SKLearn	0.78
Newton Raphson	0.81
gradient descent	0.78
Stochastic descent	0.81
All zeros	0.78
All ones	0.22
Random 1 or 0	0.51

Table 3: Comparison of different logistic methods

From these results we see that using our solver with simple gradient descent gets the same accuracy as SKlearn. Both of these seem to get stuck in a local minimum that does not perform better than guessing all predictions to be zero. When we use Newton Raphson method or SGD instead, we are able to avoid it and get a considerably better accuracy of roughly 81%

We then used only these 6 selected features as input data in both SKLearn and home made logistic regression and got the following results:

Method	Prediction
SKLearn LogReg	0.806
HomeMade LogReg(SGD)	0.773
HomeMade LogReg(GD)	0.748
HomeMade LogReg(NR)	0.766
HomeMade NN(GD)	0.792
HomeMade NN(SGD)	0.790

Table 4: Comparison of different logistic methods using only selected features, SGD = Stochastic Gradient Descent, GD = Gradient Descent, NR = Newton Raphson

From the above table, we could find that these 6 selected features keep the main characters of the credit card data; The neural network model has a better performance than the Logistic Regression model. In the later work,

we will use both models again to fit the whole data set.

8.2.3 Classification with Neural Network

In this part, we use the Feed Forward Neural Network to implement the back propagation algorithm. In order to get the best results, we use four different combinations to get our Cost function:
1) softMax function + MSE with gradient descent
2) softMax function + MSE with stochastic gradient descent
3) softMax function + Cross-Entropy with gradient descent
4) softMax function + Cross-Entropy with stochastic gradient descent
And then we get the results as follows:

Area ratio was calculated from equation: [6]

$$A = \frac{A_{cb}}{A_{bcb}} \tag{58}$$

where A is area ratio, A_{cb} is area between curve and baseline curve and A_{bcb} is area between theoretically best curve and baseline curve.

Comparison of accuracy between different cost functions and gradient descent algorithms using full data set:

First of all, we combined the Softmax with MSE using gradient descent method to get our cost function. As showing in figure 9a, we calculate the accuracy of the train data and the validation data and show how they vary with the increase of the number of epochs. At epoch 500, we get that the accuracy on the test set is 0.8057.

Secondly, we combined the Softmax with MSE using stochastic gradient descent method to get our cost function. As showing in figure 9b, the accuracy on test set is 0.8249 at the epochs 17.5.

Thirdly, we combined the Softmax with cross-Entropy using gradient descent method to get our cost function. In figure 10a, we get that the predict on test set is 0.8161 at the epochs 500.

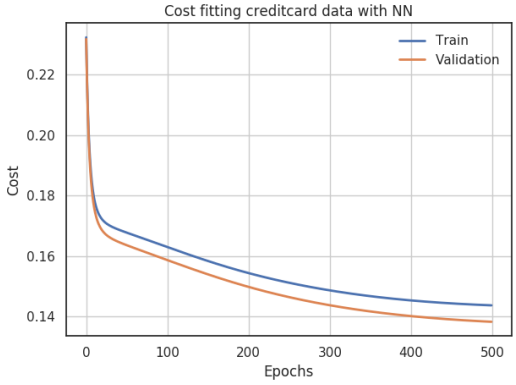
Finally,we combined the Softmax with cross-Entropy using stochastic gradient descent method to get our cost function. In figure 10b, we get the predict on test set is 0.8170 at the epochs 17.5.

Compare these four types of combinations, we could find that combination 2 is the best; Stochastic Gradient Descent method could improve the accuracy and save the computation at the same time.

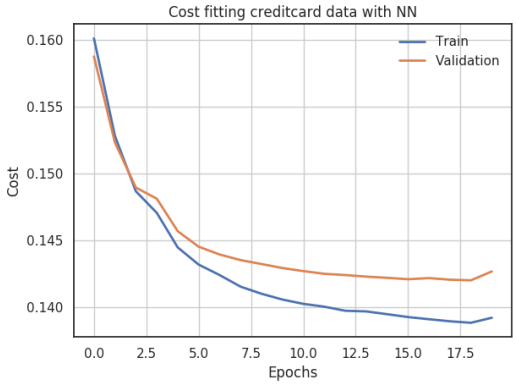
Comparison of cost between different cost functions and gradient descent algorithms using the full dataset:

	GD		SGD	
	sMax + MSE	sMax + xE	sMax + MSE	sMax + xE
Area ratio	0.4394	0.5059	0.47	0.4805
Accuracy	0.8057	0.8161	0.8249	0.8170
best η	1	1.71	1.71	1.42
best λ	0.1	0.0667	0.0333	0.05

Table 5: Table showing how well different settings did on the credit card data. GD is Gradient Descent, SGD is Stochastic Gradient Descent, sMax is softMax, MSE is Mean Squared Error and xE is cross-Entropy.



(a) MSE with gradient descent



(b) MSE with stochastic gradient descent

Figure 11: Comparing gradient descent with stochastic gradient descent for MSE cost function 1

We then calculated the loss of the train data and the validation data and drew pictures to show how they varies with the increase of the epochs. First of all, we combine the Softmax with MSE using gradient descent method to get our cost function. As we can see in the figure 11a, the cost of the train data and the validation data are close to 0.14 at the epochs 500.

Secondly, we combined the Softmax with MSE using stochastic Gradient descent method to get our Cost function.As we can see in the figure 11b, the cost of the train data and the validation data are close to 0.14 at the epochs 17.5.

And then, we combined the Softmax with crossEntropy using Gradient descent method to get our Cost function. In figure 12a, the cost of the train data and the validation data are close to 0.46 finally.

Finally, we combined the Softmax with crossEntropy using stochastic Gradient descent method to get our Cost function. In figure 12b, the cost of the train data and the validation data are close to 0.45 at the end.

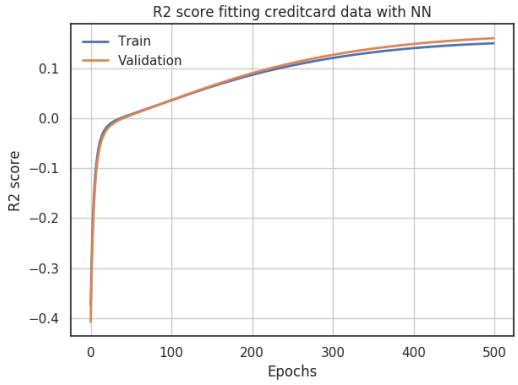
Compare these four types of combinations, we could find that combination 2 is the best; Stochastic Gradient Descent method could reduce the loss and save the computation at the same time.

Comparison of cumulative curve between different cost functions and gradi-

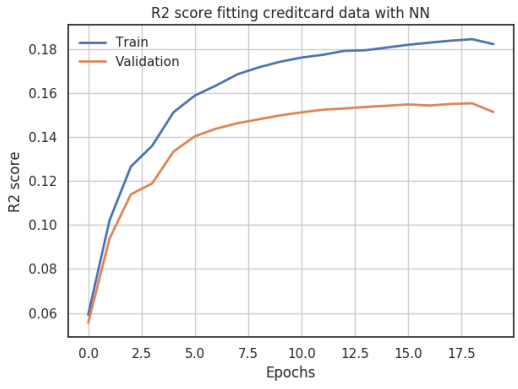
ent descent algorithms using full dataset:

In figure 13 and 14 we reproduced the Neural Network analysis of the scientic article, and draw the same type of figure as we see section 3.2 Results of that scientific paper.[6]

R^2 comparison between different cost functions and gradient descent algorithms:

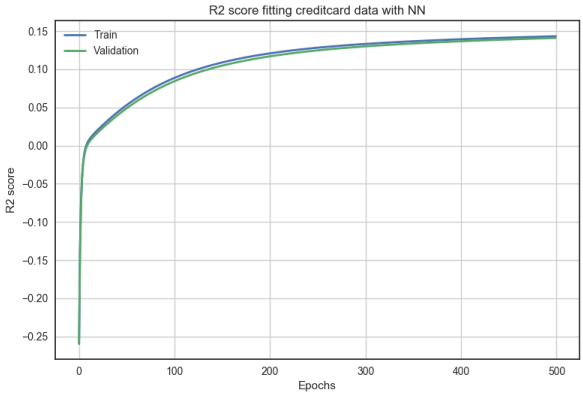


(a) MSE with gradient descent

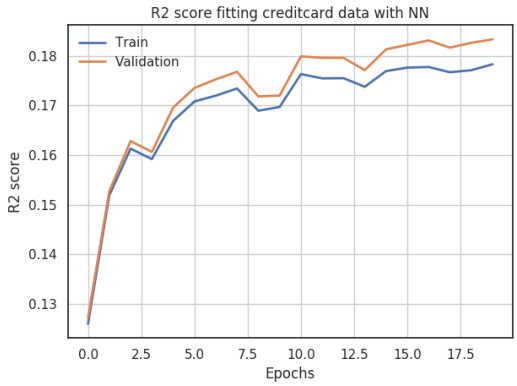


(b) MSE with stochastic gradient descent

Figure 15: Comparing gradient descent with stochastic gradient descent for MSE cost function 1



(a) Cross-entropy with normal gradient descent



(b) Cross-entropy with stochastic gradient descent

Figure 16: Comparing gradient descent with stochastic gradient descent for cross-Entropy cost function 2

In the following part, we calculated the R^2 of the train data and the validation data and show how they varies with the increase of the epochs. First of all, we combine the Softmax with MSE using Gradient descent method to get our Cost function. As we can see from the figure 15a,the R^2 scores of the train data and the validation data are close to 0.14 at the epochs 500.

Secondly, we combined the Softmax with MSE using stochastic Gradient descent method to get our Cost function.As we can

see from the figure 15b, the R^2 scores of the train data and the validation data are close to 0.16 and 0.14 respectively at the end.

And then, we combined the Softmax with crossEntropy using Gradient descent method to get our Cost function. In figure 16a, we calculate the R^2 of the train data and the validation data and show how they varies with the increasing epochs.As we can see, the R^2 scores of the train data and the validation data are close to 0.14 at the epochs 500.

Finally, we combined the Softmax with crossEntropy using stochastic Gradient descent method to get our Cost function. In figure 16b, the R^2 scores of the train data and the validation data are close to 0.18 at epochs 17.5.

Compare these four types of combinations, we could find that combination 1,2,3 got the high R^2 score but only the combination 2 can save the computation at the same time, which means that Stochastic Gradient Descent method could reduce the loss and save the computation at the same time.

Grid search for Test data and gradient Training data:

In Figure 17a and 17b, we draw the matrix, to see how we use Grid search to find the best learning rate and regularization factor. Finally, the best learning rate we got is 1.42 and the best regularization λ is 0.05.

8.2.4 Classification with PCA

We saw that PCA in most cases would lead to good fits and often faster convergence. The best learningrate would often change as well as the best value for the regularization parameter figures for grid searches for best learning rates, cumulative plots and cost history during training with PCA can be seen in figures (18) through (29)

9 Conclusions and Perspectives

The main content in this project is to use both classication and regression methods, including the regression algorithms studied in project 1, to solve the credit card problem.

In classification problem, we found that neural networks can get a higher Accuracy than logistic regression on the binary case of the credit card data to classify the binary case of default/no default.

When we use different kinds of solvers in our model, Newton-Raphson method has the best performance in accuracy while the gradient descent is the worst and the Stochastic Gradient descent is somewhere in between.
We used different combinations of cost functions and optimization algorithm to get the optimal settings for our dataset and we found that MSE generally suited better than cross-Entropy and stochastic gradient descent is better than normal Gradient descent. So the combination 2) achieved the highest Accuracy in all Cost functions.

Then, we used PCA to pick up the main characters of the dataset and do the same analysis. We found that PCA can basically remain the characters of the data and save the computation at the same time, which is good way in analyzing large number of data.

In Regression problem, we used neural networks to predict terrain data. Comparing this method to the linear regression methods that we used in project 1, we found that our Neural Network did not perform as well as ridge regression from project one. But SKlearns NN however performed better than ridge regression. That our Neural network performed so much worse than SKlearn indi-

	GD (PCA)		SGD (PCA)	
	sMax + MSE	sMax + xE	sMax + MSE	sMax + xE
Area ratio	0.1712	0.1571	0.2675	0.2918
Accuracy	0.7842	0.7838	0.7875	0.7864
best η	0.833	0.583	1.71	1.71
best λ	0.09	0.127	0.05	0.0667

Table 6: Table showing how well different settings did when using Principal Component Analysis on the credit card data. GD is Gradient Descent, SGD is Stochastic Gradient Descent, sMax is softMax, MSE is Mean Squared Error and xE is cross-Entropy.

cates that there is room for big improvements in our NN.

Further improvements that could be implemented or explored in this project

- exploring the effect of more layers/nodes in each layer
- exploring different activation functions
- implementing more advanced forms of gradient descent that could lead to more stable performance, like ADAM.
- implementing different types of Neural Networks

Appendix

9.1 List of codes on GitHub

`homeBrewLogReg.py`

This is used for Logisitic regression with gradient descent and contains the class that performs regression.

`homeBrewNN.py`

Contains the neural network class that performs regression and classification

`creditCardLogReg.py`

This is used for performing the training of the logisitic regression class in homeBrewLogReg.

`creditCardNN.py`

This is used for using classification based on creditcard data and performing learning on a neural network using the class from homeBrewNN.

`terrainDataNN.py`

This uses our own NN code from homeBrewNN to create a Nerual Network that predicts terrain data

`terrainSKlearn.py`

This uses sklearns MLP_regressor to train a neural network that predicts terrain data

`NN_test.py`

Used to test the network to ensure it behaves as expected.

References

[1] *CompPhysics/MachineLearning*. GitHub. URL: https://github.com/CompPhysics/MachineLearning (visited on 11/07/2019).

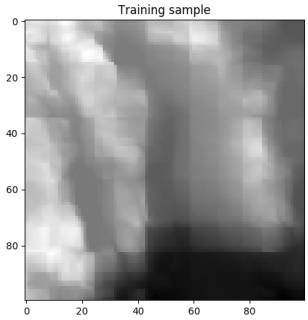
[2] James Dellinger. *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming*. Medium. Apr. 4, 2019. URL: https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79 (visited on 11/07/2019).

[3] Huang Haiguang. *fengdu78/Coursera-ML-AndrewNg-Notes*. original-date: 2017-11-04T10:04:08Z. Nov. 7, 2019. URL: https://github.com/fengdu78/Coursera-ML-AndrewNg-Notes (visited on 11/07/2019).

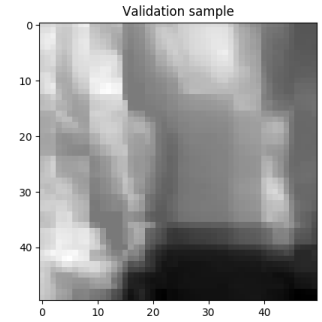
[4] Stan Lipovetsky. "Analytical closed-form solution for binary logit regression by categorical predictors". In: *Journal of Applied Statistics* 42.1 (Jan. 2, 2015), pp. 37–49. ISSN: 0266-4763. DOI: 10.1080/02664763.2014.932760. URL: https://doi.org/10.1080/02664763.2014.932760 (visited on 11/13/2019).

[5] *UCI Machine Learning Repository: default of credit card clients Data Set*. URL: https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients (visited on 11/12/2019).

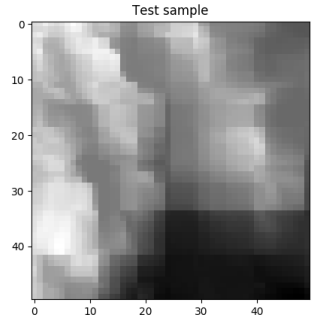
[6] I-Cheng Yeh and Che-hui Lien. "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients". In: *Expert Systems with Applications* 36.2 (Mar. 2009), pp. 2473–2480. ISSN: 09574174. DOI: 10.1016/j.eswa.2007.12.020. URL: https://linkinghub.elsevier.com/retrieve/pii/S0957417407006719 (visited on 11/10/2019).



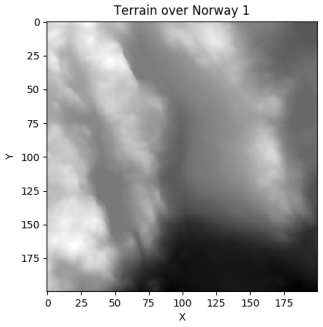
(a) The terrain training sample.



(b) The terrain validation sample.

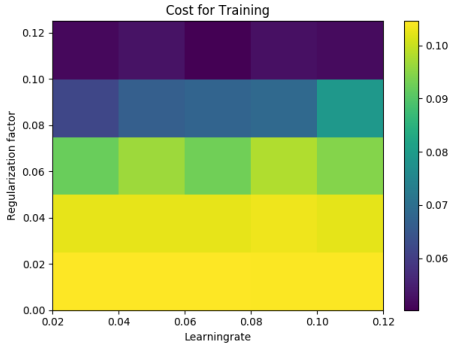


(c) Terrain test sample.

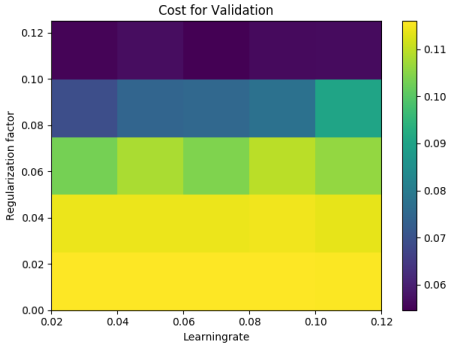


(d) Original terrain before split.

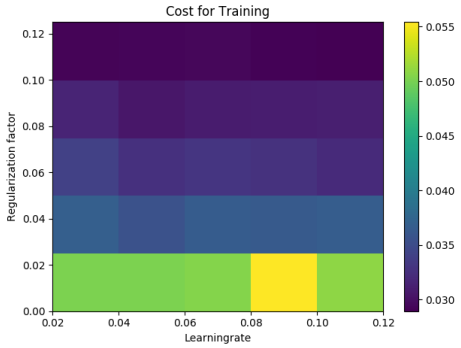
Figure 2: The different splits used to train the terrain regression.



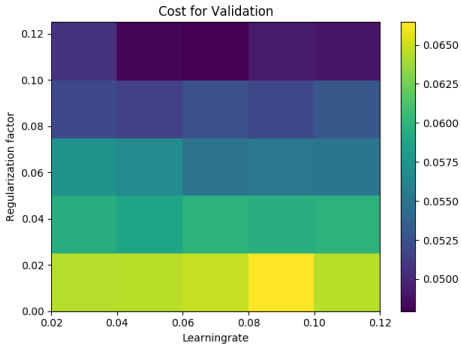
(a) The cost train matrix.



(b) The cost test matrix.

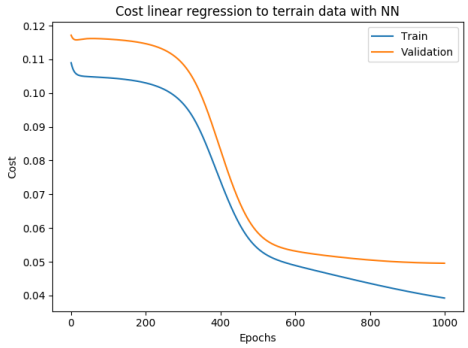


(c) The cost test matrix when using stochastic training.

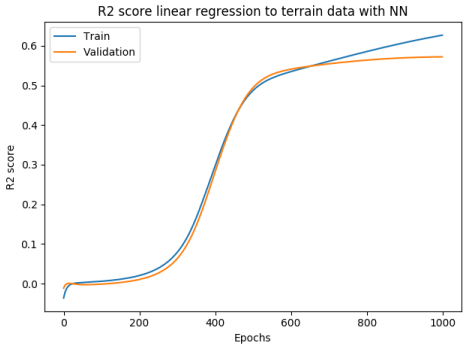


(d) The cost train matrix when using stochastic training.

Figure 3: The cost for different learning rates and regularization factors for plain gradient descent and stochastic gradient descent.

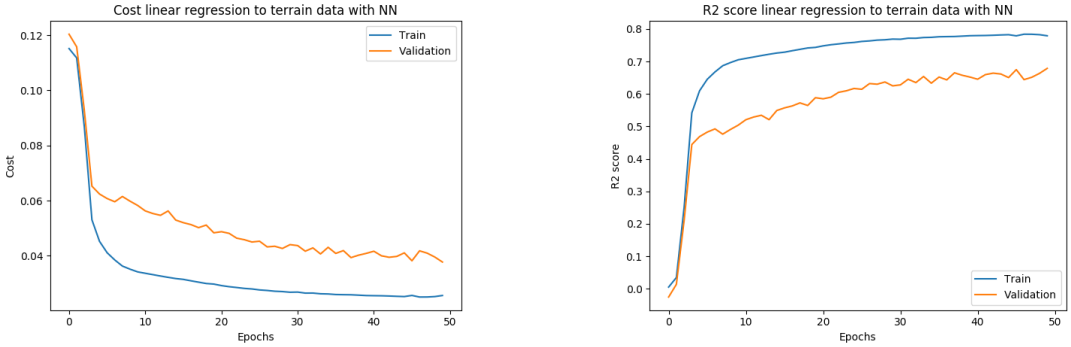


(a) The cost history.



(b) The R2 score history.

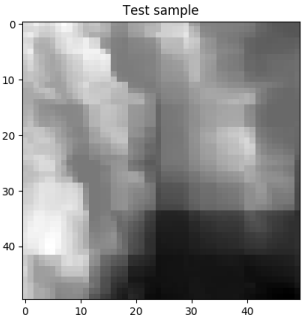
Figure 4: How stochastic gradient descent compares to gradient descent during linear regression.



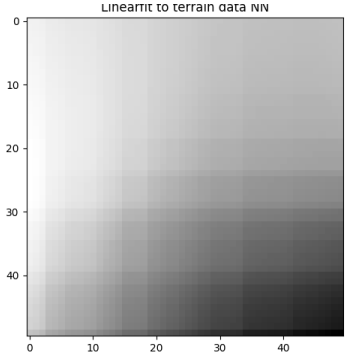
(a) The cost history using stochastic learning

(b) The R2 score while using stochastic training.

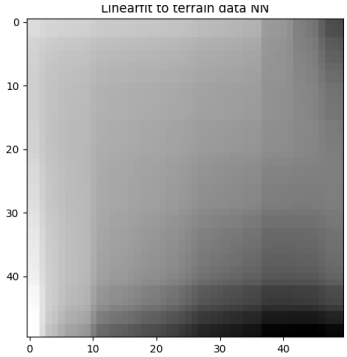
Figure 5: How stochastic gradient descent compares to gradient descent during linear regression.



(a) The test sample.

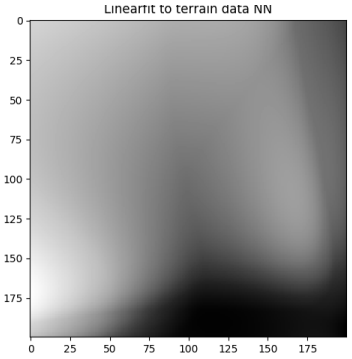


(b) Terrain fit for test sample using gradient descent.

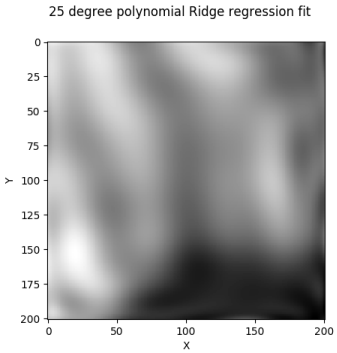


(c) Terrain fit for test sample using stochastic learning

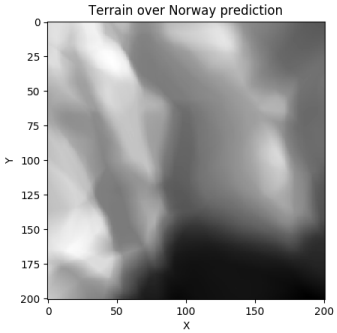
Figure 6: How stochastic gradient descent compares to gradient descent during linear regression.



(a) Our Neural network

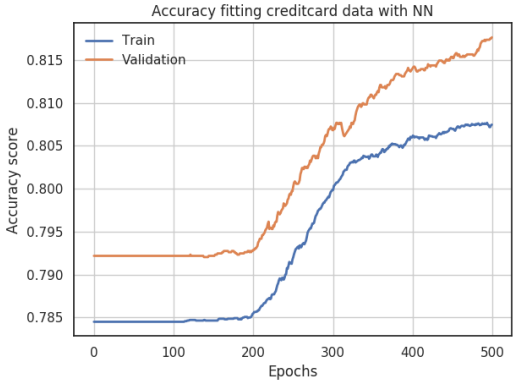


(b) Ridge regression

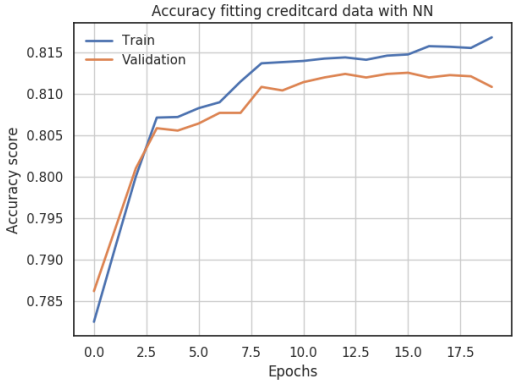


(c) Sklearn NN

Figure 7: Our NN, our ridge regression and SKlearns Neural Network.

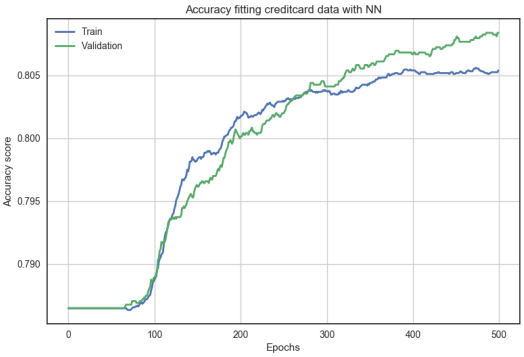


(a) MSE with gradient descent

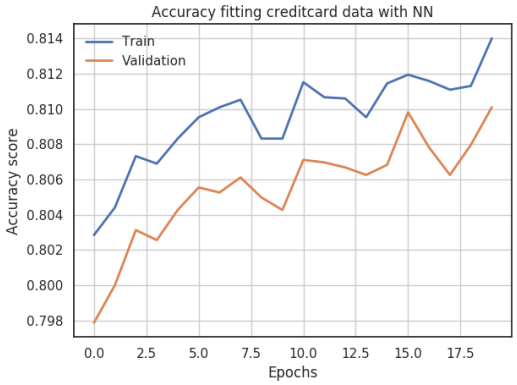


(b) MSE and stochastic gradient descent

Figure 9: Comparing gradient descent with stochastic gradient descent for MSE cost function 1

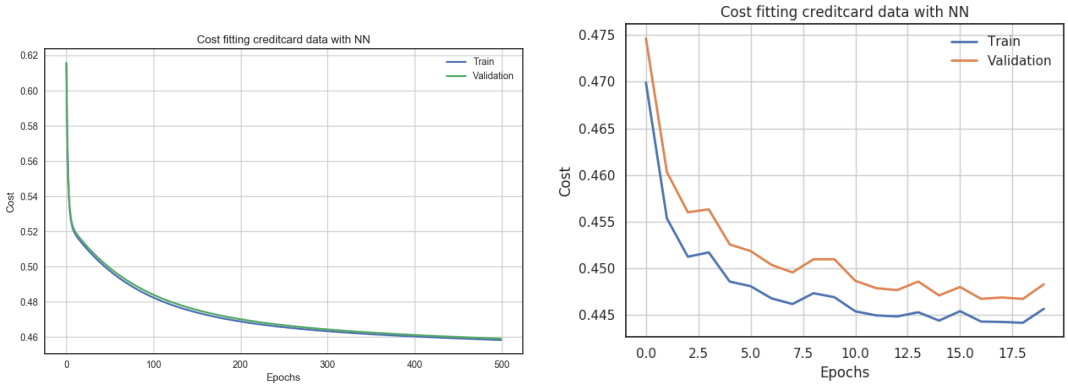


(a) Cross-Entropy with gradient descent



(b) Cross-entropy with stochastic gradient descent

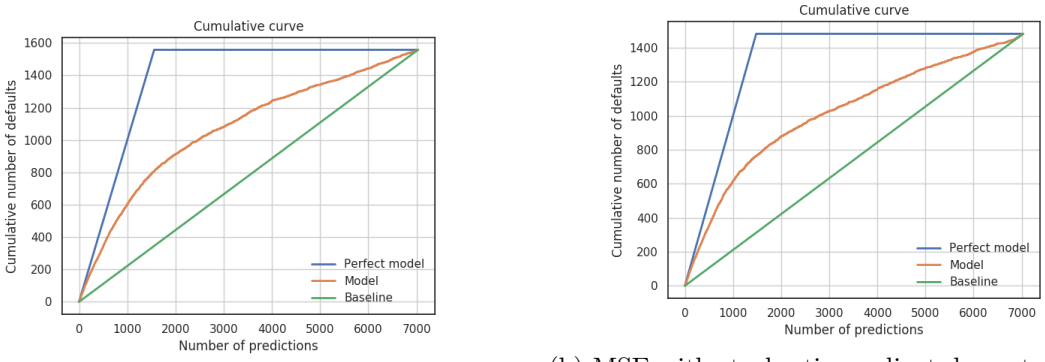
Figure 10: Comparing gradient descent with stochastic gradient descent for cross-Entropy cost function 2



(a) Cross-entropy with gradient descent

(b) Cross entropy with stochastic gradient descent

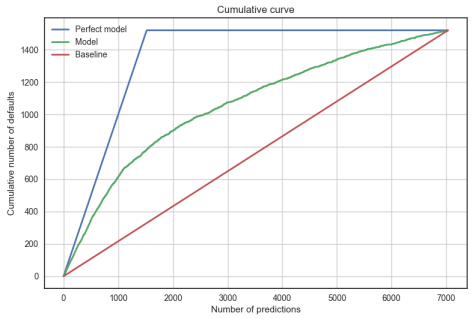
Figure 12: Comparing gradient descent with stochastic gradient descent for cross-Entropy cost function 2



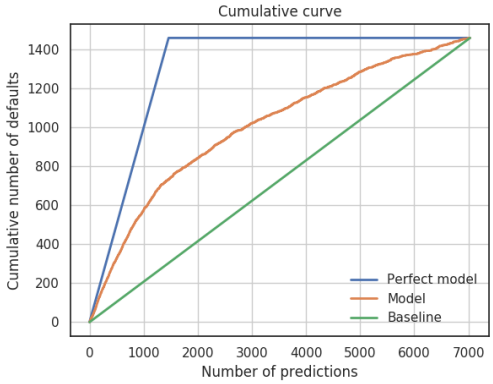
(a) MSE with gradient descent. AUC = 0.4805

(b) MSE with stochastic gradient descent. Area ratio = 0.47

Figure 13: Comparing gradient descent with stochastic gradient descent for MSE cost function 1

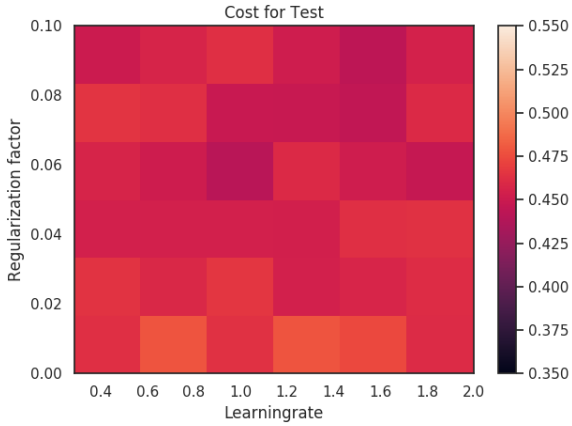


(a) Cross-entropy with gradient descent. Area ratio = 0.4394

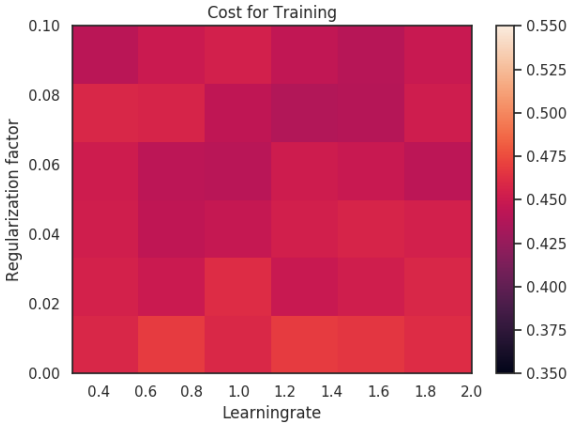


(b) Cross-entropy with stochastic gradient descent. Area ratio = 0.5059

Figure 14: Comparing gradient descent with stochastic gradient descent for cross-Entropy cost function 1



(a) Test data



(b) Training data

Figure 17: Grid search

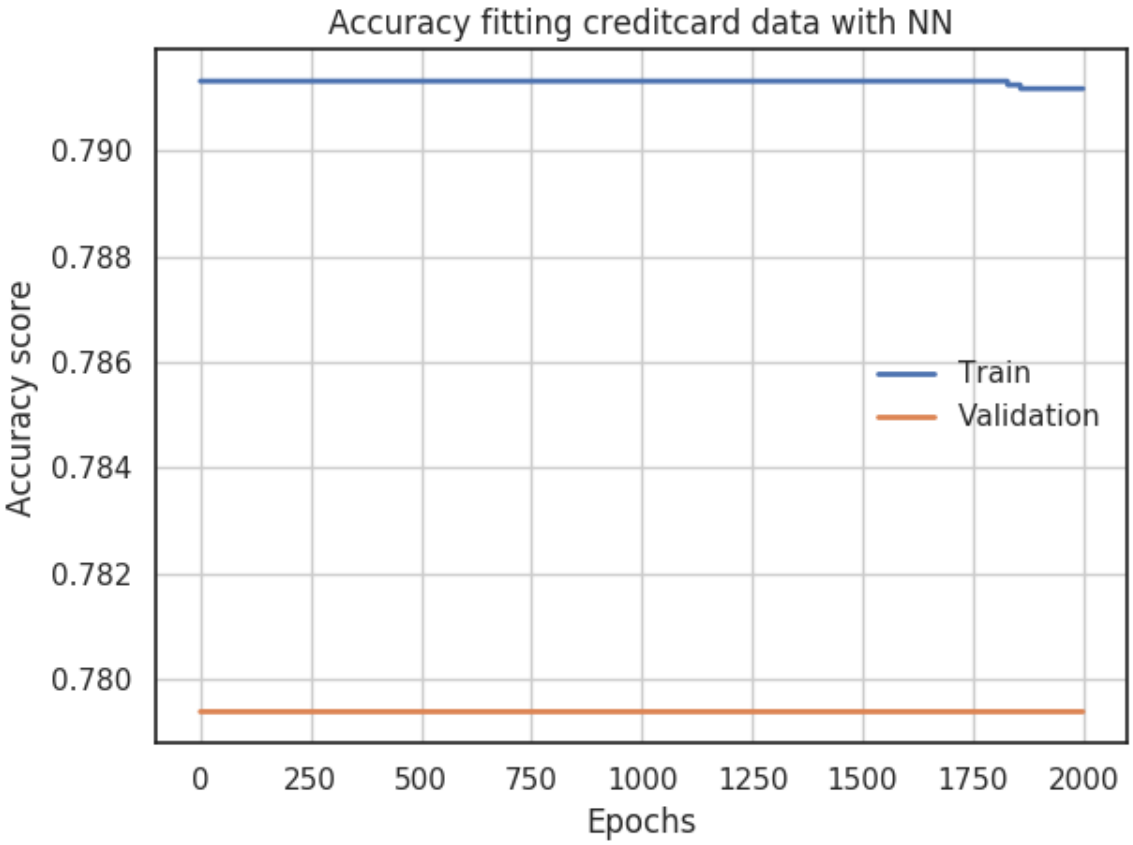


Figure 18: Progression of accuracy over epochs(PCA analysis)

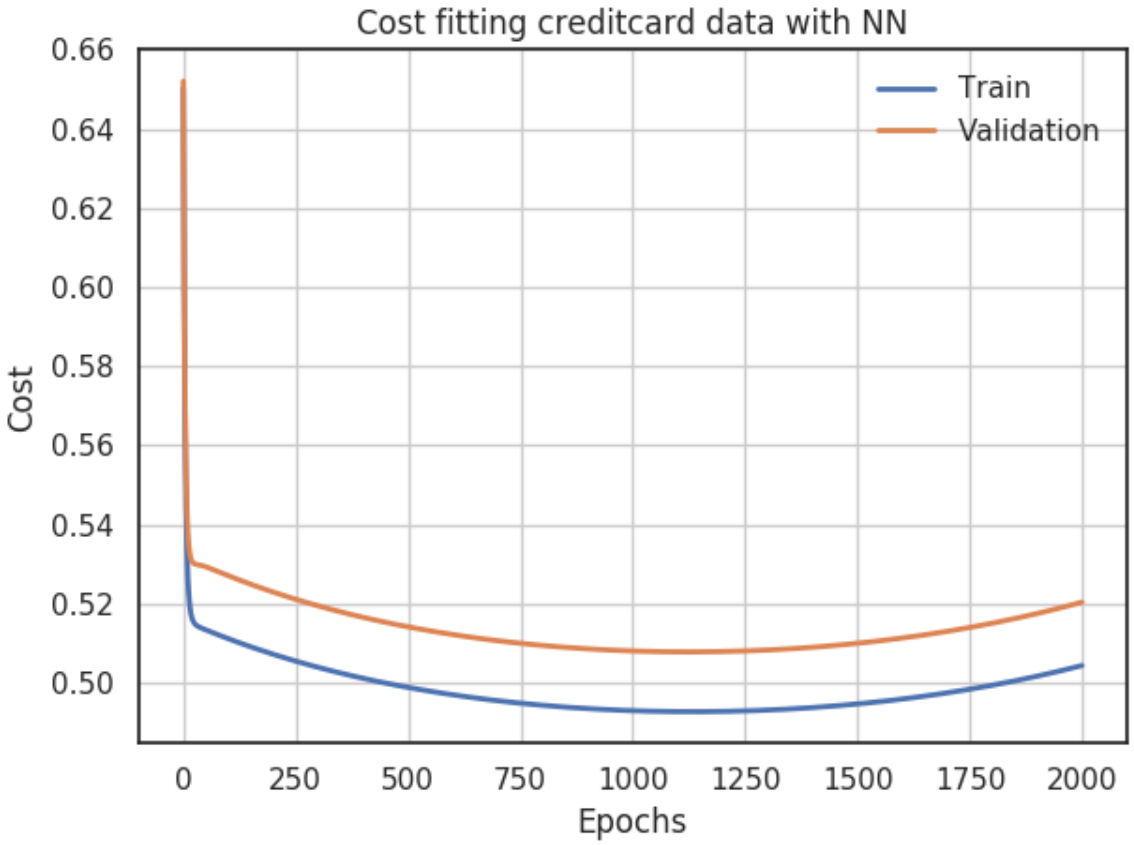


Figure 19: Progression of cost function in classification(PCA analysis)

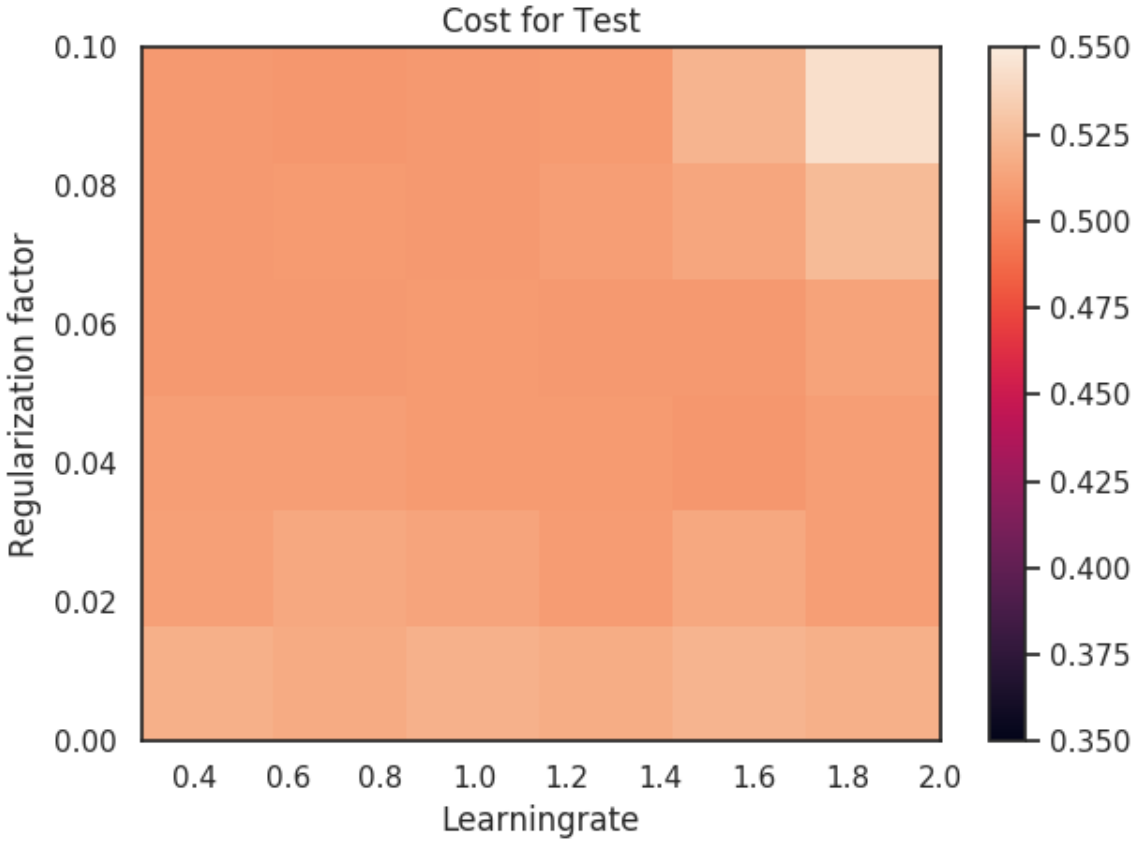


Figure 20: cost Test Matrix (PCA analysis)

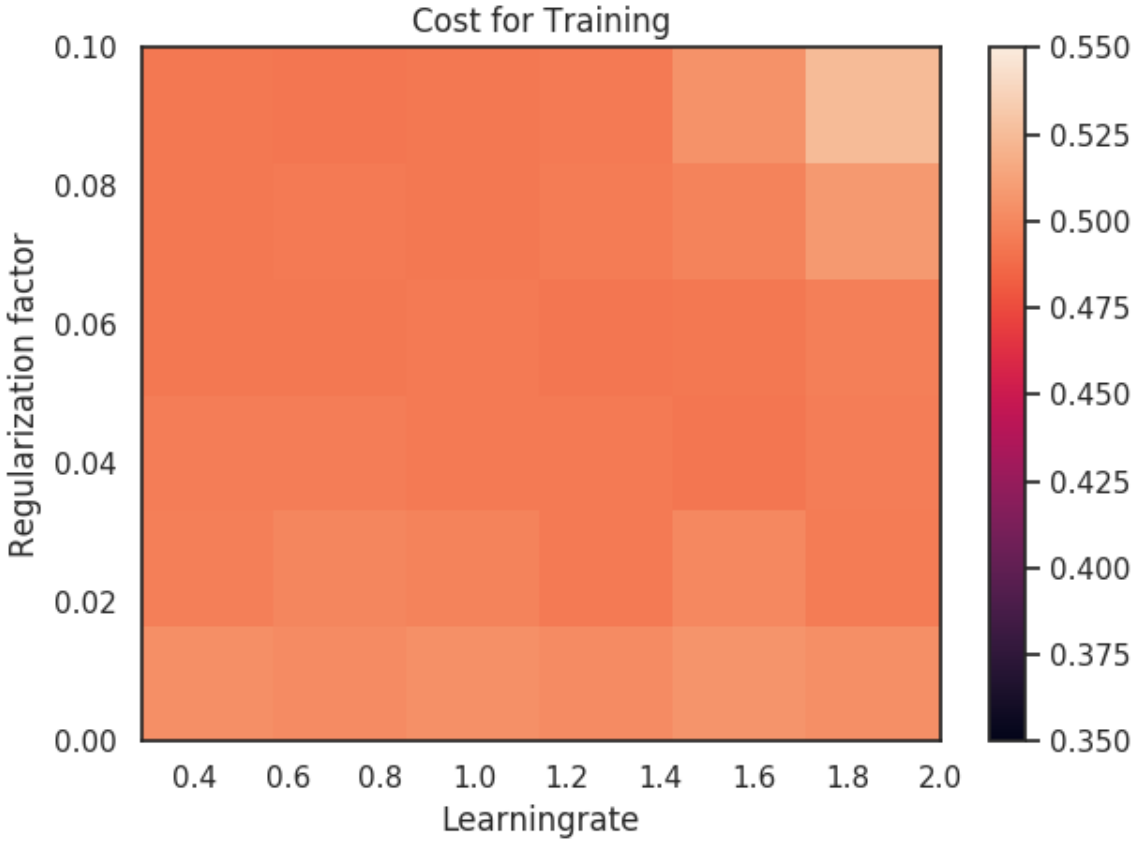


Figure 21: costTrainMatrix(PCA analysis)

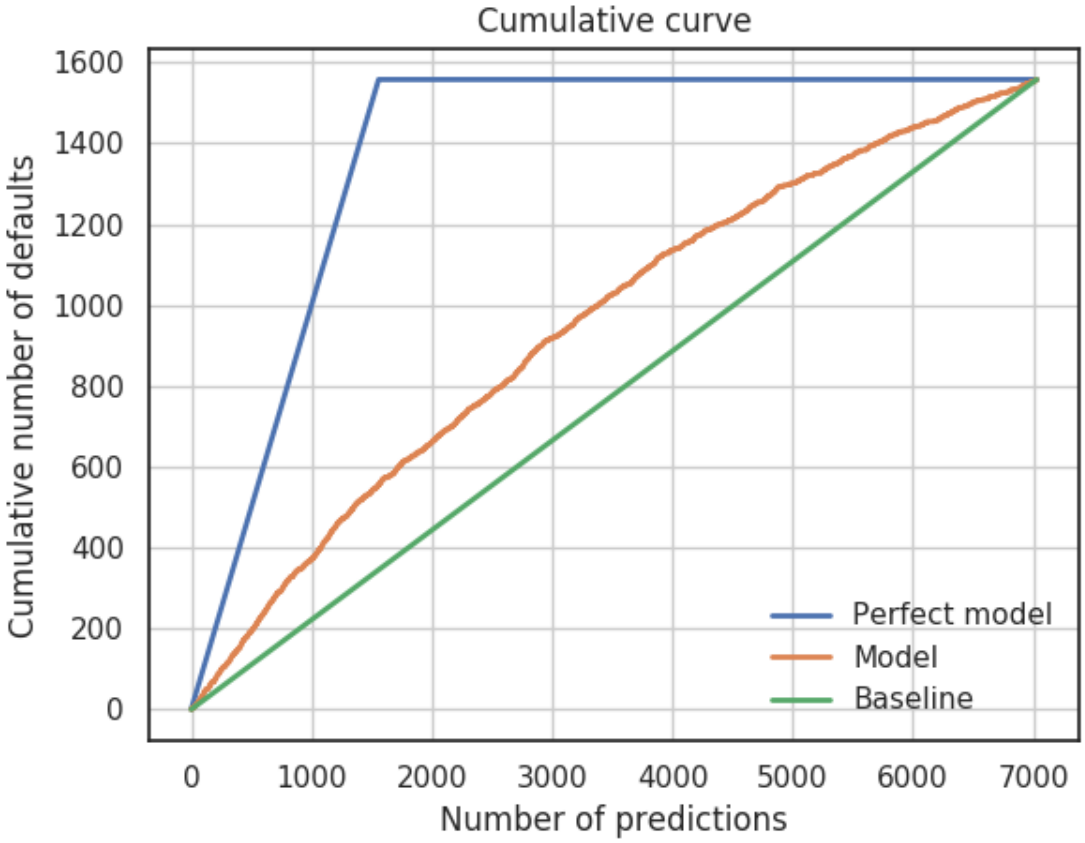


Figure 22: cumulativePlot (PCA analysis)

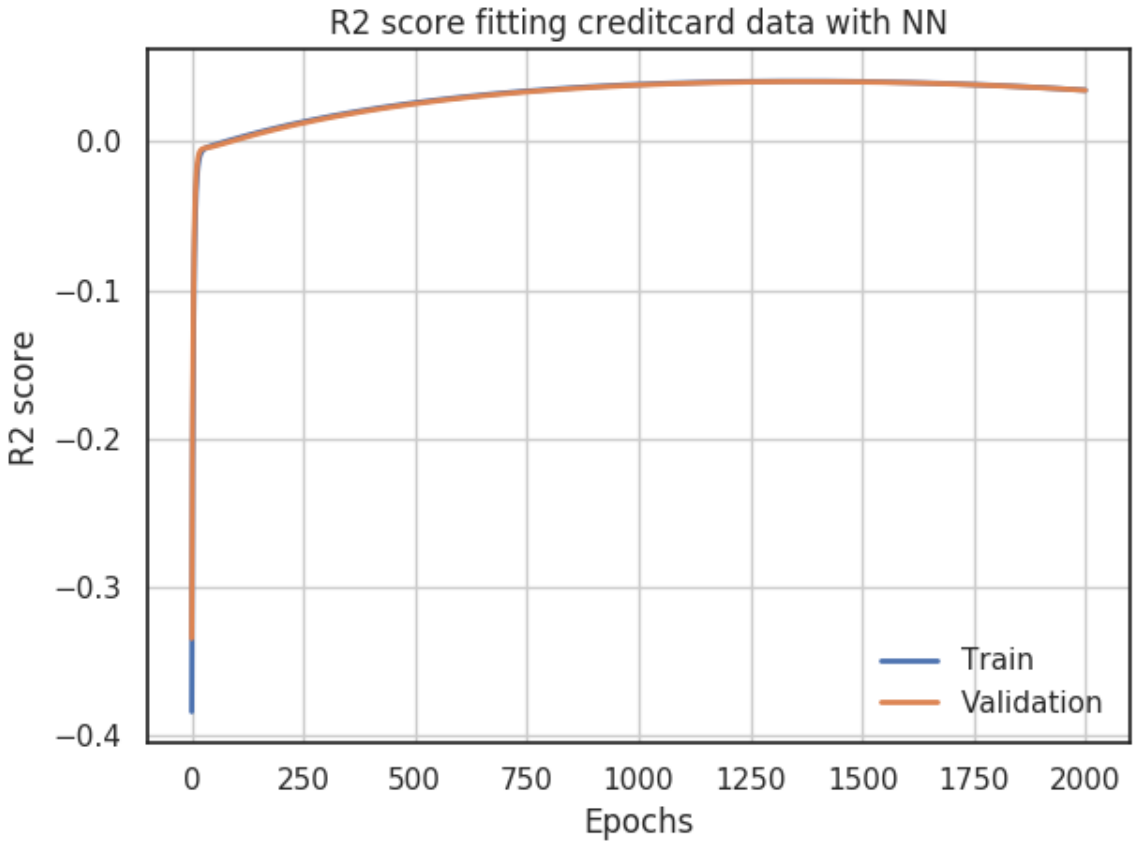


Figure 23: R2 History(PCA analysis)

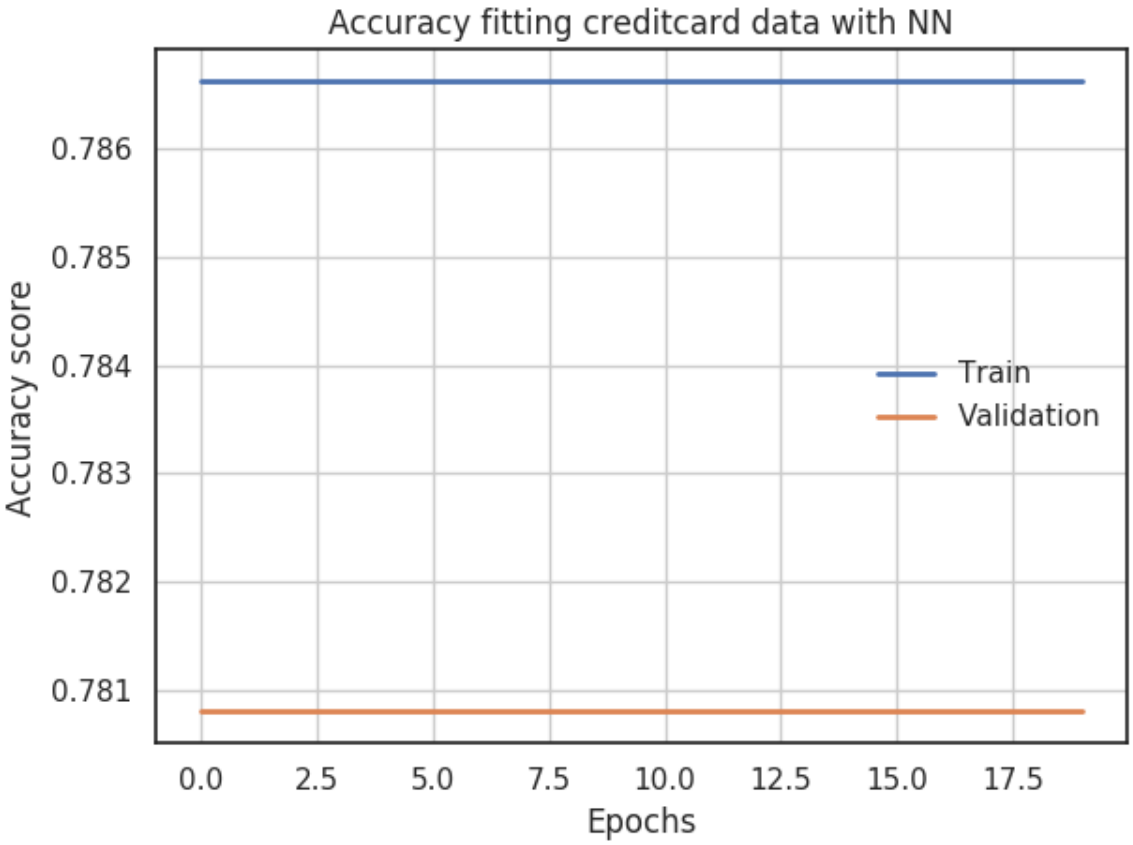


Figure 24: Progression of accuracy over epochs(PCA analysis with stochastic)

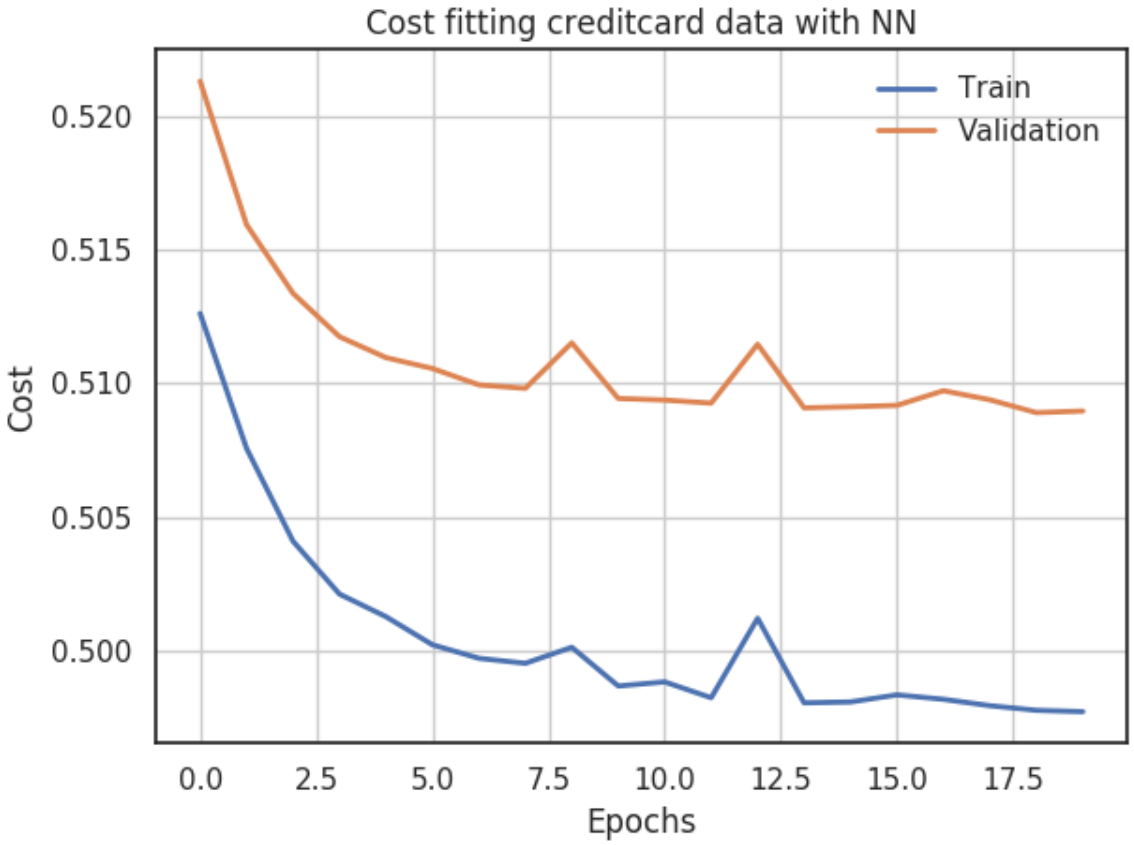


Figure 25: Progression of cost function in classification(PCA analysis with stochastic)

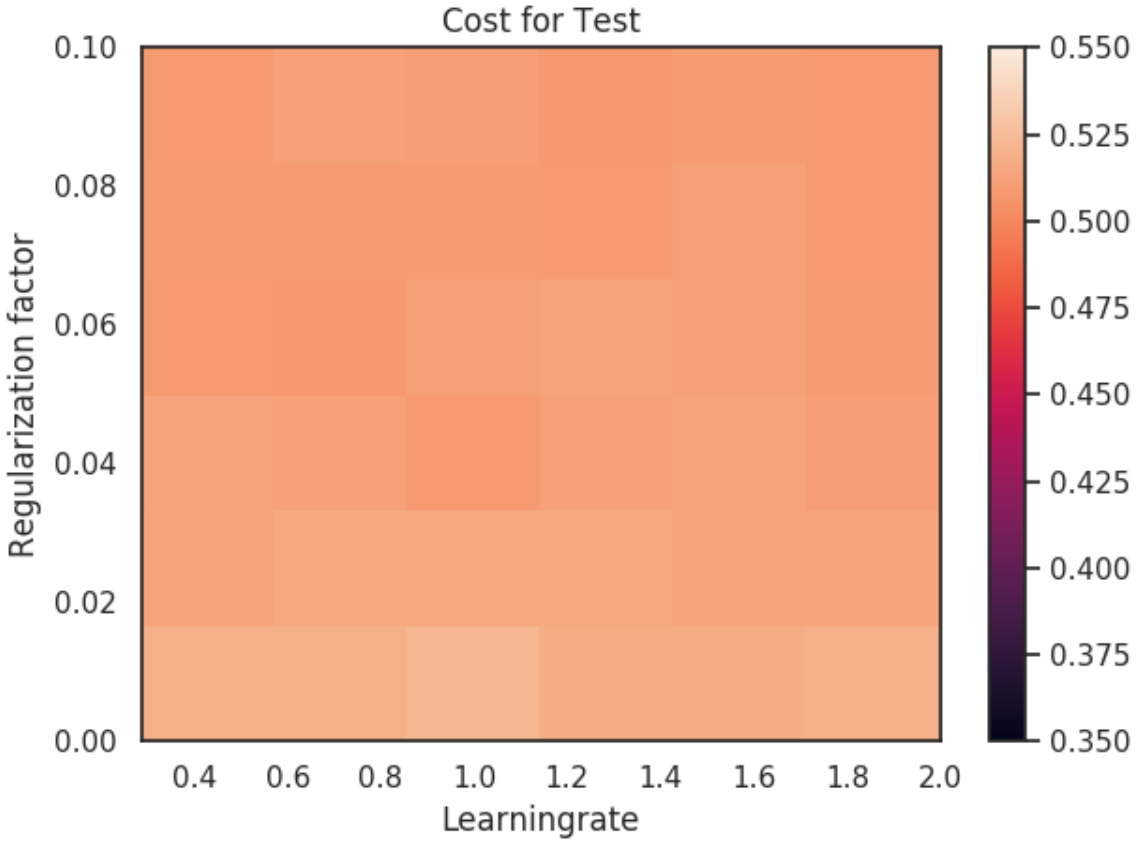


Figure 26: cost Test Matrix (PCA analysis with stochastic)

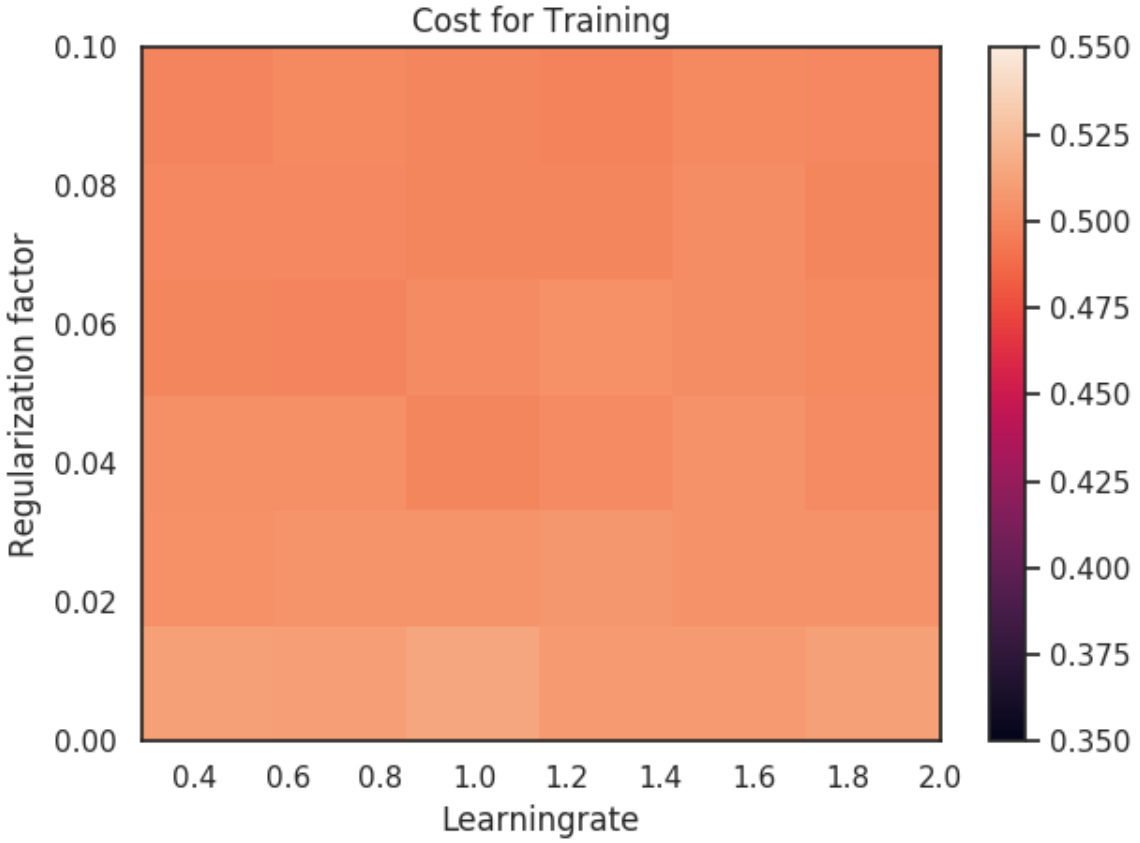


Figure 27: costTrainMatrix(PCA analysis with stochastic)

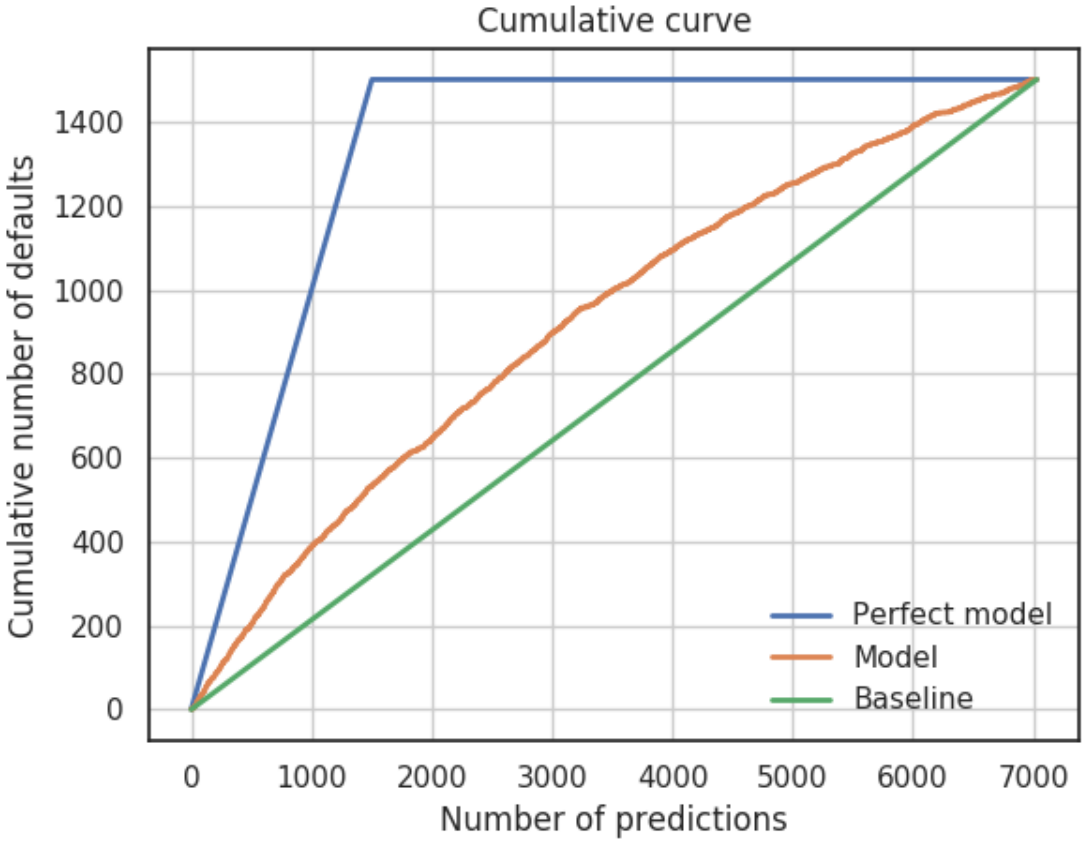


Figure 28: cumulative Plot (PCA analysis with stochastic)

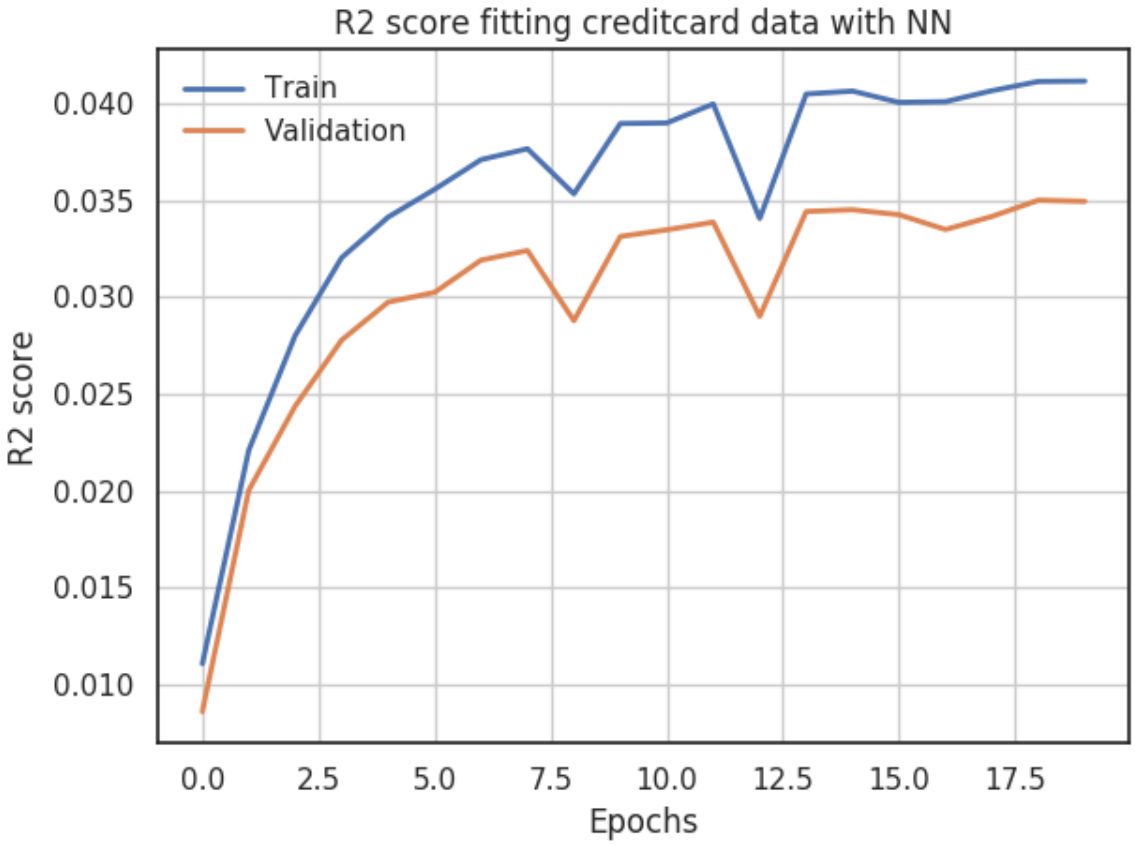
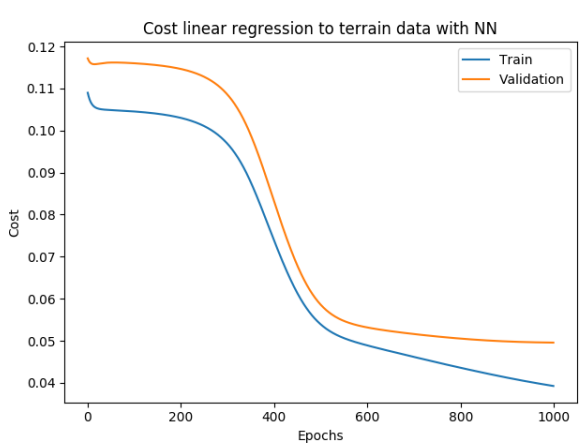
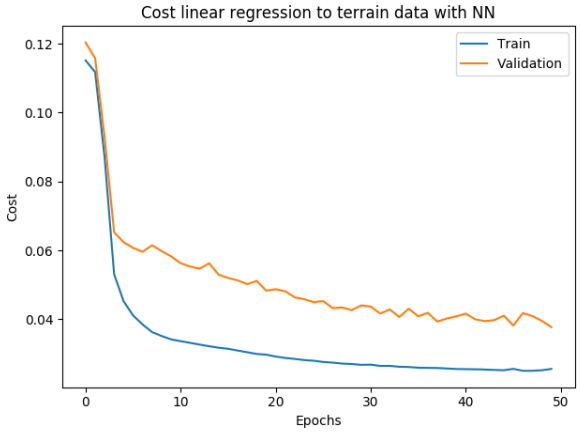


Figure 29: R2 History(PCA analysis with stochastic)

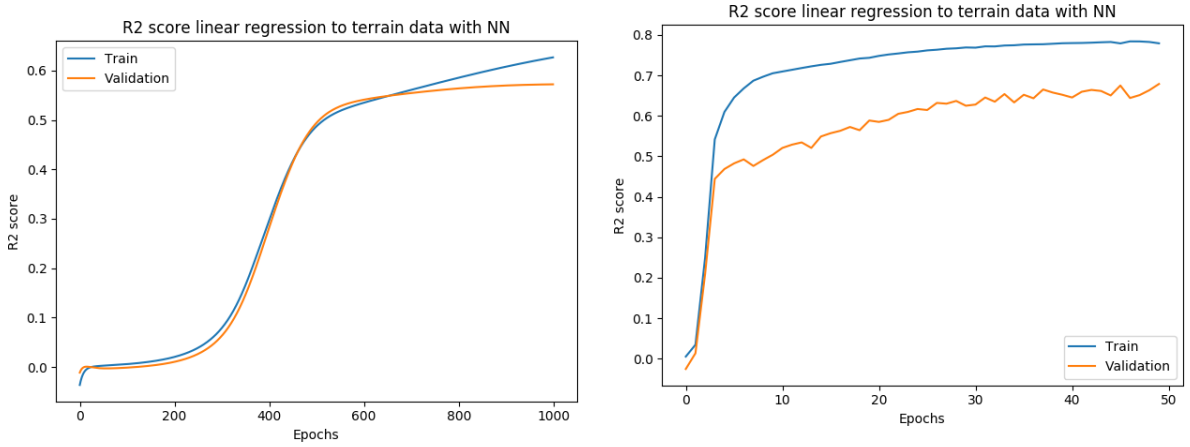


(a) Cost Terrain data with gradient descent



(b) Cost Terrain data with stochastic gradient descent

Figure 30: Comparing cost between Gradient Descent and Stochastic Gradient Descent for linear regression.



(a) R2 score Terrain data with gradient descent.

(b) R2 score Terrain data with stochastic gradient descent.

Figure 31: Comparing R2 scores between Gradient Descent and Stochastic Gradient Descent for linear regression.

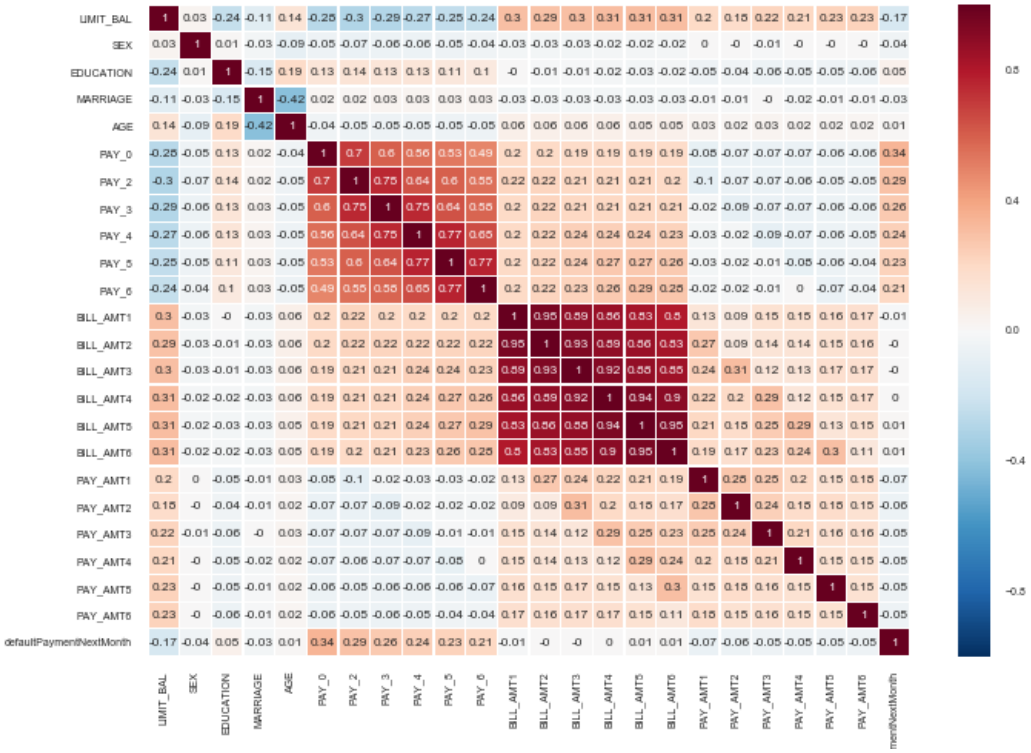


Figure 32: correlation matrix