**Text Classification using TensorFlow**

This assignment as we know can be done using multiple techniques like using **neural networks(TensorFlow or Pytorch), Bert, logistic regression, naive bayes model, etc**. I decided to go with neural networks as I have been studying the topic and training models for a while now using TensorFlow and I feel it is much easier to tune hyperparameters make pipelines for accurate models.

My initial approach to this assignment consisted of understanding the problem statement, followed by **preparing and exploring the data** and ultimately finding the required insights (accuracy/precision/recall in this case). Then I tuned the hyperparameters (**hyperparameter** **optimisation**) to maximise the results in the favour of the problem statement, since there are no real features, only linguistic ones, feature selection could not be performed to further improvise on the approach.

Breakdown, to go in detail, I have started by importing the essential libraries followed by reading the data as asked in the part 1 of the assignment. Then I have tried to explore the dataset myself by using functions like ‘**.describe**’**,** ‘**.summary**’ **and** ‘**.info**’. That gives me the volume/shape, features, missing values (if any). That allowed me to have a look at the dimensions and inspect the dataset. Furthermore, to part 1, I then split the dataset into training and testing set using the **frac** parameter and setting it to 0.8 which makes sure that once ready, the model trains on 4000 instances and tests on the rest of 10000 of 50000 instances as mentioned in the problem statement.

Next, I pre-process the data, as asked in part 2 of the assignment. Starting with **tokenizing** as it is one of basic steps, which basically splits words into separate tokens using space/period as a delimiter. The parameter in keras tokenizer – num\_words has been set to 8000 giving the most common 8000 words amongst the word index which is as its name suggests the index of all the words. Using the len feature we find out the length of the index and add 1 to it giving us **124252** distinct tokens.

Since I have used neural networks here, it is only fair I briefly introduce the topic. Neural networks are a part of Artificial Intelligence used for predictive modelling and adaptive control using multiple layers of biological/artificial neurons stacked upon each other. The successive layer feeds off the result of the predecessor layer and hence combines to give an output.

While using TensorFlow and neural networks, the ideal model to go with a sequential model which is a linear stack of layers, it can be created using the **sequential**() function at the end of the ‘tf.layers.’. Next layer(first official layer) we add is the embeddings which turns positive integers (indexes) into dense vectors of fixed size. Next, we add a layer of **Global average pooling** or just pooling, which smoothens out the function by taking the average values and not the maximum ones into consideration thereby giving a better pre-processed data from vectors to further work on. The next layer is the **dense** layer, which is what I explained earlier, neurons feeding off information from their previous connection all the way back to the input and hence actually training the model. Multiple of these can be added keeping in mind the desired time at hand as it is directly proportional to the number of dense layers. But so is the accuracy, only until the point where it starts overfitting due to too much generalisation.

I have used ‘**Relu**’ (Rectified linear unit) as my activation function which is the most commonly used activation function in the deep learning models. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. So, it can be represented as f(x)=max(0,x) .

When compiling the model, I have used ‘**adam**’ as my optimizer. Adam realizes the benefits of both **AdaGrad** and **RMSProp**. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). The loss function I have chosen is **Binary cross entropy** which calculates the loss between the actual and predicted labels. Last the metric is accuracy. In Keras, we can see the summary of the model using the ‘**.summary**’ function and it tells us all the details of the layers and the model itself.

I also implemented a **callback** function, which makes sure the model stops running for the extra epochs(iterations) if it reaches the target accuracy in lesser epochs, saving time and GPU power both. It monitors the validation accuracy. While actually fitting the model the values of the epochs and the batch size can be given, deciding how many vectors are evaluated at once and how many epochs do they go for. I started with 10, ultimately coming down to **5** as after that point it was only a marginal difference for twice as much time.

**Batch size** is one of the most important hyperparameters to tune in modern deep learning systems. Usually, we tend to make the batch sizes larger to speedup the process but that leads to poor generalization and ultimately less accurate models when tested on test data. It basically is a trade off between time and accuracy. I believe as long as the model gives me around >=**80**% accuracy, it is a good model. I have chosen the batch size to be 128, which is a bit unorthodox as keeping it as **32** or **64** gives the best generalisation but **128** seemed to work fine with this dataset and give a decent score proving my hypothesis.

Ultimately, I evaluated the model, and received the highest accuracy of **90**% while training and when tested, the accuracy of the model was **86**%. That tells be the dataset if ran for a greater number of epochs eventually will overfit giving a greater test accuracy than train accuracy meaning that simply the model is too used to the data and will perform poorly on a complete new set of vectors.

Further ideas which can be explored to improve the result.

* Introducing stemming and removing special characters.
* Implementing stopwords and removing the parsed html/tags.
* Using k fold cross validation, we can use 0.8 ratio for training data and 0.2 as test data to fine tune the model which will improve the accuracy furthermore.
* Using a graph, we can plot the accuracy curve and loss curve seeing the potential overfitting I have mentioned.

The only reason I have just mentioned these ideas as I write this is because I was short on time, but I am working on them after the deadline which I will show in my next submission hopefully if need be.