Exploring Deep Learning Models for Sentiment Analysis on Tesla News

MSc Research Project
Data Analytics

Rajat Patil
Student ID: x22162259

School of Computing
National College of Ireland

Supervisor: Shubham Subhnil

National College of IrelandProject Submission Sheet School of Computing

Student Name:	Rajat Patil
Student ID:	x22162259
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Shubham Subhnil
Submission Due Date:	14/12/2023
Project Title:	Exploring Deep Learning Models For Sentimental Analysis on Tesla News
Word Count:	1000
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at therear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	13th December 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator office must be placedinto the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Exploring Deep Learning Models For Sentimental Analysis on Tesla News

Rajat Patil
x22162259

1 Introduction

This Configuration Manual lists together all prerequisites needed to duplicate the studies and its effects on a specific setting. A glimpse of the source for Data Scarping & sentiment analysis of the news and after that Lexical tokenization is done after that LEBert is implemented for all three types of n-gram and all the algorithms are created, and Evaluations is also supplied, together with the necessary hardware components as well as Software applications. The report is organized as follows, with details relating environment configuration provided in Section 2.
Information about data scraping is detailed in Section 3. Sentiment Analysis is done in Section 4. Lexical Tokenisation is included in Section 5. In section 6, the LeBert Algorithm is described. Details well about models that were created and tested are provided in Section 7. How the results are calculated and shown is described in Section 8.

2 System Requirements

The specific needs for hardware as well as software to put the research into use are detailed in this section.

2.1 Hardware Requirements

The necessary hardware specs are shown in Figure 1 below. MacOs M1 Chip, macOS 10.15.x (Catalilna) operating system, 8GB RAM, 256GB Storage, 24'' Display.

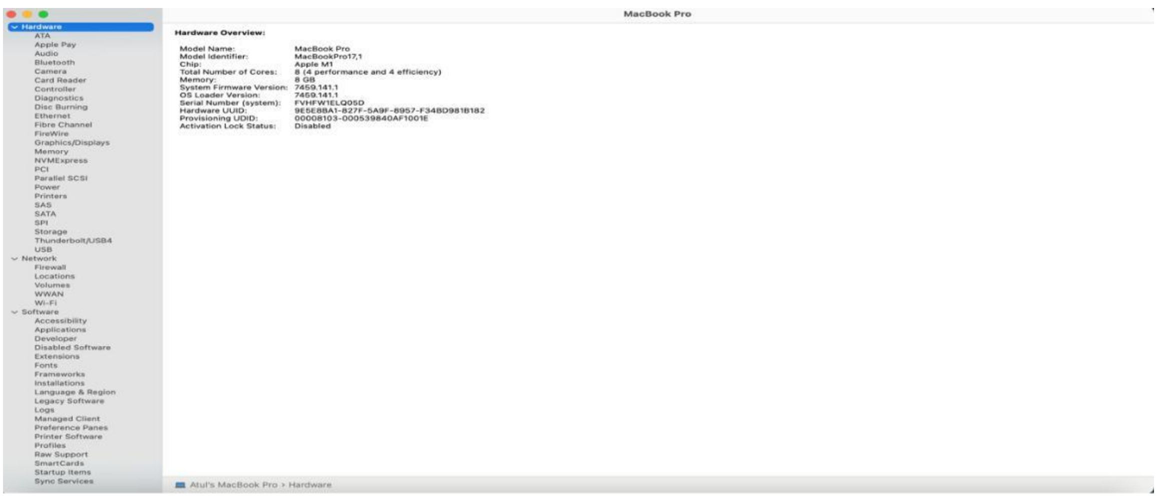

Figure 1: Hardware Requirements

2.2 Software Requirements

- Anaconda 3 (Version 4.8.0)
- Jupyter Notebook (Version 6.0.3)
- Python (Version 3.7.6)

2.3 Code Execution

The code can be run in jupyter notebook. The jupyter notebook comes with Anaconda 3, run the jupyter notebook from startup. This will open jupyter notebook in web browser. The web browser will show the folder structure of the system, move to the folder where the code file is located. Open the code file from the folder and to run the code, go to Kernel menu and Run all cells.

3 Data Scraping

The dataset is scraped from Google.com using beautiful soup.

Figure 2 includes a list of every Python library necessary to complete the project.

```
from bs4 import BeautifulSoup
import json
import requests
import re
import pandas as pd
import numpy as np
import nltk
nltk.download('vader_lexicon')
nltk.download('punkt')
nltk.download('treebank')
from nltk.sentiment.vader import SentimentIntensityAnalyzer
import matplotlib.pyplot as plt
from tqdm import tqdm
import warnings
# Suppress FutureWarning messages
warnings.simplefilter(action='ignore', category=FutureWarning)
from sklearn.model_selection import train_test_split
from nltk.tokenize import word_tokenize
from nltk.tokenize import RegexpTokenizer
from collections import Counter
from nltk.util import ngrams
from tensorflow.keras import Sequential
from tensorflow.keras import layers
from transformers import BertTokenizer
from sklearn.metrics import accuracy_score
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tensorflow as tf
from tensorflow.keras.layers import Layer
from tensorflow.keras import backend as K
from keras.layers import Dense, Flatten, Input, Dropout, Bidirectional, LSTM
from keras.models import Sequential
```

Figure 2: Necessary Python libraries

The Figure 3 represents the block of code to scrape the news from google of Tesla and creating a pandas dataframe from the same.

```
def getNewsData(ticker):
    headers = {"User-Agent":
        "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/101.0.4951.54 Safari/537.36"
    }
    news_results = []

    response = requests.get(f"https://www.google.com/search?q={ticker}&gl=ind&tbm=nws&tbs=cdr:1,cd_min:1/1/2023,cd_max:8/31/2023
    soup = BeautifulSoup(response.content, "html.parser")
    for el in soup.select("div.SoaBEf"):
        news_results.append([el.find("a")["href"], el.select_one("div.MBeuO").get_text(),
                    el.select_one(".GI74Re").get_text(), el.select_one(".LfVVr").get_text(),
                    el.select_one(".NUnG9d span").get_text()])


    return news_results
```

```
news = getNewsData('Tesla')
news_results=pd.DataFrame(news)
news_results
```

	0	1	2	3	4
0	https://www.livemint.com/technology/tech-news/...	Elon Musk's Tesla looks to set up India factor...	Tesla is reportedly in talks with the Indian g...	13 Jul 2023	Mint
1	https://www.reuters.com/business/autos-transpo...	Tesla looking to make about half million EVs a...	The starting price for the vehicles will be 2 ...	13 Jul 2023	Reuters
2	https://www.businesstoday.in/technology/news/s...	Tesla in talks to set up factory in India for ...	In addition to engaging with the government, T...	13 Jul 2023	Business Today
3	https://www.reuters.com/breakingviews/tesla-pu...	Breakingviews - Tesla pushes limits of India's...	Back in 2021, the company won approval to sell...	17 May 2023	Reuters
4	https://www.livemint.com/news/india/elon-musk-...	Tesla stops short of committing to India plant...	Discussions between Tesla and India reached a ...	19 May 2023	Mint

Figure 3: News scraping and data generation

As seen in Figure 4, the column names are assigned to the data and saved into a csv file.

```
try:
    news_results.columns = ["Link", "Title", "Snippet", "Date",  "Source"]
    news_results.to_csv("Nifty500News.csv")
except:
    news_results= pd.read_csv("Nifty500News.csv")  #Loading previous saved data in case google request times out

news_results
```

	Link	Title	Snippet	Date	Source
0	https://www.livemint.com/technology/tech-news/...	Elon Musk's Tesla looks to set up India factor...	Tesla is reportedly in talks with the Indian g...	13 Jul 2023	Mint
1	https://www.reuters.com/business/autos-transpo...	Tesla looking to make about half million EVs a...	The starting price for the vehicles will be 2 ...	13 Jul 2023	Reuters
2	https://www.businesstoday.in/technology/news/s...	Tesla in talks to set up factory in India for ...	In addition to engaging with the government, T...	13 Jul 2023	Business Today
3	https://www.reuters.com/breakingviews/tesla-pu...	Breakingviews - Tesla pushes limits of India's...	Back in 2021, the company won approval to sell...	17 May 2023	Reuters
4	https://www.livemint.com/news/india/elon-musk-...	Tesla stops short of committing to India plant...	Discussions between Tesla and India reached a ...	19 May 2023	Mint

Figure 4: Gathered data

4 Sentiment Analysis

In figure 5, the code to initialize SentimentIntensityAnalyzer and generate polatily score compound for the news title.

```
vader = SentimentIntensityAnalyzer()
```

```
sent_analyse = lambda title: vader.polarity_scores(title)['compound']
news_results['Compound'] = news_results['Title'].apply(sent_analyse)
news_results
```

	Link	Title	Snippet	Date	Source	Compound
0	https://www.livemint.com/technology/tech-news/...	Elon Musk's Tesla looks to set up India factor...	Tesla is reportedly in talks with the Indian g...	13 Jul 2023	Mint	0.4019
1	https://www.reuters.com/business/autos-transpo...	Tesla looking to make about half million EVs a...	The starting price for the vehicles will be 2 ...	13 Jul 2023	Reuters	0.0000
2	https://www.businesstoday.in/technology/news/s...	Tesla in talks to set up factory in India for ...	In addition to engaging with the government, T...	13 Jul 2023	Business Today	0.0000
3	https://www.reuters.com/breakingviews/tesla-pu...	Breakingviews - Tesla pushes limits of India's...	Back in 2021, the company won approval to sell...	17 May 2023	Reuters	0.0000
4	https://www.livemint.com/news/india/elon-musk-...	Tesla stops short of committing to India plant...	Discussions between Tesla and India reached a ...	19 May 2023	Mint	-0.0772

Figure 5: Class count

The Figure 6, illustrate the plot for the compound score for 00 new item.

```
plt.figure(figsize=(10,8))
plt.plot(news_results['Compound'], label='Sentiment Score')
```

```
[<matplotlib.lines.Line2D at 0x153717ceb10>]
```

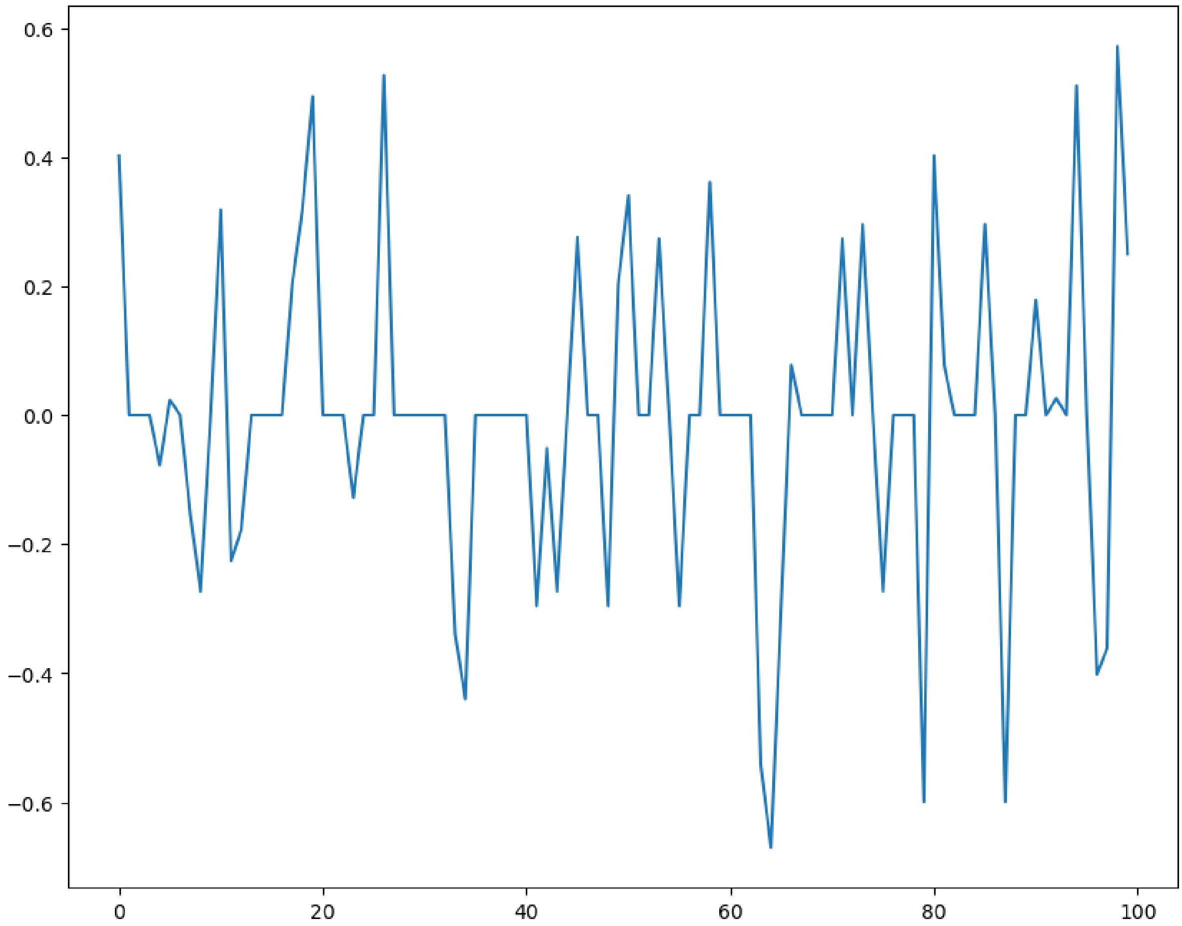

Figure 6: Compound score

Figure 7 includes a criteria to generate positive and negative sentiment based on compound score. Also, shows the plot for value counts on sentiment.

```
s=[]
for compound in news_results['Compound']:
        if compound > 0:
            s.append(1)
        else:
             s.append(0)
news_results['Sentiment'] = s
```

```
news_results['Sentiment'].value_counts()
news_results['Sentiment'].value_counts().plot(kind='bar')
```

```
<Axes: >
```

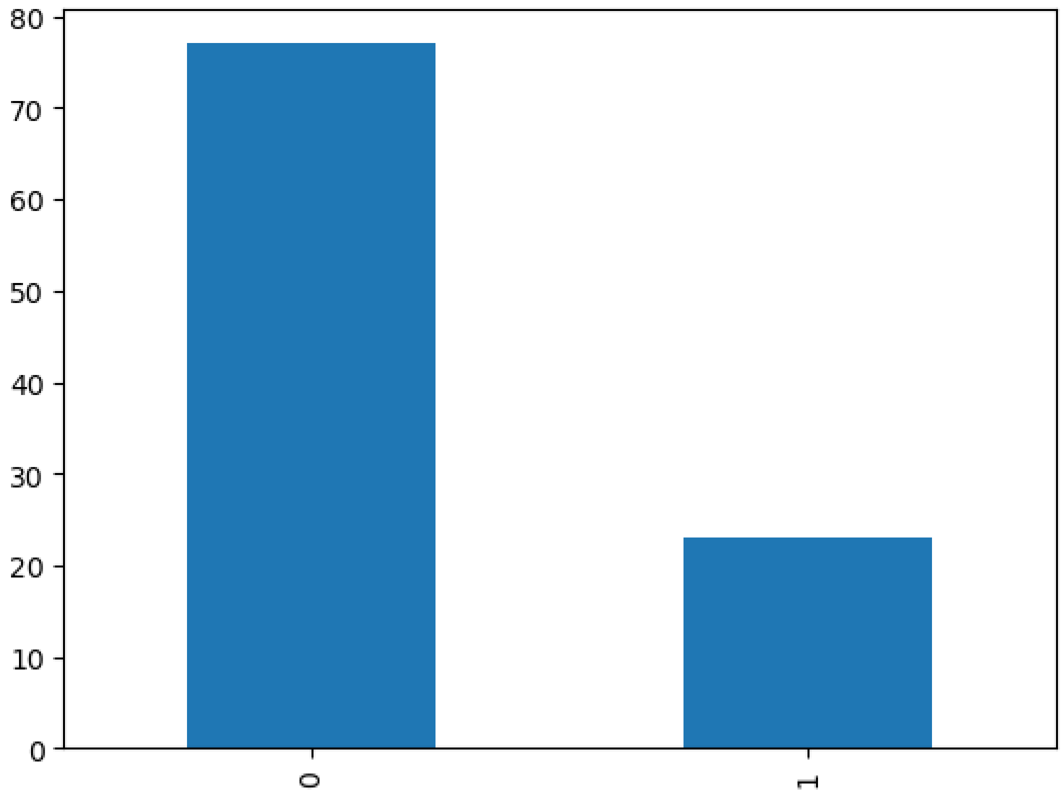

Figure 7: Sentiment Score

5 Lexical Tokenisation

Figures 8 show the code to read the Treebank tagged sentences.

```
# reading the Treebank tagged sentences
treebank = list(nltk.corpus.treebank.tagged_sents())
```

```
print(treebank[:3])
```

```
[[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'),
('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.',
'NNP'), ('29', 'CD'), ('.', '.')], [('Mr.', 'NNP'), ('Vinken', 'NNP'), ('is', 'VBZ'), ('chairman', 'NN'), ('of', 'IN'), ('Elsev
ier', 'NNP'), ('N.V.', 'NNP'), (',', ','), ('the', 'DT'), ('Dutch', 'NNP'), ('publishing', 'VBG'), ('group', 'NN'), ('.',
'.')], [('Rudolph', 'NNP'), ('Agnew', 'NNP'), (',', ','), ('55', 'CD'), ('years', 'NNS'), ('old', 'JJ'), ('and', 'CC'), ('forme
r', 'JJ'), ('chairman', 'NN'), ('of', 'IN'), ('Consolidated', 'NNP'), ('Gold', 'NNP'), ('Fields', 'NNP'), ('PLC', 'NNP'), (',',
','), ('was', 'VBD'), ('named', 'VBN'), ('*-1', '-NONE-'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), ('of', 'I
N'), ('this', 'DT'), ('British', 'JJ'), ('industrial', 'JJ'), ('conglomerate', 'NN'), ('.', '.')]]
```

Figure 8: Treebank

The Figure 9, illustrate the code to converting the list of sentences to a list of (word, pos tag) tuples.

```
# converting the list of sents to a list of (word, pos tag) tuples
tagged_words = [tup for sent in treebank for tup in sent]
print(len(tagged_words))
tagged_words[:10]
```

```
100676

[('Pierre', 'NNP'),
 ('Vinken', 'NNP'),
 (',', ','),
 ('61', 'CD'),
 ('years', 'NNS'),
 ('old', 'JJ'),
 (',', ','),
 ('will', 'MD'),
 ('join', 'VB'),
 ('the', 'DT')]
```

Figure 9: Treebank to tuple

The Figure 10, illustrate the parts od speech analysis.

```
tags = [pair[1] for pair in tagged_words]

# create a list of JJ tags
jj_tags = [t for t in tags if t == 'JJ']

# create a list of (JJ, NN) tags
jj_nn_tags = [(t, tags[index+1]) for index, t in enumerate(tags)
              if t=='JJ' and tags[index+1]=='NN']

print(len(jj_tags))
print(len(jj_nn_tags))
print(len(jj_nn_tags) / len(jj_tags))
```

```
5834
2611
0.4475488515598217
```

```
dt_tags = [t for t in tags if t == 'DT']
dt_nn_tags = [(t, tags[index+1]) for index, t in enumerate(tags)
              if t=='DT' and tags[index+1]=='NN']

print(len(dt_tags))
print(len(dt_nn_tags))
print(len(dt_nn_tags) / len(dt_tags))
```

```
8165
3844
0.470789957134109
```

```
md_tags = [t for t in tags if t == 'MD']
md_vb_tags = [(t, tags[index+1]) for index, t in enumerate(tags)
              if t=='MD' and tags[index+1]=='VB']

print(len(md_tags))
print(len(md_vb_tags))
print(len(md_vb_tags) / len(md_tags))
```

```
927
756
0.8155339805825242
```

Figure 10: POS-Tagging

The Figure 11, illustrate the section to split Treebank into training and test set to analyse the impact of taggers.

```
# splitting into train and test
train_set, test_set = train_test_split(treebank, test_size=0.3)

print(len(train_set))
print(len(test_set))
print(train_set[:2])
```

```
2739
1175
[[('The', 'DT'), ('city', 'NN'), ("'s", 'POS'), ('Campaign', 'NNP'), ('Finance', 'NNP'), ('Board', 'NNP'), ('has', 'VBZ'), ('re
fused', 'VBN'), ('*-1', '-NONE-'), ('to', 'TO'), ('pay', 'VB'), ('Mr.', 'NNP'), ('Dinkins', 'NNP'), ('$', '$'), ('95,142', 'C
D'), ('*U*', '-NONE-'), ('in', 'IN'), ('matching', 'JJ'), ('funds', 'NNS'), ('because', 'IN'), ('his', 'PRP$'), ('campaign', 'N
N'), ('records', 'NNS'), ('are', 'VBP'), ('incomplete', 'JJ'), ('.', '.')], [('With', 'IN'), ('membership', 'NN'), ('of', 'I
N'), ('the', 'DT'), ('Church', 'NNP'), ('of', 'IN'), ('England', 'NNP'), ('steadily', 'RB'), ('dwindling', 'VBG'), (',', ','),
('strong-willed', 'JJ'), ('vicars', 'NNS'), ('are', 'VBP'), ('pressing', 'VBG'), ('equally', 'RB'), ('strong-willed', 'JJ'),
('and', 'CC'), ('often', 'RB'), ('non-religious', 'JJ'), ('ringers', 'NNS'), ('to', 'TO'), ('attend', 'VB'), ('services', 'NN
S'), ('.', '.')]]
```

Figure 11: Train test split

The Figure 12, illustrate the unigram tagger and its performance.

```
# Lexicon (or unigram tagger)
unigram_tagger = nltk.UnigramTagger(train_set)
unigram_tagger.evaluate(test_set)
```

```
C:\Users\SHILPA\AppData\Local\Temp\ipykernel_11972\2670269465.py:3: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  unigram_tagger.evaluate(test_set)
```

```
0.8709104659191793
```

Figure 12: Unigram Tagger

The Figure 13, illustrate the Regular expression tagger and cobining regex and ingram tagger.

```
regexp_tagger = nltk.RegexpTagger(patterns)
regexp_tagger.evaluate(test_set)
```

```
C:\Users\SHILPA\AppData\Local\Temp\ipykernel_11972\3736896624.py:2: Depre
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  regexp_tagger.evaluate(test_set)
```

```
0.21934698977410977
```

```
# rule based tagger
rule_based_tagger = nltk.RegexpTagger(patterns)

# lexicon backed up by the rule-based tagger
lexicon_tagger = nltk.UnigramTagger(train_set, backoff=rule_based_tagger)

lexicon_tagger.evaluate(test_set)
```

```
C:\Users\SHILPA\AppData\Local\Temp\ipykernel_11972\2535250660.py:7: Depre
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  lexicon_tagger.evaluate(test_set)
```

```
0.9039226646499852
```

Figure 13: Regex and Unigram Tagger

The Figure 14, illustrate the regex tokeniser and vocab building.

```
tokenizer = RegexpTokenizer(r'\w+')
words_descriptions = news_results['Title'].apply(tokenizer.tokenize)
words_descriptions.head()
```

```
0    [Elon, Musk, s, Tesla, looks, to, set, up, Ind...
1    [Tesla, looking, to, make, about, half, millio...
2    [Tesla, in, talks, to, set, up, factory, in, I...
3    [Breakingviews, Tesla, pushes, limits, of, Ind...
4    [Tesla, stops, short, of, committing, to, Indi...
Name: Title, dtype: object
```

```
all_words = [word for tokens in words_descriptions for word in tokens]
news_results['description_lengths']= [len(tokens) for tokens in words_descriptions]
VOCAB = sorted(list(set(all_words)))
print("%s words total, with a vocabulary size of %s" % (len(all_words), len(VOCAB)))
```

```
1148 words total, with a vocabulary size of 510
```

Figure 14: Regex Tokenizer

The Figure 15, illustrate the counter for most common words in the news data.

```
count_all_words = Counter(all_words)
count_all_words.most_common(100)
```

```
[('Tesla', 102),
 ('in', 37),
 ('s', 35),
 ('to', 35),
 ('India', 24),
 ('Musk', 22),
 ('Elon', 16),
 ('Mint', 14),
 ('Model', 13),
 ('of', 12),
 ('for', 11),
 ('EV', 11),
 ('on', 11),
 ('and', 11),
 ('the', 11),
 ('after', 10),
 ('China', 9),
 ('up', 8),
 ('car', 8),
 ('as', 8),
 ('factory', 7),
```

Figure 15: Common word counter

The Figure 16, illustrate the unigram generation.

```
# generate unigrams
unigrams  = (news_results['Title'].str.lower()
                .str.replace(r'[^a-z\s]', '')
                .str.split(expand=True)
                .stack())

unigrams
```

```
0   0         elon
    1        musks
    2        tesla
    3        looks
    4           to
           ...
99  3        trial
    4         over
    5    autopilot
    6          car
    7        crash
Length: 1069, dtype: object
```

Figure 16: Generating Unigram

The Figure 17, illustrate the bigram and trigram generation.

```
# generate bigrams by concatenating unigram columns
bigrams = unigrams + ' ' + unigrams.shift(-1)
bigrams
```

```
0   0        elon musks
    1       musks tesla
    2       tesla looks
    3          looks to
    4            to set
                ...
99  3        trial over
    4    over autopilot
    5     autopilot car
    6         car crash
    7               NaN
Length: 1069, dtype: object
```

```
# generate trigrams by concatenating unigram and bigram columns
trigrams = bigrams + ' ' + unigrams.shift(-2)
trigrams
```

```
0   0        elon musks tesla
    1       musks tesla looks
    2          tesla looks to
    3            looks to set
    4               to set up
                   ...
99  3    trial over autopilot
    4      over autopilot car
    5     autopilot car crash
    6                     NaN
    7                     NaN
Length: 1069, dtype: object
```

Figure 17: Bigram and Trigram

The Figure 18, illustrate creating a function to generate N-Grams.

```
# Creating a function to generate N-Grams
def generate_ngrams(n):
    ngram = []
    for sentence in news_results['Title']:
        s = sentence.lower()
        s = re.sub(r'[^a-zA-Z0-9\s]', ' ', s)
        tokens = [token for token in s.split(" ") if token != ""]
        output = list(ngrams(tokens, n))
        ngram.append(output)
    return ngram
```

```
unigram = generate_ngrams(1)
news_results['unigram'] = unigram
unigram
```

```
 ('support',),
 ('electric',),
 ('cars',),
 ('\nreport',),
 ('mint',)],
[('tesla',),
 ('looking',),
 ('to',),
 ('make',),
 ('about',),
 ('half',),
 ('million',),
 ('evs',),
```

Figure 18: n-gram generator

The Figure 19, illustrate creating bigram and trigram using N-Grams.

```
bigram = generate_ngrams(2)
news_results['bigram'] = bigram
bigram
```

```
[[('elon', 'musk'),
  ('musk', 's'),
  ('s', 'tesla'),
  ('tesla', 'looks'),
  ('looks', 'to'),
  ('to', 'set'),
  ('set', 'up'),
  ('up', 'india'),
  ('india', 'factory'),
  ('factory', 'to'),
  ('to', 'support'),
  ('support', 'electric'),
  ('electric', 'cars'),
  ('cars', '\nreport'),
  ('\nreport', 'mint')],
 [('tesla', 'looking'),
  ('looking', 'to'),
  ('to', 'make'),
  ('make', 'about'),
```

```
trigram = generate_ngrams(3)
news_results['trigram'] = trigram
trigram
```

```
[[('elon', 'musk', 's'),
  ('musk', 's', 'tesla'),
  ('s', 'tesla', 'looks'),
  ('tesla', 'looks', 'to'),
  ('looks', 'to', 'set'),
  ('to', 'set', 'up'),
  ('set', 'up', 'india'),
  ('up', 'india', 'factory'),
  ('india', 'factory', 'to'),
  ('factory', 'to', 'support'),
  ('to', 'support', 'electric'),
  ('support', 'electric', 'cars'),
  ('electric', 'cars', '\nreport'),
  ('cars', '\nreport', 'mint')],
 [('tesla', 'looking', 'to'),
  ('looking', 'to', 'make'),
  ('to', 'make', 'about'),
  ('make', 'about', 'half'),
  ('about', 'half', 'million'),
```

Figure 19: n-gram generator

The Figure 20, illustrate creating training and test sets of actual data.

```
# Spliting data in test and train data set(80:20)
x= news_results.drop('Sentiment', axis=1)
y = news_results["Sentiment"].values
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.20)
```

```
modelScore = pd.DataFrame()
uniScore = pd.DataFrame()
biScore = pd.DataFrame()
triScore = pd.DataFrame()
```

Figure 20: Training and testing data generator

6 LeBert Algorithm

Figures 21 show the code to create LeBert Unigram.

LeBert Unigram

```
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased')
```

```
X_train_unigram = [tokenizer.convert_tokens_to_ids(com) for com in X_train['unigram']]
X_train_unigram = pad_sequences(X_train_unigram, maxlen=31, truncating='post', padding='post')
X_train_unigram.shape
```

```
(80, 31)
```

```
X_test_unigram = [tokenizer.convert_tokens_to_ids(com) for com in X_test['unigram']]
X_test_unigram = pad_sequences(X_test_unigram, maxlen=31, truncating='post', padding='post')
X_test_unigram.shape
```

```
(20, 31)
```

Figure 21: LeBert

Figures 22 show the code to create LeBert Bigram.

LeBert Bigram

```
X_train_bigram = [tokenizer.convert_tokens_to_ids(com) for com in X_train['bigram']]
X_train_bigram = pad_sequences(X_train_bigram, maxlen=31, truncating='post', padding='post')
X_train_bigram.shape
```

```
(80, 31)
```

```
X_test_bigram = [tokenizer.convert_tokens_to_ids(com) for com in X_test['bigram']]
X_test_bigram = pad_sequences(X_test_bigram, maxlen=31, truncating='post', padding='post')
X_test_bigram.shape
```

```
(20, 31)
```

Figure 22: LeBert

The Figure 23, illustrate the code to create LeBert Trigram

.

LeBert Trigram

```
X_train_trigram = [tokenizer.convert_tokens_to_ids(com) for com in X_train['trigram']]
X_train_trigram = pad_sequences(X_train_trigram, maxlen=31, truncating='post', padding='post')
X_train_trigram.shape
```

```
(80, 31)
```

```
X_test_trigram = [tokenizer.convert_tokens_to_ids(com) for com in X_test['trigram']]
X_test_trigram = pad_sequences(X_test_trigram, maxlen=31, truncating='post', padding='post')
X_test_trigram.shape
```

```
(20, 31)
```

Figure 23: LeBert

7 Machine Learning Models

7.1 RNN Unigram

```
rnn = Sequential()
rnn.add(Input(shape=(X_train_unigram.shape[1],1)))
rnn.add(Dense(64, activation='sigmoid'))
rnn.add(Dropout(0.2))
rnn.add(Dense(32, activation='relu'))
rnn.add(Dropout(0.2))
rnn.add(Dense(1, activation='relu'))
print(rnn.output_shape)
print(rnn.compute_output_signature)
rnn.compile(loss="binary_crossentropy", metrics=["accuracy"])
rnn.summary()
```

```
(None, 31, 1)
<bound method Layer.compute_output_signature of <keras.src.engine.sequential.Sequential object at 0x0000015370415290>>
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_21 (Dense)            (None, 31, 64)            128

 dropout_12 (Dropout)        (None, 31, 64)            0

 dense_22 (Dense)            (None, 31, 32)            2080

 dropout_13 (Dropout)        (None, 31, 32)            0

 dense_23 (Dense)            (None, 31, 1)             33

=================================================================
Total params: 2241 (8.75 KB)
Trainable params: 2241 (8.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= rnn.fit(X_train_unigram, y_train, validation_data=(X_test_unigram, y_test), epochs=15)
```

```
Epoch 1/15
3/3 [==============================] - 23s 3s/step - loss: 2.1017 - accuracy: 0.6465 - val_loss: 0.4707 - val_accuracy: 0.8500
Epoch 2/15
3/3 [==============================] - 1s 265ms/step - loss: 1.5607 - accuracy: 0.6135 - val_loss: 0.4569 - val_accuracy: 0.850
0
Epoch 3/15
3/3 [==============================] - 0s 221ms/step - loss: 1.5231 - accuracy: 0.6499 - val_loss: 0.5093 - val_accuracy: 0.850
0
Epoch 4/15
3/3 [==============================] - 1s 322ms/step - loss: 1.4267 - accuracy: 0.6213 - val_loss: 0.4939 - val_accuracy: 0.850
0
Epoch 5/15
3/3 [==============================] - 1s 346ms/step - loss: 1.3906 - accuracy: 0.6106 - val_loss: 0.5044 - val_accuracy: 0.850
0
```

Figure 24: Implementation of RNN

7.2 LSTM Unigram

```
lstm = Sequential()
lstm.add(Input(shape=(X_train_unigram.shape[1],1)))
lstm.add(LSTM(64, activation='relu'))
lstm.add(Dense(16, activation='relu'))
lstm.add(Dropout(0.3))
lstm.add(Dense(1, activation='tanh'))
print(lstm.output_shape)
print(lstm.compute_output_signature)
lstm.compile(optimizer='rmsprop', loss="binary_crossentropy", metrics=["accuracy"])
lstm.summary()
```

```
(None, 1)
<bound method Layer.compute_output_signature of <keras.src.engine.sequential.Sequential object at 0x000001536C802A90>>
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_3 (LSTM)               (None, 64)                16896

 dense_24 (Dense)            (None, 16)                1040

 dropout_14 (Dropout)        (None, 16)                0

 dense_25 (Dense)            (None, 1)                 17

=================================================================
Total params: 17953 (70.13 KB)
Trainable params: 17953 (70.13 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= lstm.fit(X_train_unigram, y_train, validation_data=(X_test_unigram, y_test), epochs=10)
```

```
Epoch 1/10
3/3 [==============================] - 42s 3s/step - loss: 4.5769 - accuracy: 0.6625 - val_loss: 2.3137 - val_accuracy: 0.8500
Epoch 2/10
3/3 [==============================] - 1s 342ms/step - loss: 3.8963 - accuracy: 0.7375 - val_loss: 2.3137 - val_accuracy: 0.850
0
Epoch 3/10
3/3 [==============================] - 1s 314ms/step - loss: 3.6590 - accuracy: 0.7625 - val_loss: 2.8631 - val_accuracy: 0.800
0
Epoch 4/10
3/3 [==============================] - 1s 394ms/step - loss: 4.7599 - accuracy: 0.6875 - val_loss: 2.3137 - val_accuracy: 0.850
0
Epoch 5/10
3/3 [==============================] - 1s 329ms/step - loss: 4.0587 - accuracy: 0.7250 - val_loss: 2.3137 - val_accuracy: 0.850
0
```

Figure 25: Implementation of LSTM

7.3 CNN Unigram

```
X_train_unigram = np.reshape(X_train_unigram, (X_train_unigram.shape[0], X_train_unigram.shape[1], 1))
X_train_unigram.shape
```

```
(80, 31, 1)
```

```
X_test_unigram = np.reshape(X_test_unigram, (X_test_unigram.shape[0], X_test_unigram.shape[1], 1))
X_test_unigram.shape
```

```
(20, 31, 1)
```

```
model = Sequential()
model.add(layers.Conv1D(64, 2, activation="relu", padding="same", name="convLayer", input_shape=(X_train_unigram.shape[1],1)))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation ='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(1, activation ='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 convLayer (Conv1D)          (None, 31, 64)            192

 flatten_3 (Flatten)         (None, 1984)              0

 dense_26 (Dense)            (None, 64)                127040

 dropout_15 (Dropout)        (None, 64)                0

 dense_27 (Dense)            (None, 1)                 65

=================================================================
Total params: 127297 (497.25 KB)
Trainable params: 127297 (497.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= model.fit(X_train_unigram, y_train, validation_data=(X_test_unigram, y_test), epochs=10)
```

```
Epoch 1/10
3/3 [==============================] - 22s 2s/step - loss: 3.4968 - accuracy: 0.6375 - val_loss: 0.6837 - val_accuracy: 0.8500
Epoch 2/10
3/3 [==============================] - 1s 292ms/step - loss: 1.6067 - accuracy: 0.6125 - val_loss: 0.8019 - val_accuracy: 0.550
0
Epoch 3/10
3/3 [==============================] - 1s 288ms/step - loss: 1.0409 - accuracy: 0.7000 - val_loss: 0.6621 - val_accuracy: 0.850
```

Figure 26: Implementation of CNN

7.4 RNN Bigram

```
rnn = Sequential()
rnn.add(Input(shape=(X_train_bigram.shape[1],1)))
rnn.add(Dense(64, activation='tanh'))
rnn.add(Dropout(0.2))
rnn.add(Dense(32, activation='tanh'))
rnn.add(Dropout(0.2))
rnn.add(Dense(1, activation='tanh'))
print(rnn.output_shape)
print(rnn.compute_output_signature)
rnn.compile(loss="binary_crossentropy", metrics=["accuracy"])
rnn.summary()
```

```
(None, 31, 1)
<bound method Layer.compute_output_signature of <keras.src.engine.sequential.Sequential object at 0x0000015367CBBA90>>
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_28 (Dense)            (None, 31, 64)            128

 dropout_16 (Dropout)        (None, 31, 64)            0

 dense_29 (Dense)            (None, 31, 32)            2080

 dropout_17 (Dropout)        (None, 31, 32)            0

 dense_30 (Dense)            (None, 31, 1)             33

=================================================================
Total params: 2241 (8.75 KB)
Trainable params: 2241 (8.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= rnn.fit(X_train_bigram, y_train, validation_data=(X_test_bigram, y_test), epochs=10)
```

```
Epoch 1/10
3/3 [==============================] - 23s 3s/step - loss: 3.4525 - accuracy: 0.5914 - val_loss: 1.8413 - val_accuracy: 0.5994
Epoch 2/10
3/3 [==============================] - 1s 268ms/step - loss: 3.1532 - accuracy: 0.6231 - val_loss: 1.9943 - val_accuracy: 0.599
4
Epoch 3/10
3/3 [==============================] - 1s 311ms/step - loss: 3.1171 - accuracy: 0.6232 - val_loss: 1.9531 - val_accuracy: 0.599
4
Epoch 4/10
3/3 [==============================] - 1s 311ms/step - loss: 3.0701 - accuracy: 0.6422 - val_loss: 1.8073 - val_accuracy: 0.599
4
Epoch 5/10
3/3 [==============================] - 1s 273ms/step - loss: 3.1561 - accuracy: 0.6656 - val_loss: 1.7249 - val_accuracy: 0.850
0
```

Figure 27: Implementation of RNN

7.5 LSTM Bigram

```
lstm = Sequential()
lstm.add(Input(shape=(X_train_bigram.shape[1],1)))
lstm.add(LSTM(64, activation='relu'))
lstm.add(Dense(16, activation='tanh'))
lstm.add(Dropout(0.3))
lstm.add(Dense(1, activation='sigmoid'))
print(lstm.output_shape)
print(lstm.compute_output_signature)
lstm.compile(loss="binary_crossentropy", metrics=["accuracy"])
lstm.summary()
```

```
(None, 1)
<bound method Layer.compute_output_signature of <keras.src.engine.sequential.Sequential object at 0x00000153631626D0>>
Model: "sequential_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 64)                16896

 dense_31 (Dense)            (None, 16)                1040

 dropout_18 (Dropout)        (None, 16)                0

 dense_32 (Dense)            (None, 1)                 17

=================================================================
Total params: 17953 (70.13 KB)
Trainable params: 17953 (70.13 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= lstm.fit(X_train_bigram, y_train, validation_data=(X_test_bigram, y_test), epochs=10)
```

```
Epoch 1/10
3/3 [==============================] - 41s 3s/step - loss: 0.7353 - accuracy: 0.6000 - val_loss: 0.6697 - val_accuracy: 0.7000
Epoch 2/10
3/3 [==============================] - 1s 341ms/step - loss: 0.8449 - accuracy: 0.6000 - val_loss: 0.8204 - val_accuracy: 0.450
0
Epoch 3/10
3/3 [==============================] - 1s 333ms/step - loss: 0.9791 - accuracy: 0.5000 - val_loss: 0.8066 - val_accuracy: 0.500
0
Epoch 4/10
3/3 [==============================] - 1s 324ms/step - loss: 0.7951 - accuracy: 0.6000 - val_loss: 0.6893 - val_accuracy: 0.600
0
Epoch 5/10
3/3 [==============================] - 1s 311ms/step - loss: 0.6530 - accuracy: 0.6625 - val_loss: 0.6042 - val_accuracy: 0.550
0
```

Figure 28: Implementation of LSTM

7.6 CNN Bigram

```
X_train_bigram = np.reshape(X_train_bigram, (X_train_bigram.shape[0], X_train_bigram.shape[1], 1))
X_train_bigram.shape

(80, 31, 1)

X_test_bigram = np.reshape(X_test_bigram, (X_test_bigram.shape[0], X_test_bigram.shape[1], 1))
X_test_bigram.shape

(20, 31, 1)

model = Sequential()
model.add(layers.Conv1D(64, 2, activation="relu", padding="same", name="convLayer", input_shape=(X_train_bigram.shape[1],1)))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation ='relu'))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(1, activation ='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

model.summary()

Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 convLayer (Conv1D)          (None, 31, 64)            192

 flatten_4 (Flatten)         (None, 1984)              0

 dense_33 (Dense)            (None, 64)                127040

 dropout_19 (Dropout)        (None, 64)                0

 dense_34 (Dense)            (None, 1)                 65

=================================================================
Total params: 127297 (497.25 KB)
Trainable params: 127297 (497.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

history= model.fit(X_train_bigram, y_train, validation_data=(X_test_bigram, y_test), epochs=10)

Epoch 1/10
3/3 [==============================] - 23s 2s/step - loss: 1.6410 - accuracy: 0.6125 - val_loss: 1.7551 - val_accuracy: 0.600
0
Epoch 2/10
3/3 [==============================] - 0s 232ms/step - loss: 1.3777 - accuracy: 0.6875 - val_loss: 1.5184 - val_accuracy: 0.8
000
Epoch 3/10
3/3 [==============================] - 1s 299ms/step - loss: 1.1110 - accuracy: 0.7750 - val_loss: 1.2727 - val_accuracy: 0.5
```

Figure 29: Implementation of CNN

7.7 RNN Trigram

```
rnn = Sequential()
rnn.add(Input(shape=(X_train_trigram.shape[1],1)))
rnn.add(Dense(64, activation='tanh'))
rnn.add(Dropout(0.2))
rnn.add(Dense(32, activation='tanh'))
rnn.add(Dropout(0.2))
rnn.add(Dense(1, activation='tanh'))
print(rnn.output_shape)
print(rnn.compute_output_signature)
rnn.compile(loss="binary_crossentropy", metrics=["accuracy"])
rnn.summary()

(None, 31, 1)
<bound method Layer.compute_output_signature of <keras.src.engine.sequential.Sequential object at 0x00000153628EB190>>
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_35 (Dense)            (None, 31, 64)            128

 dropout_20 (Dropout)        (None, 31, 64)            0

 dense_36 (Dense)            (None, 31, 32)            2080

 dropout_21 (Dropout)        (None, 31, 32)            0

 dense_37 (Dense)            (None, 31, 1)             33

=================================================================
Total params: 2241 (8.75 KB)
Trainable params: 2241 (8.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

history= rnn.fit(X_train_trigram, y_train, validation_data=(X_test_trigram, y_test), epochs=20)

Epoch 1/20
3/3 [==============================] - 22s 2s/step - loss: 1.9102 - accuracy: 0.7140 - val_loss: 0.6192 - val_accuracy: 0.6219
Epoch 2/20
3/3 [==============================] - 1s 259ms/step - loss: 0.9095 - accuracy: 0.6643 - val_loss: 0.4804 - val_accuracy: 0.850
0
Epoch 3/20
3/3 [==============================] - 1s 325ms/step - loss: 0.9989 - accuracy: 0.6927 - val_loss: 0.6381 - val_accuracy: 0.621
9
Epoch 4/20
3/3 [==============================] - 1s 344ms/step - loss: 0.8858 - accuracy: 0.6591 - val_loss: 0.5255 - val_accuracy: 0.621
9
```

Figure 30: Implementation of RNN

7.8 LSTM Trigram

```
lstm = Sequential()
lstm.add(Input(shape=(X_train_trigram.shape[1],1)))
lstm.add(LSTM(64, activation='relu'))
lstm.add(Dense(16, activation='tanh'))
lstm.add(Dropout(0.3))
lstm.add(Dense(1, activation='sigmoid'))
print(lstm.output_shape)
print(lstm.compute_output_signature)
lstm.compile(loss="binary_crossentropy", metrics=["accuracy"])
lstm.summary()
```

```
(None, 1)
<bound method Layer.compute_output_signature of <keras.src.engine.sequential.Sequential object at 0x0000015361FA9B90>>
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_5 (LSTM)               (None, 64)                16896

 dense_38 (Dense)            (None, 16)                1040

 dropout_22 (Dropout)        (None, 16)                0

 dense_39 (Dense)            (None, 1)                 17

=================================================================
Total params: 17953 (70.13 KB)
Trainable params: 17953 (70.13 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= lstm.fit(X_train_trigram, y_train, validation_data=(X_test_trigram, y_test), epochs=20)
```

```
Epoch 1/20
3/3 [==============================] - 32s 2s/step - loss: 0.7725 - accuracy: 0.6375 - val_loss: 0.5817 - val_accuracy: 0.8500
Epoch 2/20
3/3 [==============================] - 1s 285ms/step - loss: 0.5267 - accuracy: 0.7375 - val_loss: 0.5702 - val_accuracy: 0.850
0
Epoch 3/20
3/3 [==============================] - 1s 278ms/step - loss: 0.6142 - accuracy: 0.6875 - val_loss: 0.4733 - val_accuracy: 0.850
0
Epoch 4/20
3/3 [==============================] - 1s 259ms/step - loss: 0.5411 - accuracy: 0.7500 - val_loss: 0.4850 - val_accuracy: 0.850
0
Epoch 5/20
3/3 [==============================] - 1s 255ms/step - loss: 0.6285 - accuracy: 0.6875 - val_loss: 0.4772 - val_accuracy: 0.850
```

Figure 31: Implementation of LSTM

7.9 CNN Unigram

```
X_train_trigram = np.reshape(X_train_trigram, (X_train_trigram.shape[0], X_train_trigram.shape[1], 1))
X_train_trigram.shape
```

```
(80, 31, 1)
```

```
X_test_trigram = np.reshape(X_test_trigram, (X_test_trigram.shape[0], X_test_trigram.shape[1], 1))
X_test_trigram.shape
```

```
(20, 31, 1)
```

```
model = Sequential()
model.add(layers.Conv1D(64, 2, activation="relu", padding="same", name="convLayer", input_shape=(X_train_trigram.shape[1],1)))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation ='relu'))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(1, activation ='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential_17"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 convLayer (Conv1D)          (None, 31, 64)            192

 flatten_5 (Flatten)         (None, 1984)              0

 dense_40 (Dense)            (None, 64)                127040

 dropout_23 (Dropout)        (None, 64)                0

 dense_41 (Dense)            (None, 1)                 65

=================================================================
Total params: 127297 (497.25 KB)
Trainable params: 127297 (497.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history= model.fit(X_train_trigram, y_train, validation_data=(X_test_trigram, y_test), epochs=10)
```

```
Epoch 1/10
3/3 [==============================] - 13s 1s/step - loss: 7.0007 - accuracy: 0.5500 - val_loss: 5.9957 - val_accuracy: 0.8500
Epoch 2/10
3/3 [==============================] - 0s 157ms/step - loss: 5.4052 - accuracy: 0.7500 - val_loss: 2.9280 - val_accuracy: 0.700
0
Epoch 3/10
3/3 [==============================] - 0s 155ms/step - loss: 3.0118 - accuracy: 0.6125 - val_loss: 4.1591 - val_accuracy: 0.400
0
```

Figure 32: Implementation of CNN

7 Model result

This section explains the performance of the models.

7.1 Model Scores

	Model	Accuracy
0	RNN Unigram	79.999977
0	LSTM Unigram	80.000001
0	CNN Unigram	85.000000
0	RNN Bigram	61.225814
0	LSTM Bigram	55.000001
0	CNN Bigram	75.000000
0	RNN Trigram	63.161284
0	LSTM Trigram	80.000001
0	CNN Trigram	70.000000

Figure 33: Model Performance Overall

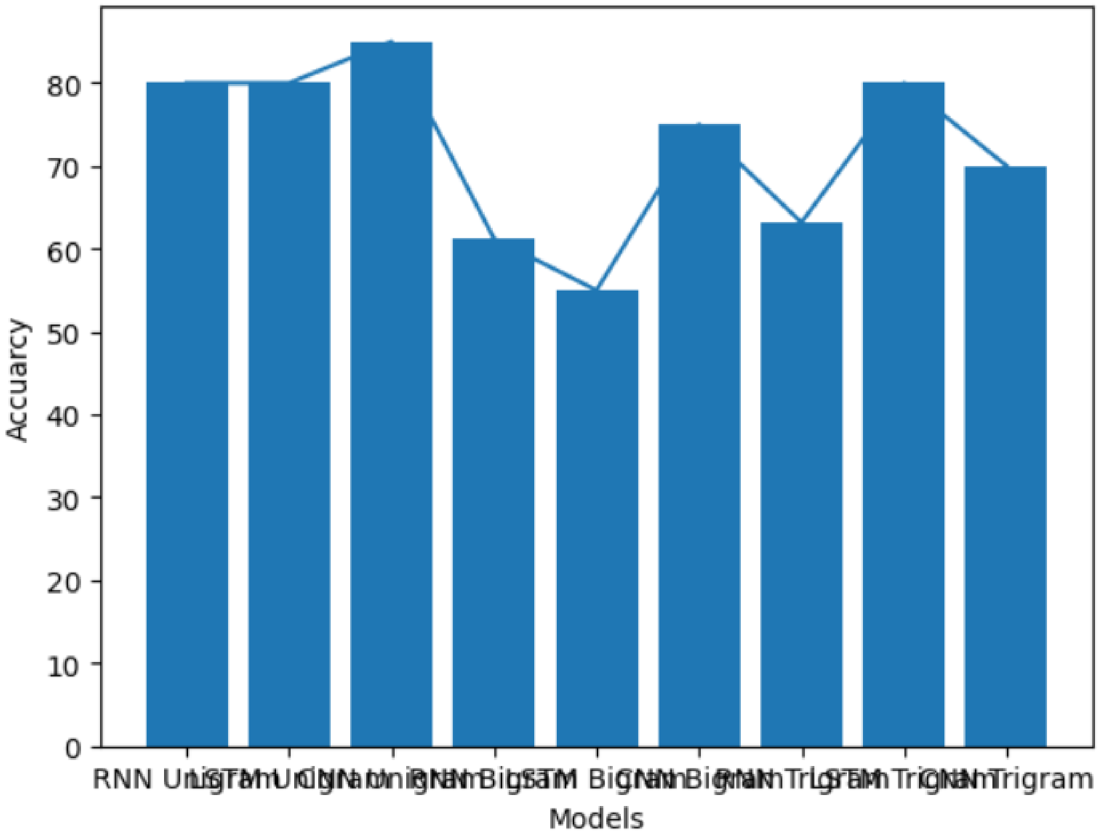

Figure 34: Model Performance Overall

	Model	Accuracy
0	RNN	79.999977
0	LSTM	80.000001
0	CNN	85.000000

```
plt.plot(uniScore['Model'], uniScore['Accuracy'])
plt.bar(uniScore['Model'], uniScore['Accuracy'])
plt.xlabel('Models')
plt.ylabel('Accuarcy')
```

```
Text(0, 0.5, 'Accuarcy')
```

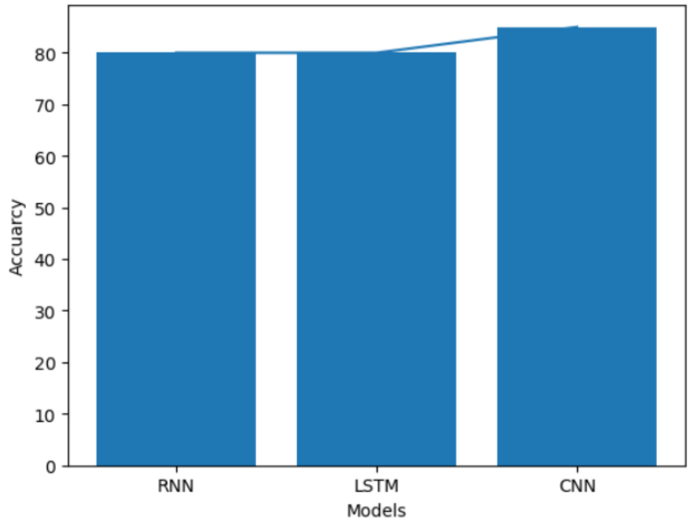

Figure 35: Model Performance Unigram

```
In [119]: biScore.columns = ['Model' , 'Accuracy']
          biScore
```

Out[119]:

	Model	Accuracy
0	RNN	61.225814
0	LSTM	55.000001
0	CNN	75.000000

```
In [120]: plt.plot(biScore['Model'], biScore['Accuracy'])
          plt.bar(biScore['Model'], biScore['Accuracy'])
          plt.xlabel('Models')
          plt.ylabel('Accuarcy')
```

```
Out[120]: Text(0, 0.5, 'Accuarcy')
```

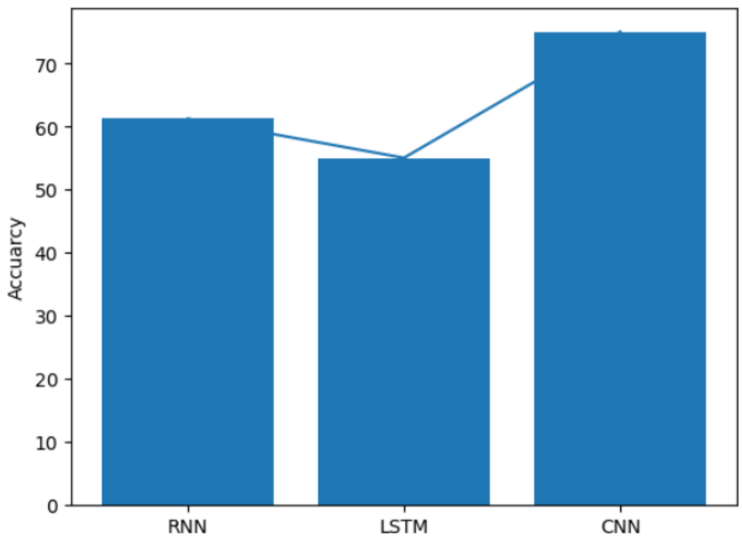

Figure 36: Model Performance Bigram

In [121]:
```
triScore.columns = ['Model' , 'Accuracy']
triScore
```

Out[121]:

	Model	Accuracy
0	RNN	63.161284
0	LSTM	80.000001
0	CNN	70.000000

In [122]:
```
plt.plot(triScore['Model'], triScore['Accuracy'])
plt.bar(triScore['Model'], triScore['Accuracy'])
plt.xlabel('Models')
plt.ylabel('Accuarcy')
```

Out[122]: Text(0, 0.5, 'Accuarcy')

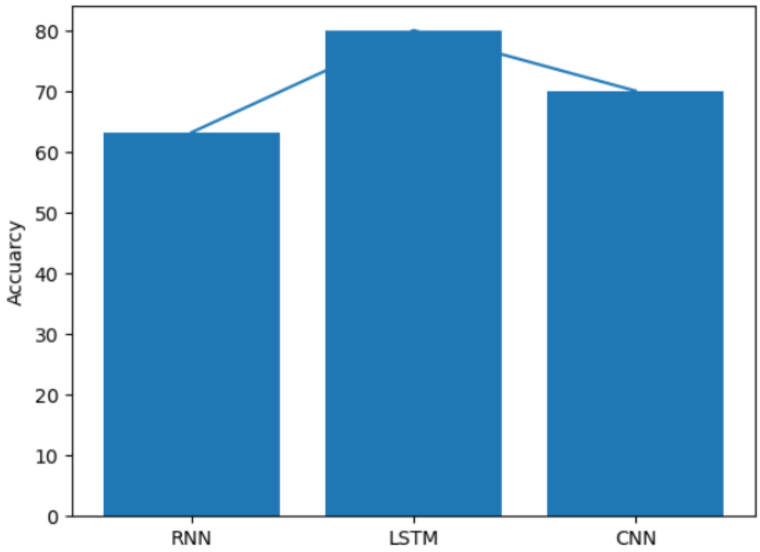

Figure 37: Model Performance Trigram

References

https://www.topcoder.com/thrive/articles/web-scraping-with-beautiful-soup

https://www.analyticsvidhya.com/blog/2021/06/vader-for-sentiment-analysis/

https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

https://www.analyticsvidhya.com/blog/2021/09/an-explanatory-guide-to-bert-tokenizer/

https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm

https://www.geeksforgeeks.org/introduction-convolution-neural-network/