Introduction to Natural Language Processing

Natural Language Processing (NLP): The discipline of computer science, artificial intelligence and linguistics that is concerned with the creation of computational models that process and understand natural language. These include: making the computer understand the semantic grouping of words (e.g. cat and dog are semantically more similar than cat and spoon), text to speech, language translation and many more.

Sentiment Analysis: It is the interpretation and classification of emotions (positive, negative and neutral) within text data using text analysis techniques. Sentiment analysis allows organizations to identify public sentiment towards certain words or topics.

We'll develop a **Sequence model** that can perform **Sentiment Analysis** to categorize a tweet as **Positive or Negative.**

Importing Dataset

The dataset being used is the **sentiment140 dataset**. It contains 1,600,000 tweets extracted using the **Twitter API**. The tweets have been annotated **(0 = Negative, 4 = Positive)** and they can be used to detect sentiment.

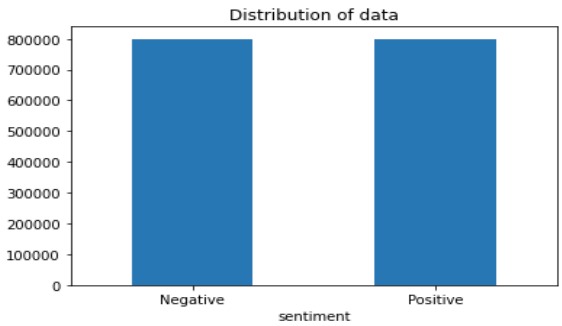
It contains the following 6 fields:

1. **sentiment**: the polarity of the tweet *(0 = negative, 4 = positive)*
2. **ids**: The id of the tweet
3. **date**: the date of the tweet
4. **flag**: The query. If there is no query, then this value is NO_QUERY.
5. **user**: the user that tweeted
6. **text**: the text of the tweet

Furthermore, we're remapping the **sentiment** field so that it has new values to reflect the sentiment. **(0 = Negative, 1 = Positive)**

We're **plotting the distribution** for the dataset to see whether we have equal number of positive and negatives tweets or not.

As we can see from the graphs, we have equal number of Positive/Negative tweets. Both equaling to 800,000 tweets. This means our dataset is **not skewed** which makes working on the dataset easier for us.



Preprocessing the Text

Text Preprocessing is traditionally an important step for **Natural Language Processing (NLP)** tasks. It transforms text into a more digestible form so that deep learning algorithms can perform better.

Tweets usually contains a lot of information apart from the text, like mentions, hashtags, urls, emojis or symbols. Since normally, NLP models cannot parse those data, we need to clean up the tweet and replace tokens that actually contains meaningful information for the model.

As much as the preprocessing steps are important, the actual sequence is also important while cleaning up the text. For example, removing the punctuations before replacing the urls means the regex expression cannot find the URLs. Same with mentions or hashtags. So make sure, the actual sequence of cleaning makes sense.

Splitting the Data

Machine Learning models are trained and tested on different sets of data. This is done so to reduce the chance of the model overfitting to the training data, i.e it fits well on the training dataset but a has poor fit with new ones.

sklearn.model_selection.train_test_split shuffles the dataset and splits it into train and test dataset.

The Pre-processed Data is divided into 2 sets of data:

1. **Training Data:** The dataset upon which the model would be trained on. Contains 95% data.
2. **Test Data:** The dataset upon which the model would be tested against. Contains 5% data

Creating Word Embeddings using Word2Vec model

Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc. Loosely speaking, word embeddings are **vector representations** of a particular word.

Word2Vec was developed by Google and is one of the most popular technique to learn word embeddings using shallow neural network. Word2Vec can create word embeddings using two methods (both involving Neural Networks): **Skip Gram** and **Bag Of Words (BOW)**.

Word2Vec() function creates and trains the word embeddings using the data passed.

Tokenizing and Padding datasets

Tokenization is a common task in **Natural Language Processing (NLP)**. It's a fundamental step in both traditional NLP methods like **Count Vectorizer** and Advanced Deep Learning-based architectures like **Transformers**.

Tokenization is a way of separating a piece of text into smaller units called **tokens**. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization.

All the neural networks require to have inputs that have the same shape and size. However, when we pre-process and use the texts as inputs for our model e.g. LSTM, not all the sentences have the same length. We need to have the inputs with the same size, this is where the **padding** is necessary.

Padding is the process by which we can add padding tokens at the start or end of a sentence to increase it's length up to the required size. If required, we can also drop some words to reduce to the specified length.

Creating Embedding Matrix

Embedding Matrix is a matrix of all words and their corresponding embeddings. We use embedding matrix in an **Embedding layer** in our model to embedded a token into it's vector representation, that contains information regarding that token or word.

We get the embedding vocabulary from the tokenizer and the corresponding vectors from the Embedding Model, which in this case is the Word2Vec model.

Shape of Embedding matrix is usually the **Vocab Length * Embedding Dimension.**

Creating the Model

There are different approaches which we can use to build our Sentiment analysis model. We're going to build a deep learning **Sequence model.**

Sequence model are very good at getting the context of a sentence, since it can understand the meaning rather than employ techniques like counting positive or negative words like in a **Bag-of-Words model**.

Model Architecture

```
Model: "Sentiment_Model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           27283600
_________________________________________________________________
bidirectional (Bidirectional (None, 60, 200)           160800
_________________________________________________________________
bidirectional_1 (Bidirection (None, 60, 200)           240800
_________________________________________________________________
conv1d (Conv1D)              (None, 56, 100)           100100
_________________________________________________________________
global_max_pooling1d (Global (None, 100)               0
_________________________________________________________________
dense (Dense)                (None, 16)                1616
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 27,786,933
Trainable params: 503,333
Non-trainable params: 27,283,600
_________________________________________________________________
```

1) **Embedding Layer:** Layer responsible for converting the tokens into their vector representation that is generated by Word2Vec model. We're using the predefined layer from TensorFlow in out model.

2) **Bidirectional:** Bidirectional wrapper for RNNs. It means the context are carried from both left to right and right to left in the wrapped RNN layer.

3) **LSTM: Long Short-Term Memory**, it's a variant of **RNN** which has memory state cell to learn the context of words which are at further along the text to carry contextual meaning rather than just neighboring words as in case of RNN.

4) **Conv1D:** This layer creates a convolution kernel that is convolved with the layer input over a single dimension to produce a tensor of outputs.

5) **GlobalMaxPool1D:** layer creates a convolution kernel that is convolved with the layer input over a single dimension to produce a tensor of outputs

6) **Dense:** Dense layer adds a fully connected layer in the model. The argument passed specifies the number of nodes in that layer.

The last dense layer has the activation **"Sigmoid"**, which is used to transform the input to a number between 0 and 1. Sigmoid activations are generally used when we have 2 categories to output in.

Training the Model

Model Callbacks

Callbacks are objects that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc.).

We can use callbacks to write **TensorBoard logs** after every batch of training, periodically save our model, stop training early or even to get a view on internal states and statistics during training.

Model Compile

The Model must be compiled to define the **loss, metrics and optimizer**. Defining the proper loss and metric is essential while training the model.

Loss: We're using **Binary Crossentropy**. It is used when we have binary output categories.

Metric: We've selected **Accuracy** as it is one of the common evaluation metrics in classification problems when the category data is equal.

Optimizer: We're using **Adam**, optimization algorithm for Gradient Descent.

We'll now train our model using the **fit** method and store the output learning parameters in **history**, which can be used to plot out the learning curve.

```
Epoch 1/10
1336/1336 [==============================] - 127s 87ms/step - loss: 0.4618 - accuracy: 0.7785 - val_loss: 0.3897 - val_accuracy: 0.8236
Epoch 2/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.4001 - accuracy: 0.8164 - val_loss: 0.3758 - val_accuracy: 0.8314
Epoch 3/10
1336/1336 [==============================] - 114s 85ms/step - loss: 0.3868 - accuracy: 0.8241 - val_loss: 0.3677 - val_accuracy: 0.8357
Epoch 4/10
1336/1336 [==============================] - 114s 86ms/step - loss: 0.3773 - accuracy: 0.8296 - val_loss: 0.3632 - val_accuracy: 0.8391
Epoch 5/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.3719 - accuracy: 0.8316 - val_loss: 0.3618 - val_accuracy: 0.8396
Epoch 6/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.3672 - accuracy: 0.8345 - val_loss: 0.3604 - val_accuracy: 0.8407
Epoch 7/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.3629 - accuracy: 0.8372 - val_loss: 0.3554 - val_accuracy: 0.8431
Epoch 8/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.3610 - accuracy: 0.8377 - val_loss: 0.3554 - val_accuracy: 0.8427
Epoch 9/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.3588 - accuracy: 0.8388 - val_loss: 0.3568 - val_accuracy: 0.8425
Epoch 10/10
1336/1336 [==============================] - 115s 86ms/step - loss: 0.3552 - accuracy: 0.8407 - val_loss: 0.3544 - val_accuracy: 0.8430
```

Evaluating Model

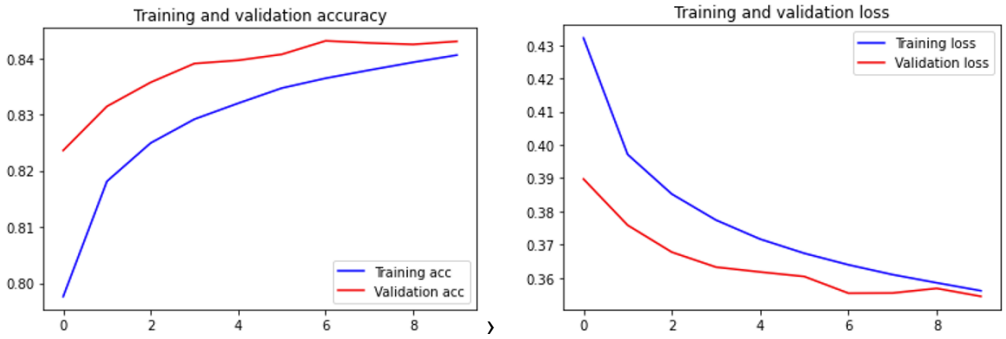
Since our dataset is not **skewed**, i.e. it has equal number of **Positive and Negative Predictions**. We're choosing **Accuracy** as our evaluation metric. Furthermore, we're plotting the **Confusion Matrix** to get an understanding of how our model is performing on both classification types.

```
79/79 [==============================] - 3s 35ms/step - loss: 0.3545 - accuracy: 0.8430

Accuracy of model is : 0.84
Loss of model is : 0.35
```

Printing out the Learning curve

Learning curves show the relationship between training set size and your chosen evaluation metric (e.g. RMSE, accuracy, etc.) on your training and validation sets. They can be an extremely useful tool when diagnosing your model performance, as they can tell you whether your model is suffering from bias or variance.

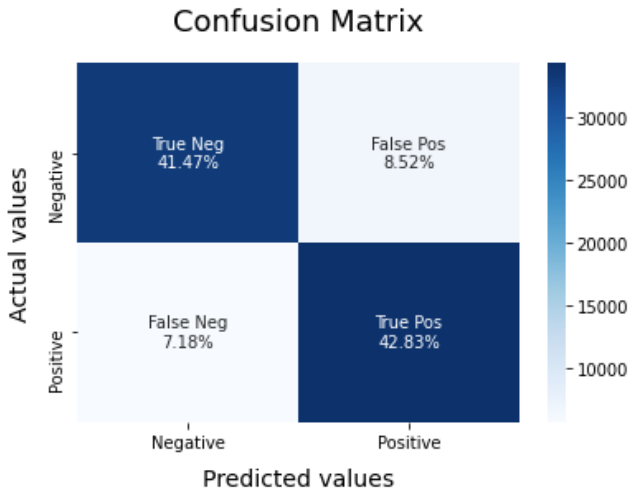


From the training curve we can conclude that our model doesn't have bias nor is it overfitting. The accuracy curve has flattened but is still rising, which means training for more epochs can yield better results.

The Validation loss is lower than the training loss because the dropouts in LSTM aren't active while evaluating the model.

Confusion Matrix

From the confusion matrix, it can be concluded that the model makes more False Negative predictions than positive. This means that the model is **somewhat biased** towards predicting negative sentiment.



Classification Report

Using the classification report, we can see that that the model achieves nearly **84% Accuracy** after training for just **10 epochs.** This is really good and better than most other models achieve.

	precision	recall	f1-score	support
0	0.85	0.83	0.84	39989
1	0.83	0.86	0.85	40011
accuracy			0.84	80000
macro avg	0.84	0.84	0.84	80000
weighted avg	0.84	0.84	0.84	80000

Saving the Model

Saving the **Tokenizer and TensorFlow model** for use later.