Machine Learning Techniques for Classification

Daniel Smith

June 15, 2024

1 Introduction

In STATS216 in order to solve classification problems we learned logistic regression. This assigns a probability rather than a scalar value to our prediction. In this paper we will cover how Regression techniques can be used in Machine Learning to build Fully Connected Neural Networks.

2 Regression Theory Overview

In essence, regression involves a Loss Function that we are attempting to optimize. Take the function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{1}$$

where:

- $J(\theta)$ is the lost function in this instance we use Mean Squared Error.
- m is the number of training examples.
- $h_\theta(x^{(i)})$ is the predicted value for the i-th training example.
- $y^{(i)}$ is the actual value for the i-th training example.
- θ represents the parameters of the model.

2.1 predicted value

For the predicted value $h_\theta(x^{(i)})$, this would be what we classify as the output to a model as a review take the function

$$f(x_1, x_2, x_3) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \tag{2}$$

in this instance theta would be the the weights of the model and x would be the data points we have. In vector form this would be

$$f(\mathbf{x}) = \theta^T \mathbf{x} \tag{3}$$

where:

- $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$ is the vector of input features. The 1 allows us to append the constant term theta
- $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_0 \end{bmatrix}$ is the vector of parameters.
- θ^T is the transpose of the parameter vector, making it a row vector.

2.2 Linear Regression

The Mean Squared Error (MSE) Lost Function $J(\theta)$ for linear regression is defined as:

$$J(\theta) = \frac{1}{2m} (\mathbf{X}\theta - \mathbf{y})^T (\mathbf{X}\theta - \mathbf{y}) \tag{4}$$

Where:

- $J(\theta)$ is the Loss Function.
- m is the number of training data points.
- $\mathbf{X}$ is a matrix of input features from the data.
- θ is the vector of parameters.
- $\mathbf{y}$ is the vector of actual values we want from the data.

Equations 1 and 4 are the same, however equation 4 represents the equation in vector form. In the loss function known as Mean Squared Error, the predicted values are being compared against the true values, the closer those values are the less the model needs to shift its parameters, while values too distinct from each other will notify the model that there needs to be a bigger shift. This will be the output as a scalar value of the loss function. In order to compute this mathematically we use a method known as Gradient Descent.

The gradient of the Mean Squared Error (MSE) objective function $J(\theta)$ with respect to θ is:

$$\nabla_\theta J(\theta) = \frac{1}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y}) \tag{5}$$

Where:

- $\nabla_\theta J(\theta)$ is the gradient of the cost function.
- m is the number of training examples.
- $\mathbf{X}$ is the $m \times (n+1)$ matrix of input features.
- θ is the $(n+1) \times 1$ vector of parameters (including the bias term).
- $\mathbf{y}$ is the $m \times 1$ vector of actual values.
- $\mathbf{X}^T$ denotes the transpose of $\mathbf{X}$.

The function below is the key to using Gradient Descent as it's name implies, it updates the Weights based on a Gradient in order to descend the Loss Function and find the optimal point. This formula is called the delta rule:

$$\theta := \theta - \alpha \nabla_\theta J(\theta) \tag{6}$$

$$\theta := \theta - \alpha \frac{1}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y}) \tag{7}$$

where

- alpha is a hyper parameter selected by the designer

Now that the model has had its weights, theta, updated. The model should ideally perform better as more and more training data is used. There will be times that the same data is used multiple times to train the model, each time the model has had the chance to see every data point in the training set as well as update all it's weights accordingly. We call this an epoch. This process ultimately repeats over and over again until the function is able to approximate the data well enough. Consider reading PAC theory if you are interested in the proof. Figure 1 shows a graphical diagram of our model.

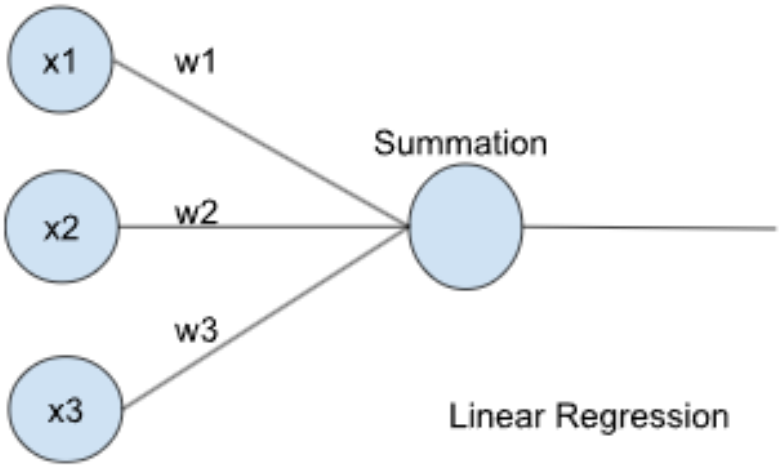

Figure 1: Linear regression model

2.3 Logistic Regression

Using the model we have above we can perform linear regression which would give us a model to predict any numerical data. We can use this same model to also predict classification tasks. In order to do this we have to use the sigmoid function.

$$\sigma(f(x)) = \frac{1}{1 + e^{-f(x)}} = J(\theta) \tag{8}$$

where

- $\mathbf{f}(\mathbf{x})$ is the output of our linear regression model

The reason this specific equation is chosen is

- The output is bounded between 0 and 1. This allows us to use it to map a function to its output and have that output be a probability
- The derivative shown in figure 9 is recursive meaning its a function multiplied by itself and thus is extremely easy and efficient to compute during the backpropogation step

$$\nabla_\theta J(\theta) = \sigma(f(x)) \cdot (1 - \sigma(f(x))) \tag{9}$$

- where z is the input
- i through K are the one hot encoding of possible outputs
- z is the output of the linear model

Figure 2 is a graphical representation of our logistic regression model. This model is the same as the linear regression model in Figure 1, however there is no a final sigmoid layer before the activation. In many instances the sigmoid layer will be combined with the activation layer as well as many other types of functions.

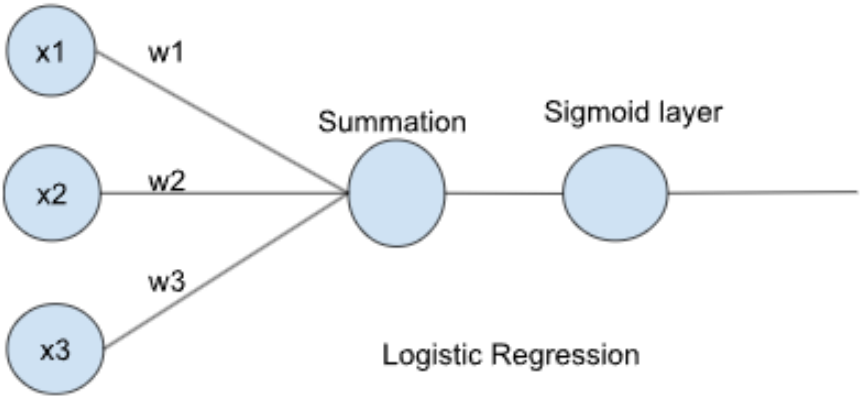

Figure 2: Logistic Regression Model

2.4 Multinomial Logistic Regression

Often times the task we are trying to complete involves creating a model that can classify more than one output. In this instance we can alter our model to have more than one output. To do this we use the softmax function $\sigma(\mathbf{z})$ is defined as:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K \tag{10}$$

This allows us to one hot encode a value to all possible outputs and assign a probability distribution. Figure 3 shows the graphical representation of our model. you can think of the softmax as a function that outputs a probability that has K distinct rows. These rows will be the one hot probability distribution where each row corresponds to the predicted probability of that class.

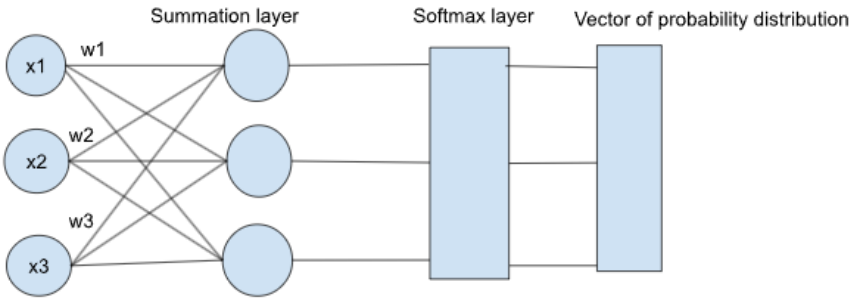

Figure 3: Multinomial regression diagram

2.5 Activation Layer

As you can see these layers can be chained together as long as we can calculate the gradient we need to use in the Loss Function. However these functions before they are passed into the softmax or sigmoid layer are all linear functions. Figure 4 an example of a classification problem that no linear model will be unable to classify

Figure 4: Non linearly separable problem

As seen in Figure 4 if the red dots represented one class and the green dots represent the other then no linear boundary will be able to fully separate the data into 2 classes. In order to fix this we have to use something known as an Activation function. Every Activation function is non linear, which means that the outputs of the function will also be non linear. The common used Activation function, Linear Rectified Unit, ReLU for short, is a simple and efficient function described as:
or simply

$$ReLU(z) = \max(0, z) \tag{11}$$

This equation essentially only allows positive values to propagate through thus making the output matrix non linear. Similarly as with previous layers we defined above, we need to calculate the derivative of this function. This function is not derivable at the point 0, however we can treat it as such by splitting the function into two parts shown in equation 12.

$$ReLU(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases} \tag{12}$$

The derivative is shown in equation 13.

$$\nabla_\theta \text{ReLU}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{13}$$

2.6 Deep Learning Networks

Connecting it all together we can combine the outputs of one layer onto another until we form what is known as a Fully Connected Network. It is fully connected since each of the outputs from a previous layer all get sent to each node in the next layer. In figure 5 we can see that the layers we have can normally be collapsed as such. In order to now compute the gradients of this multi layered network we need to define the chain rule below as follows in order to get the gradient of the Loss Function with respect to the weights as seen in equation 14

$$\nabla_\theta J(\theta) = \frac{\partial L}{\partial \mathbf{a}^{(L)}} \cdot \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} \cdot \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{a}^{(L-1)}} \cdots \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{z}^{(l+1)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} \tag{14}$$

where:

- $\mathbf{z}^{(l)}$ is the pre-activation at layer l.
- $\mathbf{a}^{(l)}$ is the activation at layer l.

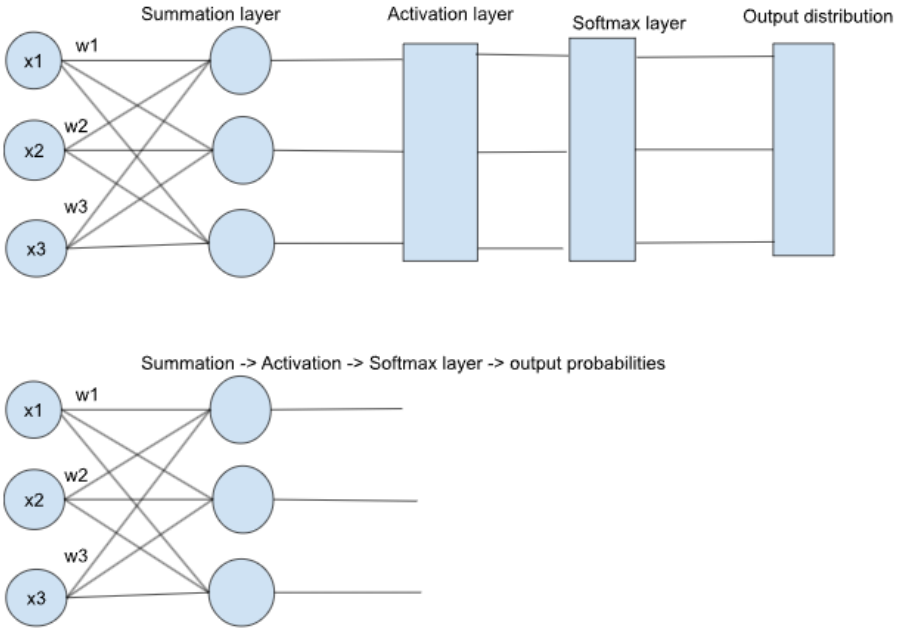

Figure 5: Collapsed Network

With our collapsed network we are able to now chain many more layers of differing sizes as well such as shown in figure 6. The only difference is that now the softmax or activation layer is only at the final layer instead of the inputs and hidden layers. In other words we only need to use it once at the very end of the model.

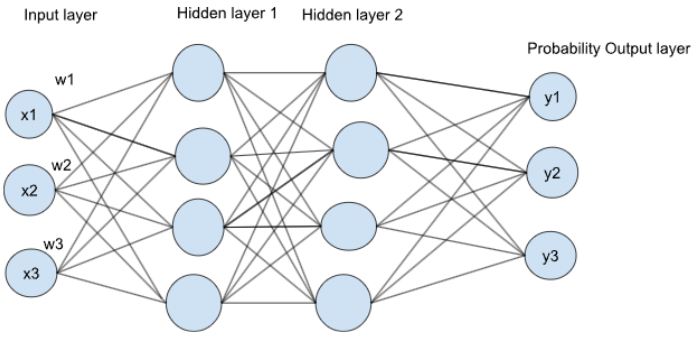

Figure 6: Fully Connected Network

Equation 14 lets us follow the delta rule described in equation 4 to update our weights as this chain rule produces the gradient of the loss function in respect to its weights/parameters. The important part is the gradient needs to be calculated for each layer and eventually passed through, this step is known as back propagation. The weights are then subsequently updated.

2.7 Regularizer

If we take a look at our our loss function described in equation 5 we can see that the only metric we are concerned with is the difference between the predicted value and the actual value. While this is certainly helpful, ultimately this causes an issue of over fitting, meaning the model will perform well on data it has seen during training, however new data will cause the model to predict incorrectly. A remedy that is often used are L1 and L2 penalties. The intuitive reason as to why we would want these penalties is due to the effect it has on the Loss function, in equation 1 we can append the L1 or L2 penalties to get a Loss function that incorporates the states of the weights. In general Regularizers work but removing the impact that any individual weight can have on the model. In essence they try to ensure that not one set of weights dominates the behavior of the model. In addition they use a hyper parameter lambda which is used as a constant scaling factor to the penalty.

L1 penalty:

$$\text{L1} = \lambda \sum_i |w_i| \tag{15}$$

L1 penalty attached to Loss Function:

$$J(\theta) = \frac{1}{2m} \left(\mathbf{X}\theta - \mathbf{y}\right)^T \left(\mathbf{X}\theta - \mathbf{y}\right) + \lambda \sum_i |w_i| \tag{16}$$

For an L1 penalty we use the absolute value because we only care about the magnitude that weight has and as a result if the weights become too large the penalty increases proportionally which increases the value of the Loss function. This results in a model with some weights that are 0 or very close to 0, ie they are not used or have little to no impact on the model's behavior

L2 penalty:

$$\text{L2} = \lambda \sum_i |w_i^2| \tag{17}$$

L2 penalty attached to Loss Function:

$$J(\theta) = \frac{1}{2m} \left(\mathbf{X}\theta - \mathbf{y}\right)^T \left(\mathbf{X}\theta - \mathbf{y}\right) + \lambda \sum_i |w_i^2| \tag{18}$$

For an L2 Penalty we use the variance between the weights of the model as a penalty indicator. If the model's weights have a high variance between them that indicates that a few weights are dominating the behavior of the model. This increases the L2 penalty which results in a higher value for the Loss Function. The effect this penalty has on the model is it reduces the variance between the weights to ensure that no set of weights dominates the behavior. This differs from the L1 penalty as the weights will tend not to approach 0.

3 Regression using Python

Typically we are not expected to calculate or even derive any of the gradients or formulas in order to effectively use a package such as PyTorch. PyTorch is one of two popular Deep Learning Frameworks, the other being TensorFlow/Keras. The reason for its popularity is ease of use and similarity to other packages such as Numpy. All the code needed to recreate Model will be available at

`https://github.com/jakq277/Stats216_analysis`

3.1 Data

The data selected is going to be from the moviesgraded3.csv file found on the website at

`https://faculty.stat.ucla.edu/handcock/216/datasets/`

inside this dataset there will be a couple of features we want to select, some we would like to predict, and others that we may ignore. Since this dataset has only 168 data points, when we exclude the NaN values, we need to make sure that our feature space is has far less dimensionality than our data points. There are two types of features that we can have categorical and numerical that are selected below:

- numerical: "prodcost", "salesopen", "salesus", "salesos"
- categorical: "dist"

our end goal is to predict "grade" so we encode each grade with a numerical value and have our model predict the probability of being assigned that value. In addition we need to scale our numerical values by standardizing them as they all have vastly different ranges.

3.2 Creating the Model

The model we will create will be a fully connected network shown as code in figure 7

```
class FCN(nn.Module):
    def __init__(self, num_features, num_labels, hidden_layers = [15,20]):
        super(FCN, self).__init__()
        self.fc1 = nn.Linear(num_features, hidden_layers[0])
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_layers[0] , hidden_layers[1] )
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_layers[1], num_labels)

    def forward(self, x):
        out = self.fc1(x)

        out = self.relu1(out)
        out = self.fc2(out)
        out = self.relu2(out)
        out = self.fc3(out)
        out = nn.functional.softmax(out, dim=1) #softmax function
        return out
```

Figure 7: Fully Connected Network Code

We create a class FCN that has several layers. The first layer is a Linear layer that specifies how many features we want in and how many come out, This is since nn.Linear is a special function that creates a matrix that represents the weights and connections to each model, wheres num features corresponds to the number of rows and output dimension corresponds to the number of columns.

Then we define a forward function. This explains how we want our model to pass in the data through the layers. Shown also is the softmax layer which turns the inputs into an output with a probability distribution

3.3 Training the model

While this is certainly not the easiest code to write in general there will be patterns that all training methods have. The first one is defining your Loss function, in this case the Criterion, followed by the Gradient Descent Method you want to use. in this case it is ADAM which is a variant of Stochastic Gradient Descent.
The next step is to loop through a set amount of epochs and pass the data through our model. Using

```python
criterion = nn.CrossEntropyLoss()  # Cross Entropy Loss
optimizer = optim.Adam(model.parameters(), lr=0.001) #Adam Optimzer, based off of SGD
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0.0
    for inputs, targets in train_loader:
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        # L2 penalty (weight decay)
        if regression_penalty != None:
          loss += lam * regression_penalty(model)
        else:
          loss = loss

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Figure 8: Training Code for Model

the forward function in this case we can just call the model to call the forward function with the data as inputs. We can get the loss from that first pass and apply the L1 or L2 penalty if needed. Then we just use backpropogation with the optimizer or gradient descent method to update our weights.

3.4 Comparing Results

In figure 9 we see that most values of lambda eventually converge to an accuracy around 0.59. the only outliers are for the l1 penalty as when lambda gets larger the penalty because greater than the original loss function used as shown in equation 16. The last value when lambda = 100 is caused by NaN issues from floating point conversion. In general lambda is best to be kept at a small value.

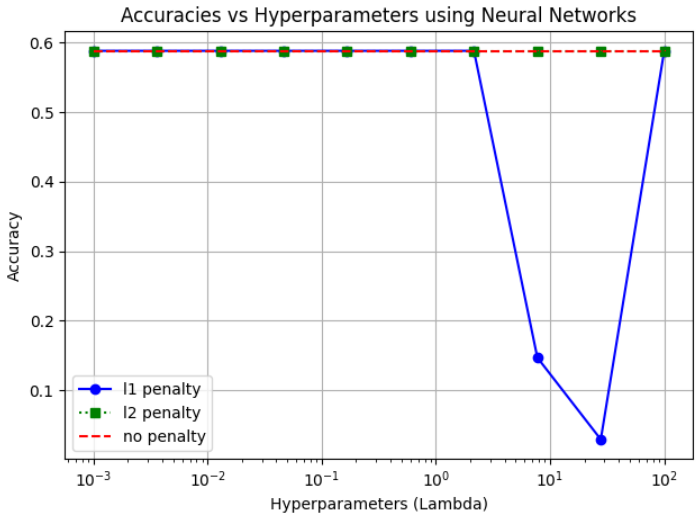

Figure 9: Accuracy using Neural Networks

What is interesting to note is that in figure 9 we see that the Neural Network is performing on par without any penalty being used. This is due to the softmax layer as it already turns the final output into a probality distribution. This might make us wonder why we need a penalty then, if in the case of figure 9 choosing a relatively large lambda can cause issues.

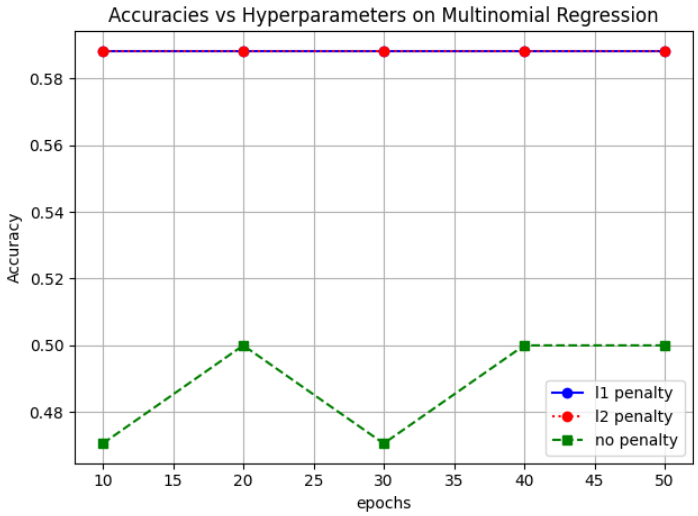

Figure 10: Accuracy using SKLEARN Multinomial Regression

The reason is shown in figure 10. Using the base model of multinomial regression from SKLEARN we are able to select which penalty we would like to use. We can see that the model converges to a solution in a relatively short amount of epochs when we have either penalty used, however the model is not able to converge to the optimal solution without the penalty, despite the increasing amount of epochs In this instance, both penalties converged to a similar value that matches our Neural Network.

4 Conclusion

Overall we were able to use data on movies to create a Neural Network that can accurately classify the Grade of the film based on categorical and numerical features. Neural Networks are a powerful tool as was shown that even without a penalty the optimal solution was able to be discovered. This does not come without its drawbacks as Neural Networks are prone to over fitting, even when there is a penalty attached. This is often known as the Bias-Variance Trade off that all models are affected by. This was a very small sample in addition there were some issues as a result of that namely, it being difficult to encode all the categorical values since most models work better when the number of samples is far larger than the number of features. The purpose of this paper is to give people a theoretical understanding of the basics of Neural Networks and Deep Learning, as well as demonstrate the effectiveness of Neural Networks and Regularizers.