# Assignment Two: Sentiment Classification for social media

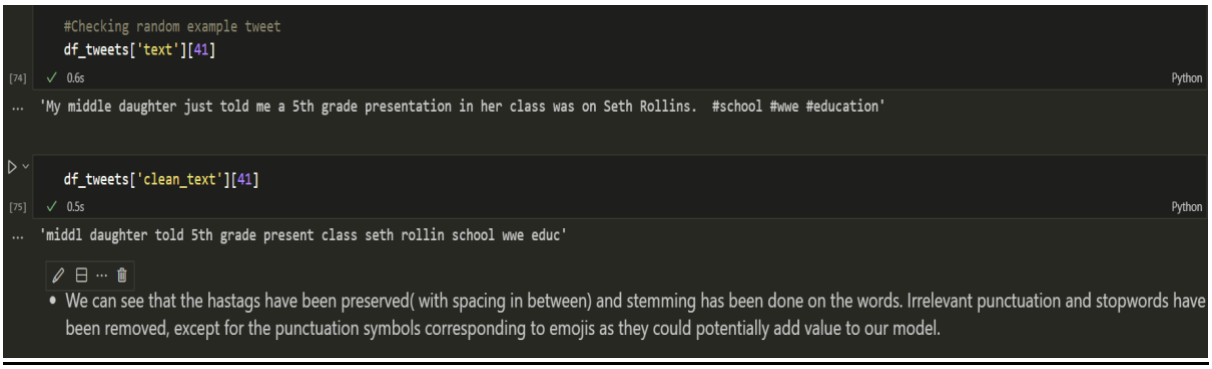
## Geo George University ID : 2135337

## EDA

- Firstly, we load in the training data 'twitter-training-data.txt' into a dataframe using pandas. This enables easy manipulation and greater interpretability. Then we move on to explore the data (EDA). Here, we observe that the target class i.e sentiment is imbalanced and has higher number of 'neutral' values than 'positive' or 'negative'.
- Although techniques like SMOTE() can be used to oversample and solve the class imbalance we do not use that approach here, but keep this in mind while doing our predictions.
- Next, we look at the length of the tweets by making a new column for the same. We can see that there are some outliers that are either too long or too short.

## Text pre-processing

- We define a function to pre-process the tweets. Each tweet is first lowered. Reusing regex code from assignment 1, we remove urls that start with http or https or www.
- It is noticed that there are a few html reference characters of the form '&'. These are removed as well.
- @ mentions are removed with the text accompanying it as these are usually names and don't give us much semantic information.
- Except for the punctuation used in emojis like (): etc. all other non-alphanumeric and punctuation is removed.
- We use a special tokenizer called the TweetTokenizer to tokenize our texts. This handles emojis, hashtags and other elements of a tweet well.
- We also remove all stopwords and perform stemming using the SnowballStemmer, which is an enhanced version of the standard porter stemmer. We use stemming instead of lemmatization because we are not really interested in meaningful words here as this is a sentiment analysis question. We merely want to find out the sentiment of the tweet, so the words need not make sense.



## Classifiers

### Multinomial Naïve Bayes

- Although TF-IDF is also a viable option, since multinomial naïve bayes is a probabilistic model, it is better to use bag of words model (BOW) instead.
- For BOW we extract unigram, bigram and trigram features using the ngram_range parameter in the CountVectorizer. We also set binary=True as it is recommended when using probabilistic models like Naïve Bayes, max_features is set to 20000.
- We then fit the model and test our predicitions on the development data after applying the same preprocessing function on it, use bow.transform to get the word vectors here.
- We obtain an accuracy of 62.4 % on our development data.
- Macro-average f1 scores for 3 testsets:

```
twitter-test1.txt (clf_nb): 0.517
            positive  negative  neutral
positive    0.651     0.071     0.277
negative    0.188     0.596     0.216
neutral     0.284     0.152     0.564
```

```
twitter-test2.txt (clf_nb): 0.545
            positive  negative  neutral
positive    0.704     0.062     0.234
negative    0.176     0.598     0.225
neutral     0.373     0.102     0.525
```

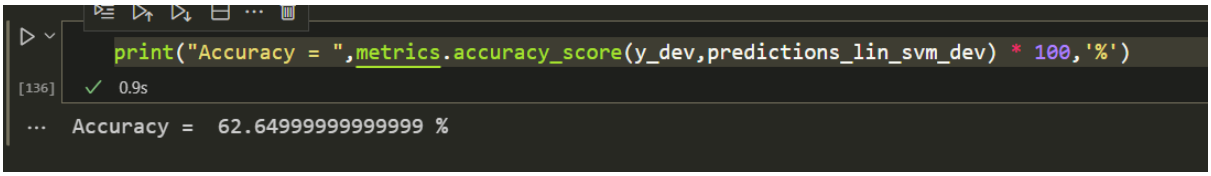
```
twitter-test3.txt (clf_nb): 0.505
            positive  negative  neutral
positive    0.662     0.064     0.273
negative    0.223     0.524     0.253
neutral     0.321     0.144     0.535
```

We can see consistent performance across all 3 test sets.
On the other hand when we use TFIDF for Naïve Bayes using the same number of features i.e 20000 and n-grams, we get much lower f1 scores. Testset 1 f1 score is shown below.

```
twitter-test1.txt (clf_nb): 0.336
            positive  negative  neutral
positive    0.695     0.081     0.224
negative    0.143     0.857     0.000
neutral     0.295     0.184     0.521
```

## Linear SVC model

- We use TFIDF instead of bag of words when it comes to the svm model (LinearSVC). TF-IDF model contains information on the more important words and the less important ones as well and considers the frequency of the word in the entire corpus. This makes it particularly useful in ML models.
- We intialize tfidf vectorizer with max_features = 40,000 and ngram_range = (1,3) to generate extract unigram, bigram and trigram features.
- Here, n_samples > n_features so we set the dual parameter = False. Also, class_weights are automatically balanced used the class_weight='balanced' parameter. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data
- On the development dataset we obtain an accuracy of 62.649 %




- Evaluating on testsets we obtain f1 scores as –

```
twitter-test1.txt (clf_lin_svm): 0.549
            positive  negative  neutral
positive    0.688     0.065     0.247
negative    0.184     0.538     0.278
neutral     0.286     0.134     0.581
```

```
twitter-test2.txt (clf_lin_svm): 0.590
            positive  negative  neutral
positive    0.722     0.041     0.237
negative    0.215     0.554     0.232
neutral     0.405     0.083     0.511
```

```
twitter-test3.txt (clf_lin_svm): 0.531
            positive  negative  neutral
positive    0.698     0.052     0.250
negative    0.234     0.477     0.289
neutral     0.318     0.131     0.550
```

## Random Forest model

- Same as above we utilize TFIDF vectors for our random forest model. We initialize the random forest with 30 trees, max tree depth as 30 and class_weight = 'balanced'. The class_weight parameter helps to take care of the class imbalance.
- Accuracy on development set = 57.65 %
- Macro averaged f1 score on testsets = 0.473, 0.503, 0.452.

It is clear that Multinomial Naïve Bayes and LinearSVC are the better models for this scenario.

## LSTM

- The torchtext library enables us to load tab separated csv files like the training data in our case
- We specify a TEXT and a LABEL corresponding to the tweets and sentiment columns.
- We load in the development data as the test data and build vocabulary for the tweets and the label from the training data using the pre-trained glove embedding vectors.
- We set the max_size as 25000 while building the vocabulary we can see that the size of TEXT vocab is only 10152 and size of Label vocabulary is 3 corresponding to the three sentiments.
- To understand which sentiment corresponds to which number we use the LABEL.vocab.stoi, so neutral = 0, positive = 1, negative = 2.
- We use the GPU for training

Model Architecture

The nn module from torch is the base model for all models. We define the model class.

The constructor has the definitions for the embedding layer for which as per question the requires_grad_ has been set to False so as to freeze the embedding layer and prevent change of weights during training.

We also have an LSTM layer and 2 dense layers. The RelU activation function is used. Also, we make use of dropout regularization with a dropout of 0.2 to ensure overfitting doesn't occur.

We use adam optimizer and initially set the learning rate to 0.1 but this is adaptively changed if the metric is stagnant using a scheduler.

The loss criterion chosen is crossentropyloss because it is a multiclass classification problem.

We use a batch size of 100 and train the model on 20 epochs,

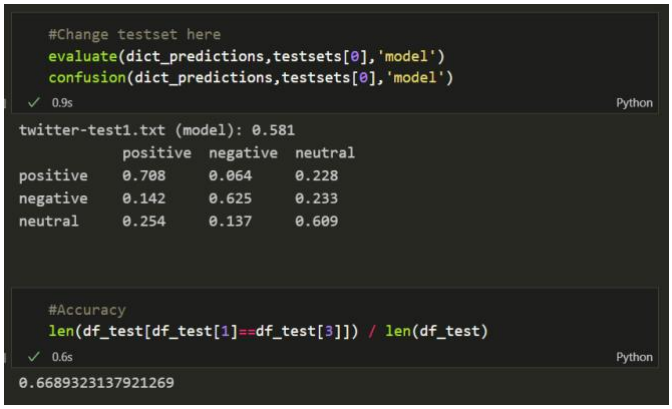
We can clearly see that the model doesn't converge and the loss remains the same even though the scheduler keeps reducing the learning rate. Understanding that the initial learning rate set was too high, we reduce it to 0.001 and try again.

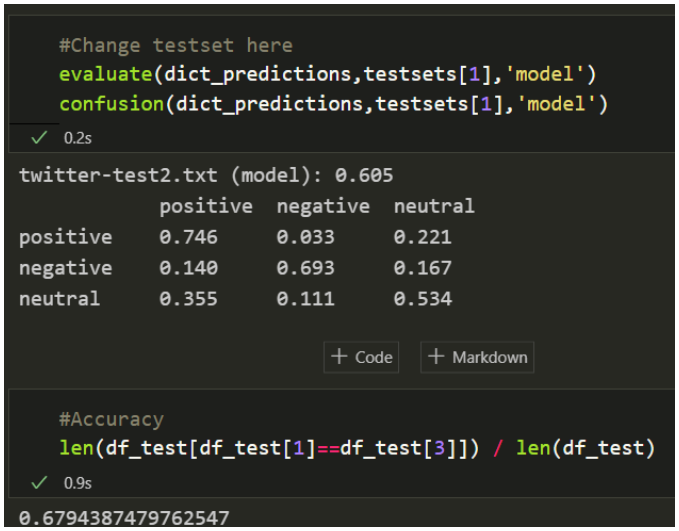
This time the loss reduces and the number of correct samples in each epoch increases.

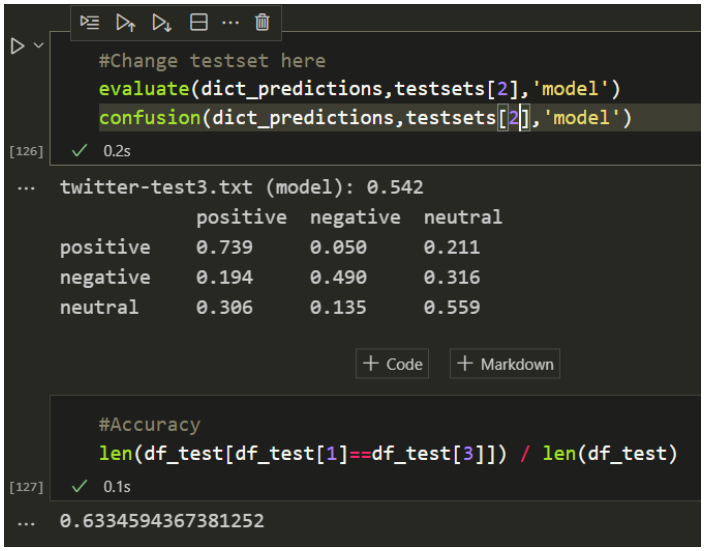
We define a function named predict to predict our model output given a tweet as input. It tokenises the tweet using spacy tokenizer. Since the model gives a tensor output of length 3,

using argmax we can find out the most likely sentiment among index 0,1,2 ( 0 = neutral, 1 = positive, 2 = negative)

The macroaverage f1 score and the confusion matrix for the 3 testsets are listed below:




To improve our model we increase the epochs to 50 and run the model again.

Now we can see the f1 scores are better over the test sets and the accuracy is better too. The screenshots below show the f1 scores over all 3 test sets.

```
#Change testset here
evaluate(dict_predictions,testsets[0],'model')
confusion(dict_predictions,testsets[0],'model')
```

✓ 0.1s

```
twitter-test1.txt (model): 0.562
           positive  negative  neutral
positive     0.735     0.055     0.210
negative     0.166     0.675     0.159
neutral      0.246     0.153     0.601
```

```
#Accuracy
len(df_test[df_test[1]==df_test[3]]) / len(df_test)
```

✓ 0.7s

```
0.6745964316057774
```

```
#Change testset here
evaluate(dict_predictions,testsets[1],'model')
confusion(dict_predictions,testsets[1],'model')
```

✓ 0.4s

```
twitter-test2.txt (model): 0.576
           positive  negative  neutral
positive     0.771     0.046     0.183
negative     0.132     0.714     0.154
neutral      0.352     0.107     0.541
```

```
#Accuracy
len(df_test[df_test[1]==df_test[3]]) / len(df_test)
```

✓ 0.6s

```
0.6864543982730706
```

```
    #Change testset here
    evaluate(dict_predictions,testsets[2],'model')
    confusion(dict_predictions,testsets[2],'model')
```

✓ 0.6s

```
twitter-test3.txt (model): 0.546
            positive  negative  neutral
positive    0.762     0.048     0.190
negative    0.177     0.575     0.248
neutral     0.295     0.142     0.564
```

```
    #Accuracy
    len(df_test[df_test[1]==df_test[3]]) / len(df_test)
```

✓ 0.8s

```
0.6540563261874738
```