

****Final Project****

By: Alex Tran, Annabelle Le, Darren Liang, Eden Fraczekiewicz, and Malina Martinez

****Project description:****

For our project, we want to perform sentiment analysis on movie reviews and see if we can predict whether that user gave the movie a positive or negative review based on what they wrote. We will specifically be looking at critic reviews from Rotten Tomatoes across many different movies. Through this, we seek to discover what features are unique in making a movie review positive or negative.

****Dataset:****

GitHub page: <https://github.com/nicolas-gervais/rotten-tomatoes-dataset/blob/master/README.md>

Dataset: https://drive.google.com/file/d/1N8WCMci_jpDHwCVgSED-B9yts-q9_Bb5/view

****Learning Techniques:****

KNN Classification: KNN classification is a supervised learning technique that uses proximity between the k-number of data points and a given data point to classify it.

Multinomial Naive Bayes: Multinomial Naive Bayes is a probabilistic supervised learning method that is based on Bayes theorem. It calculates the probability of a given data point being in a class for each class and returns the highest probability.

Decision Tree: A Decision Tree is a supervised hierarchical model that uses a tree-like model of decisions to determine the class of a given data point.

Random Forest: A Random Forest is a supervised learning technique that uses the output of multiple decision trees to classify a given data point.

KMeans Clustering: KMeans Clustering is an unsupervised learning technique that uses vector quantization to cluster data together in order to identify different classes.

TextBlob: TextBlob is a Python library for processing textual data. It contains an unsupervised lexicon-based sentiment analyzer which uses pre-defined rules for how words are classified.

VADER: Valence Aware Dictionary and sEntiment Reasoner (VADER) is an unsupervised lexicon-based sentiment analysis tool that is specifically attuned for sentiment analysis of social media posts. Like TextBlob, it uses pre-defined rules for how words are classified.

****Data Cleaning and Vectorization:****

TF-IDF: TF-IDF is a vectorizing method that calculates the relative frequency of words across all documents instead of the raw count among each document. We chose this method over Bag of Words, another vectorizing method, because using the normalized frequency would help us identify unique words, rather than just the most common.

Stop Word Removal: Stop words are common words that are filtered out because they do not provide any value or meaning. In the context of movie reviews, stop words could include commonly used words like "the," "and," "is," "in," and so on. We chose to remove these stop words to lessen the amount of features and get to unique words that could distinguish positive and negative reviews.

Lemmatization: Lemmatization is the process of breaking down a word to its root and grouping words with common roots together. This further reduces the amount of features in our data as it combines words with the same meaning. In order to perform lemmatization, we used the WordNetLemmatizer from the nltk library.

Least Common/Most Common Removal: In our EDA below we found the most common words between rotten and fresh reviews. After doing so, we chose to remove the words that had no meaning in terms of positivity and negativity, and were the most common and least common between them. This reduces the number of features even further, making our models faster and more accurate.

****Contributions:****

Annabelle Le: Data cleaning/Vectorization, EDA, KNN Analysis, Model Analysis

Alex Tran: Data cleaning/Vectorization, EDA, Model Analysis

Eden Frackiewicz: Data cleaning/Vectorization, EDA, Multinomial Naive Bayes, Decision Tree, Random Forest, Model Analysis

Darren Liang: Data cleaning/Vectorization, EDA, KMeans Clustering, KNN Analysis, Model Analysis

Malina Martinez: Data cleaning/Vectorization, EDA, KMeans Clustering, KNN Analysis, TextBlob vs VADER, Model Analysis

```
In [1]: conda install -c conda-forge wordcloud
!pip install textblob
nltk.download('all')
```

```
In [2]: import string
import math
import nltk
import pandas as pd
import matplotlib.pyplot as plot
import seaborn as sns
from PIL import Image
from wordcloud import WordCloud
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from nltk.corpus import stopwords
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from collections import Counter
from string import punctuation
from collections import defaultdict
from time import time
from sklearn import metrics
from sklearn.cluster import KMeans
from textblob import TextBlob
%matplotlib inline

data = pd.read_csv("rotten_tomatoes_reviews.csv")
data
```

Out[2]:

	Freshness	Review
0	1	Manakamana doesn't answer any questions, yet ...
1	1	Wilfully offensive and powered by a chest-thu...
2	0	It would be difficult to imagine material mor...
3	0	Despite the gusto its star brings to the role...
4	0	If there was a good idea at the core of this ...
...
479995	0	Zemeckis seems unable to admit that the motio...
479996	1	Movies like The Kids Are All Right -- beautif...
479997	0	Film-savvy audiences soon will catch onto Win...
479998	1	An odd yet enjoyable film.
479999	1	No other animation studio, even our beloved P...

480000 rows × 2 columns

****Pre-EDA Data Cleaning:****

In [3]: `data.isnull().sum()`

Out[3]:

Freshness	0
Review	0
dtype:	int64

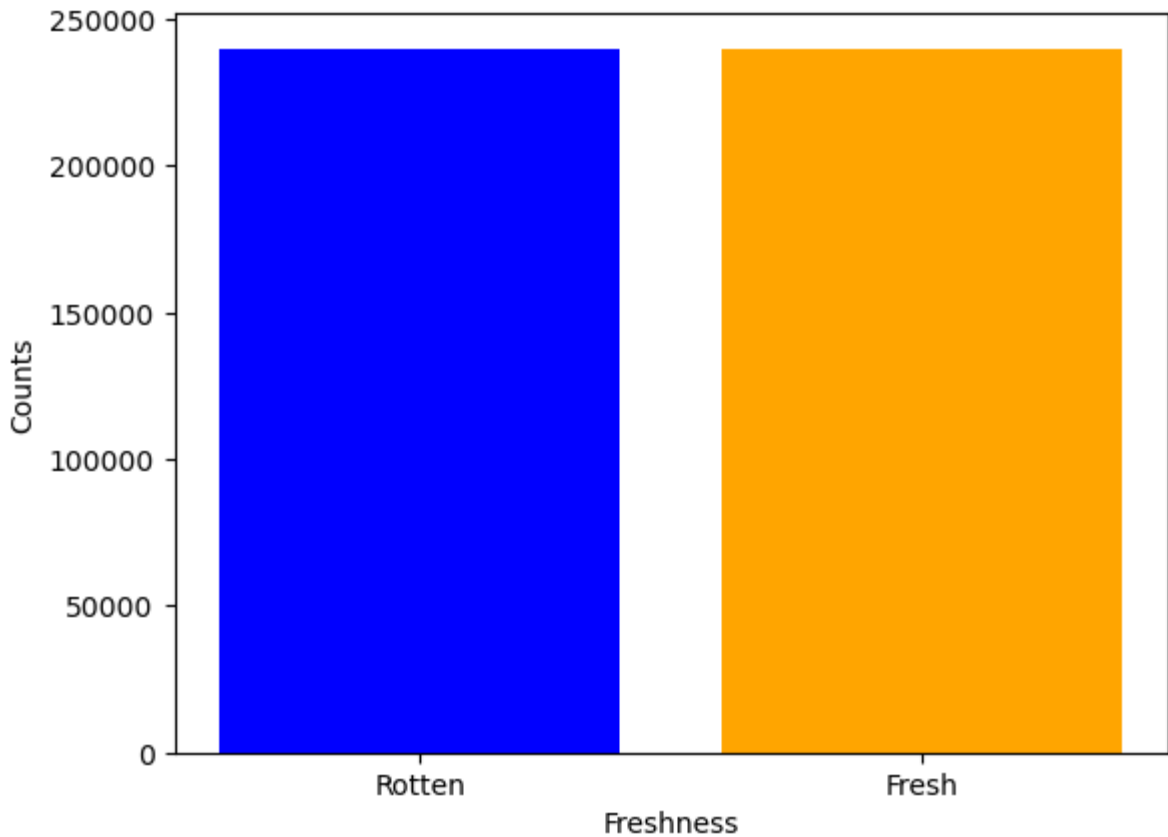
The creator of the dataset claimed that it was already clean, and this is confirmed by the fact that there are no NA values.

****Exploratory Data Anlysis:****

In [4]:

```
data.Freshness = data.Freshness.astype(str)
digitQuantity = data.Freshness.value_counts().sort_index()

plot.bar(digitQuantity.index, digitQuantity, color = ['blue', 'orange'])
plot.xlabel('Freshness')
plot.ylabel('Counts')
plot.xticks(ticks = [0, 1], labels = ['Rotten', 'Fresh'], rotation = 0)
plot.show()
```

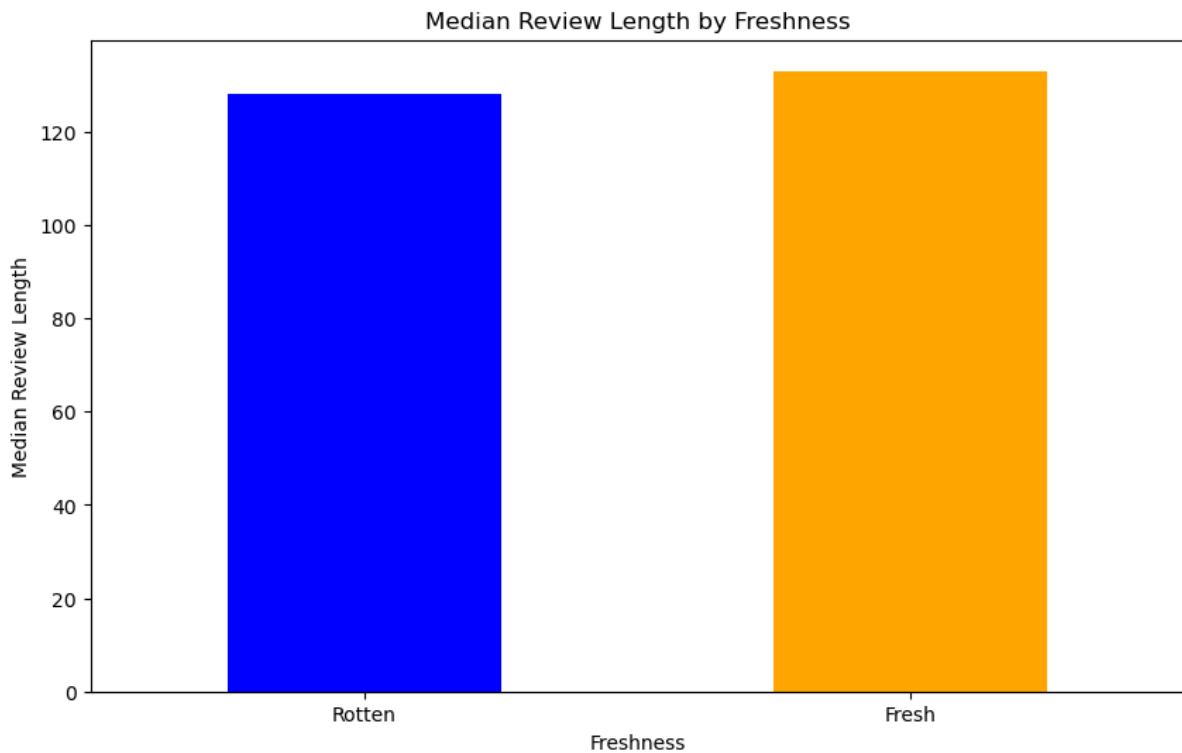


This bar chart shows the proportion of rotten reviews (0) and fresh reviews (1). The proportion is evenly split at 240,000.

```
In [5]: data['Review_Length'] = data['Review'].str.len()
median_lengths = data.groupby('Freshness')['Review_Length'].median()

plot.figure(figsize = (10, 6))
median_lengths.plot(kind = 'bar', color = ['blue', 'orange'])
plot.title('Median Review Length by Freshness')
plot.xlabel('Freshness')
plot.ylabel('Median Review Length')
plot.xticks(ticks = [0, 1], labels = ['Rotten', 'Fresh'], rotation = 0)
plot.show()

print(median_lengths)
```



```
Freshness
0    128.0
1    133.0
Name: Review_Length, dtype: float64
```

In this graph, we explored the median review length of rotten vs fresh reviews. We can see that fresh reviews had a slightly higher median review length than rotten reviews. We used the groupby function to group Freshness and review length.

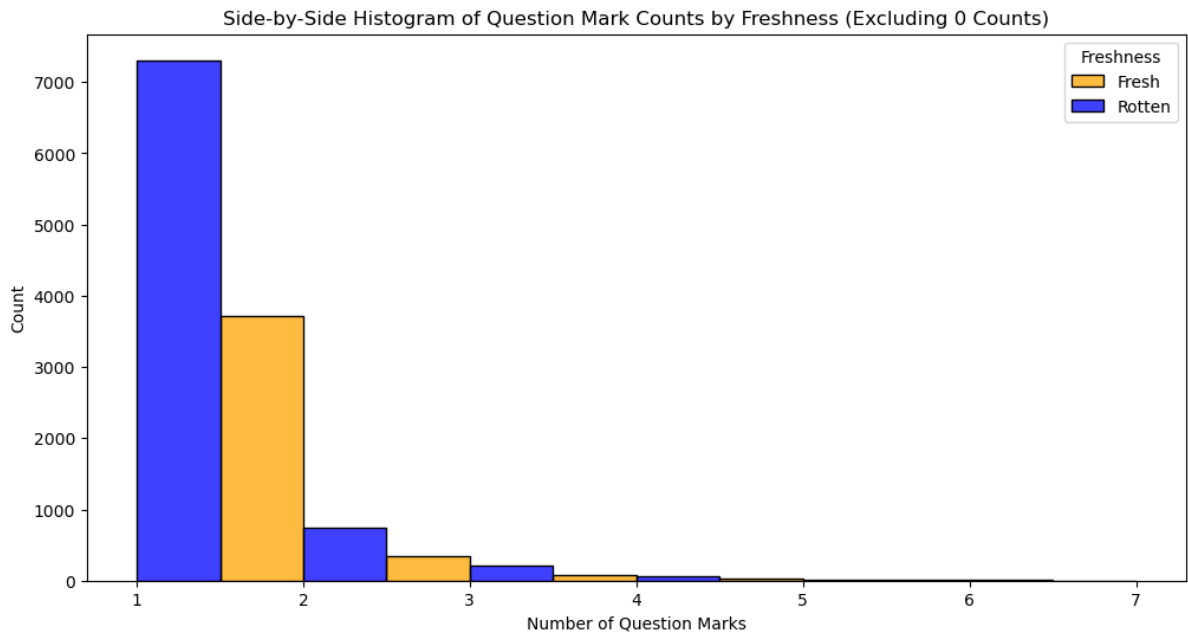
```
In [6]: data['Question_Marks'] = data['Review'].str.count('\?')

question_marks_crosstab = pd.crosstab(data['Freshness'], data['Question_Marks'])

question_marks_crosstab_filtered = question_marks_crosstab.loc[:, question_marks_cr

question_marks_melted = question_marks_crosstab_filtered.reset_index().melt(id_vars

plot.figure(figsize = (12, 6))
sns.histplot(data = question_marks_melted, x = 'Question_Marks', hue = 'Freshness',
plot.title('Side-by-Side Histogram of Question Mark Counts by Freshness (Excluding
plot.xlabel('Number of Question Marks')
plot.ylabel('Count')
plot.legend(title = 'Freshness', labels = ['Fresh', 'Rotten'])
plot.show()
```

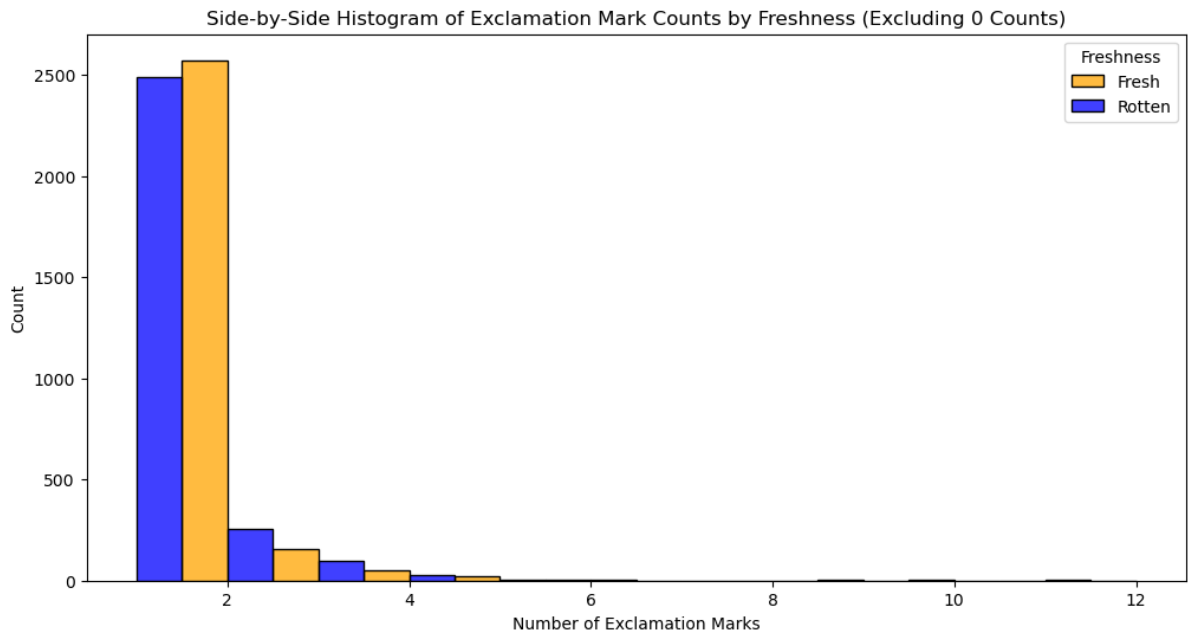


For this histogram, we wanted to see if there was a correlation between the number of question marks in a review and the review rating. We observed that in rotten reviews there were often more question marks than fresh reviews. However we had to exclude reviews with no question marks because overall, most reviews didn't use them at all.

```
In [7]: data['Exclamation_Marks'] = data['Review'].str.count('\!')

exclamation_marks_crosstab = pd.crosstab(data['Freshness'], data['Exclamation_Marks'])
exclamation_marks_crosstab_filtered = exclamation_marks_crosstab.loc[:, exclamation_marks_crosstab > 0]
exclamation_marks_melted = exclamation_marks_crosstab_filtered.reset_index().melt(id_vars='Freshness', var_name='Exclamation_Marks', value_name='Count')

plot.figure(figsize = (12, 6))
sns.histplot(data = exclamation_marks_melted, x = 'Exclamation_Marks', hue = 'Freshness')
plot.title('Side-by-Side Histogram of Exclamation Mark Counts by Freshness (Excluding 0 Counts)')
plot.xlabel('Number of Exclamation Marks')
plot.ylabel('Count')
plot.legend(title = 'Freshness', labels = ['Fresh', 'Rotten'])
plot.show()
```



For this histogram, we found the number of exclamation marks instead of question marks. We saw that fresh and rotten both had roughly the same distribution of exclamation points so we believe there was no correlation to be found there. Again, we had to exclude the reviews with no exclamation marks because overall, most reviews didn't use them at all.

```
In [8]: #Fresh word cloud

fresh = data[data["Freshness"] == '1']
reviews_subset = fresh['Review'].head(480000)
all_reviews = ' '.join(reviews_subset)

wordcloud = WordCloud(width = 1000, height = 500, background_color = 'white').gener

#Plot the word cloud
plot.figure(figsize = (10, 5))
plot.imshow(wordcloud, interpolation = 'bilinear')
plot.axis('off')
plot.show()
```



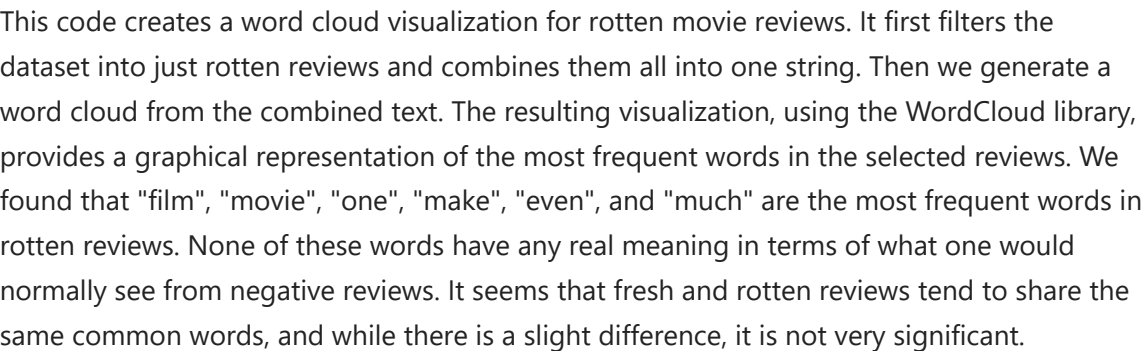
This code creates a word cloud visualization for fresh movie reviews. It first filters the dataset into just fresh reviews and combines them all into one string. Then we generate a word cloud from the combined text. The resulting visualization, using the WordCloud library, provides a graphical representation of the most frequent words in the selected reviews. We found that "movie", "film", "character", "one", and "story" are the most frequent words in fresh reviews. None of these words have any real meaning in terms of what one would normally see from positive reviews.

```
In [9]: #Rotten word cloud

rot = data[data["Freshness"] == '0']
reviews_subset = rot['Review'].head(480000)
all_reviews = ' '.join(reviews_subset)

wordcloud = WordCloud(width = 1000, height = 500, background_color = 'white').generate_from_text(all_reviews)

#Plot the word cloud
plot.figure(figsize = (10, 5))
plot.imshow(wordcloud, interpolation = 'bilinear')
plot.axis('off')
plot.show()
```



****First Data Cleaning Attempt:****

12/13/2023, 12:30 AM

```
In [11]: def processAndVocabularize(review):
    review = review.lower() # make everything lower case, so they aren't marked as
    review = review.replace("full review in spanish",'')#this was slapped on review
    review = review.replace("'s",'') #get rid of excess punctuation/weird chars
    review = review.replace("n't",'')
    review = review.replace("'re",'')
    review = review.replace("films",'')#get rid of words heavily present in both ty
    review = review.replace("film",'')
    review = review.replace("movies",'')
    review = review.replace("movie",'')
    review = review.replace("makes",'')
    review = review.replace("make",'')
    review = review.replace("one",'')
    review = review.replace("ones",'')
    review = review.replace("story",'')
    review = review.replace("stories",'')
    review = review.replace("character",'')
    review = review.replace("characters",'')
    review = review.replace("review",'')
    review = review.replace("reviews",'')
    review = review.replace("even",'')
    review = review.replace("time",'')
    tokens = word_tokenize(review) # tokenize
    table = str.maketrans('', '', punctuation) # remove punctuation from each toke
    tokens = [w.translate(table) for w in tokens]
    tokens = [word for word in tokens if word.isalpha()] # remove remaining tokens
    tokens = [word for word in tokens if len(word) > 1] #get rid of short tokens
    filteredStopTokens = [token for token in tokens if token not in stopwords.words
    lemmatizer = WordNetLemmatizer() # Lemmatize
    lemmatizedTokens = [lemmatizer.lemmatize(token) for token in filteredStopTokens
    vocab.update(lemmatizedTokens)
    processedReview = ' '.join(lemmatizedTokens) #put everything back together beca
    return processedReview

vocab = Counter() #Create a vocabulary
df['PostProcessReview'] = df['Review'].apply(processAndVocabularize) #run the funct
print(len(vocab)) #print how many words in vocab
print(vocab.most_common(50)) #show top 50 words in the vocab
```

108155

```
[('like', 35716), ('much', 22104), ('good', 18867), ('feel', 18064), ('comedy', 170
58), ('performance', 17052), ('way', 16878), ('get', 15660), ('work', 14676), ('nev
er', 14244), ('little', 14135), ('best', 14033), ('director', 13894), ('life', 1383
2), ('enough', 13317), ('would', 12276), ('take', 12225), ('could', 12212), ('actio
n', 12196), ('u', 11997), ('thing', 11763), ('come', 11407), ('may', 11283), ('grea
t', 11097), ('year', 11072), ('every', 10961), ('funny', 10920), ('drama', 10667),
('well', 10650), ('first', 10634), ('go', 10532), ('end', 10504), ('love', 10447),
('fun', 10322), ('something', 10229), ('look', 10228), ('new', 10029), ('see', 1001
6), ('better', 10007), ('made', 9974), ('still', 9968), ('many', 9856), ('really',
9799), ('two', 9772), ('audience', 9677), ('also', 9595), ('moment', 9529), ('might
', 9435), ('bad', 9042), ('world', 8793)]
```

In the above function we first make all the reviews lower case, so that words like "new" and "New" are not counted as separate words. Then we removed excess characters left behind like "n't". Next, we removed words that commonly occur across both reviews that don't provide any meaning, as seen in our EDA word cloud. After cleaning up the strings we finally tokenize them (separate by word). Then we remove any punctuation or character that isn't a letter. Additionally, we got rid of words that weren't at least 2 characters long, as they don't provide any meaning. Next, we removed all the English stop words. Finally, we then lemmatized the words to further reduce the amount of unique words that aren't actually unique. We could have used stemming instead of lemmatization here, but both were good options and we could only choose one.

We also created a library to keep count of the word occurrences. As seen above, doing all this brings down to 108,155 unique words.

```
In [12]: #keep tokens with a min occurrence
min_occurene = 5
tokens = [k for k,c in vocab.items() if c >= min_occurene]
print(len(tokens))

39219
```

Here we remove words that do not occur at least 5 times. Words that appear too little also aren't helpful because they aren't present enough to be a unique identifier. This brings us all way down to 39,219, less than half the words we previously had!

```
In [13]: #TF-IDF
vectorizer = TfidfVectorizer(max_df = 0.75, min_df = 5)
X_tfidf = vectorizer.fit_transform(df["PostProcessReview"])
print(f"n_samples: {X_tfidf.shape[0]}, n_features: {X_tfidf.shape[1]}")

n_samples: 480000, n_features: 39097
```

Here we vectorize the processed reviews so they can be used by our learning models. When vectorizing using TF-IDF we can reduce the number of words even further by setting the max_df to 0.75, which means that words that appear in more than 75% of reviews aren't counted. Our final count of features is 39,097, across 480,000 different reviews! Removing words is also really important because the more features we have, the longer the models take to run.

We used TF-IDF over bag of words because it focused on the relative "importance" of words to the documents rather than just pure counts, which can be misleading.

```
In [14]: #setting up training and testing
x = X_tfidf
y = df["Freshness"]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_s
```

Here we take our vectorized reviews and ratings and split them into testing and training data. We made sure to stratify the Freshness ratings so there are an equal amount of positive and negative reviews in each group. This ensures that our training and testing data isn't skewed.

****Learning Models:****

In [15]:

```
df
```

Out[15]:

	Freshness	Review	PostProcessReview
0	1	Manakamana doesn't answer any questions, yet ...	manakamana answer question yet point nepal lik...
1	1	Wilfully offensive and powered by a chest-thu...	wilfully offensive powered chestthumping machi...
2	0	It would be difficult to imagine material mor...	would difficult imagine material wrong spade l...
3	0	Despite the gusto its star brings to the role...	despite gusto star brings role hard ride shotg...
4	0	If there was a good idea at the core of this ...	good idea core buried unsightly pile flatulenc...
...
479995	0	Zemeckis seems unable to admit that the motio...	zemeckis seems unable admit motion capture ani...
479996	1	Movies like The Kids Are All Right -- beautif...	like kid right beautifully written impeccably ...
479997	0	Film-savvy audiences soon will catch onto Win...	savvy audience soon catch onto winterbottom at...
479998	1	An odd yet enjoyable film.	odd yet enjoyable
479999	1	No other animation studio, even our beloved P...	animation studio beloved pixar quite replicate...

480000 rows × 3 columns

Now we are ready to use our vectorized reviews to run models on them.

****KNN:****

In [16]: *#KNN classification*

```
t0 = time() #used for timing
knn = KNeighborsClassifier(n_neighbors = 7)
knn.fit(x_train, y_train)

y_pred = knn.predict(x_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Score: {accuracy:.2f}")
print("Classification Report:")
print(classification_report(y_test, y_pred))
print(f"Total run time: {time() - t0:.3f} s")
```

Score: 0.52

Classification Report:

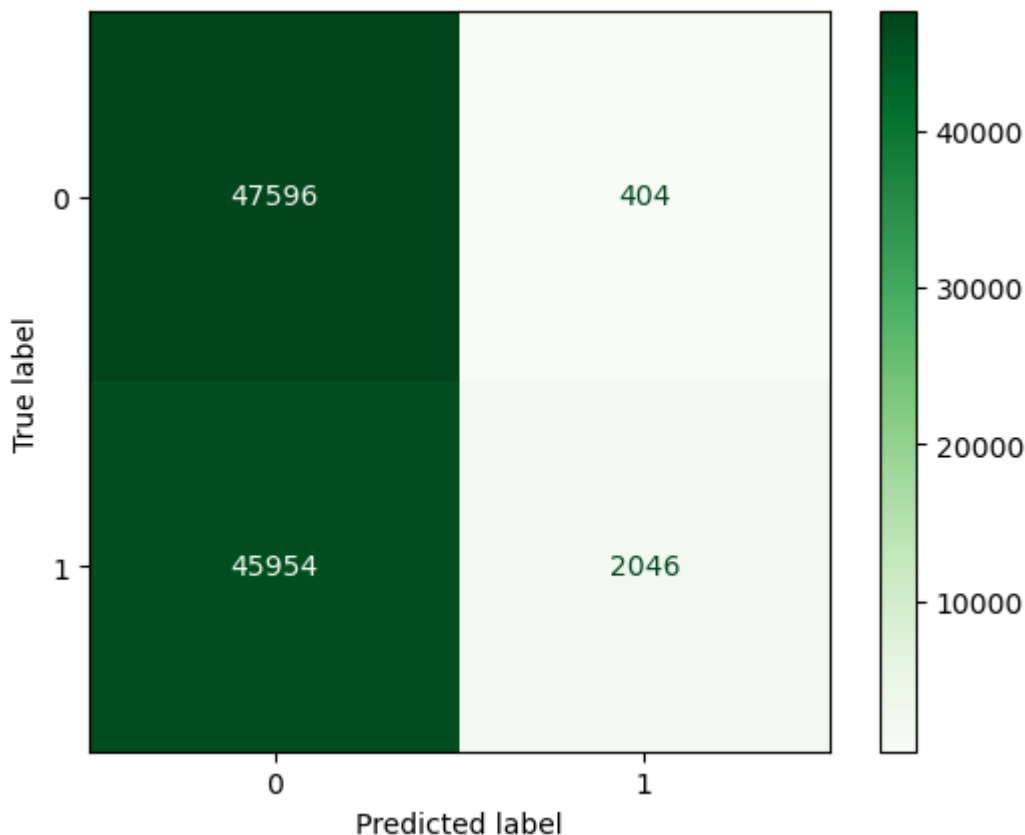
	precision	recall	f1-score	support
0	0.51	0.99	0.67	48000
1	0.84	0.04	0.08	48000
accuracy			0.52	96000
macro avg	0.67	0.52	0.38	96000
weighted avg	0.67	0.52	0.38	96000

Total run time: 1965.037 s

In [17]:

```
print("Confusion Matrix for KNN Classification:")
ConfusionMatrixDisplay.from_estimator(knn, x_test, y_test, cmap = 'Greens', display
plot.show()
```

Confusion Matrix for KNN Classification:



We decided to use $k = 7$ because we got about the same amount of accuracy as high k values but it was way faster to run. Overall KNN was about 52% accurate. The recall for negative reviews was very high, meaning it identified almost all the reviews as rotten, but its precision for rotten reviews was only 0.51, meaning it was only classifying rotten reviews about 51% of the time. Interestingly enough, the opposite happened for positive reviews. Its recall was quite low but its precision was much higher. It had a very hard time identifying the fresh reviews, but it was very good at classifying them.

KNN most likely suffers from the curse of dimensionality due to the amount of features we have, with reviews not looking very "close" to each other.

****Multinomial Naive Bayes:****

```
In [18]: #multinomial naive bayes

t0 = time() #used for timing
mnb = MultinomialNB()
mnb.fit(x_train,y_train)
predmnb = mnb.predict(x_test)
score_mnb = round(accuracy_score(y_test, predmnb) * 100, 2)

print("Score:", score_mnb)
print("Classification Report:")
print(classification_report(y_test, predmnb))
print(f"Total run time: {time() - t0:.3f} s")
```

Score: 79.56

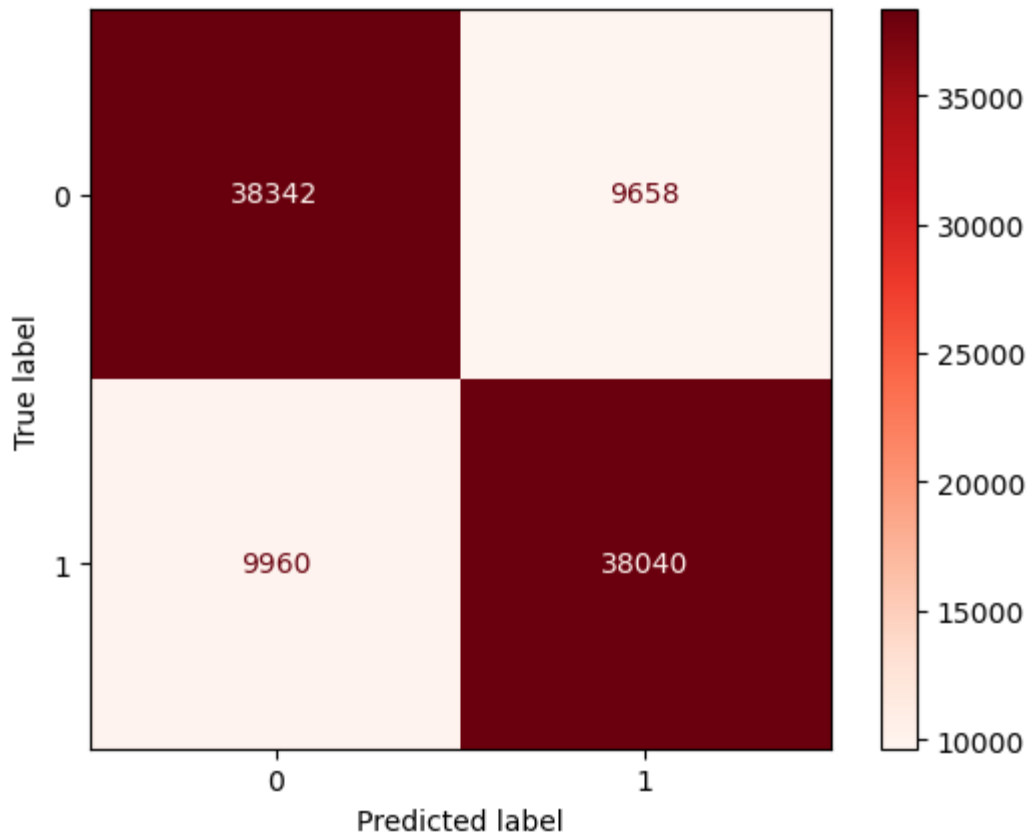
Classification Report:

	precision	recall	f1-score	support
0	0.79	0.80	0.80	48000
1	0.80	0.79	0.80	48000
accuracy			0.80	96000
macro avg	0.80	0.80	0.80	96000
weighted avg	0.80	0.80	0.80	96000

Total run time: 0.232 s

```
In [19]: print("Confusion Matrix for Multinomial Naive Bayes:")
ConfusionMatrixDisplay.from_estimator(mnb, x_test, y_test, cmap = 'Reds', display_1
plot.show())
```

Confusion Matrix for Multinomial Naive Bayes:



Overall, the accuracy of the Multinomial Naive Bayes (MNB) model is about 80%. The recall for rotten reviews is 0.80 and 0.79 for fresh reviews. For precision, rotten reviews are 0.79 and fresh reviews are 0.80. This indicates the effectiveness of this model, as it tends to give higher and more stable results. This is most likely due to the fact that this model works well with discrete data, such as term frequencies, and also performs well with large datasets. The confusion matrix also reflects this by the fact that it has very high true positive and true negative rates.

****Decision Tree:****

```
In [20]: #decision tree

t0 = time() #used for timing
dt = DecisionTreeClassifier()
dt.fit(x_train, y_train)
pred_dt = dt.predict(x_test)
score_dt = round(accuracy_score(y_test, pred_dt) * 100, 2)

print("Score:", score_dt)
print("Classification Report:")
print(classification_report(y_test, pred_dt))
print(f"Total run time: {time() - t0:.3f} s")
```

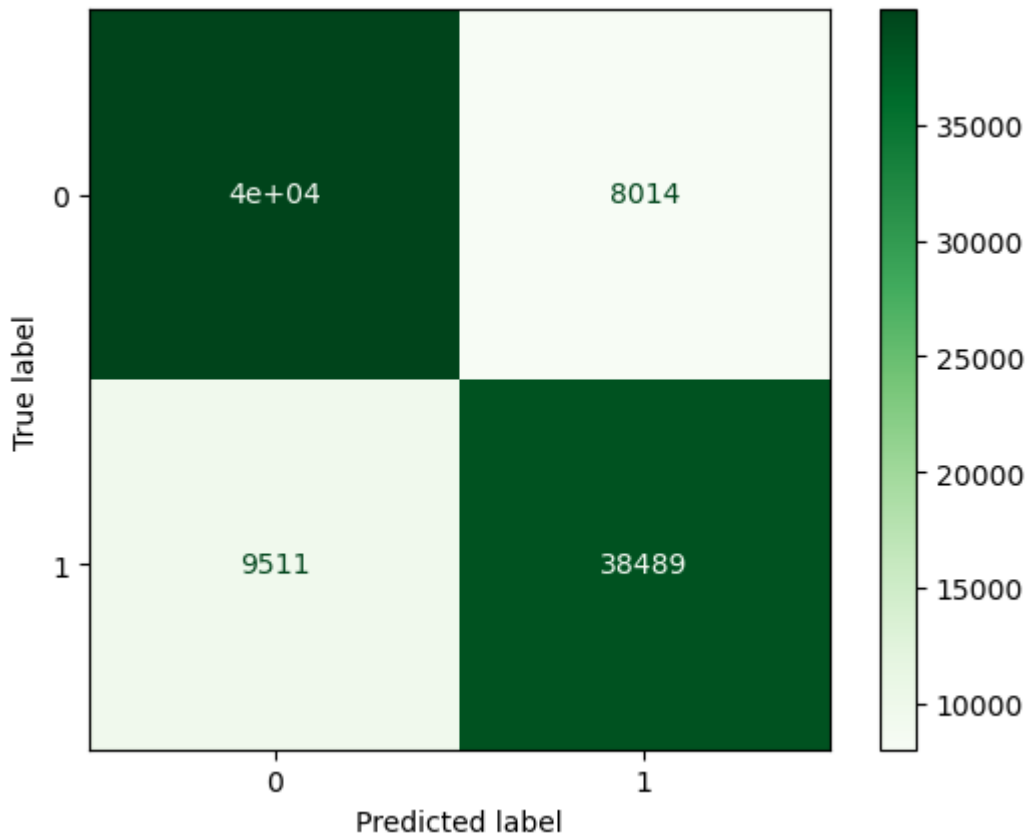
Score: 81.74

Classification Report:

	precision	recall	f1-score	support
0	0.81	0.83	0.82	48000
1	0.83	0.80	0.81	48000
accuracy			0.82	96000
macro avg	0.82	0.82	0.82	96000
weighted avg	0.82	0.82	0.82	96000

```
In [21]: print("Confusion Matrix for Decision Tree:")
ConfusionMatrixDisplay.from_estimator(dt, x_test, y_test, cmap = 'Greens', display_
plot.show())
```

Confusion Matrix for Decision Tree:



Overall, the accuracy of the decision tree is 82%, which is the second highest accuracy out of all the models. The recall was decently high as well at 0.83 for rotten reviews and 0.80 for fresh reviews. The precision for rotten reviews was 0.81 while for fresh it was 0.83. From these percentages, we can see that the decision tree's recall and precision was consistent for both rotten and fresh, showing its effectiveness. The confusion matrix also reflects this by the fact that it has very high true positive and true negative rates.

****Random Forest:****

In [22]: *#random forest*

```
t0 = time() #used for timing
rmfr = RandomForestClassifier()
rmfr.fit(x_train, y_train)
predrmfr = rmfr.predict(x_test)
score_rmfr = round(accuracy_score(y_test, predrmfr) * 100, 2)

print("Score:", score_rmfr)
print("Classification Report:")
print(classification_report(y_test, predrmfr))
print(f"Total run time: {time() - t0:.3f} s")
```

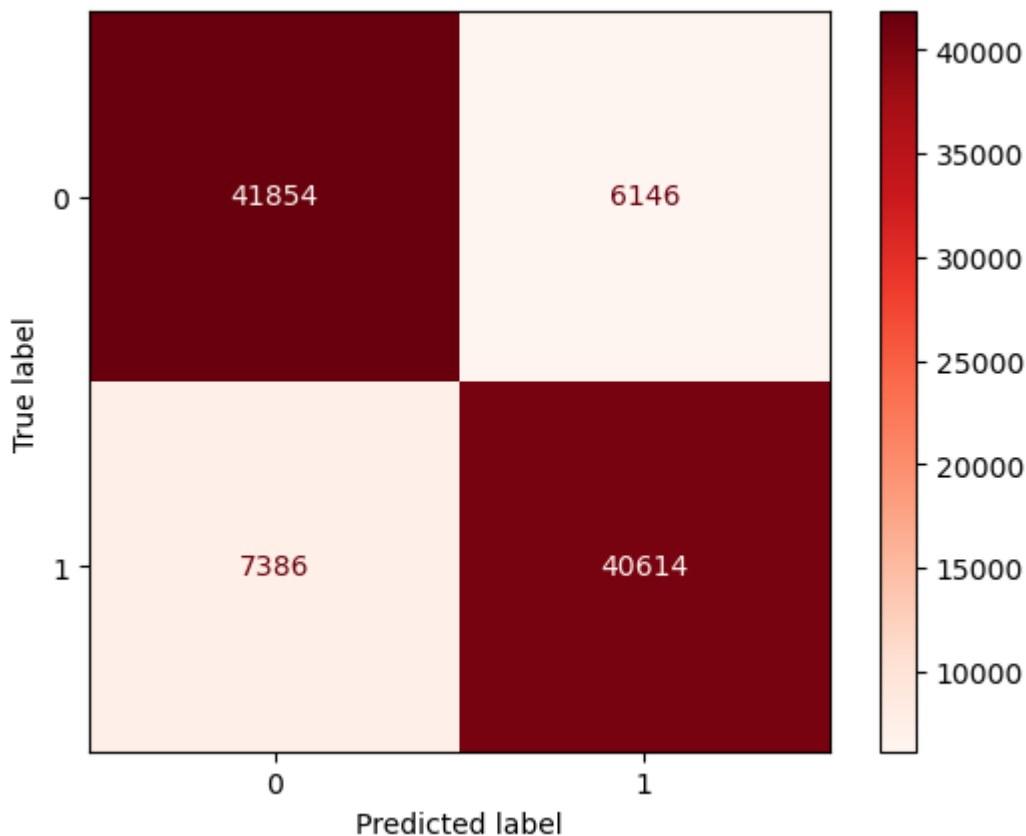
Score: 85.9

Classification Report:

	precision	recall	f1-score	support
0	0.85	0.87	0.86	48000
1	0.87	0.85	0.86	48000
accuracy			0.86	96000
macro avg	0.86	0.86	0.86	96000
weighted avg	0.86	0.86	0.86	96000

In [23]: `print("Confusion Matrix for Random Forest:")`
`ConfusionMatrixDisplay.from_estimator(rmfr, x_test, y_test, cmap = 'Reds', display_`
`plot.show())`

Confusion Matrix for Random Forest:



The random forest also has a very high accuracy score at 85.98%, making it our most accurate model yet. The precision and recall for both the rotten and fresh reviews were pretty high, with precision values of 0.85 and 0.87 and recall values of 0.87 and 0.85 for rotten and fresh reviews respectively. This, combined with the fact that the confusion matrix has very high true positive and true negative values, show how effective this model is.

****KMeans Clustering:****

```
In [24]: #kmeans clustering

kmeans = KMeans(n_clusters = 2, max_iter = 1000, n_init = 10).fit(X_tfidf)
labels = df["Freshness"]
unique_labels, category_sizes = np.unique(labels, return_counts = True)
cluster_ids, cluster_sizes = np.unique(kmeans.labels_, return_counts = True)
print(f"Number of reviews assigned to each cluster: {cluster_sizes}")
```

Number of reviews assigned to each cluster: [445865 34135]

```
In [25]: dfKM = pd.read_csv("rotten_tomatoes_reviews.csv")
dfKM['Cluster ID'] = kmeans.labels_
dfKM
```

```
Out[25]:
```

	Freshness	Review	Cluster ID
0	1	Manakamana doesn't answer any questions, yet ...	1
1	1	Wilfully offensive and powered by a chest-thu...	0
2	0	It would be difficult to imagine material mor...	0
3	0	Despite the gusto its star brings to the role...	0
4	0	If there was a good idea at the core of this ...	0
...
479995	0	Zemeckis seems unable to admit that the motio...	0
479996	1	Movies like The Kids Are All Right -- beautif...	1
479997	0	Film-savvy audiences soon will catch onto Win...	0
479998	1	An odd yet enjoyable film.	0
479999	1	No other animation studio, even our beloved P...	0

480000 rows × 3 columns

```
In [26]: print(classification_report(dfKM['Freshness'], dfKM['Cluster ID']))
```

	precision	recall	f1-score	support
0	0.49	0.92	0.64	240000
1	0.41	0.06	0.10	240000
accuracy			0.49	480000
macro avg	0.45	0.49	0.37	480000
weighted avg	0.45	0.49	0.37	480000

As can be seen in the classification report, the KMeans model only has an accuracy of 49%, slightly below KNN clustering. The recall and precision were pretty low for rotten reviews, but the recall for fresh reviews was quite high. This means that the model identified most reviews as fresh, but wasn't able to classify them well. Meanwhile, the model didn't work well with rotten reviews in general.

We knew that KMeans wouldn't work well for our project, but we were curious to see if the reviews would naturally group themselves into positive and negative. They did not, as the clusters are very unevenly split.

****Textblob vs VADER built-in sentiment analysis:****

****Textblob:****

Textblob is a Python library capable of doing all the natural language processing (NLP) we did earlier such as tokenization and lemmization. It is interesting because it has POS tagging, which can identify the nouns and verbs in words! This is a lexicon based sentiment analyzer, which means it has rules already defined for words it comes across. Since it uses pre-defined rules for words rather than learning based on what other words were classified as, it is an unsupervised technique.

In [27]: *#using TextBlob built in sentiment analysis*

```
from textblob import TextBlob
def getFreshnessTextBlob(review):
    scores = TextBlob(review).sentiment
    freshness = 1 if scores.polarity > 0 else 0
    return freshness

t0 = time()
df['TextBlob_PredictedFreshness'] = df['Review'].apply(getFreshnessTextBlob)
print(f"analyzed Non Processed in {time() - t0:.3f} s") #non processed
t0 = time()
df['TextBlob_PredictedFreshness_P'] = df['PostProcessReview'].apply(getFreshnessTex
print(f"analyzed Processed in {time() - t0:.3f} s") #on processed reviews
df
```

analyzed Non Processed in 134.291 s
analyzed Processed in 96.493 s

Out[27]:

	Freshness	Review	PostProcessReview	TextBlob_PredictedFreshness	TextBlob_Predicti
		Manakamana doesn't answer any questions, yet ...	manakamana answer question yet point nepal lik...	1	
0	1				
		Wilfully offensive and powered by a chest-thu...	wilfully offensive powered chestthumping machi...	1	
1	1				
		It would be difficult to imagine material mor...	would difficult imagine material wrong spade l...	0	
2	0				
		Despite the gusto its star brings to the role...	despite gusto star brings role hard ride shotg...	0	
3	0				
		If there was a good idea at the core of this ...	good idea core buried unsightly pile flatulenc...	0	
4	0				
...	
		Zemeckis seems unable to admit that the motio...	zemeckis seems unable admit motion capture ani...	0	
479995	0				
		Movies like The Kids Are All Right -- beautif...	like kid right beautifully written impeccably ...	1	
479996	1				
		Film-savvy audiences soon will catch onto Win...	savvy audience soon catch onto winterbottom at...	0	
479997	0				
		An odd yet enjoyable film.	odd yet enjoyable	1	
479998	1				
		No other animation studio, even our beloved P...	animation studio beloved pixar quite replicate...	1	
479999	1				

480000 rows × 5 columns

```
In [28]: print(classification_report(df['Freshness'], df['TextBlob_PredictedFreshness']))
```

	precision	recall	f1-score	support
0	0.64	0.50	0.56	240000
1	0.59	0.72	0.65	240000
accuracy			0.61	480000
macro avg	0.61	0.61	0.60	480000
weighted avg	0.61	0.61	0.60	480000

We ran TextBlob sentiment analysis on all of our reviews to see how accurate it was. Above, we first tried the raw, un-processed data to see the accuracy, as this model says it doesn't require text preprocessing. Unfortunately the accuracy isn't very good, only about 61%. It runs a lot faster than the other models, but the models that are actually trained with our data, such as the decision tree and MNB, are way more accurate.

```
In [29]: print(classification_report(df['Freshness'], df['TextBlob_PredictedFreshness_P']))
```

	precision	recall	f1-score	support
0	0.63	0.54	0.58	240000
1	0.60	0.69	0.64	240000
accuracy			0.61	480000
macro avg	0.61	0.61	0.61	480000
weighted avg	0.61	0.61	0.61	480000

Above is TextBlob being run with the processed data. It does run faster than before (~30 sec), but the accuracy is still the about the same. This was expected though because the data doesn't require cleaning or processing for TextBlob, since it has its own built-in ways of handling the relationship between words and predetermined values.

****VADER:****

VADER (Valence Aware Dictionary for Sentiment Reasoning) is another lexicon based sentiment analyzer. We wanted to run VADER to see how it performs against Textblob. VADER is known for performing well with social media analysis because it takes punctuation, such as exclamation points, into account. It provides dictionary definitions for words and associates each word as positive or negative. Then it uses their counts to determine the sentiment of a sentence.

```
In [30]: #using VADER built in sentiment analysis
analyze = SentimentIntensityAnalyzer()

def getFreshness(review):
    scores = analyze.polarity_scores(review)
    freshness = 1 if scores['pos'] > 0 else 0
    return freshness

t0 = time()
df['VADER_PredictedFreshness'] = df['Review'].apply(getFreshness) #non processed
print(f"analyzed Non Processed in {time() - t0:.3f} s")
t0 = time()
df['VADER_PredictedFreshness_P'] = df['PostProcessReview'].apply(getFreshness)
print(f"analyzed Processed in {time() - t0:.3f} s") #on processed reviews
df
```

analyzed Non Processed in 124.732 s

analyzed Processed in 77.530 s

Out[30]:

	Freshness	Review	PostProcessReview	TextBlob_PredictedFreshness	TextBlob_Predicti
0	1	Manakamana doesn't answer any questions, yet ...	manakamana answer question yet point nepal lik...	1	
1	1	Wilfully offensive and powered by a chest-thu...	wilfully offensive powered chestthumping machi...	1	
2	0	It would be difficult to imagine material mor...	would difficult imagine material wrong spade l...	0	
3	0	Despite the gusto its star brings to the role...	despite gusto star brings role hard ride shotg...	0	
4	0	If there was a good idea at the core of this ...	good idea core buried unsightly pile flatulenc...	0	
...	
479995	0	Zemeckis seems unable to admit that the motio...	zemeckis seems unable admit motion capture ani...	0	
479996	1	Movies like The Kids Are All Right -- beautif...	like kid right beautifully written impeccably ...	1	
479997	0	Film-savvy audiences soon will catch onto Win...	savvy audience soon catch onto winterbottom at...	0	
479998	1	An odd yet enjoyable film.	odd yet enjoyable	1	
479999	1	No other animation studio, even our beloved P...	animation studio beloved pixar quite replicate...	1	

480000 rows × 7 columns

```
In [31]: print(classification_report(df['Freshness'], df['VADER_PredictedFreshness']))
```

	precision	recall	f1-score	support
0	0.64	0.35	0.45	240000
1	0.55	0.81	0.66	240000
accuracy			0.58	480000
macro avg	0.60	0.58	0.55	480000
weighted avg	0.60	0.58	0.55	480000

We ran VADER sentiment analysis on all of our reviews to see how accurate it was compared to Textblob. Above, we again first tried feeding it the un-processed data. Overall, it also isn't the most accurate, and is slightly worse than TextBlob at 58%. It also doesn't really need preprocessing on its input. Again, the models we trained (except KNN) do a lot better but are slower.

```
In [32]: print(classification_report(df['Freshness'], df['VADER_PredictedFreshness_P']))
```

	precision	recall	f1-score	support
0	0.64	0.33	0.43	240000
1	0.55	0.82	0.66	240000
accuracy			0.57	480000
macro avg	0.59	0.57	0.54	480000
weighted avg	0.59	0.57	0.54	480000

Above we can see how the results are affected by processed data. Again, it does run faster than before, but the results aren't significantly different. It is interesting to note though that TextBlob and VADER take around the same amount of time to run.

Overall, they were not as accurate as most of the models we trained but they did run a lot faster. One drawback to these lexicon models is that they don't do very well in context-specific situations. The word values are predefined and their values are not affected by the context of the data. Our learning models were actually trained on the data we had, so they are a better fit for our project. Another drawback is these lexicon models just ignore words that don't have a pre-defined value, where our models don't care about the actual definitions.

****Conclusion:****

In conclusion, we set out to determine if it was possible to analyze movie reviews from critics to determine if the critic gave the movie a fresh (positive) or rotten (negative) review. We were relatively successful, but it is fairly difficult to execute. Even pre-trained professionally available models such as VADER and TextBlob weren't able to perfectly classify the sentiment from the reviews. Due to the sheer amount of reviews we had, it generated a lot of unique words. Too many words makes the models run slow, and it makes them usually run worse, due to having too many features and not being able to properly focus on the ones that matter. We saw this curse of dimensionality in effect in our models, and ultimately that is what lead to them not being as accurate as they could have been.

Our best performing model was: Random Forest

The Random Forest algorithm had an accuracy of about 86%. This is because random forests perform well for datasets with a large number of features and where the number of features is much greater than the number of samples. They are well suited to modeling complex relationships and interactions in the data, which makes it work very effectively in our dataset. In the algorithm, the precision and recall for both positive and negative reviews were very high, 85% and 87% for negative, and 87% and 85% for positive, respectively. This means that the model was able to accurately identify and classify both negative and positive reviews.

Our worst performing model was: KNN

The KNN algorithm had an accuracy of about 52%. This is because KNN is sensitive to noisy data and suffers from the curse of dimensionality. Despite our extensive data cleaning, we still have a lot of features and outliers which makes it hard for the KNN algorithm to produce accurate results. For our k value as well, we could not find an ideal k value. We tried using the elbow method but each run of KNN would take hours and by the time we got to $k = 30$, there was no clear elbow point. We decided to use $k = 7$ for convenience as it would produce similar results at a much faster time. In the algorithm, the recall for negative reviews was high (99%) and its precision very low (50%), meaning that although it identified almost all of the rotten reviews, it was sensitive to rotten reviews and would classify them incorrectly. For positive reviews it was the opposite, marking a review correctly 84% of the time, but identifying fresh reviews at a very low 4%.