AMAZON FINE FOOD REVIEWS

Final Report

Team

Mohtih Sai Nattam (mohithsa) 50495770
Sai Durga Guru Sandeep Lankey (slankey) 50496401
Prem Salina (premsali) 50496096

Table of Contents

Problem Statement

This dataset consists of reviews of fine foods from Amazon. For a given review, determining whether the review is positive (rating of 4 or 5) or negative (rating of 1 or 2).

Background of the Problem

This data spans more than ten years and includes over 500k customer reviews as of October 2012. Reviews contain information on the product and the user, as well as ratings and simple language. Additionally, all other Amazon categories reviews are included. This dataset is available on the Kaggle website (https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews).

Significance of the problem

- The reviews in this dataset are actual consumer-generated content from the Amazon platform. This means, it reflects the language, writing style, and sentiments expressed by real people when reviewing food products.

- The dataset spans over a decade, allowing for the analysis of changing trends and sentiments over time. This longitudinal perspective can be valuable for understanding how consumer opinions about food products evolve.

Project Potential towards Sentiment Analysis

- It poses various natural language processing (NLP) challenges like dealing with stop words, stemming, and Lemmatization. This makes it a good benchmark for evaluating the robustness of sentiment analysis models.

- The dataset is not limited to a specific category of food products. It covers a wide range of food items, making it applicable to a broad spectrum of food-related sentiment analysis tasks.

- The dataset contains a wide range of opinions, from highly positive to highly negative, as well as neutral reviews. This diversity is important for training models that can handle various sentiment expressions.

Overview of Dataset

- Number of reviews: 568,454
- Number of users: 256,059
- Number of products: 74,258
- Timespan: Oct 1999 — Oct 2012
- Number of Attributes/Columns in data: 10

Attributes Information

- Id - Amazon Customer ID.
- ProductId - Unique identifier for the product.
- UserId - Unqiue identifier for the user.
- ProfileName - Profile Name of the Customer.
- HelpfulnessNumerator - Number of users who found the review helpful.
- HelpfulnessDenominator - Number of users who found the review helpful or not.
- Score - Rating between 1 to 5.
- Time - Timestamp for the review.
- Summary - Brief Summary of the review.
- Text - Text of the review.

Source for this Dataset

This dataset is available on the Kaggle website.
Link: https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews

Data Cleaning and Featurization – EDA with code and working(Phase-1)

In data cleaning we have included the following:

1. Loading / Count / Converting feature to Numeric data.

```
raw_data= pd.read_csv("/Users/mohithsainattam/Desktop/Reviews.csv")
raw_data.head()
```

```
filtered_data = raw_data[raw_data['Score']!=3]
filtered_data.shape
```

```
(525814, 10)
```

```
warnings.filterwarnings('ignore')
def partition(x):
    if x<3:
        return 'negative'
    else:
        return 'positive'
dummy_data=filtered_data['Score']
review_column_data=dummy_data.map(partition)
filtered_data['Score']=review_column_data
print(filtered_data.shape)
filtered_data.head()
```

```
filtered_data['Score'].value_counts()
```

```
positive    443777
negative     82037
Name: Score, dtype: int64
```

Firstly, we have read the data from our chosen dataset and checking the size, shape and count of the dataset. Secondly, we have converted the score feature to positive or negative from the rating score given by the customers. i.e., scores 1 and 2 are considered as 'Negative'. Scores 4 and 5 are considered as 'Positive'. Whereas score 3 is considered as 'Neutral'. There are 500K users reviewed for the product and there are 10 different type of feature columns given in our dataset.

The data is imbalanced, and dataset consist of more 5 rating reviews and more positive reviews(score>3) compared to negative reviews. we have considered score = 3 as neutral and this cannot be used to judge if user is positive or negative towards the product review. So, we eliminate or clean those data points in our dataset. We have converted scores>3 as positive and scores<3 as negative, so our problem is now a binary class classification problem.

2. Drop Duplicates.

```
duplicates = filtered_data[filtered_data.duplicated(['ProductId','UserId','ProfileName','HelpfulnessNumerator','Help
print(duplicates.shape)
duplicates
```

```
_data=filtered_data.sort_values('ProductId',axis=0,ascending=True,inplace=False,kind='quicksort',na_position='last')
sorted_data.shape)
_data.head()
```

```
(525814, 10)
```

```
#drop_duplicates
final_data=sorted_data.drop_duplicates(subset={'ProfileName','UserId','Time','Text'},keep='first',inplace=False)
print(final_data.shape)
```

```
(364173, 10)
```

Removing duplicates from our dataset and displaying the number of duplicates present in the dataset. We can see that there are 256 rows of duplicates found for one product and same user in our dataset that needs to be eliminated. From the above screen shots, we have performed the duplication removal, So, the duplicated rows have been removed and rows are reduced from 525814 to 364173.

3. Finding Invalid data and cleaning.

```
#HelpfulnessNumerator should be less than HelpfulnessDenominator. checking if any records are invalid with this scen
invalid_rows = final_data[final_data['HelpfulnessNumerator'] > final_data['HelpfulnessDenominator']]
invalid_rows
```

we see that there are two records with HelpfulnessNumerator greater than HelpfulnessDenominator, which is practically not possible as HelpfulnessNumerator represents how many customers find the review to be only helpful whereas HelpfulnessDenominator represents how many customers find the review to be both helpful and not helpful (few may find helpful, and few may not. SO, in short, summation of both is HelpfulnessDenominator).

4. Checking missing values and Balancing data.

```
missing_data_percentage = (final_data.isna().mean() * 100).round(2)
missing_data_percentage
```

```
final_data['Score'].value_counts()
```

```
positive    307061
negative     57110
Name: Score, dtype: int64
```

```
positive_reviews=final_data[final_data['Score'] == 'positive'].sample(n=20000, random_state=42)
negative_reviews=final_data[final_data['Score'] == 'negative'].sample(n=20000, random_state=42)
final_balanced_data=pd.concat([positive_reviews, negative_reviews])
print(final_balanced_data.iloc[0])
print('************************************************************')
print(final_balanced_data.iloc[20001])
```

Checking the percentage of data after the cleaning has been done. Then see that our data consists of more positive reviews and a smaller number of negative reviews. So, our dataset is imbalanced. Models may not perform well for imbalanced datasets. So, we should take balanced amount of positive and negative reviews. At the end we have balanced the data with equal criteria.

5. Text preprocessing ad Removing html tags.

```
#printing html tags in one of our review text

count=0
for sent in final_balanced_data['Text'].values:
    if(len(re.findall('<.*?>',sent))):
        print(sent)
        break
    count+=1
print(count)
```

```
Great Idea.  Saves money, and you have your choice of<br />coffee.  I have told several others about this easy and<
br />creative new product.  Thank you Amazon.
3
```

```
#defining functions for removing html tags and punctuations
def removehtml(review_text):
    clean_html = re.compile('<.*?>')
    cleantext = re.sub(clean_html, ' ', review_text)
    return cleantext
```

We have checked whether the data contain any html tags, so we have added the code to remove html tags and clean the data.

6. Removing Punctuations words.

```
def removepunc(review_text):
    cleaned_punc = re.sub(r'[?|!|\'|"|#]',r'',review_text)
    cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned_punc)
    return  cleaned
```

Reviews also contain various punctuations which needs to be removed, the above function is to remove the punctuations from the data.

7. Removing stopwords.

```
warnings.filterwarnings('ignore')
nltk.download('stopwords')
stopwords_list = stopwords.words('english')
stopwords = set(stopwords_list)

print(stopwords)
```

Reviews contain all the words which are not necessarily needed for the predictions. We need to clean the unnecessary words from our data to increase the accuracy in the predictions. The above function identifies all the stop words from the given language (English).

8. Stemming the text data.

```
sn = nltk.stem.SnowballStemmer('english')
print(sn.stem('tasty'))
```

```
tasti
```

Stemming categorize the words that are present in the data and relate required words to predict. For instance, we have given the word ‘tasty’ and result given is ‘tasti’. we perform stemming for each word to eliminate data like (tasty, tastful, taste to tasti, which gives same meaning and with only one word) to achieve lemmatization.

```
a=0
string1=' '
final_string=[]
all_positive_words=[]
all_negative_words=[]
s=''
for sentence in final_balanced_data['Text'].values:
    filtered_sentence=[]
    sent=removehtml(sentence) # remove HTMl tags
    for word in sent.split():
        for cleaned_words in removepunc(word).split(): #remove puncuations
            if((cleaned_words.isalpha()) & (len(cleaned_words)>2)):
                if(cleaned_words.lower() not in stopwords): #Removing stopwords
                    sen=(sn.stem(cleaned_words.lower())).encode('utf8') #perform stemming and encoding
                    filtered_sentence.append(sen)
                    if (final_balanced_data['Score'].values)[a] == 'positive':
                        all_positive_words.append(sen)
                    if(final_balanced_data['Score'].values)[a] == 'negative':
                        all_negative_words.append(sen)
                else:
                    continue
            else:
                continue
    string1 = b" ".join(filtered_sentence)

    final_string.append(string1)
    a+=1
```

9. Adding “CleanedText” column to the dataset.

```
#adding a column of CleanedText which displays the data after pre-processing of the review
final_balanced_data['CleanedText']=final_string
final_balanced_data['CleanedText']=final_balanced_data['CleanedText'].str.decode("utf-8")
print(final_balanced_data.shape)
final_balanced_data.head()
```

```
(40000, 11)
```

We have added new column named "CleanedText" with these cleaned punctuations, HTML tags, stemming, lemmatization, removing special characters and alphanumeric data. We have also converted the text to utf-8 encoding for further text to vector supporting.

10. Sorting data based on Timestamp.

```
# converting timestamp to datetime and sorting the data
final_balanced_data['Time']=pd.to_datetime(final_balanced_data['Time'],unit='s')
final_balanced_data=final_balanced_data.sort_values(by='Time')
final_balanced_data.head()
```

In the real-world unseen data, we get the reviews that are latest. So, it’s better to train our model with old timestamp data and test the model with new timestamp data to perform well on real world unseen data (which is of latest time). As the time varies customer interest may vary or product quality may vary. So, take this into consideration we perform sorting based on timestamp.

For Featurization we have added

1. Word Frequency Analysis

```
nltk.download('punkt')
final_balanced_data['Tokenized_Text'] = final_balanced_data['CleanedText'].apply(lambda x: word_tokenize(x.lower()))

# Flatten the list of tokens
all_words = [word for tokens in final_balanced_data['Tokenized_Text'] for word in tokens]

# Calculate word frequencies
freq_dist = FreqDist(all_words)

# Get the most common words
top_words = freq_dist.most_common(20)

# Convert to a DataFrame for visualization
top_words_df = pd.DataFrame(top_words, columns=['Word', 'Frequency'])

# Plot the word frequencies
plt.figure(figsize=(10, 6))
plt.bar(top_words_df['Word'], top_words_df['Frequency'])
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 20 Most Common Words')
plt.xticks(rotation=45)
plt.show()
```

Analyze the frequency distribution of words in the dataset to understand which words are most used. This Analysis will help us to check the weightage of words when the text is converted to vectors before modelling. Also, we have added new column with "Tokenized_Text" that has list of words in our sentences. This will help us to analyze word frequencies and to convert word to vector embedding.

2. Vocabulary Size

```
all_words = [word for tokens in final_balanced_data['Tokenized_Text'] for word in tokens]
vocabulary_size = len(set(all_words))
print(all_words[3])
print(f"The vocabulary size is: {vocabulary_size}")
```

```
make
The vocabulary size is: 25278
```

Vocabulary Size provides a numerical representation of how many unique words are present in the dataset. As per the above observation, a very large vocabulary might lead to overfitting if not properly handled. Techniques like dimensionality reduction or sentence to vectors (like TF-IDF or word embeddings) may be applied before modeling as a feature engineering.

3. Word Clouds

```
all_reviews = ' '.join(final_balanced_data['CleanedText'])
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_reviews)
plt.figure(figsize=(10, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Larger and bolder words in the word cloud are indicative of frequently occurring words in the text. These are likely to be significant keywords or terms in the dataset. In

our dataset, Use, one, tast, good, flavor, love, product, food, amazon, order, great are few frequently occurring words in our text and are considered to be more important in giving weightage while modelling.

4. Sentiment by Document Length

```
final_balanced_data['Review Length'] = final_balanced_data['CleanedText'].apply(lambda x: len(str(x).split()))
def partition(x):
    if x =='negative':
        return 0
    else:
        return 1
dummy_data=filtered_data['Score']
review_column_data=dummy_data.map(partition)
final_balanced_data['Numeric_Score']=review_column_data
final_balanced_data.head()
```

We have converted Numeric scores to 0 and 1 for further analysis and also, we have added new column with "Review Length" that has count of words in our review text and "Numeric_Score" with 0 and 1 for negative and positive reviews respectively, which can be used for further analysis.

```
# Filter positive and negative reviews
positive_reviews = final_balanced_data[final_balanced_data['Numeric_Score'] == 1]
negative_reviews = final_balanced_data[final_balanced_data['Numeric_Score'] == 0]

plt.figure(figsize=(10, 6))
plt.scatter(positive_reviews['Review Length'], positive_reviews['Numeric_Score'], color='green', label='Positive Rev
plt.scatter(negative_reviews['Review Length'], negative_reviews['Numeric_Score'], color='red', label='Negative Revie
plt.xlabel('Length of Review Text')
plt.ylabel('Sentiment (1 for Positive, 0 for Negative)')
plt.title('Scatter Plot of Review Text Lengths for Positive vs Negative Reviews')
plt.legend()
plt.show()
```

From the above code we can check the review text length, for positive reviews lie between 0 to 400 words, whereas for negative reviews, that length lie between 0 to 500 except for few points that crossed 500 lengths. This can help us to see the average count of words that customers given for both positive and negative reviews. As they are overlapping (similar for both positive and negative), we cannot judge sentiments based on length of review text alone.

5. Analysis of Helpful Votes

```
filtered_data = final_balanced_data[final_balanced_data['HelpfulnessDenominator'] > 0]
filtered_data['HelpfulnessPercentage'] = filtered_data['HelpfulnessNumerator'] / filtered_data['HelpfulnessDenominat
helpful_percentage_by_sentiment = filtered_data.groupby('Numeric_Score')['HelpfulnessPercentage'].mean()
plt.figure(figsize=(10, 6))
plt.bar(helpful_percentage_by_sentiment.index, helpful_percentage_by_sentiment.values, color=['red', 'green'])
plt.xlabel('Sentiment (0 for Negative, 1 for Positive)')
plt.ylabel('Average Helpful Percentage')
plt.title('Relationship between Sentiment and Helpful Votes')
plt.xticks([0, 1], ['Negative', 'Positive'])
plt.show()
```

From the above code, we can see that positive reviews are found more helpful for customers compared to negative reviews. So, this shows that positive reviews can be

similar in real world data and model can perform better for positive reviews compared to negative reviews as most of the customers agree with positive words that are present in our dataset.

6. Length of Titles vs Length of Review text

```
final_balanced_data['Title Length'] = final_balanced_data['Summary'].apply(lambda x: len(str(x).split()))
plt.figure(figsize=(10, 6))
plt.scatter(final_balanced_data['Title Length'], final_balanced_data['Review Length'], alpha=0.5)
plt.xlabel('Title Length')
plt.ylabel('Review Text Length')
plt.title('Length of Title vs Lenth of Review Text')
plt.show()
```

From the above code, we can check the review title lengths. So, we can tell that customers are not willing to explain their thought clearly in title itself. So, considering the review text, it would be better and provide more information than title for modelling.

7. Feature Engineering BAD of WORDS and Bi-Grams

```
#splitting the data into train and test
x_train,x_test,y_train,y_test=train_test_split(final_balanced_data['CleanedText'],final_balanced_data['Score'],test_
print(x_train.shape,y_train.shape,y_test.shape,x_test.shape)
```

```
(28000,) (28000,) (12000,) (12000,)
```

Code to split the data to train and test, so that test data(text) is not seen by training data while applying bag of words, tf-idf, word2vec, avg w2v techniques to convert text to vectors.

```
count_vect = CountVectorizer()
count_vect.fit(x_train)
print("some feature names ", count_vect.get_feature_names_out()[:20])
print('-'*50)

final_counts = count_vect.transform(x_train)
print("Type of count vectorizer ",type(final_counts))
print("Shape of BOW vectorizer ",final_counts.get_shape())
print("Number of unique words ", final_counts.get_shape()[1])
print(final_counts.toarray())
```

The "bags-of-words" form, which ignores structure and instead counts the frequency of each word and is the simplest and most natural way to accomplish. The bags-of-words is applied using CountVectorizer, which transforms a group of text documents into a matrix of token counts. Our collection of text documents is transformed into a token count matrix by instantiating the CountVectorizer and fitting it to our training data.

BI – GRAMS

```
count_vect = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
final_bigram = count_vect.fit_transform(x_train)
print("some feature names ", count_vect.get_feature_names_out()[:20])
print('-'*50)
print("Type of count vectorizer ",type(final_bigram))
print("Shape of BOW vectorizer ",final_bigram.get_shape())
print("Number of unique words with both unigrams and bigrams ", final_bigram.get_shape()[1])
print(final_bigram.toarray())
```

Bag-of-bigrams representation is much more powerful than bag-of-words. bigram refers to a combination of two adjacent words in a text.

8. Term Frequency – Inverse Document Frequency (TF - IDF)

```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2), min_df=10)
tf_idf_vect.fit(x_train)
print("some sample features(unique words in the corpus)",tf_idf_vect.get_feature_names_out()[0:20])
print('-'*50)
final_tf_idf = tf_idf_vect.transform(x_train)
print("Type of count vectorizer ",type(final_tf_idf))
print("Shape of TFIDF vectorizer ",final_tf_idf.get_shape())
print("Number of unique words including both unigrams and bigrams ", final_tf_idf.get_shape()[1])
```

TF-IDF allows us to weight terms based on how important they are to a document. These extremely common phrases would cover the frequencies of less common but more interesting terms if we sent the count data straight to a classifier.

9. Word2Vec and Average Word2Vec

```
# we convert our review sentence text to list of sentence as our word2vec model takes list of sentences
list_sent=[]
for sent in final_balanced_data['CleanedText']:
    cleaned_str=[]
    sent=removehtml(sent)
    for word in sent.split():
        for w in removepunc(word).split():
            if((w.isalpha())&(len(w)>2)):
                cleaned_str.append(w.lower())
            else:
                continue
    list_sent.append(cleaned_str)
final_balanced_data['list_sent']=list_sent
```

```
#splitting the data into train and test
x_train,x_test,y_train,y_test=train_test_split(final_balanced_data['list_sent'],final_balanced_data['Score'],test_si
print(x_train.shape,y_train.shape,y_test.shape,x_test.shape)
```

```
(28000,) (28000,) (12000,) (12000,)
```

```
w2v_model=Word2Vec(x_train,workers=4,min_count=50,vector_size=50)
print('-'*50)
# Get the dictionary mapping words to integer indices
w2v_words_dict = w2v_model.wv.key_to_index
# Get the list of word strings
w2v_words_list = w2v_model.wv.index_to_key
print("number of words that occured minimum 5 times ", len(w2v_words_list))
print("sample words ", w2v_words_list[0:50])
```

```
word_to_check = 'awesom'

if word_to_check in w2v_model.wv.key_to_index:
    similar_words = w2v_model.wv.most_similar(word_to_check)
    print(f"Words similar to '{word_to_check}':")
    for word, similarity in similar_words:
        print(f"{word}: {similarity}")
else:
    print(f"'{word_to_check}' is not in the vocabulary.")
```

```
Words similar to 'awesom':
amaz: 0.8615049719810486
fantast: 0.8211140036582947
delici: 0.7812532782554626
fabul: 0.779184103012085
yummi: 0.7003751397132874
wonder: 0.6980993747711182
great: 0.6980451941490173
winner: 0.6830552220344543
good: 0.656852662563324
homemad: 0.6429780721664429
```

Word2Vec captures semantic relationships between words. Words with similar meanings tend to be closer in the embedding space. This can be very helpful and with less dimensions with meaningful words given equal weightage. So, model can give better results if it sees similar kind of words in unseen data.

Average Word 2 Vec

```
sent_vectors=[]
for sent in x_train:
    sent_vector=np.zeros(50)
    count_words=0
    for word in sent:
        if word in w2v_words_list:
            vector=w2v_model.wv[word]
            sent_vector+=vector
            count_words+=1
    if count_words!=0:
        sent_vector/=count_words
    sent_vectors.append(sent_vector)
```

word2vec produces individual word vectors for each word in a text. whereas AvgWord2vec aggregates the vectors of all words in a text to create a single vector representation for the entire text. AvgWord2vec results in a single vector of the same dimensionality as the individual word vectors. But, it treats all words in a text equally, potentially losing some contextual information. we need to decide which (w2v or AvgW2v) is best based on the computational complexity and metric scores of model that is being used.

10. TFIDF Weighted w2v

```
#tf-idf-weightedw2v
tf_idf_sent_vector=[]
dictionary = dict(zip(tf_idf_vect.get_feature_names_out(), list(tf_idf_vect.idf_)))
i=0
for sent in x_train:
    sent_vector=np.zeros(50)
    weight_sum=0
    for word in sent:
        if word in w2v_words_list:
            vec=w2v_model.wv[word]
            tf_idf = dictionary[word]*(sent.count(word)/len(sent))
            sent_vector += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vector/=weight_sum
    tf_idf_sent_vector.append(sent_vector)
    i+=1
len(tf_idf_sent_vector)
```

TFIDF-Word2Vec Combines the principles of TF-IDF weighting and averaging Word2Vec vectors to create text representations. It incorporates TF-IDF scores to give more weight to important words while creating the average vector. Provides a weighted average representation that focuses more on important words in the context of the entire corpus. TF-IDF-Word2Vec is particularly useful in scenarios where the importance of words varies within the corpus. Like word2vec and Avg-w2v, we need to decide which (w2v or AvgW2v or TF-IDF w2v) is best based on the computational complexity and metric scores of models that is being used.

ML Algorithms used and Model Execution with code working (Phase-2)

Machine Learning Algorithms Used:

1. KNN
2. Naive Bayes
3. Logistic Regression
4. Support Vector Machines
5. Decision Trees
6. Random Forests

In the phase 2 report, we have clearly mentioned the reason for choosing those algorithms, Source of the algorithm.

For every model we have implemented four techniques, bag of words and TF-IDF. The results for each technique had been mentioned in the phase 2 report along with the required results and graphs. From the phase 2 results for all the algorithms and techniques are mentioned below:

KNN

```
from sklearn.metrics import roc_curve, auc
def test_data(x_train,y_train,x_test,y_test,algorithm):
    neigh = KNeighborsClassifier(n_neighbors=best_k,algorithm=algorithm, n_jobs=-1)
    neigh.fit(x_train, y_train)

    # roc_auc_score(y_true, y_score) the 2nd parameter should be probability estimates of the positive class
    # not the predicted outputs

    train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_proba(x_train)[:,1])
    test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(x_test)[:,1])

    sns.set()
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.legend()
    plt.xlabel("False_positive_rate")
    plt.ylabel("True positive_rate")
    plt.title("ROC_Curve")
    plt.grid()
    plt.show()
    print('The AUC_score of test_data is :',auc(test_fpr, test_tpr))
```

The KNN algorithm is implemented as shown above, we have used the KNN classifier from sklearns and done the implementation. For the above implementation, we have the different parameters, best_k, algorithm, and n-Jobs. We have chosen Brute force and KD Tree for our analysis.

To get the best K we have done the grid search function as shown below, from the Grid search we get the Cross validation and best k which will used in the KNN implementation to predict the results.

```
def Grid_search(x_train,y_train,algorithm):
    cv=KFold(n_splits=5)
    myList = list(range(0,50))
    K=list(filter(lambda x: x % 2 != 0, myList))
    neigh=KNeighborsClassifier(algorithm=algorithm)
    parameters = {'n_neighbors':list(filter(lambda x: x % 2 != 0, myList))}
    clf = GridSearchCV(neigh, parameters, cv=cv, scoring='roc_auc',return_train_score=True,verbose=1)
    clf.fit(x_train, y_train)

    results = pd.DataFrame.from_dict(clf.cv_results_)
    results = results.sort_values(['param_n_neighbors'])

    train_auc= clf.cv_results_['mean_train_score']
    train_auc_std= clf.cv_results_['std_train_score']
    cv_auc = clf.cv_results_['mean_test_score']
    cv_auc_std= clf.cv_results_['std_test_score']
    best_k = clf.best_params_['n_neighbors']

    sns.set()
    plt.plot(K, train_auc, label='Train AUC')
    plt.gca().fill_between(K,train_auc - train_auc_std,train_auc + train_auc_std,alpha=0.2,color='darkblue')

    plt.plot(K, cv_auc, label='CV AUC')
    plt.gca().fill_between(K,cv_auc - cv_auc_std,cv_auc + cv_auc_std,alpha=0.2,color='darkorange')
    plt.scatter(K, train_auc, label='Train AUC points')
    plt.scatter(K, cv_auc, label='CV AUC points')
    plt.legend()
    plt.xlabel("K: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()

    print("Best cross-validation score: {:.3f}".format(clf.best_score_))
    print('The best k from gridsearch :',best_k)
    return  best_k
```

```
!pip install prettytable
from prettytable import PrettyTable

table = PrettyTable()
table.field_names = ["Vectorizer", "Model", "Hyper_Parameter(K)", "AUC_Score"]
table.add_row(["Bow", 'K_NN_Brute_Force', 49,78.7 ])
table.add_row(["TFIDF", 'K_NN_Brute_Force', 49, 84.1])
table.add_row(["Avg_Word2vec", 'K_NN_Brute_Force', 47, 49.9,])
table.add_row(["TFIDF_Word2vec", 'K_NN_Brute_Force',1 ,49.14 ])
print(table)
```

```
Requirement already satisfied: prettytable in ./anaconda3/lib/python3.10/site-packages (3.9.0)
Requirement already satisfied: wcwidth in ./anaconda3/lib/python3.10/site-packages (from prettytable) (0.2.5)
+---------------+-----------------+-------------------+-----------+
|   Vectorizer  |      Model      | Hyper_Parameter(K) | AUC_Score |
+---------------+-----------------+-------------------+-----------+
|      Bow      | K_NN_Brute_Force |         49        |    78.7   |
|     TFIDF     | K_NN_Brute_Force |         49        |    84.1   |
```

Naive Bayes

```
def test_data(x_train,y_train,x_test,y_test):
    model=MultinomialNB(alpha=best_alpha)
    model.fit(x_train, y_train)
    train_fpr, train_tpr, thresholds = roc_curve(y_train, model.predict_proba(x_train)[:,1])
    test_fpr, test_tpr, thresholds = roc_curve(y_test, model.predict_proba(x_test)[:,1])
    sns.set()
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.plot([0, 1], [0, 1], color='green', lw=1, linestyle='--')
    plt.legend()
    plt.xlabel("False_positive_rate")
    plt.ylabel("True positive_rate")
    plt.title("ROC_Curve")
    plt.grid()
    plt.show()
    print('The AUC_score of test_data is :',auc(test_fpr, test_tpr))
```

```
from sklearn.naive_bayes import MultinomialNB
import math
def Grid_search(X_train,Y_train):
    cv=KFold(n_splits=5)
    alpha_values = [0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000,10000,100000]#alpha from 10^-5 to 10^5
    model=MultinomialNB()
    parameters = {'alpha':alpha_values}
    clf = GridSearchCV(model,parameters, cv=cv, scoring='roc_auc',return_train_score=True,verbose=1)
    clf.fit(X_train, Y_train)
    results = pd.DataFrame.from_dict(clf.cv_results_)
    results = results.sort_values(['param_alpha'])
    train_auc= clf.cv_results_['mean_train_score']
    train_auc_std= clf.cv_results_['std_train_score']
    cv_auc = clf.cv_results_['mean_test_score']
    cv_auc_std= clf.cv_results_['std_test_score']
    best_alpha= clf.best_params_['alpha']
    sns.set()
    alpha_values=[math.log(x) for x in alpha_values]
    plt.plot(alpha_values, train_auc, label='Train AUC')
    plt.gca().fill_between(alpha_values,train_auc - train_auc_std,train_auc + train_auc_std,alpha=0.2,color='darkblue')
    plt.plot(alpha_values, cv_auc, label='CV AUC')
    plt.gca().fill_between(alpha_values,cv_auc - cv_auc_std,cv_auc + cv_auc_std,alpha=0.2,color='darkorange')
    plt.scatter(alpha_values, train_auc, label='Train AUC points')
    plt.scatter(alpha_values, cv_auc, label='CV AUC points')
    plt.legend()
    plt.xlabel("alpha_values: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()

    print("Best cross-validation score: {:.3f}".format(clf.best_score_))
    print('The best alpha from gridsearch :',best_alpha)
    return  best_alpha
```

Evaluating a Multinomial Naive Bayes classifier using ROC curves and AUC scores. The function takes training and testing data along with labels as input, fits the classifier, computes ROC curves for both training and testing sets, and plots them. The green dashed line represents a random classifier. The AUC scores for the test data are printed, providing a quantitative measure of classifier performance. The function utilizes scikit-learn, seaborn, and matplotlib libraries to visualize and assess the classifier's ability to distinguish between classes based on false positive and true positive rates. The function

Grid_search that performs hyperparameter tuning for a Multinomial Naive Bayes (NB) model using Grid Search and k-fold cross-validation. It searches for the best alpha value (smoothing parameter) by evaluating the model's performance based on the area under the Receiver Operating Characteristic (ROC) curve. The function plots error curves for both training and cross-validation sets, visualizing the impact of different alpha values. The optimal alpha and corresponding cross-validation score are printed. This approach aids in selecting the best hyperparameter for enhancing the Multinomial NB model's performance on the given dataset.

```
def metric(x_train,y_train,x_test,y_test):
    model=MultinomialNB(alpha=best_alpha)
    model.fit(x_train, y_train)
    predict=model.predict(x_test)

    conf_mat = confusion_matrix(y_test, predict)
    class_label = ["Negative", "Positive"]
    df = pd.DataFrame(conf_mat, index = class_label, columns = class_label)

    report=classification_report(y_test,predict)
    print(report)

    sns.set()
    sns.heatmap(df, annot = True,fmt="d")
    plt.title("Test_Confusion_Matrix")
    plt.xlabel("Predicted_Label")
    plt.ylabel("Actual_Label")
    plt.show()
```

The function initializes the NB model with an optimized alpha parameter, fits it to the training data, and predicts labels for the test data. It then generates and displays a confusion matrix, a classification report, and a heatmap visualization of the confusion matrix. The function serves as a convenient tool for assessing the performance of the NB classifier, particularly in a binary classification scenario, providing key metrics and visualizations for model evaluation.

```
from prettytable import PrettyTable

table = PrettyTable()
table.field_names = ["Vectorizer", "Feature engineering", " Hyper Parameter (alpha)", "AUC_Score"]
table.add_row(["Bow", 'Featurized', 10,88.9 ])
table.add_row(["TFIDF", 'Featurized', 1, 92.92])
print(table)
```

```
+------------+--------------------+--------------------------+-----------+
| Vectorizer | Feature engineering |  Hyper Parameter (alpha) | AUC_Score |
+------------+--------------------+--------------------------+-----------+
|    Bow     |      Featurized     |            10            |    88.9   |
|   TFIDF    |      Featurized     |            1             |   92.92   |
+------------+--------------------+--------------------------+-----------+
```

Logistic Regression

```
def test_data(model,x_train,y_train,x_test,y_test):

    model.fit(x_train, y_train)
    train_fpr, train_tpr, thresholds = roc_curve(y_train, model.predict_proba(x_train)[:,1])
    test_fpr, test_tpr, thresholds = roc_curve(y_test, model.predict_proba(x_test)[:,1])
    sns.set()
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.plot([0, 1], [0, 1], color='green', lw=1, linestyle='--')
    plt.legend()
    plt.xlabel("False_positive_rate")
    plt.ylabel("True positive_rate")
    plt.title("ROC_Curve")
    plt.grid()
    plt.show()
    print('The AUC_score of test_data is :',auc(test_fpr, test_tpr))
```

```
def Grid_search(model,X_train,Y_train):

    parameters = {'C':C}
    cv=KFold(n_splits=5)
    clf = GridSearchCV(model,parameters, cv=cv, scoring='roc_auc',return_train_score=True)
    clf.fit(X_train, Y_train)

    results = pd.DataFrame.from_dict(clf.cv_results_)
    results = results.sort_values(['param_C'])

    train_auc= clf.cv_results_['mean_train_score']
    train_auc_std= clf.cv_results_['std_train_score']
    cv_auc = clf.cv_results_['mean_test_score']
    cv_auc_std= clf.cv_results_['std_test_score']
    best_C= clf.best_params_['C'] #c=1/lamda

    sns.set()
    C_values=[math.log(x) for x in C]
    plt.plot(C, train_auc, label='Train AUC')
    plt.gca().fill_between(C,train_auc - train_auc_std,train_auc + train_auc_std,alpha=0.2,color='darkblue')

    plt.plot(C, cv_auc, label='CV AUC')
    plt.gca().fill_between(C,cv_auc - cv_auc_std,cv_auc + cv_auc_std,alpha=0.2,color='darkorange')
    plt.scatter(C, train_auc, label='Train AUC points')
    plt.scatter(C, cv_auc, label='CV AUC points')
    plt.legend()
    plt.xlabel("C = 1/λ: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()

    print("Best cross-validation score: {:.3f}".format(clf.best_score_))
    print('The best C from gridsearch :',best_C)
    return  best_C
```

```python
def metric(model,x_train,y_train,x_test,y_test):

    model.fit(x_train, y_train)
    predict=model.predict(x_test)

    conf_mat = confusion_matrix(y_test, predict)
    class_label = ["Negative", "Positive"]
    df = pd.DataFrame(conf_mat, index = class_label, columns = class_label)

    report=classification_report(y_test,predict)
    print(report)

    sns.set()
    sns.heatmap(df, annot = True,fmt="d")
    plt.title("Test_Confusion_Matrix")
    plt.xlabel("Predicted_Label")
    plt.ylabel("Actual_Label")
    plt.show()
```

The provided code consists of three functions for evaluating a binary classification model, particularly for Support Vector Machines (SVMs) in the context of text data. The test_data function assesses the model's performance by generating a Receiver Operating Characteristic (ROC) curve and calculating the Area Under the Curve (AUC) on both training and testing datasets. This graphical representation helps visualize the model's ability to distinguish between positive and negative instances. The Grid_search function performs hyperparameter tuning using GridSearchCV, specifically for the regularization parameter 'C' in SVM. It visualizes the impact of different 'C' values on training and cross-validation AUC scores, assisting in identifying the optimal hyperparameter value. Finally, the metric function evaluates the model on the testing dataset, providing a confusion matrix and a classification report, offering insights into the model's precision, recall, and F1-score for positive and negative classes. The visualizations, such as ROC curves and error plots, aid in understanding the model's performance and guide the selection of hyperparameters for optimal results.

```python
table = PrettyTable()
table.field_names = ["Vectorizer","Regularization", "Hyperameter(C=1/lamda)", "AUC"]
table.add_row(["BOW","L1",0.1,89.26])
table.add_row(["TFIDF","L1",0.1,91.09])
table.add_row(["AvgW2v","L1",10,88.6])
table.add_row(["TFIDF_AvgW2v","L1",10,49.3])
print(table)
```

```
+--------------+----------------+------------------------+-------+
|  Vectorizer  | Regularization | Hyperameter(C=1/lamda) |  AUC  |
+--------------+----------------+------------------------+-------+
|     BOW      |       L1       |          0.1           | 89.26 |
|    TFIDF     |       L1       |          0.1           | 91.09 |
```

SVM

```
def test_data(x_train,y_train,x_test,y_test):
    model = SGDClassifier(loss='hinge', penalty=best_penalty, alpha=best_alpha, n_jobs=-1)
    clf = CalibratedClassifierCV(base_estimator=model, cv=None)

    clf.fit(x_train, y_train)

    # roc_auc_score(y_true, y_score) the 2nd parameter should be probability estimates of the positive class
    # not the predicted outputs

    train_fpr, train_tpr, thresholds = roc_curve(y_train, clf.predict_proba(x_train)[:,1])
    test_fpr, test_tpr, thresholds = roc_curve(y_test, clf.predict_proba(x_test)[:,1])

    sns.set()
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.plot([0, 1], [0, 1], color='green', lw=1, linestyle='--')
    plt.legend()
    plt.xlabel("False_positive_rate")
    plt.ylabel("True positive_rate")
    plt.title("ROC_Curve")
    plt.grid()
    plt.show()
    print('The AUC_score of test_data is :',auc(test_fpr, test_tpr))
```

```
def Hyper_parameter(X_train,X_cv,Y_train,Y_cv):
    import warnings
    max_roc_auc=-1
    cv_scores = []
    train_scores = []
    penalties = ['l1', 'l2']
    C = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]
    for i in C:
        for p in penalties:
            model= SGDClassifier(alpha=i, penalty=p, loss='hinge', random_state=42)
            model.fit(X_train,Y_train)
            clf = CalibratedClassifierCV(model, method="sigmoid")
            clf.fit(X_train, Y_train)
            y_score=clf.predict_proba(X_cv)[:,1]
            scores = roc_auc_score(Y_cv, y_score)
            cv_scores.append(scores)
            y_score=clf.predict_proba(X_train)[:,1]
            scores = roc_auc_score(Y_train, y_score)
            train_scores.append(scores)
            s=['0.00001+L1', '0.00001+L2', '0.0001+L1', '0.0001+L2', '0.001+L1', '0.001+L2', '0.01+L1', '0.01+L2',
             '0.1+L1', '0.1+L2', '1+L1', '1+L2', '10+L1', '10+L2', '100+L1', '100+L2','1000+L1','1000+L2','10000+L1','10000+L2']
    optimal_alpha= s[cv_scores.index(max(cv_scores))]
    alpha=[math.log(x) for x in C]#converting values of alpha into logarithm
    fig = plt.figure(figsize=(20,5))
    ax = plt.subplot()
    ax.plot(s, train_scores, label='AUC train')
    ax.plot(s, cv_scores, label='AUC CV')
    plt.title('AUC vs hyperparameter')
    plt.xlabel('alpha')
    plt.ylabel('AUC')
    plt.xticks()
    ax.legend()
    plt.show()
    print('best Cross validation score: {:.3f}'.format(max(cv_scores)))
    print('optimal alpha and penalty for which auc is maximum : ',optimal_alpha)
```

```
def metric(x_train,y_train,x_test,y_test):
    model = SGDClassifier(loss='hinge', penalty=best_penalty, alpha=best_alpha, n_jobs=-1)
    clf = CalibratedClassifierCV(base_estimator=model, cv=None)
    clf.fit(x_train, y_train)
    predict=clf.predict(x_test)
    conf_mat = confusion_matrix(y_test, predict)
    class_label = ["Negative", "Positive"]
    df = pd.DataFrame(conf_mat, index = class_label, columns = class_label)
    report=classification_report(y_test,predict)
    print(report)
    sns.set()
    sns.heatmap(df, annot = True,fmt="d")
    plt.title("Test_Confusion_Matrix")
    plt.xlabel("Predicted_Label")
    plt.ylabel("Actual_Label")
    plt.show()
```

The provided code defines functions for hyperparameter tuning, testing model performance, and visualizing the receiver operating characteristic (ROC) curve for a Support Vector Machine (SVM) classifier using stochastic gradient descent (SGD). The code utilizes scikit-learn and seaborn for machine learning and visualization.

The Hyper_parameter function performs a grid search over a range of hyperparameters (alpha and penalty) for the SVM model and plots the corresponding area under the ROC curve (AUC) scores on both the training and cross-validation sets. This aids in selecting the optimal hyperparameters that maximize AUC.

The test_data function evaluates the SVM model on test data and generates a ROC curve with AUC scores, providing a visual representation of the model's discriminatory ability.

The metric function assesses the model's performance on the test set, producing a confusion matrix and classification report, aiding in the interpretation of the model's predictive capabilities. These functions collectively offer a comprehensive framework for hyperparameter tuning, testing, and evaluating the SVM model's performance on a text classification task. Users can leverage this code for efficient model development and analysis.

```
table = PrettyTable()
table.field_names = ["Vectorizer","Regularization", "Hyperameter(C)", "AUC_Score"]
table.add_row(["BOW","L2",0.1,92.9])
table.add_row(["TFIDF","L2",1,92.03])
table.add_row(["AvgW2v","L2",0.5,51.1])
table.add_row(["TFIDF_AvgW2v","L1",0.5,50.0])
print(table)
```

```
+--------------+----------------+----------------+-----------+
|  Vectorizer  | Regularization | Hyperameter(C) | AUC_Score |
+--------------+----------------+----------------+-----------+
|     BOW      |       L2       |      0.1       |    92.9   |
|    TFIDF     |       L2       |       1        |   92.03   |
```

Decision Trees

```
def test_data(x_train,y_train,x_test,y_test):

    model=DecisionTreeClassifier(max_depth=depth, min_samples_split =split, class_weight='balanced')

    model.fit(x_train, y_train)

    # roc_auc_score(y_true, y_score) the 2nd parameter should be probability estimates of the positive class
    # not the predicted outputs

    train_fpr, train_tpr, thresholds = roc_curve(y_train, model.predict_proba(x_train)[:,1])
    test_fpr, test_tpr, thresholds = roc_curve(y_test, model.predict_proba(x_test)[:,1])

    sns.set()
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.plot([0, 1], [0, 1], color='green', lw=1, linestyle='--')
    plt.legend()
    plt.xlabel("False_positive_rate")
    plt.ylabel("True positive_rate")
    plt.title("ROC_Curve")
    plt.grid()
    plt.show()
    print('The AUC_score of test_data is :',auc(test_fpr, test_tpr))
```

```
from sklearn.tree import DecisionTreeClassifier
def Grid_search(X_train,Y_train):
    Depths=[4,6, 8, 9,10,50,100,500]
    min_split= [10,20,30,40,50,100,500]
    param_grid = {'max_depth': Depths, 'min_samples_split': min_split}

    clf = GridSearchCV(DecisionTreeClassifier(), param_grid, scoring = 'roc_auc', cv=3 , n_jobs = -1, pre_dispatch=2,return_train_score=True)
    clf.fit(X_train, Y_train)


    print("\n**********AUC Score for CV data **********\n")
    print("\nOptimal depth:", clf.best_estimator_.max_depth)
    print("\nOptimal split:", clf.best_estimator_.min_samples_split)
    print("\nBest Score:", clf.best_score_)

    sns.set()
    df_gridsearch = pd.DataFrame(clf.cv_results_)
    max_scores = df_gridsearch.groupby(['param_max_depth','param_min_samples_split']).max()
    max_scores = max_scores.unstack()[['mean_test_score', 'mean_train_score']]
    sns.heatmap(max_scores.mean_test_score, annot=True, fmt='.4g')
    plt.show()
```

```
def metric(x_train,y_train,x_test,y_test):

    model=DecisionTreeClassifier(max_depth=depth, min_samples_split =split, class_weight='balanced')

    model.fit(x_train, y_train)
    predict=model.predict(x_test)

    conf_mat = confusion_matrix(y_test, predict)
    class_label = ["Negative", "Positive"]
    df = pd.DataFrame(conf_mat, index = class_label, columns = class_label)

    report=classification_report(y_test,predict)
    print(report)

    sns.set()
    sns.heatmap(df, annot = True,fmt="d")
    plt.title("Test_Confusion_Matrix")
    plt.xlabel("Predicted_Label")
    plt.ylabel("Actual_Label")
    plt.show()
```

The provided code defines functions for testing a Decision Tree classifier's performance, conducting a grid search for hyperparameter tuning, and visualizing the results. The test_data function evaluates the classifier's performance on the test set, generating a Receiver Operating Characteristic (ROC) curve and calculating the Area Under the Curve (AUC) score. It also displays the AUC score for the test data. The Grid_search function performs a grid search to find optimal hyperparameters (max_depth and min_samples_split) for the Decision Tree classifier using cross-validated data. It prints the optimal hyperparameters, the corresponding AUC score, and a heatmap visualizing the grid search results.

The metric function assesses the model's performance on the test set by generating a confusion matrix and a classification report, providing insights into precision, recall, and F1-score for each class. The confusion matrix is visualized as a heatmap for better interpretation. The code uses scikit-learn and seaborn for machine learning and visualization tasks, offering a comprehensive analysis of the Decision Tree classifier's performance and parameter tuning results.

```
table = PrettyTable()
table.field_names = ["Vectorizer","Optimal Min_split", "Optimal Depth", "AUC_Score"]
table.add_row(["BOW",50,500,75.75])
table.add_row(["TFIDF",10,500,72.49])
table.add_row(["AvgW2v",500,500,47.81])
table.add_row(["TFIDF_AvgW2v",40,500,50.88])
print(table)
```

```
+-------------+------------------+--------------+-----------+
|  Vectorizer | Optimal Min_split | Optimal Depth | AUC_Score |
+-------------+------------------+--------------+-----------+
|     BOW     |         50       |      500     |   75.75   |
|    TFIDF    |         10       |      500     |   72.49   |
```

Random Forests

```
def test_data(model,x_train,y_train,x_test,y_test):


    model.fit(x_train, y_train)

    train_fpr, train_tpr, thresholds = roc_curve(y_train, model.predict_proba(x_train)[:,1])
    test_fpr, test_tpr, thresholds = roc_curve(y_test, model.predict_proba(x_test)[:,1])

    sns.set()
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.plot([0, 1], [0, 1], color='green', lw=1, linestyle='--')
    plt.legend()
    plt.xlabel("False_positive_rate")
    plt.ylabel("True positive_rate")
    plt.title("ROC_Curve")
    plt.grid()
    plt.show()
    print('The AUC_score of test_data is :',auc(test_fpr, test_tpr))
```

```
from sklearn.ensemble import RandomForestClassifier
def Grid_search(model,X_train,Y_train):
    estimators = [50,100,200,300,400,500]
    Depths = [10,20,30,40,50,60]

    param_grid = {'max_depth': Depths, 'n_estimators': estimators}

    clf = GridSearchCV(model, param_grid, scoring = 'roc_auc', cv=3 , n_jobs = -1, pre_dispatch=2,return_train_score=True)
    clf.fit(X_train, Y_train)


    print("\n**********AUC Score for CV data **********\n")
    print("\nOptimal depth:", clf.best_estimator_.max_depth)
    print("\nOptimal estimators:", clf.best_estimator_.n_estimators)
    print("\nBest Score:", clf.best_score_)

    sns.set()
    df_gridsearch = pd.DataFrame(clf.cv_results_)
    max_scores = df_gridsearch.groupby(['param_max_depth','param_n_estimators']).max()
    max_scores = max_scores.unstack()[['mean_test_score', 'mean_train_score']]
    sns.heatmap(max_scores.mean_test_score, annot=True, fmt='.4g')
    plt.show()
```

```python
def metric(model,x_train,y_train,x_test,y_test):

    model.fit(x_train, y_train)
    predict=model.predict(x_test)

    conf_mat = confusion_matrix(y_test, predict)
    class_label = ["Negative", "Positive"]
    df = pd.DataFrame(conf_mat, index = class_label, columns = class_label)

    report=classification_report(y_test,predict)
    print(report)

    sns.set()
    sns.heatmap(df, annot = True,fmt="d")
    plt.title("Test_Confusion_Matrix")
    plt.xlabel("Predicted_Label")
    plt.ylabel("Actual_Label")
    plt.show()
```

The provided code includes functions for testing, grid search, and metric evaluation for a Random Forest classifier. The test_data function assesses the model's performance on both the training and testing datasets by plotting the Receiver Operating Characteristic (ROC) curve and calculating the Area Under the Curve (AUC) score. This provides insights into the classifier's ability to discriminate between positive and negative classes.

The Grid_search function performs hyperparameter tuning using Grid Search Cross-Validation. It explores different combinations of maximum depth and the number of estimators in the Random Forest, aiming to optimize the model's performance. The results, including optimal hyperparameters and AUC scores, are displayed.

The metric function evaluates the model's performance on the testing dataset by generating a confusion matrix and a classification report. The confusion matrix visualizes the count of true positive, true negative, false positive, and false negative predictions, while the classification report provides metrics such as precision, recall, and F1-score for each class.

These functions collectively offer a comprehensive analysis of the Random Forest classifier, guiding users in understanding its effectiveness and facilitating model optimization through hyperparameter tuning.

```
table = PrettyTable()
table.field_names = ["Model","Vectorizer","Optimal Depth", "Optimal n_estimator", "AUC_Score"]
table.add_row(["Random Forest","BOW",20,400,88.66])
table.add_row(["Random Forest","TFIDF",30,400,90.40])
table.add_row(["Random Forest","AvgW2v",60,100,51.35])
table.add_row(["Random Forest","TFIDF_AvgW2v",60,50,49.4])
print(table)
```

```
+---------------+--------------+---------------+---------------------+-----------+
|     Model     |  Vectorizer  | Optimal Depth | Optimal n_estimator | AUC_Score |
+---------------+--------------+---------------+---------------------+-----------+
| Random Forest |     BOW      |       20      |         400         |   88.66   |
| Random Forest |    TFIDF     |       30      |         400         |    90.4   |
```

Out of all the machine learning algorithms and techniques that we implemented; the best performed model for the taken text-based dataset is **Support Vector Machine.**

From the two techniques: Bag of Words and TF – IDF, we got the best results with SVM, and accuracies are 92.9 and 92.03 respectively. The high accuracy indicates that our model is effective in classifying the text data based on the chosen feature representations.

In summary, the implemented SVM model demonstrates strong performance in text classification tasks, achieving high accuracy with both Bag of Words and TF-IDF representations, showcasing the effectiveness of SVM in handling text data.

Working Instructions (Phase-3)

1. **Cloning the Repository.**
   You may either download or extract the project files from the given source code.

2. **Navigating to project Directory.**
   Locate the project files by opening a terminal or command prompt and going to the directory.

3. **Preparing the Data**
   Create a sample text dataset or work with the similar data. Make that the format and preprocessing of the data meet the needs of the model.

4. **Run the Script**
   Execute the script.

5. **Executing the Model**
   - The text data will be loaded by the script.
   - It will do tokenization and other required preprocessing on the data.
   - It will build the TF-IDF representations and the Bag of Words.
   - The training set will be used to train the SVM model.
   - The testing set will be used to assess the model.
   - The model's accuracy using TF-IDF and Bag of Words will be shown.

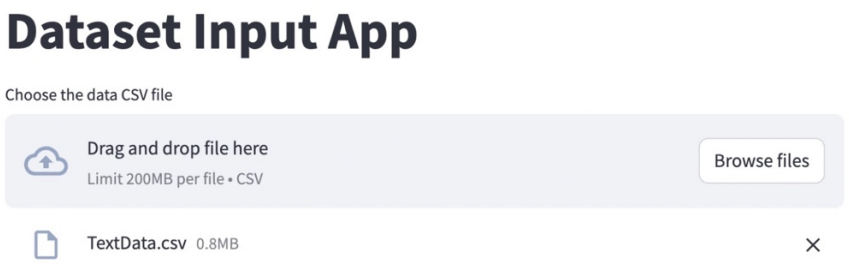
6. **Interpreting the results**
   - To find out the results with the SVM model worked dataset and examine the accuracy metrics.
   - The efficacy of the model is demonstrated for taken dataset, the accuracy results, which range from 92.03% with TF-IDF to 92.9% with Bag of Words.

7. **Experimenting with the data**
   To test the SVM model, modify the sample script to use your own text data. As required, change settings or hyperparameters to suit your unique needs.

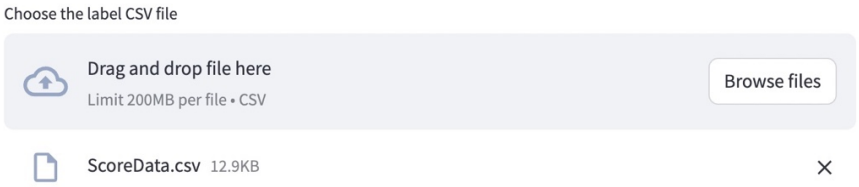
8. **Output Analysis**
   Depending on what is used in the script, investigate other outputs or metrics produced by the script, such as confusion matrices, accuracy, recall, etc.

You may conduct a demonstration of the implemented SVM model using TF-IDF and Bag of Words representations by following these instructions. This will enable you to observe the model's performance on your text data and may be used as a foundation for additional customization or application integration.

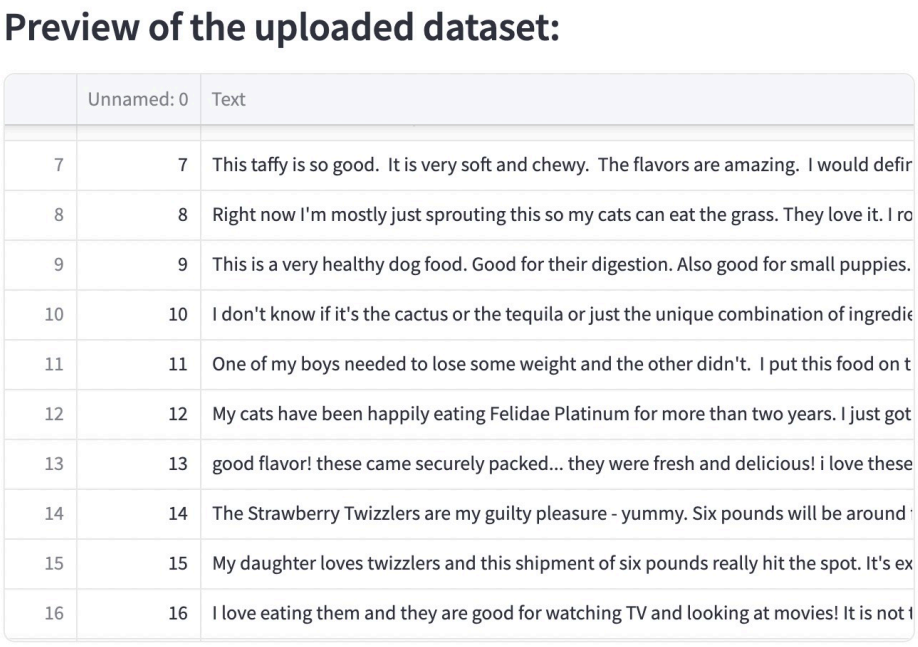
UI Interface Demo (Phase 3)



Users need to select the test data file in the first step of process like shown above.

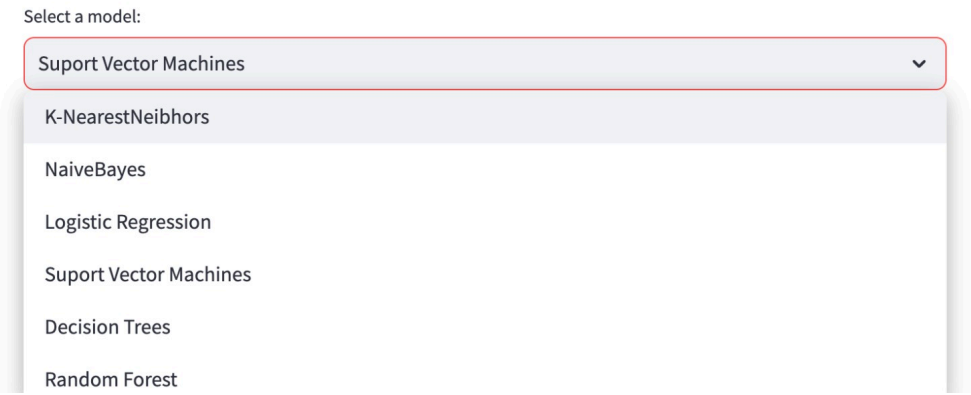


Next, users need to select the scorceData file that contain the new data to make the predictions

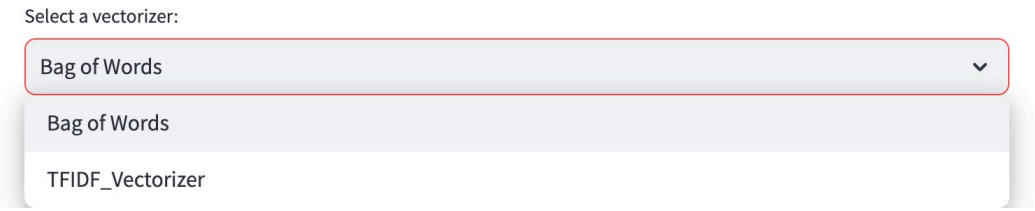


Then, the preview of the data is shown below to check the uploaded file was correct or not.

Next the preview of additional dataset is shown.



Then, there is an option to select the implemented models from the dropdown option provided, users can select the required model as per the data to view the predictions.



Next, there is an option to select the one of the two vectorizers, which we implemented. Users can select either Bag of Words or TFIDF.

Preprocessed text:

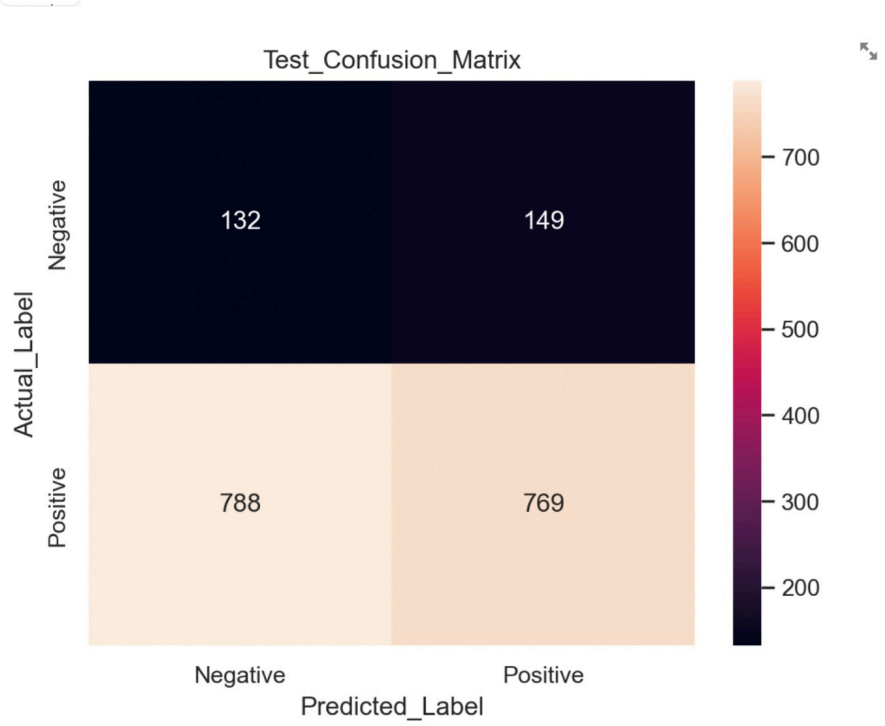
	CleanedText
0	bought sever vital can dog food product found good qualiti product look like stew pr
1	product arriv label jumbo salt peanut peanut actual small size unsalt sure error vend
2	confect around centuri light pillowi citrus gelatin nut case filbert cut tini squar liber c
3	look secret ingredi robitussin believ found got addit root beer extract order good mac
4	great taffi great price wide assort yummi taffi deliveri quick taffi lover deal

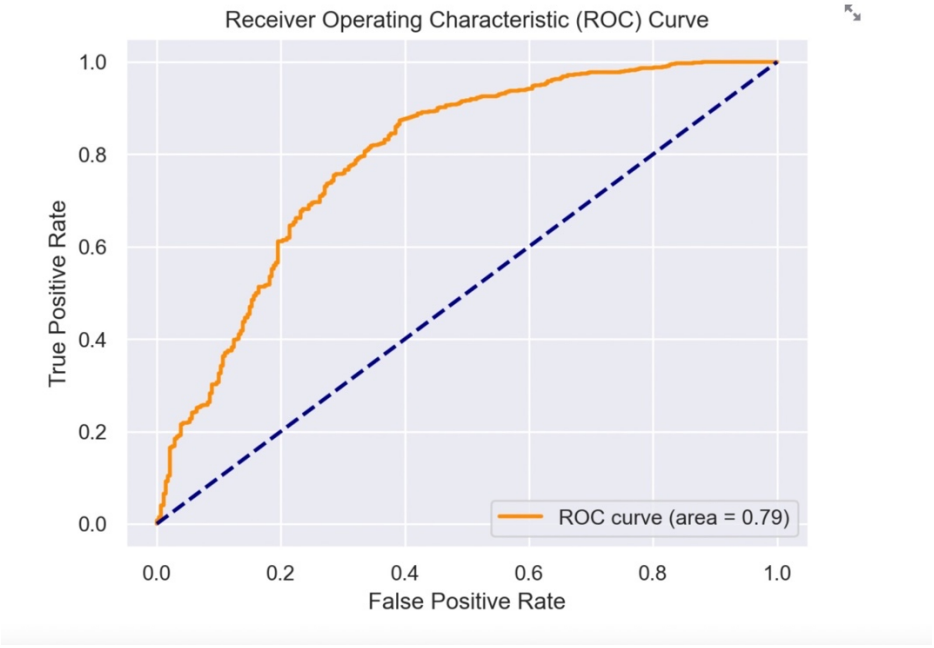
Then, the preprocessed text is shown for the verification.

Then, by clicking the predict button provided, the users can be able to submit the selections and see the results displayed as below,

	precision	recall	f1-score	support
negative	0.14	0.47	0.22	281
positive	0.84	0.49	0.62	1557
accuracy			0.49	1838
macro avg	0.49	0.48	0.42	1838
weighted avg	0.73	0.49	0.56	1838

value
positive
positive
negative
positive
negative
negative
positive
positive
negative
positive





From the above results, the User can be able to check the accuracy and also view the confusion matrix and ROC curves to check the output.

Notes on Selected Model that performed well (Phase-3)

Out of all the chosen Algorithms and techniques, **Support Vector Machine** performed well for our dataset. The results had been included in the phase 2 report and the final accuracy shown below for the reference.

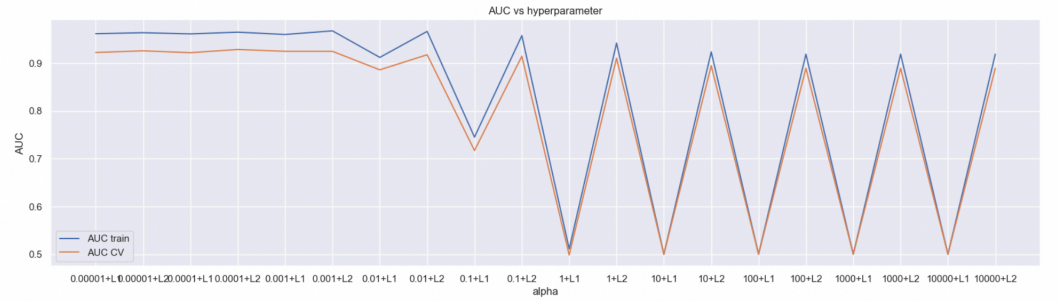
Notes on SVM:

Support Vector Machine on Bag of words, TF-IDF. The reasons for choosing SVM are, SVM can process high dimensional data, Effective with the sparse data, strongly oppose overfitting the model, uses binary classification with kernel tricks, SVM has better interpretability, optimization, and well supported libraries.

SVM with Bag of Words:

Cross Validation

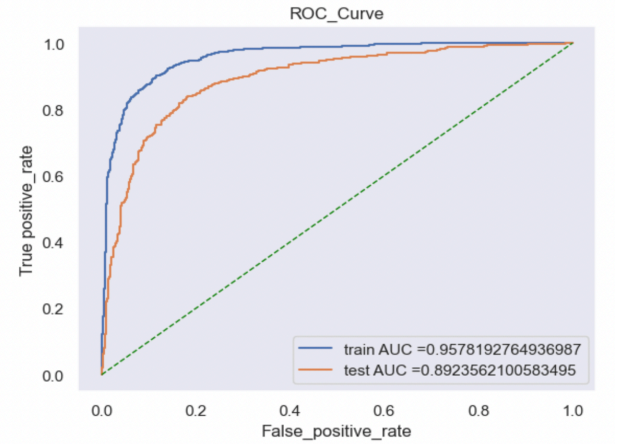
```
import warnings as w
w.filterwarnings("ignore")
Hyper_parameter(x_train_bow,x_cv_bow,y_train,y_cv)
```




```
best Cross validation score: 0.929
optimal alpha and penalty for which auc is maximum :  0.0001+L2
```

Metric Evaluation

```
test_data(x_train_bow,y_train_binary,x_test_bow,y_test_binary)
```

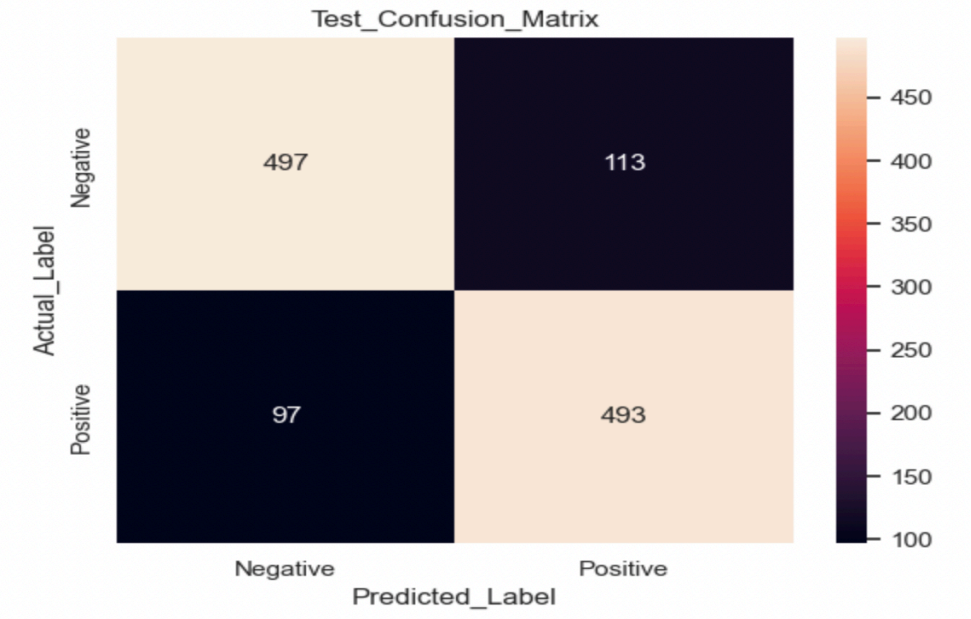



```
The AUC_score of test_data is : 0.8923562100583495
```

Confusion Matrix

```
metric(x_train_bow,y_train,x_test_bow,y_test)
```

	precision	recall	f1-score	support
negative	0.84	0.81	0.83	610
positive	0.81	0.84	0.82	590
accuracy			0.82	1200
macro avg	0.83	0.83	0.82	1200
weighted avg	0.83	0.82	0.83	1200



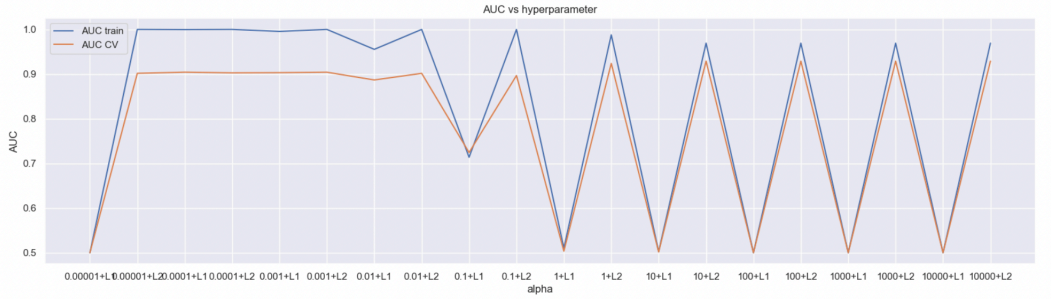
From the analysis and results above, we can say that the predictions made by the model are approximately correct. By looking at the confusion matrix above, we can check the actual positive and negative values along with the predictions made by our model.

From the graph, the actual negative that predicted negative values are 497 and actual negative that predicted positive are 97. On the other hand, actual positive that predicted positive values are 493 and actual positive that predicted negative are 113. At the end, giving the accuracy of 92.9.

SVM with TF-IDF:

Hyperparameter

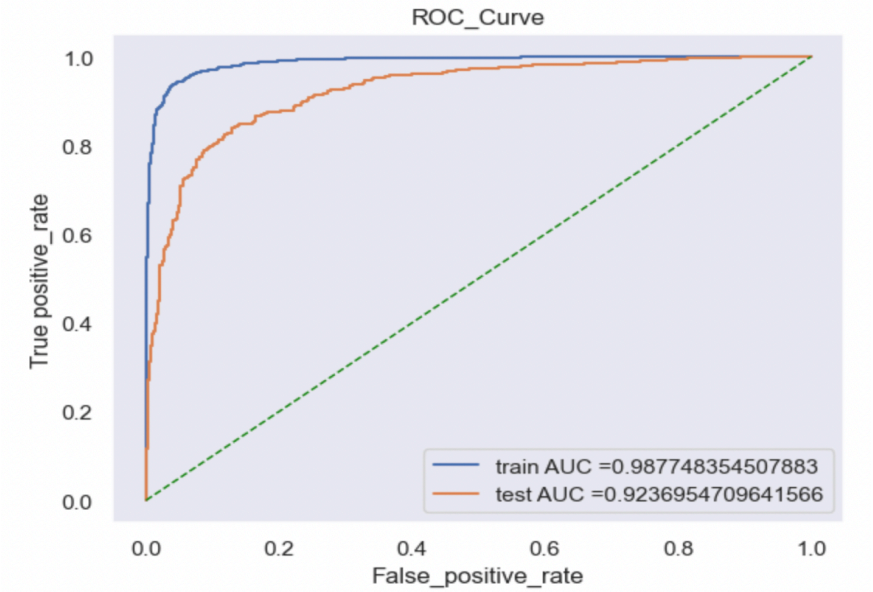
```
Hyper_parameter(x_train_tf_idf,x_test_tf_idf,y_train,y_test)
```




```
best Cross validation score: 0.929
optimal alpha and penalty for which auc is maximum :  10+L2
```

Metric Evaluation

```
test_data(x_train_tf_idf,y_train_binary,x_test_tf_idf,y_test_binary)
```




```
The AUC_score of test_data is : 0.9236954709641566
```

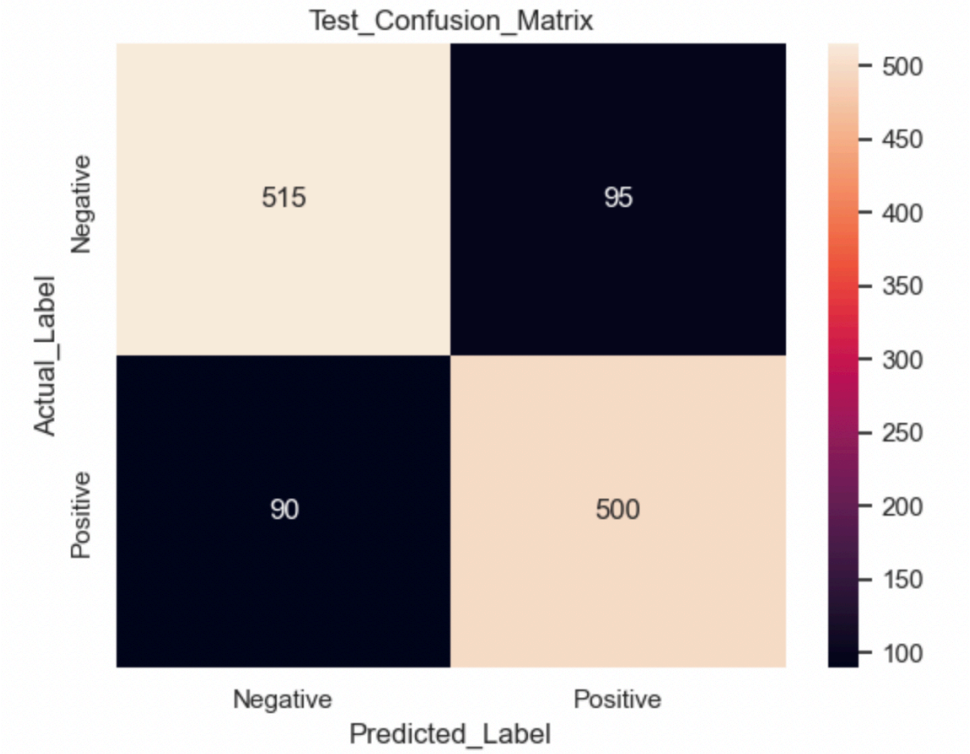
Confusion Matrix

```
metric(x_train_tf_idf,y_train,x_test_tf_idf,y_test)
```

```
              precision    recall  f1-score   support

    negative       0.85      0.84      0.85       610
    positive       0.84      0.85      0.84       590

    accuracy                           0.85      1200
   macro avg       0.85      0.85      0.85      1200
weighted avg       0.85      0.85      0.85      1200
```



From the analysis and results above, we can say that the predictions made by the model are approximately correct. By looking at the confusion matrix above, we can check the actual positive and negative values along with the predictions made by our model.

From the graph, the actual negative that predicted negative values are 515 and actual negative that predicted positive are 90. On the other hand, actual positive that predicted positive values are 500 and actual positive that predicted negative are 95. At the end, giving the accuracy of 92.03.

Recommendations related to the problem Statement (Phase-3)

Interpretation of Confusion Matrix:

Users can learn how to interpret a confusion matrix to understand the true positives, true negatives, false positives, and false negatives generated by the model. In the context of sentiment analysis, understanding these metrics provides insights into the model's performance in correctly identifying positive and negative reviews.

Model Accuracy and Misclassifications:

The accuracy of 92.03% suggests that the model is performing well in classifying reviews as positive or negative. However, users should be aware of misclassifications, such as false positives and false negatives, which are essential for understanding the model's limitations.

Optimizing for Specific Metrics:

Depending on the application, users may want to prioritize certain metrics over others. For example, in sentiment analysis, false negatives (positive reviews predicted as negative) might be more critical than false positives. Users can adjust the model or choose evaluation metrics that align with their specific goals.

Fine-Tuning the Model:

Users can explore hyperparameter tuning and feature engineering techniques to further enhance the model's performance. Adjusting parameters like the number of trees in the Random Forest or experimenting with different text representations (e.g., TF-IDF) could lead to improved accuracy.

Problem-Solving and Use Cases:

Sentiment Analysis Applications:

The model, with its accuracy of 92.03%, can be used for sentiment analysis in various applications, helping businesses and organizations automatically categorize customer reviews as positive or negative. This can aid in understanding customer sentiment and making data-driven decisions.

User Feedback and Model Improvement:

Encourage users to provide feedback on misclassified reviews. This user feedback loop can be incorporated to continuously improve the model, making it more robust to diverse language patterns and specific domain nuances.

Project Extensions:

Multiclass Sentiment Analysis:

Extend the project to handle multiclass sentiment analysis, where reviews can be classified into more than just two categories (e.g., positive, neutral, negative). This provides a more nuanced understanding of sentiment.

Exploration of Neural Models:

Investigate the use of more complex models like recurrent neural networks (RNNs) or transformer-based models (e.g., BERT) for sentiment analysis. These models may capture intricate relationships within text data and potentially lead to performance improvements.

User-Friendly Interface:

Develop a user-friendly interface that allows users to input their own reviews and receive sentiment predictions. This could be useful for businesses wanting to analyze customer feedback in real-time.

Future Avenues:

Temporal Analysis:

Explore how sentiments evolve over time. Implement a temporal analysis to identify trends and changes in customer opinions, helping businesses stay informed about shifts in sentiment.

Domain-Specific Sentiment Analysis:

Adapt the model for domain-specific sentiment analysis. Fine-tune the model on data specific to a particular industry or domain to improve its accuracy in that context.

Integration with Customer Support:

Explore the integration of the sentiment analysis model with customer support systems. Automatically categorizing customer messages can help prioritize and address issues more efficiently.

By considering these recommendations and exploring potential extensions, users can derive more value from the sentiment analysis model, improving its accuracy and applicability to specific use cases.