CSCI544: APPLIED NATURAL LANGUAGE PROCESSING

HOMEWORK ASSIGNMENT – 1 – 01/25/2023

Text Classification for Amazon Reviews

Name: Reuel Samuel Sam
Email: rsam@usc.edu

I. Problem Statement

This assignment has tasked the students with comparing different classification models and their performance when it comes to Text Classification with regards to Amazon Jewelry/Beauty Product Reviews.

II. Libraries Used

For the assignment, Python 3.10.8 and the following libraries were used:

- pandas
- numpy
- nltk
- sklearn
- bs4 *(not used but was provided in the ipynb given for the assignment)*
- contractions

III. Dataset Preparation

The provided PDF (and Piazza responses) instructs us to make use or 'reviews' and 'star rating' columns for training purposes. Since no specification is provided considering the use which review column – review_body or review_heading – this submission has made use of both columns. This is achieved by concatenating review_body TO review_heading separated by a fullstop/period. To differentiate between review_heading and review_body, the review_heading is capitalized while the review_body is left as lowercase (this is done since TF-IDF is case sensitive). This concatenation, however, is only done at the end of both data cleaning and preprocessing since review_headline and review_body are processed individually.

Furthermore, the classification task is multi-class in nature with 3 target classes using assignClass function:

- Class 1: Contains reviews with Star Ratings of 1 and 2 with a total of 20,000 elements (done using sample())
- Class 2: Contains reviews with Star Ratings of 3 with a total of 20,000 elements (done using sample())

- Class 3: Contains reviews with Star Ratings of 4 and 5 with a total of 20,000 elements (done using sample())

IV. Data Cleaning

For data cleaning, the following preprocessing has been done for both review_body and review_headline:

- Texts were first converted to all lower cases (done since capitalization does not affect sentiment, at least when TF-IDF is used)
- Any URLs present in the dataset set were removed since they do not provide any useful information regarding sentiment
- All E-mail IDs were also removed since they do not provide any useful information regarding sentiment
- Additionally, any HTML tags that may have been present have also been removed since they do not provide any useful information regarding sentiment
- Contractions (if present) have been expanded (done using python's contraction package)
- All non Alpha-numeric characters have also been removed since these also do not provide any useful data regarding sentiment
- Any isolated characters that exist were also removed (whether at the start or in between a sentence)
- Consecutive repeated words have also been omitted and replaced by just a single occurrence of word
- Furthermore, the succeeding words that follow any negation words (not, never, no) have been substituted with 'NEG_' along with the corresponding word until the end of the sentence to indicate the sentiment of the review (done in accordance with 'Speech and Language Processing' by Daniel Jurafsky Section 4.4)
- Finally, all extra white spaces were removed since they do not provide any information about sentiment of the review.

At this point, the review_body is concatenated to review_headline into a new variable known as reviews_vanilla.

Average length of each review BEFORE data cleaning: 312.3514
Average length of each review AFTER data cleaning: 336.12825

V. Pre-processing

The pre-processing steps that were to be followed involve Stop Word Removal and Lemmatization for both review_body and review_headline.

Stop Word Removal:

For this, in initial attempts, all stop words except for 'not' were removed since the presence of a negation may influence the sentiment of the review. However, upon experimentation, it was noticed that the models perform better if the stop words were left untouched.

Lemmatization:

Similarly, when it comes to lemmatization, initial attempts had made use of NLTK's WordNet Lemmatizer. However, certain words were not lemmatized correctly since they have ambiguous tags based on context. To resolve this, the reviews (body and headline) were first tagged with Part-Of-Speech tags using NLTK's pos_tag method. The word and its corresponding tag were then converted to WordNet format before being provided as arguments to the Lemmatizer.

On inspecting the dataset after the use of POS tagging and Lemmatization, it was observed that the words were being represented by their root words more often than not. However, just as in the case of Stop Word Removal, upon experimentation, it was noticed that all the models perform better when the reviews were left as it is, without lemmatization. The comparisons can be seen in the tables below (fitted with TF-IDF vectorizer that uses 5000 features).

Stopword Removal:	**Yes**	**Lemmatization:**	**Yes**	**Features:**	**5000**
Model	Accuracy		Scores		
	Train	Test	Precision	Recall	F1
Perceptron	0.7843958333	0.7101666667	0.7116164939	0.7101666667	0.7101116271
SVM	0.8335208333	0.7594166667	0.7579174386	0.7594166667	0.7585006644
Logistic Regression	0.820625	0.7716666667	0.7714081596	0.7716666667	0.7715280819
Naive Bayes	0.7610625	0.7301666667	0.7309591179	0.7301666667	0.7305105435

Table 1: Results using both Stop word Removal and Lemmatization for 5000 features

Stopword Removal:	**Yes**	**Lemmatization:**	**No**	**Features:**	**5000**

Model	Accuracy		Scores		
	Train	Test	Precision	Recall	F1
Perceptron	0.7892916667	0.7208333333	0.7188009795	0.7208333333	0.7180206969
SVM	0.8377708333	0.766	0.7644389668	0.766	0.7650124404
Logistic Regression	0.8255625	0.7741666667	0.7736375278	0.7741666667	0.7738747659
Naive Bayes	0.76725	0.7385833333	0.7395782064	0.7385833333	0.7390154656

Table 2: Results using only Stop word Removal for 5000 features

Stopword Removal:	**No**	**Lemmatization:**	**Yes**	**Features:**	**5000**

Model	Accuracy		Scores		
	Train	Test	Precision	Recall	F1
Perceptron	0.7965833333	0.7215833333	0.7251862771	0.7215833333	0.7226009519
SVM	0.83675	0.7665	0.7658802911	0.7665	0.7661491511
Logistic Regression	0.823	0.7749166667	0.7753135698	0.7749166667	0.7751016378
Naive Bayes	0.7629166667	0.73175	0.7326786412	0.73175	0.7321358783

Table 3: Results using only Lemmatization for 5000 features

Stopword Removal:	**No**	**Lemmatization:**	**No**	**Features:**	**5000**

Model	Accuracy		Scores		
	Train	Test	Precision	Recall	F1
Perceptron	0.78875	0.7216666667	0.7253814604	0.7216666667	0.715790025
SVM	0.8466666667	0.7755833333	0.7743340339	0.7755833333	0.7748469486
Logistic Regression	0.830375	0.7835833333	0.7834071727	0.7835833333	0.7834872945
Naive Bayes	0.7702083333	0.7453333333	0.7457754334	0.7453333333	0.7454476261

Table 4: Results without using Stop Word Removal or Lemmatization for 5000 features

As seen from the above tables, results have improved after stop word removal and lemmatization were omitted from the preprocessing step. Ideally, since the results do not vary much between **Table-3** and **Table-**4, keeping lemmatization may seem beneficial. However, I have decided to omit lemmatization since this takes roughly 4 minutes to execute on my PC. Since preprocessing step has been completely omitted, there is no change in the average length of each review.

If pre-processing has been omitted:

Average length of each review BEFORE pre-processing: 336.12825

Average length of each review AFTER pre-processing: 336.12825

If pre-processing is done:

Average length of each review BEFORE pre-processing: 336.12825
Average length of each review AFTER pre-processing: 258.05358333333334

As proof of implementation, the .ipynb file appended to this report implements both **Table-1** and **Table-4.** *'_vanilla'* is used to indicate all the implementations that do not involve both stop word removal and lemmatization. Likewise, *'_processed'* is used to indicate all the implementations that do not involve both stop word removal and lemmatization.

NOTE: *Please note that these steps were skipped in the .py file as per an instructor's response to Piazza Question @116 that we are allowed skip steps if justification is provided. Additionally, Prof. Rostami has permitted the skipping of these steps as per Piazza Private Question @194 that I had posted.*

At this point, the review_body is concatenated to review_headline into a new variable known as reviews_processed.

VI. TF-IDF Feature Extraction

For feature extraction, as instructed, TF-IDF has been adopted. However, before continuing with TF-IDF feature extraction, the dataset was divided into train and test data using Sklearn's train_test_split method (done for both reviews_vanilla and reviews_processed).

```
X_train_vanilla, X_test_vanilla, Y_train_vanilla, Y_test_vanilla = train_test_split(reviews_vanilla,
df['class'], test_size = 0.2, random_state = 0, stratify=df['class'])
```

The train data contains 80% of the total dataset while the test data contains 20% of the dataset. The stratify parameter ensures that the split up of number of instances per class is equivalent within the train and test data. With this split up, the following describes the number of instances per class:

- Test Set
  - Class 1: 16000; Class 2: 16000; Class 3: 16000
- Train Set
  - Class 1: 4000; Class 2: 4000; Class 3: 4000

After the split, I have fit a TF-IDF Vectorizer (taken from Sklearn) on the train dataset to learn only the top 5000 features/words. This was selected upon experimentation as it yielded the best results when compared with smaller or larger values. Also during experimentation, min_df and max_df were also used, but max_features parameter remained more consistent. Using this fitted vectorizer, the test data was also transformed.

VII. Perceptron

Sklearn's inbuilt linear model Perceptron has been used for this training. As for parameters, the tolerance was set to 0.001 with a validation fraction of 0.3 so that 30% of the training set is set apart for validation.

```
prctrn_vanilla = Perceptron(tol=1e-3, random_state=0, validation_fraction=0.3)
```

The performance results for this have been tabulated below in **Table-5.**

Perceptron

Training Accuracy:	0.8009583333	Testing Accuracy:	0.7305
	Precision	Recall	F1-Score
Class 1	0.7538666667	0.70675	0.7295483871
Class 2	0.6369470548	0.69475	0.6645940452
Class 3	0.8129662979	0.79	0.8013186256
Average	0.7345933398	0.7305	0.7318203526

Table-5: Performance Results for Perceptron without stop word removal and lemmatization

VIII. SVM

Sklearn's inbuilt SVM LinearSVC has been used for this training using all the default parameters set by Sklearn.

```
SVM_vanilla = LinearSVC()
```

The performance results for this have been tabulated below in **Table-6.**

SVM

Training Accuracy:	0.8433958333	Testing Accuracy:	0.7746666667
	Precision	Recall	F1-Score
Class 1	0.773483366	0.7905	0.78189911
Class 2	0.712156863	0.681	0.696230032
Class 3	0.834352826	0.8525	0.843328799
Average	0.773331018	0.774666667	0.773819314

Table-6: Performance Results for SVM without stop word removal and lemmatization

IX. Logistic Regression

Sklearn's inbuilt linear model LogisticRegression has been used for this training using all the default parameters set by Sklearn apart from setting multi_class to 'multinomial' since this is not a binary classification task. Additionally, max_iter was updated to 10000 since the default number of iterations (100) did not lead to convergence.

```
lr_vanilla = LogisticRegression(multi_class='multinomial', max_iter=10000)
```

The performance results for this have been tabulated below in **Table-7.**

Logistic Regression

Training Accuracy:	0.8304166667	Testing Accuracy:	0.7869166667
	Precision	Recall	F1-Score
Class 1	0.788012773	0.802	0.794944864
Class 2	0.717676768	0.7105	0.714070352
Class 3	0.854875283	0.84825	0.851549755
Average	0.786854941	0.786916667	0.78685499

Table-7: Performance Results for Logistic Regression without stop word removal and lemmatization

X. Naïve Bayes

Sklearn's inbuilt Naïve Bayes model MultinomialNB has been used for this training using all the default parameters set by Sklearn.

```
MNB_vanilla = MultinomialNB()
```

The performance results for this have been tabulated below in **Table-8.**

Naive Bayes

Training Accuracy:	0.7681458333	Testing Accuracy:	0.7414166667
	Precision	Recall	F1-Score
Class 1	0.767310167	0.74525	0.756119214
Class 2	0.66715293	0.686	0.676445211
Class 3	0.792603698	0.793	0.7928018
Average	0.742355598	0.741416667	0.741788742

Table-8: Performance Results for Naïve Bayes without stop word removal and lemmatization

Imports

In [1]:

```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
import re
from bs4 import BeautifulSoup
```

```
c:\Users\reuel\anaconda3\envs\pytorch\lib\site-packages\scipy\__init__.py:146: UserWarning: A NumPy vers
ion >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version 1.23.5
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\reuel\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

In [2]:

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
```

Read Data

In [3]:

```
dataframe = pd.read_csv("dataset/amazon_reviews_us_Beauty_v1_00.tsv", sep="\t", on_bad_lines='skip')
```

```
C:\Users\reuel\AppData\Local\Temp\ipykernel_19364\47645999.py:1: DtypeWarning: Columns (7) have mixed ty
pes. Specify dtype option on import or set low_memory=False.
  dataframe = pd.read_csv("dataset/amazon_reviews_us_Beauty_v1_00.tsv", sep="\t", on_bad_lines='skip')
```

Keep Reviews and Ratings

In [4]:

```
df = dataframe.loc[:, ['star_rating', 'review_body', 'review_headline']]
df['review_body'] = df['review_body'].astype(str)
df['review_headline'] = df['review_headline'].astype(str)
```

We form three classes and select 20000 reviews randomly from each class.

In [5]:

```
def assignClass(rating):
    if rating == 1 or rating == 2:
        return 1
    elif rating == 3:
        return 2
    elif rating == 4 or rating == 5:
        return 3
    return -1
```

In [6]:

```
df['class'] = df['star_rating'].map(assignClass)
```

In [7]:

```
df_1 = df[df["class"] == 1].sample(n=20000)
df_2 = df[df["class"] == 2].sample(n=20000)
df_3 = df[df["class"] == 3].sample(n=20000)
# print(len(df_1), len(df_2), len(df_3))
```

In [8]:

```
df = pd.concat([df_1, df_2, df_3])
```

In [9]:

```
print("Dataframe that is being worked on:")
df.head(10)
```

```
Dataframe that is being worked on:
```

Out[9]:

	star_rating	review_body	review_headline	class
4550602	1	I am so upset and wanted to return. However th...	picture shows 17 brushes.the titel says 13 pc ...	1
2368402	2	horrid name! kids like but why the name?	kids like but why the name	1
2184565	2	Its very thick and I could have gotten it chea...	Two Stars	1
3878729	1	Do not order this product it is not the origin...	FAKE!!!	1
3013017	1	Ive waxed before so im not new to painfully ri...	DONT BUY THIS!!	1
3635167	1	The scarf does not look anything like the pict...	DOES NOT LOOK LIKE THE PICTURE!	1
3572617	1	A friend bought some of this sunscreen for me,...	Very bad experience	1
3530113	2	I developed a fondness for the June Jacobs Gre...	Underwhelmed	1
4966424	2	This stuff works, but be careful! it actually ...	burned my gum	1
4183200	2	Doesn't heat up enough. I had it on my hair f...	Leaves much to be desired	1

In [10]:

```
# calculate length of raw reviews before any cleaning or preprocessing
len_before_cleaning = (df['review_headline'] + ". " + df['review_body']).apply(len).mean()
```

Data Cleaning

In [11]:

```
import contractions
```

In [12]:

```
def clean_data(text):
    text = str(text)

    # convert text to lowercase
    text = text.lower()

    # remove urls

    text = re.sub(r'(http(s)?:\/\/.)?(www\.)?[-a-zA-Z0-9@:%._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%
_\+.~#?&//=]*)',' ',text)
    # remove email ids
    text = re.sub(r'([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9_-]+)',' ',text)
    # html tag
    text = re.sub('<[^<]+?>', '', text)


    # expand contractions
    text = contractions.fix(text)

    # replace non aplha numeric characters
    text = re.sub(r'[^a-zA-Z0-9. ]',' ',text)

    # remove isolated characters
    text = re.sub(r'\s+[a-zA-Z]\s+', ' ', text)
    # remove consecutively repeating words

    text = re.sub(r'\b(\w+)(?:\W+\1\b)+', r'\1', text, flags=re.IGNORECASE)
    # replace every word following not/never/no as NEG_word until a fullstop is found
    text = re.sub(r'\b(?:not|never|no)\b[\w\s]+[^\w\s]',lambda match: re.sub(r'(\s+)(\w+)', r'\1NEG_\2'
, match.group(0)), text, flags=re.IGNORECASE)

    # remove extra spaces
    text = re.sub(r'\s+', ' ', text)
```

```
    return text
```

In [13]:

```
review_body = df["review_body"].apply(clean_data)
review_headline = df["review_headline"].apply(clean_data)
```

Comparing length of raw uncleaned review with length of cleaned review

In [14]:

```
len_after_cleaning = (review_headline + ". " + review_body).apply(len).mean()
print(f"{len_before_cleaning}, {len_after_cleaning}")
```

```
313.3815, 337.23428333333334
```

In [15]:

```
# store the cleaned data that will NOT be pre-processed
reviews_vanilla = review_headline.str.upper() + ". " + review_body
```

Pre-processing

remove the stop words

In [16]:

```
from nltk.corpus import stopwords
nltk.download('stopwords')
stop_words = stopwords.words('english')
stop_words.remove('not')

def remove_stop_words(text):
    tokens = [w for w in text.split() if not w in stop_words]


    text = " ".join(tokens)

    return text
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\reuel\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [17]:

```
review_body = review_body.apply(remove_stop_words)
review_headline = review_headline.apply(remove_stop_words)
```

perform lemmatization

In [18]:

```
from nltk.corpus import wordnet

def get_wordnet_pos(treebank_tag):

    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return ''
```

In [19]:

```
from nltk.stem import WordNetLemmatizer
from nltk.tag import pos_tag
nltk.download('averaged_perceptron_tagger')
nltk.download('omw-1.4')

lemmatizer = WordNetLemmatizer()
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\reuel\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\reuel\AppData\Roaming\nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
```

In [20]:

```
def lemmatization(reviews):
    lemmatized_reviews = []
    for sent in reviews.values:
        tagged = pos_tag(sent.split(" "))
        temp_review = ""
        for token in tagged:
            tag = get_wordnet_pos(token[1])
            if tag:
                temp_review += lemmatizer.lemmatize(token[0], tag)
            else:
                temp_review += token[0]
            temp_review += " "

        lemmatized_reviews.append(temp_review)

    return  pd.Series(lemmatized_reviews)
```

In [21]:

```
review_body = lemmatization(review_body)
review_headline = lemmatization(review_headline)
```

In [22]:

```
# store the cleaned and pre-processed data
reviews_processed = review_headline.str.upper() + ". " + review_body
```

Comparing length of review that is NOT preprocessed with length of review before preprocessing

In [23]:

```
len_without_preprocessing = reviews_vanilla.apply(len).mean()

print(f"{len_after_cleaning}, {len_without_preprocessing}")
```

```
337.23428333333334, 337.23428333333334
```

In [24]:

```
print("Sample of cleaned Data:")
reviews_vanilla.head(10)
```

```
Sample of cleaned Data:
```

Out[24]:

```
4550602    PICTURE SHOWS 17 TITEL SAYS 13 PC AND RECEIVED...
2368402    KIDS LIKE BUT WHY THE NAME. horrid name kids l...
2184565    TWO STARS. its very thick and could have gotte...
3878729    FAKE . do not NEG_order NEG_this NEG_product N...
3013017    DO NOT BUY THIS . i have waxed before so am no...
3635167    DOES NOT LOOK LIKE THE PICTURE . the scarf doe...
3572617    VERY BAD EXPERIENCE. a friend bought some of t...
3530113    UNDERWHELMED. i developed fondness for the jun...
4966424    BURNED MY GUM. this stuff works but be careful...
4183200    LEAVES MUCH TO BE DESIRED. does not NEG_heat N...
dtype: object
```

Comparing length of review that is preprocessed with length of review before preprocessing

In [25]:

```
len_after_preprocessing = reviews_processed.apply(len).mean()

print(f"{len_after_cleaning}, {len_after_preprocessing}")
```

```
337.23428333333334, 249.59886666666668
```

In [26]:

```
print("Sample of preprocessed Data:")
reviews_processed.head(10)
```

```
Sample of preprocessed Data:
```

Out[26]:

```
0    PICTURE SHOW 17 TITEL SAY 13 PC RECEIVE 12 . u...
1           KID LIKE NAME . horrid name kid like name
2    TWO STAR . thick could get cheap local beauty ...
3    FAKE . not NEG_order NEG_this NEG_product NEG_...
4    NOT BUY . waxed not NEG_new NEG_to NEG_painful...
5    NOT LOOK LIKE PICTURE . scarf not look anythin...
6    BAD EXPERIENCE . friend buy sunscreen know yea...
7    UNDERWHELMED . develop fondness june jacob gre...
8    BURN GUM . stuff work careful actually burn gu...
9    LEAF MUCH DESIRE . not NEG_heat NEG_up NEG_eno...
dtype: object
```

TF-IDF Feature Extraction

In [27]:

```
from sklearn.model_selection import train_test_split
X_train_vanilla, X_test_vanilla, Y_train_vanilla, Y_test_vanilla = train_test_split(reviews_vanilla, df
['class'], test_size = 0.2, random_state = 0, stratify=df['class'])
```

In [28]:

```
X_train_processed, X_test_processed, Y_train_processed, Y_test_processed = train_test_split(reviews_pro
cessed, df['class'], test_size = 0.2, random_state = 0, stratify=df['class'])
```

In [29]:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tf_idf_vect_vanilla = TfidfVectorizer(use_idf=True, max_features=5000)
X_train_tfidf_vanilla = tf_idf_vect_vanilla.fit_transform(X_train_vanilla)
X_test_tfidf_vanilla = tf_idf_vect_vanilla.transform(X_test_vanilla)
```

In [30]:

```
print("For cleaned/vanilla data:")
print(f"\tX_train shape: {X_train_tfidf_vanilla.shape}")
print(f"\tY_train shape: {Y_train_vanilla.shape}\n")

print(f"\tX_test shape: {X_test_tfidf_vanilla.shape}")
print(f"\tY_test shape: {Y_test_vanilla.shape}\n")
```

```
For cleaned/vanilla data:
 X_train shape: (48000, 5000)
 Y_train shape: (48000,)

 X_test shape: (12000, 5000)
 Y_test shape: (12000,)
```

In [31]:

```
tf_idf_vect_processed = TfidfVectorizer(use_idf=True, max_features=5000)
X_train_tfidf_processed = tf_idf_vect_processed.fit_transform(X_train_processed)
X_test_tfidf_processed = tf_idf_vect_processed.transform(X_test_processed)
```

In [32]:

```
print("For pre-processed data:")
```

```
print(f"\tX_train shape: {X_train_tfidf_processed.shape}")
print(f"\tY_train shape: {Y_train_processed.shape}\n")

print(f"\tX_test shape: {X_test_tfidf_processed.shape}")
print(f"\tY_test shape: {Y_test_processed.shape}\n")
```

```
For pre-processed data:
 X_train shape: (48000, 5000)
 Y_train shape: (48000,)

 X_test shape: (12000, 5000)
 Y_test shape: (12000,)
```

Perceptron

Without Stop Word Removal and Without Lemmatization

In [33]:

```
from sklearn.linear_model import Perceptron

prctrn_vanilla = Perceptron(tol=1e-3, random_state=0, validation_fraction=0.3)
prctrn_vanilla.fit(X_train_tfidf_vanilla, Y_train_vanilla)

# training accuracy
print(prctrn_vanilla.score(X_train_tfidf_vanilla, Y_train_vanilla))

# testing accuracy
print(prctrn_vanilla.score(X_test_tfidf_vanilla, Y_test_vanilla))
```

```
0.7999166666666667
0.7331666666666666
```

In [34]:

```
# classification report
prctrn_pred_vanilla = prctrn_vanilla.predict(X_test_tfidf_vanilla)
print(classification_report(Y_test_vanilla, prctrn_pred_vanilla))

# class-wise precision, recall and f1-score
prctrn_report_vanilla = classification_report(Y_test_vanilla, prctrn_pred_vanilla, output_dict=True)
print(f"{prctrn_report_vanilla['1']['precision']}, {prctrn_report_vanilla['1']['recall']}, {prctrn_repor
t_vanilla['1']['f1-score']}")
print(f"{prctrn_report_vanilla['2']['precision']}, {prctrn_report_vanilla['2']['recall']}, {prctrn_repor
t_vanilla['2']['f1-score']}")
print(f"{prctrn_report_vanilla['3']['precision']}, {prctrn_report_vanilla['3']['recall']}, {prctrn_repor
t_vanilla['3']['f1-score']}")
print(f"{prctrn_report_vanilla['macro avg']['precision']}, {prctrn_report_vanilla['macro avg']['recall']
}, {prctrn_report_vanilla['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.74      0.73      0.73      4000
           2       0.67      0.63      0.65      4000
           3       0.78      0.84      0.81      4000

    accuracy                           0.73     12000
   macro avg       0.73      0.73      0.73     12000
weighted avg       0.73      0.73      0.73     12000

0.7402696514881709, 0.7275, 0.7338292775185978
0.6719083155650319, 0.63025, 0.6504127966976265
0.7799397729905027, 0.84175, 0.8096669472165444
0.7307059133479018, 0.7331666666666666, 0.7313030071442562
```

With Stop Word Removal and Lemmatization

In [35]:

```
prctrn_processed = Perceptron(tol=1e-3, random_state=0, validation_fraction=0.3)
prctrn_processed.fit(X_train_tfidf_processed, Y_train_processed)

# training accuracy
```

```
print(prctrn_processed.score(X_train_tfidf_processed, Y_train_processed))

# testing accuracy
print(prctrn_processed.score(X_test_tfidf_processed, Y_test_processed))
```

```
0.7800416666666666
0.7121666666666666
```

In [36]:

```
# classification report
prctrn_pred_processed = prctrn_processed.predict(X_test_tfidf_processed)
print(classification_report(Y_test_processed, prctrn_pred_processed))

# class-wise precision, recall and f1-score
prctrn_report_processed = classification_report(Y_test_processed, prctrn_pred_processed, output_dict=Tr
ue)
print(f"{prctrn_report_processed['1']['precision']}, {prctrn_report_processed['1']['recall']}, {prctrn_r
eport_processed['1']['f1-score']}")
print(f"{prctrn_report_processed['2']['precision']}, {prctrn_report_processed['2']['recall']}, {prctrn_r
eport_processed['2']['f1-score']}")
print(f"{prctrn_report_processed['3']['precision']}, {prctrn_report_processed['3']['recall']}, {prctrn_r
eport_processed['3']['f1-score']}")
print(f"{prctrn_report_processed['macro avg']['precision']}, {prctrn_report_processed['macro avg']['reca
ll']}, {prctrn_report_processed['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.68      0.77      0.72      4000
           2       0.66      0.57      0.61      4000
           3       0.79      0.79      0.79      4000

    accuracy                           0.71     12000
   macro avg       0.71      0.71      0.71     12000
weighted avg       0.71      0.71      0.71     12000

0.6790177592633194, 0.77425, 0.7235136082233383
0.6633663366336634, 0.5695, 0.6128598331988162
0.7917602996254681, 0.79275, 0.7922548407245471
0.7113814651741502, 0.7121666666666666, 0.7095427607155672
```

SVM

Without Stop Word Removal and Without Lemmatization

In [37]:

```
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score

SVM_vanilla = LinearSVC()
SVM_vanilla.fit(X_train_tfidf_vanilla, Y_train_vanilla)

# training accuracy
print(SVM_vanilla.score(X_train_tfidf_vanilla, Y_train_vanilla))

# test accuracy
print(SVM_vanilla.score(X_test_tfidf_vanilla, Y_test_vanilla))
```

```
0.843125
0.77325
```

In [38]:

```
# classification report
SVM_pred_vanilla = SVM_vanilla.predict(X_test_tfidf_vanilla)
print(classification_report(Y_test_vanilla, SVM_pred_vanilla))

# class-wise precision, recall and f1-score
SVM_report_vanilla = classification_report(Y_test_vanilla, SVM_pred_vanilla, output_dict=True)
print(f"{SVM_report_vanilla['1']['precision']}, {SVM_report_vanilla['1']['recall']}, {SVM_report_vanilla
['1']['f1-score']}")
print(f"{SVM_report_vanilla['2']['precision']}, {SVM_report_vanilla['2']['recall']}, {SVM_report_vanilla
['2']['f1-score']}")
print(f"{SVM_report_vanilla['3']['precision']}, {SVM_report_vanilla['3']['recall']}, {SVM_report_vanilla
```

```
['3']['f1-score']}")
print(f"{SVM_report_vanilla['macro avg']['precision']}, {SVM_report_vanilla['macro avg']['recall']}, {SV
M_report_vanilla['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.77      0.78      0.78      4000
           2       0.70      0.68      0.69      4000
           3       0.84      0.86      0.85      4000

    accuracy                           0.77     12000
   macro avg       0.77      0.77      0.77     12000
weighted avg       0.77      0.77      0.77     12000

0.7717256746719485, 0.77925, 0.7754695857693744
0.7048673705897502, 0.68425, 0.6944056831155653
0.8398724865129966, 0.85625, 0.8479821738053974
0.7721551772582318, 0.77325, 0.7726191475634456
```

With Stop Word Removal and Lemmatization

In [39]:

```
SVM_processed = LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True, intercept_scaling=1,
loss='squared_hinge', max_iter=1000, multi_class='ovr', penalty='l2', random_state=None, tol=0.0001, ve
rbose=0)
SVM_processed.fit(X_train_tfidf_processed, Y_train_processed)

# training accuracy
print(SVM_processed.score(X_train_tfidf_processed, Y_train_processed))

# test accuracy
print(SVM_processed.score(X_test_tfidf_processed, Y_test_processed))
```

```
0.8331458333333334
0.7536666666666667
```

In [40]:

```
# classification report
SVM_pred_processed = SVM_processed.predict(X_test_tfidf_processed)
print(classification_report(Y_test_processed, SVM_pred_processed))

# class-wise precision, recall and f1-score
SVM_report_processed = classification_report(Y_test_processed, SVM_pred_processed, output_dict=True)
print(f"{SVM_report_processed['1']['precision']}, {SVM_report_processed['1']['recall']}, {SVM_report_pro
cessed['1']['f1-score']}")
print(f"{SVM_report_processed['2']['precision']}, {SVM_report_processed['2']['recall']}, {SVM_report_pro
cessed['2']['f1-score']}")
print(f"{SVM_report_processed['3']['precision']}, {SVM_report_processed['3']['recall']}, {SVM_report_pro
cessed['3']['f1-score']}")
print(f"{SVM_report_processed['macro avg']['precision']}, {SVM_report_processed['macro avg']['recall']}
, {SVM_report_processed['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.75      0.76      0.76      4000
           2       0.68      0.65      0.67      4000
           3       0.82      0.84      0.83      4000

    accuracy                           0.75     12000
   macro avg       0.75      0.75      0.75     12000
weighted avg       0.75      0.75      0.75     12000

0.750122970978849, 0.7625, 0.7562608480039673
0.6826597131681877, 0.6545, 0.6682833439693681
0.8236155159795072, 0.844, 0.8336831707618224
0.7521327333755147, 0.7536666666666666, 0.7527424542450527
```

Logistic Regression

Without Stop Word Removal and Without Lemmatization

In [41]:

```
from sklearn.linear_model import LogisticRegression

lr_vanilla = LogisticRegression(multi_class='multinomial', max_iter=10000)
lr_vanilla.fit(X_train_tfidf_vanilla, Y_train_vanilla)

# training accuracy
print(lr_vanilla.score(X_train_tfidf_vanilla, Y_train_vanilla))

# test accuracy
print(lr_vanilla.score(X_test_tfidf_vanilla, Y_test_vanilla))
```

```
0.8291041666666666
0.78325
```

In [42]:

```
# classification report
lr_pred_vanilla = lr_vanilla.predict(X_test_tfidf_vanilla)
print(classification_report(Y_test_vanilla, lr_pred_vanilla))

# class-wise precision, recall and f1-score
lr_report_vanilla = classification_report(Y_test_vanilla, lr_pred_vanilla, output_dict=True)
print(f"{lr_report_vanilla['1']['precision']}, {lr_report_vanilla['1']['recall']}, {lr_report_vanilla['1
']['f1-score']}")
print(f"{lr_report_vanilla['2']['precision']}, {lr_report_vanilla['2']['recall']}, {lr_report_vanilla['2
']['f1-score']}")
print(f"{lr_report_vanilla['3']['precision']}, {lr_report_vanilla['3']['recall']}, {lr_report_vanilla['3
']['f1-score']}")
print(f"{lr_report_vanilla['macro avg']['precision']}, {lr_report_vanilla['macro avg']['recall']}, {lr_
report_vanilla['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.78      0.78      0.78      4000
           2       0.71      0.71      0.71      4000
           3       0.86      0.85      0.86      4000

    accuracy                           0.78     12000
   macro avg       0.78      0.78      0.78     12000
weighted avg       0.78      0.78      0.78     12000

0.7806645016237822, 0.78125, 0.7809571410720979
0.7085813492063492, 0.71425, 0.7114043824701196
0.8617906683480454, 0.85425, 0.8580037664783428
0.7836788397260589, 0.7832500000000001, 0.78345509667352
```

With Stop Word Removal and Lemmatization

In [43]:

```
lr_processed = LogisticRegression(multi_class='multinomial', max_iter=10000)
lr_processed.fit(X_train_tfidf_processed, Y_train_processed)

# training accuracy
print(lr_processed.score(X_train_tfidf_processed, Y_train_processed))

# test accuracy
print(lr_processed.score(X_test_tfidf_processed, Y_test_processed))
```

```
0.8192916666666666
0.7645
```

In [44]:

```
# classification report
lr_pred_processed = lr_processed.predict(X_test_tfidf_processed)
print(classification_report(Y_test_processed, lr_pred_processed))

# class-wise precision, recall and f1-score
lr_pred_processed = lr_processed.predict(X_test_tfidf_processed)

lr_report_processed = classification_report(Y_test_processed, lr_pred_processed, output_dict=True)
print(f"{lr_report_processed['1']['precision']}, {lr_report_processed['1']['recall']}, {lr_report_proces
sed['1']['f1-score']}")
print(f"{lr_report_processed['2']['precision']}, {lr_report_processed['2']['recall']}, {lr_report_proces
sed['2']['f1-score']}")
print(f"{lr_report_processed['3']['precision']}, {lr_report_processed['3']['recall']}, {lr_report_proces
```

```
sed['3']['f1-score']}")
print(f"{lr_report_processed['macro avg']['precision']}, {lr_report_processed['macro avg']['recall']},
{lr_report_processed['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.77      0.77      0.77      4000
           2       0.69      0.69      0.69      4000
           3       0.84      0.84      0.84      4000

    accuracy                           0.76     12000
   macro avg       0.76      0.76      0.76     12000
weighted avg       0.76      0.76      0.76     12000

0.766541822721598, 0.7675, 0.7670206121174266
0.6879076768690416, 0.6855, 0.6867017280240422
0.8386131204789224, 0.8405, 0.8395555000624297
0.764354206689854, 0.7645, 0.7644259467346327
```

Naive Bayes

Without Stop Word Removal and Without Lemmatization

In [45]:

```
from sklearn.naive_bayes import MultinomialNB

MNB_vanilla = MultinomialNB()
MNB_vanilla.fit(X_train_tfidf_vanilla, Y_train_vanilla)

# training accuracy
print(MNB_vanilla.score(X_train_tfidf_vanilla, Y_train_vanilla))

# test accuracy
print(MNB_vanilla.score(X_test_tfidf_vanilla, Y_test_vanilla))
```

```
0.7667083333333333
0.7415833333333334
```

In [46]:

```
# classification report
MNB_pred_vanilla = MNB_vanilla.predict(X_test_tfidf_vanilla)
print(classification_report(Y_test_vanilla, MNB_pred_vanilla))

# class-wise precision, recall and f1-score
MNB_report_vanilla = classification_report(Y_test_vanilla, MNB_pred_vanilla, output_dict=True)
print(f"{MNB_report_vanilla['1']['precision']}, {MNB_report_vanilla['1']['recall']}, {MNB_report_vanilla
['1']['f1-score']}")
print(f"{MNB_report_vanilla['2']['precision']}, {MNB_report_vanilla['2']['recall']}, {MNB_report_vanilla
['2']['f1-score']}")
print(f"{MNB_report_vanilla['3']['precision']}, {MNB_report_vanilla['3']['recall']}, {MNB_report_vanilla
['3']['f1-score']}")
print(f"{MNB_report_vanilla['macro avg']['precision']}, {MNB_report_vanilla['macro avg']['recall']}, {MN
B_report_vanilla['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.76      0.74      0.75      4000
           2       0.66      0.68      0.67      4000
           3       0.80      0.81      0.80      4000

    accuracy                           0.74     12000
   macro avg       0.74      0.74      0.74     12000
weighted avg       0.74      0.74      0.74     12000

0.7639462809917356, 0.7395, 0.7515243902439025
0.6599465630313335, 0.67925, 0.6694591597880991
0.8037895786586886, 0.806, 0.804893271751342
0.742560807560586, 0.7415833333333334, 0.7419589405944479
```

With Stop Word Removal and Lemmatization

In [47]:

```
from sklearn.naive_bayes import MultinomialNB

MNB_processed = MultinomialNB()
MNB_processed.fit(X_train_tfidf_processed, Y_train_processed)

# train accuracy
print(MNB_processed.score(X_train_tfidf_processed, Y_train_processed))

# test accuracy
print(MNB_processed.score(X_test_tfidf_processed, Y_test_processed))
```

```
0.7568958333333333
0.7286666666666667
```

In [48]:

```
# classification report
MNB_pred_processed = MNB_processed.predict(X_test_tfidf_processed)
print(classification_report(Y_test_processed, MNB_pred_processed))

# class-wise precision, recall and f1-score
MNB_report_processed = classification_report(Y_test_processed, MNB_pred_processed, output_dict=True)
print(f"{MNB_report_processed['1']['precision']}, {MNB_report_processed['1']['recall']}, {MNB_report_pro
cessed['1']['f1-score']}")
print(f"{MNB_report_processed['2']['precision']}, {MNB_report_processed['2']['recall']}, {MNB_report_pro
cessed['2']['f1-score']}")
print(f"{MNB_report_processed['3']['precision']}, {MNB_report_processed['3']['recall']}, {MNB_report_pro
cessed['3']['f1-score']}")
print(f"{MNB_report_processed['macro avg']['precision']}, {MNB_report_processed['macro avg']['recall']}
, {MNB_report_processed['macro avg']['f1-score']}")
```

```
              precision    recall  f1-score   support

           1       0.76      0.73      0.74      4000
           2       0.65      0.66      0.65      4000
           3       0.78      0.80      0.79      4000

    accuracy                           0.73     12000
   macro avg       0.73      0.73      0.73     12000
weighted avg       0.73      0.73      0.73     12000

0.756078634247284, 0.73075, 0.7431985761505213
0.6491228070175439, 0.65675, 0.6529141294892506
0.781502324443357, 0.7985, 0.7899097316681093
0.7289012552360616, 0.7286666666666667, 0.7286741457692938
```