Amazon Fine Food Reviews Analysis

Data Source: https://www.kaggle.com/snap/amazon-fine-food-reviews (https://www.kaggle.com/snap/amazon-fine-food-reviews)

EDA: https://nycdatascience.com/blog/student-works/amazon-fine-foods-visualization/ (https://nycdatascience.com/blog/student-works/amazon-fine-foods-visualization/)

The Amazon Fine Food Reviews dataset consists of reviews of fine foods from Amazon.

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

Attribute Information:

1. Id
2. ProductId - unique identifier for the product
3. UserId - unqiue identifier for the user
4. ProfileName
5. HelpfulnessNumerator - number of users who found the review helpful
6. HelpfulnessDenominator - number of users who indicated whether they found the review helpful or not
7. Score - rating between 1 and 5
8. Time - timestamp for the review
9. Summary - brief summary of the review
10. Text - text of the review

Objective:

Given a review, determine whether the review is positive (rating of 4 or 5) or negative (rating of 1 or 2).

[Q] How to determine if a review is positive or negative?

[Ans] We could use Score/Rating. A rating of 4 or 5 can be cosnidered as a positive review. A rating of 1 or 2 can be considered as negative one. A review of rating 3 is considered nuetral and such reviews are ignored from our analysis. This is an approximate and proxy way of determining the polarity (positivity/negativity) of a review.

[1]. Reading Data

[1.1] Loading the data

In [1]:

```
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

import sqlite3
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re

import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem import SnowballStemmer

from bs4 import BeautifulSoup
from sklearn.feature_extraction.text import TfidfVectorizer,TfidfTransformer,CountVecto
rizer
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler

from sklearn.metrics import roc_auc_score, roc_curve,confusion_matrix,auc,accuracy_scor
e,classification_report,precision_score,recall_score,f1_score

from sklearn.naive_bayes import MultinomialNB

from prettytable import PrettyTable

from tqdm import tqdm
```

In [2]:

```
# using SQLite Table to read data.
con = sqlite3.connect('D:\Study_materials\Applied_AI\Assignments\database.sqlite')
filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3""", con)

# Give reviews with Score>3 a positive rating(1), and reviews with a score<3 a negative
rating(0).
def partition(x):
    if x < 3:
        return 0
    return 1

#changing reviews with score less than 3 to be positive and vice-versa
actualScore = filtered_data['Score']
positiveNegative = actualScore.map(partition)
filtered_data['Score'] = positiveNegative
print("Number of data points in our data", filtered_data.shape)
```

```
Number of data points in our data (525814, 10)
```

[2] Exploratory Data Analysis

[2.1] Data Cleaning: Deduplication

It is observed that the reviews data had many duplicate entries. Hence it was necessary to remove duplicates in order to get unbiased results for the analysis of the data. Following is an example:

In [3]:

```
#Deduplication of entries
final=filtered_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"}, keep
='first', inplace=False)
final.shape
```

Out[3]:

```
(364173, 10)
```

In [4]:

```
#Checking to see how much % of data still remains
(final['Id'].size*1.0)/(filtered_data['Id'].size*1.0)*100
```

Out[4]:

```
69.25890143662969
```

In [5]:

```
final=final[final.HelpfulnessNumerator<=final.HelpfulnessDenominator]
```

In [6]:

```
#Before starting the next phase of preprocessing lets see the number of entries left
print(final.shape)

#How many positive and negative reviews are present in our dataset?
final['Score'].value_counts()
```

```
(364171, 10)
```

Out[6]:

```
1    307061
0     57110
Name: Score, dtype: int64
```

In [7]:

```
final = final.sort_values(['Time'], axis = 0)
```

In [8]:

```
final = final.head(100000)
final_x = final['Text']
final_y = final['Score']
```

[3] Preprocessing

Steps will involve:

- Stopword removal
- Stemming
- Punctuation removal

In [9]:

```
stopwords= set(['br', 'the', 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've",\
            "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', \
            'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their',\
            'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', \
            'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', \
            'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', \
            'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after',\
            'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further',\
            'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',\
            'most', 'other', 'some', 'such', 'only', 'own', 'same', 'so', 'than', 'too', 'very', \
            's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', \
            've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn',\
            "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn',\
            "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", \
            'won', "won't", 'wouldn', "wouldn't"])
```

In [10]:

```
def decontracted(phrase):
    # specific
    phrase = re.sub(r"won't", "will not", phrase)
    phrase = re.sub(r"can\'t", "can not", phrase)

    # general
    phrase = re.sub(r"n\'t", " not", phrase)
    phrase = re.sub(r"\'re", " are", phrase)
    phrase = re.sub(r"\'s", " is", phrase)
    phrase = re.sub(r"\'d", " would", phrase)
    phrase = re.sub(r"\'ll", " will", phrase)
    phrase = re.sub(r"\'t", " not", phrase)
    phrase = re.sub(r"\'ve", " have", phrase)
    phrase = re.sub(r"\'m", " am", phrase)
    return phrase
```

In [11]:

```
# Code to remove URLs,HTML tags, words with numbers and special characters, making words lower
preprocessed_reviews = []
# tqdm is for printing the status bar
for sentance in tqdm(final_x):
    sentance = re.sub(r"http\S+", "", sentance)
    sentance = BeautifulSoup(sentance, 'lxml').get_text()
    sentance = decontracted(sentance)
    sentance = re.sub("\S*\d\S*", "", sentance).strip()
    sentance = re.sub('[^A-Za-z]+', ' ', sentance)
    # https://gist.github.com/sebleier/554280
    sentance = ' '.join(e.lower() for e in sentance.split() if e.lower() not in stopwords)
    preprocessed_reviews.append(sentance.strip())
```

```
100%|██████████████████████████████████████████████████████████████| 100000/100000 [00:57<00:00, 1732.59it/s]
```

In [12]:

```
final_x= preprocessed_reviews
```

In [13]:

```
# Let's split final_x and final_y datsets into train and test sets
X_train = final_x[:80000]
X_test = final_x[80000:]
y_train = final_y[:80000]
y_test = final_y[80000:]
```

Applying Multinomial Naive Bayes

[4.1] Naive bayes algo implementation on BOW

In [14]:

```
count_vec = CountVectorizer(max_features=5000)
bow_X_train = count_vec.fit_transform(X_train)
bow_X_test = count_vec.transform(X_test)
```

In [15]:

```
alpha = [0.0001,0.001,0.01,0.1,1,10,100,1000,10000]
est = MultinomialNB(class_prior = [0.5,0.5])
param_grid = { 'alpha':alpha}
grid = GridSearchCV(estimator=est,param_grid=param_grid, scoring = 'roc_auc',cv=10, n_jobs= -1)
grid.fit(bow_X_train,y_train)
train_acc1 = grid.best_score_*100
al1 = grid.best_params_.get('alpha')
print('The best alpha:',al1)
print('Accuracy on Training data:',train_acc1,'%')
```

```
The best alpha: 1
Accuracy on Training data: 91.03445940935924 %
```

In [16]:

```
grid.cv_results_
```

Out[16]:

```
{'mean_fit_time': array([0.12497199, 0.13122008, 0.14840267, 0.14059293,
0.13434446,
        0.13121972, 0.12965734, 0.13746848, 0.13121958]),
 'mean_score_time': array([0.0124975 , 0.01249743, 0.01249738, 0.00781052,
0.00937235,
        0.01405942, 0.01406057, 0.00937328, 0.00468659]),
 'mean_test_score': array([0.90613906, 0.90699971, 0.90794926, 0.90891198,
0.91034459,
        0.90331396, 0.74051724, 0.54768228, 0.48935779]),
 'mean_train_score': array([0.9339973 , 0.93392658, 0.93380352, 0.9335641
, 0.93271787,
        0.92217362, 0.75057886, 0.54945454, 0.49022653]),
 'param_alpha': masked_array(data=[0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1
000, 10000],
              mask=[False, False, False, False, False, False, False, Fals
e,
                    False],
        fill_value='?',
             dtype=object),
 'params': [{'alpha': 0.0001},
  {'alpha': 0.001},
  {'alpha': 0.01},
  {'alpha': 0.1},
  {'alpha': 1},
  {'alpha': 10},
  {'alpha': 100},
  {'alpha': 1000},
  {'alpha': 10000}],
 'rank_test_score': array([5, 4, 3, 2, 1, 6, 7, 8, 9]),
 'split0_test_score': array([0.90960283, 0.91100992, 0.91255259, 0.9142590
4, 0.91739839,
        0.90856368, 0.77361133, 0.59297315, 0.5331399 ]),
 'split0_train_score': array([0.93504101, 0.9349716 , 0.93485756, 0.934624
67, 0.93371054,
        0.92270929, 0.74723659, 0.54470051, 0.48535505]),
 'split1_test_score': array([0.91764405, 0.91800622, 0.91843606, 0.9189970
6, 0.91969121,
        0.91273538, 0.75510751, 0.5532276 , 0.49015817]),
 'split1_train_score': array([0.93398082, 0.93392142, 0.93381745, 0.933609
81, 0.93284814,
        0.92257251, 0.75026637, 0.548638  , 0.4897703 ]),
 'split2_test_score': array([0.91077883, 0.91124811, 0.91203857, 0.9130509
4, 0.9148804 ,
        0.91122162, 0.74878324, 0.54206663, 0.47941662]),
 'split2_train_score': array([0.93403518, 0.93396585, 0.9338478 , 0.933622
38, 0.9327991 ,
        0.92239371, 0.75035673, 0.54984967, 0.49086812]),
 'split3_test_score': array([0.90367165, 0.90511276, 0.9066797 , 0.9082660
9, 0.91016731,
        0.90270861, 0.73900036, 0.54938241, 0.49111491]),
 'split3_train_score': array([0.93389933, 0.93382728, 0.9336993 , 0.933448
76, 0.93263258,
        0.92237893, 0.75043093, 0.54909676, 0.49004098]),
 'split4_test_score': array([0.90760383, 0.9082709 , 0.90881555, 0.9092102
7, 0.90929881,
        0.90144851, 0.7565703 , 0.57283988, 0.51539391]),
 'split4_train_score': array([0.93483851, 0.93477407, 0.93466405, 0.934453
94, 0.9337273 ,
        0.92369654, 0.75055282, 0.54714807, 0.48753771]),
```

```
 'split5_test_score': array([0.86695007, 0.86720379, 0.86740198, 0.8675006
2, 0.86716987,
        0.84934102, 0.66888664, 0.493549  , 0.44216787]),
 'split5_train_score': array([0.93248171, 0.93240817, 0.93229103, 0.932067
44, 0.93121586,
        0.9203025 , 0.75463688, 0.55506645, 0.49524161]),
 'split6_test_score': array([0.90424381, 0.90471379, 0.90520066, 0.9056155
4, 0.90628429,
        0.90183867, 0.71805495, 0.5226327 , 0.46940831]),
 'split6_train_score': array([0.93439073, 0.93432506, 0.93420802, 0.933970
5 , 0.93308094,
        0.92236462, 0.75338108, 0.5521116 , 0.49258856]),
 'split7_test_score': array([0.90749664, 0.90920355, 0.91109036, 0.9129625
7, 0.91532158,
        0.91256828, 0.74653544, 0.55230428, 0.49447023]),
 'split7_train_score': array([0.93453128, 0.93445156, 0.93430285, 0.934021
18, 0.93308935,
        0.92273923, 0.75114408, 0.54973845, 0.49062795]),
 'split8_test_score': array([0.92156284, 0.92221352, 0.92310105, 0.9237678
8, 0.92565098,
        0.92196195, 0.7551974 , 0.55090737, 0.48933667]),
 'split8_train_score': array([0.93292163, 0.93285579, 0.93273738, 0.932488
23, 0.93159316,
        0.92063718, 0.74816109, 0.54881542, 0.49000275]),
 'split9_test_score': array([0.91183139, 0.91300982, 0.91417137, 0.9154849
5, 0.91757816,
        0.91074687, 0.74340762, 0.54692173, 0.48895571]),
 'split9_train_score': array([0.93385279, 0.93376496, 0.93360976, 0.933334
11, 0.93248169,
        0.92194166, 0.74962199, 0.54938053, 0.49023225]),
 'std_fit_time': array([0.00988035, 0.00765232, 0.01746511, 0.01562154, 0.
0076522 ,
        0.01593077, 0.01220133, 0.0136166 , 0.0103626 ]),
 'std_score_time': array([0.01169016, 0.00624871, 0.00937293, 0.00781052,
0.01036181,
        0.00468647, 0.00468686, 0.01036241, 0.00715889]),
 'std_test_score': array([0.01409052, 0.01421836, 0.01442158, 0.01467577,
0.01533128,
        0.01893724, 0.02744352, 0.02517831, 0.02312679]),
 'std_train_score': array([0.00075652, 0.00075726, 0.00075758, 0.00075894,
0.00076693,
        0.0009554 , 0.00206739, 0.00260614, 0.00250114])}
```

In [17]:

```
mean_train_score = []
for i in grid.cv_results_['mean_train_score']:
    mean_train_score.append(i*100)
```

In [18]:

```
mean_train_score
```

Out[18]:

```
[93.39972985513987,
 93.3926575899755,
 93.38035202437872,
 93.35641022256199,
 93.27178675609638,
 92.21736152785505,
 75.05788569345964,
 54.94545446876734,
 49.02265273341882]
```

In [19]:

```
mean_test_score = []
for j in grid.cv_results_['mean_test_score']:
    mean_test_score.append(j*100)
```

In [20]:

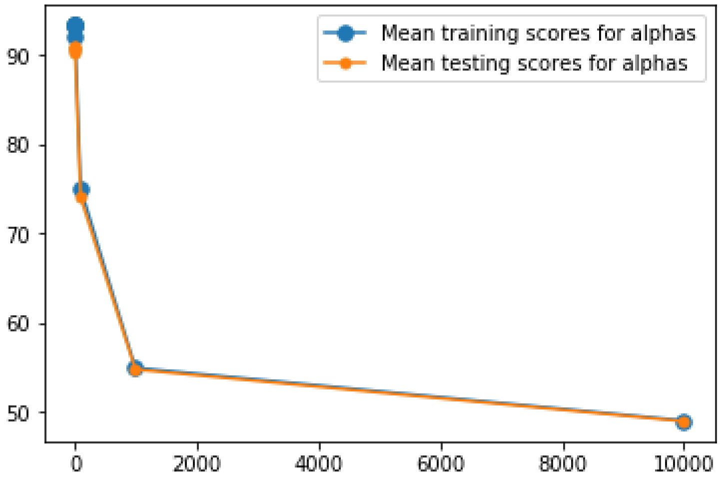
```
mean_test_score
```

Out[20]:

```
[90.61390593623938,
 90.69997052129615,
 90.79492578582293,
 90.89119774835716,
 91.03445940935924,
 90.33139622367639,
 74.05172424947342,
 54.768227815448334,
 48.93577902032044]
```

In [21]:

```
plt.plot(alpha,mean_train_score,marker='.',markersize = 15,label='Mean training scores
 for alphas')
plt.plot(alpha,mean_test_score,marker='o', markersize= 5, label = 'Mean testing scores
 for alphas')
plt.legend(loc='upper right')
plt.show()
```



In [22]:

```
est_opt1 = MultinomialNB(alpha=al1)
est_opt1.fit(bow_X_train,y_train)
predict = est_opt1.predict(bow_X_test)
```

Measure of effectiveness

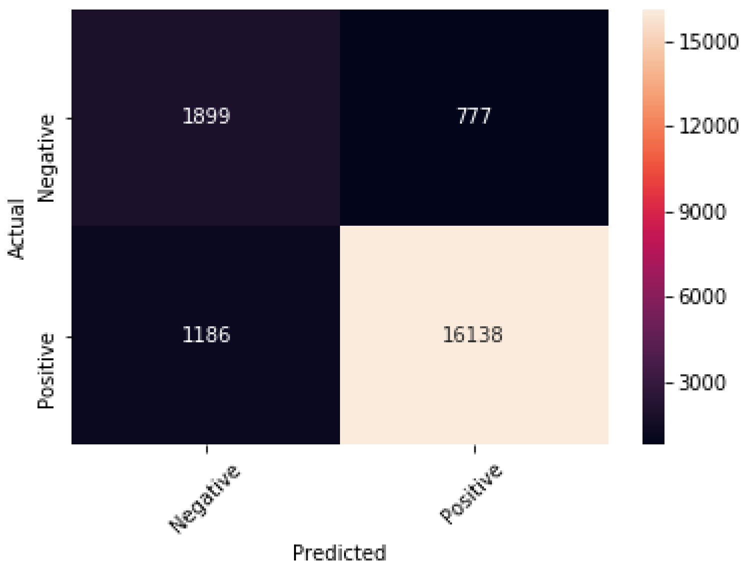
In [23]:

```
test_acc1 = accuracy_score(y_test,predict)*100
precision1 = precision_score(y_test,predict)*100
recall1 = recall_score(y_test,predict)*100
f11 = f1_score(y_test,predict)*100
print('Accuracy on test data:',test_acc1,'%')
print('Precision Score:',precision1,'%')
print('Recall Score:',recall1,'%')
print('F1 score:',f11,'%')

cm = confusion_matrix(y_test,predict)
print('Confusion Matrix:','\n',cm)
```

```
Accuracy on test data: 90.185 %
Precision Score: 95.40644398462904 %
Recall Score: 93.15400600323251 %
F1 score: 94.26677180992435 %
Confusion Matrix:
 [[ 1899   777]
 [ 1186 16138]]
```

In [24]:

```
cm_df = pd.DataFrame(cm, index = ['Negative','Positive'])
sns.heatmap(cm_df, annot = True,fmt = 'd')
plt.xticks([0.5,1.5],['Negative','Positive'],rotation = 45)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```



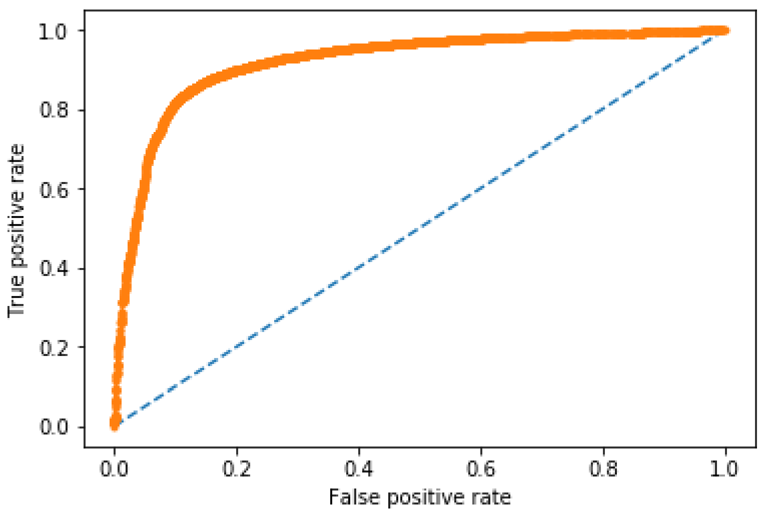
In [25]:

```
print(classification_report(y_test,predict))
```

```
              precision    recall  f1-score   support

           0       0.62      0.71      0.66      2676
           1       0.95      0.93      0.94     17324

   micro avg       0.90      0.90      0.90     20000
   macro avg       0.78      0.82      0.80     20000
weighted avg       0.91      0.90      0.90     20000
```

In [26]:

```
#Plotting Roc curve

y_pred = est_opt1.predict_proba(bow_X_test)[:,1]

fpr,tpr,threshold = roc_curve(y_test,y_pred)

plt.plot([0,1],[0,1],linestyle = '--')
plt.plot(fpr,tpr,marker='.')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.show()
```



In [27]:

```
roc_score1 = roc_auc_score(y_test,y_pred)
print(roc_score1)
```

```
0.919653140238673
```

In [28]:

```
neg_imp_word = list(map(abs,est_opt1.feature_log_prob_))[0].argsort()[0:10]
pos_imp_word = list(map(abs,est_opt1.feature_log_prob_))[1].argsort()[0:10]
```

[4.1.1] Top 10 important features of positive class

In [29]:

```
print('Below are the Top 10 Positive impacting words:')
pos1 = []
for i in pos_imp_word:
    for j in count_vec.vocabulary_:
        if count_vec.vocabulary_[j] == i:
            pos1.append(j)
print(pos1)
```

```
Below are the Top 10 Positive impacting words:
['not', 'like', 'good', 'great', 'tea', 'one', 'taste', 'flavor', 'produc
t', 'love']
```

[4.1.2] Top 10 important features of negative class

In [30]:

```
print('Below are the Top 10 Negative impacting words: ')
neg1 = []
for i in neg_imp_word:
    for j in count_vec.vocabulary_:
        if count_vec.vocabulary_[j] == i:
            neg1.append(j)
print(neg1)
```

```
Below are the Top 10 Negative impacting words:
['not', 'like', 'product', 'taste', 'would', 'one', 'good', 'no', 'flavo
r', 'tea']
```

[4.2] Naive bayes algo implementation on TFIDF

In [31]:

```
tfidf_vec = TfidfVectorizer(max_features=5000)
tfidf_X_train = tfidf_vec.fit_transform(X_train)
tfidf_X_test = tfidf_vec.transform(X_test)
```

In [32]:

```
alpha = [0.0001,0.001,0.01,0.1,1,10,100,1000,10000]
est = MultinomialNB(class_prior = [0.5,0.5])
param_grid = { 'alpha':alpha}
grid = GridSearchCV(estimator=est,param_grid=param_grid, scoring = 'roc_auc',cv=10, n_j
obs= -1)
grid.fit(tfidf_X_train,y_train)
train_acc2 = grid.best_score_*100
al2 = grid.best_params_.get('alpha')
print('The best alpha:',al2)
print('Accuracy on Training data:',train_acc2,'%')
```

```
The best alpha: 0.1
Accuracy on Training data: 92.23124992729143 %
```

In [33]:

```
grid.cv_results_
```

Out[33]:

```
{'mean_fit_time': array([0.11614771, 0.12809553, 0.12184703, 0.13278198,
0.1299963 ,
        0.13122084, 0.13278201, 0.12184758, 0.11872199]),
 'mean_score_time': array([0.00781074, 0.00937347, 0.00624855, 0.01093528,
0.01093578,
        0.01249635, 0.01093507, 0.01249716, 0.00781038]),
 'mean_test_score': array([0.91780182, 0.91884827, 0.9201667 , 0.9223125 ,
0.92215364,
        0.86755657, 0.72023386, 0.6388261 , 0.60761955]),
 'mean_train_score': array([0.94589006, 0.94583293, 0.94571585, 0.9453292
8, 0.94008739,
        0.8813211 , 0.72596093, 0.64152472, 0.60975908]),
 'param_alpha': masked_array(data=[0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1
000, 10000],
              mask=[False, False, False, False, False, False, False, Fals
e,
                    False],
        fill_value='?',
             dtype=object),
 'params': [{'alpha': 0.0001},
  {'alpha': 0.001},
  {'alpha': 0.01},
  {'alpha': 0.1},
  {'alpha': 1},
  {'alpha': 10},
  {'alpha': 100},
  {'alpha': 1000},
  {'alpha': 10000}],
 'rank_test_score': array([5, 4, 3, 1, 2, 6, 7, 8, 9]),
 'split0_test_score': array([0.91417279, 0.9153408 , 0.91711715, 0.9204739
8, 0.92135291,
        0.86445743, 0.73900754, 0.66593627, 0.63398576]),
 'split0_train_score': array([0.94642592, 0.94637119, 0.94625198, 0.945800
92, 0.94008447,
        0.88112309, 0.7238146 , 0.63851968, 0.60672258]),
 'split1_test_score': array([0.92449137, 0.92532503, 0.92634377, 0.9277847
3, 0.92615277,
        0.86561997, 0.71461766, 0.63215867, 0.59939891]),
 'split1_train_score': array([0.94566561, 0.94561783, 0.94552159, 0.945184
86, 0.94004495,
        0.88119347, 0.72552957, 0.64120851, 0.60947865]),
 'split2_test_score': array([0.91887708, 0.91965941, 0.92069515, 0.9227475
7, 0.92440527,
        0.87503403, 0.72825593, 0.64099811, 0.60726341]),
 'split2_train_score': array([0.94618706, 0.94613284, 0.94601864, 0.945632
45, 0.94036827,
        0.88091652, 0.72471119, 0.6404911 , 0.60874331]),
 'split3_test_score': array([0.91509943, 0.91685818, 0.91918828, 0.9224581
2, 0.92187493,
        0.86473978, 0.71926225, 0.63935055, 0.60843001]),
 'split3_train_score': array([0.94567786, 0.94561752, 0.94549667, 0.945126
14, 0.94005165,
        0.88194482, 0.72600305, 0.64137518, 0.60962128]),
 'split4_test_score': array([0.92147348, 0.92220684, 0.92302585, 0.9241737
4, 0.92175946,
        0.86734262, 0.73055748, 0.65608948, 0.62684026]),
 'split4_train_score': array([0.94627827, 0.94622792, 0.94612435, 0.945777
81, 0.94073076,
        0.88195854, 0.72478979, 0.63934809, 0.60726603]),
```

```
 'split5_test_score': array([0.89721582, 0.89756331, 0.89805076, 0.8991256
6, 0.89552994,
        0.82582013, 0.66981897, 0.59307058, 0.56433445]),
 'split5_train_score': array([0.94544602, 0.94539246, 0.94528827, 0.944945
48, 0.93986557,
        0.88223679, 0.73038437, 0.64619829, 0.61421443]),
 'split6_test_score': array([0.91974469, 0.92042426, 0.92115966, 0.9225235
4, 0.92257167,
        0.86857144, 0.70425462, 0.61966426, 0.59122999]),
 'split6_train_score': array([0.94604566, 0.9459887 , 0.9458697 , 0.945479
22, 0.94029306,
        0.88231069, 0.72911299, 0.64472157, 0.61268996]),
 'split7_test_score': array([0.91635522, 0.91829661, 0.92047587, 0.9233785
5, 0.92539148,
        0.8780579 , 0.73621552, 0.65580091, 0.62393577]),
 'split7_train_score': array([0.94605891, 0.9459902 , 0.94585124, 0.945436
37, 0.94026845,
        0.88173944, 0.72586249, 0.6411533 , 0.60974396]),
 'split8_test_score': array([0.92911784, 0.92966558, 0.93047356, 0.9323954
, 0.93342497,
        0.88751674, 0.73805263, 0.6505276 , 0.61741086]),
 'split8_train_score': array([0.9453591 , 0.94530132, 0.94518051, 0.944773
93, 0.93938161,
        0.87941542, 0.7238811 , 0.64019945, 0.60873237]),
 'split9_test_score': array([0.92146916, 0.92314141, 0.92513553, 0.9280622
, 0.92907177,
        0.87840583, 0.7222884 , 0.6346545 , 0.60335666]),
 'split9_train_score': array([0.94575616, 0.94568935, 0.94555549, 0.945135
65, 0.93978511,
        0.88037221, 0.72552012, 0.64203199, 0.6103782 ]),
 'std_fit_time': array([0.01157365, 0.01530623, 0.01530647, 0.01600642, 0.
01712152,
        0.01593203, 0.01746483, 0.01361861, 0.01249568]),
 'std_score_time': array([0.00781074, 0.00765341, 0.00765287, 0.01000299,
0.0100033 ,
        0.00937275, 0.00715868, 0.00937302, 0.00781038]),
 'std_test_score': array([0.00806815, 0.00813245, 0.00823355, 0.00842885,
0.00957189,
        0.0156407 , 0.01982336, 0.01996352, 0.01911879]),
 'std_train_score': array([0.00034223, 0.00034272, 0.00034189, 0.00033239,
0.00034781,
        0.00086829, 0.00204353, 0.00221749, 0.00215506])}
```

In [34]:

```
mean_train_score = []
for i in grid.cv_results_['mean_train_score']:
    mean_train_score.append(i*100)
```

In [35]:

```
mean_test_score = []
for j in grid.cv_results_['mean_test_score']:
    mean_test_score.append(j*100)
```

In [36]:

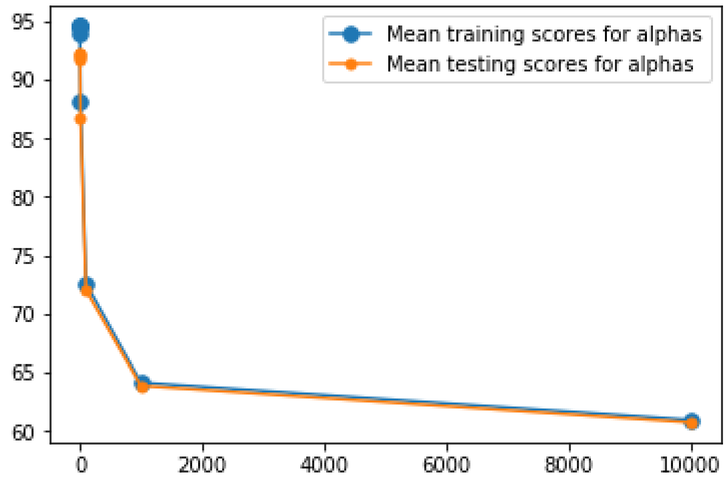
```
mean_test_score
```

Out[36]:

```
[91.78018162909734,
 91.88482701201886,
 92.01666980375109,
 92.23124992729143,
 92.21536357329208,
 86.75565731204598,
 72.02338629838847,
 63.882610225162274,
 60.76195536341158]
```

In [37]:

```
plt.plot(alpha,mean_train_score,marker='.',markersize = 15,label='Mean training scores
 for alphas')
plt.plot(alpha,mean_test_score,marker='o', markersize= 5, label = 'Mean testing scores
 for alphas')
plt.legend(loc='upper right')
plt.show()
```



In [38]:

```
est_opt2 = MultinomialNB(alpha=al2)
est_opt2.fit(tfidf_X_train,y_train)
predict = est_opt2.predict(tfidf_X_test)
```

Measure of effectiveness

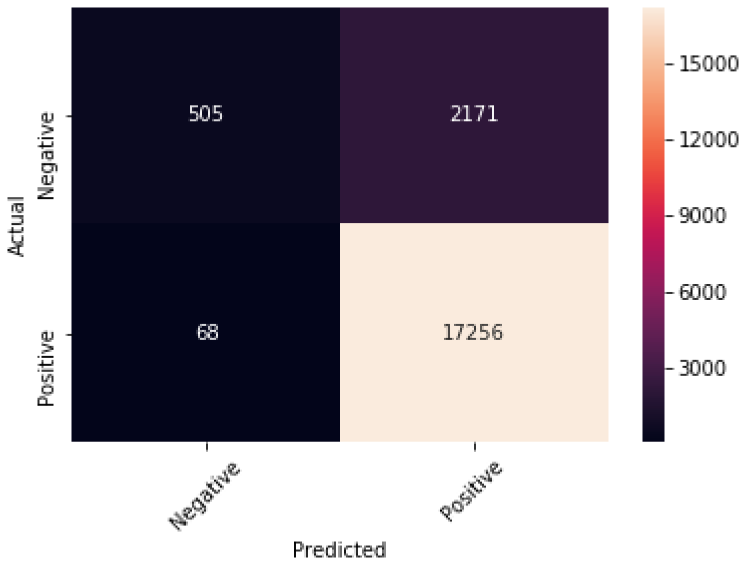
In [39]:

```
test_acc2 = accuracy_score(y_test,predict)*100
precision2 = precision_score(y_test,predict)*100
recall2 = recall_score(y_test,predict)*100
f12 = f1_score(y_test,predict)*100

print('Accuracy on test data:',test_acc2,'%')
print('Precision score:',precision2,'%')
print('Recall score:',recall2,'%')
print('F1 score:',f12,'%')

cm = confusion_matrix(y_test,predict)
print('Confusion Matrix:','\n',cm)
```

```
Accuracy on test data: 88.805 %
Precision score: 88.8248314201884 %
Recall score: 99.60748095128146 %
F1 score: 93.907648771462 %
Confusion Matrix:
 [[  505  2171]
 [   68 17256]]
```

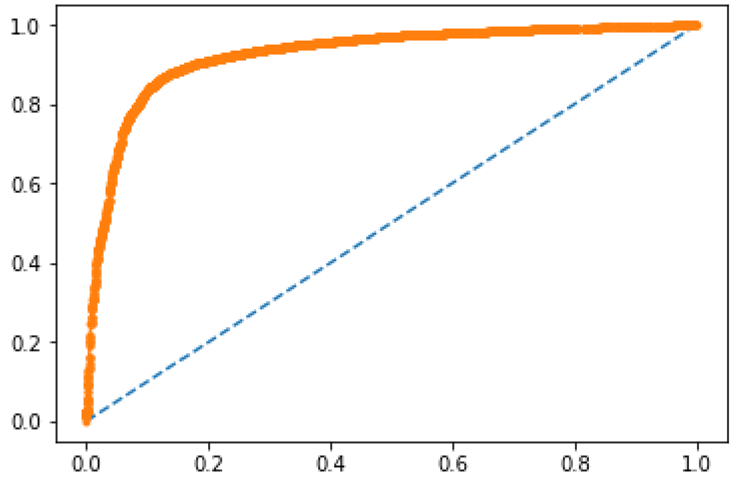
In [40]:

```
cm_df = pd.DataFrame(cm, index = ['Negative','Positive'])
sns.heatmap(cm_df, annot = True,fmt = 'd')
plt.xticks([0.5,1.5],['Negative','Positive'],rotation = 45)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```



In [41]:

```
#plotting Roc curve

y_pred = est_opt2.predict_proba(bow_X_test)[:,1]
fpr,tpr,threshold = roc_curve(y_test,y_pred)
plt.plot([0,1],[0,1],linestyle='--')
plt.plot(fpr,tpr,marker='.')
plt.show()
```



In [42]:

```
roc_score2 = roc_auc_score(y_test,y_pred)
print(roc_score2)
```

```
0.9262057803460227
```

In [43]:

```
neg_imp_word = list(map(abs,est_opt2.feature_log_prob_))[0].argsort()[0:10]
pos_imp_word = list(map(abs,est_opt2.feature_log_prob_))[1].argsort()[0:10]
```

[4.2.1] Top 10 important features of positive class

In [44]:

```
print('Below are the Top 10 positive impacting words:')
pos2 = []
for i in pos_imp_word:
    for j in tfidf_vec.vocabulary_:
        if tfidf_vec.vocabulary_[j] == i:
            pos2.append(j)
print(pos2)
```

```
Below are the Top 10 positive impacting words:
['not', 'great', 'tea', 'good', 'like', 'love', 'product', 'taste', 'coffe
e', 'one']
```

[4.2.2] Top 10 important features of negative class

In [45]:

```
print('Below are the Top 10 negative impacting words:')
neg2 = []
for i in neg_imp_word:
    for j in tfidf_vec.vocabulary_:
        if tfidf_vec.vocabulary_[j] == i:
            neg2.append(j)
print(neg2)
```

```
Below are the Top 10 negative impacting words:
['not', 'like', 'product', 'taste', 'would', 'one', 'no', 'flavor', 'goo
d', 'coffee']
```

Formatting using Pretty table:

In [46]:

```
x = PrettyTable()

model1 = 'NB using BOW'
model2 = 'NB using TFIDF'

x.field_names = ['Model','Alpha','Train Acc(%)','Test Acc(%)','AUC Score','Precision(%)','Recall(%)','F1 score(%)']

train_acc1 = np.around(train_acc1,decimals=2)
train_acc2 = np.around(train_acc2,decimals= 2)

test_acc1 = np.around(test_acc1,decimals=2)
test_acc2 = np.around(test_acc2,decimals=2)

roc_score1 = np.around(roc_score1,decimals=2)
roc_score2 = np.around(roc_score2,decimals=2)

precision1 = np.around(precision1,decimals=2)
precision2 = np.around(precision2,decimals=2)

recall1 = np.around(recall1,decimals=2)
recall2 = np.around(recall2,decimals=2)

f11 = np.around(f11,decimals=2)
f12 = np.around(f12,decimals=2)

x.add_row([model1,al1,train_acc1,test_acc1,roc_score1,precision1,recall1,f11])
x.add_row([model2,al2,train_acc2,test_acc2,roc_score2,precision2,recall2,f12])

print(x)
```

```
+----------------+-------+--------------+-------------+-----------+-------
-------+-----------+-------------+
|     Model      | Alpha | Train Acc(%) | Test Acc(%) | AUC Score | Precis
ion(%) | Recall(%) | F1 score(%) |
+----------------+-------+--------------+-------------+-----------+-------
-------+-----------+-------------+
|  NB using BOW  |   1   |    91.03     |    90.18    |    0.92   |    95.
41     |   93.15   |    94.27    |
| NB using TFIDF |  0.1  |    92.23     |     88.8    |    0.93   |    88.
82     |   99.61   |    93.91    |
+----------------+-------+--------------+-------------+-----------+-------
-------+-----------+-------------+
```

Summary and Inference:

- I have considered 100k data after time based sorting.
- I have done time based splitting with 80:20 ratio for train and test respectively.
- In BOW model, I got optimal alpha as 1 where in TFIDF model the optimal alpha was found as 0.1.
- AUC score for both the models are almost same. F1 score of NB using BOW is higher compared to NB using TFIDF.
- PrettyTable has been used to make a tabular summary of all the metrices for both models.
- However during observation,I found that in above 2 models, top 10 positive and negative words which were impacting the models were semantically similar. The cause which i am guessing is: all these words were mostly repeated in both the classes.

In [ ]:

```

```