Twitter Sentiment Analysis

PROJECT SUBMITTED FOR THE AWARD OF THE DEGREE OF

Bachelors of Computer Application (BCA)

IN INFORMATION TECHNOLOGY / COMPUTER APPLICATION FACULTY OF SCIENCE AND TECHNOLOGY

TO THE

AMITY UNIVERSITY, MAHARASHTRA, INDIA

BY

Ms. Janani Jaganathan Naidu

(Enrolment No: A71004821023)

Under the Guidance of

Dr. Karuna Kirwale

Assistant Professor, Amity Institute of Information Technology

AMITY INSTITUTE OF INFORMATION TECHNOLOGY

AMITY UNIVERSITY, MAHARASHTRA, INDIA

MAY 2024

TABLE OF CONTENTS

DECLARATION AND CERTIFICATE

This is to certify that this project report entitled: **"Twitter Sentiment Analysis"** submitted by **Ms**. **Janani Jaganathan Naidu** in partial fulfilment of the requirement of the degree of **BCA** (**Bachelors of Computer Application)** in the Amity Institute of Information Technology, Amity University Maharashtra, is based on the project and research work carried under the guidance and supervision of **Dr. Karuna Kirwale.** The manuscript has been subjected to plagiarism check by TURNITIN software. This project report and any part thereof had not been submitted for any purpose to any University or Institute.

Place: Mumbai
Date: 10/05/2024

Ms. Janani Jaganathan Naidu
Student

Dr. Karuna Kirwale
Guide

Prof. (Dr.) Manoj Devare
HOI, Amity Institute of Information Technology Amity University Maharashtra.

ACKNOWLEDGEMENT

Many people assisted me in successfully finishing this project. I want to thank everyone involved in this initiative. I’d like to thank my HOI **Prof. (Dr.) Manoj Devare** and my Mentor **Dr. Karuna Kirwale**, who helped me learn a lot about this project. There ideas and comments aided in the completion of this project.

I am grateful to the college administration for providing me with such a significant chance. I believe I will participate in more such activities in the future. I guarantee that this project was created entirely by me and is not a forgery. Finally, I’d like to express my gratitude to my Mentor for the excellent comment and guidance during the completion of this project.

Place: Mumbai

Date: 10/05/2024

Ms. Janani Jaganathan Naidu

Student

List of Acronyms and Abbreviation

Abbreviation	Description
TSA	Twitter Sentiment Analysis
SNS	Social Networking Service
SA	Sentiment Analysis
NB	Naive Bayes
NLP	Natural Language Processing
SVM	Support Vector Machine
LSTM	Long Short-Term Memory
BERT	Bidirectional Encoder Representations from Transformers
RoBERTa	Robustly Optimized BERT Approach

List of Figures

Abstract

Twitter has grown to be a significant social media site and has drawn a lot of attention from sentiment analysis academics. The study of Twitter Sentiment Analysis (TSA) is a developing area of text mining research. The term "TSA" describes the process of processing subjective Twitter data—including views and sentiments—by means of computers. This study examines a wide range of recently suggested algorithms and applications in addition to providing a complete evaluation of the most current advancements in the field. Every article is categorized according to how important it is to a certain kind of TSA procedure. This survey aims to give a succinct, almost all-inclusive review of TSA methods and associated disciplines.

The rise of SNS gave rise to a number of microblogging services, including Facebook, Instagram, and Twitter. Users of the popular social networking site Twitter may communicate by exchanging 140-character messages, or "tweets". Twitter has over 300 million registered users and produces over 500 million updates daily. Due to its ease of sharing, Twitter has become one of the most significant sources of user-generated content.

The most significant aspects of Twitter are enumerated below:

Tweet: A tweet is the maximum data unit of 140 characters that may be sent via Twitter. Its content includes images, movies, links, and personal reflections on various events. All of these may be shared with others with ease. Its material includes images, videos, links, and personal reflections on various events. Users may share any or all of these items with their connections with ease.

Handle: This describes how you update your tweets or message other individuals in public. The @ sign is used to identify the individual or group that the tweets are associated with, and it is written as "@username".

Hashtag: Hashtags are a type of metadata tag that may be used on many social networking sites (SNS). Users can use dynamic, user-generated hashtags to help other users identify tweets that are relevant to a certain topic.

Retweet: Allowing users to re-post tweets they find interesting; this is one of the most helpful tools on Twitter for spreading information. Here, the writers' initial username is shortened after the original tweets, which usually stay the same.

Search: With the help of this useful tool, users may look up relevant tweets about their interests in real time on Twitter by using keywords and phrases. Users are more inclined to sign up for Twitter accounts due to this search feature, which makes it easier to find and share relevant material.

Chapter 1: Introduction

This is the machine learning project for sentiment analysis on Twitter. Here, we implement our model using matplotlib, numpy, scikit-learn, pandas, and python.

Twitter is a social networking site, as we all know. Everyone is able to register here and express their own opinions. It appears that there are drawbacks to expressing independent opinions on social media, despite its immense influence. On social media, people occasionally express their opinions without doing enough investigation or knowing the whole story. People are thus becoming perplexed about which information is false and which is true.

Our objective is to assess a tweet's sentiment regarding a subject. It is incredibly useful for gauging consumer opinion of new products. Three conclusions may be drawn from the sentiment analysis. The emotion may be neutral, negative, or pleasant. We may evaluate the new product's openness based on that outcome.

Analysis and Explanation of the Machine Learning Approaches (Naïve bayes, SVM): -

Naïve bayes approaches is one of the most known Machine Learning Concept. Text preparation, sentiment labeling, and text classification using a Naive Bayes classifier are done on Twitter data that is saved in a pandas Data Frame.

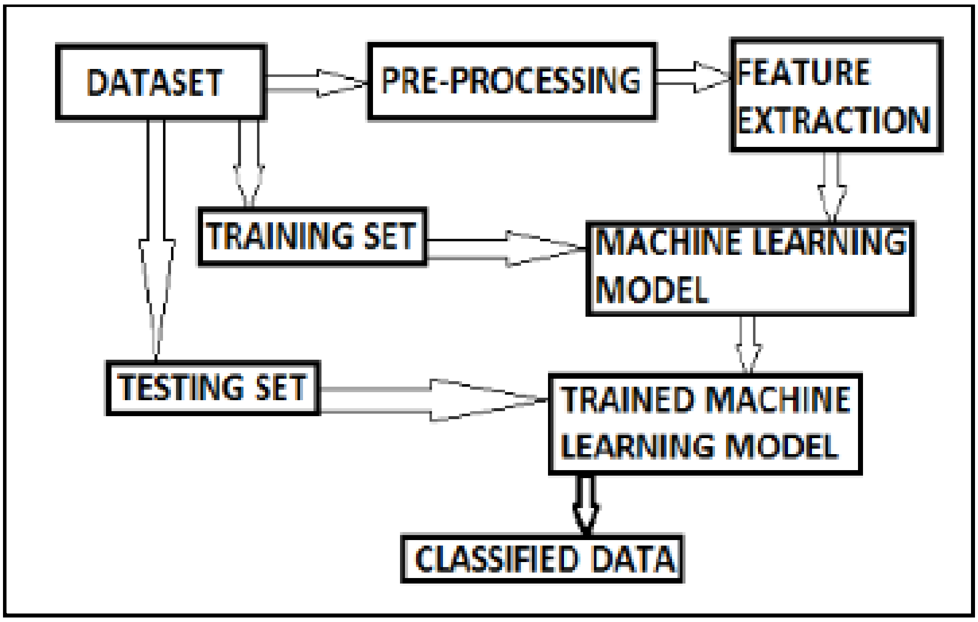


Fig.1: General Processing Cycle in Machine Learning

First, the code looks at the Data Frame’s structure, deleting rows that have missing data by looking for missing values in the "clean_text" and "category" columns. Next, based on this mapping, a new ‘sentiment' column is added to the DataFrame, mapping sentiment values (-1, 0, 1) to sentiment labels (negative, neutral, and positive).

Using a custom pre-processor function, the text data is pre-processed to exclude mentions, hashtags, URLs, and punctuation. After tokenizing the text, the text data is transformed into numerical characteristics for classification using a CountVectorizer. A Bernoulli Naive Bayes classifier is trained on the training data after the data is divided into training and testing sets.

Finally, the code evaluates the classifier's performance by predicting sentiment labels the test calculating the accuracy score using scikit-learn's ***metrics.accuracy_score() function.***

Overall, the code processes Twitter data, prepares it for classification, trains a Naive Bayes classifier, and assesses the model's accuracy in predicting sentiment labels.

Here is a breakdown of the code:

data.head(): Displays the first few rows of the DataFrame data.

data.shape: Returns the shape (number of rows and columns) of the DataFrame.

data.category.unique(): Returns the unique values in the 'category' column of the DataFrame.

data.isna().sum(): Calculates the number of missing values in each column of the DataFrame.

data[data['category'].isn()} : This line seems to have a syntax error. It should be data[data['category'].isna()], which filters rows where the 'category' column is NaN.

data[data['clean_text'].isna()]: Filters rows where the 'clean_text' column is NaN.

data.drop(data[data['clean_text'].isna()].index, inplace=True): Drops rows where the 'clean_text' column is NaN from the DataFrame.

data.drop(data[data['category'].isna()].index, inplace=True): Drops rows where the 'category' column is NaN from the DataFrame.

sentiment_map={-1:'negative',1:'positive',0:'neutral'}: Creates a mapping from sentiment values to sentiment labels.

data.insert(2,'sentiment',[sentiment_map[s] for s in data.category],True): Inserts a new column 'sentiment' in the DataFrame based on the mapping of sentiment values.

reviews = np.array(data['clean_text'])[:]: Extracts the 'clean_text' column values as an array.

labels = np.array(data['sentiment'])[:]: Extracts the 'sentiment' column values as an array.

The code then proceeds to pre-process the text data, vectorize it using CountVectorizer, split the data into training and testing sets, and train a Naive Bayes classifier on the text data.

Finally, it the accuracy of the classifier on the test data using ***metrics.accuracy_score()*** from scikit-learn.This code snippet essentially prepares the text data, performs classification using a Naive Bayes classifier, and evaluates the model's accuracy.

Secondly, from Machine Learning Concept, **SVM** is chosen in this project. The code employs a Linear Support Vector Classification (LinearSVC) model to do sentiment analysis on Twitter data. The code begins by importing the required libraries, including metrics, itertools, matplotlib.pyplot, and svm from sklearn. After that, a function called ***plot_confusion_matrix*** is defined to show the confusion matrix and assess the model's effectiveness.

Using the training data (x_train and y_train), the script trains a LinearSVC classifier, and then uses the test data (x_test) to generate predictions. It uses metrics.accuracy_score to determine the prediction accuracy score. The ***plot_confusion_matrix*** function is used to display the confusion matrix in both non-normalized and normalized formats after it has been generated using the ***confusion_matrix*** function from ***sklearn.metrics***.

Two plots of the confusion matrix are used to show how well the model predicts the sentiment ('positive, negative, and neutral'). This code involves the use of a Support Vector Machine (SVM) classifier and a Linear Support Vector Classification (LinearSVC) model for sentiment analysis on Twitter data. Here is a summary of the code:

The code imports necessary libraries such as svm from sklearn, matplotlib.pyplot, metrics, and itertools.It defines a function ***plot_confusion_matrix*** to visualize the confusion matrix, allowing for normalization of the matrix if required.

The code trains a LinearSVC classifier on the training data (x_train and y_train) and makes predictions on the test data (x_test). It then calculates the accuracy score of the predictions using ***metrics.accuracy_score***.

The confusion is computed using the ***confusion_matrix*** function from ***sklearn.metrics***, and it is displayed in both non-normalized and using the ***plot_confusion_matrix*** function.

The confusion matrix is plotted twice, first without normalization and then with normalization, to visualize the performance of the LinearSVC classifier in predicting sentiment labels ('positive', 'negative', 'neutral').

Overall, the code trains a LinearSVC classifier for sentiment analysis, evaluates its performance using the confusion matrix, and visualizes the results to understand the model's predictive accuracy for different sentiment classes.

Analysis and Explanation of the Deep Learning Approaches (LSTM): -

This code represents a PyTorch implemented sentiment analysis model. Importing the required libraries, such as NumPy, Pandas, and PyTorch, is the first step. The CSV file containing tweets and the sentiment labels that go with them is used to load the dataset. Pre-processing operations are performed on the text data, including the removal of numerals, URLs, IDs, and punctuation.

Tokenizing the text, generating a word dictionary, encoding labels, and padding sequences to a predetermined length are further pre-processing steps. A recurrent neural network (RNN) with an embedding layer, long short-term memory (LSTM) layers, dropout, and a fully connected layer with a sigmoid activation for binary classification is the definition of the model architecture.

Binary cross-entropy loss and the Adam optimizer are used to train the model. Training is done in epochs, where batches of data are iterated through throughout each epoch. The application of gradient trimming stops gradients from bursting. The code monitors training and validation losses and provides steps for testing, and training. Metrics like test loss and accuracy are computed and shown during testing, and the model's performance is assessed on a different test set. When compared to the actual labels in the test set, the accuracy shows how well the predicts the sentiment labels.

This code illustrates a deep learning-based sentiment analysis pipeline that includes preparing data, building a model, training it, and evaluating it. This is a thorough implementation for text data sentiment analysis jobs.

Here's a detailed explanation of the code's functionality:

1. ***Data Pre-processing:***

   - Imports necessary libraries such as NumPy, Pandas, and collections.
   - Reads a CSV file ('Tweets.csv') containing Twitter data into a Pandas DataFrame.
   - Processes the text data, removes punctuation, URLs, Twitter handles, and digits.
   - Tokenizes the text data and encodes it into integer values.
   - Prepares the data for training, validation, and testing sets.

2. ***Model Architecture:***

   - Defines a class SentimentRNN that represents the sentiment analysis model using an RNN architecture.
   - Sets up embedding, LSTM (Long Short-Term Memory) layers, dropout, linear, and sigmoid layers in the model.
   - Initializes hidden states for the LSTM.

3. ***Training:***

- Sets hyperparameters like learning rate, epochs, and batch size.
- Initializes the model and moves it to GPU if available.
- Performs training using the training dataset and calculates validation loss.
- Clips gradients to prevent exploding gradients.
- Prints training and validation losses during training.

4. ***Testing:***

- Evaluates the trained model using the test dataset.
- Calculates test loss and accuracy over the test data.

5. ***Output:***

- Prints out test loss and accuracy after training and testing the model.

Analysis and Explanation of the Transformers Approaches (BERTs, Roberta): -

The code clearly explains in detail how to refine a BERT model that has already been trained for sentiment classification using Twitter data. Let's dissect the main elements and features one by one:

1. Importing Libraries: The code starts by importing the required libraries, including matplotlib, pandas, numpy, seaborn, pandas, PyTorch, and Transformers (Hugging Face's library for pre-trained models). It also imports a number of utilities for training and assessment.

2. Configuring Device and Seed: The training device (GPU or CPU) is recognized and defined. For repeatability, a seed is also set.

3. Loading and Pre-processing Data: A CSV file called df_train is used to load and pre-process the Twitter data. It is then verified for null values. Tokenizing sentences using BERT's tokenizer and encoding labels with LabelEncoder are two pre-processing steps for text data. To distinguish between padding tokens and genuine tokens, attention masks are developed.

4. Data Splitting: Train_test_split is used to divide the pre-processed data into training and validation sets. For effective training, the input data, labels, and attention masks are transformed into torch tensors and arranged into data loaders using DataLoader.

5. Initializing BERT Model: The device (CPU or GPU) that has been designated is loaded with the BERT model for sequence classification (BertForSequenceClassification). A pretrained Roberta model (a variant of BERT) for sequence classification is loaded from the Transformers library. The model is configured with the appropriate number of labels

6. Training Loop: Throughout several epochs, the code conducts a training loop. It iterates over batches of training set data inside each epoch, executing forward and backward passes, updating model parameters, and modifying the learning rate via a scheduler. Learning rates and training loss are monitored.

7. Validation and Evaluation: The model is assessed on the validation set following each epoch. A number of metrics are computed and printed, including accuracy and Matthews Correlation Coefficient (MCC). For the purpose of displaying classification results, confusion matrix charting functionality is also offered.

8. Saving the Model and Tokenizer: Lastly, the optimized model and tokenizer are stored for later use in designated folders. This bit of code implements the use of PyTorch and the Transformers module from Hugging Face to fine-tune the Roberta model for sentiment categorization. Importing the required libraries for data management, model training, and assessment is the first step. The device designated for model training (CPU or GPU) is recognized and defined. After loading the dataset, it undergoes several pre-processing steps such as managing missing values, label encoding, BERT tokenization of text, attention mask creation, and data division into training and validation sets.

After that, a classification layer is added to the top of the Roberta model for sentiment analysis. In order to modify the learning rate during training, a scheduler is created, and training parameters like the learning rate, optimizer, and number of epochs are configured. The training loop computes loss, updates gradients, and evaluates on the validation set as iteratively goes over the data for the predetermined number of epochs.

Metrics like accuracy and the Matthews correlation coefficient (MCC) are computed and monitored during the validation process. In addition, the code has functions for creating a classification report based on the model's predictions and plotting a confusion matrix.

In summary, this code creates an all-inclusive pipeline that includes data preparation, model setup, training, evaluation, and result visualization for the purpose of fine-tuning a Roberta model for sentiment categorization.

Chapter 2: Literature Review

2.1 Related Work

Zhang et al. (2021) explored fine-tuning RoBERTa for sentiment analysis in social media data, showcasing its ability to handle informal language, slang, and context-dependent sentiment expressions effectively.

Hassonah et al. (2020) recommended a hybrid machine learning algorithm for sentiment analysis, integrating feature selection methods and achieving superior performance.

Xu et al. (2020) extended the Naive Bayes method for large-scale sentiment classification, yielding high accuracy.

Fang et al. (2018) introduced multi-strategy sentiment analysis models leveraging semantic fuzziness to address inherent challenges, achieving high efficiency.

Afzaal et al. (2019) recommended an aspect-based sentiment classification approach, developing a mobile application for tourists to identify the best hotels. Their model, analysed with real-world data, showed effective recognition and classification capabilities.

Feizollah et al. (2019) focused on sentiment analysis of tweets related to halal products, employing deep learning models like RNN, CNN, and LSTM for accuracy enhancement.

Hochreiter and Schmidhuber (1997) introduced LSTM as a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem, making it well-suited for sequence modelling tasks.

Graves et al. (2009) further refined LSTM for handwriting recognition, showcasing its ability to capture long-range dependencies and remember relevant information over extended sequences.

Sutskever et al. (2014) demonstrated the effectiveness of LSTM in natural language processing (NLP) tasks such as language translation, where maintaining context over lengthy sentences is crucial.

Abdi et al. (2018) proposed a feature-rich machine learning technique for summarizing user opinions from reviews, with SVM-based classification and feature selection enhancing performance significantly.

Ray and Chakrabarti (2019) employed deep learning algorithms for feature extraction and sentiment analysis, achieving high accuracy.

Zhao et al. (2019) developed a multi-modal sentiment evaluation model, integrating text and image features effectively.

Vashishtha and Susan (2019) introduced a fuzzy rule-based model for social media sentiment analysis, outperforming existing models.

Devlin et al. (2018) introduced BERT as a transformer-based language model pre-trained on large text corpora, achieving state-of-the-art results across various NLP benchmarks by leveraging bidirectional context.

Liu et al. (2019) extended BERT with enhancements like RoBERTa, focusing on larger training datasets and longer sequences, leading to improved performance on tasks like question answering and sentiment analysis.

Yang et al. (2020) explored fine-tuning strategies for BERT in specific domains, showcasing its adaptability and effectiveness in specialized tasks such as biomedical text mining and financial sentiment analysis.

Liu et al. (2019) introduced RoBERTa as an optimized version of BERT, incorporating larger training datasets, longer sequences, and dynamic masking strategies to enhance model robustness and performance.

Yasunaga et al. (2020) applied RoBERTa to information extraction tasks in the medical domain, demonstrating its capability to extract structured information from unstructured clinical text with high accuracy and efficiency.

These studies collectively demonstrate the diverse approaches and advancements in Twitter sentiment analysis, from feature-rich machine learning techniques to hybrid deep learning models, addressing the intricacies of sentiment extraction from short-form, dynamic content.

Chapter 3: Methodology

Here, there are three type of approach.

3.1 Machine Learning - SVM, Naive Bayes

3.2 Deep Learning - LSTM

3.3 Transformer - BERT, ROBERTA

3.1 Machine Learning: -

SVM:

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms. One of the most widely used supervised learning techniques for both classification and regression issues is support vector machine, or SVM. But it's mostly applied to machine learning classification challenges.

In order to make it simple to classify fresh data points in the future, the SVM method seeks to identify the optimal line or decision boundary that may divide n-dimensional space into classes. We refer to this optimal decision boundary as a hyperplane.

SVM selects the extreme vectors and points to aid in the creation of the hyperplane. The technique is referred regarded as a Support Vector Machine since these extreme situations are known as support vectors. Examine the picture below, which shows two distinct groups that are categorized using a decision boundary or hyperplane:

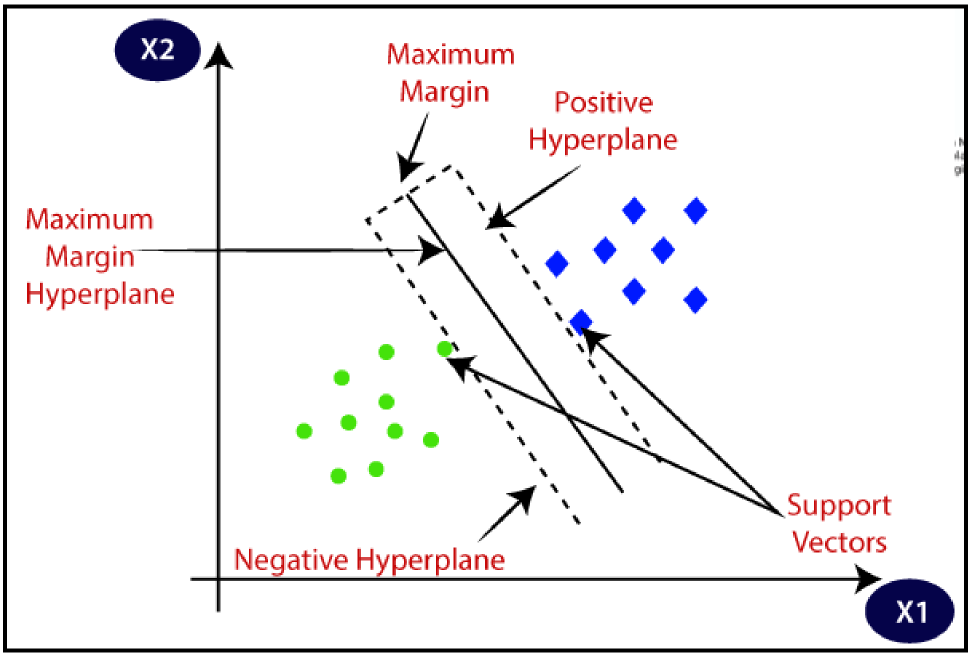


Fig.2: Graphic representation od SVM

Naive Bayes:

Naive Bayes classifier is a probabilistic machine learning model that’s used for classification task. The crux of the classifier is based on the Bayes theorem.

Bayes Theorem:

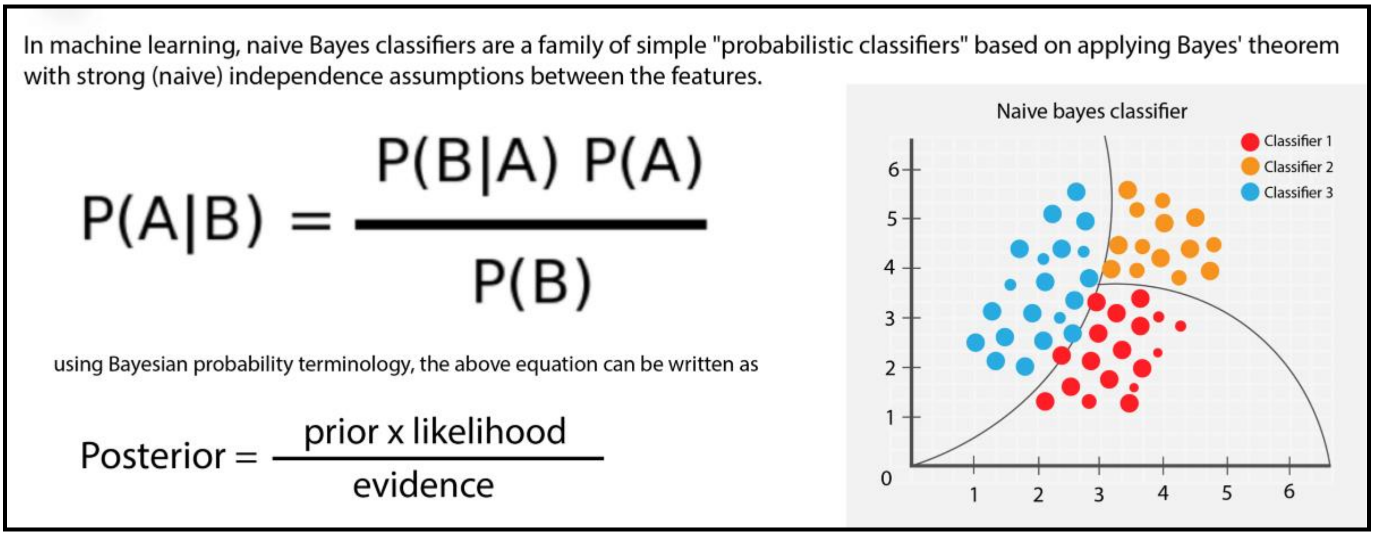


Fig.3: Formula and Graphic representation of Bayes Theorem

Using Bayes theorem, we can find the probability of A happening, given that B has occurred. Here, B is the evidence and A is the hypothesis. The assumption made here is that the predictors/features are independent. That is presence of one particular feature does not affect the other. Hence it is called naive.

3.2 Deep Learning: -

LSTM:

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are purposefully made to circumvent the issue of long-term reliance. They don't strive to learn; rather, being able to retain knowledge for extended periods of time is basically their default habit!

Every recurrent neural network is composed of a series of neural network modules that repeat. This repeating module in conventional RNNs will have a very basic structure, like a single tanh layer.

This chain-like structure is also present in LSTMs, albeit the repeating module is structured differently. There are four neural network layers instead of just one, and they interact in a unique way.

3.3 Transformer: -

BERT:

BERT model was proposed in BERT: - Pre-training of Deep Bidirectional Transformers for Language Understanding by Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. It's a bidirectional transformer pre-trained using a combination of masked language modelling objective and next sentence prediction on a large corpus comprising the Toronto Book Corpus and Wikipedia.

The abstract from the paper is the following:

We describe BERT, an acronym for Bidirectional Encoder Representations from Transformers, a novel approach to language representation. BERT, in contrast to other language representation models, is intended to jointly train on both left and right context in all layers in order to pre-train deep bidirectional representations from unlabelled text. Therefore, without requiring significant task-specific architectural adjustments, the pre-trained BERT model may be refined with just one more output layer to provide state-of-the-art models for a variety of tasks, including question answering and language inference. BERT is both powerful experimentally and theoretically.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement).

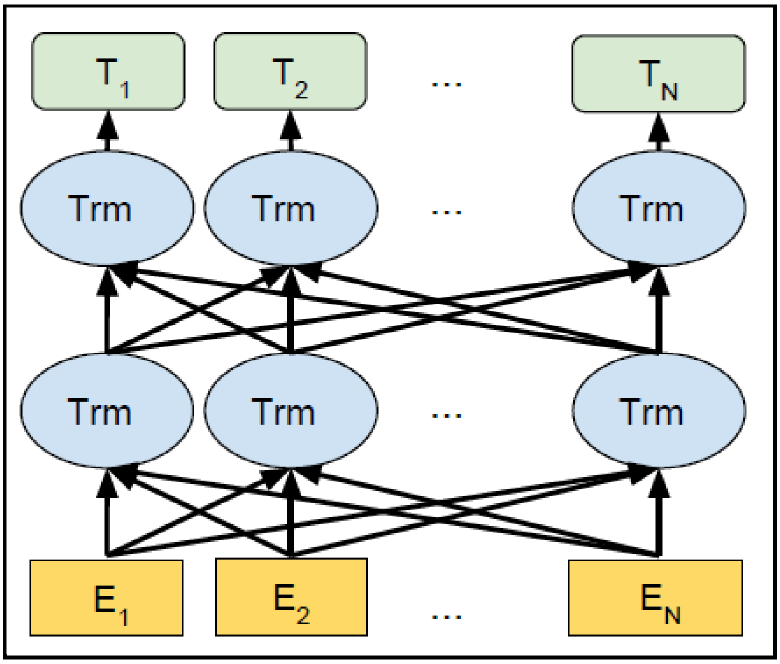


Fig.4: BERT model Pre-training structure diagram.

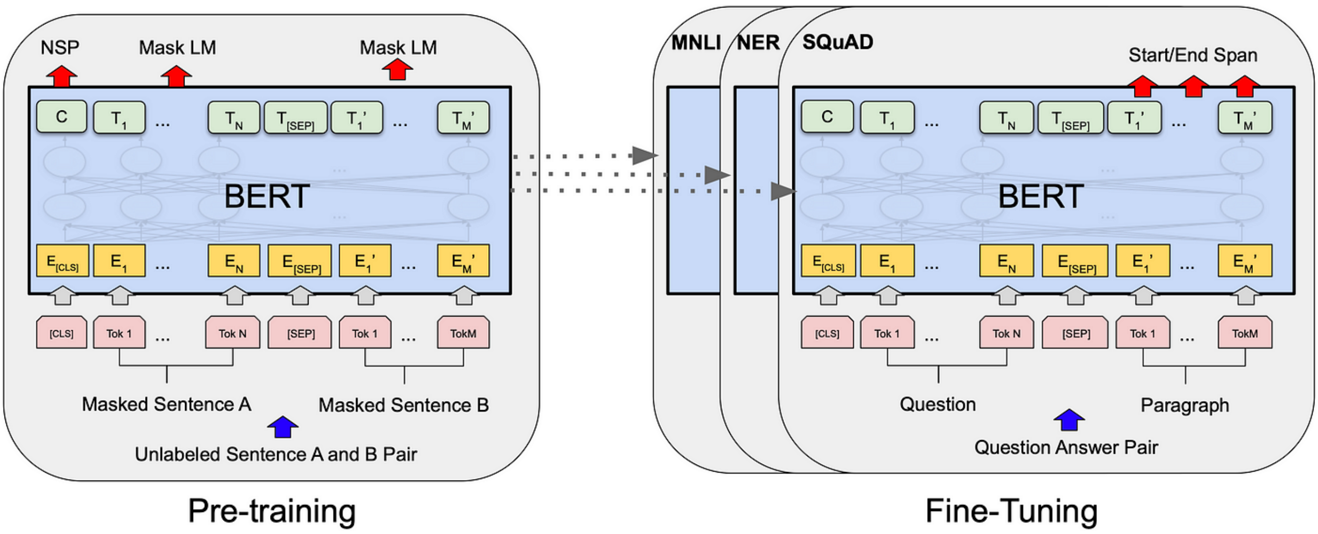


Fig.5: Two steps of BERT Model

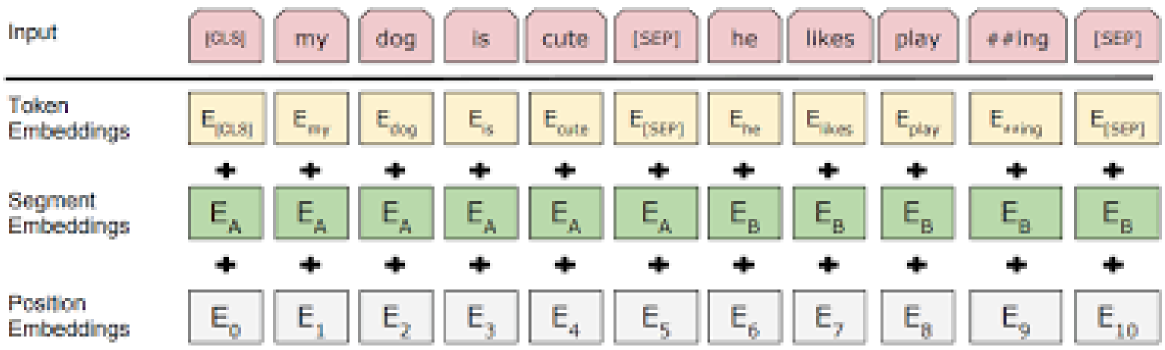


Fig.6: Data going through NLP process

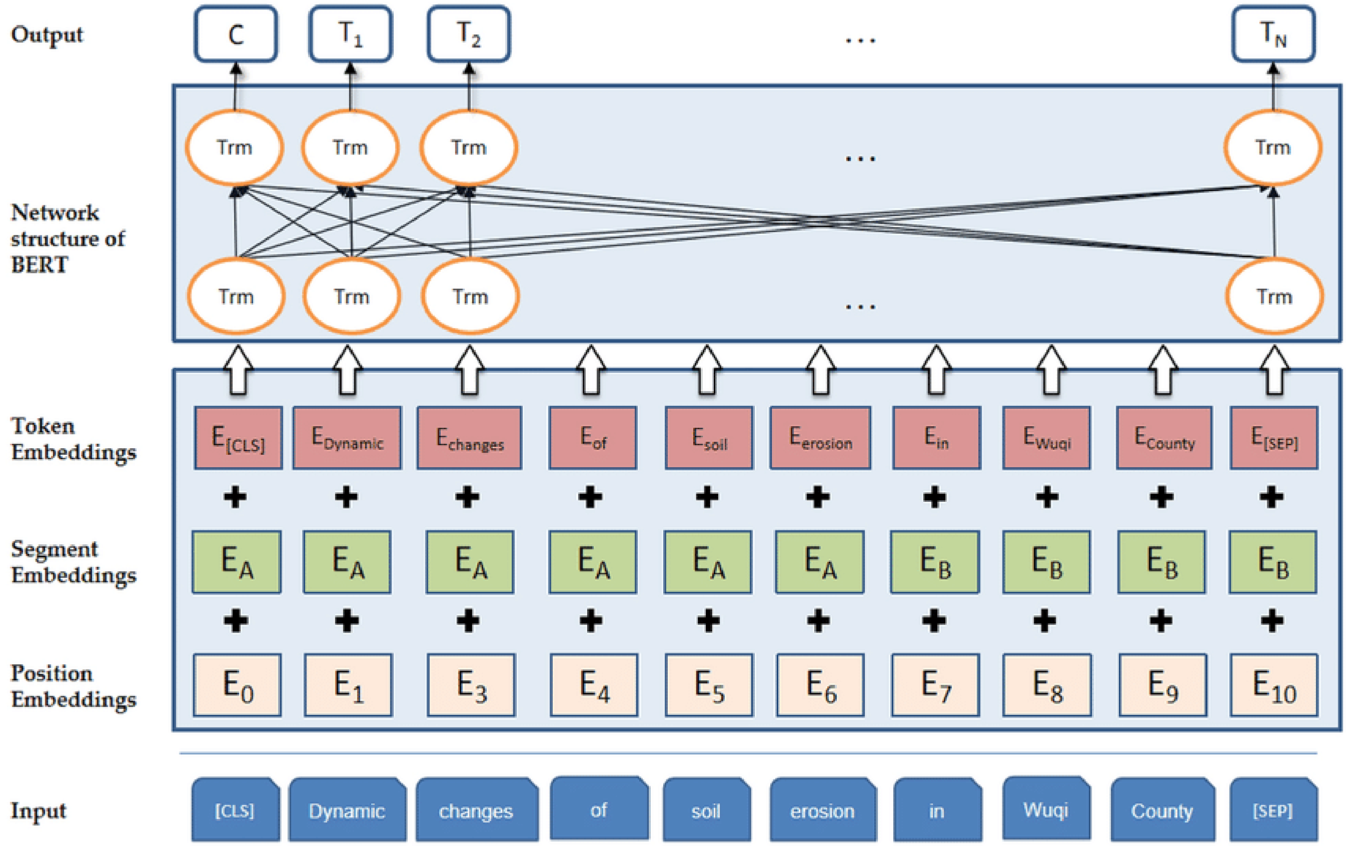


Fig.7: Whole process of BERT model

RoBERTA:

In 2018 Google AI released a self-supervised learning model called BERT for learning language representations. And then, in 2019, Yinhan Liu et al. (Meta AI) proposed a robustly optimized approach called Roberta (Robustly Optimized BERT-Pretraining Approach) for pre-training natural language processing (NLP) systems that improve on Bidirectional Encoder Representations from Transformers (BERT).

What Prompted the Researchers to Develop a Roberta - like Model?

The Facebook AI and the University of Washington researchers found that the BERT model was remarkably undertrained, and they suggested making several changes to the pre-training process to improve the BERT model's performance.

Roberta Model Architecture: -

The BERT model and the Roberta model have the same architecture. It is a reimplementation of BERT with a few small embedding fixes and adjustments to the important hyper parameters.

The graphic below illustrates the general pre-training and fine-tuning operations for the BERT. Pre-training and fine-tuning in BERT employ the identical designs, with the exception of the output layers. Models are initialized for various downstream tasks using the same pre-trained model parameters. Fine-tuning involves adjusting every parameter.

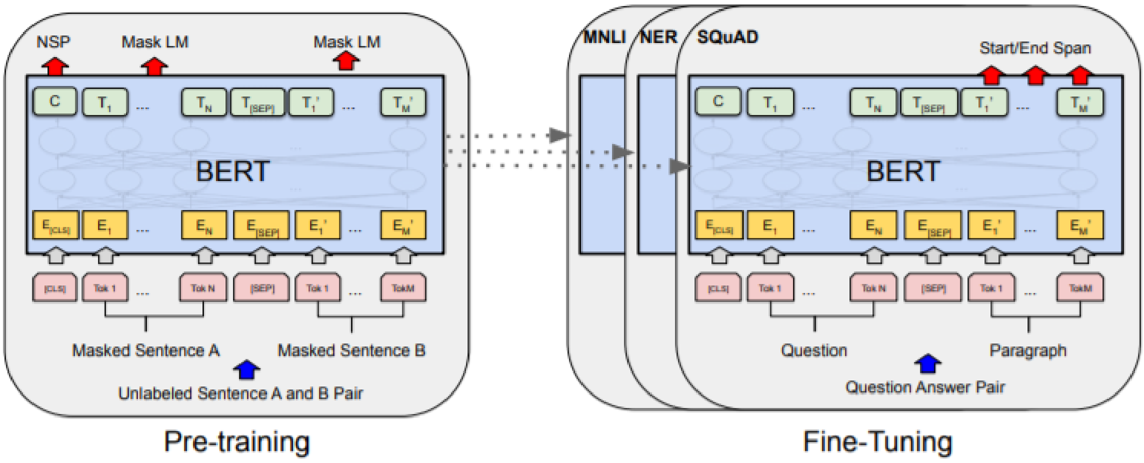


Fig.5: Architecture of BERT model

However, the Roberta model is trained with substantially bigger mini-batches and learning rates, and it does not employ the next-sentence pre-training goal. Furthermore, Roberta (like GPT-2) replaces the byte-level BPE tokenizer with a character-level BPE vocabulary and employs a different pre-training strategy. It also lacks token_type_ids, so we don't need to specify which token belongs to which segment. We may split the segments simply with the help of the separation token tokenizer.sep_token. Furthermore, Roberta is trained on an enormous dataset spanning over 160GB of uncompressed text, as opposed to the 16GB dataset that was first used to train BERT. The English Wikipedia and Books Corpus (16GB) used in BERT are included in the dataset for Roberta.

Additional data included the Web text corpus (38 GB), CommonCrawl News dataset (63 million articles, 76 GB), and Stories from Common Crawl (31 GB). This dataset, along with 1024 running V100 Tesla GPUs for a day, was used to pre-train Roberta.

To create Roberta, the Facebook team first ported BERT from Google's TensorFlow , PyTorch.

Roberta is trained with: -

- FULL-SENTENCES without NSP loss,
- Dynamic masking,
- Large mini-batches, and
- a larger byte-level BPE.

Chapter 4: Results and Analysis

Algorithm	Accuracy	Dataset Size
SVM	85.47	1,62,981
Naive Bayes	74.25	1,62,981
LSTM (20 epochs)	83.71	14,000
BERT (3 epochs)	97.61	1,62,981
ROBERTA (3 epochs)	94.19	1,62,981

4.1 Result and Analysis of Machine Learning approach: -

Dataset: The dataset is based on the tweet of Indian Prime Minister Narendra Modi. By using this dataset, we will identify the sentiment of common people. Here the number of the tweet is 1,62,981. The dataset has three sentiments namely, negative (-1), neutral (0), and positive (+1). It contains two fields for the tweet and label.

Link of the dataset:

https://www.kaggle.com/datasets/saurabhshahane/twitter-sentiment-dataset/data

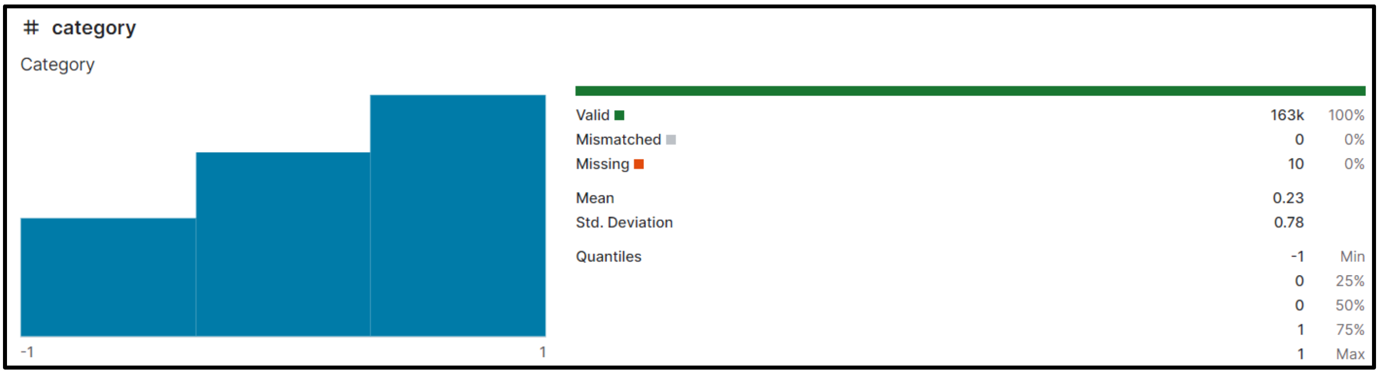


Fig.8: Sentiment Analysis graph of dataset(Twitter_Data.csv)

1. At first we have to read the dataset.

```
[1]: import io

     import pandas as pd
     import matplotlib.pyplot as plt
     import numpy as np
     data = pd.read_csv('Twitter_Data.csv')
```

2. Then we will view the dataset.

```
[2]: data.head()
```

[2]:

	clean_text	category
0	when modi promised “minimum government maximum...	-1.0
1	talk all the nonsense and continue all the dra...	0.0
2	what did just say vote for modi welcome bjp t...	1.0
3	asking his supporters prefix chowkidar their n...	1.0
4	answer who among these the most powerful world...	1.0

```
[3]: data.shape
```

```
[3]: (162980, 2)
```

3. Data Pre-processing: -

Because it is a real dataset, there are lots of null value present in our dataset. So at very beginning we have to get rid of null values.

```
[4]: data.category.unique()

[4]: array([-1.,  0.,  1., nan])

[5]: data.isna().sum()

[5]: clean_text    4
     category      7
     dtype: int64

[6]: data[data['category'].isna()]
```

[6]:

	clean_text	category
130448	the foundation stone northeast gas grid inaugu...	NaN
155642	dear terrorists you can run but you cant hide ...	NaN
155698	offense the best defence with mission shakti m...	NaN
155770	have always heard politicians backing out thei...	NaN
158693	modi government plans felicitate the faceless ...	NaN
159442	chidambaram gives praises modinomics	NaN
160559	the reason why modi contested from seats 2014 ...	NaN

[7]:
```
data[data['clean_text'].isna()]
```

[7]:

	clean_text	category
148	NaN	0.0
158694	NaN	-1.0
159443	NaN	0.0
160560	NaN	1.0

[8]:
```
data.drop(data[data['clean_text'].isna()].index, inplace=True)
data.drop(data[data['category'].isna()].index, inplace=True)
```

[9]:
```
sentiment_map={-1:'negative',1:'positive',0:'neutral'}
data.insert(2,'sentiment',[sentiment_map[s] for s in data.category],True)
#data['sentiment_int']=[sentiment_map[s] for s in data.sentiment]

data.head()
```

[9]:

	clean_text	category	sentiment
0	when modi promised “minimum government maximum...	-1.0	negative
1	talk all the nonsense and continue all the dra...	0.0	neutral
2	what did just say vote for modi welcome bjp t...	1.0	positive
3	asking his supporters prefix chowkidar their n...	1.0	positive
4	answer who among these the most powerful world...	1.0	positive

[10]:
```
#labeling
reviews = np.array(data['clean_text'])[:]
labels = np.array(data['sentiment'])[:]
```

[11]:
```
from collections import Counter

Counter(labels)
```

[11]:
```
Counter({'positive': 72249, 'neutral': 55211, 'negative': 35509})
```

4. Pre-processing in Tweet:

Here we remove all the special characters (@, #, $, etc.), punctuations, and URL from all the esteemed tweets. Next, we have given the token ids to each of the words in the tweet. Then we vectorize all the tokens.

```
[13]: from sklearn.feature_extraction.text import CountVectorizer
      from nltk.tokenize import RegexpTokenizer
      import csv


      def preProcessor(Tweet):
          import re
          from string import punctuation
          text=re.sub(r'(http|ftp|https):\/\/([\w\-_]+(?:(?:\.[\w\-_]+)+))([\w\-\.,@?^=%&:/~\+#]*[\w\-\@?^=%&/~\+#])?', ' ', Tweet)
          text=re.sub(r'['+punctuation+']',' ',Tweet)
          text=re.sub(r'#(\w+)',' ',Tweet)
          text=re.sub(r'@(\w+)',' ',Tweet)
          #print(token.tokenize(text))
          return Tweet

      token=RegexpTokenizer(r'\w+')
      cv=CountVectorizer(lowercase=True,preprocessor=preProcessor,stop_words='english',ngram_range=(1,1),tokenizer=token.tokenize)
      #text_counts=cv.fit_transform(data['Tweet'])
      text_counts=cv.fit_transform(data['clean_text'].values.astype('U'))
```

5. Splitting of Dataset:

Here we split the dataset in test and train part. We use train dataset for training purposes and test dataset for checking the accuracy of our model.

```
[14]: from sklearn.model_selection import train_test_split
      # x_train, x_test, y_train, y_test = train_test_split(text_counts,data['sentiment'],test_size=0.3)
      x_train, x_test, y_train, y_test = train_test_split(text_counts,data['sentiment'],test_size=0.3)
```

```
[15]: #Ber_NB
      from sklearn.naive_bayes import *
      from sklearn import metrics

      clf=BernoulliNB()
      clf.fit(x_train,y_train)
      clf.fit(x_train,y_train)
      pred=clf.predict(x_test)
      metrics.accuracy_score(y_test, pred)

[15]: 0.7424679388844573
```

= Naïve Bayes

```
[16]: from sklearn import svm
      clf = svm.LinearSVC()
      clf.fit(x_train,y_train)
      pred=clf.predict(x_test)
      metrics.accuracy_score(y_test, pred)

      D:\Users\janan\anaconda3\Lib\site-pac
      iblinear failed to converge, increase
        warnings.warn(
[16]: 0.8550858031130474
```

= SVM

Results: -

After all the training and testing, we get that the accuracies of sentiment analysis are **74.25%(Naive Bayes) and 85.47%(SVM).** We can see from the result that the accuracy of SVM is better than Naive Bayes.

Analysis of Graph: -

We can get the better analysis from this confusion matrix. In this matrix if the diagonal elements are higher, the model will be more efficient.

```
Confusion matrix, without normalization
[[ 8347   895  1389]
 [  521 15127   932]
 [ 1316  2035 18329]]
Normalized confusion matrix
[[0.79 0.08 0.13]
 [0.03 0.91 0.06]
 [0.06 0.09 0.85]]
```

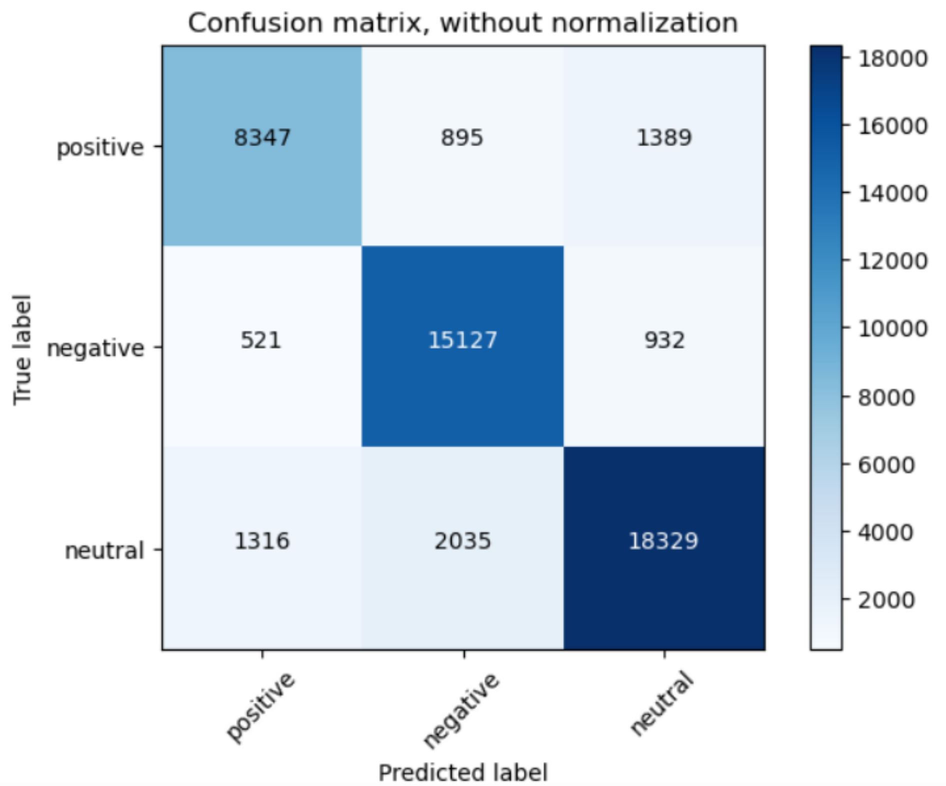


Fig.9: Confusion Matrix, without Normalization

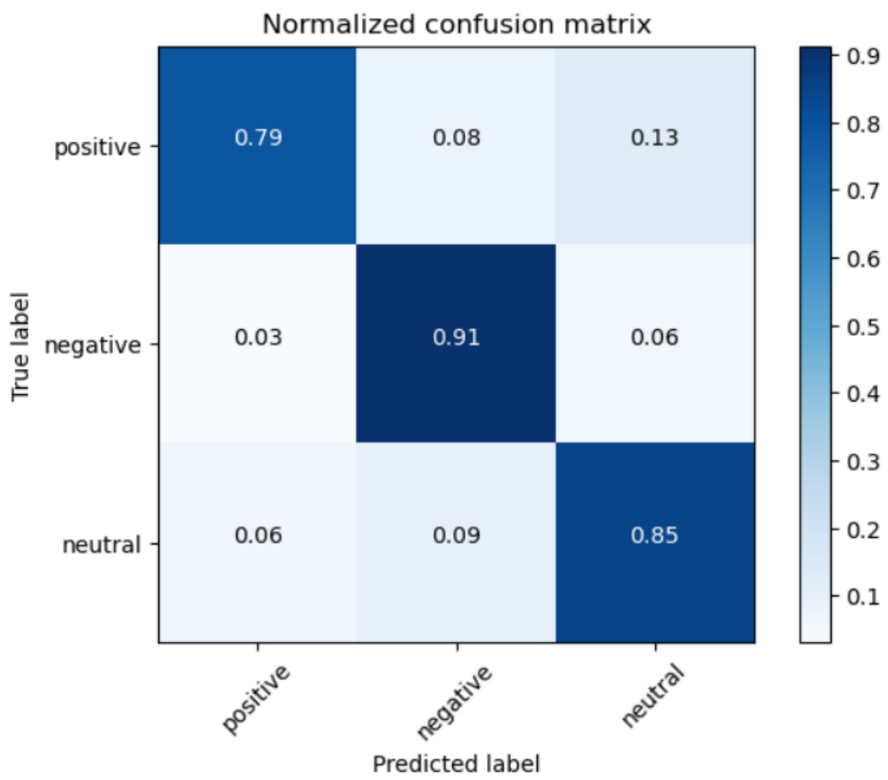


Fig.10: Normalized Confusion Matrix

4.2 Result and Analysis of Deep Learning approach: -

Dataset: A sentiment analysis job about the problems of each major U.S. airline. Twitter data was scraped from February of 2015 and contributors were asked to first classify positive, negative, and neutral tweets, followed by categorizing negative reasons (such as "late flight" or "rude service").

Link of the dataset :

https://www.kaggle.com/datasets/crowdflower/twitter-airline-sentiment

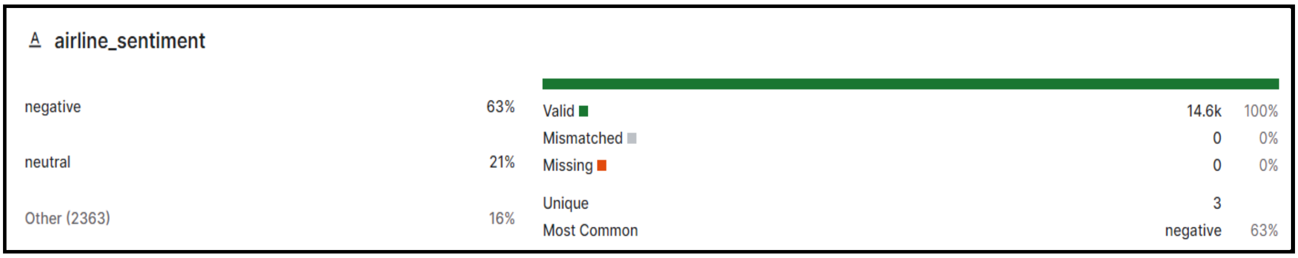


Fig.11: Sentiment Analysis Graph of dataset(Tweets.csv)

1. Load in and visualize the data :

[1]:
```
import numpy as np
import pandas as pd

data = pd.read_csv('Tweets.csv')
data.head()
```

[1]:

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativereason_confidence	airline	airline_sentiment_gold	name	negative
0	570306133677760513	neutral	1.0000	NaN	NaN	Virgin America	NaN	cairdin	
1	570301130888122368	positive	0.3486	NaN	0.0000	Virgin America	NaN	jnardino	
2	570301083672813571	neutral	0.6837	NaN	NaN	Virgin America	NaN	yvonnalynn	
3	570301031407624196	negative	1.0000	Bad Flight	0.7033	Virgin America	NaN	jnardino	
4	570300817074462722	negative	1.0000	Can't Tell	1.0000	Virgin America	NaN	jnardino	

```
[2]: reviews = np.array(data['text'])[:14000]
     labels = np.array(data['airline_sentiment'])[:14000]
```

```
[3]: data['text'].loc[14639]
```

```
[3]: '@AmericanAir we have 8 ppl so we need 2 know how many seats ar
     t us on standby for 4 people on the next flight?'
```

```
[4]: data['airline_sentiment'].loc[14639]
```

```
[4]: 'neutral'
```

```
[5]: from collections import Counter

     Counter(labels)
```

```
[5]: Counter({'negative': 8679, 'neutral': 3017, 'positive': 2304})
```

2. Data pre-processing:

The first step when building a neural network model is getting the data into the proper form to feed into the network. Since we're using embedding layers, we'll need to encode each word with an integer. We'll also want to clean it up a bit.

Here are the processing steps, we'll want to take:

We'll want to get rid of periods and extraneous punctuation. We'll want to remove web address, twitter id, and digit. First, let's remove all punctuation. Then get all the text without the newlines and split it into individual words.

```python
punctuation = '!"#$%&\'()*+,-./:;<=>?[\\]^_`{|}~'

# get rid of punctuation
all_reviews = 'separator'.join(reviews)
all_reviews = all_reviews.lower()
all_text = ''.join([c for c in all_reviews if c not in punctuation])

# split by new lines and spaces
reviews_split = all_text.split('separator')
all_text = ' '.join(reviews_split)

# create a list of words
words = all_text.split()
```

```python
# get rid of web address, twitter id, and digit
new_reviews = []
for review in reviews_split:
    review = review.split()
    new_text = []
    for word in review:
        if (word[0] != '@') & ('http' not in word) & (~word.isdigit()):
            new_text.append(word)
    new_reviews.append(new_text)
```

3. Encoding the words:

The embedding lookup requires that we pass in integers to our network. The easiest way to do this is to create dictionaries that map the words in the vocabulary to integers. Then, we can convert each of our reviews into integers so they can be passed into the network.

```
[8]: ## Build a dictionary that maps words to integers
counts = Counter(words)
vocab = sorted(counts, key=counts.get, reverse=True)
vocab_to_int = {word: ii for ii, word in enumerate(vocab, 1)}

## use the dict to tokenize each review in reviews_split
## store the tokenized reviews in reviews_ints
reviews_ints = []
for review in new_reviews:
    reviews_ints.append([vocab_to_int[word] for word in review])
```

```
[9]: # stats about vocabulary
print('Unique words: ', len((vocab_to_int)))  # should ~ 74000+
print()

# print tokens in first review
print('Tokenized review: \n', reviews_ints[:1])
```

```
Unique words:  16727

Tokenized review: 
 [[57, 213]]
```

4. Encoding the labels:

As mentioned before, our goal is to identify whether a tweet is negative or non-negative (positive or neutral). Our labels are "positive", "negative", or "neutral. To use these labels in our network, we need to convert them to 0 and 1.

```
[10]: # 2=positive, 1=neutral, 0=negative label conversion
encoded_labels = []
for label in labels:
    if label == 'neutral':
        encoded_labels.append(1)
    elif label == 'negative':
        encoded_labels.append(0)
    else:
        encoded_labels.append(1)

encoded_labels = np.asarray(encoded_labels)
```

```
[11]: encoded_labels
```

```
[11]: array([1, 1, 1, ..., 1, 0, 0])
```

5. Padding sequences:

To deal with both short and very long reviews, we'll pad or truncate all our reviews to a specific length. For reviews shorter than some seq_length, we'll pad with 0s.

For reviews longer than seq_length, we can truncate them to the first seq_length words. A good seq_length, in this case, is 30, because the maximum review length from the data is 32.

[12]:
```
def pad_features(reviews_ints, seq_length):
    ''' Return features of review_ints, where each review is padded with 0's
        or truncated to the input seq_length.
    '''

    # getting the correct rows x cols shape
    features = np.zeros((len(reviews_ints), seq_length), dtype=int)

    # for each review, I grab that review and
    for i, row in enumerate(reviews_ints):
        features[i, -len(row):] = np.array(row)[:seq_length]

    return features
```

[13]:
```
# Test implementation!

seq_length = 30

features = pad_features(reviews_ints, seq_length=seq_length)

## test statements
assert len(features)==len(reviews_ints), "The features should |
assert len(features[0])==seq_length, "Each feature row should 

# print first 10 values of the first 30 batches
print(features[:10,:10])
```

```
[[  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 446]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0]]
```

6. Training, validation, and test:

```
[14]:
split_frac = 0.8

## split data into training, validation, and test data (features and labels,

split_idx = int(len(features)*split_frac)
train_x, remaining_x = features[:split_idx], features[split_idx:]
train_y, remaining_y = encoded_labels[:split_idx], encoded_labels[split_idx:]

test_idx = int(len(remaining_x)*0.5)
val_x, test_x = remaining_x[:test_idx], remaining_x[test_idx:]
val_y, test_y = remaining_y[:test_idx], remaining_y[test_idx:]

## print out the shapes of the resultant feature data
print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
      "\nValidation set: \t{}".format(val_x.shape),
      "\nTest set: \t\t{}".format(test_x.shape))
```

```
                        Feature Shapes:
Train set:              (11200, 30)
Validation set:         (1400, 30)
Test set:               (1400, 30)
```

7. DataLoaders and Batching:

After creating training, test, and validation data, we can create DataLoaders for this data by following two steps:

Create a known format for accessing our data, using TensorDataset which takes in an input set of data and a target set of data with the same first dimension, and creates a dataset.
Create DataLoaders and batch our training, validation, and test Tensor datasets.

[15]:
```python
import torch
from torch.utils.data import TensorDataset, DataLoader

# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
valid_data = TensorDataset(torch.from_numpy(val_x), torch.from_numpy(val_y))
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))

# dataloaders
batch_size = 50

# make sure the SHUFFLE the training data
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

[16]:
```python
# obtain one batch of training data

dataiter = iter(train_loader)
sample_x, sample_y = dataiter.__next__()

print('Sample input size: ', sample_x.size()) # batch_size, seq_length
print('Sample input: \n', sample_x)
print()
print('Sample label size: ', sample_y.size()) # batch_size
print('Sample label: \n', sample_y)
```

```
Sample input size:  torch.Size([50, 30])
Sample input: 
 tensor([[    0,     0,     0,  ...,     9,  2990,  1043],
        [    0,     0,     0,  ...,    95,   249,  1467],
        [    0,     0,     0,  ...,   105,    11,  1104],
        ...,
        [    0,     0,     0,  ...,     9,    11,     8],
        [    0,     0,     0,  ...,    23,     2,   319],
        [    0,     0,     0,  ...,   243, 13906, 13907]], dtype=torch.int32)

Sample label size:  torch.Size([50])
Sample label: 
 tensor([1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1,
        1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0,
        0, 1], dtype=torch.int32)
```

[63]:
```python
# First checking if GPU is available
train_on_gpu=torch.cuda.is_available()

if(train_on_gpu):
    print('Training on GPU.')
else:
    print('No GPU available, training on CPU.')
```

```
Training on GPU.
```

8. Sentiment Network with PyTorch:

[18]:
```python
import torch.nn as nn

class SentimentRNN(nn.Module):
    """
    The RNN model that will be used to perform Sentiment analysis.
    """

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, drop_p
        """
        Initialize the model by setting up the layers.
        """
        super(SentimentRNN, self).__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()
```

```python
    def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hidden state.
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        x = x.long()
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)
        # sigmoid function
        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden
```

```python
def init_hidden(self, batch_size):
    ''' Initializes hidden state '''
    # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
    # initialized to zero, for hidden state and cell state of LSTM
    weight = next(self.parameters()).data

    if (train_on_gpu):
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda()
              weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda())
    else:
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())

    return hidden
```

[19]:

```
# Instantiate the model w/ hyperparams
vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding + our word tokens
output_size = 1
embedding_dim = 200
hidden_dim = 128
n_layers = 2

net = SentimentRNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)

print(net)
```

```
SentimentRNN(
  (embedding): Embedding(16728, 200)
  (lstm): LSTM(200, 128, num_layers=2, batch_first=True, dropout=0.5)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc): Linear(in_features=128, out_features=1, bias=True)
  (sig): Sigmoid()
)
```

[66]:

```
# loss and optimization functions
lr=0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
```

[67]:

```
# training params

epochs = 20

counter = 0
print_every = 100
clip=5 # gradient clipping

# move model to GPU, if available
if(train_on_gpu):
    net.cuda()

net.train()
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    for inputs, labels in train_loader:
        counter += 1

        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()
```

```
# Creating new variables for the hidden state, otherwise
# we'd backprop through the entire training history
h = tuple([each.data for each in h])

# zero accumulated gradients
net.zero_grad()

# get the output from the model
output, h = net(inputs, h)

# calculate the loss and perform backprop
loss = criterion(output.squeeze(), labels.float())
loss.backward()
# `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
nn.utils.clip_grad_norm_(net.parameters(), clip)
optimizer.step()

# loss stats
if counter % print_every == 0:
    # Get validation loss
    val_h = net.init_hidden(batch_size)
    val_losses = []
    net.eval()
    for inputs, labels in valid_loader:

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        val_h = tuple([each.data for each in val_h])

        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()
```

```
            output, val_h = net(inputs, val_h)
            val_loss = criterion(output.squeeze(), labels.float())

            val_losses.append(val_loss.item())

        net.train()
        print("Epoch: {}/{}...".format(e+1, epochs),
              "Step: {}...".format(counter),
              "Loss: {:.6f}...".format(loss.item()),
              "Val Loss: {:.6f}".format(np.mean(val_losses)))
```

```
Epoch: 1/20... Step: 100... Loss: 0.536182... Val Loss: 0.494789
Epoch: 1/20... Step: 200... Loss: 0.342242... Val Loss: 0.481267
Epoch: 2/20... Step: 300... Loss: 0.424031... Val Loss: 0.430037
Epoch: 2/20... Step: 400... Loss: 0.384714... Val Loss: 0.436275
Epoch: 3/20... Step: 500... Loss: 0.344898... Val Loss: 0.512167
Epoch: 3/20... Step: 600... Loss: 0.211354... Val Loss: 0.443102
Epoch: 4/20... Step: 700... Loss: 0.218554... Val Loss: 0.540556
Epoch: 4/20... Step: 800... Loss: 0.282090... Val Loss: 0.538329
Epoch: 5/20... Step: 900... Loss: 0.116927... Val Loss: 0.568285
Epoch: 5/20... Step: 1000... Loss: 0.091082... Val Loss: 0.586823
Epoch: 5/20... Step: 1100... Loss: 0.064183... Val Loss: 0.673234
Epoch: 6/20... Step: 1200... Loss: 0.100836... Val Loss: 0.820781
Epoch: 6/20... Step: 1300... Loss: 0.029550... Val Loss: 0.851291
Epoch: 7/20... Step: 1400... Loss: 0.007672... Val Loss: 0.840712
Epoch: 7/20... Step: 1500... Loss: 0.017947... Val Loss: 0.887218
Epoch: 8/20... Step: 1600... Loss: 0.064415... Val Loss: 0.852542
Epoch: 8/20... Step: 1700... Loss: 0.005255... Val Loss: 0.983539
Epoch: 9/20... Step: 1800... Loss: 0.013786... Val Loss: 0.926213
Epoch: 9/20... Step: 1900... Loss: 0.010355... Val Loss: 0.969622
Epoch: 9/20... Step: 2000... Loss: 0.037333... Val Loss: 0.965913
Epoch: 10/20... Step: 2100... Loss: 0.006790... Val Loss: 1.071350
Epoch: 10/20... Step: 2200... Loss: 0.041043... Val Loss: 0.992211
Epoch: 11/20... Step: 2300... Loss: 0.012228... Val Loss: 0.994055
Epoch: 11/20... Step: 2400... Loss: 0.006787... Val Loss: 1.101592
Epoch: 12/20... Step: 2500... Loss: 0.003977... Val Loss: 1.040750
Epoch: 12/20... Step: 2600... Loss: 0.051130... Val Loss: 1.006338
Epoch: 13/20... Step: 2700... Loss: 0.023056... Val Loss: 1.045345
Epoch: 13/20... Step: 2800... Loss: 0.000416... Val Loss: 1.193788
Epoch: 13/20... Step: 2900... Loss: 0.005105... Val Loss: 1.229952
Epoch: 14/20... Step: 3000... Loss: 0.003819... Val Loss: 1.195661
Epoch: 14/20... Step: 3100... Loss: 0.038420... Val Loss: 1.150890
Epoch: 15/20... Step: 3200... Loss: 0.001121... Val Loss: 1.111401
Epoch: 15/20... Step: 3300... Loss: 0.004511... Val Loss: 1.114257
Epoch: 16/20... Step: 3400... Loss: 0.002109... Val Loss: 1.138698
Epoch: 16/20... Step: 3500... Loss: 0.001940... Val Loss: 1.292685
Epoch: 17/20... Step: 3600... Loss: 0.015803... Val Loss: 1.122164
Epoch: 17/20... Step: 3700... Loss: 0.001603... Val Loss: 1.134488
Epoch: 17/20... Step: 3800... Loss: 0.173662... Val Loss: 1.122638
Epoch: 18/20... Step: 3900... Loss: 0.016160... Val Loss: 1.115329
Epoch: 18/20... Step: 4000... Loss: 0.003738... Val Loss: 1.257620
Epoch: 19/20... Step: 4100... Loss: 0.001353... Val Loss: 1.307187
Epoch: 19/20... Step: 4200... Loss: 0.008524... Val Loss: 1.519380
Epoch: 20/20... Step: 4300... Loss: 0.004698... Val Loss: 1.343439
Epoch: 20/20... Step: 4400... Loss: 0.018503... Val Loss: 1.413271
```

```python
•[68]: # Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0

# init hidden state
h = net.init_hidden(batch_size)

net.eval()
# iterate over test data
for inputs, labels in test_loader:

    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])

    if(train_on_gpu):
        inputs, labels = inputs.cuda(), labels.cuda()

    # get predicted outputs
    output, h = net(inputs, h)

    # calculate loss
    test_loss = criterion(output.squeeze(), labels.float())
    test_losses.append(test_loss.item())

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())  # rounds to the nearest integer

    # compare predictions to true label
    correct_tensor = pred.eq(labels.float().view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(correct_tensor.cpu().numpy())
    num_correct += np.sum(correct)

# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}".format(test_acc))
```

```
Test loss: 1.150
Test accuracy: 0.837
```

= LSTM

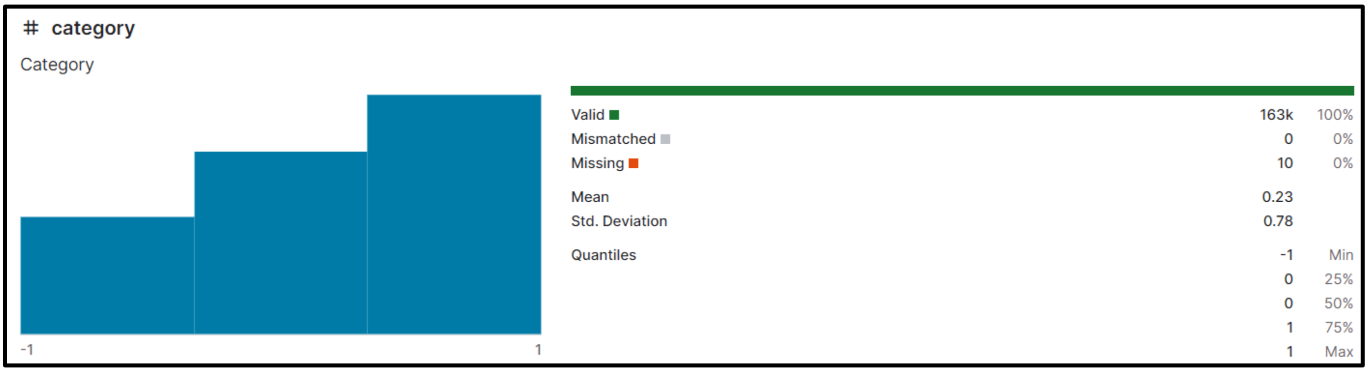
Results: -

After all the training and testing, we get that the accuracies of sentiment analysis are **83.71%(LSTM)**.

4.3 Result and Analysis of Transformers approach: -

Dataset: The dataset is based on the tweet of Indian Prime Minister Narendra Modi. By using this dataset, we will identify the sentiment of common people. Here the number of the tweet is 1,62,981. The dataset has three sentiments namely, negative (-1), neutral (0), and positive (+1). It contains two fields for the tweet and label.

Link of the dataset: -
https://www.kaggle.com/datasets/saurabhshahane/twitter-sentiment-dataset/data



BERT and ROBERTA - Twitter Sentiment Classifiers

**The code of both the classifier are almost the same.

1. Import the required library which helps both the Classifiers

[2]:
```
import torch
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
import torch.nn.functional as F
from transformers import BertTokenizer, BertConfig,AdamW, BertForSequenceClassification,get_linear_schedule_with_warm


import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,classification_report
# Import and evaluate each test batch using Matthew's correlation coefficient
from sklearn.metrics import accuracy_score,matthews_corrcoef

from tqdm import tqdm, trange,tnrange,tqdm_notebook
import random
import os
import io
% matplotlib inline
```

2. Identify and specify the GPU as the device, later in training loop we will load data into device.

```
[4]: # identify and specify the GPU as the device, later in training loop we will load data into device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
n_gpu = torch.cuda.device_count()
torch.cuda.get_device_name(0)

SEED = 19

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
if device == torch.device("cuda"):
    torch.cuda.manual_seed_all(SEED)
```

3. Read File

```
df_train = pd.read_csv("/kaggle/input/twitter-sentiment-dataset/Twitter_Data.csv")
```

4. Checking Null Value

```
df_train.isnull().sum()
```

```
df_train.head()
```

5. Target Distribution

```
df_train['category'].unique()
```

```
df_train['category'].value_counts()
```

6. Removing Null value

```
df_train = df_train[~df_train['category'].isnull()]
```

```
df_train = df_train[~df_train['clean_text'].isnull()]
```

7. Target Encoding

```
from sklearn.preprocessing import LabelEncoder
labelencoder = LabelEncoder()
df_train['category_1'] = labelencoder.fit_transform(df_train['category'])
```

```
df_train[['category','category_1']].drop_duplicates(keep='first')
```

```
df_train.rename(columns={'category_1':'label'},inplace=True)
```

8. Data Preparation for BERT and Roberta Model

▪

```
## create label and sentence list
sentences = df_train.clean_text.values

#check distribution of data based on labels
print("Distribution of data based on labels: ",df_train.label.value_counts())
MAX_LEN = 256

## Import ROBERTA tokenizer, that is used to convert our text into tokens that corres
tokenizer = RobertaTokenizer.from_pretrained('roberta-base',do_lower_case=True)
```

```
## create label and sentence list
sentences = df_train.clean_text.values

#check distribution of data based on labels
print("Distribution of data based on labels: ",df_train.label.value_counts())
MAX_LEN = 256

## Import BERT tokenizer, that is used to convert our text into tokens that corresp
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',do_lower_case=True)
```

```
input_ids = [tokenizer.encode(sent, add_special_tokens=True,
                              max_length=MAX_LEN,
                              pad_to_max_length=True,truncation=True) for sent in sentences]
```

[17]:

```python
labels = df_train.label.values

print("Actual sentence before tokenization: ",sentences[2])
print("Encoded Input from dataset: ",input_ids[2])

## Create attention mask
attention_masks = []
## Create a mask of 1 for all input tokens and 0 for all padding tokens
attention_masks = [[float(i>0) for i in seq] for seq in input_ids]
print(attention_masks[2])
```

```
Actual sentence before tokenization:  what did just say vote for modi  welcome bjp told you rahul the main campaigner
for modi think modi should just relax
Encoded Input from dataset:  [101, 2054, 2106, 2074, 2360, 3789, 2005, 16913, 2072, 6160, 24954, 2409, 2017, 10958, 2
1886, 1996, 2364, 3049, 2121, 2005, 16913, 2072, 2228, 16913, 2072, 2323, 2074, 9483, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

[18]:

```python
train_inputs,validation_inputs,train_labels,validation_labels = train_test_split(input_ids,labels,random_state=41,tes
train_masks,validation_masks,_,_ = train_test_split(attention_masks,input_ids,random_state=41,test_size=0.1)
```

[19]:

```python
# convert all our data into torch tensors, required data type for our model
train_inputs = torch.tensor(train_inputs)
validation_inputs = torch.tensor(validation_inputs)
train_labels = torch.tensor(train_labels)
validation_labels = torch.tensor(validation_labels)
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)
```

```python
# Select a batch size for training. For fine-tuning BERT on a specific task, the authors recommend a batch size of 1
batch_size = 32

# Create an iterator of our data with torch DataLoader. This helps save on memory during training because, unlike a
# with an iterator the entire dataset does not need to be loaded into memory
train_data = TensorDataset(train_inputs,train_masks,train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data,sampler=train_sampler,batch_size=batch_size)

validation_data = TensorDataset(validation_inputs,validation_masks,validation_labels)
validation_sampler = RandomSampler(validation_data)
validation_dataloader = DataLoader(validation_data,sampler=validation_sampler,batch_size=batch_size)
```

9. Let’s see how the training data looks like

```
[20]: train_data[0]

[20]: (tensor([  101, 16913,  2072,  4067,  2003,  3217,  6529,  2003,  3217,  9058,
               6950,  1998,  7558, 21665,  6505,  2634,  3352,  2686,  3565, 11452,
               3519,  4283, 23556,  2008,  2634,  2085,  2686,  3565,  2373,  4489,
                100,   102,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                  0,     0,     0,     0,     0,     0]),
       tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0.]),
       tensor(2))
```

10. Load BERT and RobertaForSequenceClassification, the pre-trained ROBERTA model with a single linear classification layer on top.

[21]:
```
# Load BertForSequenceClassification, the pretrained BERT model with a single linear classification layer on top.
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=3).to(device)

# Parameters:
lr = 2e-5
adam_epsilon = 1e-8

# Number of training epochs (authors recommend between 2 and 4)
epochs = 3

num_warmup_steps = 0
num_training_steps = len(train_dataloader)*epochs

### In Transformers, optimizer and schedules are splitted and instantiated like this:
optimizer = AdamW(model.parameters(), lr=lr,eps=adam_epsilon,correct_bias=False)  # To reproduce BertAdam specific be
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=num_warmup_steps, num_training_steps=num_trai
```

```
# Load RobertaForSequenceClassification, the pretrained ROBERTA model with a single linear classification layer on to
model = RobertaForSequenceClassification.from_pretrained('roberta-base',num_labels=3).to(device)

# Parameters:
lr = 2e-5
adam_epsilon = 1e-8

# Number of training epochs (authors recommend between 2 and 4)
epochs = 3

num_warmup_steps = 0
num_training_steps = len(train_dataloader)*epochs

### In Transformers, optimizer and schedules are splitted and instantiated like this:
optimizer = AdamW(model.parameters(), lr=lr,eps=adam_epsilon,correct_bias=False)  # To reproduce AdamW specific behav
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=num_warmup_steps, num_training_steps=num_trai
```

```
<====================== Epoch 1 ======================>

        Current Learning rate:  1.3333333333333333e-05

        Average Training loss: 0.5272794122950711

        Validation Accuracy: 0.9178921568627451

        Validation MCC Accuracy: 0.873094630026624
<====================== Epoch 2 ======================>

        Current Learning rate:  6.666666666666667e-06

        Average Training loss: 0.24999627158375715

        Validation Accuracy: 0.9471200980392157

        Validation MCC Accuracy: 0.9179650004820201
<====================== Epoch 3 ======================>

        Current Learning rate:  0.0

        Average Training loss: 0.1854583954417844

        Validation Accuracy: 0.9519539760348584

        Validation MCC Accuracy: 0.9247192878171798
```

= RoBERTa

```
<====================== Epoch 1 ======================>

        Current Learning rate:  1.3333333333333333e-05

        Average Training loss: 0.17299689453832293

        Validation Accuracy: 0.9749387254901961

        Validation MCC Accuracy: 0.9613292569719919
<====================== Epoch 2 ======================>

        Current Learning rate:  6.666666666666667e-06

        Average Training loss: 0.06296881614167849

        Validation Accuracy: 0.9831767429193901

        Validation MCC Accuracy: 0.9739491094919626
<====================== Epoch 3 ======================>

        Current Learning rate:  0.0

        Average Training loss: 0.03844679174764193

        Validation Accuracy: 0.9859068627450981

        Validation MCC Accuracy: 0.9779271882799906
```

= BERT

11. Train Loop

```
[22]: ## Store our loss and accuracy for plotting
train_loss_set = []
learning_rate = []

# Gradients gets accumulated by default
model.zero_grad()

# tnrange is a tqdm wrapper around the normal python range
for _ in tnrange(1,epochs+1,desc='Epoch'):
  print("<" + "="*22 + F" Epoch {_} "+ "="*22 + ">")
  # Calculate total loss for this epoch
  batch_loss = 0

  for step, batch in enumerate(train_dataloader):
    # Set our model to training mode (as opposed to evaluation mode)
    model.train()

    # Add batch to GPU
    batch = tuple(t.to(device) for t in batch)
    # Unpack the inputs from our dataloader
    b_input_ids, b_input_mask, b_labels = batch

    # Forward pass
    outputs = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask, labels=b_labels)
    loss = outputs[0]

    # Backward pass
    loss.backward()

    # Clip the norm of the gradients to 1.0
    # Gradient clipping is not in AdamW anymore
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    # Update parameters and take a step using the computed gradient
    optimizer.step()
```

```
  # Update learning rate schedule
  scheduler.step()

  # Clear the previous accumulated gradients
  optimizer.zero_grad()

  # Update tracking variables
  batch_loss += loss.item()

# Calculate the average loss over the training data.
avg_train_loss = batch_loss / len(train_dataloader)

#store the current learning rate
for param_group in optimizer.param_groups:
  print("\n\tCurrent Learning rate: ",param_group['lr'])
  learning_rate.append(param_group['lr'])

train_loss_set.append(avg_train_loss)
print(F'\n\tAverage Training loss: {avg_train_loss}')

# Validation

# Put model in evaluation mode to evaluate loss on the validation set
model.eval()

# Tracking variables
eval_accuracy,eval_mcc_accuracy,nb_eval_steps = 0, 0, 0

# Evaluate data for one epoch
for batch in validation_dataloader:
  # Add batch to GPU
  batch = tuple(t.to(device) for t in batch)
  # Unpack the inputs from our dataloader
  b_input_ids, b_input_mask, b_labels = batch
  # Telling the model not to compute or store gradients, saving memory and speeding up validation
  with torch.no_grad():
    # Forward pass, calculate logit predictions
    logits = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask)
```

```
    # Move logits and labels to CPU
    logits = logits[0].to('cpu').numpy()
    label_ids = b_labels.to('cpu').numpy()

    pred_flat = np.argmax(logits, axis=1).flatten()
    labels_flat = label_ids.flatten()

    df_metrics=pd.DataFrame({'Epoch':epochs,'Actual_class':labels_flat,'Predicted_class':pred_flat})

    tmp_eval_accuracy = accuracy_score(labels_flat,pred_flat)
    tmp_eval_mcc_accuracy = matthews_corrcoef(labels_flat, pred_flat)

    eval_accuracy += tmp_eval_accuracy
    eval_mcc_accuracy += tmp_eval_mcc_accuracy
    nb_eval_steps += 1

print(F'\n\tValidation Accuracy: {eval_accuracy/nb_eval_steps}')
print(F'\n\tValidation MCC Accuracy: {eval_mcc_accuracy/nb_eval_steps}')
```

[23]:

```
from sklearn.metrics import confusion_matrix,classification_report
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    import itertools
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

```
[24]: ## emotion labels
label2int = {
  "Negative": 0,
  "Neutral": 1,
  "Positive": 2
}
```

```
[25]: print(classification_report(df_metrics['Actual_class'].values, df_metrics['Predicted_class'].values,
                             target_names=label2int.keys(), digits=len(label2int)))
```

```
              precision    recall  f1-score   support

    Negative      1.000     1.000     1.000         4
     Neutral      1.000     1.000     1.000         1
    Positive      1.000     1.000     1.000         4

    accuracy                          1.000         9
   macro avg      1.000     1.000     1.000         9
weighted avg      1.000     1.000     1.000         9
```

BERT Model ending code: -

```
model_save_folder = 'model/'
tokenizer_save_folder = 'tokenizer/'

path_model = F'/kaggle/working/{model_save_folder}'
path_tokenizer = F'/kaggle/working/{tokenizer_save_folder}'

##create the dir

!mkdir -p {path_model}
!mkdir -p {path_tokenizer}

### Now let's save our model and tokenizer to a directory
model.save_pretrained(path_model)
tokenizer.save_pretrained(path_tokenizer)

model_save_name = 'fineTuneModel.pt'
path = path_model = F'/kaggle/working/{model_save_folder}/{model_save_name}'
torch.save(model.state_dict(),path);
```

RoBERTA Model ending code: -

```
confusion_matrix(df_metrics['Actual_class'].values, df_metrics['Predicted_class'].values)
```

```
plot_confusion_matrix(cm=confusion_matrix(df_metrics['Actual_class'].values, df_metrics['Predicted_class'].values),
                      classes=[0  ,1  ,2],
                          normalize=True,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues)
```

```
Normalized confusion matrix
[[0.  0.5 0.5]
 [0.  1.  0. ]
 [0.  0.  1. ]]
```

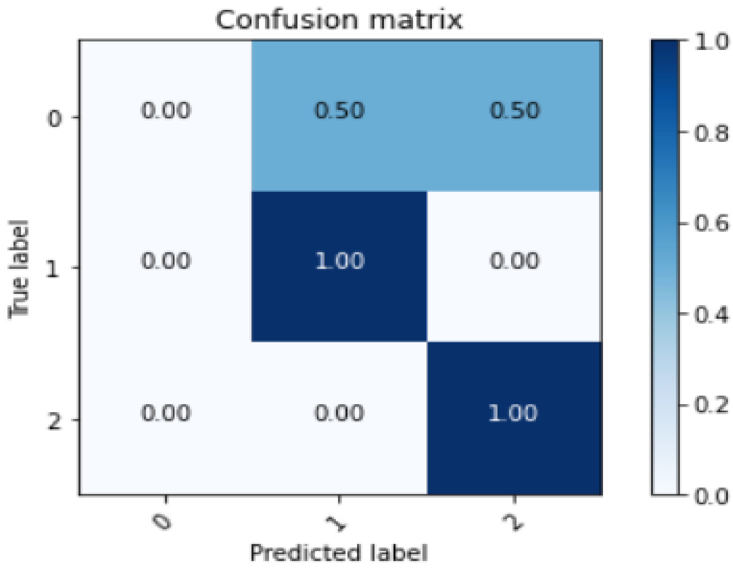


Fig.12: Confusion Matrix created by RoBERTa model

Results: -

After all the training and testing, we get that the accuracies of sentiment analysis are **97.61%(BERT) and 94.19%(RoBERTa).** We can see from the result that the accuracy of BERT is better than RoBERTa. Here Roberta model didn't work properly in case of negative tweets.

****

Overall Analysis:-

By comparing all approaches, Research comes to a conclusion that, the result of **BERT** is dependent. **RoBERTa** seems to work good but in situation of negative tweets it stops working. In the other hand, **SVM, Naïve Bayes and LSTM** is very accurate and easy to train it comparatively.

Chapter 5: Conclusions and Future Work

5.1 Conclusion

The outcome shows that SVM has higher accuracy than Naive Bayes. The confusion matrix indicates that the sentiment analysis's negative result is more prevalent than its positive and neutral outcomes. This is an illustration of sentiment analysis in action. This method may also be used to analyse the opinions of other emerging disciplines. The comprehensive pipeline for fine-tuning a BERT model for sentiment classification on Twitter data. It covers data pre-processing, model initialization, training, validation, evaluation, and model saving. The inclusion of metrics like accuracy, MCC, and visualization aids in understanding and analysing the model's performance.

Text and opinion mining include the examination of sentiment on Twitter. It focuses on examining the attitudes expressed in the tweets and putting the information into a machine learning model to train it and subsequently assess its correctness. Based on the model's performance, we may utilize this model going forward. Data collection, text pre-processing, sentiment detection, sentiment categorization, model training, and model testing are some of the phases that make up this process. Over the past ten years, this study area has changed, with models now attaining about 85%–90% efficiency. However, the aspect of data variety is still absent. In addition, there are other application problems due to the terminology and abbreviations employed. The performance of many analysers decreases as the number of classes increases. Furthermore, the model's accuracy for topics other than the one under discussion has not yet been evaluated. As a result, sentiment analysis has a highly promising future.

Researchers' interest in examining tweets according to the emotions they convey has grown in recent years. This interest stems from the huge number of tweets published on Twitter, which offer important insights into the opinions of the general public on a wide range of topics. The purpose of this study is to provide the fundamental ideas and methods for sentiment analysis of tweets. Observing the most current TSA applications can help you understand sentiment analysis. The field of TSA research is anticipated to grow quickly in the upcoming years. There will be more research on TSA done in the future.

5.2 Future Work

Future attempts are in progress by use BERT (Bidirectional Encoder Representations from Transformers) and Roberta to analyse sentiment on Twitter. Additionally, the task is to perform the sentiment analysis process in other languages (which now have scripts) like Hindi, Tamil, Telugu, Marathi, and so on, instead of English.

References

1. Link of Research Paper(RoBEARTa): https://arxiv.org/pdf/1907.11692.pdf
2. Yili Wang, Jiaxuan Guo, Chengsheng Yuan and Baozhu Li,” Sentiment Analysis of Twitter Data” Available online November 2022
3. ParamitaRay, and AmlanChakrabarti, "A Mixed approach of Deep Learning method and Rule-Based method to improve Aspect Level Sentiment Analysis”, Applied Computing and Informatics, Available online 4 March 2019.
4. ZiyuanZhao, HuiyingZhu, ZehaoXue, ZhaoLiu, JingTian, Matthew Chin HengChua, and MaofuLiu, "An image-text consistency driven multimodal sentiment analysis approach for social media", Information Processing & Management, vol. 56, no. 6, November 2019.
5. SaeromPark, JaewookLee, and KyoungokKim, "Semi-supervised distributed representations of documents for sentiment analysis", Neural Networks, vol. 119, pp. 139-150, November 2019.
6. MohammadAl-Smadi, OmarQawasmeh, MahmoudAl-Ayyoub, YaserJararweh, and BrijGupta, "Deep Recurrent neural network vs. support vector machine for aspect-based sentiment analysis of Arabic hotels’ reviews", Journal of Computational Science, vol. 27, pp. 386-393, July 2018.
7. AsadAbdi, Siti MariyamShamsuddin, ShafaatunnurHasan, and JalilPiran, "Deep learning-based sentiment classification of evaluative text based on Multi-feature fusion", Information Processing & Management, vol. 56, no. 4, pp. 1245-1259, July 2019.
8. Hyun-jungPark, MinchaeSong, and Kyung-ShikShin, "Deep learning models and datasets for aspect term sentiment classification: Implementing holistic recurrent attention on target-dependent memories", Knowledge-Based Systems, vol. 187, January 2020
9. Zimbra, D.; Abbasi, A.; Zeng, D.; Chen, H. The State-of-the-Art in Twitter Sentiment Analysis: A Review and BenchmarkEvaluation. ACM Trans. Manag. Inf. Syst. 2018,9, 5. [CrossRef]
10. Wang, H.; Can, D.; Kazemzadeh, A.; Bar, F.; Narayanan, S. A System for Real-time Twitter Sentiment Analysis of 2012 U.S.Presidential Election Cycle. In Proceedings of the ACL 2012 System Demonstrations, Jeju Island, Korea, 10 July 2012.
11. Reyna, N.S.; Pruett, C.; Morrison, M.; Fowler, J.; Pandey, S.; Hensley, L. Twitter: More than tweets for undergraduate studentresearchers. J. Microbiol. Biol. Educ. 2022,23, e00326-21. [CrossRef] [PubMed]
12. Khan, I.U.; Khan, A.; Khan, W.; Su’ud, M.M.; Alam, M.M.; Subhan, F.; Asghar, M.Z. A review of Urdu sentiment analysis withmultilingual perspective: A case of Urdu and roman Urdu language. Computers 2022,11, 3. [CrossRef]
13. Pang, B.; Lee, L. Opinion mining and sentiment analysis. Found. Trends®Inf. Retr. 2008,2, 1–135. [CrossRef]
14. Adwan, O.; Al-Tawil, M.; Huneiti, A.; Shahin, R.; Zayed, A.A.; Al-Dibsi, R. Twitter sentiment analysis approaches: A survey. Int.J. Emerg. Technol. 2020,15, 79–93. [CrossRef]
15. Kulkarni, S.; Nadaph, A.; Narang, G. Survey on Twitter Sentiment Analysis using Supervised Machine Learning Algorithms. Int.J. Res. Trends Innov. 2022,7, 2456–3315
16. Giachanou, A.; Crestani, F. Like it or not: A survey of Twitter sentiment analysis methods. ACM Comput. Surv. (CSUR)2016,49, 28. [CrossRef]
17. Giachanou, A.; Crestani, F. Like it or not: A survey of Twitter sentiment analysis methods. ACM Comput. Surv. (CSUR)2016,49, 28. [CrossRef]
18. Swathi, T.; Kasiviswanath, N.; Rao, A.A. An optimal deep learning-based LSTM for stock price prediction using twitter sentimentanalysis. Appl. Intell. 2022,52, 13675–13688. [CrossRef]
19. Jose, A.K.; Bhatia, N.; Krishna, S. Twitter Sentiment Analysis; Seminar Report; National Institute of Technology Calicut: Kozhikode,India, 2010.
20. Kouloumpis, E.; Wilson, T.; Moore, J. Twitter sentiment analysis: The good the bad and the omg! In Proceedings of theInternational AAAI Conference on Web and Social Media, Barcelona, Spain, 17 July 2011.

21. Pang, B.; Lee, L.; Vaithyanathan, S. Thumbs up? Sentiment classification using machine learning techniques. In Proceedings ofthe Conference on Empirical Methods in Natural Language Processing, Philadelphia, PA, USA, 6 July 2002.
22. Nasukawa, T.; Yi, J. Sentiment analysis, Capturing favorability using natural language processing. In Proceedings of theConference on Knowledge Capture, Sanibel Island, FL, USA, 23 October 2003.
23. Jiang, L.; Wang, D.; Cai, Z. Discriminatively weighted naive Bayes and its application in text classification. Int. J. Artif. Intell. Tools2012,21, 1250007. [CrossRef]
24. Zhang, L.; Jiang, L.; Li, C.; Kong, G. Two feature weighting approaches for naive Bayes text classifiers. Knowl.-Based Syst.2016,100, 137–144. [CrossRef]
25. Kim, S.B.; Han, K.S.; Rim, H.C.; Myaeng, S.H. Some effective techniques for naive bayes text classification. IEEE Trans. Knowl.Data Eng. 2006,18, 1457–1466.
26. Jiang, L.; Li, C.; Wang, S.; Zhang, L. Deep feature weighting for naive Bayes and its application to text classification. Eng. Appl.Artif. Intell. 2016,52, 26–39. [CrossRef].
27. Deng, Z.H.; Luo, K.H.; Yu, H.L. A study of supervised term weighting scheme for sentiment analysis. Expert Syst. Appl.2014,41,3506–3513. [CrossRef].
28. Rodrigues, A.P.; Fernandes, R.; Shetty, A.; Lakshmanna, K.; Shafi, R.M. Real-time twitter spam detection and sentiment analysisusing machine learning and deep learning techniques. Comput. Intell. Neurosci. 2022,2022, 5211949. [CrossRef] [PubMed]
29. Mohammad, S.M. From once upon a time to happily ever after: Tracking emotions in mail and books. Decis. Support Syst.2012,53, 730–741. [CrossRef]
30. Chen, L.S.; Liu, C.H.; Chiu, H.J. A neural network based approach for sentiment classification in the blogosphere. J. Informetr.2011,5, 313–322. [CrossRef]
31. Xia, R.; Zong, C.; Li, S. Ensemble of feature sets and classification algorithms for sentiment classification. Inform. Sci.2011,181,1138–1152. [CrossRef]
32. Prabowo, R.; Thelwall, M. Sentiment analysis: A combined approach. J. Informetr. 2009,3, 143–157. [CrossRef].
33. Zhang, L.; Ghosh, R.; Dekhil, M.; Hsu, M.; Liu, B. Combining Lexicon-Based and Learning-Based Methods for Twitter Sentiment Analysis;Technical Report HPL-2011-89; Hewlett-Packard Development Company: Palo Alto, CA, USA, 2011.
34. David Zimbra, M. Ghiassi and Sean Lee, "Brand-Related Twitter Sentiment Analysis using Feature Engineering and the Dynamic Architecture for Artificial Neural Networks", IEEE 1530-1605, 2016.
35. Varsha Sahayak, Vijaya Shete and Apashabi Pathan, "Sentiment Analysis on Twitter Data", (IJIRAE) ISSN: 2349-2163, January 2015.
36. Neethu M S and Rajasree R, "Sentiment Analysis in Twitter using Machine Learning Techniques", IEEE – 31661, 4th ICCCNT 2013.