In [*]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from tqdm import tqdm
from sklearn.model_selection import train_test_split
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer,PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
```

In [*]:
```
df=pd.read_csv('C:\\Users\\rupin\\OneDrive\\Desktop\\Data mining\\Assignment-
```

In [*]:
```
df
```

In [*]:
```
#removing null values
df= df.dropna()
print(df.isnull().sum())
```

In [*]:
```
train_df, dev_test_df = train_test_split(df, test_size=0.4, random_state=42)
dev_df, test_df = train_test_split(dev_test_df, test_size=0.5, random_state=4
```

In [*]:
```
# Create an empty dictionary to hold the word counts
word_counts = {}

# Iterate over each review in the DataFrame
for review in df['Review']:

    # Split the review into words
    words = review.split()

    # Update the word counts
    for word in words:
        if word in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1
```

In [*]:
```
# Create a list of words that occur more than 5 times
vocabulary = [word for word, count in word_counts.items() if count > 5 and le
```

In [*]:
```
# Convert each word to lowercase using a list comprehension
vocabulary = [word.lower() for word in vocabulary]
```

In [*]:
```
import re

cleaned_strings = []

def remove_unterminated_chars(vocabulary):

    for string in vocabulary:
        try:
            cleaned_strings.append(re.sub(r'[()\[\]{}"\']', '', string))
        except re.error:
            # Handle exceptions raised by the re module
            print(f"Error cleaning string '{string}'")
    return cleaned_strings

remove_unterminated_chars(vocabulary)
```

In [*]:
```
#now we will use lemmatization so that words which are similar can be removed

nltk.download('stopwords')
nltk.download('wordnet')


# initialize the stop words set and lemmatizer object
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()


# remove stop words and perform lemmatization on the vocabulary
updated_vocabulary = []
for word in cleaned_strings:
    if word not in stop_words:
        updated_vocabulary.append(lemmatizer.lemmatize(word))

# print the processed vocabulary
print(updated_vocabulary)
```

In [*]:
```
print(list(word_counts.items())[:5])
print(updated_vocabulary[:5])
```

In [*]:
```
# Create a reverse index dictionary
reverse_index = {word: index for index, word in enumerate(updated_vocabulary)}
```

In [*]:
```
print(reverse_index)
```

In [*]:
```
# sort the dictionary by value in ascending order
sorted_reverse_index = dict(sorted(reverse_index.items(), key=lambda x: x[1]))

# print the sorted dictionary
print(sorted_reverse_index)
```

In [*]:
```
# Print the first 10 words in the vocabulary list and their indices
for word in updated_vocabulary[:10]:
    print(f"{word}: {reverse_index[word]}")
```

In [*]:
```
# c. Calculation of probability

#calculating prior probabilities of each category i.e. fresh or rotten
category_counts = train_df["Freshness"].value_counts()
category_counts_sum = np.sum(category_counts)

### Calculating the Prior Probability for Categories
pr_category_prior = pd.DataFrame(category_counts).T
pr_category_prior = pr_category_prior/category_counts_sum
pr_category_prior
```

In [*]:
```
print(len(vocabulary))
print(len(cleaned_strings))
print(len(updated_vocabulary))
```

In [*]:
```
#Calculating probability of each word in the vocabulary list
def word_probability(df, vocab):
    p_wrd_d = {}
    for word in tqdm(vocab):
        wrd_o = 0
        for sen in df['Review'].values:
            if (word in sen):
                wrd_o += 1
        p_wrd_d[word] = wrd_o / df.shape[0]
    return p_wrd_d

wrd_p = word_probability(df=train_df, vocab=updated_vocabulary)
```

In [*]:
```
print('Each word probability in the vocabulary ')
print(wrd_p)
```

In [*]:
```
top_10_words = sorted(wrd_p.items(), key=lambda x: x[1], reverse=True)[:10]
print("Top 10 words with highest probabilities:")
for word, probability in top_10_words:
    print(f"{word}: {probability}")
```

In [*]:
```
# Separate the positive and negative reviews
fresh_reviews = train_df[train_df['Freshness'] == 'fresh']
rotten_reviews = train_df[train_df['Freshness'] == 'rotten']

# Define the target word
target_word = 'the'

# Count the number of documents with the target sentiment containing the targe
fresh_containing_the = len(fresh_reviews[fresh_reviews['Review'].str.contains
rotten_containing_the = len(rotten_reviews[rotten_reviews['Review'].str.conta

# Count the total number of documents with the target sentiment
total_positive_documents = len(fresh_reviews)
total_negative_documents = len(rotten_reviews)

# Calculate the conditional probability of the target word given the target se
probability_given_positive = fresh_containing_the / total_positive_documents
probability_given_negative = rotten_containing_the / total_negative_documents

print(f"Probability of '{target_word}' given for fresh review : {probability_
print(f"Probability of '{target_word}' given for rotten review : {probability_
```

In [*]:
```
conditional_probabilities = {}
def get_word_probability(word):
    # Count the number of documents with the target sentiment containing the t
    fresh_containing_word = len(fresh_reviews[fresh_reviews['Review'].str.cont
    rotten_containing_word = len(rotten_reviews[rotten_reviews['Review'].str.

    # Count the total number of documents with the target sentiment
    total_positive_documents = len(fresh_reviews)
    total_negative_documents = len(rotten_reviews)

    # Calculate the conditional probability of the target word given the targe
    probability_given_positive = fresh_containing_word / total_positive_docume
    probability_given_negative = rotten_containing_word / total_negative_docu

     conditional_probabilities[word] = {
        'Positive': probability_given_positive,
        'Negative': probability_given_negative,
    }

    print(f"Probability of '{word}' given for fresh review : {probability_give
    print(f"Probability of '{word}' given for rotten review : {probability_gi
```

In [*]:
```
for word in updated_vocabulary:
    get_word_probability(word)
```

In [*]:
```
# Compute the conditional probabilities of each word given each class
p_word_given_fresh = {word: conditional_probabilities[word]['Positive'] for w
p_word_given_rotten = {word: conditional_probabilities[word]['Negative'] for

# Sort the words by their conditional probability for each class
top_fresh_words = sorted(p_word_given_fresh, key=p_word_given_fresh.get, reve
top_rotten_words = sorted(p_word_given_rotten, key=p_word_given_rotten.get, r

# Print the top 10 words for each class
print("Top 10 words predicting 'fresh':")
for word in top_fresh_words:
    print(f"{word}: {p_word_given_fresh[word]:.4f}")

print("\nTop 10 words predicting 'rotten':")
for word in top_rotten_words:
    print(f"{word}: {p_word_given_rotten[word]:.4f}")
```

In [*]:
```
# Define a function to classify a review based on the probabilities
def classify_review(review, wrd_p, conditional_probabilities):
    words = review.split()
    positive_prob = 1.0
    negative_prob = 1.0
    for word in words:
        if word in updated_vocabulary:
            positive_prob *= conditional_probabilities[word]['Positive']
            negative_prob *= conditional_probabilities[word]['Negative']
    if word in wrd_p:
        positive_prob *= wrd_p[word]
        negative_prob *= wrd_p[word]
        if positive_prob > negative_prob:
            return 'fresh'
        else:
            return 'rotten'

# Classify the development reviews and calculate the accuracy
dev_predictions = dev_df['Review'].apply(lambda x: classify_review(x, wrd_p,
accuracy = (dev_predictions == dev_df['Freshness']).sum() / len(dev_df)
print(f"Development accuracy: {accuracy:.4f}")
```

In [*]:
```
# Classify the test reviews and calculate the accuracy
test_predictions = test_df['Review'].apply(lambda x: classify_review(x, wrd_p
accuracy_test = (test_predictions == test_df['Freshness']).sum() / len(test_d
print(f"Test accuracy: {accuracy_test:.4f}")
```

In [*]:
```
# Define the data
models = ['Dev', 'Test']
accuracies = [accuracy_dev, accuracy_test]

# Set the plot size
plt.figure(figsize=(8, 5))

# Create a bar chart
plt.bar(models, accuracies, color=['blue', 'red'], width=0.4)

# Add labels and title
plt.xlabel("Model")
plt.ylabel("Accuracy")
plt.title("Comparison of Accuracies")

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