



Text Analytics and Sentiment Analysis

Module Code	:	CT107-3-3-TXSA Text Analytics and Sentiment Analysis
Intake Code	:	APU3F2311CS(AI)
Lecturer Name	:	Raheem Mafas
Hand in Date	:	29 March 2024
Lab No.	:	11
Group No.	:	39

Student ID	Student Name
TP065697	LOW SIM CHUAN
TP061839	LIAW YU JAY
TP062192	NIVETHAN RAMESH
TP060722	NG ZHI YAO

Table of Contents

Part A	4
Q1: Form tokenization and Filter stop words & punctuation.....	4
Word tokenization	4
Justifications for picking the most suitable method	10
Stop words and punctuations removal.....	11
Explaining the importance of filtering stop words and punctuation	12
Q2: Form word stemming	13
Importance of stemming.....	13
Performing Word Stemming	15
Differences among Regular Expression stemmer, Porter Stemmer and Lancaster Stemmer	18
Q3: Form Parts of Speech (POS) taggers & Syntactic Analyzers.....	19
Performing POS.....	19
The differences of the POS taggers	22
Drawing possible parse trees using Python codes	23
Q4: Work on sentence probabilities	25
Manual computation of sentence probability using unsmoothed bigram model.....	25
Manual computation of sentence probability using smoothed bigram model.....	25
Python implementation of both types of bigram models.....	26
Part B	28
Q1: Dataset and Exploratory Data Analysis (EDA).....	28
The Problem and Literature Review.....	28
Data Cleaning, EDA, Suggestions of Predictive Models	31
Q2: Supervised Text Classification.....	36
LSTM: Low Sim Chuan	36

SVM – LinearSVC: Nivethan.....	46
One Dimensional Convolutional Neural Network – Liaw Yu Jay	52
Naïve Bayes: Ng Zhi Yao.....	62
Building A Supervised Text Classification model using SAS Text Miner	66
Q3: Hyperparameter selection.....	71
LSTM: Low Sim Chuan	71
SVM-LinearSVC: Nivethan	76
One Dimensional Convolutional Neural Network: Liaw Yu Jay	79
Naïve Bayes: Ng Zhi Yao.....	84
Q4: Evaluation & Discussion of the predictive models’ results	85
LSTM: Low Sim Chuan	85
SVM-LinearSVC: Nivethan	88
One Dimensional Convolutional Neural Network: Liaw Yu Jay	90
Naïve Bayes: Ng Zhi Yao.....	92
Picking best model with discussions and analyses	95
EXTRA FEATURES	96
References	97

Part A

Q1: Form tokenization and Filter stop words & punctuation.

Word tokenization

```
[ ] from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

[ ] %cd '/content/drive/MyDrive/Colab Notebooks/Text Analysis and Sentiment Analysis/Assignment/Assignment Data/Assignment Data'
/content/drive/MyDrive/Colab Notebooks/Text Analysis and Sentiment Analysis/Assignment/Assignment Data/Assignment Data

[ ] with open('Data_1.txt', 'r') as file:
    text_data = file.read()

print(text_data)

Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities, events, and their properties. Opinions are usually subjective expressions that describe people's sent
```

Figure 1

The figure shows that the first step of the tokenization process is loading the dataset Data_1.txt into google colab where the process is conducted. The data loaded is stored under the variable text_data. The contents of text_data are printed out to ensure the correct data is loaded.

Split() Function

```
#Perform tokenization using python split function
# Default splitting without using specifying string parameter

tokens = text_data.split()

counter = 0
for w in tokens:
    counter+=1
    print(counter, ".", w)

# Print all the tokens
print(tokens)
print()
```

Figure 2

The figure shows the code to perform tokenization using python's built-in function split(). The function can which is **delimiter/separator** parameter that specifies the character or string which the input string will be split based on. The other parameter is **Max Split** that can be used to represent the maximum number of splits allowed on the input string. The code above shows that the text data stored in the variable text_data are split into tokens using the default split() function

which will split the input string based on empty space and stored in the variable **tokens**. Then, the result tokens are printed out one by one using a **for** loop.

```

1 tokens = text_data.split()
2 for token in tokens:
3     print(token)
4
5 # Output:
6 Textual
7 Information
8 in
9 the
10 world
11 can
12 be
13 broadly
14 categorized
15 into
16 two
17 main
18 types:
19 facts
20 and
21 opinions.
22 Facts
23 are
24 objective
25 expressions
26 about
27 entities,
28 events,
29 and
30 their
31 properties.
32 Opinions
33 are
34 usually
35 subjective
36 expressions
37 that
38 describe
39 people's
40 sentiments,
41 appraisals,
42 or
43 feelings
44 toward
45 entities,
46 events,
47 and
48 their
49 properties.
50 [Textual, 'Information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', 'facts', 'and', 'opinions', 'Facts', 'are', 'objective', 'expressions', 'about', 'entities', 'events', 'and', 'their', 'properties', 'Opinions', 'are', 'usually', 'subjective', 'expressions', 'that', 'describe', 'people's', 'sentiments', 'appraisals', 'or', 'feelings', 'toward', 'entities', 'events', 'and', 'their', 'properties']

```

Figure 3

The figure above shows the result obtained for tokenization process using split method. The split() method can split the token properly except for the fact it does not separate the punctuations into a different token as shown in the figure above. This is because it splits purely based on the empty spaces.

```

1 # Perform splitting specifying specifying comma as string parameter
2 tokens = text_data.split(",")
3 counter = 0
4 for token in tokens:
5     counter+=1
6     print(counter, ",", token)
7
8 # Print all the tokens
9 print(tokens)
10
11 # Specifying the new split parameter before splitting
12 tokens = text_data.split(",", 2)
13 counter = 0
14 for token in tokens:
15     counter+=1
16     print(counter, ",", token)
17
18 # Print all the tokens
19 print(tokens)
20
21 1 - Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities,
22 2 - events,
23 3 - and their properties. Opinions are usually subjective expressions that describe people's sentiments,
24 4 - appraisals,
25 5 - or feelings toward entities
26 6 - events
27 7 - and their properties.
28 [Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities, ' events', ' and their properties. Opinions are usually subjective expressions that describe people's sentiments, ' appraisals', ' or feelings toward entities', ' events', ' and their properties.]
29
30 1 - Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities
31 2 - events,
32 3 - and their properties. Opinions are usually subjective expressions that describe people's sentiments,
33 4 - appraisals,
34 5 - or feelings toward entities,
35 6 - events,
36 7 - and their properties.
37 [Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities', ' events', ' and their properties. Opinions are usually subjective expressions that describe people's sentiments, appraisals, or feelings toward entities, events, and their properties.]

```

Figure 4

The figure above shows another alternative where the delimiter and max split parameter are specified. In this case, the delimiter is set to “,” which means the tokens will be split based on “,” and the max split is set to 2 which means only a maximum of two splits will be performed on the input string as shown in the figure above. The second splitting ignores all the “,” characters in the sequence and groups them all in a token.

Tokenization using regular expression

```

import re

tokens = re.findall("[\w"]+", text_data)
counter = 0
for w in tokens:
    counter += 1
    print(counter, ".", w)

print(tokens)
print()

```

Figure 5

The figure above shows the tokenization process done using regular expression. Firstly, the `re` library is imported. Then, the tokenization process is performed on the dataset `text_data` using `re.findall` function. The parameter inside `[\w"]+` is a regular expression pattern that specifies what type of tokens it is looking for. The `[]` represents a character set that matches any character inside the bracket. The `"\w"` matches any word character while the `"+"` matches one or word characters. The result of tokenization is stored under the `tokens`. Then, each token is printed out using a for loop.

```

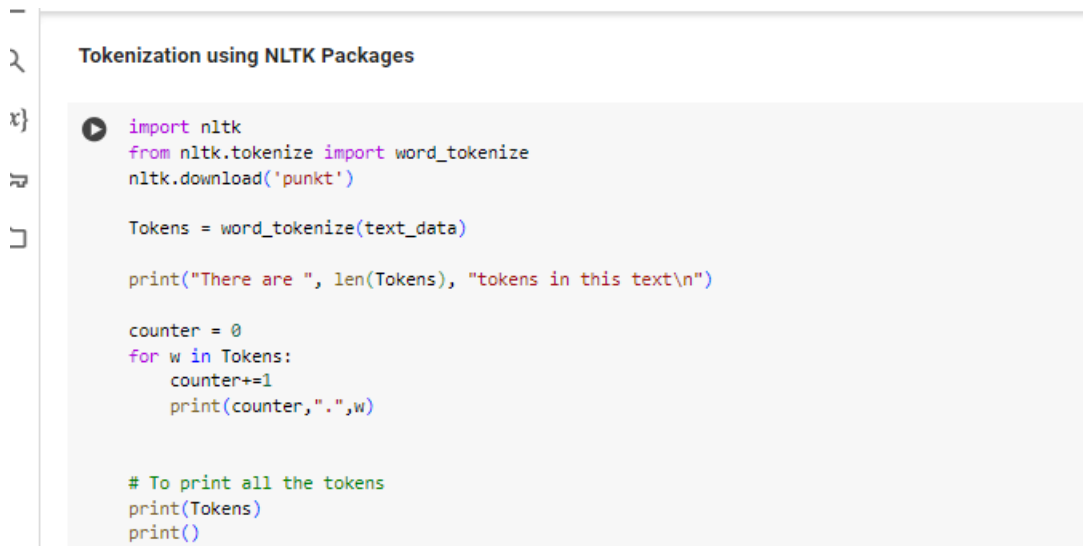
1: text_data
2: information
3: the
4: world
5: is
6: a
7: complex
8: system
9: that
10: can
11: be
12: used
13: to
14: store
15: data
16: and
17: process
18: it
19: in
20: a
21: way
22: that
23: is
24: efficient
25: and
26: reliable
27: and
28: secure
29: and
30: easy
31: to
32: use
33: and
34: maintain
35: and
36: update
37: and
38: delete
39: and
40: insert
41: and
42: search
43: and
44: sort
45: and
46: print
47: and
48: display
49: and
50: format
51: and
52: convert
53: and
54: compare
55: and
56: calculate
57: and
58: evaluate
59: and
60: analyze
61: and
62: synthesize
63: and
64: create
65: and
66: destroy
67: and
68: restore
69: and
70: backup
71: and
72: recover
73: and
74: archive
75: and
76: purge
77: and
78: migrate
79: and
80: clone
81: and
82: snapshot
83: and
84: virtualize
85: and
86: containerize
87: and
88: orchestrate
89: and
90: manage
91: and
92: monitor
93: and
94: log
95: and
96: audit
97: and
98: secure
99: and
100: compliant

```

Figure 6

The above figure shows the result of tokenization process conducted using regular expression. It can be observed that almost every word is properly tokenized except the word with apostrophe punctuation where the regular expression splits both the original word and the apostrophe s into two tokens. Not only the regular expression also does not take punctuation as tokens.

Tokenization using NLTK libraries



The screenshot shows a Jupyter Notebook interface with a code cell titled "Tokenization using NLTK Packages". The code imports the NLTK library and the word_tokenize function from nltk.tokenize. It downloads the 'punkt' tokenizer model. The text_data is tokenized using word_tokenize, and the resulting tokens are stored in the Tokens variable. The number of tokens is printed, and a loop prints each token with a counter. Finally, all tokens are printed.

```
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')

Tokens = word_tokenize(text_data)

print("There are ", len(Tokens), "tokens in this text\n")

counter = 0
for w in Tokens:
    counter+=1
    print(counter, ".", w)

# To print all the tokens
print(Tokens)
print()
```

Figure 7

The figure above shows the tokenization process done using word_tokenize function under the nltk.tokenize library. Firstly, the necessary libraries are imported. Then, text_data is tokenized using the word_tokenize function and results are stored in tokens variable. The number of tokens formed, and each token is printed out.



The screenshot shows the output of the NLTK tokenization process. It lists 56 tokens, including words like 'broadly', 'categorized', 'into', 'two', 'main', 'types', 'facts', 'and', 'opinions', 'are', 'objective', 'expressions', 'about', 'entities', 'events', 'properties', 'subjective', 'usually', 'that', 'describe', 'people', 's', 'sentiments', 'appraisals', 'or', 'feelings', 'toward', 'entities', 'events', 'and', 'their', 'properties'. The output is displayed as a list of tokens, with some tokens highlighted in red boxes.

```
[ ] 8 . broadly
    9 . categorized
    10 . into
    11 . two
    12 . main
    13 . types
    14 . :
    15 . facts
    16 . and
    17 . opinions
    18 . ,
    19 . Facts
    20 . are
    21 . objective
    22 . expressions
    23 . about
    24 . entities
    25 . ,
    26 . events
    27 . ,
    28 . and
    29 . their
    30 . properties
    31 . ,
    32 . Opinions
    33 . are
    34 . usually
    35 . subjective
    36 . expressions
    37 . that
    38 . describe
    39 . people
    40 . ,
    41 . s
    42 . sentiments
    43 . ,
    44 . appraisals
    45 . ,
    46 . or
    47 . feelings
    48 . toward
    49 . entities
    50 . ,
    51 . events
    52 . ,
    53 . and
    54 . their
    55 . properties
    56 .

['Textual', 'Information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', ':', 'facts', 'and', 'opinions', ',', 'Facts', 'are', 'objective', 'expressions', 'about', 'entities', ', events', 'properties', 'subjective', 'usually', 'that', 'describe', 'people', 's', 'sentiments', 'appraisals', 'or', 'feelings', 'toward', 'entities', 'events', 'and', 'their', 'properties']
```

Figure 8

The above figure shows the result obtained for tokenization process done using `word_tokenize` function. The main difference is that the function considers punctuation by placing punctuation as individual tokens.



```

1 # Import the word_tokenize function from the nltk library
2 from nltk.tokenize import word_tokenize
3
4 # Define the sentence to be tokenized
5 sentence = "The cat sat on the mat."
6
7 # Tokenize the sentence
8 tokens = word_tokenize(sentence)
9
10 # Print the tokens
11 print(tokens)

```

Output:

```

['The', 'cat', 'sat', 'on', 'the', 'mat', '.']

```

Figure 9

The figure above shows the code and result of using `word_tokenize` library for tokenization. It can be observed that `word_tokenize` library successfully tokenized the sentence, and the apostrophe is identified as an individual token.


```
import spacy
nlp = spacy.load("en_core_web_sm")

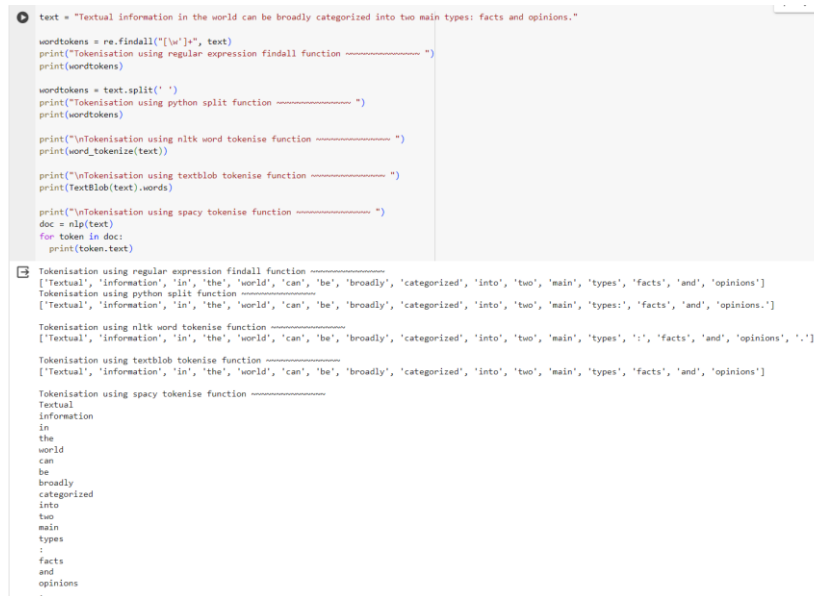
doc = nlp(text_data)
for token in doc:
    print(token.text)
```

```
Textual
information
in
the
world
can
be
broadly
categorized
into
two
main
types
:
facts
and
opinions
.
Facts
are
objective
expressions
about
entities
,
events
,
and
their
properties
.
Opinions
are
usually
subjective
expressions
that
describe
people
's
sentiments
,
appraisals
,
or
feelings
toward
entities
,
events
,
and
their
properties
```

Figure 10

The above figure shows the tokenization process done using the spacy library. One of the differences between the spacy library and the other libraries is that it can work in other languages as well which is why at the beginning of the code, we need to load the correct language. The result is almost the same as word_tokenize function but spacy classifies “’s” as an individual token.

Justifications for picking the most suitable method



```

text = "Textual information in the world can be broadly categorized into two main types: facts and opinions."

wordtokens = re.findall("[\w]", text)
print("Tokenisation using regular expression findall function -----")
print(wordtokens)

wordtokens = text.split(' ')
print("Tokenisation using python split function -----")
print(wordtokens)

print("\nTokenisation using nltk word tokenize function -----")
print(word_tokenize(text))

print("\nTokenisation using textblob tokenize function -----")
print(TextBlob(text).words)

print("\nTokenisation using spacy tokenize function -----")
doc = nlp(text)
for token in doc:
    print(token.text)

Tokenisation using regular expression findall function -----
['Textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', 'facts', 'and', 'opinions']
Tokenisation using python split function -----
['Textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types:', 'facts', 'and', 'opinions.']
Tokenisation using nltk word tokenize function -----
['Textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', '!', 'facts', 'and', 'opinions', '.']
Tokenisation using textblob tokenize function -----
['Textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', '!', 'facts', 'and', 'opinions']
Tokenisation using spacy tokenize function -----
Textual
information
in
the
world
can
be
broadly
categorized
into
two
main
types
:
facts
and
opinions
.

```

Figure 11

Based on the figure above, the most suitable tokenization operation for text analytics is the `nltk word_tokenize` function. The NLTK's `word_tokenize()` function uses more complex tokenization rules. It provides more concise punctuation handling where punctuation marks are marked as separate tokens. In the provided example, it appropriately divides the colon (':') and period ('.') as distinct tokens, whereas the other techniques do not. This can be useful if researchers need to evaluate or process punctuation marks apart from the text. Not only that, The NLTK's `word_tokenize()` method handles contractions correctly. Contractions are word combinations in which an apostrophe replaces deleted letters such as "can't" for "cannot". It accurately separates contractions into individual tokens. In the provided example, both "can't" and "types:" are split into two tokens to preserve the contractions' meaning.

Stop words and punctuations removal



```

import nltk
import string
nltk.download('punkt')
nltk.download('stopwords')

text_lower = text_data.lower()
wordtokens = nltk.tokenize.word_tokenize(text_lower)
print(wordtokens)

print("\nTotal number of words in this text corpus is", len(wordtokens))

stopTokens = nltk.corpus.stopwords.words("english") + list(string.punctuation)
filteredTokens = []

stopWordsFound = []

for w in wordtokens:
    if w in stopTokens:
        stopWordsFound.append(w)
    else:
        filteredTokens.append(w)

print("There are", len(filteredTokens), "words in this text corpus after removing stop words and punctuations \n")
print(filteredTokens)
print("There are", len(stopWordsFound), "stop words and punctuations in this text corpus:\n")
print(stopWordsFound)

```

['textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', ':', 'facts', 'and', 'opinions', '.', 'facts', 'are', 'object
Total number of words in this text corpus is 56
There are 30 words in this text corpus after removing stop words and punctuations

['textual', 'information', 'world', 'broadly', 'categorized', 'two', 'main', 'types', 'facts', 'opinions', 'facts', 'objective', 'expressions', 'entities', 'events', 'properties', '
There are 26 stop words and punctuations in this text corpus:

['in', 'the', 'can', 'be', 'into', ':', 'and', '.', 'are', 'about', ',', ' ', 'and', 'their', '.', 'are', 'that', 's', ',', ' ', 'on', ',', ' ', 'and', 'their', '.']

Figure 12

The code above demonstrates the code for the removal of stop words and punctuation and the result obtained from it. The *text_data* that contains Data_1.txt is first converted to lowercase. Then, it is tokenized into individual words using the *word_tokenize()* method from NLTK. A list of stop words stored in *stopTokens* is then obtained by combining NLTK's English stop words and the string punctuation. Then, the code iterates through each word in *wordtokens* and checks if it is present in *stopTokens*. If it is a stop word or punctuation, the word is appended to *stopWordsFound*. Otherwise, it is appended to *filteredTokens*. The code then prints the number of words after removing stop words and punctuations and displays the filtered words stored in *filteredTokens*. It also prints the number of stop words and punctuations found in the text corpus and displays the list of stop words and punctuations stored in *stopWordsFound*.

It can be seen that the output showed that stop words and punctuations are successfully removed from the text corpus as shown that only 30 words remain in the text corpus are remained after removing stop words and punctuations. The output of the filtered tokens is printed out. For the stop words and punctuation, it can be seen that 26 words and punctuations are successfully identified. The result of these stop words and punctuation are clearly printed out as shown in the output above.

Explaining the importance of filtering stop words and punctuation

First of all, filtering out stop words and punctuation allows for more attention on relevant terms, as stop words like "the," "is," and "and" are common but have little meaning and contribute little to the essence of the text. Removing them allows the model to focus on the more meaningful words, which are frequently the subject of investigation (GeekforGeeks, n.d.). Secondly, it provides improved computational efficiency where processing text data can be computationally intensive. Eliminating stop words and punctuation makes the dataset smaller and more manageable, resulting in faster processing times (GeekforGeeks, n.d.). Thirdly, it provides increased accuracy. For many NLP applications, such as text classification, the existence of stop words can introduce noise and lower algorithmic accuracy. Filtering them away may lead to more accurate analysis and outcomes (GeekforGeeks, n.d.). Fourthly, it also ensures better context management. While stop words are typically deemed low information, they can occasionally give context to a query. In such circumstances, removing stop words may be based on the individual application and desired outcome (Ganesan, 2024). Finally, punctuation marks can result in different variants of the same word. So, removing punctuation helps to normalize words, guaranteeing that they are regarded the same manner by analytical algorithms (Sydney, 2020).

Q2: Form word stemming

Importance of stemming

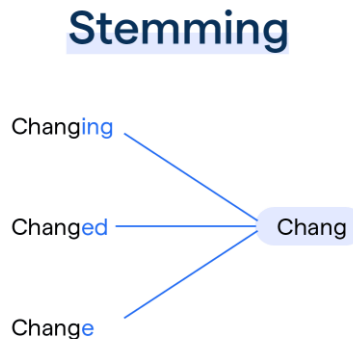


Figure 13: Example of Stemming (BotPenguin, n.d.)

Word stemming is one of the popular text processing techniques that is used to remove prefixes and suffixes from words which convert them to their core or root form (Jain, 2024). The main purpose of this process is to streamline and standardize words when performing natural language processing (NLP) tasks. It is undeniable that stemming process is an important step to execute during text analysis as it helps to **normalize words to their original form**. For instance, English words often exist in different forms due to inflectional morphology such as plurals and verb conjugations. By performing stemming process, it can ensure the variations of the same word are treated as identical which can simplify analysis and improve accuracy when training model.

Apart from that, stemming can **reduce the dimensionality of the data** by consolidating words with similar meaning into a single representation. By doing so, it simplifies the process of subsequent text processing tasks such as classification and clustering which makes them more efficient and effective. Furthermore, stemming can **enhance the consistency of text analysis** by ensuring different variations of the same words are being processed uniformly. As a result, tasks like sentiment analysis could still maintain their performances.

Moreover, stemming can **improve the effectiveness of information retrieval systems** by allowing users to retrieve documents containing variations of search terms faster. To quote an example, a search for “run” word can retrieve documents containing “running” and “ran” if stemming has been applied to both the search query. Lastly, stemming algorithms can be **applied across various languages**. This is because the concept of stemming is still applicable even though the effectiveness across languages may be slightly different. As a result, it can help in processing and analysis of text in various linguistic contexts, making it a valuable tool for researchers when analyzing text combining various languages.

Performing Word Stemming

```
import re

def stem_word(word):
    # Define regex pattern for common suffixes
    pattern = r'(.?)(ly|es|s|y|ing)?$'

    # Apply regex substitution
    stem, suffix = re.match(pattern, word).groups()

    return stem

# Sample paragraph
paragraph = "Textual information in the world can be broadly categorized into two main types: qualitative and quantitative."

# Split the paragraph into words
words = re.findall(r'\b\w+\b', paragraph)

# Stem each word and print the results
for word in words:
    print(word + " -> " + stem_word(word))
```

Figure 14: Sample Code of Regex on Word Stemming

```
Textual -> Textual
information -> information
in -> in
the -> the
world -> world
can -> can
be -> be
broadly -> broad
categorized -> categorized
into -> into
two -> two
main -> main
types -> typ
facts -> fact
and -> and
opinions -> opinion
Facts -> Fact
are -> are
objective -> objective
expressions -> expression
about -> about
entities -> entiti
events -> event
and -> and
their -> their
properties -> properti
Opinions -> Opinion
are -> are
usually -> usual
subjective -> subjective
expressions -> expression
that -> that
describe -> describe
people -> people
```

Figure 15: Sample output of Regex on Word Stemming

Figure above shows the algorithms to perform word stemming using Regex where a function is created to put the character slicing logic. For instance, words ending with “ly”, “es”, “s”, “s”, “y” and “ing” are being cut off from the string and display the remaining characters in the console.


```

] from nltk.stem import PorterStemmer
import re

# Create a Porter Stemmer instance
porter_stemmer = PorterStemmer()

# Sample paragraph
paragraph = "Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities, events, and t

# Split the paragraph into words
words = re.findall(r'\b\w+\b', paragraph)

# Apply stemming to each word using Porter Stemmer
stemmed_words = [porter_stemmer.stem(word) for word in words]

# Print the results
print("Original words:", words)
print("Stemmed words with Porter Stemmer:", stemmed_words)

```

Original words: ['Textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', 'facts', 'and', 'opinions', 'Facts', 'a
Stemmed words with Porter Stemmer: ['textual', 'inform', 'in', 'the', 'world', 'can', 'be', 'broadli', 'categor', 'into', 'two', 'main', 'type', 'fact', 'and', 'opinion', 'fact

Figure 16: Sample Code and Output of Porter Stemmer on Word Stemming

Figure above describes Porter Stemmer which is one of the popular techniques used in word stemming. By using “nltk” modules, researchers are able to acquire the base form of the word. However, some words may still be invalid due to the principle of word stemming.

```

from nltk.stem import LancasterStemmer
import re

# Create a Lancaster Stemmer instance
stemmer = LancasterStemmer()

# Sample paragraph
paragraph = "Textual information in the world can be broadly categorized into two main types: facts and opinions. Facts are objective expressions about entities, events, and

# Split the paragraph into words
words = re.findall(r'\b\w+\b', paragraph)

# Apply Lancaster Stemmer to each word
stemmed_words = [stemmer.stem(word) for word in words]

# Print the results
print("Original words:", words)
print("Stemmed words with Lancaster Stemmer:", stemmed_words)

```

Original words: ['Textual', 'information', 'in', 'the', 'world', 'can', 'be', 'broadly', 'categorized', 'into', 'two', 'main', 'types', 'facts', 'and', 'opinions', 'Facts', 'a
Stemmed words with Lancaster Stemmer: ['text', 'inform', 'in', 'the', 'world', 'can', 'be', 'broad', 'categ', 'into', 'two', 'main', 'typ', 'fact', 'and', 'opin', 'fact', 'an

Figure 17: Sample Code and Output of Lancaster Stemmer on Word Stemming

Figure above demonstrates the use of Lancaster Stemmer to perform word stemming. The output is printed out in the console after the process.

Differences among Regular Expression stemmer, Porter Stemmer and Lancaster Stemmer

In short, each stemmer offers distinct approaches to word stemming in NLP. The Regex stemmer provides a simple and straightforward method based on regex patterns to remove common suffixes which make it easy to implement but limited in linguistic accuracy. For example, words like 'categorized' stemmed to 'categor'. On the other hand, Porter and Lancaster Stemmer utilize more sophisticated algorithms and thus offer an improved accuracy in generating stems that capture the root meaning of the words. For instance, words like 'broadly' becoming 'broadli' in Porter Stemmer which strikes a balance between complexity and accuracy to generate stems that are generally recognizable. However, Lancaster Stemmer performs stemming to word like 'broadly' to 'broad' which takes a more aggressive stance by sacrificing readability for shorter stems. These examples illustrate the trade-offs between simplicity, accuracy and aggressiveness in stemming algorithms while highlighting the importance of choosing the most suitable stemmer based on the specific requirements and goals of NLP tasks. For applications where simplicity and basic stemming suffice, Regex stemmer is sufficient. For more nuanced and accurate stemming, Porter Stemmer offers a balanced approach while the Lancaster Stemmer is suitable for cases where aggressive stemming is acceptable or even beneficial with potential trade-offs in readability and interpretability.

Q3: Form Parts of Speech (POS) taggers & Syntactic Analyzers

Performing POS

NLTK POS TAGGER

NLTK POS TAGGER

```

import nltk
from nltk.tokenize import word_tokenize

# Ensure the necessary NLTK resources are available
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')

data = "The big black dog barked at the white cat and chased away."

# Tokenizing and POS Tagging using NLTK
nltk_pos_tagged = nltk.pos_tag(word_tokenize(data))
print("NLTK POS Tagging:", nltk_pos_tagged)

[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data] /root/nltk_data...
[nltk_data] Unzipping taggers/averaged_perceptron_tagger.zip.
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
NLTK POS Tagging: [('The', 'DT'), ('big', 'JJ'), ('black', 'JJ'), ('dog', 'NN'), ('barked', 'VBD'), ('at', 'IN'), ('the', 'DT'), ('white', 'JJ'), ('cat', 'NN'), ('and', 'CC'),

```

Figure 18

```

('chased', 'VBD'), ('away', 'RB'), ('.', '.'))

```

Figure 19

The NLTK POS tagger has been integrated into the Google Colab platform, as seen in the photos above. The pictures demonstrate a multi-step procedure, the first of which is importing libraries for the NLTK POS tagger. The next step is to get the NLTK resources. In order to tag the sentence with NLTK POS, it is first put in a variable. At last, the NLTK POS tagger approach was used for tokenization and POS tagging. As the photographs up top show, the product is then printed. There is a tuple structure in the output that displays the input phrase words coupled with their respective part of speech tags. For instance, the fact that "The" is marked as a determiner ('DT') is shown by the syntax ('The','DT'). The Pennsylvania Treebank Project's tag set is followed by the POS tags. The NLTK library often makes use of this collection of POS tags.

TEXTBLOB POS TAGGER

TextBlob POS Tagger

```

from textblob import TextBlob

data = "The big black dog barked at the white cat and chased away."

# Creating a TextBlob object and POS Tagging using TextBlob
tb = TextBlob(data)
textblob_pos_tagged = tb.tags
print("TextBlob POS Tagging:", textblob_pos_tagged)

TextBlob POS Tagging: [('The', 'DT'), ('big', 'JJ'), ('black', 'JJ'), ('dog', 'NN'), ('barked', 'VBD'), ('at', 'IN'), ('the', 'DT'), ('white', 'JJ'), ('cat', 'NN'), ('and', 'CC'),

```

Figure 20

```

('chased', 'VBD'), ('away', 'RB')]

```

Figure 21

Using the Google Colab platform, the aforementioned pictures demonstrate the integration of the TextBlob POS tagger. It all starts with importing the 'textblob' library. A further step in data preparation is to save the phrase in a variable. The next step is to create an object of type "TextBlob" and pass the variable into it. You can see the result in figures 3.3 and 3.4, which show that the POS tagging finally took place. There is a tuple structure in the output that displays the input phrase words coupled with their respective part of speech tags. To illustrate, the notation ('The','DT') shows that 'The' is a determiner ('DT'). The Pennsylvania Treebank Project's tag set is followed by the POS tags. In the TextBlob library, this is the typical, standard collection of POS tags.

REGULAR EXPRESSION TAGGER

Regular Expression Tagger

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.tag import RegexpTagger

data = "The big black dog barked at the white cat and chased away."

# Define the regex patterns for POS tagging
patterns = [
    (r'^-?[0-9]+(\.[0-9]+)?$', 'CD'), # cardinal numbers
    (r'(The|the|A|a|An|an)$', 'DT'), # articles
    (r'.*able$', 'JJ'), # adjectives
    (r'.*ness$', 'NN'), # nouns formed from adjectives
    (r'.*ly$', 'RB'), # adverbs
    (r'.*\'s$', 'POS'), # possessive nouns
    (r'.*ment$', 'NN'), # nouns that end in -ment
    (r'.*town$', 'NN'), # nouns that end in -town
    (r'.*ness$', 'NN'), # nouns ending in -ness
    (r'.*ing$', 'VBG'), # gerunds
    (r'.*ed$', 'VBD'), # simple past
    (r'.*es$', 'VBZ'), # 3rd singular present
    (r'^[A-Z].*$', 'NNP'), # proper nouns
    (r'.*ould$', 'MD'), # modals
    (r'.*s$', 'NNS'), # plural nouns
    (r'.*', 'NN') # nouns (default)
]

# Apply the Regular Expression Tagger
regexp_tagger = RegexpTagger(patterns)
regexp_pos_tagged = regexp_tagger.tag(word_tokenize(data))
print("Regular Expression POS Tagging:", regexp_pos_tagged)
```

Figure 22

Regular Expression POS Tagging: [('The', 'DT'), ('big', 'NN'), ('black', 'NN'), ('dog', 'NN'), ('barked', 'VBD'), ('at', 'NN'), ('the', 'DT'), ('white', 'NN'), ('cat', 'NN'),

Figure 23

('cat', 'NN'), ('and', 'NN'), ('chased', 'VBD'), ('away', 'NN'), ('.', 'NN')]

Figure 24

You can see the Regular Expression POS tagger in action on the Google Colab platform in these photographs. As is customary, importing the libraries is the first step in the implementation process. After that, the data is ready by putting the phrase into a variable. Afterwards, the regular expression patterns are specified. Subsequently, the 'RegexpTagger' object is created. Lastly, the 'RegexpTagger' object is used for tokenization and tagging. In tuple format, the output displays the relevant part of speech tag for each word in the input phrase. For instance, the fact that "The" is marked as a determiner ('DT') is shown by the syntax ('The','DT').

The differences of the POS taggers

"The big black dog barked at the white cat and chased away." is an example of a statement with a lot of potential POS tagger output variations. A manually annotated corpus of English text serves as the basis for the NLTK POS tagger's training. This tagger would figure out what words go with what tags by studying the words and sentences. Consequently, this tagger has an extremely high rate of accuracy. In addition, as we saw with the NLTK POS tagger, TextBlob's POS tagger is quite similar. The fact that the TextBlob encapsulates the NLTK's POS tagger is what gives rise to these similarities. Because of this, the TextBlob POS tagger's user interface may become much more intuitive. This tagger streamlines the interface while using the same NLTK-based technique and resources. After taking the word's context and the model's training patterns into account, the TextBlob POS tagger produces results similar to those of the NLTK POS tagger.

On the other hand, the Regular Expression Tagger differs from the two taggers that were previously discussed. All of the regex patterns dictate the tagger's output. Words with numerous possible tags or those requiring context may not be as accurate. Due to the simplicity of regex matching, this tagger may be quicker for really basic tagging jobs. But precision and durability suffer as a result.

Drawing possible parse trees using Python codes

```
import nltk

# Define the grammar for parsing the sentence
grammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> Det AdjP N | Det N
AdjP -> Adj AdjP | Adj
VP -> V NP PP | V PP | V NP | VP Conj VP
VP -> V Adv | V
Adv -> 'away'
PP -> P NP
Det -> 'The' | 'the'
N -> 'dog' | 'cat'
Adj -> 'big' | 'black' | 'white'
V -> 'barked' | 'chased'
P -> 'at'
Conj -> 'and'
""")

# Sentence to be parsed
sentence = "The big black dog barked at the white cat and chased away"

# Tokenize the sentence
tokens = nltk.word_tokenize(sentence)

# Create a parser with the defined grammar
parser = nltk.ChartParser(grammar)

# Generate parse trees
trees = list(parser.parse(tokens))
for tree in trees:
    tree.pretty_print()
```

Figure 25

For the phrase "The big black dog barked at the white cat and chased away", the code constructs a context-free grammar (CFG), as shown in the figure above. The phrase structures that may be used in this statement are defined by this CFG. One example is the way in which sentences are defined by grammar. A sentence is said to consist of certain components of speech, such as a noun phrase (NP), a verb phrase (VP), and so on.

Following the receipt of the CFG and the tokenized input text, the 'nltk.ChartParser' is used to produce parse trees. A phrase may be parsed into its component words using the 'nltk.word_tokenize' tokenizer.

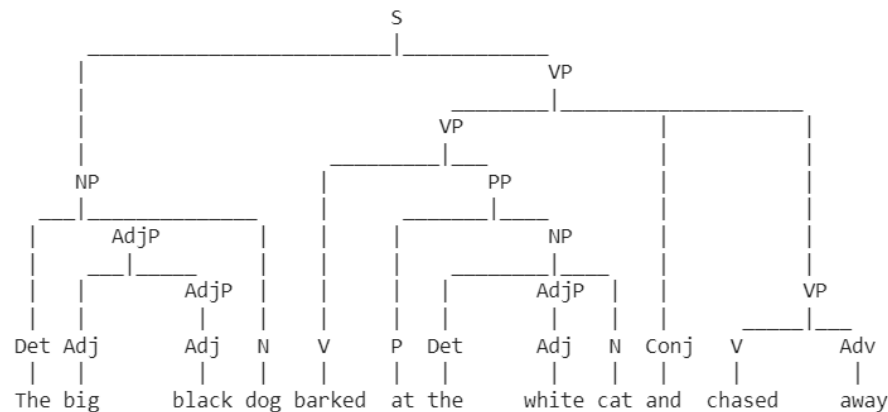


Figure 26

The diagram in the figure above shows the sentence's hierarchical structure as it is described by the CFG. An element of grammar that is specified in the CFG is represented by each branch of the tree. Depicted as tree leaves are the sentence's tokens.

If you look at the tree diagram, you can see that the sentence (S) is at the very top. A noun phrase (NP) and a verb phrase (VP) are formed when the structure is divided under a sentence. A determiner (Det) and an adjective phrase (AdjP) are the building blocks of NP, which is then further subdivided into a noun (N). Following this, the AdjP was enabled to break into two parts which are an adjective (Adj) and another AdjP. Finally, the VP is split to reveal the conjunction (Conj) of two verb phrases, one of which contains a verb (V) and the other also contains additional components.

All the grammatical connections between the sentence's components are shown in the above tree. This is done by demonstrating the deconstruction of each word into its components in accordance with the requirements of a CFG. The NLTK package includes functions such as 'CFG.fromstring,' 'ChartParser,' and 'word_tokenize' that would be used to create grammar, parse sentences, and construct these trees. Linguistic analysis and NLP applications will find this quite useful.

Q4: Work on sentence probabilities

Manual computation of sentence probability using unsmoothed bigram model

Using unsmoothed bigram model:

$$\begin{aligned}
 &P(< s > \text{ I read a book by Danielle } < / s >) \\
 &= P(I | < s >) \cdot P(\text{read} | I) \cdot P(a | \text{read}) \cdot P(\text{book} | a) \cdot P(\text{by} | \text{book}) \cdot P(\text{Danielle} | \text{by}) \cdot P(< / s > | \text{Danielle}) \\
 &= \frac{1}{3} \cdot \frac{1}{1} \cdot \frac{3}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{1}{1} \cdot \frac{1}{1} = 0.07407407407
 \end{aligned}$$

Figure 27: Manual computation using unsmoothed bigram model

Manual computation of sentence probability using smoothed bigram model

Using smoothed bigram model:

$$\begin{aligned}
 &P(< s > \text{ I read a book by Danielle } < / s >) \\
 &= P(I | < s >) \cdot P(\text{read} | I) \cdot P(a | \text{read}) \cdot P(\text{book} | a) \cdot P(\text{by} | \text{book}) \cdot P(\text{Danielle} | \text{by}) \cdot P(< / s > | \text{Danielle}) \\
 &= \frac{1+1}{3+10+1} \cdot \frac{1+1}{1+10+1} \cdot \frac{3+1}{3+10+1} \cdot \frac{2+1}{3+10+1} \cdot \frac{1+1}{3+10+1} \cdot \frac{1+1}{1+10+1} \cdot \frac{1+1}{1+10+1} = 0.00000578462677588
 \end{aligned}$$

Figure 28: Manual computation using smoothed bigram model

Python implementation of both types of bigram models

```
def compute_sentence_probability_unsmoothed_bigram(sentence, vocabs, text_corpus):
    tokens = sentence.split()
    bigram_counts = {}
    # Sliding through corpus to get bigram counts
    for i in range(len(text_corpus) - 1):
        # Getting bigram at each slide
        bigram = (text_corpus[i], text_corpus[i+1])
        # Keeping track of the bigram counts
        if bigram in bigram_counts.keys():
            bigram_counts[bigram] += 1
        else:
            bigram_counts[bigram] = 1

    vocab_counts = {}
    # Sliding through vocabs to get vocab counts
    for vocab in vocabs:
        for word in text_corpus:
            if word == vocab:
                if vocab in vocab_counts.keys():
                    vocab_counts[vocab] += 1
                else:
                    vocab_counts[vocab] = 1
    sentence_prob = 1
    # Sliding through the `tokens` to get bigrams
    for i in range(len(tokens) - 1):
        # Getting bigram at each slide
        bigram = (tokens[i], tokens[i+1])
        sentence_prob *= bigram_counts.get(bigram, 0) / (vocab_counts.get(tokens[i]))
    return sentence_prob

compute_sentence_probability_unsmoothed_bigram(test_sentence, vocabs, text_corpus)

0.07407407407407407
```

Figure 29: Python implementation for unsmoothed bigram model

The figure above shows the python implementation for the unsmoothed bigram model. First, the **sentence** “<s> I read a book by Danielle </s>” is split into tokens, then the text corpus is looped through to get the counts of each possible bigram. Then, **vocabs** is also looped through to get the counts for each vocabulary in the **text_corpus**. Finally, the sentence probability is computed using the formula for the unsmoothed bigram model. The sentence probability is about 0.0740741, which is exactly the same as the manual computations.

```
def compute_sentence_probability_smoothed_bigram(sentence, vocabs, text_corpus):
    tokens = sentence.split()
    bigram_counts = {}
    # Sliding through corpus to get bigram counts
    for i in range(len(text_corpus) - 1):
        # Getting bigram at each slide
        bigram = (text_corpus[i], text_corpus[i+1])
        # Keeping track of the bigram counts
        if bigram in bigram_counts.keys():
            bigram_counts[bigram] += 1
        else:
            bigram_counts[bigram] = 1

    vocab_counts = {}
    # Sliding through vocabs to get vocab counts
    for vocab in vocabs:
        for word in text_corpus:
            if word == vocab:
                if vocab in vocab_counts.keys():
                    vocab_counts[vocab] += 1
                else:
                    vocab_counts[vocab] = 1
    sentence_prob = 1
    # Sliding through the `tokens` to get bigrams
    for i in range(len(tokens) - 1):
        # Getting bigram at each slide
        bigram = (tokens[i], tokens[i+1])
        sentence_prob *= (bigram_counts.get(bigram, 0) + 1) / (vocab_counts.get(tokens[i]) + len(vocabs) + 1)
    return sentence_prob
```

```
compute_sentence_probability_smoothed_bigram(test_sentence, vocabs, text_corpus)
```

```
5.784626775880419e-06
```

Figure 30: Python implementation for smoothed bigram model

The figure above shows the python implementation for the unsmoothed bigram model. First, the **sentence** “<s> I read a book by Danielle </s>” is split into tokens, then the text corpus is looped through to get the counts of each possible bigram. Then, **vocabs** is also looped through to get the counts for each vocabulary in the **text_corpus**. Finally, the sentence probability is computed using the formula for the smoothed bigram model. The sentence probability is about 0.000005785, which is exactly the same as the manual computations.

Part B

Q1: Dataset and Exploratory Data Analysis (EDA)

The Problem and Literature Review

Our team has picked IMDB Dataset of 50K Movie Reviews as the dataset of this assignment. Our team got the dataset from the Kaggle platform, here is the link to the dataset: <https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews>.

The classification problem which is being solved in this assignment is sentiment analysis. Sentiment analysis is to find out what the emotion is behind every piece of text (that is every movie review in our case) (Raj, 2023). The chosen dataset is suitable for the defined problem because the labels of the dataset are either positive or negative, which aligns directly with our binary classification problem. Other than that, the dataset is also balanced over classes, that is 25000 movie reviews are labelled as positive, another 25000 are labelled as negative (Hazem, 2023). In other words, the dataset doesn't require extra steps to be handled such as using SMOTE, undersampling, oversampling etc., otherwise the trained machine learning model will produce bias results (Mazumder, 2023). Moreover, the dataset is the movie reviews from real-world, and this makes the dataset more practical and realistic.

This conference paper (Qaisar, 2020) proposed Long Short-Term Memory (LSTM) classifier to analyze the sentiments of the IMDB Dataset. Firstly, the dataset was split into train set and test set, where the train set is made up of 80% samples and the test set is made up of 20%. After splitting, the ratio of positive reviews to negative reviews was also maintained the same as in the original dataset. 10-fold cross validation was adopted to make the final classification results more convincing. For preprocessing the dataset, all the punctuations, hyperlinks, and stop words were removed first, then all letters were converted to lowercase letters, finally stemming techniques were applied. For the feature extraction on the dataset, *Doc2Vec* model from the Python library *gensim* was used to vectorize the text data. For the LSTM classifier, it was made up of 2 LSTM layers and 1 Dense layer. The first LSTM layer was setting 50 as the dimension of the hidden states, the second LSTM layer was setting 101 as the dimension of the hidden states, while

the last Dense layer consists of 2 neurons outputting the probabilities of each class. Adam optimizer was used by the LSTM classifier to optimize the loss function. To evaluate the performance of the classifier after training, metrics such as accuracy and confusion matrix were used. The test accuracy of the trained LSTM on the IMDB dataset was 89.9%, and the author proposed that the results could be further enhanced by including the concepts of ensemble learning.

According to the author, there is a more sophisticated way to analyze the sentiment on IMDB reviews (Wu, n.d). A variety of machine learning models, such as SVM, Naïve Bayes, and deep neural networks, may be used in this study. The study's rigorous data preparation and vectorization is its crucial component. The model's performance has been significantly affected by binary and 3-gram elements. This study demonstrates how crucial it is to accurately portray features and use appropriate models when doing sentiment analysis. The underlying assumption of this study is that our dataset can directly benefit from its conclusions. To validate the dataset's suitability for comparable classification tasks and the possibility of producing high-accuracy results, we used IMDB balanced movie reviews as our dataset.

In addition, research conducted by Maulana in 2020 improved the support vector machine (SVM) sentiment analysis of movie reviews by including the Information Gain (IG) feature selection method (Maulana, 2020). The study proved an increase in accuracy from 83.05% to 85.65% using the Cornell dataset. On the Stanford dataset, there was little improvement, going from 86.46% to 86.62%. Within the framework of this research article, the introduction puts sentiment analysis's significance in analyzing textual data for opinions in situations where IMDb reviews influence viewers' decisions into perspective. This research compared many kernels. Consequently, RBF proved to be the task's most successful solution. This research shows that information gain feature selection is a useful feature for support vector machines (SVM). As a result of its influence, sentiment categorization became more precise. The study's focus on the effects of feature selection on support vector machines (SVMs), one of the models used in our research, suggests that it may be relevant to our dataset. This research confirms the use of the IMDb dataset owing to its richness and promise, and it indicates that feature selection is vital in text categorization. Improving classification accuracy using sophisticated feature selection algorithms is a possibility.

In addition, we have also looked at Ramadhan's research (Ramadhan, 2021). This study's author used support vector machines (SVMs) to analyze the sentiment of IMDB movie reviews; the results showed an accuracy of 79%, precision of 75%, and recall of 87%. Compared to logistic regression, these outcomes were superior. While recognizing the difficulties of analyzing the many variants of IMDB comments, the article goes on to say that feature selection with SVM helped classification jobs a lot. It has been shown in this literature that SVM is well-suited for textual data categorization. Despite its limits in precision, this demonstrates the possibility for very accurate sentiment analysis. Having some distance from reality in repeated testing is thus suggested to lessen the constraint. Their methodology supports our choice of IMDB reviews for study when we attempt to connect this article to our dataset. The SVM's effectiveness for this classification type is shown by this technique. The paper's results back up our dataset choice, since it has shown that SVM can provide accurate results. Its appropriateness for our analytical goals is further supported by this.

Data Cleaning, EDA, Suggestions of Predictive Models

Before performing EDA, cleaning the IMDB dataset is a necessary action so that the statistical review of the dataset can be shown clearly. The cleaning of the dataset consists of the following steps:

- Convert all uppercases to lowercases.
- Expanding contractions.
- Remove HTML tags.
- Remove punctuation.
- Remove stop words.

```
def clean(review):  
    # all uppercases converted to lowercases  
    review = review.lower()  
  
    # Expanding contractions  
    expanded_words = []  
    for word in review.split():  
        expanded_words.append(contractions.fix(word))  
    review = ' '.join(expanded_words)  
  
    # remove html tags  
    review = re.sub("<[>]+>", "", review)  
    # remove punctuations  
    review = re.sub('[^a-z A-Z 0-9-]+', '', review)  
    # remove stopwords  
    review = " ".join([word for word in review.split() if word not in stopwords.words('english')])  
    return review
```

Figure 31: Cleaning the IMDB dataset

Finally, a cleaned dataset called ***cleaned_IMDB_Dataset.csv*** will be generated and it will be used to train the predictive models proposed by our team.

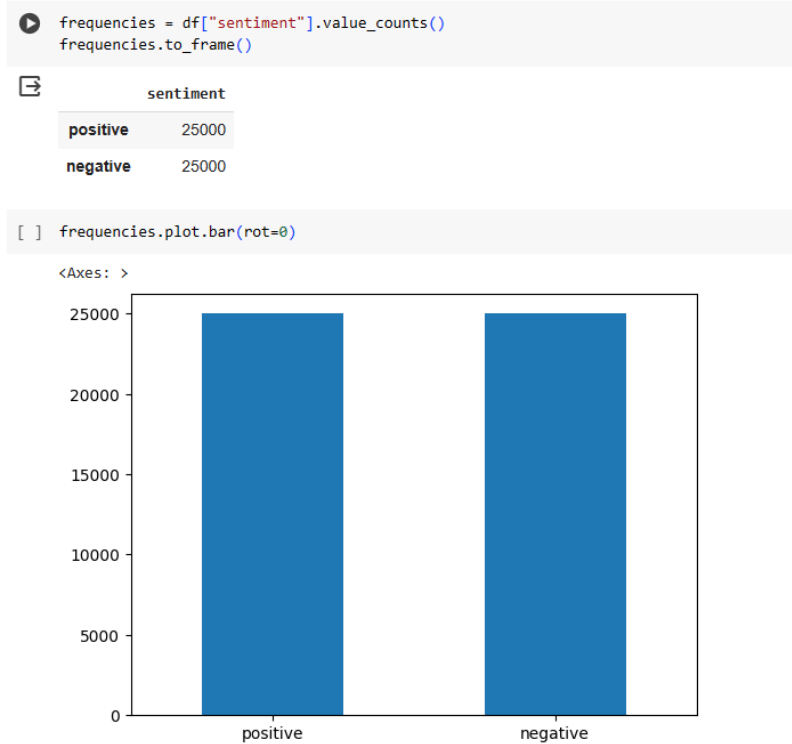


Figure 32: Distribution over classes

In the first step of EDA, the number of samples for each class is printed out and visualized using a bar chart. Based on the bar chart, the IMDB dataset is a balanced dataset.

```
# Number of words in each text/review
# Knowing this can define max sequence length
fig,(ax1,ax2)=plt.subplots(1,2,figsize=(12,8))
word_counts_pos = df[df['sentiment']=="positive"]['review'].str.split().map(lambda x: len(x))
print("Positive reviews: ")
print(word_counts_pos.describe())
ax1.hist(word_counts_pos,color='red')
ax1.set_title('Positive Reviews')

word_counts_neg = df[df['sentiment']=="negative"]['review'].str.split().map(lambda x: len(x))
print("Negative reviews: ")
print(word_counts_neg.describe())
ax2.hist(word_counts_neg,color='green')
ax2.set_title('Negative Reviews')
fig.suptitle('Frequency distribution of word counts in positive and negative reviews')
plt.show()
```

Figure 33: Codes to get word counts for each class

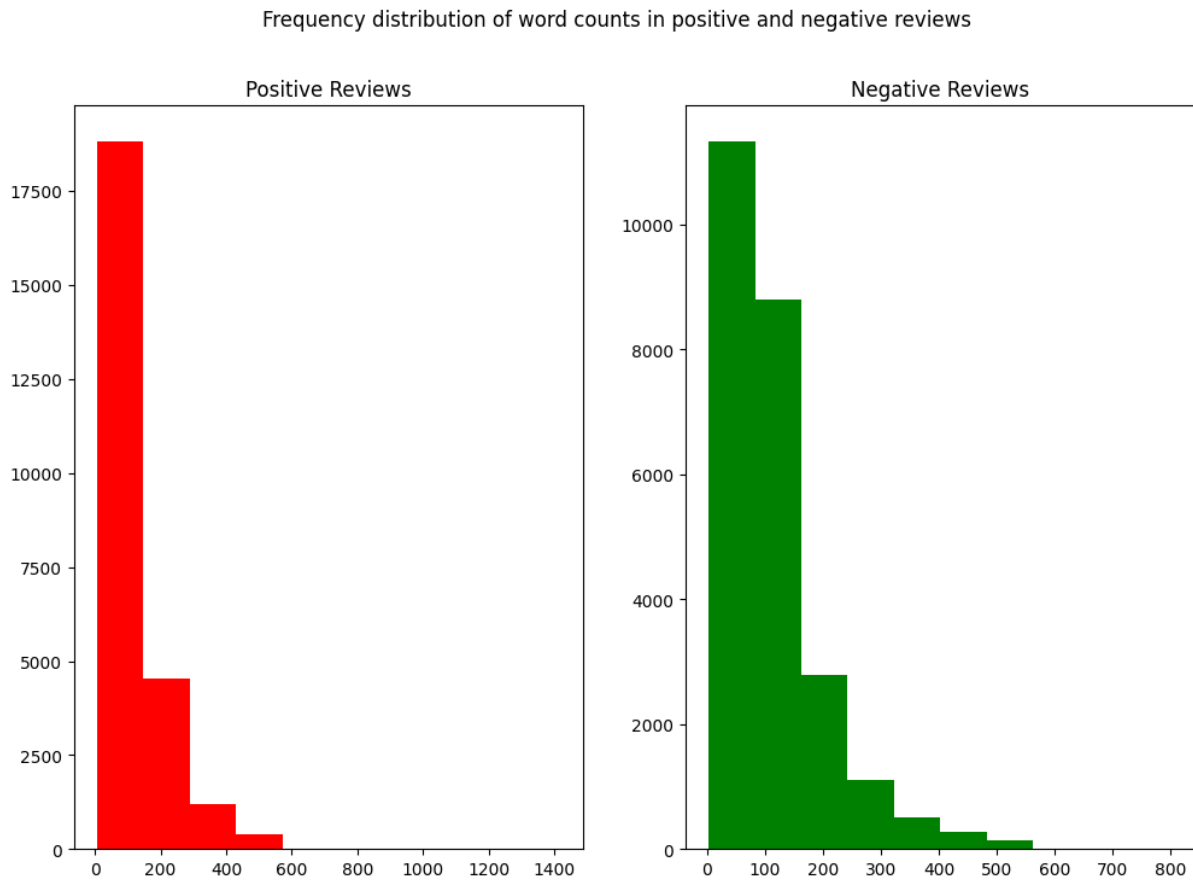


Figure 34: Frequency distribution of word counts in positive and negative reviews

Among the positive movie reviews, the histogram seems to be more spread out than negative movie reviews, which means the outliers (the maximum word count can go up to 1400) would be more common among positive movie reviews. For positive reviews, most movie reviews consist of less than around 150 words, which is about 20000. But for negative reviews, most movie reviews consist of less than 100 words, which is about 11000.

✓ Read the dataset

```
[ ] data = pd.read_csv(file_name)
    data['sentiment'] = data['sentiment'].map({'positive': 1, 'negative': 0})
```

✓ Vectorization

```
[ ] tfidf = TfidfVectorizer(stop_words='english', max_features=1000)
    features = tfidf.fit_transform(data['review']).toarray()
    labels = data['sentiment']
```

Figure 35

```
[24] word_counts = pd.DataFrame(features, columns=tfidf.get_feature_names_out()).sum().sort_values(ascending=False)
    top_words = word_counts.head(10)

    plt.figure(figsize=(10,6))
    top_words.plot(kind='bar')
    plt.title('Top 10 Most Frequent Words')
    plt.xlabel('Words')
    plt.ylabel('Word Count')
    plt.show()
```

Figure 36: Plotting barchart for top 10 most frequent words

```
[24] word_counts = pd.DataFrame(features, columns=tfidf.get_feature_names_out()).sum().sort_values(ascending=False)
      top_words = word_counts.head(10)

      plt.figure(figsize=(10,6))
      top_words.plot(kind='bar')
      plt.title('Top 10 Most Frequent Words')
      plt.xlabel('Words')
      plt.ylabel('Word Count')
      plt.show()
```

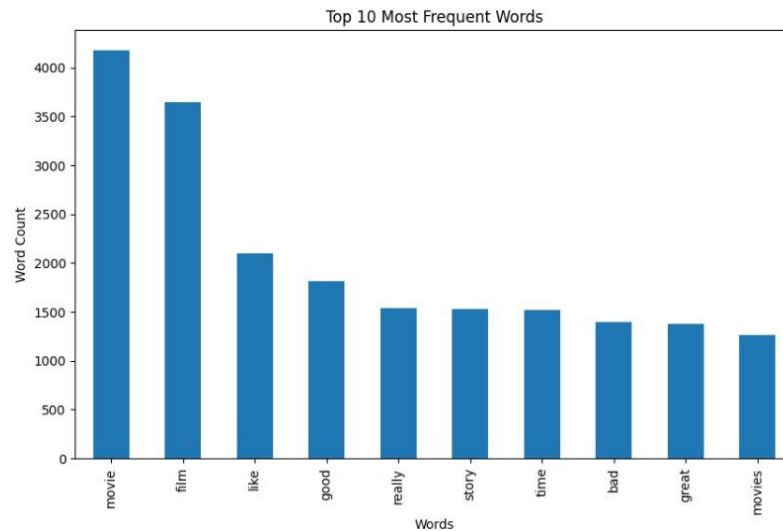


Figure 37: Barchart of top 10 most frequent words

Other than that, the top 10 most frequent words inside the dataset are also visualized. The most frequent words are “movie”, followed by “film”, followed by “like”, then all the way until the 10-th most frequent word “movies” according to the bar chart.

After performing the cleaning and EDA on the IMDB dataset, 4 predictive models are suggested by our team. Two of the predictive models are from traditional machine learning field, while the other two are from the deep learning field. This is normally when it comes to solving machine learning tasks, the models to be experimented with should start from the least complicated to the most complicated. The following predictive models are proposed:

- Naïve Bayes
- Support Vector Machine (SVM)
- Convolutional Neural Network (CNN)
- Long Short-Term Memory (LSTM)

Q2: Supervised Text Classification

LSTM: Low Sim Chuan

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

[ ] import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import string
import math
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import LSTM,Dense,Input
from tensorflow.keras.layers import Flatten, Embedding, LSTM
from tensorflow.keras.models import Model, Sequential
import keras_tuner as kt

[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Package stopwords is already up-to-date!

[ ] path_ds = "/content/drive/MyDrive/TXSA_Assignment/cleaned_IMDB_Dataset.csv"
df = pd.read_csv(path_ds)
print(df.shape)

(50000, 2)
```

Figure 38: Mounting Google Drive, importing libraries, loading dataset

Firstly, Google Drive is mounted to the colab file so that the colab file can access the dataset inside the Google Drive. Then, all the required libraries are imported. Then, the file path to the dataset is stated and the dataset is loaded. The shape of the dataset is printed out to inspect if the dataset has been loaded successfully.

```
[ ] df.sentiment.replace("positive", 1, inplace = True)
    df.sentiment.replace("negative", 0, inplace = True)
    X = df.review
    y = df.sentiment

# `stratify` to ensure that both the train and test sets have the proportion of examples in each class that is present in the provided "y" array
x_train, x_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.20,
    random_state= 0,
    stratify= y
)
print('X_train shape --> ', x_train.shape)
print('y_train shape --> ', y_train.shape)
print('X_test shape --> ', x_test.shape)
print('y_test shape --> ', y_test.shape)

X_train shape --> (40000,)
y_train shape --> (40000,)
X_test shape --> (10000,)
y_test shape --> (10000,)
```

Figure 39: Data preprocessing - Part 1

The original dataset uses texts “positive” and “negative” to label the reviews, and they are converted to corresponding integer values, 1 and 0, since LSTM model cannot understand text data. Then, the dataset is split into train set and test set, where 80% of the samples are to form the train set while 20% of the samples are to form the test set. Notice that the *stratify* parameter is to ensure that the train set, and test set have the same proportion of samples in each class as the original dataset. Lastly, the shapes of the resulting train features, train labels, test features, and test labels are all printed out to inspect if the splitting has been done successfully.

```
max_sequence_len = 150
vocab_num = 5000

word_tokenizer = Tokenizer(num_words=vocab_num, oov_token="<OOV>")
word_tokenizer.fit_on_texts(x_train)

X_train = word_tokenizer.texts_to_sequences(x_train)
X_test = word_tokenizer.texts_to_sequences(x_test)

X_train = pad_sequences(X_train, padding='post', maxlen=max_sequence_len)
X_test = pad_sequences(X_test, padding='post', maxlen=max_sequence_len)

import matplotlib.pyplot as plt
def plot_accuracies_losses(history, optimizer_name):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])

    plt.title('Model accuracy using optimizer ' + optimizer_name)
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])

    plt.title('Model loss using optimizer ' + optimizer_name)
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
```

Figure 40: Data preprocessing - Part 2

The maximum sequence length is set to 150 to ensure that each review has uniform length, so that they can be fed into LSTM. Other than that, the vocabulary number is set to 5000 to ensure the LSTM know 5000 most frequent vocabularies to determine whether a movie review is positive or negative. The ***Tokenizer*** object is instantiated and used to keep 5000 most frequent words in the train set, and it will replace a word that it has never encountered before with a special <OOV> (Out-Of-Vocabulary) token. Then, the tokenizer is fitted on both train set, and test set to convert the texts to sequences of not uniform lengths. Then, the ***pad_sequences()*** function is called on the train sequences and test sequences to ensure that they have uniform length before feeding them into LSTM model, by the operations padding and truncating the sequences according to the maximum sequence length set before.

```
def model_builder_sgd(hp):  
    hp_embedding_dim = hp.Int("embedding_dim", min_value=300, max_value=500, step=100)  
    hp_hidden_unit_dim = hp.Int("hidden_unit_dim", min_value=32, max_value=128, step=32)  
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])  
    hp_momentum = hp.Choice("momentum", values=[0.0, 0.5, 0.9, 0.99])  
  
    model = Sequential()  
    model.add(Embedding(vocab_num, hp_embedding_dim, input_length=max_sequence_len))  
    model.add(LSTM(hp_hidden_unit_dim))  
    model.add(Dense(1, activation='sigmoid'))  
    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(  
        hp_learning_rate,  
        decay_steps=100000,  
        decay_rate=0.96,  
        staircase=True  
    )  
    sgd_opt = keras.optimizers.SGD(  
        learning_rate=lr_schedule,  
        momentum=hp_momentum  
    )  
  
    model.compile(  
        optimizer=sgd_opt,  
        loss="binary_crossentropy",  
        metrics=['accuracy']  
    )  
  
    return model
```

Figure 41: Defining model builder for the LSTM

The architecture of the LSTM model is defined in the function ***model_builder_sgd(hp)***. First, all the ranges of the hyperparameters of the LSTM are defined: range of embedding dimension in Embedding layer, range of hidden unit dimension of LSTM layer, range of learning rate, and range of momentum in the SGD optimizer. The embedding dimension could range from 300 to 500 in steps of 100, the hidden unit number could range from 32 to 128 in steps of 32, the learning rate could be 0.01, 0.001, or 0.0001, and the momentum could be 0.0, 0.5, 0.9 or 0.99. Then, the model architecture is defined. The Embedding layer is first added to the model for vectorizing the input sequences. LSTM layer is then added to the model. Lastly, the Dense layer of 1 unit is added to the model with sigmoid function as the activation function. After defining the model architecture, the learning rate scheduler is created using exponential decay for faster convergence, technically the learning rate will decay to 96% of its previous value every 100000

steps. Then, the Stochastic Gradient Descent (SGD) optimizer is initialized with the learning rate scheduler and the range of momentum. Lastly, the model is compiled with the optimizer, binary cross entropy as the loss function, and the accuracy as the evaluation metric.

```
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
tuner = kt.GridSearch(
    model_builder_sgd,
    objective='val_accuracy',
    directory='GridSearchResults',
    project_name='LSTM_SGD',
)

EPOCH_NUM = 30

tuner.search(X_train, y_train, epochs=EPOCH_NUM, validation_data=(X_test, y_test), callbacks=[stop_early])

Trial 10 Complete [00h 07m 50s]
val_accuracy: 0.5077999830245972

Best val_accuracy So Far: 0.8790000081062317
Total elapsed time: 01h 02m 12s

Search: Running Trial #11

Value          |Best Value So Far|Hyperparameter
300             |300              |embedding_dim
32              |32               |hidden_unit_dim
0.0001          |0.01             |learning_rate
0.9             |0.99             |momentum

Epoch 1/30
1250/1250 [=====] - 16s 12ms/step - loss: 0.6937 - accuracy: 0.4922 - val_loss: 0.6932 - val_accuracy: 0.4983
Epoch 2/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6933 - accuracy: 0.4933 - val_loss: 0.6932 - val_accuracy: 0.4998
Epoch 3/30
1250/1250 [=====] - 17s 13ms/step - loss: 0.6933 - accuracy: 0.4971 - val_loss: 0.6931 - val_accuracy: 0.5004
Epoch 4/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6932 - accuracy: 0.4979 - val_loss: 0.6931 - val_accuracy: 0.5006
Epoch 5/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6932 - accuracy: 0.5026 - val_loss: 0.6931 - val_accuracy: 0.5045
Epoch 6/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6931 - accuracy: 0.5042 - val_loss: 0.6931 - val_accuracy: 0.5052
Epoch 7/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6931 - accuracy: 0.5071 - val_loss: 0.6931 - val_accuracy: 0.5020
Epoch 8/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6931 - accuracy: 0.5080 - val_loss: 0.6931 - val_accuracy: 0.5035
Epoch 9/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6930 - accuracy: 0.5076 - val_loss: 0.6931 - val_accuracy: 0.5040
Epoch 10/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6930 - accuracy: 0.5081 - val_loss: 0.6931 - val_accuracy: 0.5031
Epoch 11/30
1250/1250 [=====] - 15s 12ms/step - loss: 0.6930 - accuracy: 0.5085 - val_loss: 0.6931 - val_accuracy: 0.5029
Epoch 12/30
1250/1250 [=====] - ETA: 0s - loss: 0.6930 - accuracy: 0.5082
```

Figure 42: Start training and hyperparameter tuning

The early stopping mechanism is also included in the LSTM model is stop the training if the model is not improving for 5 epochs in terms of validation loss, and this is to prevent overfitting. Then, grid search is applied to search for the best set of hyperparameters based on the defined ranges of the hyperparameters inside the function *model_builder_SGD(hp)*. Then, the maximum epoch number is set for the LSTM model, which is 30.

Best set of hyperparameters for SGD optimizer:

```
# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The hyperparameter search is complete.
The optimal dimension of embedding vector is {best_hps.get('embedding_dim')}
The optimal dimension of hidden units in LSTM layer is {best_hps.get('hidden_unit_dim')}
The optimal learning rate for the optimizer is {best_hps.get('learning_rate')}
The optimal momentum for the optimizer is {best_hps.get('momentum')}
""")
```

```
The hyperparameter search is complete.
The optimal dimension of embedding vector is 300
The optimal dimension of hidden units in LSTM layer is 64
The optimal learning rate for the optimizer is 0.01
The optimal momentum for the optimizer is 0.99
```

Figure 43: Getting the optimal set of hyperparameters

After applying grid search, the best set of hyperparameters is printed out.

```

model = Sequential()
model.add(Embedding(vocab_num, 300, input_length=max_sequence_len))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    0.01,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=True
)
sgd_opt = keras.optimizers.SGD(
    learning_rate=lr_schedule,
    momentum=0.99
)

model.compile(
    optimizer=sgd_opt,
    loss="binary_crossentropy",
    metrics=['accuracy']
)

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)

EPOCH_NUM = 30

history = model.fit(X_train, y_train, epochs=EPOCH_NUM, validation_data=(X_test, y_test), callbacks=[stop_early])

Epoch 1/30
1250/1250 [=====] - 43s 32ms/step - loss: 0.6947 - accuracy: 0.5031 - val_loss: 0.6969 - val_accuracy: 0.5000
Epoch 2/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6955 - accuracy: 0.5019 - val_loss: 0.6938 - val_accuracy: 0.5000
Epoch 3/30
1250/1250 [=====] - 13s 11ms/step - loss: 0.6949 - accuracy: 0.5013 - val_loss: 0.6922 - val_accuracy: 0.5227
Epoch 4/30
1250/1250 [=====] - 17s 13ms/step - loss: 0.6935 - accuracy: 0.5153 - val_loss: 0.6904 - val_accuracy: 0.5133
Epoch 5/30
1250/1250 [=====] - 17s 14ms/step - loss: 0.6914 - accuracy: 0.5206 - val_loss: 0.7046 - val_accuracy: 0.5248
Epoch 6/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6855 - accuracy: 0.5329 - val_loss: 0.7019 - val_accuracy: 0.5162
Epoch 7/30
1250/1250 [=====] - 12s 10ms/step - loss: 0.6780 - accuracy: 0.5389 - val_loss: 0.7004 - val_accuracy: 0.5138
Epoch 8/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.6746 - accuracy: 0.5415 - val_loss: 0.6810 - val_accuracy: 0.5295
Epoch 9/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.6644 - accuracy: 0.5536 - val_loss: 0.6871 - val_accuracy: 0.5249
Epoch 10/30
1250/1250 [=====] - 10s 8ms/step - loss: 0.6435 - accuracy: 0.5683 - val_loss: 0.6828 - val_accuracy: 0.5545
Epoch 11/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.4550 - accuracy: 0.7619 - val_loss: 0.3632 - val_accuracy: 0.8426
Epoch 12/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.2655 - accuracy: 0.8908 - val_loss: 0.3017 - val_accuracy: 0.8774
Epoch 13/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.2195 - accuracy: 0.9129 - val_loss: 0.3089 - val_accuracy: 0.8784
Epoch 14/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.1916 - accuracy: 0.9263 - val_loss: 0.3199 - val_accuracy: 0.8774
Epoch 15/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.1647 - accuracy: 0.9360 - val_loss: 0.3468 - val_accuracy: 0.8739
Epoch 16/30
1250/1250 [=====] - 10s 8ms/step - loss: 0.1430 - accuracy: 0.9462 - val_loss: 0.3745 - val_accuracy: 0.8698
Epoch 17/30
1250/1250 [=====] - 11s 9ms/step - loss: 0.1305 - accuracy: 0.9526 - val_loss: 0.4176 - val_accuracy: 0.8647

```

Figure 44: Rebuilding and retraining the LSTM with the best set of hyperparameters

The LSTM model is rebuilt and retrained with the best set of hyperparameters. Early stopping mechanism is applied as usual to prevent overfitting. Finally, the training stopped at epoch 17 out of 30.

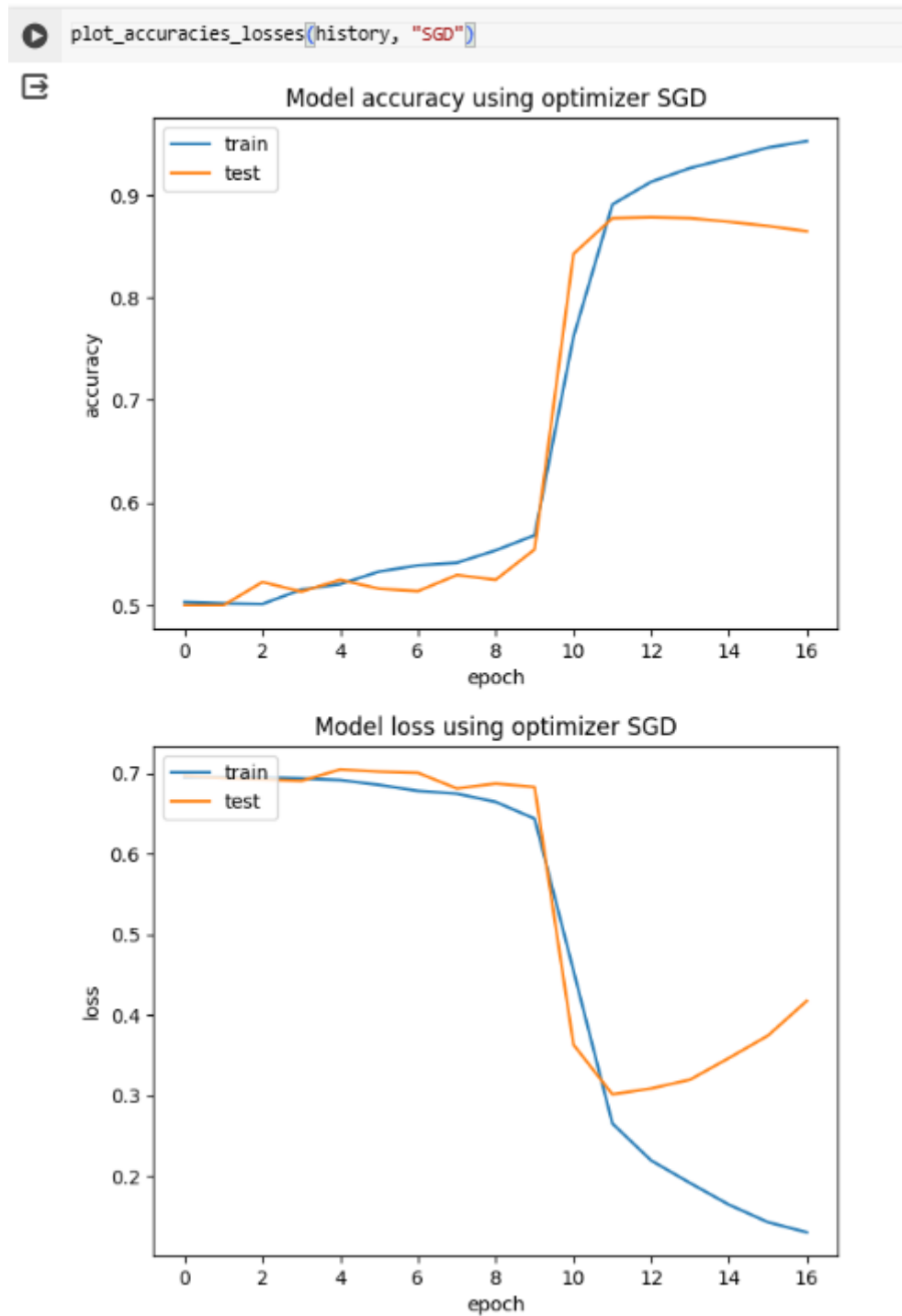


Figure 45: Plotting the training results and testing results in terms of loss and accuracy

The training results and the testing results are plotted using *matplotlib* library, in terms of the loss and the accuracy.

```
[24] y_pred_train = model.predict(X_train)
      y_pred_train = np.where(y_pred_train >= 0.5, 1, 0)
      y_pred_test = model.predict(X_test)
      y_pred_test = np.where(y_pred_test >= 0.5, 1, 0)

1250/1250 [=====] - 6s 4ms/step
313/313 [=====] - 2s 5ms/step
```

Figure 46: Predictions on train set and test set

```
# Evaluate the model
train_loss, train_acc = model.evaluate(X_train, y_train, verbose=2)
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print("Train accuracy: {:.2f}%".format(100 * train_acc))
print("Test accuracy: {:.2f}%".format(100 * test_acc))

1250/1250 - 5s - loss: 0.0908 - accuracy: 0.9695 - 5s/epoch - 4ms/step
313/313 - 1s - loss: 0.4176 - accuracy: 0.8647 - 1s/epoch - 4ms/step
Train accuracy: 96.95%
Test accuracy: 86.47%
```

Figure 47: Getting train accuracy and test accuracy

Then, the model is used to predict on both train set and test set to get the losses and accuracies. The train accuracy and the test accuracy are printed out in terms of percentage.

```
[42] def plot_confusion_matrix(actual, predicted, ds_type):
      cm = tf.math.confusion_matrix(actual, predicted)
      ax = sns.heatmap(cm, annot=True, fmt='g')
      sns.set(rc={'figure.figsize':(5, 5)})
      sns.set(font_scale=1.0)
      ax.set_title('Confusion matrix of sentiment analysis for ' + ds_type)
      ax.set_xlabel('Predicted Sentiment')
      ax.set_ylabel('Actual Sentiment')
      plt.xticks(rotation=90)
      plt.yticks(rotation=0)
```

Figure 48: Defining function for plotting confusion matrix

The function to plot the confusion matrix is defined.

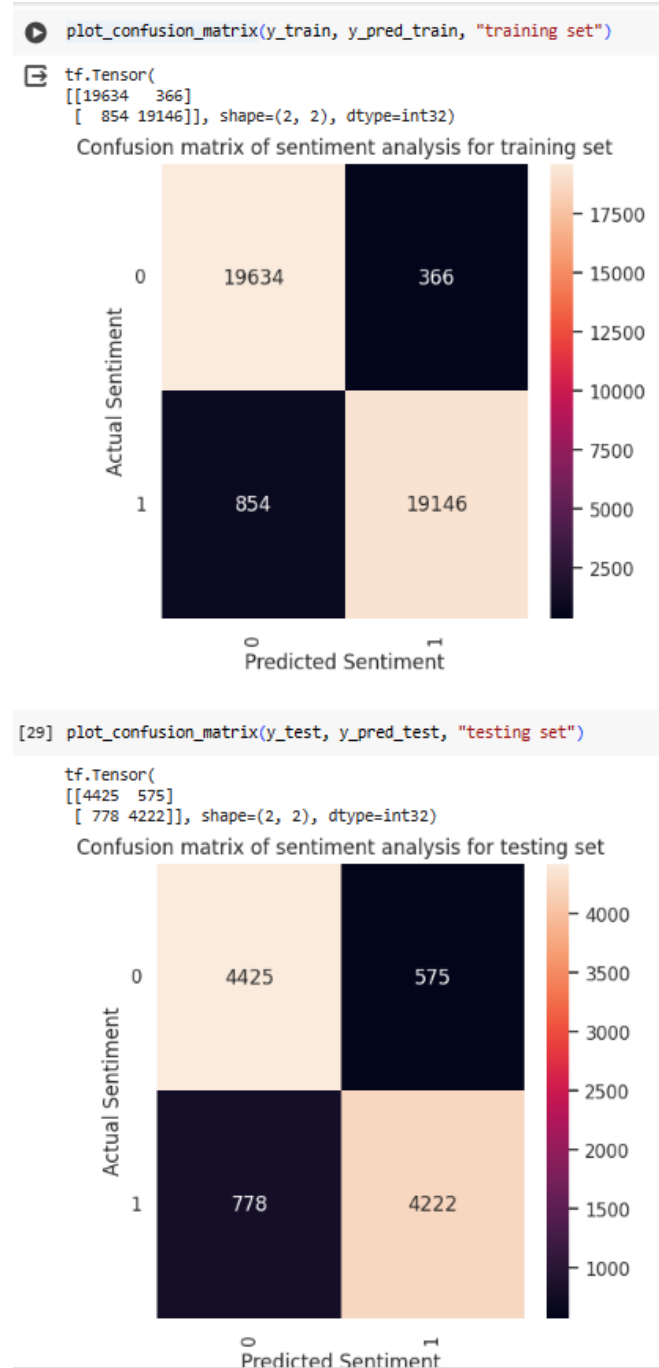


Figure 49: Confusion matrix for train and test sets

Finally, the performance of the trained LSTM model on both train set and test set are evaluated using the confusion matrix. From the confusion matrices, for each class it can be seen how many samples are correctly classified and how many samples are misclassified.

SVM – LinearSVC: Nivethan

Importing libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from google.colab import files
```

Figure 50

All the required modules and libraries are imported into this data cell. For data management, there are pandas, and for machine learning tasks like model selection, there are several modules from 'sklearn'. Moreover, for the sake of visualization, libraries like "matplotlib" and "seaborn" are being loaded. Finally, file operations in a Google Colab context are handled by the 'google.colab' module.

Upload the file

```
uploaded_files = files.upload()
file_name = next(iter(uploaded_files))
```

No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving cleaned_IMDB_Dataset.csv to cleaned_IMDB_Dataset (1).csv

Figure 51

When working in Google Colab, this cell is where you will submit your files. The name of the uploaded file is saved in this box for future use.

Read the dataset

```
data = pd.read_csv(file_name)
data['sentiment'] = data['sentiment'].map({'positive': 1, 'negative': 0})
```

Figure 52

This cell would load the pandas DataFrame with the contents of the CSV file. Additionally, it would convert the values in the 'sentiment' column from strings ('positive', 'negative') to integers (1,0). When it comes to binary classification jobs, this is the standard procedure.

Vectorization

```
] tfidf = TfidfVectorizer(stop_words='english', max_features=1000)
features = tfidf.fit_transform(data['review']).toarray()
labels = data['sentiment']
```

Figure 53

The 'TfidfVectorizer' is used to transform text data into a TF-IDF feature matrix in this case. The purpose of these characteristics is to restrict the number of features to 1000 while ignoring English stop words. For machine learning purposes, this effectively converts the 'review' column of the DataFrame into a numerical representation.

Split the dataset

```
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)
```

Figure 54

There are two sets of data in the source dataset which are training and test. We have set aside 20% of the data for testing purposes. To test how well the model performs on new data, this is done.

Initialize the model

```
| svm = LinearSVC(random_state=42)
```

Figure 55

The cell above shows that the instance of the 'LinearSVC' model is created with a fixed random state for reproducibility.

Set up the hyperparameter search

```
param_distributions = {  
    'C': [0.001, 0.01, 0.1, 1, 10], # Example range, can include more  
    'max_iter': [1000, 2000, 3000] # Example range, can include more  
}
```

Figure 56

A range of values for the 'C' parameter and 'max_iter' may be specified in the parameter distributions for a randomized search, as seen in the cell above. In this context, "max_iter" refers to the maximum number of iterations, while "C" indicates the regularization strength.

Perform Random Search

```
] random_search = RandomizedSearchCV(  
    svm, param_distributions, n_iter=10, cv=5, random_state=42, n_jobs=-1  
)
```

Figure 57

The 'LinearSVC' model is set up using the 'RandomizedSearchCV' object. The model includes a fixed random state, tasks to use all the available cores, a distribution of parameters, several search iterations, cross-validation folds, and more.

Fit the random search model to the training data

```
] random_search.fit(X_train, y_train)
```

```
graph TD; subgraph DashedBox [ ]; direction TB; A[RandomizedSearchCV]; B[estimator: LinearSVC]; end; C[LinearSVC] --- B;
```

Figure 58

The cell above shows that this line fits the model on the training data using the randomized search to find the best hyperparameters.

Best estimator



```
best_svm = random_search.best_estimator_
```

Print Accuracy and Classification Report

```
print("Best hyperparameters:", random_search.best_params_)
# Predictions on the training set
y_train_pred = best_svm.predict(X_train)

# Predictions on the test set
y_test_pred = best_svm.predict(X_test)

# Calculate accuracy on the training set
train_accuracy = accuracy_score(y_train, y_train_pred)
print("Training Set Accuracy:", train_accuracy)

# Calculate accuracy on the test set
test_accuracy = accuracy_score(y_test, y_test_pred)
print("Test Set Accuracy:", test_accuracy)

# Print Classification Report for the test set
print(classification_report(y_test, y_test_pred))
```

```
Best hyperparameters: {'max_iter': 3000, 'C': 0.1}
Training Set Accuracy: 0.867
Test Set Accuracy: 0.8612
precision    recall  f1-score   support
```

Figure 59

	precision	recall	f1-score	support
0	0.87	0.84	0.86	4961
1	0.85	0.88	0.86	5039
accuracy			0.86	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.86	0.86	0.86	10000

Figure 60

You can see that the 'best_svm,' the top SVM model, gets retrieved from the randomized search in the cells up there. After that. The hyperparameters that work best are written out. Then, the accuracy scores and predictions from the training and test sets are computed. The next step is to generate a classification report that displays the specific performance data, including the F1-score, recall, and accuracy.

```
# Confusion Matrix Plot
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Negative', 'Positive'], yticklabels=['Negative', 'Positive'])
plt.title('Confusion Matrix')
plt.ylabel('Actual Values')
plt.xlabel('Predicted Values')
plt.show()
```

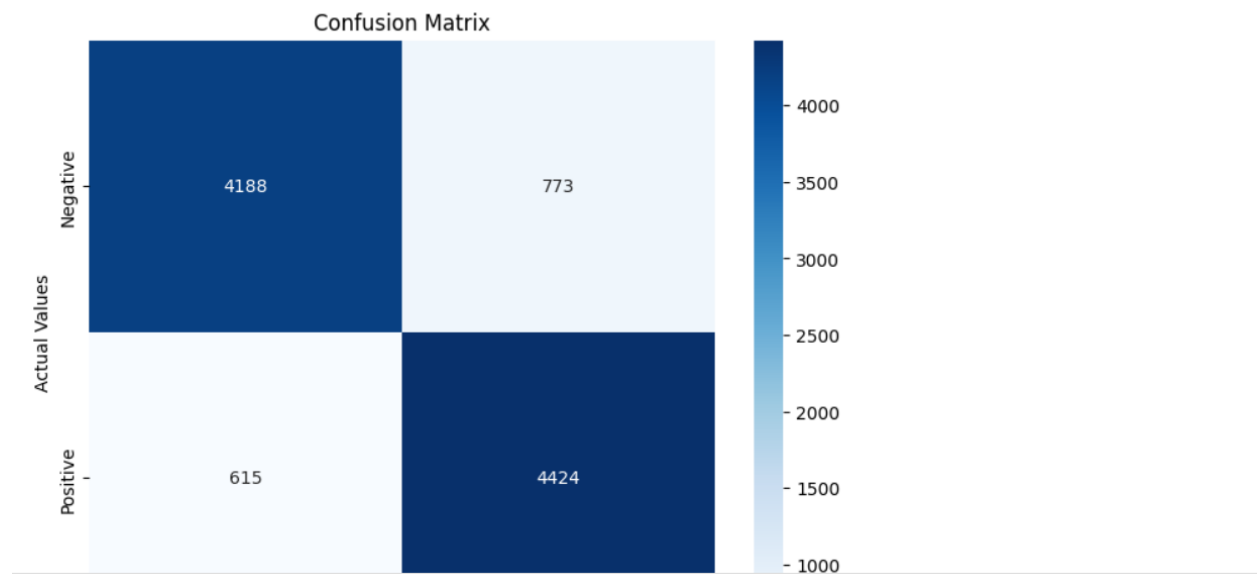


Figure 61

The cell above shows that the confusion matrix is plotted for the test set predictions using the 'matplotlib' and 'seaborn'. In addition, the matrix provides a visual representation of the model's performance by comparing the actual values to the predicted values for both classes which are ('Negative', 'Positive').

One Dimensional Convolutional Neural Network – Liaw Yu Jay

```
[ ] from google.colab import drive
drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

[ ] import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import string
import math
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from sklearn.model_selection import train_test_split

[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Package stopwords is already up-to-date!

path_ds = "/content/drive/MyDrive/Colab Notebooks/Text Analysis and Sentiment Analysis/Assignment/cleaned_IMDB_Dataset.csv"
df = pd.read_csv(path_ds)

print(df.shape)
```

Figure 62

The figure shows the first step in this project which is mounting the colab file to the google drive so that it can access the dataset which is stored in the google drive. All the necessary libraries are imported and the dataset is loaded from the google drive based on the file path provided. The shape of the dataset is checked and printed out.

```
[ ] frequencies = df["sentiment"].value_counts()
frequencies.to_frame()
```

sentiment		
positive	25000	
negative	25000	

Figure 63

The distribution of the dataset is checked to ensure that the dataset is not an imbalanced dataset.

```

from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import regularizers
from tensorflow import keras
from kerastuner import GridSearch
from kerastuner.engine.hyperparameters import HyperParameters
from sklearn.model_selection import RandomizedSearchCV

[ ] df.sentiment.replace("positive", 1, inplace = True)
df.sentiment.replace("negative", 0, inplace = True)
X = df.review
y = df.sentiment

[ ] x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state= 0, stratify= y)
print('X_train shape --> ', x_train.shape)
print('y_train shape --> ', y_train.shape)
print('X_test shape --> ', x_test.shape)
print('y_test shape --> ', y_test.shape)

X_train shape --> (40000,)
y_train shape --> (40000,)
X_test shape --> (10000,)
y_test shape --> (10000,)

```

Figure 64

The figure above shows that some other important libraries are also imported. The labels of dataset that are originally positive and negative are replaced with 1 and 0 as text data set cannot be understood by the model. Then, the dataset is split into 20 percent for test set and 80 percent for train set. The output of the datasets that are split are printed to ensure the correct splitting ratio is applied.

```

[ ] word_tokenizer = Tokenizer()
word_tokenizer.fit_on_texts(x_train)

X_train = word_tokenizer.texts_to_sequences(x_train)
X_test = word_tokenizer.texts_to_sequences(x_test)

[ ] vocab_length = len(word_tokenizer.word_index) + 1

vocab_length

168143

[ ] max_sequence_len = 100

X_train = pad_sequences(X_train, padding='post', maxlen=max_sequence_len)
X_test = pad_sequences(X_test, padding='post', maxlen=max_sequence_len)

```

Figure 65

The code above shows the code for performing text preprocessing and sequence padding for training the model in textual data using the Tokenizer() function where the internal vocabulary of the function will be updated based on the text data. Then, the text data is converted into word indexes using the texts_to_sequences() function. The vocabulary length of the text dataset is

identified and shown. The `pad_sequences()` function is used to ensure that all sequences in `X_train` and `X_test` are the same length (`max_sequence_len`). Padding is applied at the conclusion of each sequence with zeros (`padding="post"`). This step ensures that all sequences have the same dimensions, which is required when feeding the data into a neural network model.

```
from gensim.scripts.glove2word2vec import glove2word2vec

# Load GloVe word embeddings and create an Embeddings Dictionary

from numpy import asarray
from numpy import zeros

embeddings_dictionary = dict()

glove_file = open('/content/drive/MyDrive/Colab Notebooks/Text Analysis and Sentiment Analysis/Assignment/glove.6B/glove.6B.300d.txt', encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary[word] = vector_dimensions
glove_file.close()
```

Figure 66

The code above loads the pre-trained word embeddings from the Glove model and creates an embeddings dictionary. To handle the Glove file, it begins by importing the necessary libraries and the `glove2word2vec` function from `gensim.scripts.glove2word2vec`. The word embeddings are initially stored in an empty dictionary called `embeddings_dictionary`. The function iterates through each line after opening the Glove file. To extract the word and the matching vector dimensions, it parses each line. With the word serving as the key, the vector dimensions are transformed into a NumPy array and saved in the embeddings dictionary. The Glove file is closed at last. The word embeddings, which link each word to a vector that depicts its semantic meaning in a high-dimensional space, are contained in the resultant `embeddings_dictionary`.

```
# Create Embedding Matrix having 300 columns
# Containing 300-dimensional GloVe word embeddings for all words in our corpus.

embedding_matrix = zeros((vocab_length, 300))
for word, index in word_tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix[index] = embedding_vector
embedding_matrix.shape
```

(168143, 300)

Figure 67

The code above creates an embedding matrix for words in the corpus using pre-trained GloVe word embeddings. Initially, an empty matrix called `embedding_matrix` is initialized with dimensions (`vocab_length`, 300), where 300 denotes the dimensionality of the GloVe word embeddings and `vocab_length` is the total number of unique words in the corpus. The word index mappings discovered from the training data are stored in the word tokenizer's word index dictionary, which is iterated over for every word and its matching index. It gets the GloVe word embedding vector for every word from the previously made `embeddings_dictionary`. The word's index determines which row in the embedding matrix the relevant embedding vector belongs to if the word is present in the embeddings dictionary. The row in the embedding matrix stays a zero vector if the term is not present in the embeddings dictionary. The `embedding_matrix` obtained from this step will be used for initialization process later at the One Dimensional Convolutional Neural Network. From the output it can be seen that the `embedding_matrix` has 168,143 rows and 300 columns.

```
# Define the model creation function
def create_model(filters=32, kernel_size=7, optimizer='adam', dropout_rate=0.2):
    model = Sequential()
    model.add(layers.Embedding(vocab_length, 300, weights=[embedding_matrix], input_length=max_sequence_len, trainable=False))
    model.add(layers.Conv1D(filters, kernel_size, activation='relu'))
    model.add(layers.MaxPooling1D(5))
    model.add(layers.Dropout(dropout_rate))
    model.add(layers.Conv1D(filters, kernel_size, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dense(1, activation='sigmoid'))
    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Figure 68

The code above shows the `create_model` function which creates the One Dimensional Convolutional Neural Network for the text classification process. Several parameters such as filters, kernel size, optimizer and dropout_rate are taken in by the function. In the function, a sequential model is initialized, where the first layer is an Embedding layer that takes `vocab_length` as input and `embedding_matrix` created above as initial weights. *The maximum length of input sequences, `max_sequence_len`, is the value assigned to `input_length`.* In order to prevent the embedding layer from updating the word embeddings during training, the trainable option is set to False. Then, a Conv1D layer is added, where `kernel_size` represents the size of the convolutional kernels and `filters` is the number of filters. 'relu' is the activation function that is employed. The next layer is called MaxPooling1D, and it runs max pooling across a size 5 window. By reducing the dimensionality of the feature maps, this technique extracts the most significant features. A Dropout

layer is inserted after the pooling layer to reduce overfitting. During training, it arbitrarily sets a portion of the input units to 0, which aids in preventing the model from depending too much on particular features. A second layer called Conv1D is added, followed by a GlobalMaxPooling1D layer. The most crucial data from the preceding levels is extracted and captured in these layers, along with additional features. The last layer is a Dense layer with a 'sigmoid' activation function and a single unit. It generates an output for binary classification that shows the probability of a positive class. Finally, the model is compiled with the specified optimizer, 'binary_crossentropy' as the loss function for binary classification, and 'accuracy' as the evaluation metric.

```
## Create the Keras classifier
model = KerasClassifier(build_fn=create_model)
```

Figure 69

A KerasClassifier is supplied with the create_model function above to build the One-Dimensional Convolutional Neural Network classifier and later stored in the variable model.

```
[ ] # Define the hyperparameters grid
param_grid = {
    'filters': [16, 32, 64],
    'kernel_size': [3, 5, 7],
    'batch_size': [32, 64, 128],
    'optimizer': ['adam', 'sgd'],
}
```

Figure 70

The figure above shows the creation of param_grid dictionary that specifies the hyperparameter grid for hyperparameter tuning in the next stage. For the filters hyperparameter that represents the number of filters in the convolutional layers a range of 16, 32 and 64 is defined. For the kernel_size' hyperparameter that represents the size of the convolutional kernel a range of 3, 5, and 7 is defined. For the 'batch_size' hyperparameter that represents the number of samples processed in each training batch range of 32, 64, and 128 is defined. Lastly, the hyperparameter optimizer which is the optimizer algorithm for model training is set to adam and sgd.


```
# Define the early stopping callback

early_stopping = EarlyStopping(monitor='loss', patience=3)
```

Figure 71

An early stopping callback is defined to show that the model training can be stopped early. The early stopping features will monitor the loss value during the training and stop the training after the loss does not improve after three epochs.

```
[ ] # Create a RandomizedSearchCV object
random_search = RandomizedSearchCV(model, param_grid, cv=3, n_iter=10, random_state=42)

# Fit the random search with early stopping
random_result = random_search.fit(X_train, y_train, callbacks=[early_stopping])
```

209/209 [=====] - 1s 3ms/step - loss: 0.6586 - accuracy: 0.6386
417/417 [=====] - 3s 4ms/step - loss: 0.6544 - accuracy: 0.6089
209/209 [=====] - 1s 3ms/step - loss: 0.5966 - accuracy: 0.7034
209/209 [=====] - 4s 7ms/step - loss: 0.6868 - accuracy: 0.5492
105/105 [=====] - 0s 3ms/step - loss: 0.6579 - accuracy: 0.6429
209/209 [=====] - 2s 6ms/step - loss: 0.6826 - accuracy: 0.5588
105/105 [=====] - 1s 3ms/step - loss: 0.6641 - accuracy: 0.6105
209/209 [=====] - 2s 5ms/step - loss: 0.6832 - accuracy: 0.5615
105/105 [=====] - 0s 3ms/step - loss: 0.6586 - accuracy: 0.6498
209/209 [=====] - 3s 7ms/step - loss: 0.4863 - accuracy: 0.7586
105/105 [=====] - 1s 4ms/step - loss: 0.3798 - accuracy: 0.8319
209/209 [=====] - 2s 5ms/step - loss: 0.4901 - accuracy: 0.7538
105/105 [=====] - 0s 3ms/step - loss: 0.3817 - accuracy: 0.8303
209/209 [=====] - 2s 6ms/step - loss: 0.4856 - accuracy: 0.7567
105/105 [=====] - 0s 3ms/step - loss: 0.3791 - accuracy: 0.8307
834/834 [=====] - 7s 6ms/step - loss: 0.4353 - accuracy: 0.7922
417/417 [=====] - 1s 3ms/step - loss: 0.3975 - accuracy: 0.8137
834/834 [=====] - 6s 6ms/step - loss: 0.4413 - accuracy: 0.7874
417/417 [=====] - 2s 3ms/step - loss: 0.3909 - accuracy: 0.8234
834/834 [=====] - 5s 5ms/step - loss: 0.4447 - accuracy: 0.7841
417/417 [=====] - 1s 3ms/step - loss: 0.3590 - accuracy: 0.8397
209/209 [=====] - 4s 8ms/step - loss: 0.5303 - accuracy: 0.7227
105/105 [=====] - 1s 5ms/step - loss: 0.4317 - accuracy: 0.7992
209/209 [=====] - 2s 6ms/step - loss: 0.5025 - accuracy: 0.7473
105/105 [=====] - 1s 4ms/step - loss: 0.3967 - accuracy: 0.8230
209/209 [=====] - 2s 5ms/step - loss: 0.4864 - accuracy: 0.7570
105/105 [=====] - 0s 3ms/step - loss: 0.4037 - accuracy: 0.8154
834/834 [=====] - 6s 6ms/step - loss: 0.6442 - accuracy: 0.6249
417/417 [=====] - 1s 3ms/step - loss: 0.5659 - accuracy: 0.7182

Figure 72

The figure above shows the conduct of hyperparameter tuning for the proposed model using random search algorithm. The RandomizedSearchCV object is provided with three main parameters which are the model created above, param_grid which stores the hyperparameter grid defined earlier and the cross-validation folds which is set to 3. Then, the fit() method is called to start the hyperparameter tuning with the early stopping callback defined.

```
# Print the best parameters and score
print("Best: %f using %s" % (random_result.best_score_, random_result.best_params_))

Best: 0.830950 using {'optimizer': 'adam', 'kernel_size': 3, 'filters': 64, 'batch_size': 128}
```

Figure 73

Then, the best accuracy obtained, and the best parameters obtained from the hyperparameter tuning process are printed out. It can be seen that the best accuracy obtained was 0.830950 using the hyperparameters of adam optimizer, kernel_size 3, filters of 64 and batch_size 128.

```
[ ] # Get the best model and its parameters
best_model = random_result.best_estimator_
best_params = random_result.best_params_
```

Figure 74

The best model and parameters are stored in the variables above.

```
# Train the best model to obtain the training history
history = best_model.fit(X_train, y_train, epochs=30, batch_size=32, validation_data=(X_test, y_test), callbacks=[early_stopping])

# Extract loss and accuracy values from the training history
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

# Plot the loss curve
plt.figure(figsize=(8, 6))
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True)
plt.show()

# Plot the accuracy curve
plt.figure(figsize=(8, 6))
plt.plot(accuracy, label='Training Accuracy')
plt.plot(val_accuracy, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 75

The code above trains the best model derived from hyperparameter adjustment and displays its training history. The fit() method is used to train the best model on the training dataset (X_train

and y_{train}). The training is conducted for a set number of epochs (30) and batch sizes of 32. During training, the model's performance is evaluated using the validation datasets (X_{test} and y_{test}). The `early_stopping` function is used to halt training if the loss does not improve after three consecutive epochs. The training history is then extracted, which includes the loss and accuracy values for the training and validation sets. Matplotlib is used to generate two different plots: one for the training and validation loss curves and another for the training and validation accuracy curves. These plots provide a visual depiction of the model's training progress, allowing researchers to monitor convergence behavior, identify potential overfitting or underfitting, and evaluate the model's generalization ability. The plots are provided, providing an intuitive overview of the model's performance throughout training.

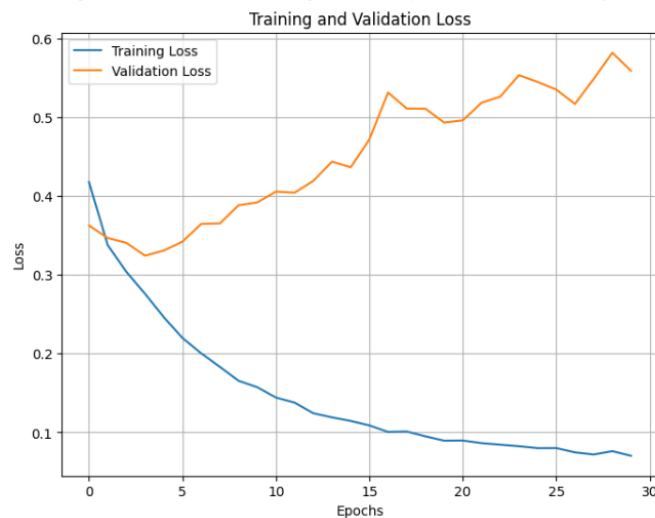


Figure 76

From the training and validation loss plot, it can be seen that there is clear sign of overfitting. It can be observed that the training loss decreases as the epochs increases but the validation loss only decreases until a certain point and starts increasing afterwards. This suggests that the model might be overfitting the training data or failing to generalize to new examples.

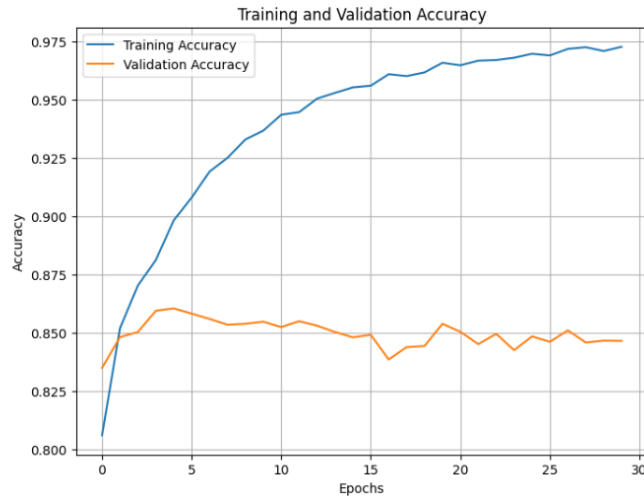


Figure 77

The figures show the result of training and validation accuracy plot. The plot shows signs of overfitting. This can be observed when the training accuracy increases over time, but the validation accuracy only increases for the first few epochs but then remains stagnant for the rest of the epochs. The stagnant validation accuracy indicates that the model may have already learned all it can from the training data and is not able to generalize well to unseen data.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report

# Evaluate the best model on the test set
y_pred = best_model.predict(X_test)
y_pred = np.round(y_pred).flatten()

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)

# Create a confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Print classification report
report = classification_report(y_test, y_pred)

# Print accuracy and confusion matrix
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", cm)
print("Classification Report:\n", report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
plt.xticks([0, 1], ['Negative', 'Positive'])
plt.yticks([0, 1], ['Negative', 'Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

Figure 78

The code above assesses the performance of the best model on the test set, producing a confusion matrix and classification report. The model first predicts the labels for the test set (X_{test}) using `best_model.predict()`. The predicted labels are then rounded and flattened to match the

format of the actual labels (`y_test`). Next, the accuracy is calculated by comparing the predicted labels to the true labels with `np.mean(y_pred == y_test)`. A confusion matrix is then generated using scikit-learn's `confusion_matrix()` method. The confusion matrix contains data on the number of true positives, true negatives, false positives, and false negatives. Scikit-learn's `classification_report()` function generates the classification report. It provides measurements like precision, recall, F1-score, and class support. Finally, the accuracy, confusion matrix, and classification report are printed on the console. The confusion matrix is also represented as a heatmap graphic made with Matplotlib.

Naïve Bayes: Ng Zhi Yao

1.0 Import Libraries and Dataset

```

✓ [2] # Import required module
48 from sklearn.model_selection import train_test_split, GridSearchCV, learning_curve
    from sklearn.naive_bayes import GaussianNB
    from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay, roc_auc_score
    from google.colab import drive
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt

✓ [4] # Load the dataset
218 drive.mount('/content/drive')
    df = pd.read_csv('/content/drive/My Drive/cleaned_IMDB_Dataset.csv')
    print(df.shape)

Mounted at /content/drive
(50000, 2)

```

Figure 79: Naive Bayes Code - Part 1

Figure above demonstrates the method to import require libraries and dataset that will be used to develop Naïve Bayes classification model. For example, “sklearn” library is being used to perform various ML operations such as preprocessing, train models and so on. The team also retrieve the dataset from their cloud storage and display its total rows and columns.

2.0 Preprocessing

```

[5] # Convert string to float for review column
    df.sentiment.replace("positive", 1, inplace = True)
    df.sentiment.replace("negative", 0, inplace = True)
    df.head()

```

	review	sentiment
0	one reviewers mentioned watching 1 oz episode ...	1
1	wonderful little production filming technique ...	1
2	thought wonderful way spend time hot summer we...	1
3	basically family little boy jake thinks zombie...	0
4	petter matteis love time money visually stunni...	1

Figure 80: Naive Bayes Code - Part 2

In the image above, the team convert the datatype of “sentiment” features to integer value where 1 indicates positive review while 0 means bad reviews. This is to enhance the training speed and thus provide a more accurate result.

3.0 Train Model

```
✓ [6] x = df['review']  
0s    y = df['sentiment']  
  
✓ [7] from sklearn.feature_extraction.text import CountVectorizer  
11s    num_features = 1000  
    vectorizer = CountVectorizer(analyzer = "word",  
                                tokenizer = None,  
                                preprocessor = None,  
                                stop_words = None,  
                                max_features = num_features)  
  
    X = vectorizer.fit_transform(X)  
    assert len(vectorizer.get_feature_names_out()) == num_features  
    X = X.toarray()  
    print(X.shape)  
  
    (50000, 1000)  
  
✓ [8] # Split the dataset into training and testing sets  
0s    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Figure 81: Naive Bayes Code - Part 3

The team starts with separating target variables from input and stores them into different variables. By using feature extraction techniques, the team can eliminate the number of words in each review and thus enhance the training speed. Also, it prevents the model from overfitting due to excessive features. Then, they split the output into training and testing set by following a ratio of 8:2. This is also done to prevent excessive data used to train the data and thus does not have sufficient data to evaluate the model's performance.

```
# Perform hyperparameter tuning using GridSearchCV
param_grid = {
    'var_smoothing': np.logspace(0,-9, num=100),
    'priors': [[0.2, 0.8], [0.9, 0.1]]
}

grid_search = GridSearchCV(estimator=GaussianNB(), param_grid=param_grid, cv=5, scoring='accuracy', verbose=10 )
grid_search.fit(X, y)

# Get the best estimator
best_classifier = grid_search.best_estimator_
```

Figure 82: Naive Bayes Code - Part 4

In this illustration above, the team declare a list to store hyperparameters that will be used in Grid Search for tuning the model. This is because hyperparameter tuning can significantly improve the accuracy and performance of the model by finding the best configurations. The detail explanation on the hyperparameters will be done in the bottom section. Next, the team starts to build the model by passing necessary parameters such as estimator which is Naïve Bayes classifier, scoring criteria which is based on accuracy and number of folds for cross validation which is 5-fold cross-validation in this case. Lastly, the team get the Naïve Bayes classifier with the best hyperparameters and results and store it into a variable.

```
# Make predictions on the test set
y_pred_test = best_classifier.predict(X_test)
y_pred_train = best_classifier.predict(X_train)

# Evaluate the classifier
train_accuracy = accuracy_score(y_train, y_pred_train)
test_accuracy = accuracy_score(y_test, y_pred_test)
roc_auc = roc_auc_score(y_test, y_pred_test)

print('Best Hyperparameters:', grid_search.best_params_)
print('ROC-AUC Score:', roc_auc)
print('Train Accuracy:', train_accuracy)
print(classification_report(y_train,y_pred_train))

print("\n-----\n")

print('Test Accuracy:', test_accuracy)
print(classification_report(y_test,y_pred_test))

print('# ----- #')
```

Figure 83: Naive Bayes Code - Part 5

Image above shows the code to perform prediction using the classifier that is equipped with best hyperparameters and store its result into a variable. Also, the results such as accuracy score, ROC-AUC score and so on are displayed in the console which will be show in the section below.


```
# Define the GaussianNB classifier
classifier = GaussianNB(var_smoothing=best_classifier.var_smoothing)

# Create a learning curve
train_sizes, train_scores, test_scores = learning_curve(
    classifier, X, y, cv=3, scoring='accuracy', train_sizes=np.linspace(0.1, 1.0, 10), verbose=1
)

# Calculate the mean and standard deviation of training and test scores
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# Plot the learning curve
plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', marker='o')
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.15)
plt.plot(train_sizes, test_mean, label='Validation Accuracy', marker='o')
plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, alpha=0.15)

# Add labels and a legend
plt.title('Learning Curve')
plt.xlabel('Training Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')

# Show the plot
plt.show()
```

Figure 84: Naive Bayes Code - Part 6

Figure above shows the step to plot a line graph using Python built-in library which is “matplotlib”. First, it starts with declaring a new Gaussian Naïve Bayes classifier and passing in the best hyperparameters from the previous finding. Then, it executes a built-in function to predict the accuracy of the model with the increase of training size. The results are being plotted and labelled on the graph to visualize the output. Through this diagram, the team can better understand their model is if it is overfitting or underfitting.

```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred_test)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()
```

Figure 85: Naive Bayes Code - Part 7

This code is used to plot the confusion matrix for the model. As a result, the team is able to gain insights into how well the model is performing and identify areas for improvement by checking on the main components in confusion matrix such as true positive (TP) where it indicates the total number of positive class cases that the model predicts correctly.

Building A Supervised Text Classification model using SAS Text Miner

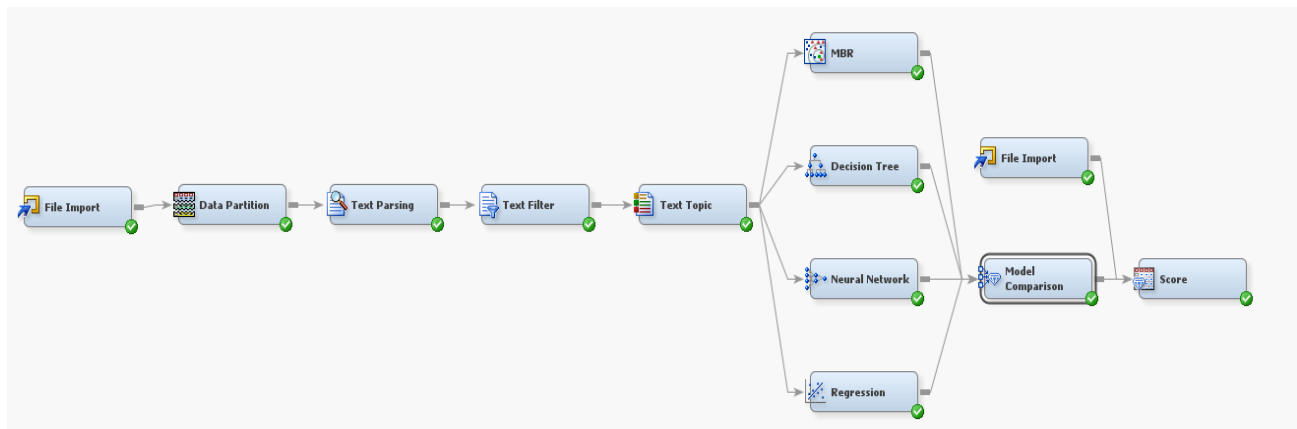


Figure 86: Process Flow of Building A Supervised Model

In this section, the team decided to use SAS Enterprise Miner to build a supervised model that can be used for sentiment analysis. They start by importing the data source which is the same movie dataset mentioned in the above section. Then, they use “Data Partition” node to split the data into training and test set where 70% of the data for training and the remaining for test. This is because it can prevent overfitting in predictive modelling. It is known that the proportion for training data can vary depending on the scenario and problem. Next, they perform text parsing process where text data is breaking down into smaller and meaning components such as words, phrases or token to extract relevant information from raw text data. This can be done by identifying and separating different elements like words, numbers, punctuation and so on. Also, text filter is done by them to filter out irrelevant or unwanted text data which can help in removing noises and stop words which can improve the quality and relevance of analysis results. Furthermore, they pass the output to “Text Topic” node which is used to identify and extract key topics or themes from a collection of text documents. This allows deeper insights into the underlying content.

After the node, the flow branches out to multiple nodes such as Regression, Neural Network, Decision Tree and Memory Based Reasoning (MBR). These classification algorithms are being built in the application and thus the team can easily access and use them. Default settings are initially used on those algorithms. It is also known that multiple nodes of the same type can be added in separate branches to test various models simultaneously.

Then, all modelling nodes are connected to a “Model Comparison” node which can evaluate and compare the output from each model to determine the best one for the dataset. Statistics such as reports and graphs about all models are displayed in the node result. For example, the misclassification rate for categorical data for specific model. Lastly, the output from the node will pass into “Score” node which is used to predict the data once again using the best model. Reports and justification will be provided below.

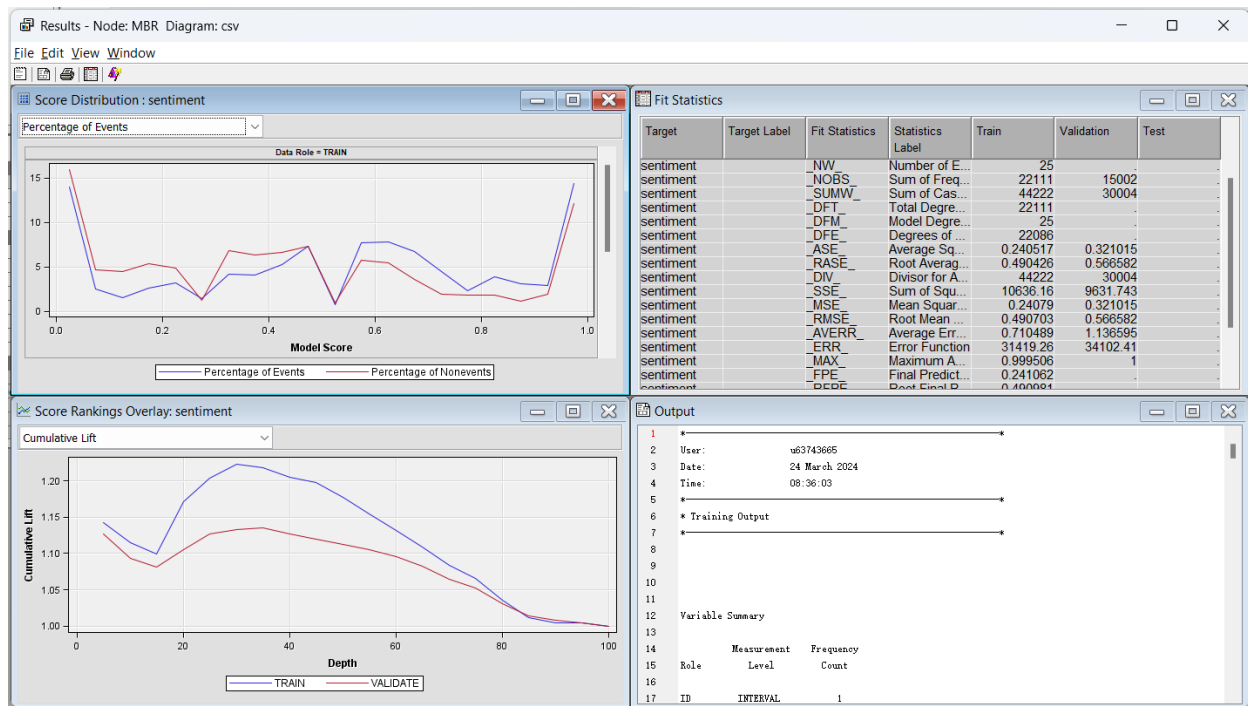


Figure 87: Statistic for MBR Model

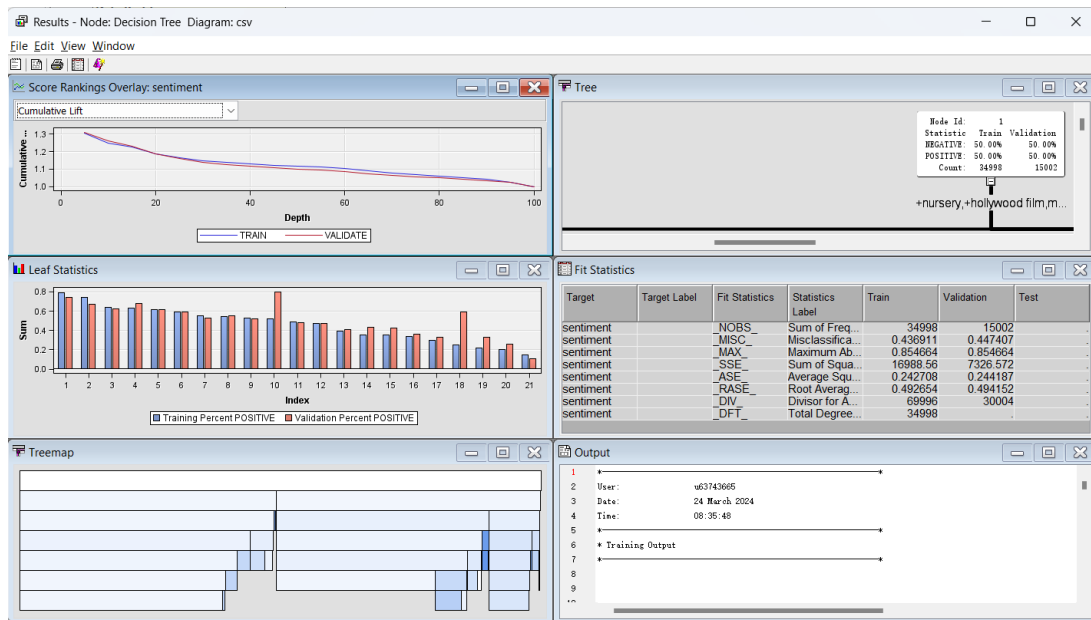


Figure 88: Statistic for Decision Tree model

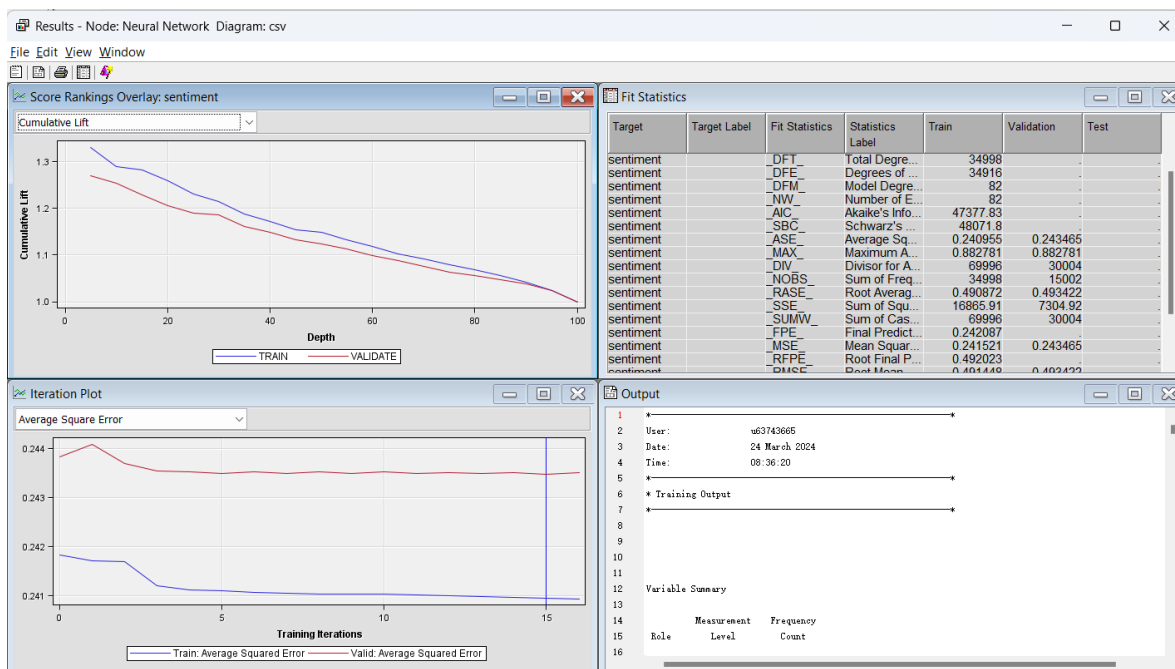


Figure 89: Statistic for Neural Network Model

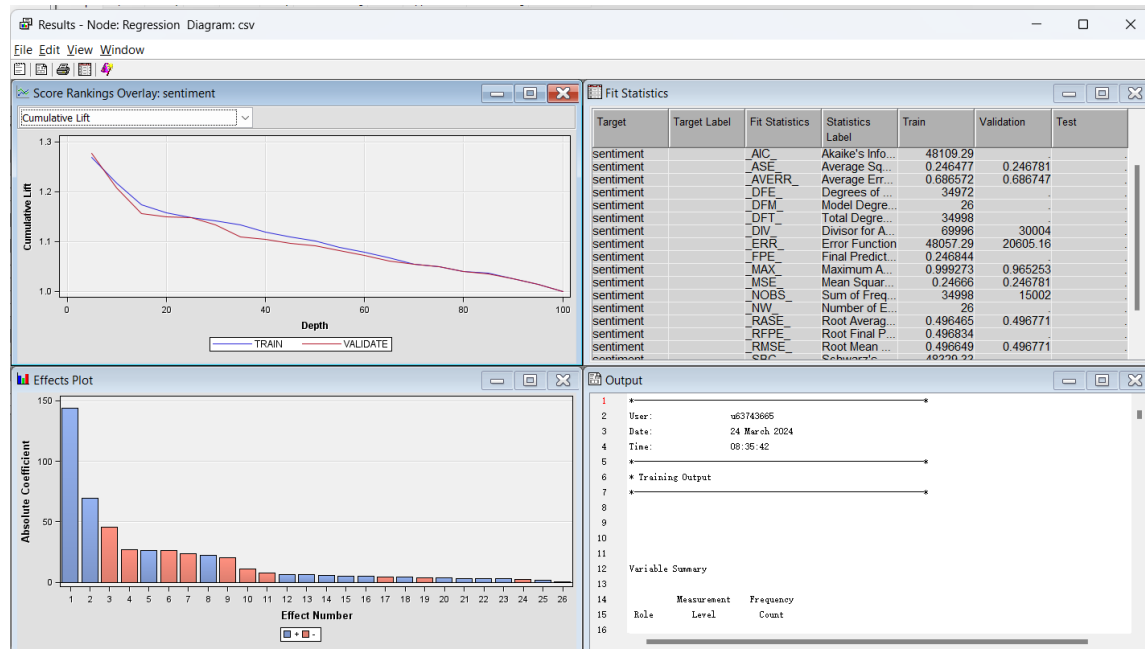


Figure 90: Statistic for Regression Model

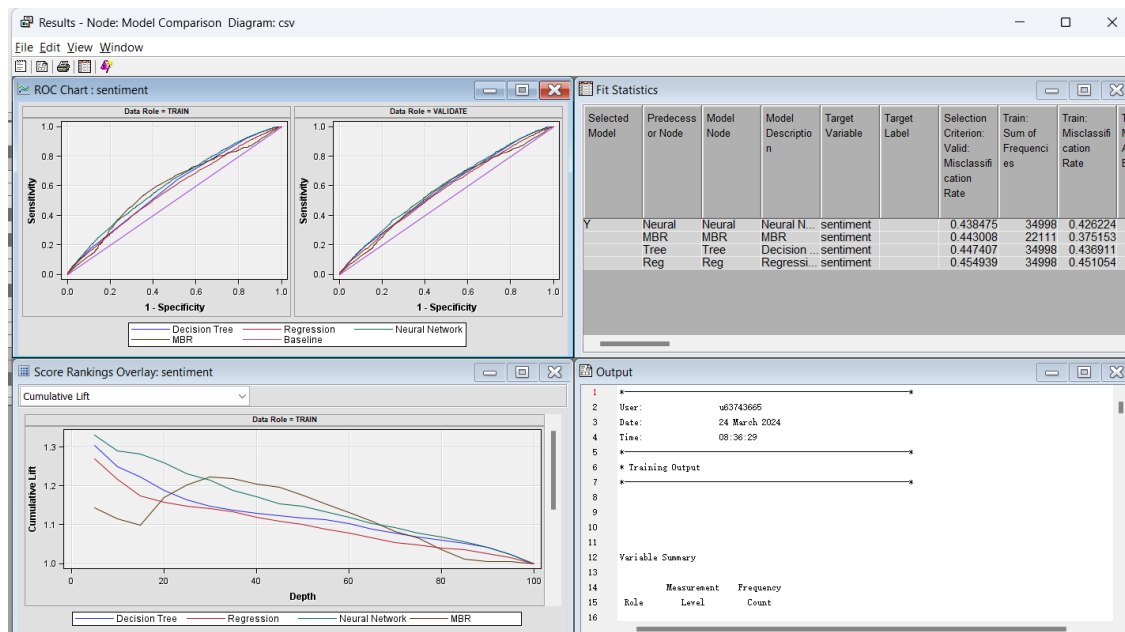


Figure 91: Statistics of Models' Comparison

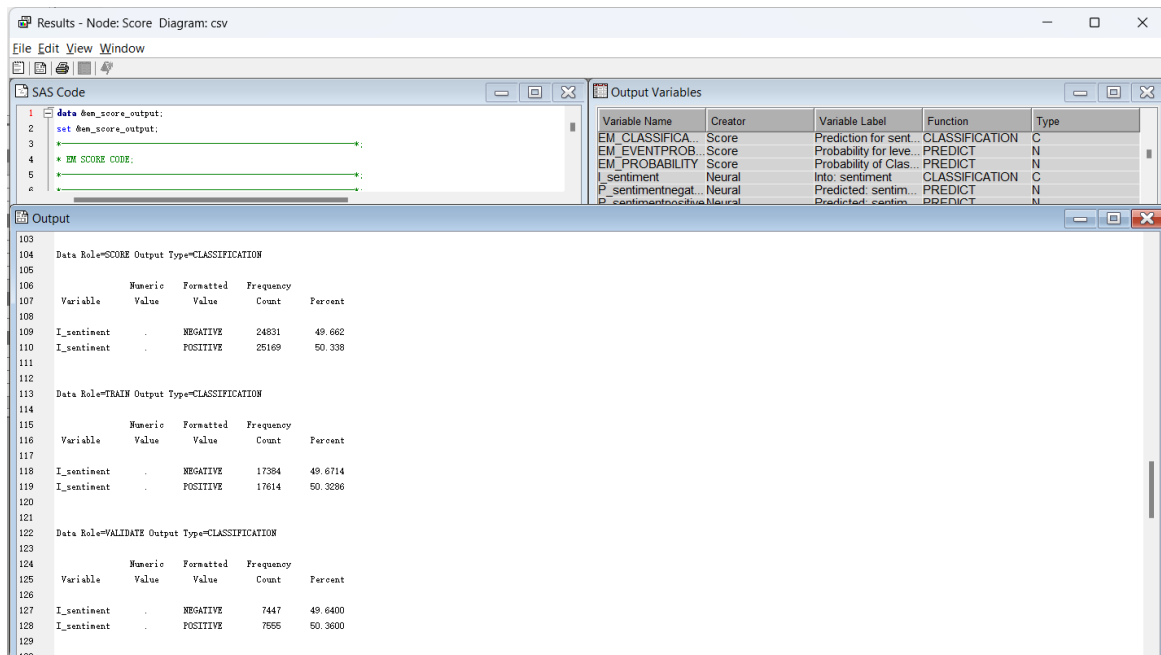


Figure 92: Final Score for Best Model

Based on the figures above, it shows the performance metrics of different models which consist of graphs and messages. Due to lengthy report, the team summarized the report and selected the best model that is suited for the dataset. Firstly, Decision Tree and Regression model have a higher misclassification rate compared to other two models when it comes to categorizing the reviews' sentiments. Therefore, it indicates that both models have relatively poorer performance in terms of classification accuracy. On the other hand, MRB model shows a competitive performance which it has a lower misclassification rate for the validation dataset compared to other models. Lastly, Neural Network model has the lowest misclassification rate on both training and validation datasets among the models which suggest that it can perform well and complete the classification task with ease.

Q3: Hyperparameter selection

LSTM: Low Sim Chuan

For the LSTM model, there are various hyperparameters to be tuned for achieving better training results, such as:

- Embedding dimension of the Embedding layer: it controls the dimension of the vectors for each word in the vocabulary. A larger embedding dimension means a more complicated representation can be learned for each word, but it simply means more parameters for the LSTM model to learn, eventually leading to longer training time.
- Number of hidden units of the LSTM layer: it controls the capacity of the LSTM layer. If the number is increased, it means complex patterns within the data can be learned but it is also prone to overfitting and more time-consuming.
- Learning rate: it controls how much the LSTM model's weights are updated during training, and it is analogous to the step size when someone is climbing up or going down a hill. A larger learning rate means that the weights are updated more severe at each step, this can lead to fast convergence at beginning, but it could cause the model fluctuating and even diverging from the optimal solution (Doshi, 2021). A smaller learning rate can lead to slow convergence, which means it takes longer time for the model to reach the optimal solution.
- Momentum: it accelerates the SGD in the relevant direction and dampens oscillation (abhishek, n.d.).

```
def model_builder_sgd(hp):  
    hp_embedding_dim = hp.Int("embedding_dim", min_value=300, max_value=500, step=100)  
    hp_hidden_unit_dim = hp.Int("hidden_unit_dim", min_value=32, max_value=128, step=32)  
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])  
    hp_momentum = hp.Choice("momentum", values=[0.0, 0.5, 0.9, 0.99])  
  
    model = Sequential()  
    model.add(Embedding(vocab_num, hp_embedding_dim, input_length=max_sequence_len))  
    model.add(LSTM(hp_hidden_unit_dim))  
    model.add(Dense(1, activation='sigmoid'))  
    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(  
        hp_learning_rate,  
        decay_steps=100000,  
        decay_rate=0.96,  
        staircase=True  
    )  
    sgd_opt = keras.optimizers.SGD(  
        learning_rate=lr_schedule,  
        momentum=hp_momentum  
    )  
  
    model.compile(  
        optimizer=sgd_opt,  
        loss="binary_crossentropy",  
        metrics=['accuracy']  
    )  
  
    return model
```

Figure 93: Hyperparameters of LSTM

First, all the ranges of the hyperparameters of the LSTM are defined: range of embedding dimension in Embedding layer, range of hidden unit dimension of LSTM layer, range of learning rate, and range of momentum in the SGD optimizer. The embedding dimension could range from 300 to 500 in steps of 100, the hidden unit number could range from 32 to 128 in steps of 32, the learning rate could be 0.01, 0.001, or 0.0001, and the momentum could be 0.0, 0.5, 0.9 or 0.99.


```

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
tuner = kt.GridSearch(
    model_builder_sgd,
    objective='val_accuracy',
    directory='GridSearchResults',
    project_name='LSTM_SGD',
)

EPOCH_NUM = 30

tuner.search(X_train, y_train, epochs=EPOCH_NUM, validation_data=(X_test, y_test), callbacks=[stop_early])

Trial 10 Complete [00h 07m 50s]
val_accuracy: 0.5077999830245972

Best val_accuracy So Far: 0.879000081062317
Total elapsed time: 01h 02m 12s

Search: Running Trial #11

Value          |Best Value So Far|Hyperparameter
300             |300              |embedding_dim
32              |32               |hidden_unit_dim
0.0001          |0.01             |learning_rate
0.9             |0.99             |momentum

Epoch 1/30
1250/1250 [=====] - 16s 12ms/step - loss: 0.6937 - accuracy: 0.4922 - val_loss: 0.6932 - val_accuracy: 0.4983
Epoch 2/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6933 - accuracy: 0.4933 - val_loss: 0.6932 - val_accuracy: 0.4998
Epoch 3/30
1250/1250 [=====] - 17s 13ms/step - loss: 0.6933 - accuracy: 0.4971 - val_loss: 0.6931 - val_accuracy: 0.5004
Epoch 4/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6932 - accuracy: 0.4979 - val_loss: 0.6931 - val_accuracy: 0.5006
Epoch 5/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6932 - accuracy: 0.5026 - val_loss: 0.6931 - val_accuracy: 0.5045
Epoch 6/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6931 - accuracy: 0.5042 - val_loss: 0.6931 - val_accuracy: 0.5052
Epoch 7/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6931 - accuracy: 0.5071 - val_loss: 0.6931 - val_accuracy: 0.5020
Epoch 8/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6931 - accuracy: 0.5080 - val_loss: 0.6931 - val_accuracy: 0.5035
Epoch 9/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6930 - accuracy: 0.5076 - val_loss: 0.6931 - val_accuracy: 0.5040
Epoch 10/30
1250/1250 [=====] - 16s 13ms/step - loss: 0.6930 - accuracy: 0.5081 - val_loss: 0.6931 - val_accuracy: 0.5031
Epoch 11/30
1250/1250 [=====] - 15s 12ms/step - loss: 0.6930 - accuracy: 0.5085 - val_loss: 0.6931 - val_accuracy: 0.5029
Epoch 12/30
1250/1250 [=====] - ETA: 0s - loss: 0.6930 - accuracy: 0.5082

```

Figure 94: Grid search

The grid search is then performed to search for the best set of hyperparameter, in other words it means trying all each possible combinations of the hyperparameters to see which one performs the best in terms of the validation accuracy.

Best set of hyperparameters for SGD optimizer:

```
# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The hyperparameter search is complete.
The optimal dimension of embedding vector is {best_hps.get('embedding_dim')}
The optimal dimension of hidden units in LSTM layer is {best_hps.get('hidden_unit_dim')}
The optimal learning rate for the optimizer is {best_hps.get('learning_rate')}
The optimal momentum for the optimizer is {best_hps.get('momentum')}
""")
```

```
The hyperparameter search is complete.
The optimal dimension of embedding vector is 300
The optimal dimension of hidden units in LSTM layer is 64
The optimal learning rate for the optimizer is 0.01
The optimal momentum for the optimizer is 0.99
```

Figure 95: Best set of hyperparameters for LSTM after grid search

After an hour of waiting, the best set of hyperparameters are 300 for the embedding dimension, 64 for the number of hidden units, 0.01 for the learning rate, and 0.99 for the momentum of the SGD optimizer.

```
# Evaluate the model
train_loss, train_acc = model.evaluate(X_train, y_train, verbose=2)
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print("Train accuracy: {:.2f}%".format(100 * train_acc))
print("Test accuracy: {:.2f}%".format(100 * test_acc))
```

```
1250/1250 - 5s - loss: 0.0908 - accuracy: 0.9695 - 5s/epoch - 4ms/step
313/313 - 1s - loss: 0.4176 - accuracy: 0.8647 - 1s/epoch - 4ms/step
Train accuracy: 96.95%
Test accuracy: 86.47%
```

Figure 96: Train and test accuracies

The best set of hyperparameters are used to rebuild and retrain the LSTM model, then the LSTM model is used to predict on train set and test set. The train accuracy is 96.95% while the test accuracy is 86.47%.

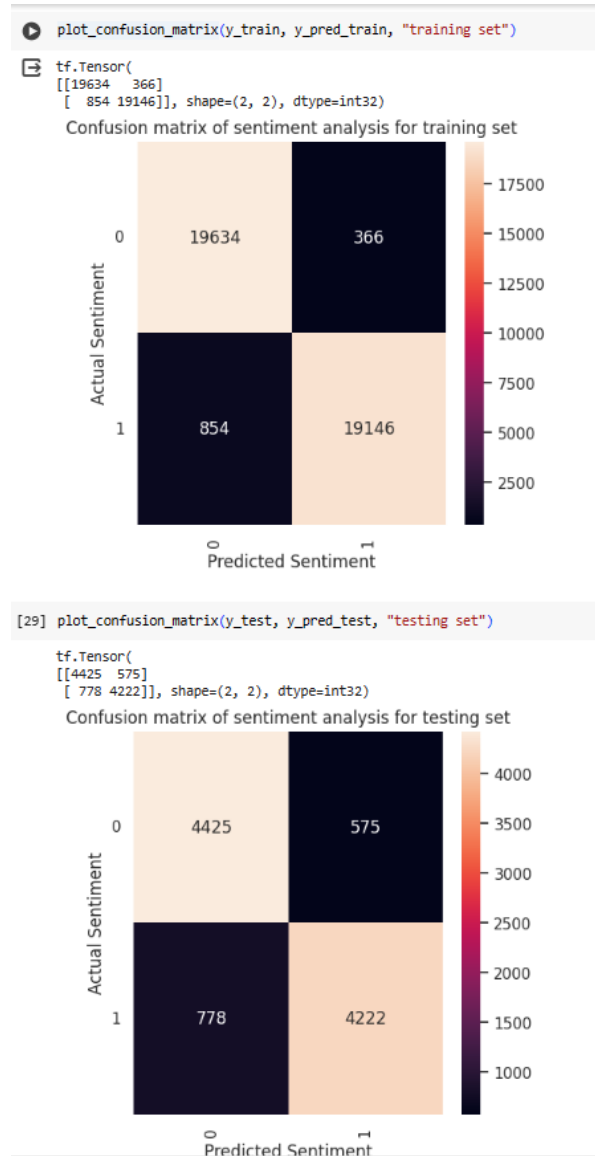


Figure 97: Confusion matrix for train set and test set

The confusion matrices are also plotted after retraining the LSTM model using the best set of hyperparameters. For the train set, there are 19634 negative movie reviews and 19146 positive movie reviews correctly classified, 366 negative movie reviews being misclassified as positive, and 854 positive movie reviews being misclassified as negative reviews. For the test set, there are 4425 negative movie reviews and 4222 positive movie reviews correctly classified, 575 negative movie reviews being misclassified as positive, and 778 positive movie review being misclassified as negative reviews.

SVM-LinearSVC: Nivethan

Linear Support Vector Classification, or "LinearSVC" as it is known in the code, is the model in use here. You may find this in Scikit-learn. This model investigates and adjusts two hyperparameters which are the regularization parameter ('C') and the maximum number of iterations ('max_iter').

There are also some other hyperparameters that are not explicitly explored but important for the model. Examples are :

- A. 'penalty' – This is the norm used in penalization.
- B. 'loss' – This specifies the loss function of the model.
- C. 'dual' – This specifies the dual or primal formulation of the model.
- D. 'tol' – This shows the tolerance for the stopping criteria of the model.
- E. 'class_weight' – This shows the weights that are associated with classes in the model.
- F. 'random_state' – This is the seed of the pseudo random number generator which controls the randomness of the estimator.

Having said that, let us return to the model's hyperparameters. First, the 'C' parameter maximizes the margin of the decision function or trades off the proper classification of training instances. If 'C' were much greater, a narrower margin would be acceptable. For this to occur, the decision function must accurately categorize each training point. A lower 'C' would also promote a wider margin. The training accuracy would suffer because of the decision boundary's simplification.

The below image is the parameter of 'C' in our model :

```
param_distributions = {  
    'C': [0.001, 0.01, 0.1, 1, 10], # Example range, can include more
```

Figure 98

The regularization parameter influences the model's tolerance to classification mistakes, which is a critical component of this model. A low number for 'C' might result in underfitting, whilst a large value for 'C' would cause overfitting.

The image below shows the parameters of maximum number of iterations:

```
'max_iter': [1000, 2000, 3000] # Example range, can include more
```

Figure 99

The maximum number of iterations required for the solver to converge is specified by the parameter above. The parameter's function in this model is to control how many times the training method is executed. In other words, the method will terminate when the maximum number of iterations required to obtain the best answer is reached. Increasing the value guarantees more thorough convergence, but it comes at the expense of processing time.

To optimize the model for a given dataset, it is necessary to fine-tune these hyperparameters, as they significantly affect the model's performance.

To determine the optimal hyperparameters for the LinearSVC model, the above code was used in conjunction with the randomized search strategy. This section explains the hyperparameters that were considered. This randomized search configuration was also used to train the model on the training dataset. Cross validation happens when training is finished. Next, the optimal hyperparameters are found.

✓ Fit the random search model to the training data

```
[ ] random_search.fit(X_train, y_train)
```

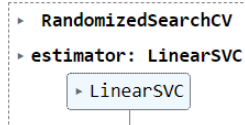


Figure 100

```
Best hyperparameters: {'max_iter': 3000, 'c': 0.1}
Training Set Accuracy: 0.867
Test Set Accuracy: 0.8612
```

	precision	recall	f1-score	support
0	0.87	0.84	0.86	4961
1	0.85	0.88	0.86	5039
accuracy			0.86	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.86	0.86	0.86	10000

Figure 101

The model is then evaluated with performance metrics like the accuracy scores, F-1 score and many more. This randomized approach can offer a balance between grid search and computational efficiency.

One Dimensional Convolutional Neural Network: Liaw Yu Jay

There are several hyperparameters that are tuned for the 1D convolutional neural network. There are listed and explained above.

- 1) filters: The filter hyperparameter determines the size of the filters in each convolutional layer. A bigger filter size can capture more complicated information, but it also adds computational complexity (Sathe, 2023).
- 2) kernel_size: The kernel_size hyperparameter controls the receptive field of each convolutional operation and it specifies the convolutional window size (Keras, n.d.).
- 3) batch_size: The batch_size specifies the number of samples handled every training iteration. Smaller batch sizes may result in slower convergence, but they can also assist minimize overfitting (Sathe, 2023).
- 4) optimizer: Optimizers can be defined as a mathematical function that modifies the weights of a network based on gradients and extra information, depending on the optimizer formulation. Optimizers are based on the concept of gradient descent, a greedy strategy to iteratively decrease the loss function by following the gradient (Park, 2021).

```
[ ] # Define the hyperparameters grid
param_grid = {
    'filters': [16, 32, 64],
    'kernel_size': [3, 5, 7],
    'batch_size': [32, 64, 128],
    'optimizer': ['adam', 'sgd'],
}
```

Figure 102

The above shows the range of hyperparameters values are defined first before hyperparameter tuning is conducted. The range of filters is set to be 16, 32, and 64. The range of kernel_size is set to be 3, 5, and 7. The range of batch_size is set to be 32, 64, and 128. The optimizer tested is either adam or sg.

```
[ ] # Define the early stopping callback
early_stopping = EarlyStopping(monitor='loss', patience=3)

CONDUCTING GRID SEARCH FOR THE MODEL.

[ ] # Create a RandomizedSearchCV object
random_search = RandomizedSearchCV(model, param_grid, cv=3, n_iter=10, random_state=42)

# Fit the random search with early stopping
random_result = random_search.fit(X_train, y_train, callbacks=[early_stopping])
```

417/417 [=====] - 7s 7ms/step - loss: 0.6819 - accuracy: 0.5673
209/209 [=====] - 1s 3ms/step - loss: 0.6511 - accuracy: 0.6638
417/417 [=====] - 3s 5ms/step - loss: 0.6870 - accuracy: 0.5558
209/209 [=====] - 1s 3ms/step - loss: 0.6586 - accuracy: 0.6386
417/417 [=====] - 3s 4ms/step - loss: 0.6544 - accuracy: 0.6089
209/209 [=====] - 1s 3ms/step - loss: 0.5966 - accuracy: 0.7034
209/209 [=====] - 4s 7ms/step - loss: 0.6868 - accuracy: 0.5492
105/105 [=====] - 0s 3ms/step - loss: 0.6579 - accuracy: 0.6429
209/209 [=====] - 2s 6ms/step - loss: 0.6826 - accuracy: 0.5588
105/105 [=====] - 1s 3ms/step - loss: 0.6641 - accuracy: 0.6105
209/209 [=====] - 2s 5ms/step - loss: 0.6832 - accuracy: 0.5615
105/105 [=====] - 0s 3ms/step - loss: 0.6586 - accuracy: 0.6498
209/209 [=====] - 3s 7ms/step - loss: 0.4863 - accuracy: 0.7586
105/105 [=====] - 1s 4ms/step - loss: 0.3798 - accuracy: 0.8319
209/209 [=====] - 2s 5ms/step - loss: 0.4901 - accuracy: 0.7538
105/105 [=====] - 0s 3ms/step - loss: 0.3817 - accuracy: 0.8303
209/209 [=====] - 2s 6ms/step - loss: 0.4856 - accuracy: 0.7567
105/105 [=====] - 0s 3ms/step - loss: 0.3791 - accuracy: 0.8307
834/834 [=====] - 7s 6ms/step - loss: 0.4353 - accuracy: 0.7922
417/417 [=====] - 1s 3ms/step - loss: 0.3975 - accuracy: 0.8137
834/834 [=====] - 6s 6ms/step - loss: 0.4413 - accuracy: 0.7874
417/417 [=====] - 2s 3ms/step - loss: 0.3909 - accuracy: 0.8234
834/834 [=====] - 5s 5ms/step - loss: 0.4447 - accuracy: 0.7841
417/417 [=====] - 1s 3ms/step - loss: 0.3590 - accuracy: 0.8397
209/209 [=====] - 4s 8ms/step - loss: 0.5303 - accuracy: 0.7227
105/105 [=====] - 1s 5ms/step - loss: 0.4317 - accuracy: 0.7992
209/209 [=====] - 2s 6ms/step - loss: 0.5025 - accuracy: 0.7473
105/105 [=====] - 1s 4ms/step - loss: 0.3967 - accuracy: 0.8230
209/209 [=====] - 2s 5ms/step - loss: 0.4864 - accuracy: 0.7570
105/105 [=====] - 0s 3ms/step - loss: 0.4037 - accuracy: 0.8154
834/834 [=====] - 6s 6ms/step - loss: 0.4442 - accuracy: 0.6249
417/417 [=====] - 1s 3ms/step - loss: 0.5659 - accuracy: 0.7182
834/834 [=====] - 6s 6ms/step - loss: 0.5859 - accuracy: 0.6842
417/417 [=====] - 2s 3ms/step - loss: 0.4945 - accuracy: 0.7632
834/834 [=====] - 5s 5ms/step - loss: 0.5936 - accuracy: 0.6698
417/417 [=====] - 1s 3ms/step - loss: 0.4756 - accuracy: 0.7790
834/834 [=====] - 7s 7ms/step - loss: 0.5941 - accuracy: 0.6736
417/417 [=====] - 2s 4ms/step - loss: 0.4874 - accuracy: 0.7701
834/834 [=====] - 6s 7ms/step - loss: 0.5787 - accuracy: 0.6883
417/417 [=====] - 2s 3ms/step - loss: 0.5615 - accuracy: 0.7025
834/834 [=====] - 6s 7ms/step - loss: 0.5936 - accuracy: 0.6727
417/417 [=====] - 2s 3ms/step - loss: 0.5129 - accuracy: 0.7457
209/209 [=====] - 4s 9ms/step - loss: 0.4975 - accuracy: 0.7510
105/105 [=====] - 1s 4ms/step - loss: 0.3981 - accuracy: 0.8208
209/209 [=====] - 3s 7ms/step - loss: 0.4834 - accuracy: 0.7628
105/105 [=====] - 0s 3ms/step - loss: 0.3985 - accuracy: 0.8193
209/209 [=====] - 2s 6ms/step - loss: 0.4783 - accuracy: 0.7657
105/105 [=====] - 0s 3ms/step - loss: 0.3999 - accuracy: 0.8159
834/834 [=====] - 6s 6ms/step - loss: 0.6342 - accuracy: 0.6313
417/417 [=====] - 1s 3ms/step - loss: 0.5289 - accuracy: 0.7539
834/834 [=====] - 5s 6ms/step - loss: 0.6145 - accuracy: 0.6581
417/417 [=====] - 2s 4ms/step - loss: 0.5462 - accuracy: 0.7215
834/834 [=====] - 6s 6ms/step - loss: 0.6587 - accuracy: 0.6151

Figure 103

In the figure, it shows that an instance of EarlyStopping is defined so that the model does not waste time continuing its training after the loss does not improve after three epochs. Then, the random search hyperparameter tuning is conducted to find the best set of combinations of hyperparameter from the hyperparameters defined above.

```
[ ] # Print the best parameters and score
print("Best: %f using %s" % (random_result.best_score_, random_result.best_params_))

Best: 0.830950 using {'optimizer': 'adam', 'kernel_size': 3, 'filters': 64, 'batch_size': 128}
```

Figure 104

The result obtained from hyperparameter tuning showed that the best optimizer is adam, best kernel_size is 3, best filters value I 64 and best batch_size is 128.


```

# Get the best model and its parameters
best_model = random_result.best_estimator_
best_params = random_result.best_params_

# Train the best model to obtain the training history
history = best_model.fit(X_train, y_train, epochs=30, batch_size=32, validation_data=(X_test, y_test), callbacks=[early_stopping])

# Extract loss and accuracy values from the training history
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

# Plot the loss curve
plt.figure(figsize=(8, 6))
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True)
plt.show()

# Plot the accuracy curve
plt.figure(figsize=(8, 6))
plt.plot(accuracy, label='Training Accuracy')
plt.plot(val_accuracy, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.grid(True)
plt.show()

```

Figure 105

The above shows that the best model is obtained and retrained on the test set to see the training history of the model.

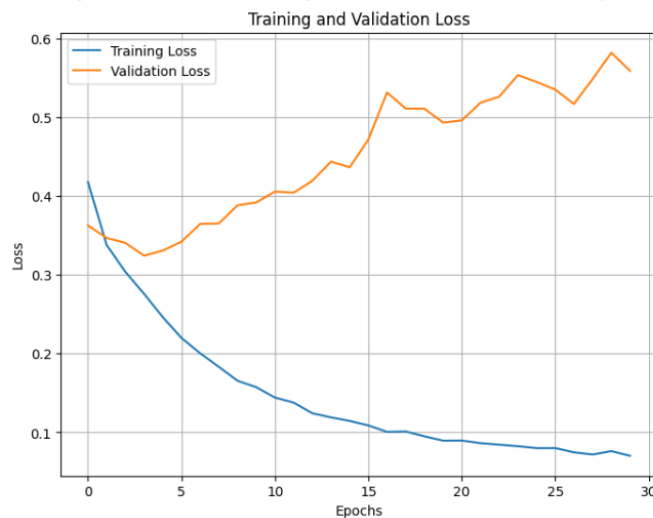


Figure 106

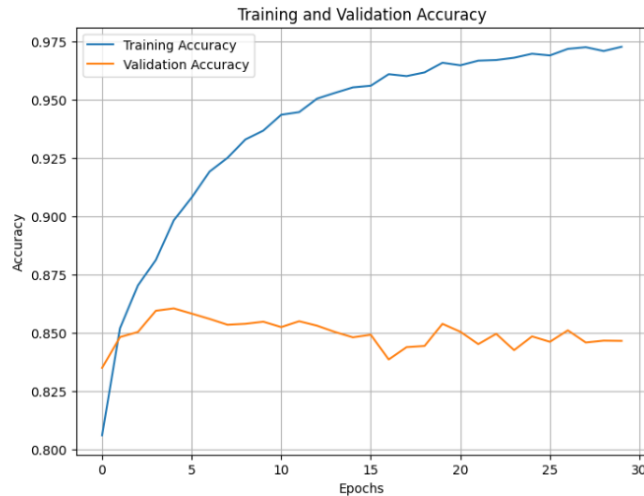


Figure 107

From the plot of the training history, it can be observed that the model shows signs of overfitting with the training loss and accuracy improving but the validation loss and accuracy declining or stagnant.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report

# Evaluate the best model on the test set
y_pred = best_model.predict(X_test)
y_pred = np.round(y_pred).flatten()

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)

# Create a confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Print classification report
report = classification_report(y_test, y_pred)

# Print accuracy and confusion matrix
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", cm)
print("Classification Report:\n", report)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
plt.xticks([0, 1], ['Negative', 'Positive'])
plt.yticks([0, 1], ['Negative', 'Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

Figure 108

The code is used to perform model evaluation on the test set.

```

Accuracy: 0.8464
Confusion Matrix:
[[4077  923]
 [ 613 4387]]
Classification Report:

```

	precision	recall	f1-score	support
0	0.87	0.82	0.84	5000
1	0.83	0.88	0.85	5000
accuracy			0.85	10000
macro avg	0.85	0.85	0.85	10000
weighted avg	0.85	0.85	0.85	10000

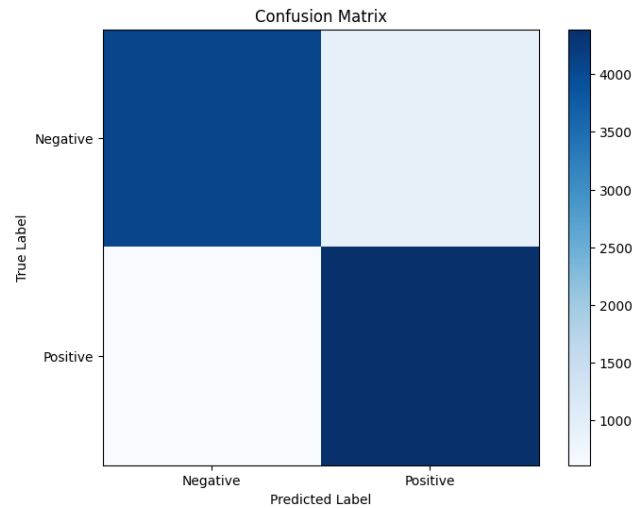


Figure 109

According to the result, the model achieved an overall accuracy of 0.864 on the test dataset. In more detail the model identified 4077 true Negatives, 923 false positives. 613 false negatives, and 4387 true positives. Additional classification reports including precision, recall, F1-score and support are also provided.

Naïve Bayes: Ng Zhi Yao

```
# Perform hyperparameter tuning using GridSearchCV
param_grid = {
    'var_smoothing': np.logspace(0,-9, num=100),
    'priors': [[0.2, 0.8], [0.9, 0.1]]
}
```

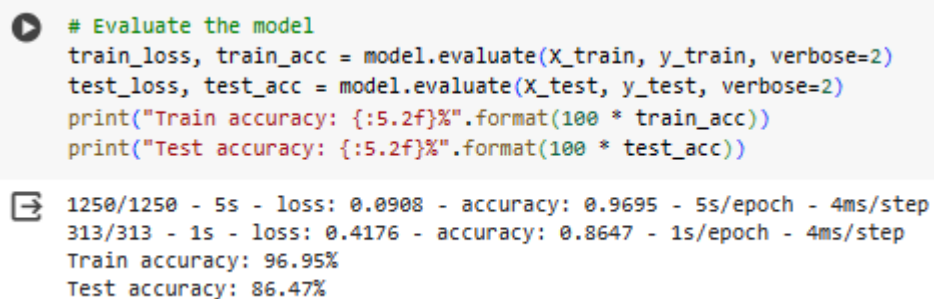
Figure 110: Hyperparameters for Naive Bayes

It is known that hyperparameters are settings that are external to the model and are not learned from the data during training. They control aspects of the learning process and can significantly impact a ML models' performance and behavior. In the figure above, two hyperparameters are used to tune Naïve Bayes model. Firstly, “var_smoothing” hyperparameter is used to control the portion of the largest variance of all features that is added to variances for calculation stability. With this hyperparameter, it can prevent numerical instability when dealing with features that have zero variance or very small variance. Next, “priors” is another hyperparameter that specifies the prior probabilities of the classes in the Naïve Bayes classifier. It allows the team to manually set the probabilities for each class, which can be useful in cases where you have prior knowledge about the class distribution.

Q4: Evaluation & Discussion of the predictive models' results

LSTM: Low Sim Chuan

Accuracy and confusion matrix are selected for evaluating the performance of LSTM model on IMDB dataset. Accuracy is just the ratio of number of correctly classified samples to the total number of samples in the dataset. In the case of binary classification, confusion matrix is a matrix talking about how many samples are correctly classified and how many samples are misclassified for each class.



```
# Evaluate the model
train_loss, train_acc = model.evaluate(X_train, y_train, verbose=2)
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print("Train accuracy: {:.2f}%".format(100 * train_acc))
print("Test accuracy: {:.2f}%".format(100 * test_acc))
```

1250/1250 - 5s - loss: 0.0908 - accuracy: 0.9695 - 5s/epoch - 4ms/step
313/313 - 1s - loss: 0.4176 - accuracy: 0.8647 - 1s/epoch - 4ms/step
Train accuracy: 96.95%
Test accuracy: 86.47%

Figure 111: Train and test accuracies

The best set of hyperparameters are used to rebuild and retrain the LSTM model, then the LSTM model is used to predict on train set and test set. The train accuracy is 96.95% while the test accuracy is 86.47%.

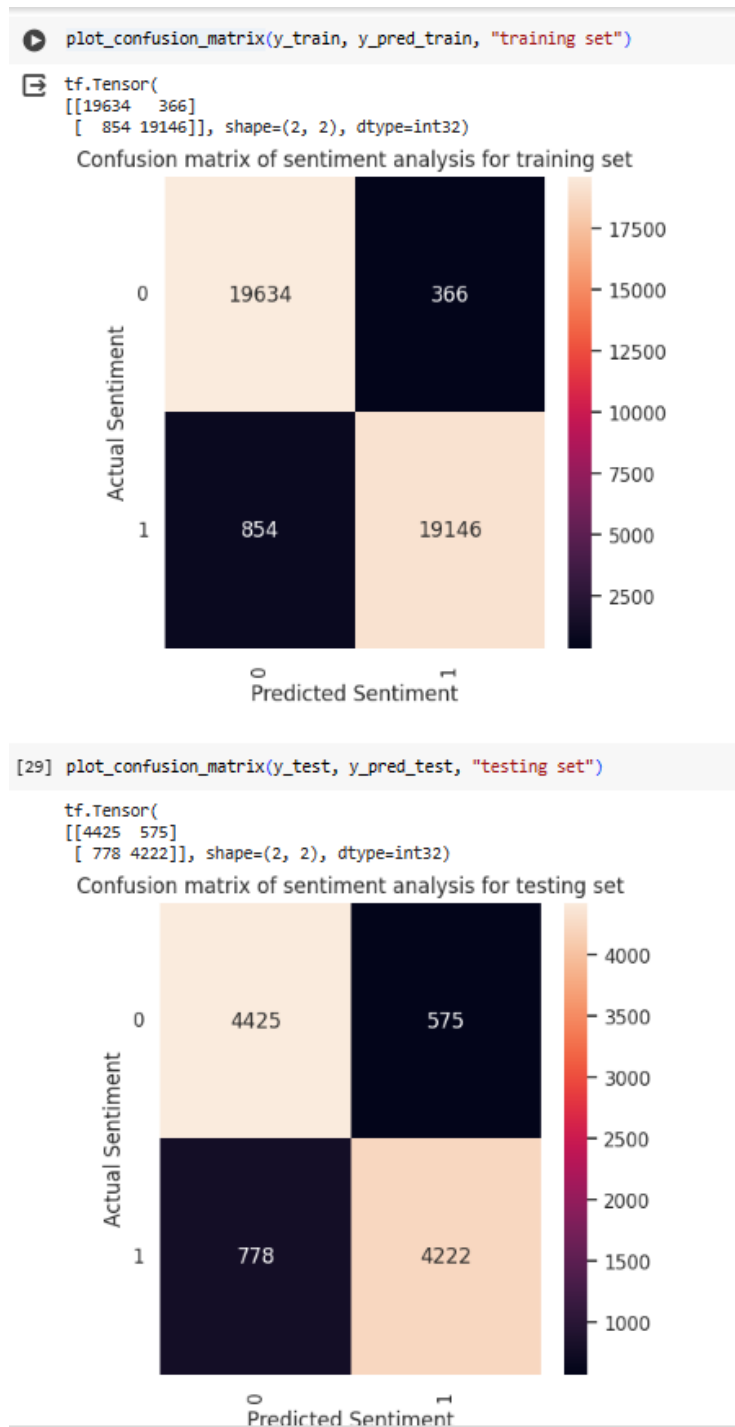


Figure 112: Confusion matrix for train and test sets

The confusion matrices are also plotted after retraining the LSTM model using the best set of hyperparameters. For the train set, there are 19634 negative movie reviews and 19146 positive movie reviews correctly classified, 366 negative movie reviews being misclassified as positive,

and 854 positive movie reviews being misclassified as negative reviews. For the test set, there are 4425 negative movie reviews and 4222 positive movie reviews correctly classified, 575 negative movie reviews being misclassified as positive, and 778 positive movie review being misclassified as negative reviews.

SVM-LinearSVC: Nivethan

Accuracy is the primary suitable measure for assessment based on the code, which handles a binary text classification job utilizing the 'LinearSVC'. To get the accuracy, I would divide the number of cases for which my predictions were accurate by the total number of occurrences.

Following that, we have the F1-score, recall, and precision. Accuracy measures the proportion of favorable forecasts that really came true. Additionally, all affirmative cases are found using the recall as the classifier. Next, we calculate the harmonic mean of recall and accuracy using the F1-Score.

Finally, the classifier's mistake kinds may be understood by examining the confusion matrix.

```
Best hyperparameters: {'max_iter': 3000, 'C': 0.1}
Training Set Accuracy: 0.867
Test Set Accuracy: 0.8612
```

	precision	recall	f1-score	support
0	0.87	0.84	0.86	4961
1	0.85	0.88	0.86	5039
accuracy			0.86	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.86	0.86	0.86	10000

Figure 113

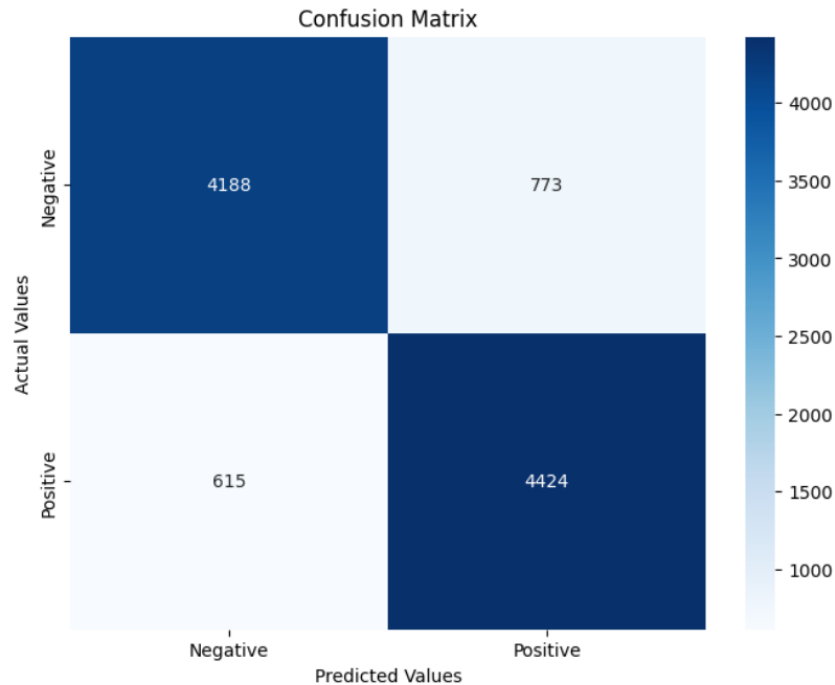


Figure 114

The results above show the accuracy calculation, classification report and confusion matrix. The cell above shows that the confusion matrix is plotted for the test set predictions using the 'matplotlib' and 'seaborn'. In addition, the matrix provides a visual representation of the model's performance by comparing the actual values to the predicted values for both classes which are ('Negative', 'Positive').

One Dimensional Convolutional Neural Network: Liaw Yu Jay

```

313/313 [=====] - 1s 2ms/step
Accuracy: 0.8464
Confusion Matrix:
[[4077  923]
 [ 613 4387]]
Classification Report:

```

	precision	recall	f1-score	support
0	0.87	0.82	0.84	5000
1	0.83	0.88	0.85	5000
accuracy			0.85	10000
macro avg	0.85	0.85	0.85	10000
weighted avg	0.85	0.85	0.85	10000

Figure 115

Accuracy and Confusion Matrix are chosen as the two main evaluation metrics for model evaluation for this project. Accuracy is defined as the percentage of correct predictions made by our classification model (Mankad, 2020). A confusion matrix summarizes the performance of a classification model in terms of True positive, True negative, False Positive and False Negative. The model trained is able to obtain an overall accuracy of 0.8454 on the test set. In terms of confusion matrix, the model identified 4077 true Negatives, 923 false positives. 613 false negatives, and 4387 true positives. Additional classification reports including precision, recall, F1-score and support are also provided. Below shows the plotting of the confusion matrix.

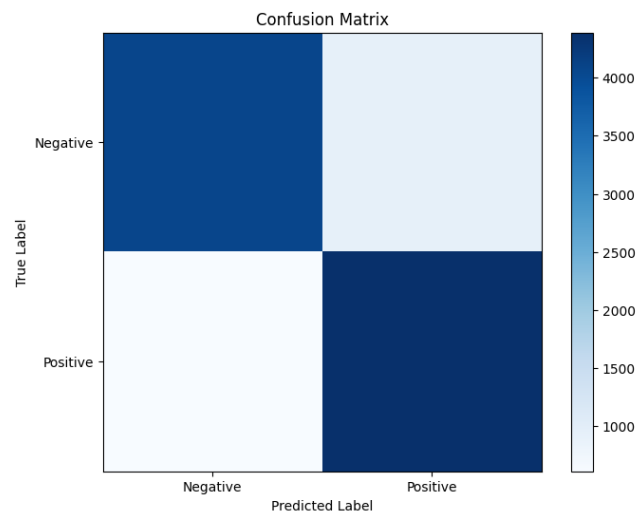


Figure 116

As shown in the diagrams above, the model trained is able to obtain an overall accuracy of 0.8454 on the test set. In terms of confusion matrix, the model identified 4077 true Negatives, 923 false positives, 613 false negatives, and 4387 true positives. For precision, which is the accuracy of positive predictions. For class 0, the precision is 0.87, indicating that 87% of the samples predicted as class 0 are actually true negatives. For class 1, the precision is 0.83, meaning that 83% of the samples predicted as class 1 are actually true positives. For Recall which represents the ability of the model to correctly identify positive samples. For class 0, the recall is 0.82, indicating that 82% of the actual class 0 samples were correctly identified. For class 1, the recall is 0.88, meaning that 88% of the actual class 1 samples were correctly identified. For F1-score, which is the harmonic mean of precision and recall. For class 0, the F1-score is 0.84, and for class 1, the F1-score is 0.85. For Support that = represents the number of samples in each class. For class 0, the support is 5000, and for class 1, the support is also 5000. The Macro Avg and Weighted Avg metrics provide the average values across all classes (0 and 1). In this result, both macro avg and weighted avg have an accuracy, precision, recall, and F1-score of approximately 0.85.

Naïve Bayes: Ng Zhi Yao

Best Hyperparameters: {'priors': [0.2, 0.8], 'var_smoothing': 0.0023101297000831605}				
ROC-AUC Score: 0.8138244773993926				
Train Accuracy: 0.81805				
	precision	recall	f1-score	support
0	0.82	0.82	0.82	19965
1	0.82	0.82	0.82	20035
accuracy			0.82	40000
macro avg	0.82	0.82	0.82	40000
weighted avg	0.82	0.82	0.82	40000

Test Accuracy: 0.8138				
	precision	recall	f1-score	support
0	0.82	0.81	0.81	5035
1	0.81	0.82	0.81	4965
accuracy			0.81	10000
macro avg	0.81	0.81	0.81	10000
weighted avg	0.81	0.81	0.81	10000
#	-----			#

Figure 117: Report for Both Training and Testing Set

Through the report, the team can figure out the best hyperparameters after fine tuning for the model. Apart from that, ROC-AUC score is also being displayed (0.8138) which indicates a reasonable level of discrimination. In general, this metric is a plot used to determine the true positive (TP) rate against the false positive (FP) rate. A higher score will indicate better discrimination between both positive and negative classes. Apart from that, this model has an accuracy of 81.805% which indicates that it can predict most cases. By looking at the classification report, it can discover that the model has great recall and f1-score which is a good sign for a model.

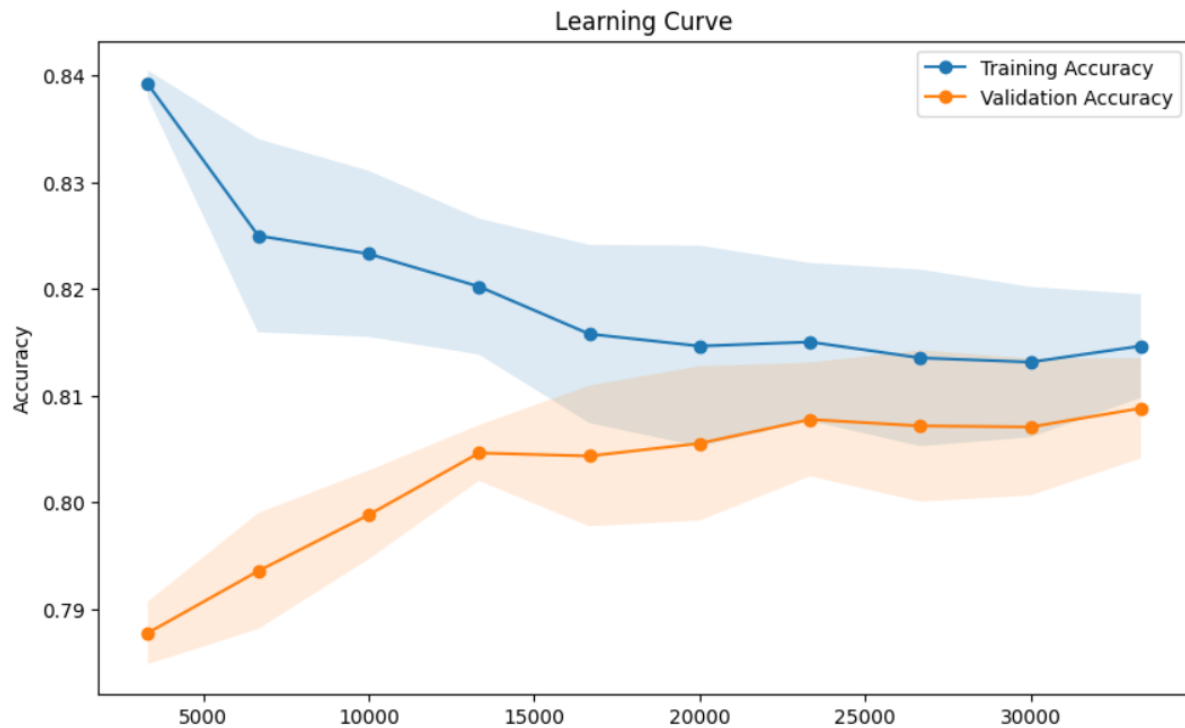


Figure 118: Learning Curve of Training and Validation Set

Through the image above, it is known that the training and validation accuracy is getting consistent and stable with the increase of training data. It is seen that the training accuracy is the highest at the start as the model learns the training data too well which includes noise and random fluctuations. Hence, it led to high accuracy on the training set but poor generalization to unseen data.

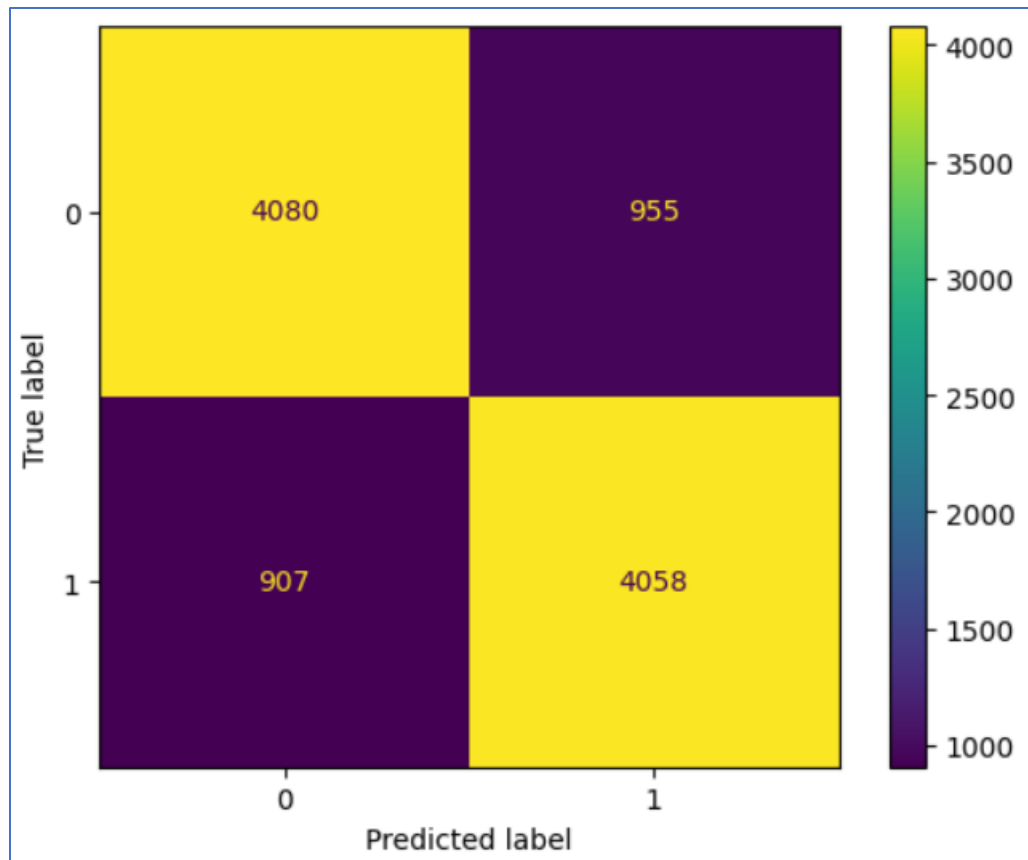


Figure 119: Confusion Matrix for Naive Bayes Model

By checking on the diagram above, the team is able to figure out true negative (TN), false positive (FP), false negatives (FN) and true positive (TP). In this case, it is known that 907 cases are misidentified as a negative case which is also known as FN. By using this information, the team can calculate various evaluation metrics such as accuracy and F1-score to assess the performance of the classification model. This is because the metrics can provide insight into how well the model is performing in terms of correctly classifying instances into their respective classes and identify potential issues such as imbalances or biases in the predictions.

Picking best model with discussions and analyses

In terms of test accuracies, LSTM got the highest (86.47%), followed by SVM (86.12%), followed by CNN (84.54%), lastly followed by Naïve Bayes (81.38%). In other words, LSTM has the highest generalizability on the dataset due to its highest test accuracy, but this is slightly higher than the test accuracy of SVM only. If train accuracy is included into the discussion, then LSTM has way higher train accuracy than SVM, that is LSTM got 96.95% while the SVM got 86.7%.

In terms of confusion matrix, LSTM seems better at classifying true negatives (4425) compared to true positives (4222), while SVM seems better at classifying true positives (4424) compared to true negatives (4188). CNN is the most imbalanced when it comes to getting true positives (4387) and true negatives (4077), with the largest difference 310. Although Naïve Bayes does not perform well in terms of test accuracy, its capability is quite balanced because it managed to get similar correct predictions for positive reviews (4058) and negative reviews (4080).

Speaking of model complexity and interpretability, LSTM and CNN are under deep learning, hence they have way more complicated architecture than SVM and Naïve Bayes. Complicated architecture allows them to capture complex relationships in the data, but it also comes at the cost of worse interpretability. SVM and Naïve Bayes offer better interpretability because they are under traditional machine learning.

Speaking of training and inference speed, SVM and Naïve Bayes are generally faster to train and make predictions if compared to the LSTM and CNN. LSTM and CNN even require GPU so that they can make predictions at a more acceptable speed, but still slower than SVM and Naïve Bayes.

Since there is no requirement about inference speed and interpretability in this assignment, prediction accuracy will be the only factor for picking the best-performing model. LSTM is finally picked as the best-performing model due to its highest test accuracy, which is 86.47%.

EXTRA FEATURES

One unique aspect of this project is the use of deep learning techniques, namely an LSTM classifier, to do sophisticated sentiment analysis. In comparison to state-of-the-art machine learning methods, this is a huge improvement. An extensive exploratory data analysis (EDA) of the IMDB dataset is also conducted as part of this project. This exemplifies a deep comprehension of the facts that goes beyond basic processing. In addition, by combining SVM with 'LinearSVC' and other machine learning models for comparison analysis, one may demonstrate a dedication to investigating several approaches to identify the best one. An in-depth exploration of model tuning for best performance is shown by the advanced hyperparameter optimization for the SVM model utilizing the 'RandomizedSearchCV'. Finally, a comprehensive study of efficacy and areas for development may be obtained by critical review and comparison of model findings with earlier efforts. Academic rigor and subject-matter mastery are on full display here.

References

- abhishek. (n.d.). *SGD with momentum : How is it different with SGD ?* Retrieved from datasciencelearner: <https://www.datasciencelearner.com/deep-learning/sgd-with-momentum/>
- BotPenguin. (n.d.). *Stemming*. Retrieved from Bot Penguin: <https://botpenguin.com/glossary/stemming>
- Doshi, S. (2021, April 6). *Cyclical Learning Rates*. Retrieved from medium: <https://medium.com/analytics-vidhya/cyclical-learning-rates-a922a60e8c04#:~:text=A%20very%20high%20learning%20rate%20causes%20the%20model%20to%20fluctuate,converge%20to%20the%20local%20minima.>
- Ganesan, K. (2024). *What Are Stop Words?* Retrieved from Opinosis Analytics: <https://www.opinosis-analytics.com/knowledge-base/stop-words-explained/>
- GeekforGeeks. (n.d.). *Removing stop words with NLTK in Python*. Retrieved from GeekforGeeks: <https://www.geeksforgeeks.org/removing-stop-words-nltk-python/>
- Hazem, K. (2023). *Sentiment Analysis using LSTM*. Retrieved from kaggle: <https://www.kaggle.com/code/karimhazem/sentiment-analysis-using-lstm>
- Jain, S. (2024, January 3). *Introduction to Stemming*. Retrieved from GeeksforGeeks: <https://www.geeksforgeeks.org/introduction-to-stemming/>
- Keras. (n.d.). *Conv1D layer*. Retrieved from Keras: https://keras.io/api/layers/convolution_layers/convolution1d/
- Mankad, S. (2020). *A Tour of Evaluation Metrics for Machine Learning*. Retrieved from Analytics Vidhya: <https://www.analyticsvidhya.com/blog/2020/11/a-tour-of-evaluation-metrics-for-machine-learning/>
- Maulana. (2020). Improved Accuracy of Sentiment Analysis Movie Review Using Support Vector Machine Based Information Gain. *Journal of Physics*, 7.
- Mazumder, S. (2023, September 27). *5 Techniques to Handle Imbalanced Data For a Classification Problem*. Retrieved from Analytics Vidhya:

<https://www.analyticsvidhya.com/blog/2021/06/5-techniques-to-handle-imbalanced-data-for-a-classification-problem/>

Park, S. (2021). *A 2021 Guide to improving CNNs-Optimizers: Adam vs SGD*. Retrieved from Medium: <https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008>

Qaisar, S. M. (2020). Sentiment Analysis of IMDb Movie Reviews Using Long Short-Term Memory. *2020 2nd International Conference on Computer and Information Sciences (ICCIS)*. doi:10.1109/ICCIS49240.2020.9257657

Raj, N. (2023, November 6). *Starters Guide to Sentiment Analysis using Natural Language Processing*. Retrieved from Analytics Vidhya: <https://www.analyticsvidhya.com/blog/2021/06/nlp-sentiment-analysis/>

Ramadhan. (2021). Analysis Sentiment based on IMDB aspects from movie reviews using SVM. *SinkrOn*, 7.

Sathe, M. R. (2023). *Fine-Tuning Convolutional Neural Networks: A Guide to Hyperparameters in CNNs with Python and Keras*. Retrieved from Medium: <https://medium.com/@sathemayuri52/fine-tuning-convolutional-neural-networks-a-guide-to-hyperparameters-in-cnns-with-python-and-keras-a6cc52926ec2>

Srivastava, T. (2024). *12 Important Model Evaluation Metrics for Machine Learning Everyone Should Know (Updated 2023)*. Retrieved from Analytics Vidhya: <https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/>

Sydney, F. (2020). *Tokenization and Filtering Stopwords with the Text Pre-Processing Tool*. Retrieved from Alteryx: <https://community.alteryx.com/t5/Data-Science/Tokenization-and-Filtering-Stopwords-with-the-Text-Pre/ba-p/607660>

Wu. (n.d). Machine Learning based classification for Sentimental analysis of IMDB reviews. *CS229 Stanford* , 6.