

# Final Report

## Sentiment Analysis of Tweets – Polarity Classification

Aashish Bansal

aashish22bansal  
@gmail.com

Priyanka Agarwal

priyanka.agarwal248  
@gmail.com

Ameya Mande

ameya.mande  
@gmail.com

Jyotish Ranjan Mallik

jyotishranjanmallik  
@gmail.com

[Capstone Project for Indian Institute of Technology, Kanpur (IIT K)-  
Applied ML and Data Science Training and Internship Program]

Team Name: TechTitans

### ABSTRACT

In the advent of increasing use of social networking platforms and e-commerce sites, has led to an upsurge in demand for analytical technologies to review opinions, tweets, and other relevant data to facilitate organisations and entities in decision making processes. In this context, Sentiment Analysis, a data mining process, comes into the picture. In this project, we incorporate algorithmic techniques to conduct sentiment analysis on tweets for envision of polarity classification. Cleaning and pre-processing data is an important milestone in the bigger process of sentiment analysis, which here in this project we accomplished using NLTK library and performing operations such as stemming and lemmatization to further process and simplify the data in Python. The data for training and testing have been retrieved successfully from the Kaggle platform where the concerned competition is hosted. In this project, we have implemented the BiLSTM algorithm using Keras and are working on classifying the tweets as positive, negative and neutral.

*Keywords:* Sentiment Analysis · Supervised learning · Pre-processing · Mining · Long short-term memory (LSTM) · BiLSTM · Tweets · Natural language Processing (NLP) · NLTK · Classification · Convolutional neural network (CNN) · Random forest · Numpy · Pandas · Keras ·

# 1 INTRODUCTION

## 1.1 TWEETS AND THEIR ANALYSIS

The proliferation of user-generated content (UGC) on social media and e-commerce platforms has made user judgement tracking a strenuous yet necessary task. Twitter, being a giant microblogging social platform for networking, could be used to accumulate views about politics, trends, and products. Sentiment analysis is a data mining technique used to examine opinions, emotions, and attitude of people toward any subject through algorithmic techniques. This is conceptualized using digital data (text, video, audio, etc.) or psychological characteristics of humans. This procedure eases the process of opinion polarity classification without having to read each tweet manually. Hence, allowing the review process to be conducted at a large economic scale. The results could be exercised to provide an additional power for organisations to make better decisions about their planning, product and processes.

## 2 DATASETS

### 2.1 Training Set

The dataset for training the model was retrieved from Kaggle; where the competition was hosted as train\_samples.txt.

### 2.2 Testing Set

The dataset for training the model was retrieved from Kaggle; where the competition was hosted as test.txt.

# STEPS AND METHODOLOGY

## 3 CLEANING AND PREPROCESSING DATA

### 3.1 Creating array of labels

We take the sentiments of the people in the tweets and we create an array out of it so that it is easy for analysis and testing of the program after processing and because it is in an array format, it will be easy for us to check with the array index of the sentiment and the tweet.

```
1 # This Python 3 environment comes with many helpful analytics libraries installed
2 # It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
3 # For example, here's several helpful packages to load
4
5 import numpy as np # linear algebra
6 import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
7
8 # Input data files are available in the read-only "../input/" directory
9 # For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory
10
11 import os
12 for dirname, _, filenames in os.walk('/kaggle/input'):
13     for filename in filenames:
14         print(os.path.join(dirname, filename))
15
16 # You can write up to 5GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using
17 # You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

```
/kaggle/input/sentiment-analysis-of-tweets/test_samples.txt
/kaggle/input/sentiment-analysis-of-tweets/train.txt
```

Fig 3.1.1: Checking the files

```
1 #importing all the libraries
2
3 import sys , os , re , csv , codecs , numpy as np, pandas as pd
4 import matplotlib.pyplot as plt
5 %matplotlib inline
6 from keras.preprocessing.text import Tokenizer
7 from keras.preprocessing.sequence import pad_sequences
8 from keras.layers import Dense, Input, LSTM, Embedding, Dropout, Activation, BatchNormalization
9 from keras.layers import Bidirectional, GlobalMaxPool1D, GlobalAvgPool1D
10 from keras.models import Model
11 from keras import initializers, regularizers, constraints , optimizers, layers
12 from keras.utils import to_categorical
13
```

```
1 #getting the train data
2 train = pd.read_csv('/kaggle/input/sentiment-analysis-of-tweets/train.txt')
3 #loading the test data
4 test = pd.read_csv('/kaggle/input/sentiment-analysis-of-tweets/test_samples.txt')
5 #display first 5 rows of train
6 train.head()
7 #one hot encoding the labels
8 df = pd.concat([train, pd.get_dummies(train['sentiment'])], axis=1)
9 #df.head()
10 train_data = df['tweet_text']
11 #train_data.head()
```

Fig 3.1.2: Importing required dependencies and Loading the datasets

```
1 test_data = test['tweet_text']
2 test_data.head()
```

```
0 @jjvueellzz down in the Atlantic city, ventnor...
1 Musical awareness: Great Big Beautiful Tomorro...
2 On Radio786 100.4fm 7:10 Fri Oct 19 Labour ana...
3 Kapan sih lo ngebuktiin, jan ngomong doang Susa...
4 Excuse the connectivity of this live stream, f...
Name: tweet_text, dtype: object
```

Fig 3.1.3: Checking if the file has been read by printing some data

```
1 #creating the array of labels in serial with their respective texts
2 classes = ['neutral' , 'negative' , 'positive']
3 y = df[classes].values
4 y
```

```
array([[0, 0, 1],
       [0, 1, 0],
       [0, 1, 0],
       ...,
       [1, 0, 0],
       [0, 0, 1],
       [1, 0, 0]], dtype=uint8)
```

Fig 3.1.4: Creating an array for the labels

## 3.2 Checking for null values

This is used to check if any tweet has any null value or not.

```
1 #checking for null values in train and test data
2 train.isnull().any()
3 test.isnull().any()
```

```
: tweet_id      False
   tweet_text   False
   dtype: bool
```

Fig 3.2.1: Checking for null elements

### 3.3 Stemming and Lemmatization and Stopwords Removal

These are done using NLTK Tools. The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. Lemmatization is a more sophisticated as stemmers operate without knowledge of context. Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words. It observes position and Parts of speech of a word before stripping anything. The goal is to remove stop-words, also termed irrelevant words, that create noise when the model is learning.

For grammatical reasons, documents can contain **different forms of a word** such as *drive, drives, driving*. Also, sometimes we have **related words** with a similar meaning, such as *nation, national, nationality*. The result of this mapping applied on a text will be something like, for example, the boy's dogs are different sizes => the boy dog be differ size.

Stemming and lemmatization are special cases of **normalization**. However, they are different from each other as:

Stemming	Lemmatization
Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes.	Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma.

#### Lemmatization:

##### Part-of-Speech:

	JJ	NNS
1	Gathered	requirements

##### Lemmas:

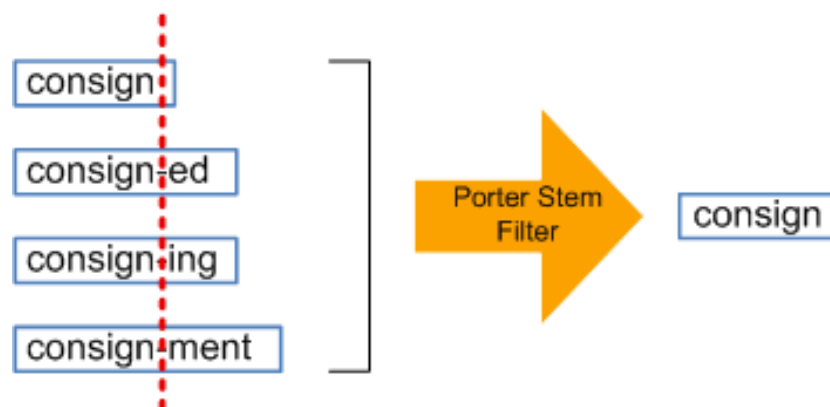
	gathered	requirement
1	Gathered	requirements

#### PICTURE REPRESENTING LEMMATIZATION

- In English (and most languages), words appear in different forms. Looks at these two sentences: 'I had a pony' and 'I had two ponies.'
  - Both sentences talk about the noun pony, but they are using different inflections. When working with text in a computer, it is helpful to know the base form of each word so that you know that both sentences are talking about the same concept.
  - Otherwise the strings "pony" and "ponies" look like two totally different words to a computer.

- In NLP, we call finding this process lemmatization – figuring out the most basic form or lemma of each word in the sentence.
- It is typically done by having a look-up table of the lemma forms of words based on their part of speech and possibly having some custom rules to handle words that you’ve never seen before.
- Lemmatization converts a word into its lemma (root form). Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words. It observes position and Parts of speech of a word before stripping anything. For example consider two lemma's listed below:
  - saw [verb] - Past tense of see
  - saw [noun] - Cutting instrument
- It normally aims to strip inflection from end of a word. For word “saw”, stemming might return just “s”, whereas lemmatization would attempt to return either “see” or “saw” depending on whether the use of the token was as a verb or a noun.
- Lemmatization is the algorithmic process of finding the lemma of a word depending on their meaning. Lemmatization usually refers to the morphological analysis of words, which aims to remove inflectional endings. It helps in returning the base or dictionary form of a word, which is known as the lemma. The NLTK Lemmatization method is based on WordNet's built-in morph function. Text pre-processing includes both stemming as well as lemmatization. Many people find the two terms confusing. Some treat these as same, but there is a difference between these both. Lemmatization is preferred over the former because of the below reason.

### Stemming:



PICTURE REPRESENTING STEMMING

- Stemming is a kind of normalisation for words. Normalization is a technique where a set of words in a sentence are converted into a sequence to shorten its lookup. The words which have the same meaning but have some variation according to the context or sentence are normalised. In another word, there is one root word, but there are many variations of the same words. For example, the root word is “eat” and with the help of Stemming, we can find the root word of any variations

- Stemming algorithms
  - Porter stemmer
  - Snowball stemmer
  - Lancaster stemmer

### Where to use stemming and where to use Lemmatization?

- It based on your requirement.
- If you are handling huge amount of text and you only want to normalize and analyze text not to visualize, then you may go with stemming.
- But if you want to visualize your normalized text then you should choose lemmatization as stem words may not necessarily the real world word.
- Also you can anti stemming to your stemmed word to get real world word but as per my experience it will take huge amount of time to execute anti stemming task.

```

1 #NLTK python library for preprocessing
2 import nltk
3 #nltk.download('wordnet')
4 #for tokenization
5 from nltk.tokenize import RegexpTokenizer
6 #for stemming
7 from nltk.stem import WordNetLemmatizer,PorterStemmer
8 #for removing stopwords
9 from nltk.corpus import stopwords
10 #importing regex library of python
11 import re
12 lemmatizer = WordNetLemmatizer()
13 stemmer = PorterStemmer()
14 #function for performing all preproccing steps at once
15 def preprocess(sentence):
16     sentence=str(sentence)
17     sentence = sentence.lower()
18     sentence=sentence.replace('{html}','')
19     cleanr = re.compile('<.*?>')
20     cleantext = re.sub(cleanr, '', sentence)
21     rem_url=re.sub(r'http\S+', '',cleantext)
22     rem_num = re.sub('[0-9]+', '', rem_url)
23     tokenizer = RegexpTokenizer(r'\w+')
24     tokens = tokenizer.tokenize(rem_num)
25     filtered_words = [w for w in tokens if not w in stopwords.words('english')]
26     stem_words=[stemmer.stem(w) for w in filtered_words]
27     lemma_words=[lemmatizer.lemmatize(w) for w in stem_words]
28     return " ".join(filtered_words)
29
30 #make a dataframe of preprocessed text
31 df['cleanText']=train_data.map(lambda s:preprocess(s))

```

Fig 3.3.1: Pre-processing the data

### 3.4 Sentence and Word Tokenization

Sentence tokenization or segmentation, is the problem that deals with dividing a string of written language into its component sentences. Word tokenization or segmentation dividing a string of written language into its component words

#### Word Tokenization

- Now that we've split out document into sentences, we can process them one at a time. Let's start with the first sentence from our document:

- “London is the capital and most populous city of England and the United Kingdom.”
- The next step in our pipeline is to break this sentence into separate words or tokens. This is called tokenization. This is the result:
  - “London”, “is”, “the”, “capital”, “and”, “most”, “populous”, “city”, “of”, “England”, “and”, “the”, “United”, “Kingdom”, “.”
- Tokenization is easy to do in English.
- We’ll just split apart words whenever there’s a space between them.
- And we’ll also treat punctuation marks as separate tokens since punctuation also has meaning.

```

1 #breaking the sentence into unique words/tokens
2 #expecting max tokens to be 20k
3 train_final = df['cleanText']
4 max_feat=40000
5 #tokenize sentence into list of words
6 tokenizer = Tokenizer(num_words=max_feat)#setting up tokenizer
7 #fitting the tokenizer on out data
8 tokenizer.fit_on_texts(list(train_final))
9

```

Fig 3.4.1: Sentence Tokenization

### 3.5 Fitting the Data

we define a function fit\_on\_texts and feed the data in the form of a list inorder to help it train the model quickly for analysis because the list index will help in checking the correct sentiment with the predicted sentiment

```

1 #Fitting the model
2 batch_size=64
3 epochs = 30
4 model.fit(x_train,y,batch_size=batch_size,epochs = epochs,validation_split=0.2)

```

Fig 3.5.1: Fitting the data into the Model

```

Epoch 1/30
269/269 [=====] - 164s 609ms/step - loss: 1.0977 - accuracy: 0.4971 - val_loss: 1.9855 - val_accuracy: 0.3988
Epoch 2/30
269/269 [=====] - 162s 602ms/step - loss: 0.7904 - accuracy: 0.6585 - val_loss: 8.8136 - val_accuracy: 0.4281
Epoch 3/30
269/269 [=====] - 162s 601ms/step - loss: 0.6312 - accuracy: 0.7466 - val_loss: 28.4012 - val_accuracy: 0.3988
Epoch 4/30
269/269 [=====] - 162s 602ms/step - loss: 0.5122 - accuracy: 0.8026 - val_loss: 32.2878 - val_accuracy: 0.4281
Epoch 5/30
269/269 [=====] - 162s 603ms/step - loss: 0.4010 - accuracy: 0.8530 - val_loss: 22.5790 - val_accuracy: 0.3990
Epoch 6/30
269/269 [=====] - 162s 602ms/step - loss: 0.3176 - accuracy: 0.8894 - val_loss: 36.6653 - val_accuracy: 0.1731
Epoch 7/30
269/269 [=====] - 162s 602ms/step - loss: 0.2551 - accuracy: 0.9094 - val_loss: 16.2503 - val_accuracy: 0.1731
Epoch 8/30
269/269 [=====] - 162s 602ms/step - loss: 0.2035 - accuracy: 0.9302 - val_loss: 9.0825 - val_accuracy: 0.4281
Epoch 9/30
269/269 [=====] - 162s 602ms/step - loss: 0.1675 - accuracy: 0.9442 - val_loss: 8.4932 - val_accuracy: 0.4298
Epoch 10/30
269/269 [=====] - 162s 601ms/step - loss: 0.1493 - accuracy: 0.9499 - val_loss: 7.6688 - val_accuracy: 0.4030
Epoch 11/30
269/269 [=====] - 162s 602ms/step - loss: 0.1296 - accuracy: 0.9580 - val_loss: 3.0278 - val_accuracy: 0.2786
Epoch 12/30
269/269 [=====] - 162s 600ms/step - loss: 0.1127 - accuracy: 0.9630 - val_loss: 3.5251 - val_accuracy: 0.3084
Epoch 13/30
269/269 [=====] - 162s 601ms/step - loss: 0.1118 - accuracy: 0.9636 - val_loss: 3.1608 - val_accuracy: 0.4642
Epoch 14/30
269/269 [=====] - 162s 603ms/step - loss: 0.0992 - accuracy: 0.9686 - val_loss: 1.9886 - val_accuracy: 0.5153
Epoch 15/30
269/269 [=====] - 162s 602ms/step - loss: 0.0879 - accuracy: 0.9723 - val_loss: 3.5870 - val_accuracy: 0.3450

```



Fig 3.5.2.1: Training the Model

```
Epoch 16/30
269/269 [=====] - 162s 601ms/step - loss: 0.0818 - accuracy: 0.9746 - val_loss: 7.3300 - val_accuracy: 0.4223
Epoch 17/30
269/269 [=====] - 162s 602ms/step - loss: 0.0752 - accuracy: 0.9773 - val_loss: 15.3390 - val_accuracy: 0.1731
Epoch 18/30
269/269 [=====] - 162s 601ms/step - loss: 0.0668 - accuracy: 0.9790 - val_loss: 2.5116 - val_accuracy: 0.5236
Epoch 19/30
269/269 [=====] - 162s 601ms/step - loss: 0.0665 - accuracy: 0.9790 - val_loss: 7.4534 - val_accuracy: 0.5022
Epoch 20/30
269/269 [=====] - 162s 601ms/step - loss: 0.0641 - accuracy: 0.9797 - val_loss: 5.5781 - val_accuracy: 0.4985
Epoch 21/30
269/269 [=====] - 162s 601ms/step - loss: 0.0575 - accuracy: 0.9826 - val_loss: 4.2691 - val_accuracy: 0.3487
Epoch 23/30
269/269 [=====] - 162s 601ms/step - loss: 0.0547 - accuracy: 0.9842 - val_loss: 3.4779 - val_accuracy: 0.4540
Epoch 24/30
269/269 [=====] - 162s 601ms/step - loss: 0.0531 - accuracy: 0.9843 - val_loss: 3.9039 - val_accuracy: 0.4642
Epoch 25/30
269/269 [=====] - 161s 600ms/step - loss: 0.0478 - accuracy: 0.9863 - val_loss: 3.6402 - val_accuracy: 0.4999
Epoch 26/30
269/269 [=====] - 162s 601ms/step - loss: 0.0530 - accuracy: 0.9841 - val_loss: 3.6919 - val_accuracy: 0.4854
Epoch 27/30
269/269 [=====] - 161s 599ms/step - loss: 0.0460 - accuracy: 0.9871 - val_loss: 2.6603 - val_accuracy: 0.5157
Epoch 28/30
269/269 [=====] - 161s 600ms/step - loss: 0.0465 - accuracy: 0.9870 - val_loss: 3.4747 - val_accuracy: 0.5073
Epoch 29/30
269/269 [=====] - 161s 600ms/step - loss: 0.0416 - accuracy: 0.9883 - val_loss: 4.9956 - val_accuracy: 0.5183
Epoch 30/30
269/269 [=====] - 161s 599ms/step - loss: 0.0426 - accuracy: 0.9877 - val_loss: 9.7257 - val_accuracy: 0.1938
```

Fig 3.5.2.2: Training the Model

### 3.6 Text to Numbers and Index Mappings

We know that machines cannot not understand words as such and hence we convert the sentences into a sequence of numbers by assigning each word a unique number and then stringing it all together.

Index mappings were refactored such that notes and NLP annotations reside in their own indexes, resolving the resource utilization issues. Notes and annotation are associated, so to speak, at runtime by matching on a shared-value, externally-derived document identifier used in the indices.

```
1 #converting text into sequence of numbers to feed in neural network
2 sequence_train = tokenizer.texts_to_sequences(train_final)
3 sequence_test = tokenizer2.texts_to_sequences(test_final)
4 # get the word to index mapping for input language
5 word2idx_inputs = tokenizer.word_index
6 print('Found %s unique input tokens.' % len(word2idx_inputs))
7
```

```
Found 34302 unique input tokens.
```

Fig 3.6.1: Converting text to numbers and mapping them

### 3.7 Embedding Matrix and Embedding Layer

On applying one-hot encoding to words, we end up with sparse vectors of high dimensionality. On large data sets, this causes performance issues. Additionally, one-hot encoding does not take into account the semantics of the words. Word embeddings address these two issues. Word embeddings are dense vectors with much lower dimensionality. Secondly, the semantic relationships between words are reflected in the distance and direction of the vectors.

Keras provides a convenient way to convert each word into a multi-dimensional vector. This can be done with the embedding layer. It will compute the word embeddings (or use pre-trained embeddings) and look up each word in a dictionary to find its vector representation. In our project we have trained word embeddings with 8 dimensions.



## What are Word Embeddings?

### Different types of Word Embeddings

1. Frequency-based Embedding
  - a. Count Vectors
  - b. TF-IDF
  - c. Co-Occurrence Matrix
2. Prediction-based Embedding
  - a. CBOW
  - b. Skip-Gram

Word Embeddings use case scenarios (what all can be done using word embeddings? E.g. similarity, odd one out etc.)

- Using pre-trained Word Vectors
- Training your own Word Vectors

```
1 #EMBEDDING MATRIX
2 # prepare embedding matrix of words for embedding layer
3 print('Filling pre-trained embeddings...')
4 num_words = min(MAX_NUM_WORDS, len(word2idx_inputs) + 1)
5 embedding_matrix = np.zeros((num_words, EMBEDDING_DIM))
6 for word, i in word2idx_inputs.items():
7     if(i < MAX_NUM_WORDS):
8         embedding_vector = word2vec.get(word)
9         if embedding_vector is not None:
10             # words not found in embedding index will be all zeros.
11             embedding_matrix[i] = embedding_vector
```

Filling pre-trained embeddings...

Fig 3.7.1: Embedding Matrix

```
1 # create embedding layer
2 embedding_layer = Embedding(
3     num_words,
4     EMBEDDING_DIM,
5     weights=[embedding_matrix],
6     input_length=max_len,
7     trainable=True
8 )
```

Fig 3.7.2: Embedding Layer

## 4 MODEL

After considering and trying out various models, we chose to work with densely connected layer and BiLSTM algorithm using Keras for predictions.

```
1 #densely connected layer
2 dense1 = Dense(128,activation='elu')(drop_layer)
3 batchnorm2 = BatchNormalization()(dense1)
4 dense2 = Dense(128,activation='elu')(batchnorm2)
5 batchnorm3 = BatchNormalization()(dense2)
6 dense3 = Dense(128,activation='elu')(batchnorm3)
```

```
1 #adding another dropout layer
2 drop_layer2 = Dropout(0.55)(dense3)
```

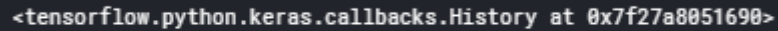
```
1 #adding the output dense layer with sigmoid activation to get result
2 #3 classes as output
3 output_dense = Dense(3,activation='softmax')(drop_layer2)
```

```
1 #connecting the inputs and outputs to create a model and compiling the model
2 from keras.optimizers import Adagrad,Adam,RMSprop
3 model = Model(inputs=input , outputs = output_dense)
4 model.compile(loss = 'categorical_crossentropy',
5               optimizer = RMSprop(lr=0.001),
6               metrics = ['accuracy'])
7 model.summary()
```

Fig 4.1: Defining the Model

Model: "model"		
Layer (type)	Output Shape	Param #
-----		
input_1 (InputLayer)	[(None, 1000)]	0
embedding (Embedding)	(None, 1000, 300)	10290900
bidirectional (Bidirectional)	(None, 1000, 512)	1140736
bidirectional_1 (Bidirection	(None, 1000, 512)	1574912
batch_normalization (BatchNo	(None, 1000, 512)	2048
global_average_pooling1d (G1	(None, 512)	0
dropout (Dropout)	(None, 512)	0
dense (Dense)	(None, 128)	65664
batch_normalization_1 (Batch	(None, 128)	512
dense_1 (Dense)	(None, 128)	16512
batch_normalization_2 (Batch	(None, 128)	512
dense_2 (Dense)	(None, 128)	16512
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 3)	387
-----		
Total params: 13,108,695		
Trainable params: 13,107,159		
Non-trainable params: 1,536		
-----		

Fig 4.2: Model Definition Output after Defining



```
<tensorflow.python.keras.callbacks.History at 0x7f27a8051690>
```

Fig 4.3: Output signifying the Trained Model

## RESULT AND CONCLUSION

After incorporating extensive steps in the process of cleaning and preprocessing data, and selecting an effective model, training it, we have successfully trained the model for the purpose of sentiment analysis of tweets.

We carried out multiple trials with various algorithms. We fixated upon using BiLSTM for training the model owing to its higher efficiency and bidirectional nature, especially in text application. The fact that BiLSTM runs backwards and Preserves information in any point of time, that is, past and future, due to its two hidden states. That is why BiLSTM is most effective at understanding contexts of the text in NLP applications.

After an exhausting training of the model, the final validation accuracy while carrying out the process of sentiment analysis was generated as 0.5053 and the tweets were classified as neutral, positive and negative depending on their respective algorithmically calculated scores.