Project Overview

In this capstone project, I will explore the basics of the Natural Language Processing (NLP) and demonstrate how to implement a pipeline that combines a traditional unsupervised learning algorithm with a deep learning algorithm to train an unlabeled large text data. Hence, the main objective is going to be to demonstrate how to set up that pipeline that facilitates collection and creation of raw text data, preprocessing and categorizing unlabeled text data to finally training and evaluating deep learning models in Keras.

Throughout this project, the following skills will be practiced:

- How to collect data from Twitter via Twitter API and Tweepy Python package
- How to efficiently read and clean up large text dataset with pandas
- How to preprocess text data using basic NLP techniques and generate features
- How to categorize unlabeled text data
- How to train, compile, fit and evaluate deep learning models in Keras

Step 1: Problem Identification

1.1 Problem Statement

What are the main opinions observed from the tweets related to the 2020 US election? What are people talking about on Twitter two weeks prior to election day?

1.2 Context

October surprise, according to Wikipedia, is a U.S. political jargon and a news event that may influence the outcome of an upcoming November election, particularly one for the U.S. presidency), whether deliberately planned or spontaneously occurring. Tweeter is a great source of unfiltered conversations, opinions and news events directly posted by the individuals themselves compared to the filtered news provided by the media outlets. This project analyses various sentiments from the tweets occurring 2 weeks prior to the election day to identify main topics people are talking about and October surprises.

1.3 Criteria For Success

Achieve at least 75% accuracy in predicting the topics of the tweets.

1.4 Scope of Solution Space

Solution scope will be limited to analysing the tweets related to the 2020 US election. In the modeling section, the main focus will be given to the training and evaluating deep learning models with the text data. While labeling of the tweets is required as part of the project, it is out of the scope of the project to improve the performance of the labeling job.

1.5 Solution Approach

- Collect data from the Twitter with the relevant keywords
- Apply Natural Language Processing (NLP) techniques to clean and preprocess the data
- Distributed computing will be used to preprocess and label the data set
- Train and evaluate SimpleRNN and LSTM models in Keras

1.6 Constraints

- Limitation of computational power/resource
- Limited hands-on experience in advanced deep learning techniques
- Access to the distributed computing platforms and technical challenges

1.7 Stakeholders

General public

1.8 Deliverables

- Final Report
- Final Presentation
- Jupyter notebooks with the code for each stage of the data science method
- Model deployment into production
- Article on Medium / TDS

1.9 Data Sources

At the time of doing this project, the US 2020 election was just around the corner and it made sense to do sentiment analysis of tweets related to the upcoming election to learn about the kind of opinions and topics being discussed in Twitter just about 2 weeks prior to the election day. Twitter is a great source for unfiltered opinions as opposed to the typical filtered news we see from the major media outlets. As such, I will create my own dataset by collecting tweets from Twitter using the Twitter API and the python package Tweepy.

Step 2: Data collection

2.1 Approach

The following approach was applied to collect live tweets from Twitter.

1. Set up data streaming pipeline
2. Stream live tweets
3. Save the tweets in a CSV file

2.2 Data streaming pipeline

Twitter API and the python package Tweepy were used to collect live tweets from Twitter. To do that, Twitter API consumer keys, authorization handler and custom function were set up as follows.

```
# Twitter API consumer keys
access_token = "  insert your key here  "
access_token_secret = "  insert your key here  "
consumer_key = "  insert your key here  "
consumer_secret = "  insert your key here  "

# Twitter API authorization
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

# Custom function to stream live tweets
class MyStreamListener(tweepy.StreamListener):
   """Function to listen and stream Twitter data"""
   def __init__(self, api=None):
       super(MyStreamListener, self).__init__()
       self.num_tweets = 0
       self.file = open("tweets.txt", "w")
def on_status(self, status):
       tweet = status._json
       self.file.write( json.dumps(tweet) + '\n' )
       self.num_tweets += 1
       if self.num_tweets < 20000:
           return True
       else:
           return False
       self.file.close()
def on_error(self, status):
       print(status)
```

2.3 Stream live tweets

The pipeline from section 2.2 was executed to stream live tweets from Twitter. Before doing that, some keywords were identified to collect tweets related to the 2020 US election. So the keywords such as “Biden”, “Trump, “Election” etc. were among some.

The objective here was to collect at least 400,000 tweets to make it a large enough text data and it is computationally taxing to collect all of that at one go. Thus, a pipeline was set up in a way to efficiently stream data in chunks. Notice from the above custom function, it will listen and stream up to 20,000 tweets at most in each chunk. As such, in order to collect over 400,000 tweets, the code was run at least 20 times and each time the data chunk was read and saved into a pandas DataFrame.

```
# Listen and stream live tweets
listener = MyStreamListener()
stream = tweepy.Stream(auth, listener)
stream.filter(track = ['US Election', 'election', 'trump', 'Mike
Pence', 'biden', 'Kamala Harris', 'Donald Trump', 'Joe Biden'])

# Read the tweets into a list
tweets_data_path = 'tweets.txt'
tweets_data=[]
tweets_file_1 = open(tweets_data_path, 'r')

# Read in tweets and store in list: tweets_data
for line in tweets_file_1:
   tweet = json.loads(line)
   tweets_data.append(tweet)

# Close connection to file
tweets_file_1.close()

# Read the data into a pandas DataFrame
names = tweets_data[0].keys()
df1 = pd.DataFrame(tweets_data, columns= names)
```

At the end of 20 separate streaming sessions, the raw tweets were collected into 20 separate pandas DataFrames and were combined into a single DataFrame which was exported as a CSV file to be used in the next steps of the data science pipeline.

```
# Concatenate dataframes into a single pandas dataframe
list_of_dataChunks = [df1, df2, df3, df4, df5, df6, df7, df8,
df9, df10, df11, df12, df13, df14, df15, df16, df17, df18, df19,
df20]
df = pd.concat(list_of_dataChunks, ignore_index=True)
# Export the dataset into a CSV file
df.to_csv('tweets.csv', index=False)
```

Step 3: Data Wrangling

Before the data could be visualized, the dataset must be cleaned and transformed into a format that can be efficiently visualized. Given the dataset with 440,000 rows, one has to find an efficient way of reading and cleaning it. To do that, pandas chunksize attribute could be used to read data from CSV file into a pandas dataframe in chunks. Also, the names of columns could be specified for the columns of interest as opposed to reading the dataset with all of its columns. With the chunksize and smaller amount of columns of interest, the large dataset could be read into a dataframe rather efficiently and quickly. So that is how the data was read and cleaned in this section.

To transform the dataset into a shape required for visualization, the following basic NLP techniques were applied:

1. Extract the only tweets that are in English language
2. Drop duplicates if any
3. Drop missing values if any
4. Tokenize (break the tweets into single words)
5. Convert the words into lowercase
6. Remove punctuations
7. Remove stopwords
8. Remove URLs, the word “twitter” and other acronyms

The approach that was followed to implement the above steps was as follows:

1. Write a custom function that tokenizes the tweets
2. Write another custom function that applies all of the above mentioned cleaning steps on the data.
3. Finally, read the data in chunks and apply these wrangling steps via the custom functions to each of the chunks of the data as they get read.

Pandas chunksize was set to 50,000 which will allow pandas to read 50,000 tweets in each chunk and apply the custom function to clean up the data on those 50,000 tweets before reading the next chunk.

```
# Read and clean the data
use_cols = ['text', 'lang'] # specify the columns
path = 'tweets.csv' # path to the raw dataset
data_iterator = pd.read_csv(path, usecols=use_cols,
chunksize=50000)
chunk_list = []
for data_chunk in data_iterator:
   filtered_chunk = clean_up(data_chunk)
   chunk_list.append(filtered_chunk)
tidy_data = pd.concat(chunk_list)
```

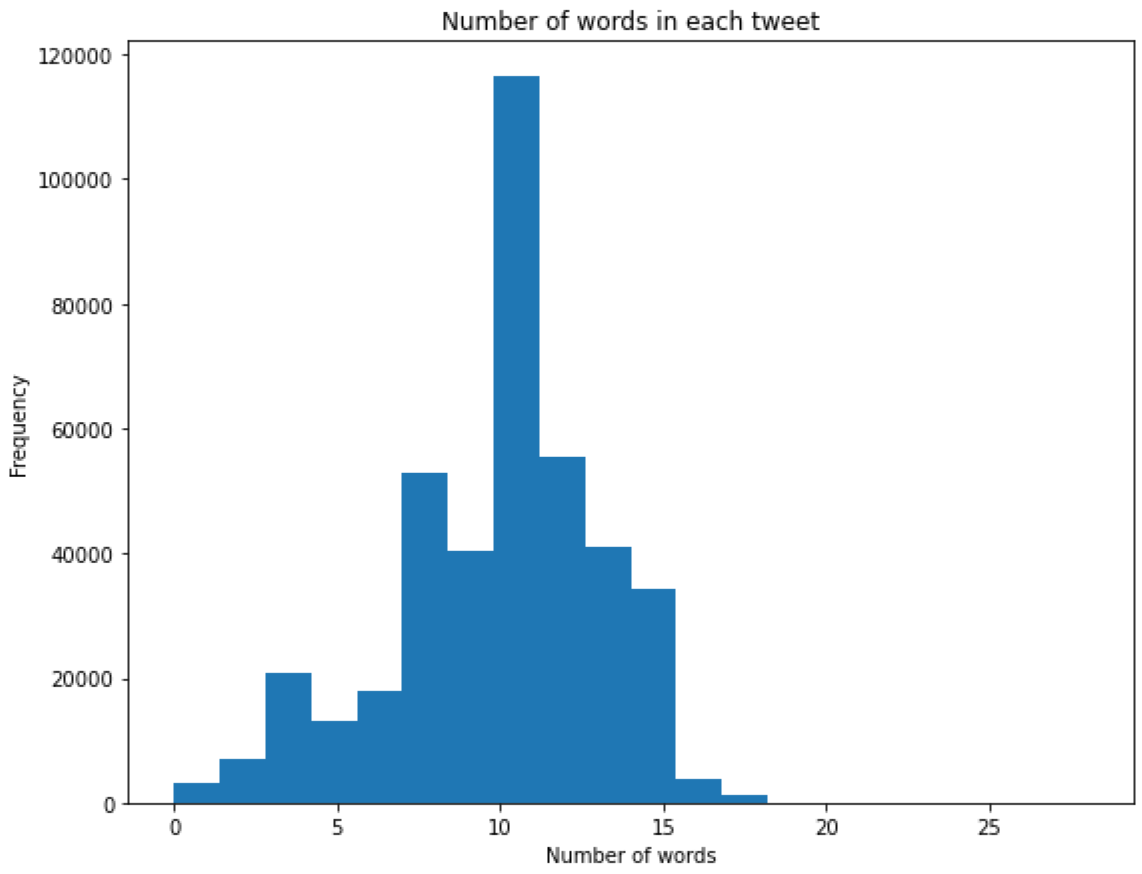
Output of this process produces a tidy data that is ready to be visualized in the next step.

Step 4: Exploratory Data Analysis

In this section, I explored and visualized the following attributes of the data to understand the nature of the dataset.

1. Number of words in a tweet
2. Average length of a word in a tweet
3. Unigram
4. Bigram
5. Trigram
6. Wordcloud

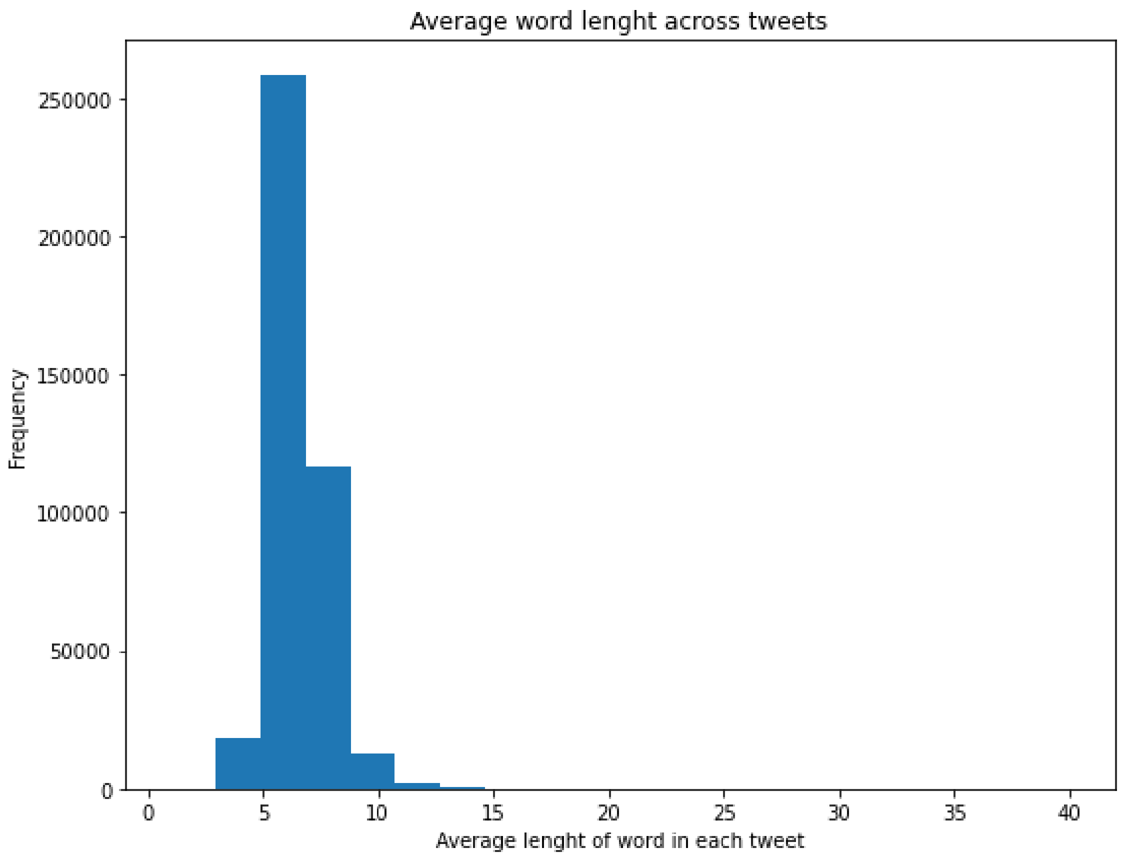
4.1 The number of words in a tweet



It appears that the number of words in each tweet range from 1 to 19 works and on average it falls between 10 to 12 words.

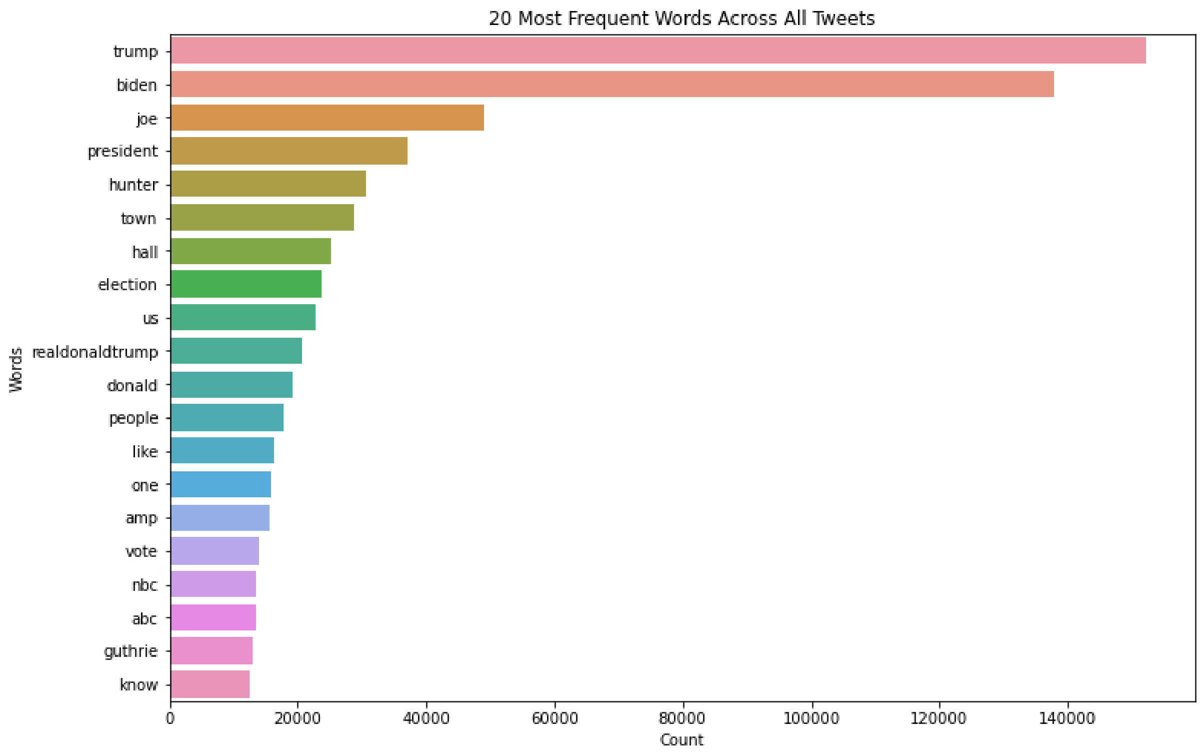
Next chart will show the average length of a word in a given tweet.

4.2 Average length of a word in a tweet



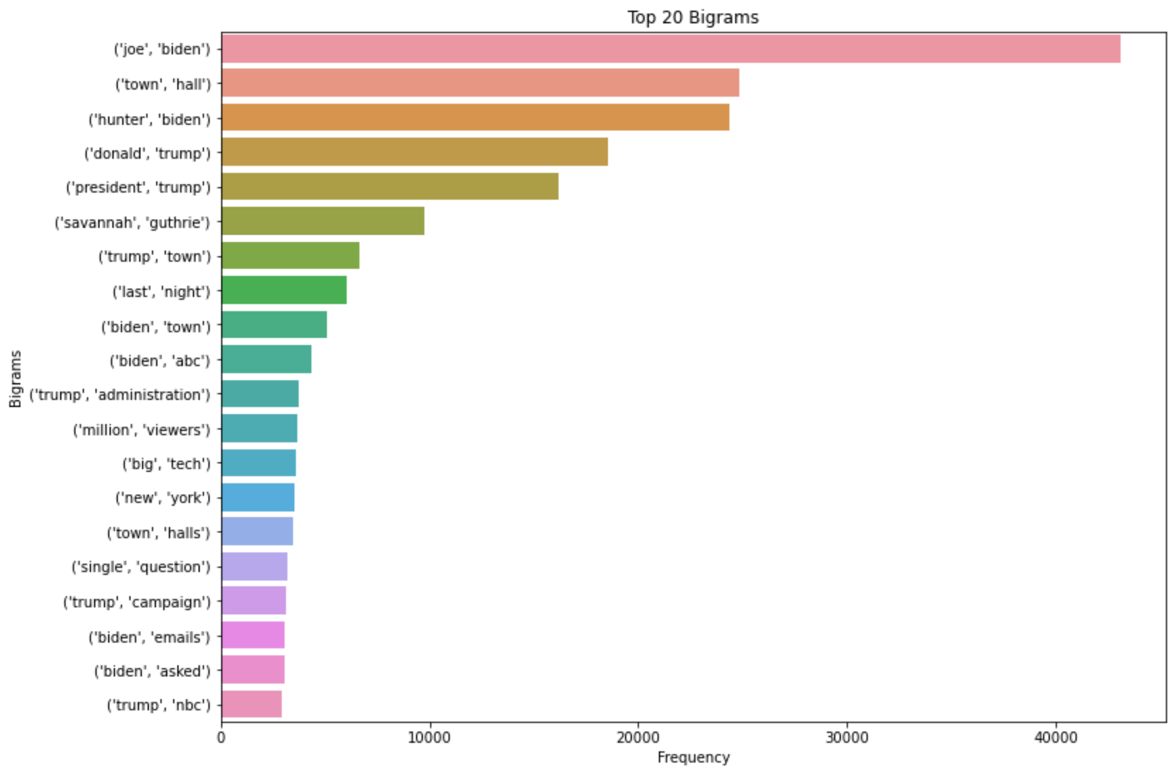
The average number of characters in a word in a tweet appears to range from 3 to 14 characters and on average occurring between 5 to 7 characters. People probably choose short words to express their opinions in the best way they can within the 280 character limit set by Twitter.

4.3 Unigram

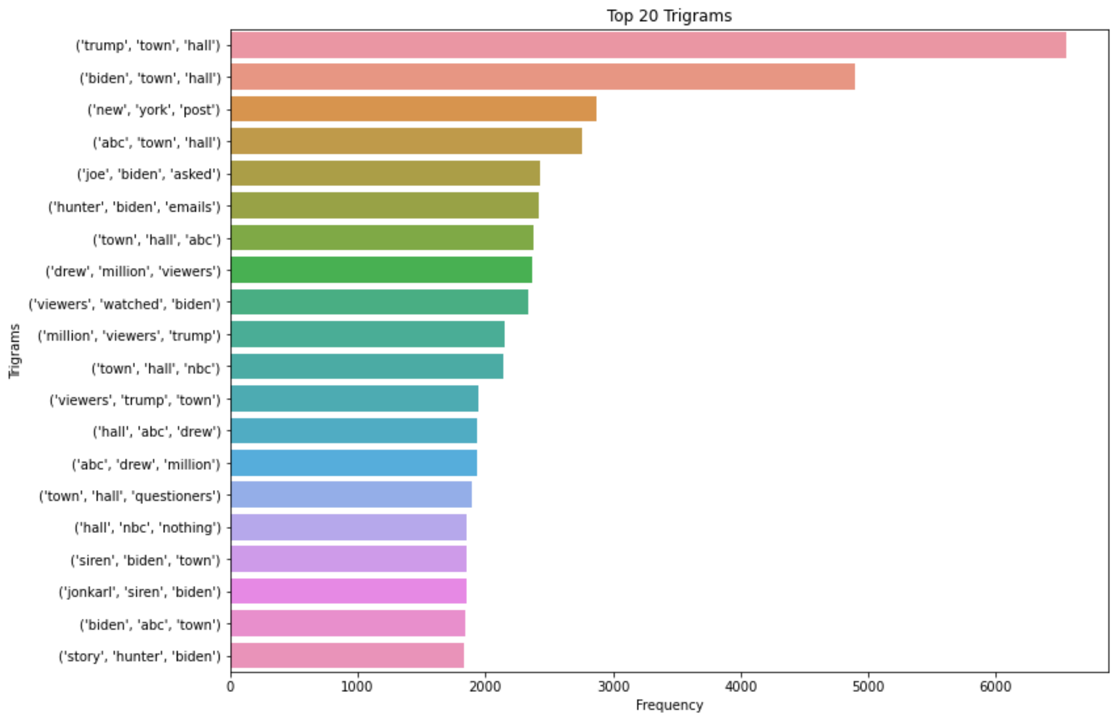


As expected, the words “trump” and “biden” dominate the 2020 US election related tweets that were pulled between Oct 15 and Oct 16.

4.4. Bigram



4.5 Trigram



4.6 Wordcloud

upenn
competing
object
made
joe
tedcruz
text
single
asked
actbrigitte
says
qanon
dangerous
yo
business
Length
debate
dtype
town
machismo
encyclopedia
president
senators
trump
quest
hunter
bidding
de
month
joke
interrupted
rudygiuliani
halls
bradleywhitford
briantylercohen
trumpwarroom
democrat
school
questions
corrupt
hate
told
semites
biden
Name
dancrenshawtx

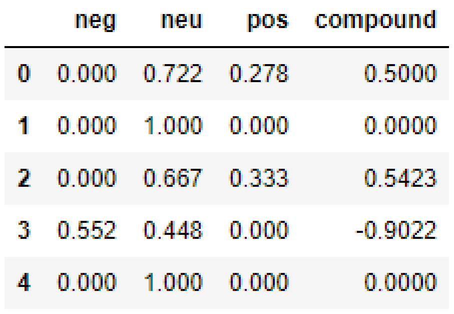
Step 5: Categorize the unlabeled text

The tweets were collected from Twitter in the form of raw tweets hence the data came as unlabeled. In order to predict the category or type of tweets with the deep learning model at the later section, first the unlabeled data must be categorized into classes.

That is, this unsupervised learning problem must be converted into a supervised learning problem for the deep learning architectures. To do that, the k-means clustering algorithm was used to categorize the unlabeled text data.

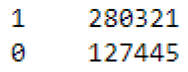
First, feature extraction was performed by using the VADER package and next the k-means model was trained on the dataset using the new features. The following table shows the first 5 rows of the newly created features from the dataset using the VADER package.

The approach applied here was to label the tweets as being either positive (1) or negative (0) based on the values of the compound. The tweets with a compound value greater or equal to 0.05 will be labeled as positive (1) and otherwise will be labeled as negative (0).



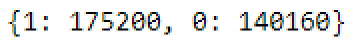
	neg	neu	pos	compound
0	0.000	0.722	0.278	0.5000
1	0.000	1.000	0.000	0.0000
2	0.000	0.667	0.333	0.5423
3	0.552	0.448	0.000	-0.9022
4	0.000	1.000	0.000	0.0000

After the labeling job, k-means clustering algorithm grouped the tweets in clusters of 1 and 0 with the following distribution.



```
1    280321
0    127445
```

The data came out as imbalanced, more than twice the tweets were put in the group 1. This is going to impact the model and as such the dataset must be balanced before it can be trained. To do that, SMOTE will be applied on the dataset in the next step. The data was upsampled and downsampled with SMOTE, the resulting dataset came out to be pretty balanced and looks as follows.



```
{1: 175200, 0: 140160}
```

Step 6: Modeling

The dataset has 440,000 rows and more than 60,000 unique words. It is large and complex enough and my computer couldn't possibly process such a large dataset. So I used Amazon SageMaker p2.xlarge GPU instance to train deep learning models with this dataset.

Two deep learning models were trained on the dataset and compared by their performances. The models are two different variants of RNN architecture, namely SimpleRNN and LSTM. The following approach was followed to train these deep learning models:

1. Tokenizing, padding and sequencing the dataset
2. Split the dataset into training and test sets
3. Train SimpleRNN and LSTM models
4. Evaluate models

6.1 Tokenize, pad and sequence

In this section, the dataset was tokenized, padded and sequenced as required before it can be fed into a deep learning architecture.

```
# prepare tokenizer tokenizer = Tokenizer()
tokenizer.fit_on_texts(tweets)

# integer encode the documents
sequences = tokenizer.texts_to_sequences(tweets)

# pad documents to a max length of 14 words maxlen = 14 X =
pad_sequences(sequences, maxlen=maxlen)
```

Notice that the maxlen argument was set to 14 which is consistent with the maximum length of a word in a tweet as observed above in the EDA section.

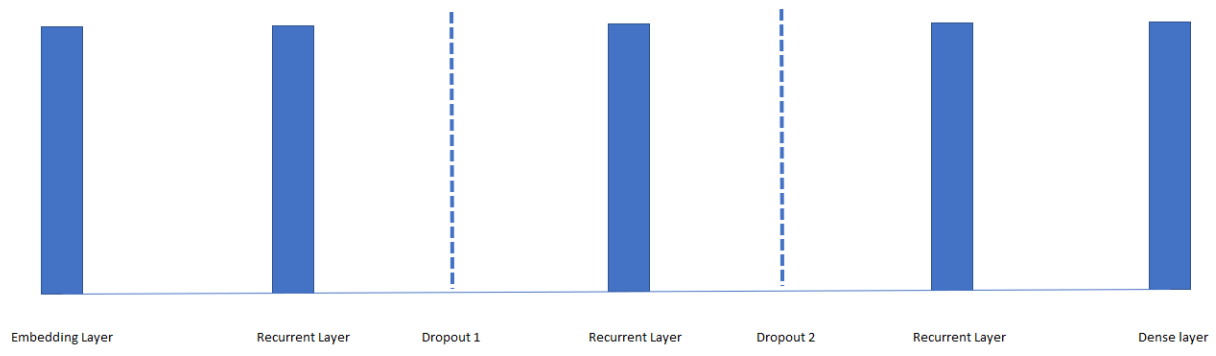
6.2 Split the data into training and test sets

Now that the data is preprocessed and is ready for training, I proceeded to split the dataset into training and test sets with the ratio of 70% to 30% respectively. That is, 30% of the dataset was put away to test the model after the model has been trained on the 70% of the dataset.

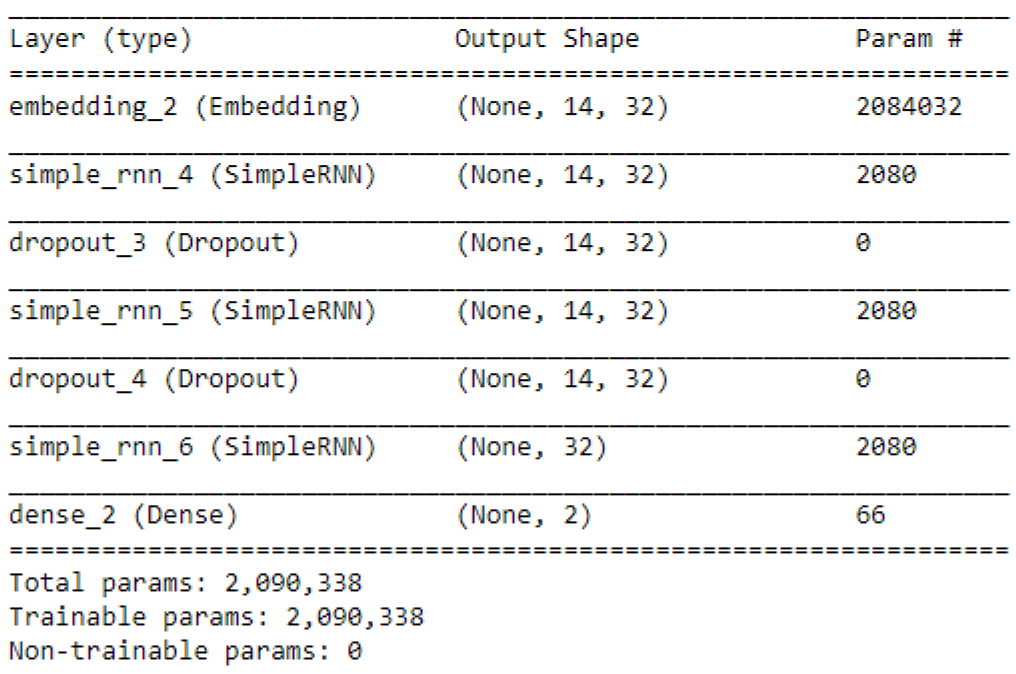
6.3 SimpleRNN and LSTM models

After multiple trials with different variants of RNNs, 3 layer SimpleRNN and 3 layer LSTM models were selected as the final models to be compared against each other. Dropout layers were used in both architectures to control overfitting. Both architectures have embedding layers as their first layer and were used to create word embeddings as an input to the hidden layers. The architecture of both models are symmetric in terms of number of layers and dropout locations in them.

SimpleRNN architecture

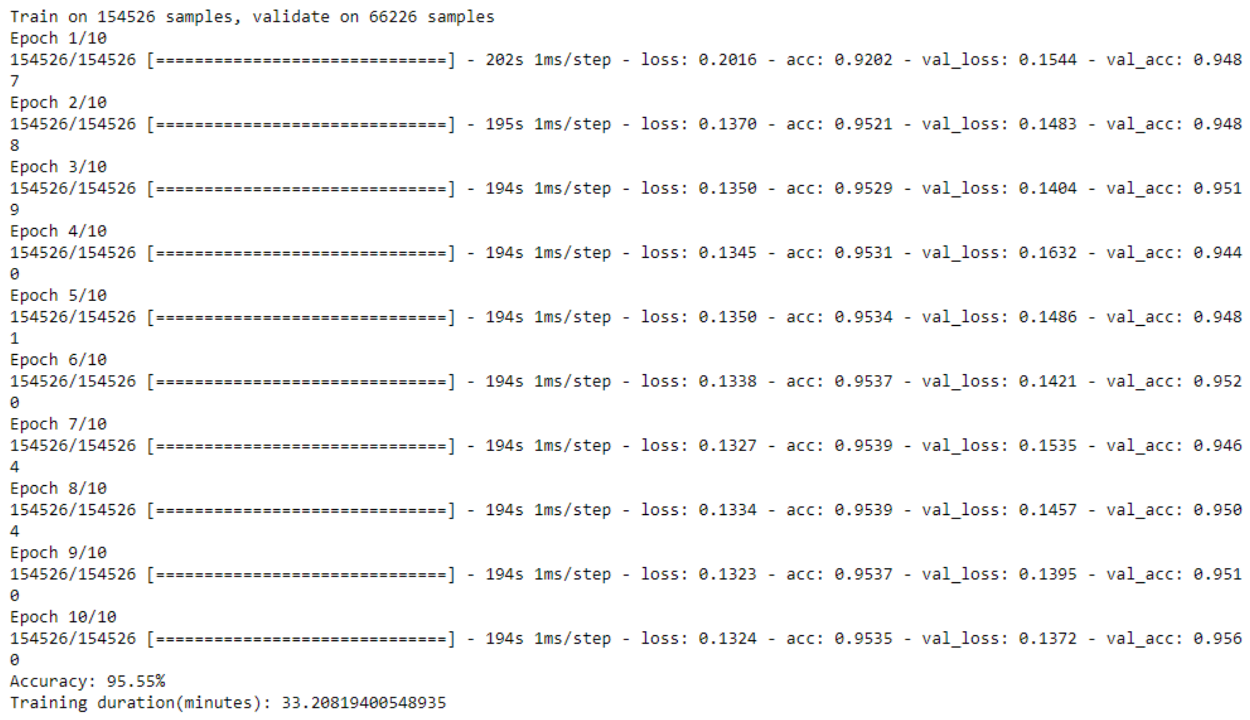


The activation function is set to “tanh” by default in SimpleRNN layers. “Adam” optimizer was used as an optimizer, “binary_crossentropy” was used as the loss function and “accuracy” was used as the metric when compiling the model. The number of epochs was set to 10 when fitting the model on a training set 70% of which was used to train the model while the remaining 30% was used for validation. This is not to be mixed up with the 30% of the original data that was stashed away as a test set.



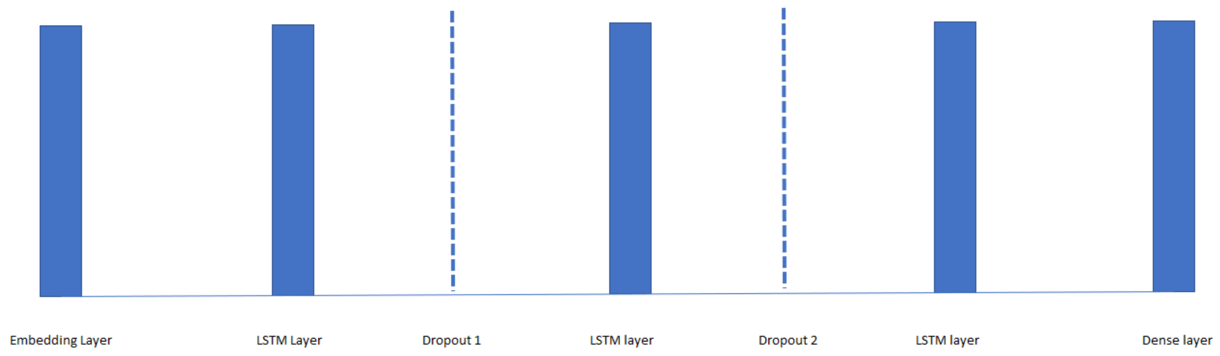
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 14, 32)            2084032
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, 14, 32)            2080
_________________________________________________________________
dropout_3 (Dropout)          (None, 14, 32)            0
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, 14, 32)            2080
_________________________________________________________________
dropout_4 (Dropout)          (None, 14, 32)            0
_________________________________________________________________
simple_rnn_6 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 66
=================================================================
Total params: 2,090,338
Trainable params: 2,090,338
Non-trainable params: 0
```

SimpleRNN results

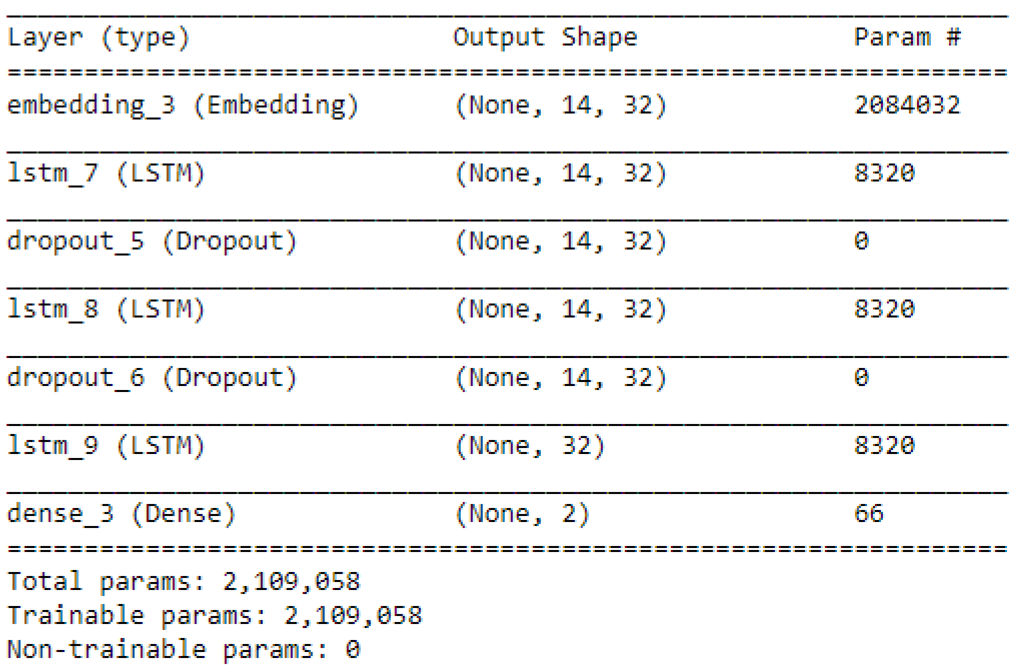


```
Train on 154526 samples, validate on 66226 samples
Epoch 1/10
154526/154526 [==============================] - 202s 1ms/step - loss: 0.2016 - acc: 0.9202 - val_loss: 0.1544 - val_acc: 0.948
7
Epoch 2/10
154526/154526 [==============================] - 195s 1ms/step - loss: 0.1370 - acc: 0.9521 - val_loss: 0.1483 - val_acc: 0.948
8
Epoch 3/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1350 - acc: 0.9529 - val_loss: 0.1404 - val_acc: 0.951
9
Epoch 4/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1345 - acc: 0.9531 - val_loss: 0.1632 - val_acc: 0.944
0
Epoch 5/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1350 - acc: 0.9534 - val_loss: 0.1486 - val_acc: 0.948
1
Epoch 6/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1338 - acc: 0.9537 - val_loss: 0.1421 - val_acc: 0.952
0
Epoch 7/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1327 - acc: 0.9539 - val_loss: 0.1535 - val_acc: 0.946
4
Epoch 8/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1334 - acc: 0.9539 - val_loss: 0.1457 - val_acc: 0.950
4
Epoch 9/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1323 - acc: 0.9537 - val_loss: 0.1395 - val_acc: 0.951
0
Epoch 10/10
154526/154526 [==============================] - 194s 1ms/step - loss: 0.1324 - acc: 0.9535 - val_loss: 0.1372 - val_acc: 0.956
0
Accuracy: 95.55%
Training duration(minutes): 33.20819400548935
```

LSTM architecture



The activation function is set to “tanh” by default in LSTM layers. “Adam” optimizer was used as an optimizer, “binary_crossentropy” was used as the loss function and “accuracy” was used as the metric when compiling the model. The number of epochs was set to 10 when fitting the model on a training set 70% of which was used to train the model while the remaining 30% was used for validation. This is not to be mixed up with the 30% of the original data that was stashed away as a test set.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 14, 32)            2084032
_________________________________________________________________
lstm_7 (LSTM)                (None, 14, 32)            8320
_________________________________________________________________
dropout_5 (Dropout)          (None, 14, 32)            0
_________________________________________________________________
lstm_8 (LSTM)                (None, 14, 32)            8320
_________________________________________________________________
dropout_6 (Dropout)          (None, 14, 32)            0
_________________________________________________________________
lstm_9 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
Total params: 2,109,058
Trainable params: 2,109,058
Non-trainable params: 0
```

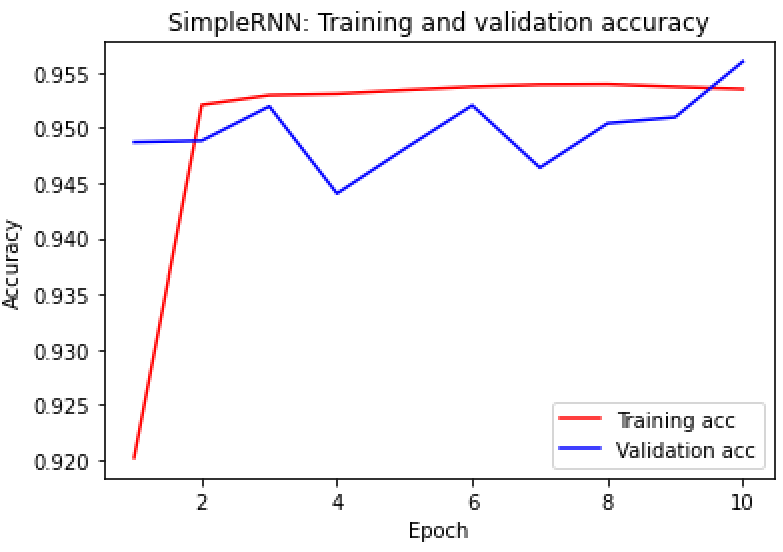
LSTM results

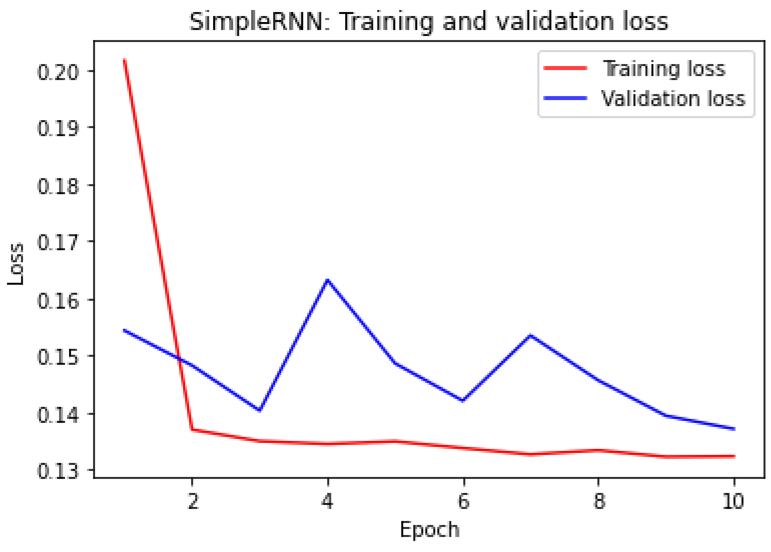
```
Train on 154526 samples, validate on 66226 samples
Epoch 1/10
154526/154526 [==============================] - 418s 3ms/step - loss: 0.1038 - acc: 0.9634 - val_loss: 0.1088 - val_acc: 0.962
6
Epoch 2/10
154526/154526 [==============================] - 418s 3ms/step - loss: 0.1030 - acc: 0.9640 - val_loss: 0.1112 - val_acc: 0.961
8
Epoch 3/10
154526/154526 [==============================] - 419s 3ms/step - loss: 0.1031 - acc: 0.9638 - val_loss: 0.1114 - val_acc: 0.961
7
Epoch 4/10
154526/154526 [==============================] - 418s 3ms/step - loss: 0.1014 - acc: 0.9645 - val_loss: 0.1060 - val_acc: 0.964
1
Epoch 5/10
154526/154526 [==============================] - 418s 3ms/step - loss: 0.0993 - acc: 0.9652 - val_loss: 0.1094 - val_acc: 0.962
1
Epoch 6/10
154526/154526 [==============================] - 418s 3ms/step - loss: 0.0993 - acc: 0.9655 - val_loss: 0.1114 - val_acc: 0.962
4
Epoch 7/10
154526/154526 [==============================] - 418s 3ms/step - loss: 0.0983 - acc: 0.9658 - val_loss: 0.1065 - val_acc: 0.963
0
Epoch 8/10
154526/154526 [==============================] - 419s 3ms/step - loss: 0.0966 - acc: 0.9664 - val_loss: 0.1083 - val_acc: 0.959
9
Epoch 9/10
154526/154526 [==============================] - 422s 3ms/step - loss: 0.0961 - acc: 0.9664 - val_loss: 0.1095 - val_acc: 0.962
5
Epoch 10/10
154526/154526 [==============================] - 425s 3ms/step - loss: 0.0962 - acc: 0.9662 - val_loss: 0.1184 - val_acc: 0.958
3
Accuracy: 96.07%
Training duration(minutes): 71.24150105714799
```

6.4 Model evaluation

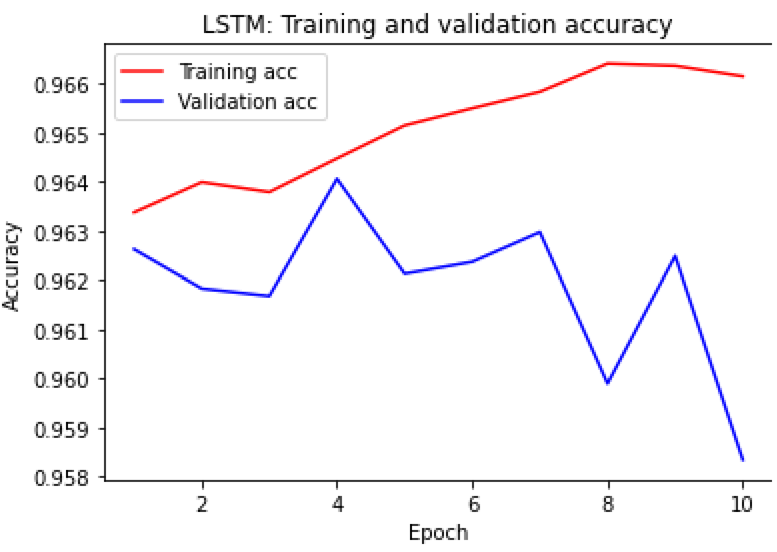
Now, lets' plot the model performances over time and look at their accuracies and losses across 10 epochs.

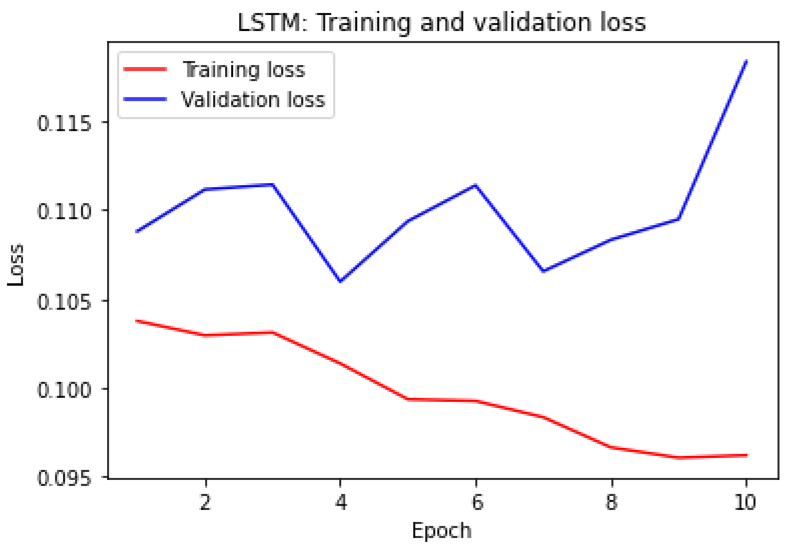
SimpleRNN model performance





LSTM model performance





Looking at the training accuracies in both models, the both SimpleRNN and LSTM appear to be overfitting the training set and the validation accuracies have high variance for the same reason. While the validation accuracy in SimpleRNN starts to increase, it starts to drop to lower values in the LSTM model.

Conclusion

In this project, I attempted to demonstrate how to set up a deep learning pipeline that predicts the sentiments of the tweets related to the 2020 US election. To do that, I first created my own dataset by scraping raw tweets via Twitter API and Tweepy package.

Over 440,000 tweets were streamed via Twitter API and stored into a CSV file. After wrangling and visualizing the data, a traditional clustering algorithm, k-means clustering in this case, was used to tag the tweets with two different labels, representing positive or negative sentiments. That is, the problem was converted into a supervised learning problem before training the deep learning models with the data. Then the dataset was split into training and test sets.

Later, the training set was used to train SimpleRNN and LSTM models respectively and were evaluated using the loss and accuracy curves from the model performances in each epoch. Overall, both models appear to be performing as they should and are likely to be overfitting the data according to what is seen from accuracy plots and as such I suggest the following recommendations for the next step.

Recommendations:

- Find another approach or different learning algorithm to label the dataset
- Try Amazon Mechanical Turk or Ground Truth to label the data set
- Try different RNN architectures
- Perform more advanced hyperparameter tuning of the RNN architectures
- Perform cross-validation

- Make the data multi-class problem

Skills Practiced During This Project/Tutorial:

1. How to efficiently collect data from Twitter via Tweepy and Twitter API
2. How to efficiently work with large dataset
3. How to build deep learning architectures, compile and fit in Keras
4. How to apply basic NLP concepts and techniques to a text data

Jupyter notebook with the code can be found here