Logistic Regression and Neural Networks

Eirik Nordgård
Geophysical Institute, University of Oslo

2019
November

Abstract

Logistic regression and neural network is used to predict the outcome variable of a credit data set. The outcome variable is paying or not paying debt, determined by features as gender, civil status, age and more. Retaining a so-called accuracy score gives a simple and intuitive measure of how well the respective models perform. My analysis of the credit card data suggest that logistic regression provided by scikit-learn is the model which best predicts whether the clients will default or not on their debt. This result is conflicting with results from other studies.

All material for this project may be found on `https://github.com/eirikngard/FYS-STK4155/tree/master/Project_2`

1 Introduction

In this project credit card data with 23 features is used to determine whether the clients will default or not default their debt. The data output thus is binary, 1 for default and 0 for not default, and prediction these includes solving a classification problem. In this project this is done using logistic regression and neural networks. Overall, the main goal is to assess which model best predicts the outcome based on the 23 features in the data, and how adjusting parameters of these models affect the result. The performance is measured by the accuracy score, which simply is a measure of how often the given model predict correct outcome. Accuracy score ranges from 0 to 1, 1 being a perfect classifier.

$$Accuracy = \frac{\sum_{i=1}^{n} I(y_i = y_i)}{n}$$

2 Theory

2.1 Logistic regression

Logistic regression is used to solve classification problems. These are problems concerning outcomes, y_i, in form of discrete variables. Commonly the classification problems in question has two possible outcomes, true or false. Using the credit card data as an example, the two outcomes would be if a client would pay his/hers debt or not. This type of outcome is often calleed binary outcome, and can easily be programmed as 1 or 0.

The dependent variables (outcomes) y_i are discrete, ranging from $k = 0, ..., K-1$ where K is classes. The main goal is to predict the output classes from a designmatrix $\hat{X} \in \mathbb{R}^{nXp}$ made of n samples carrying p features/predictors.

A descrete output can be obtained in several ways, but the simplest may be to have a sign

function, which maps the output of a linear regressor to values $0, 1$, $f(a) = sign(a) = 1$ if $a \geq 0$ and 0 if otherwise. Although this model is very simple, in many cases it might be convenient to know the probability of an output belonging to a given category rather than a single value. This is done using the *logistic function*, and is often called a "soft classifier". The classifier outputs the probability of x_i belonging to a category $y_i = \{0, 1\}$. In this case the classifier is given by a *Sigmoid* function, often reffered to as a likelihood function. The Sigmoid function $p(y)$ is given by[2]

$$p(y) = \frac{1}{1 + e^{-y}} = \frac{e^y}{1 + e^y} \tag{1}$$

and is visualized in Fig. 1.

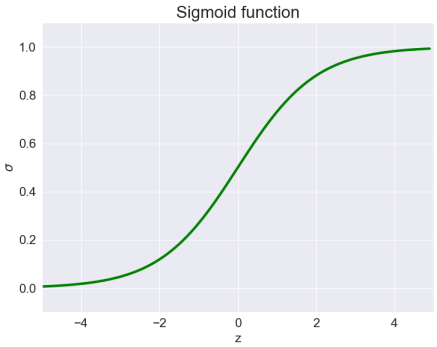

Figure 1: Plain Sigmoid as example of likelihood function

Like other likelihoods, eq. (1) here we have that the likelihood $p(y_i = 0) = 1 - p(y_i = 1)$, always resulting in a summed likelihood of 1. Assuming two classes y_i being either 0 or 1, we have

$$y_i = \beta_0 + \beta_1 x_i + ... + \beta_n x_i \tag{2}$$

Used in in eq.(1), this gives

$$p(y_i = 1 \mid x_i, \hat{\beta}) = \frac{e\beta_0 + \beta_1 x_i}{1 + e\beta_0 + \beta_1 x_i} \tag{3}$$

and

$$p(y_i = 0 \mid x_i, \hat{\beta}) = 1 - p(y_i = 1 \mid x_i \hat{\beta}) \tag{4}$$

Eq. (3) and eq. (4) can then be used to find the maximum likelihood of an event. The product of all individual probabilities of a specific outcome y_i is used to obtain a log-likelihood function, which in turn gives the cost function

$$\begin{aligned} C(\hat{\beta}) = \sum_{i=1}^{n} \big(& y_i \, log \, p(y_i = 1 | x_i, \hat{\beta}) \\ & + (1 - y_i) \, log \left[1 - p(y_i = 1 | x_i, \hat{\beta} \right] \big) \end{aligned} \tag{5}$$

Eq. (5) is called the *cross entropy*. Being convex, a local minimizer of this function will also be a global minimizer. Hence, minimizing eq. (5) with respect to each β gives

$$\frac{\partial \hat{\beta}}{\partial \beta_0} = -\sum_{i=1}^{n} \big(y_i - \frac{e^{\beta_0 + \beta_1 x_i}}{1 + e^{\beta_0 + \beta_1 x_i}} \big) \tag{6}$$

and

$$\frac{\partial \hat{\beta}}{\partial \beta_1} = -\sum_{i=1}^{n} \big(y_i x_i - x_i \frac{e^{\beta_0 + \beta_1 x_i}}{1 + e^{\beta_0 + \beta_1 x_i}} \big) \tag{7}$$

It is worth mentioning that far from all cost functions are convex. Many cost functions used in machine learning are in fact non-convex and of high dimensionality. Compressing eq. (6) and eq. (7) we get the following expressions for the first derivative. This is the gradient used in the gradient descent method explained later.

$$\frac{\partial C(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T (\hat{y} - \hat{p}) \tag{8}$$

Here $\hat{y}$ is a vector with n elements y_i, the $n \times p$ matrix $\hat{X}$ containing the x_i values and the vector $\hat{p}$ is the fitted probabilities $p(y_i \mid x_i, \hat{\beta})$. [2].

2.2 Gradient Descent Method

The idea of a gradient descent (GD) method is that a function $F(x)$, with $x = x_1, ..., x_n$, decreases fastest if one goes from x in the direction of the negative gradient $-\nabla F(x)$. Thus

$$x_{v+1} = x_v - \gamma_v \nabla F(x_v) \tag{9}$$

where γ_v is called learning rate or step length, in this case $\gamma_v > 0$. The gradient used in eq. (9) is obtained by eq. (8). For small enough γ_v we will then have $F(x_{v+1}) \geq F(x_v)$, meaning we are always moving towards smaller values and eventually a minimum of F [2].
Choosing appropriate learning rate is important. If chosen too small the method will converge very slowly, and chosen too large may result in unpredictable behaviour. The first step doing the GD is to make a guess x_0 for a minimum of F. Note that the minima is a global minima only if the function is a convex function [2].

One form of gradient descent is *stochastic gradient descent*. This algorithm is iterative, meaning the search process for the minima of the cost function occurs over multiple steps. These steps are done using so-called batches and epochs, and the intention is to improve the model parameters for each step. Batch is a hyperparameter that defines the number of data samples to work through before updating the internal model parameters. Epoch is a hyperparameter that defines the number of times the learning algorithm will work through the entire training data set. Iterating over the batches and epochs, you are very likely to find the minima of the cost function without updating the model parameters for each data sample.

2.3 Artificial Neural Networks

Artificial Neural Networks (ANN) is a computational system that by training on examples can learn to perform tasks. Increasing the number of examples the ANN can learn from will increase the accuracy of the tasks performed. It is called artificial neural network because it is supposed to mimic a biological system such as a brain, where neurons interact by sending signals in the form of mathematical functions. The ANN is constructed in layers containing an arbitrary number of neutrons, which in turn is connected by a weight variable. Similarly to an actual brain, the neurons in a ANN can only communicate with each other if the incoming signal exceeds a certain threshold value. If the signal is "strong" enough, an output is sent forward, but if this activation threshold is not reached the neuron remain inactive, providing zero output[2].

The neurons described above may be on different forms. One type of artificial neuron is called the *perceptron*, and it has the general form

$$y = \sigma(\sum_{i=1}^{n} w_i x_i + b_i) = \sigma(z) \tag{10}$$

This works by taking n binary inputs $x_1, x_2, ..., x_n$ and producing one single binary output. The importance of each binary input is represented by n weights, $w_1, w_2, ..., w_n$. The binary output is determined by if the weighted sum of inputs is less or greater than a threshold (bias) value. Thus,

$$output = 0 \; if \sum_i w_i x_i \; \leq \; threshold$$
$$output = 1 \; if \sum_i w_i x_i \; \geq \; threshold$$

This is the equivalent to whether the neuron is activated or not. Described in another way, the perceptron makes a decision (0 or 1) by weighing up arguments pro or con the decision. The network may also have several layers of neurons, each taking the

output of the "previous" layers as input. These layers are called *hidden layers*, and allows for very complicated decision making networks. It is apparent that logistic regression in a way is a single layered neural network.

If the perceptron where to make a wrong classification, a problem arises. The desired path of action would be to make small changes is the biases or weights of the network and check if it then made the correct classification. If not, you would repeat this process making small changes until the classification succeeds. However, making small changes in biases or weights is actually resulting in very large changes in the output. This problem is solved using another type of neuron, namely the *sigmoid*.

This neuron functions in the same way as the perceptron, but with different input and output. Now the inputs can be any value between 0 and 1. And instead of having output of 0 or 1, the sigmoid has output $\sigma(w * x + b)$, where b is the bias and σ is the sigmoid function defined by eq. (1). Hence the out of the sigmoid will be [3]

$$output = \frac{1}{1 + e^{-\sum w_i x_i - b}} \tag{11}$$

2.3.1 Backpropagation (ANN)

Backpropagation is an algorithm computing the gradient of the cost function. This quantity tells us what the partial derivatives with respect to both the bias and the weights are, and may be very useful if you want to know how fast the cost is changing when the bias or weights are changing. This actually allows us to get information on how the behaviour of the entire network changes when the bias or weights are changed. Like before, we have

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{12}$$

where a is the activation(referred to as output later), b is the bias, w is the weights, l is the layer number and σ is a function to every element in the vector $z^l = w^l a^{l-1} + b^l$. This way the activation in one layer is related to the activation in the previous layer. Thus we have $a^l = \sigma(z^l)$. z^l is called the weighted input to the neurons in layer l.

Then, for backpropagation to work, it is necessary to assume the the cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$ over cost functions C_x for individual training points, x. Also, the cost function must be able to be written as a function of the outputs from the neural network [3].

While computing the derivatives of the cost $\frac{\partial C}{\partial b^l}$ and $\frac{\partial C}{\partial w^l}$, an error will occur. This error is just referred to as *error*, and is written δ_j^l in the j-th neuron in the l-th layer. For later notation, the k-th neuron is belongs to the (l-1)-th layer. The quantity $\frac{\partial C}{\partial z_j^l}$ provides a good estimate of this error, and from here

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{13}$$

of neuron j in layer l. Backpropagation then relates δ to the derivatives of the cost. Rewriting eq. (13) gives

$$\delta_j^l = \frac{\partial C}{\partial a_j^l}\sigma(z_j^l) \tag{14}$$

which vectorized is

$$\delta^l = \nabla_a C \odot \sigma(z^l) \tag{15}$$

In eq. (15) $\nabla_a C$ expresses the rate of change of C with respect to the activation output. $\odot$ is the Hadamard product. Further rewriting gives

$$\delta^l = (a^l - y) \odot \sigma(z^l) \tag{16}$$

which in terms of the error in the next layer, δ^{l+1}, is

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma(z^l) \tag{17}$$

The name *backpropagation* arises from eq. (17). Here the error is moved backward through the activation function in layer l, giving the error in the weighted input to layer l.

Eq. (17) finally allows us to obtain equations for the partial derivatives with respect to the bias and the weights, respectively

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{18}$$

simplified to

$$\frac{\partial C}{\partial b} = \delta \tag{19}$$

and

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{20}$$

simplified to

$$\frac{\partial C}{\partial w} = a_{in} \delta_{out} \tag{21}$$

Eq.(20) has a very useful consequence. When the activation a is small, the derivative will also be small. This means that the gradient changes fairly little during a gradient descent, meaning the weight *learns slowly.*

The resulting backpropagation algorithm then looks something like this:

1. **Input** x: Find corresponding activation a^l for the input layer.
2. **Feedforward:** For $l = 2, 3, ..., L$ compute z^l and a^l (eq.(12)).
3. **Output error** δ^L**:** Compute vector $\delta^L = \nabla_a C \odot \sigma(z^L)$
4. **Backpropagate the error:** Compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma(z^l)$ for each l.
5. **Output:** Gradient of the cost function: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

[3]

2.4 Data description

The data set used in this analysis is credit card data from credit card holders in Taiwan. In total the dataset contains 23 variables, which is used to employ an output variable, *default payment*, with value *1 = default = not pay, 0 = not default = pay.* According to Yeh and Lien [4] the dataset contains the following parameters:

- 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.

3 Method

3.1 Data Preparation

Without processing, the dataset is 30000 × 23 data points. Within these values are several illegal ones, not mentioned in the data description. Thus, the following data are removed from the set:

$$EDUACTION = 0, 5, 6$$

$$MARRIGE = 0$$

$$PAY\_X = -2$$

$$BILL\ AMT\_X < 0$$

$$PA\ AMT\_X < 0$$

Additionally, according to Vladimir G. Drugov [5] 86, 5% of PAY_X has the illegal value 0. Removing these would mean loosing too much of the total data, so these values are not removed. We are then left with 22455 × 23 data points. A one-hot-encoder from scikit-learn is used to encode categorical integer values into one-hot numeric arrays. This feature give all the categorical columns value of 0 or 1. Our data matrix is now of dimensionality 22455 × 81. Finally, the remaining data is standardised using scikit-learns feature StandardScaler.
Containing a lot more not default (0) than default (1), the data is very skewed. This may result in models gaining better performance in predicting zeros than ones. To correct this skewed distribution the data is down-sampled so that the model can be trained on equally many zeros and ones. This leaves 12977 × 81 data points.

3.2 Execution of Python Code

Among the functions defined are *normal_gradient_descent* and *stochastic_gradient_descent*. Used in logistic regression, the (normal) gradient descent takes the arguments learning rate and number of iterations. Then it loops over all iterations, calculating the gradient in (9), which is used to obtain the betas. Using the functions *predict* and *accuracy* the accuracy score for logistic regression using a (normal) gradient descent is calculated. For comparison, the LogisitcRegression functionality from scikit-learn is used to make a reference - prediction. Also, my *predict* function is compared to scikit-learns functionality *accuracy_score*. Logistic regression is also done using the *stochastic_gradient_descent*-method with input arguments x, y, batch size, number of epochs and learning rate. A prediction is made looping over a range of learning rates and epochs. The accuracy scores are then visualized in a heatmap. Scikit-learns neural network is utilized. This implements a multi-layered perceptron algorithm that trains using backpropagation.

4 Results

Running my models typically gives results like this:

Accuracy Score		
Own model	Scikit model	Score
LogReg(GD)	-	81,708
-	LogReg	81,814
LogReg(SGD)	-	76,267
-	LogReg(SGD)	75,423
-	NeuNet	77,644

For some reason, the SGD and the NN scores is lower than the logistic regression. I find this very strange, as the paper from I. YeH and C.

Lien [4] find neural network to be 1 % more accurate than logistic regression. Also, the gap between the accuracy score of logistic regression and neural network is larger in my analysis than in Yeh and Liens. This result could be caused be the lack of fine-tuning. As of now, the neural network is *not* tuned to provide the best score possible. It is also worth mentioning that these scores are obtained *without* downsampling. With downsampling the scores are:

Accuracy Score		
Own model	Scikit model	Score
LogReg(GD)	-	72,826
-	LogReg	72,873
LogReg(SGD)	-	50,015
-	LogReg(SGD)	50,141
-	NeuNet	52,149

Now all of the accuracy scores are lowered by quite a bit. Also, the difference between the logistic regression and and the neural network is larger. Now having a score around 25 % lower than without downsampling, something is probably wrong with the neural network. Since the downsampling only leaves us with 12977 datapoints, it may actually be the case that the models don't have enough samples left to train on. The downsampled dataset may be too small to properly train the neural network. This is indicated through the fact that the accuracy score of the neural network is reduced when the training-share is reduced. Since the other score barely changes when reducing the training-share, it may indicate some kind error in the way the data is passed to net network. The other models are barley affected by the change of training-share.

5 Conclusion

My analysis of the credit card data suggest that logistic regression provided by scikit-learn is the model which best predicts the outcome variable. This is inconsistent with both Yeh and Lien [4] and Drugov [5]. Further investigating of the data and finishing of my the neural network would mots likely provide different results. It would allow me to compare the network by scikit-learn with another result, possibly determine why the accuracy score is dropping that much after performing a downsampling of the data.

6 Future Work

For future work I would finish my neural network code. This would allow me to tune all parameters to find the best accuracy, and then compare this to what was found in the paper of Yeh and Lien and Drugov [5]. It would also be interesting to investigate which features in the data are more important to whether a client will default or not on their debt. I would also like to spend more time on tuning the parameters of the neural network provided by scikit-learn. Investigate how the neural network performs with different activation functions, and with different activation threshold values. Additionally I would like to test my algorithm on other data sets, such as the Franke data stated in the project description.

7 Appendix

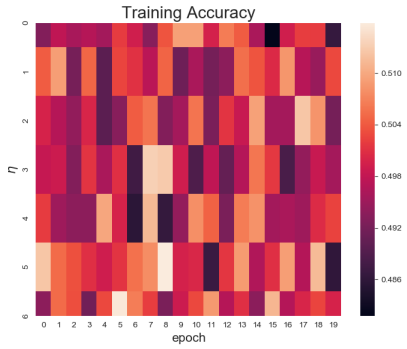

Figure 2: Heatmap of training accuracy with varying epochs and learning rate. The learningrate is spaced logarithmically.

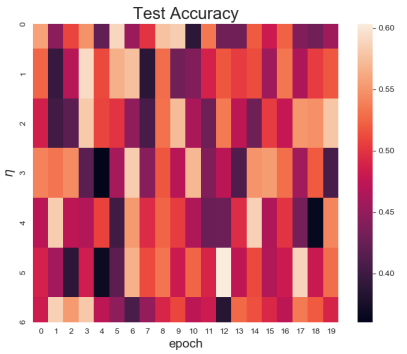

Figure 3: Heatmap of test accuracy with varying epochs and learning rate. The learningrate is spaced logarithmically.

References

[1] Hastie, Trevor. Tibshirani, Robert. Friedman, Jerome. *The Elements of Statistical Learning. Data Mining, Interference, and Prediction.*
Second Edition.
Springer, 2009.
Chapter 4, Chapter 11

[2] M. Hjorth-Jensen Lecture Notes in FYS-STK4155. *Data Analysis and Machine Learning: Logistic Regression and Neural Networks.*
URL: `https://github.com/CompPhysics/MachineLearning/tree/master/doc/pub/LogReg`
URL: `https://github.com/CompPhysics/MachineLearning/tree/master/doc/pub/NeuralNet`
Unpublished, 2019.

[3] Nielsen, Michael
URL: `http://neuralnetworksanddeeplearning.com`
Downloaded 13.10.19

[4] *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients.*
I. Yeh and C. Lien
URL: `https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf`

[5] *Default Payments of Credit Card Clients in Taiwan from 2005.*
Drugov, Vladimir G.
URL: `https://rstudio-pubs-static.s3.amazonaws.com/281390_8a4ea1f1d23043479814ec4a38dbbfd9.html`