Neural networks and logistic regression - an example

Adam Summers

08 June 2020

0.1 Simulating the data

Let's say we are trying to predict whether an applicant for a credit card will likely default or not at some point in the future and that we have three explanatory variables to predict this - a binary variable which tells us if the applicant has a mortgage or not (1 has a mortgage 0 doesn't). We also have another binary response variable which tells us the applicants gender (1 for male and 0 for female). Finally, we have another variable which tells us how many credit lines the applicant has (0, 1, 2 or 3 for simplicity). For the mortgage and gender variables we sample randomly, with replacement, from the vector (0,1) 5000 times. For the credit lines variable, we sample 5000 values from a uniform(0,3) distribution and round this value to the nearest integer. We then arbitrarily choose our beta values corresponding to the parameters in logistic regression: $\boldsymbol{\beta}$ = [-6,1.4,0.7,1.9] and we then use these values, and our simulated data, and use the equation (3.2) to give us the estimated probability, p, of default for each sample. Finally, we simulate 5000 values from a binomial distribution with one trial, with probability p for each sample, to give us our binary response variables (default or not).

Before we are ready to start building our models, we note that we need a way of actually testing their predictive performance so that we can compare results. We do this by splitting the simulated data into 2 sets - the training set and testing set. The training set is the data that we will use to build our model, and the testing set is the set that we will use to see how it performs in practice. In this case I have taken the first 4000

samples to go into the training set and the last 1000 samples to go into the training set, **but note** that in practice, with real data, the ordering has to be taken into account. For our example, a bank may have entered all of the defaults at the start of the dataset and so this means there would be no defaults in the testing set to see how good our models actually are at predicting defaults. Therefore, in practice it is necessary to shuffle your data randomly with the aim that whatever the proportion of 0's and 1's are for the response variable in the whole dataset, are the same in both the **training** and **testing** sets. This will help avoid bias both in training our model, and assessing its performance. However, since we have simulated data (pseudo) randomly, then the ordering should in a sense already be "shuffled".

0.2 The logistic regression model

In building the logistic regression model, I have used the GLM package in R using the training data. There is one important thing to note however, the logistic regression model maps any real number onto the interval (0,1), but we want to predict whether an applicant will default or not in the future (1 or 0). Hence, in order to use the logistic regression model to build a binary classifier, we need to set a threshold probability for which we will assign a 1 to an individual who's probability of default is greater than this threshold, and 0 otherwise. To do this we plot something called the receiver operating characteristic (ROC) curve. The ROC curve is created by plotting the true positive rate (sensitivity) against the false positive rate (1 - specificity) for different thresholds. We will use the pROC package in R to do this, and the plot is given by figure 1. Since the aim of this data is to predict whether an applicant may default or not in the future, the true positive rate is arguably the most important factor, since it reflects the correct rate of predicting people who will default. From a risk management perspective this is clearly important to a bank, since if they predict defaults poorly, then they will issue a credit card to many people who will end up defaulting, costing the bank significant amounts of money. Using the pROC package, we can see what threshold will give us a true positive rate of greater than 60%.

With a threshold of approximately 0.23 we have a true positive rate of approximately 83.33% and a false positive rate of approximately 20.82%. It is worth noting that this is simulated data, of only 3 explanatory variables. With real data and more variables at our disposal to predict defaults, we would hope that these results would be better. However, the purpose of this exercise is not to try and interpret if this is a good model for our simulated data, but to loosely compare the results of this model to that of a feed forward neural network, the results of which will be covered at the end of this section.

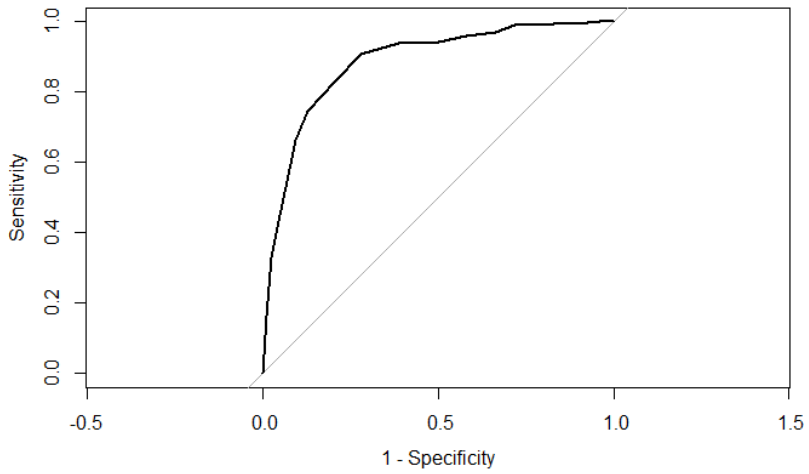

Figure 1: The ROC curve

0.3 Building the neural network

For our network, I will be using 1 hidden layer with 1 hidden neuron using the leaky ReLU activation function and an output layer with 1 neuron using the logistic activation function. When it comes to building more complicated networks for more difficult problems, I will be testing different amounts of hidden layers, hidden neurons and activation functions for the purpose of comparison. This example, however, is supposed to serve as an introduction into neural networks, and I will be building the code from scratch to implement this. I make no statement about how appropriate it may be to use one hidden layer and one hidden node with these activation functions for this example, since

it is purely for illustrating the main concepts of a neural network and how it may be implemented in practice. A graphical illustration of our neural network is given by figure 2.

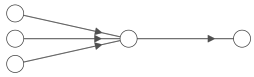

Figure 2: Our neural network

We will now go through the mathematics of the network. Since we have 3 explanatory variables, we have 3 weights connecting these to the hidden layer. We then have another weight connecting the output of the hidden layer, to the output node. Finally, we have 2 bias terms - one connected to the hidden neuron from the input layer, and another connected to the output neuron from the hidden layer. If we call our explanatory variables: x_1, x_2, x_3 (mortgage, gender and credit line variables respectively), the weights connecting these to the hidden neuron: w_1, w_2, w_3, the weight connecting the hidden neuron to the output neuron w_4, the bias terms b_1, b_2, the hidden neuron h, and the output neuron $\hat{y}$, then we have:

$$h_{in} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b_1$$

$$h_{out} = \begin{cases} h_{in} & \text{if } h_{in} > 0 \\ 0.01 h_{in} & \text{if } h_{in} \leq 0 \end{cases}$$

$$\hat{y}_{in} = w_4 h_{out} + b_2$$

$$\hat{y}_{out} = \frac{1}{1 + e^{-\hat{y}_{in}}}$$

So to build our neural network we set (arbitrarily) initial weights (and biases), and move forward in the network using these equations. However, unless we were incredibly lucky with our choice of weights (and biases), there will likely be a significant error associated with this output. Hence, we now move onto backpropagation to update these. We will be using batch gradient descent to do this (using all of the data at every iteration), but note that in practice, with very large datasets and more complicated networks, this is a rather inefficient approach. For our example though, it is sufficient.

We will be using the quadratic loss function and our aim will be to try and minimise this. Since we are using all of the data, we will be using the total cost:

$$C = \sum_{n=1}^{4000} (\hat{y}_{out_n} - y_n)^2$$

Where y_n is the actual observed value of the response variable in our dataset (default or not). The formulas for updating our weights and biases using gradient descent are:

$$w_{j_{new}} = w_{j_{old}} - \eta \frac{\partial C}{\partial w_{j_{old}}} \quad for \quad j = 1, 2, 3, 4 \tag{1}$$

$$b_{j_{new}} = b_{j_{old}} - \eta \frac{\partial C}{\partial b_{j_{old}}} \quad for \quad j = 1, 2 \tag{2}$$

Where we take the **average** of the weight/bias updates across the whole dataset and where η is the learning rate. In this problem I found a learning rate of 0.01 worked well. To work out these derivatives, we need to use the chain rule, first we define the derivative:

$$\frac{\partial C}{\partial \hat{y}_{out}} = 2 \sum_{n=1}^{4000} (\hat{y}_{out_n} - y_n) \tag{3}$$

Then for the w4 and b2 connecting the hidden layer to the output layer we have:

$$\frac{\partial C}{\partial w_{4_{old}}} = \frac{\partial C}{\partial \hat{y}_{out}} \frac{\partial \hat{y}_{out}}{\partial w_{4_{old}}} \tag{4}$$

$$\frac{\partial C}{\partial b_{2_{old}}} = \frac{\partial C}{\partial \hat{y}_{out}} \frac{\partial \hat{y}_{out}}{\partial b_{2_{old}}} \tag{5}$$

And since we are using the logistic activation function in our output, we have:

$$\frac{\partial \hat{y}_{out}}{\partial w_{4_{old}}} = \frac{h_{out} e^{\hat{y}_{in}}}{(1 + e^{\hat{y}_{in}})^2} \tag{6}$$

$$\frac{\partial \hat{y}_{out}}{\partial b_{2_{old}}} = \frac{e^{\hat{y}_{in}}}{(1 + e^{\hat{y}_{in}})^2} \tag{7}$$

And so we take equations (6), (3) and substitute them into equation (4) and finally we substitute this into equation (1) to get our formula for updating w_4. We follow a similar process to get our update for b_2. We now need to work out our updates for the weights/bias connecting the input layer to the hidden layer. Again using the chain rule:

$$\frac{\partial C}{\partial w_{j_{old}}} = \frac{\partial C}{\partial \hat{y}_{out}} \frac{\partial \hat{y}_{out}}{\partial h_{out}} \frac{\partial h_{out}}{\partial w_{j_{old}}} \quad for \quad j = 1, 2, 3 \tag{8}$$

$$\frac{\partial C}{\partial b_{2_{old}}} = \frac{\partial C}{\partial \hat{y}_{out}} \frac{\partial \hat{y}_{out}}{\partial h_{out}} \frac{\partial h_{out}}{\partial b_{2_{old}}} \tag{9}$$

The first derivative on the right hand side of (8) and (9) is given by (3). The second derivative is also common in both (8) and (9) and we define it here:

$$\frac{\partial \hat{y}_{out}}{\partial h_{out}} = \frac{w_{4_{old}} e^{\hat{y}_{out}}}{(1 + e^{\hat{y}_{in}})^2} \tag{10}$$

Finally, we are now ready to define our final derivatives:

$$\frac{\partial h_{out}}{\partial w_{j_{old}}} = \begin{cases} x_j & \text{if } h_{in} > 0 \\ 0.01 x_j & \text{if } h_{in} \leq 0 \end{cases} \quad for \quad j = 1, 2, 3 \tag{11}$$

$$\frac{\partial h_{out}}{\partial b_{1_{old}}} = \begin{cases} 1 & \text{if } h_{in} > 0 \\ 0.01 & \text{if } h_{in} \leq 0 \end{cases} \tag{12}$$

We then substitute (11), (10) and (3) into (8) to give us our required derivative that we can substitute into (1) to give us our equations to update the weights w_1, w_2, w_3. We follow a similar process to get our update for b_1.

We can now define intuitively how we build this neural network - we move forward in the network using our data and the first set of equations described in the start of this subsection and then get our output. We then try and minimise the cost of our network, and hence maximise its predictive performance by using (1) and (2) to give us an update of all of the weights and biases respectively. We then go through this process again (moving forward in the network with our updated weights, and then go back and update them again), until one of two stopping criteria is reached - either the difference in the cost function in consecutive iterations becomes less than a predefined tolerance, or, we can just run the algorithm for a fixed number of iterations. In my code, I ran the algorithm for 100 iterations, which appeared to work well and we will now discuss this more in detail in the following section.

0.4 Testing the neural network and comparison

Similarly to the logistic regression model, our neural network will output values on the interval (0,1) and hence we also need a too set a threshold here too. I tried different thresholds, and with a threshold of 0.1679 we achieved the same true positive and false positive rate given by the logistic regression model with a threshold of 0.23. In our neural network we only have one hidden neuron in the hidden layer with the leaky ReLU function, it is likely that in most if not all of the cases the input into this neuron will be the same as the output, since there is only one neuron to be "activated". We are then passing this output through the logistic function, and hence it makes intuitive sense that this neural network is making similar predictions to our logistic regression model. Clearly, a lot more work is required to build the neural network than the logistic regression model, and so we could conclude that in this case the logistic regression model would be the superior choice. However, it is worth noting, that in practice, we would build different networks with varying amounts of hidden layers and hidden neurons, and potentially different activation functions. Therefore, it may be possible to build a network which achieves greater predictive performance than the logistic regression model. This was not

the purpose of this exercise though, since I just wanted to introduce the concepts of a neural network and how a simple one may be built in practice and offer a loose comparison. When it comes to building more complicated networks on larger datasets, we will be able to make much more detailed comparisons.