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)

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

The dataset is available in two forms

1. .csv file
2. SQLite Database

In order to load the data, We have used the SQLITE dataset as it is easier to query the data and visualise the data efficiently.

Here as we only want to get the global sentiment of the recommendations (positive or negative), we will purposefully ignore all Scores equal to 3. If the score is above 3, then the recommendation wil be set to "positive". Otherwise, it will be set to "negative".

In [2]:
```
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")


import sqlite3
import pandas as pd
import numpy as np
import nltk
import string
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.metrics import roc_curve, auc
from nltk.stem.porter import PorterStemmer

import re
# Tutorial about Python regular expressions: https://pymotw.com/2/re/
import string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer

from gensim.models import Word2Vec
from gensim.models import KeyedVectors
import pickle

#from tqdm import tqdm
import os
```

```
C:\Users\himateja\Anaconda3\lib\site-packages\gensim\utils.py:1209: UserWarnin
g: detected Windows; aliasing chunkize to chunkize_serial
  warnings.warn("detected Windows; aliasing chunkize to chunkize_serial")
```

In [3]:
```
from sklearn import preprocessing
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import TimeSeriesSplit
```

In [4]:
```
# using SQLite Table to read data.
con = sqlite3.connect('database.sqlite')

filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 LIMI
filtered_data_20 = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 L

# Give reviews with Score>3 a positive rating(1), and reviews with a score<3 a ne
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)
filtered_data.head(3)
```

```
Number of data points in our data (100000, 10)
```

Out[4]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominat
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	

In [5]:
```
actualScore_20 = filtered_data_20['Score']
positiveNegative_20 = actualScore_20.map(partition)
filtered_data_20['Score'] = positiveNegative_20
print("Number of data points in our data", filtered_data_20.shape)
filtered_data_20.head(3)
```

```
Number of data points in our data (20000, 10)
```

Out[5]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominat
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	

In [6]:
```
display = pd.read_sql_query("""
SELECT UserId, ProductId, ProfileName, Time, Score, Text, COUNT(*)
FROM Reviews
GROUP BY UserId
HAVING COUNT(*)>1
""", con)
```

In [7]:
```
print(display.shape)
display.head()
```

```
(80668, 7)
```

Out[7]:

	UserId	ProductId	ProfileName	Time	Score	Text	COUNT(*)
0	#oc-R115TNMSPFT9I7	B007Y59HVM	Breyton	1331510400	2	Overall its just OK when considering the price...	2
1	#oc-R11D9D7SHXIJB9	B005HG9ET0	Louis E. Emory "hoppy"	1342396800	5	My wife has recurring extreme muscle spasms, u...	3
2	#oc-R11DNU2NBKQ23Z	B007Y59HVM	Kim Cieszykowski	1348531200	1	This coffee is horrible and unfortunately not ...	2
3	#oc-R11O5J5ZVQE25C	B005HG9ET0	Penguin Chick	1346889600	5	This will be the bottle that you grab from the...	3
4	#oc-R12KPBODL2B5ZD	B007OSBE1U	Christopher P. Presta	1348617600	1	I didnt like this coffee. Instead of telling y...	2

In [8]:
```
display[display['UserId']=='AZY10LLTJ71NX']
```

Out[8]:

	UserId	ProductId	ProfileName	Time	Score	Text	COUNT(*)
80638	AZY10LLTJ71NX	B006P7E5ZI	undertheshrine "undertheshrine"	1334707200	5	I was recommended to try green tea extract to ...	5

In [9]:
```
display['COUNT(*)'].sum()
```

Out[9]:
```
393063
```

[2] Exploratory Data Analysis

[2.1] Data Cleaning: Deduplication

It is observed (as shown in the table below) 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 [10]:
```
display= pd.read_sql_query("""
SELECT *
FROM Reviews
WHERE Score != 3 AND UserId="AR5J8UI46CURR"
ORDER BY ProductID
""", con)
display.head()
```

Out[10]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenomir
0	78445	B000HDL1RQ	AR5J8UI46CURR	Geetha Krishnan	2	
1	138317	B000HDOPYC	AR5J8UI46CURR	Geetha Krishnan	2	
2	138277	B000HDOPYM	AR5J8UI46CURR	Geetha Krishnan	2	
3	73791	B000HDOPZG	AR5J8UI46CURR	Geetha Krishnan	2	
4	155049	B000PAQ75C	AR5J8UI46CURR	Geetha Krishnan	2	

As it can be seen above that same user has multiple reviews with same values for HelpfulnessNumerator, HelpfulnessDenominator, Score, Time, Summary and Text and on doing analysis it was found that

ProductId=B000HDOPZG was Loacker Quadratini Vanilla Wafer Cookies, 8.82-Ounce Packages (Pack of 8)

ProductId=B000HDL1RQ was Loacker Quadratini Lemon Wafer Cookies, 8.82-Ounce Packages (Pack of 8) and so on

It was inferred after analysis that reviews with same parameters other than ProductId belonged to the same product just having different flavour or quantity. Hence in order to reduce redundancy it was decided to eliminate the rows having same parameters.

The method used for the same was that we first sort the data according to ProductId and then just keep the first similar product review and delelte the others. for eg. in the above just the review for ProductId=B000HDL1RQ remains. This method ensures that there is only one representative for

each product and deduplication without sorting would lead to possibility of different representatives still existing for the same product.

In [11]:
```
#Sorting data according to ProductId in ascending order
sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=True, inplac
sorted_data_20=filtered_data_20.sort_values('ProductId', axis=0, ascending=True,
```

In [12]:
```
#Deduplication of entries
final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"},
final.shape
```

Out[12]: (87775, 10)

In [13]:
```
final_20=sorted_data_20.drop_duplicates(subset={"UserId","ProfileName","Time","Te
final_20.shape
```

Out[13]: (19354, 10)

In [14]:
```
#Checking to see how much % of data still remains
(final['Id'].size*1.0)/(filtered_data['Id'].size*1.0)*100
```

Out[14]: 87.775

Observation:- It was also seen that in two rows given below the value of HelpfulnessNumerator is greater than HelpfulnessDenominator which is not practically possible hence these two rows too are removed from calcualtions

In [15]:
```
display= pd.read_sql_query("""
SELECT *
FROM Reviews
WHERE Score != 3 AND Id=44737 OR Id=64422
ORDER BY ProductID
""", con)

display.head()
```

Out[15]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenomir
0	64422	B000MIDROQ	A161DK06JJMCYF	J. E. Stephens "Jeanne"	3	
1	44737	B001EQ55RW	A2V0I904FH7ABY	Ram	3	

In [16]:
```
final=final[final.HelpfulnessNumerator<=final.HelpfulnessDenominator]
final_20=final_20[final_20.HelpfulnessNumerator<=final_20.HelpfulnessDenominator]
```

In [17]:
```
#Before starting the next phase of preprocessing lets see the number of entries l
print(final.shape)

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

```
(87773, 10)
```

Out[17]:
```
1    73592
0    14181
Name: Score, dtype: int64
```

In [18]:
```
#Before starting the next phase of preprocessing lets see the number of entries l
print(final_20.shape)

#How many positive and negative reviews are present in our dataset?
final_20['Score'].value_counts()
```

```
(19354, 10)
```

Out[18]:
```
1    16339
0     3015
Name: Score, dtype: int64
```

[3] Preprocessing

[3.1]. Preprocessing Review Text

Now that we have finished deduplication our data requires some preprocessing before we go on further with analysis and making the prediction model.

Hence in the Preprocessing phase we do the following in the order below:-

1. Begin by removing the html tags
2. Remove any punctuations or limited set of special characters like , or . or # etc.
3. Check if the word is made up of english letters and is not alpha-numeric
4. Check to see if the length of the word is greater than 2 (as it was researched that there is no adjective in 2-letters)
5. Convert the word to lowercase
6. Remove Stopwords
7. Finally Snowball Stemming the word (it was obsereved to be better than Porter Stemming)

After which we collect the words used to describe positive and negative reviews

In [20]:
```
# remove urls from text python: https://stackoverflow.com/a/40823105/4084039
sent_0 = re.sub(r"http\S+", "", sent_0)
sent_1000 = re.sub(r"http\S+", "", sent_1000)
sent_150 = re.sub(r"http\S+", "", sent_1500)
sent_4900 = re.sub(r"http\S+", "", sent_4900)

print(sent_0)
```

```
My dogs loves this chicken but its a product from China, so we wont be buying i
t anymore.  Its very hard to find any chicken products made in the USA but they
are out there, but this one isnt.  Its too bad too because its a good product b
ut I wont take any chances till they know what is going on with the china impor
ts.
```

In [21]:
```
# https://stackoverflow.com/questions/16206380/python-beautifulsoup-how-to-remove
from bs4 import BeautifulSoup

soup = BeautifulSoup(sent_0, 'lxml')
text = soup.get_text()
print(text)
print("="*50)

soup = BeautifulSoup(sent_1000, 'lxml')
text = soup.get_text()
print(text)
print("="*50)

soup = BeautifulSoup(sent_1500, 'lxml')
text = soup.get_text()
print(text)
print("="*50)

soup = BeautifulSoup(sent_4900, 'lxml')
text = soup.get_text()
print(text)
```

```
My dogs loves this chicken but its a product from China, so we wont be buying i
t anymore.  Its very hard to find any chicken products made in the USA but they
are out there, but this one isnt.  Its too bad too because its a good product b
ut I wont take any chances till they know what is going on with the china impor
ts.
==================================================
The Candy Blocks were a nice visual for the Lego Birthday party but the candy h
as little taste to it.  Very little of the 2 lbs that I bought were eaten and I
threw the rest away.  I would not buy the candy again.
==================================================
was way to hot for my blood, took a bite and did a jig  lol
==================================================
My dog LOVES these treats. They tend to have a very strong fish oil smell. So i
f you are afraid of the fishy smell, don't get it. But I think my dog likes it
because of the smell. These treats are really small in size. They are great for
training. You can give your dog several of these without worrying about him ove
r eating. Amazon's price was much more reasonable than any other retailer. You
can buy a 1 pound bag on Amazon for almost the same price as a 6 ounce bag at o
ther retailers. It's definitely worth it to buy a big bag if your dog eats them
a lot.
```

In [22]:
```
# https://stackoverflow.com/a/47091490/4084039
import re

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 [23]:
```
sent_1500 = decontracted(sent_1500)
print(sent_1500)
print("="*50)
```

```
was way to hot for my blood, took a bite and did a jig  lol
==================================================
```

In [24]:
```
#remove words with numbers python: https://stackoverflow.com/a/18082370/4084039
sent_0 = re.sub("\S*\d\S*", "", sent_0).strip()
print(sent_0)
```

```
My dogs loves this chicken but its a product from China, so we wont be buying i
t anymore.  Its very hard to find any chicken products made in the USA but they
are out there, but this one isnt.  Its too bad too because its a good product b
ut I wont take any chances till they know what is going on with the china impor
ts.
```

In [25]:
```
#remove spacial character: https://stackoverflow.com/a/5843547/4084039
sent_1500 = re.sub('[^A-Za-z0-9]+', ' ', sent_1500)
print(sent_1500)
```

```
was way to hot for my blood took a bite and did a jig lol
```

In [27]:
```
# Combining all the above stundents
from tqdm import tqdm
preprocessed_reviews = []
# tqdm is for printing the status bar
for sentance in tqdm(final['Text'].values):
    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 s
    preprocessed_reviews.append(sentance.strip())
```

```
100%|
| 87773/87773 [00:30<00:00, 2916.51it/s]
```

In [28]:
```
# Combining all the above stundents
from tqdm import tqdm
preprocessed_reviews_20 = []
# tqdm is for printing the status bar
for sentance in tqdm(final_20['Text'].values):
    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 s
    preprocessed_reviews_20.append(sentance.strip())
```

```
100%|
| 19354/19354 [00:06<00:00, 2978.14it/s]
```

In [29]:
```
preprocessed_reviews[1500]
```

Out[29]:
```
'way hot blood took bite jig lol'
```

In [30]:
```
score=final.Score
```

In [31]:
```
score_20=final_20.Score
```

[3.2] Preprocessing Review Summary

In [32]:
```
## Similartly you can do preprocessing for review summary also.
```

[4] Featurization

In [33]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(preprocessed_reviews, score, test_
```

[4.1] BAG OF WORDS

In [28]:
```
#BoW
count_vect = CountVectorizer() #in scikit-learn
count_vect.fit(preprocessed_reviews)
print("some feature names ", count_vect.get_feature_names()[:10])
print('='*50)

final_counts = count_vect.transform(preprocessed_reviews)
print("the type of count vectorizer ",type(final_counts))
print("the shape of out text BOW vectorizer ",final_counts.get_shape())
print("the number of unique words ", final_counts.get_shape()[1])
```

```
some feature names  ['aa', 'aaa', 'aaaa', 'aaaaa', 'aaaaaaaaaaa', 'aaaaaaaaaaa
aaaa', 'aaaaaaahhhhhh', 'aaaaaaarrrrrggghhh', 'aaaaaawwwwwwwww', 'aaaaah']
==================================================
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text BOW vectorizer  (87773, 54904)
the number of unique words  54904
```

[4.2] Bi-Grams and n-Grams.

In [34]:
```
#bi-gram, tri-gram and n-gram

#removing stop words like "not" should be avoided before building n-grams
# count_vect = CountVectorizer(ngram_range=(1,2))

count_vect = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
final_bigram_counts = count_vect.fit_transform(x_train)
print("the type of count vectorizer ",type(final_bigram_counts))
print("the shape of out text BOW vectorizer ",final_bigram_counts.get_shape())
print("the number of unique words including both unigrams and bigrams ", final_bi
final_bigram_counts=preprocessing.normalize(final_bigram_counts)
bdata=count_vect.transform(x_test)
bdata=preprocessing.normalize(bdata)
```

```
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text BOW vectorizer  (61441, 5000)
the number of unique words including both unigrams and bigrams  5000
```

[4.3] TF-IDF

In [49]:
```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2), min_df=10)
tdata=tf_idf_vect.fit_transform(x_train)
print("some sample features(unique words in the corpus)",tf_idf_vect.get_feature_
print('='*50)
tdata=preprocessing.normalize(tdata)

tfdata = tf_idf_vect.transform(x_test)
tfdata=preprocessing.normalize(tfdata)
print("the type of count vectorizer ",type(tdata))
print("the shape of out text TFIDF vectorizer ",tdata.get_shape())
print("the number of unique words including both unigrams and bigrams ", tdata.ge
```

```
some sample features(unique words in the corpus) ['aa', 'aafco', 'abandon', 'ab
dominal', 'ability', 'able', 'able add', 'able buy', 'able chew', 'able drink']
==================================================
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text TFIDF vectorizer  (61441, 36016)
the number of unique words including both unigrams and bigrams  36016
```

[4.4] Word2Vec

In [52]:
```
# Train your own Word2Vec model using your own text corpus
i=0
list_of_sentance=[]
for sentance in preprocessed_reviews:
    list_of_sentance.append(sentance.split())
```

In [53]:
```
is_your_ram_gt_16g=False
want_to_use_google_w2v = False
want_to_train_w2v = True

if want_to_train_w2v:
    # min_count = 5 considers only words that occured atleast 5 times
    w2v_model=Word2Vec(list_of_sentance,min_count=5,size=50, workers=4)
    print(w2v_model.wv.most_similar('great'))
    print('='*50)
    print(w2v_model.wv.most_similar('worst'))

elif want_to_use_google_w2v and is_your_ram_gt_16g:
    if os.path.isfile('GoogleNews-vectors-negative300.bin'):
        w2v_model=KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative3
        print(w2v_model.wv.most_similar('great'))
        print(w2v_model.wv.most_similar('worst'))
    else:
        print("you don't have gogole's word2vec file, keep want_to_train_w2v = Tr
```

```
[('fantastic', 0.8374910354614258), ('awesome', 0.8196477890014648), ('good',
0.8192743062973022), ('excellent', 0.8038402795791626), ('wonderful', 0.7861517
667770386), ('terrific', 0.7731014490127563), ('perfect', 0.752355694770813),
('amazing', 0.714547872543335), ('nice', 0.7011174559593201), ('decent', 0.6997
573971748352)]
==================================================
[('greatest', 0.8152581453323364), ('best', 0.7257822751998901), ('tastiest',
0.684012770652771), ('nastiest', 0.6641899943351746), ('disgusting', 0.64001631
73675537), ('nicest', 0.6153778433799744), ('horrible', 0.6099884510040283),
('closest', 0.6031430959701538), ('terrible', 0.602320671081543), ('awful', 0.5
913941860198975)]
```

In [54]:
```
w2v_words = list(w2v_model.wv.vocab)
print("number of words that occured minimum 5 times ",len(w2v_words))
print("sample words ", w2v_words[0:50])
```

```
number of words that occured minimum 5 times  17386
sample words  ['dogs', 'loves', 'chicken', 'product', 'china', 'wont', 'buyin
g', 'anymore', 'hard', 'find', 'products', 'made', 'usa', 'one', 'isnt', 'bad',
'good', 'take', 'chances', 'till', 'know', 'going', 'imports', 'love', 'saw',
'pet', 'store', 'tag', 'attached', 'regarding', 'satisfied', 'safe', 'infestati
on', 'literally', 'everywhere', 'flying', 'around', 'kitchen', 'bought', 'hopin
g', 'least', 'get', 'rid', 'weeks', 'fly', 'stuck', 'squishing', 'buggers', 'su
ccess', 'rate']
```

In [55]:
```
x_1, x_test, y_1, y_test = train_test_split(list_of_sentance, score, test_size=0.
```

[4.4.1] Converting text into vectors using Avg W2V, TFIDF-W2V

[4.4.1.1] Avg W2v

In [56]:
```
# average Word2Vec
# compute average word2vec for each review.
def avg(l):

    sent_vectors = []; # the avg-w2v for each sentence/review is stored in this l
    for sent in tqdm(l): # for each review/sentence
        sent_vec = np.zeros(50) # as word vectors are of zero length 50, you migh
        cnt_words =0; # num of words with a valid vector in the sentence/review
        for word in sent: # for each word in a review/sentence
            if word in w2v_words:
                vec = w2v_model.wv[word]
                sent_vec += vec
                cnt_words += 1
        if cnt_words != 0:
            sent_vec /= cnt_words
        sent_vectors.append(sent_vec)
    return np.matrix(sent_vectors)
```

In [57]:
```
x_1=avg(x_1)
```

```
100%|
| 61441/61441 [01:57<00:00, 524.84it/s]
```

In [58]:
```
np.asarray(x_1).shape
```

Out[58]:
```
(61441, 50)
```

In [59]:
```
x_test=avg(x_test)
```

```
100%|
| 26332/26332 [00:54<00:00, 480.39it/s]
```

In [60]:
```
np.asarray(x_test).shape
```

Out[60]:
```
(26332, 50)
```

In [61]:
```
#changing nan values if any
atrain=np.nan_to_num(x_1)
atest=np.nan_to_num(x_test)
```

[4.4.1.2] TFIDF weighted W2v

In [33]:
```
x_1, x_test, y_1, y_test = train_test_split(list_of_sentance, score, test_size=0.
```

In [34]:
```
# S = ["abc def pqr", "def def def abc", "pqr pqr def"]
model = TfidfVectorizer()
tf_idf_matrix = model.fit_transform(preprocessed_reviews)
# we are converting a dictionary with word as a key, and the idf as a value
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

In [35]:
```
# TF-IDF weighted Word2Vec
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and cell_val = t

def tw2v(l,d,t=tfidf_feat):
    tfidf_sent_vectors = [] # the tfidf-w2v for each sentence/review is stored in
    row=0
    for sent in tqdm(l): # for each review/sentence
        sent_vec = np.zeros(50) # as word vectors are of zero length
        weight_sum =0; # num of words with a valid vector in the sentence/review
        for word in sent: # for each word in a review/sentence
            if word in w2v_words and word in t:
                vec = w2v_model.wv[word]
                tf_idf = tf_idf_matrix[row, t.index(word)]
                # to reduce the computation we are
                # dictionary[word] = idf value of word in whole courpus
                # sent.count(word) = tf valeus of word in this review
                tf_idf = d[word]*(sent.count(word)/len(sent))
                sent_vec += (vec * tf_idf)
                weight_sum += tf_idf
        if weight_sum != 0:
            sent_vec /= weight_sum
        tfidf_sent_vectors.append(sent_vec)
        row += 1
    return np.matrix(tfidf_sent_vectors)
```

In [36]:
```
x_1=tw2v(x_1,dictionary)
```

```
100%|
| 61441/61441 [1:09:52<00:00, 14.65it/s]
```

In [38]:
```
x_test=tw2v(x_test,dictionary)
```

```
100%|
| 26332/26332 [23:31<00:00, 16.34it/s]
```

In [46]:
```
x_1.shape
```

Out[46]: (61441, 50)

In [47]:
```
x_test.shape
```

Out[47]: (26332, 50)

In [48]:
```
# changing 'NaN' to numeric value
train=np.nan_to_num(x_1)
test=np.nan_to_num(x_test)
```

In [35]:
```
#importing needed
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score as roc
import random
from sklearn.model_selection import RandomizedSearchCV
import scipy
```

In [41]:
```
#function with gridsearchcv
def clfg(xtrain,ytrain,xtest,ytest):

    #using the gridsearchcv for finding the best c
    sgd = SGDClassifier(loss="hinge",learning_rate='optimal')
    param_grid = {'alpha':[0.0001,0.001,0.01,0.1,1,10,100,1000]}
    #For time based splitting
    t = TimeSeriesSplit(n_splits=3)

    gsv = GridSearchCV(sgd,param_grid,cv=t,n_jobs=-1,verbose=1,scoring="roc_auc")
    gsv.fit(xtrain,ytrain)
    print("Best HyperParameter: ",gsv.best_params_)
    #assinging best alpha to optimal print("Best Accuracy: %.2f%%"%(gsv.best_scor
    print("best estimator: ",gsv.estimator)
    #print("cv results :",gsv.cv_results_)
    yy1=list(gsv.cv_results_['mean_test_score'])
    yy2=list(gsv.cv_results_['mean_train_score'])
    xx=[0.0001,0.001,0.01,0.1,1,10,100,1000]
    plt.plot(xx,yy1,label='cross validation')
    plt.plot(xx,yy2,label='train')
    plt.legend()
    plt.show()

    oo=gsv.best_params_['alpha']

    sgd=SGDClassifier(alpha=oo)
    sgd.fit(xtrain,ytrain)
    pred=sgd.predict(xtest)
    #different metrics
    acc=accuracy_score(y_test,pred)*100
    print("\nthe accuracy is %.2f%%"%acc)

    re=recall_score(y_test,pred,) * 100
    print("\nthe recall is %.2f%%"%re)

    pre=precision_score(y_test,pred) * 100
    print("the precision is %.2f%%"%pre)

    f1=f1_score(y_test,pred) * 100
    print("the f1 score is %.2f%%"%f1)

    df_cm=pd.DataFrame(confusion_matrix(y_test,pred))
    #sns.set(font_scale=1.4)
    sns.heatmap(df_cm,annot=True,fmt="d")

    fpr0, tpr0, thresholds0 = roc_curve(y_test,pred)

    sns.set()
    plt.figure(figsize=(8,5))
    plt.plot([0,1],[0,1],'k--')
    plt.plot(fpr0, tpr0, color = 'r')
    plt.xlabel('False-Positive Rate')
    plt.ylabel('True-Positive Rate')
    plt.title('ROC curve for with alpha ')
    plt.show()
    print('Area under the ROC curve is ', roc(y_test,pred))
```

In [42]:
```
#function with randomizedsearchcv
def clfr(xtrain,ytrain,xtest,ytest):

    #using the randomizedearchcv for finding the best c
    sgd = SGDClassifier(loss="hinge",learning_rate='optimal')
    param_grid = {'alpha':scipy.stats.norm(0.0001, 0.00001)}
    #For time based splitting
    t = TimeSeriesSplit(n_splits=3)

    rsv = RandomizedSearchCV(sgd,param_grid,cv=t,n_jobs=-1,verbose=1,scoring="roc
    rsv.fit(final_bigram_counts,y_train)
    print("Best HyperParameter: ",rsv.best_params_)
    #assinging best alpha to optimal print("Best Accuracy: %.2f%%"%(gsv.best_scor
    print("best estimator: ",rsv.estimator)


    oo=rsv.best_params_['alpha']

    sgd=SGDClassifier(alpha=oo)
    sgd.fit(xtrain,ytrain)
    pred=sgd.predict(xtest)
    #different metrics
    acc=accuracy_score(y_test,pred)*100
    print("\nthe accuracy is %.2f%%"%acc)

    re=recall_score(y_test,pred,) * 100
    print("\nthe recall is %.2f%%"%re)

    pre=precision_score(y_test,pred) * 100
    print("the precision is %.2f%%"%pre)

    f1=f1_score(y_test,pred) * 100
    print("the f1 score is %.2f%%"%f1)

    df_cm=pd.DataFrame(confusion_matrix(y_test,pred))
    #sns.set(font_scale=1.4)
    sns.heatmap(df_cm,annot=True,fmt="d")
```

Applying SVM

[5.1] Linear SVM

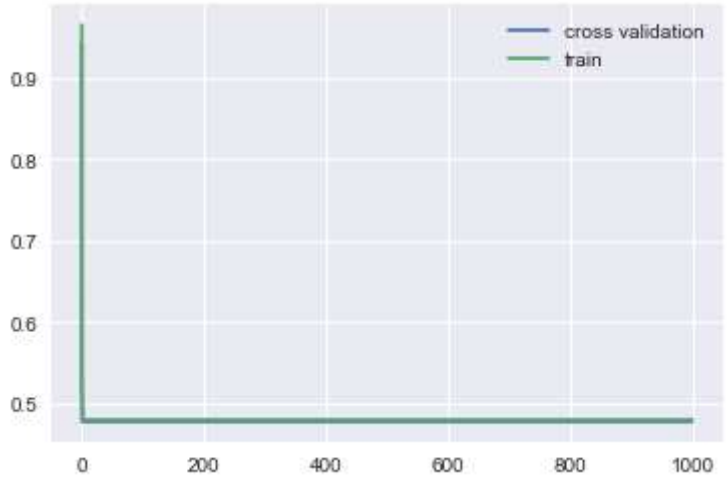
[5.1.1] Applying Linear SVM on BOW

In [43]:
```
# Please write all the code with proper documentation
```

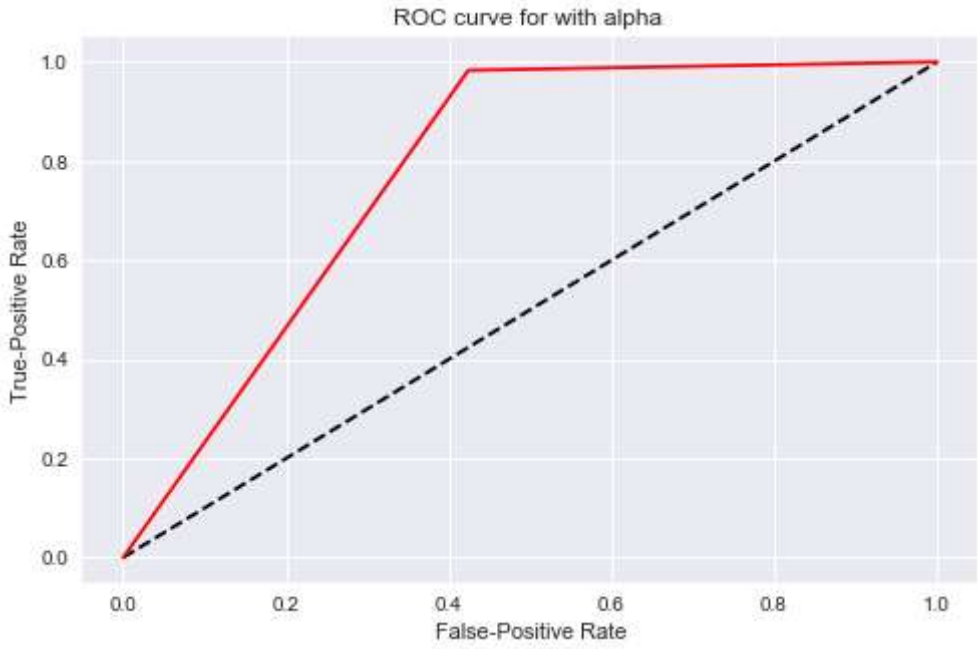
In [44]:
```
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import roc_auc_score as roc
```

In [47]:
```
clfg(final_bigram_counts,y_train,bdata,y_test)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits

[Parallel(n_jobs=-1)]: Done  24 out of  24 | elapsed:    6.1s finished

Best HyperParameter:  {'alpha': 0.0001}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```




```
the accuracy is 91.31%

the recall is 98.31%
the precision is 91.78%
the f1 score is 94.93%
```




```
Area under the ROC curve is  0.7795241155867921
```

In [48]:
```
clfr(final_bigram_counts,y_train,bdata,y_test)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits

[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    5.9s finished

Best HyperParameter:  {'alpha': 8.42035481224191e-05}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)

the accuracy is 91.40%

the recall is 98.30%
the precision is 91.88%
the f1 score is 94.98%
```

Feature importance

In [33]:
```
countvect=CountVectorizer(ngram_range=(1,2),stop_words="english",analyzer='word')
feature_data=countvect.fit_transform(x_train)
all_feat = countvect.get_feature_names()
```

In [37]:
```
clf=SGDClassifier(alpha=0.0001)
clf.fit(feature_data,y_train)
```

Out[37]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```

In [38]:
```
#shape of the array
clf.coef_.shape
```

Out[38]:
```
(1, 1013288)
```

In [39]:
```
#converting the array to one dimensional
check=clf.coef_.ravel()
```

In [40]:
```
#creating the dataframe for features importance
feature_importance=pd.DataFrame({"word":all_feat,"coef_":check})
feature_importance.head(4)
```

Out[40]:

	word	coef_
0	aa	-0.16223
1	aa caffene	0.00000
2	aa coffee	0.00000
3	aa currently	0.00000

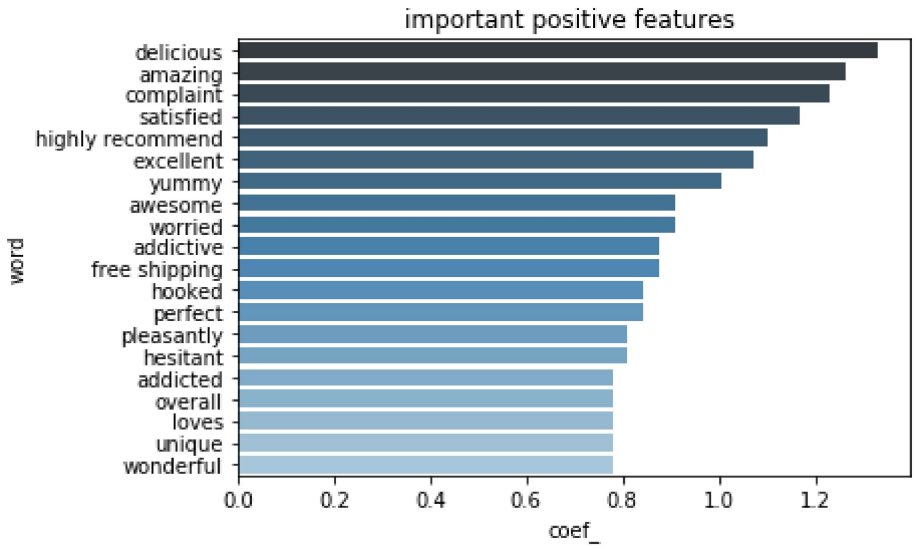
In [41]:
```
#sorting in the descending order
feature_importance.sort_values(by="coef_",ascending=False,inplace=True)
feature_importance.head(10)
```

Out[41]:

	word	coef_
234813	delicious	1.330288
22700	amazing	1.265396
183889	complaint	1.232950
765013	satisfied	1.168058
421422	highly recommend	1.103165
298988	excellent	1.070719
1011497	yummy	1.005827
45019	awesome	0.908489
1001114	worried	0.908489
8402	addictive	0.876043

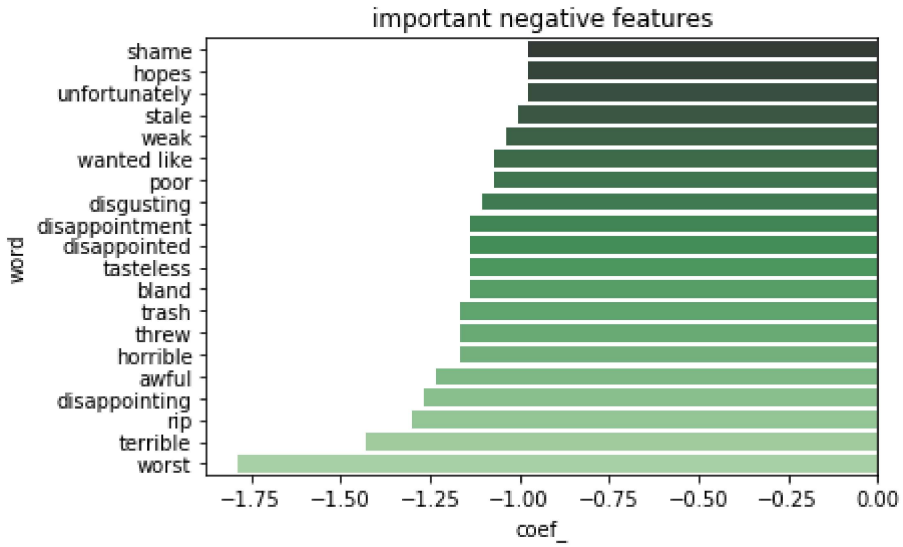
In [42]:
```
#visualising positive features
sns.barplot(y='word', x='coef_', data=feature_importance.head(20), palette="Blues_
```

Out[42]:
```
Text(0.5,1,'important positive features ')
```



In [43]:
```
#visualing the negative features
sns.barplot(y='word', x='coef_', data=feature_importance.tail(20), palette="Green
```

Out[43]: Text(0.5,1,'important negative features')



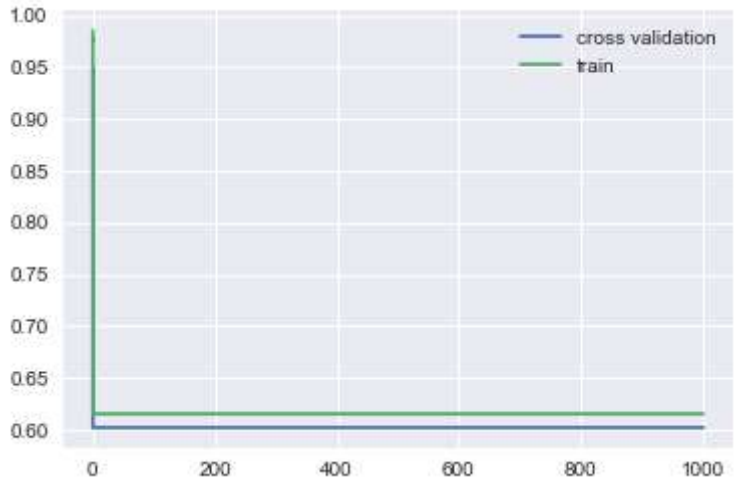
In [ ]:

[5.1.2] Applying Linear SVM on TFIDF

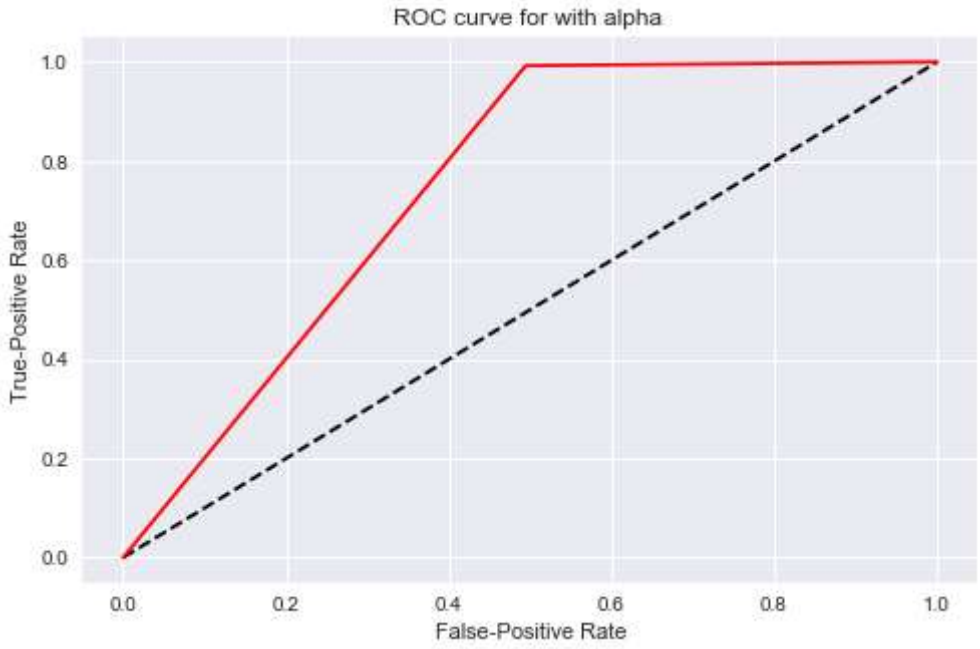
In [29]:
```
# Please write all the code with proper documentation
```

In [50]:
```
%%time
clfg(tdata,y_train,tfdata,y_test)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits

[Parallel(n_jobs=-1)]: Done  24 out of  24 | elapsed:    6.0s finished

Best HyperParameter:  {'alpha': 0.0001}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```




```
the accuracy is 90.87%

the recall is 99.24%
the precision is 90.62%
the f1 score is 94.74%
```




```
Area under the ROC curve is  0.7489768912955965
Wall time: 7.07 s
```

In [51]:
```
%%time
clfr(tdata,y_train,tfdata,y_test)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits

[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    6.0s finished

Best HyperParameter:  {'alpha': 9.085626606737568e-05}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)

the accuracy is 91.24%

the recall is 99.03%
the precision is 91.16%
the f1 score is 94.93%
Wall time: 6.84 s
```

Feature importance

In [44]:
```
tf=TfidfVectorizer(ngram_range=(1,2),stop_words="english",analyzer='word')
feature_data=tf.fit_transform(x_train)
all_feat = tf.get_feature_names()
```

In [45]:
```
clf=SGDClassifier(alpha=0.0001)
clf.fit(feature_data,y_train)
```

Out[45]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```

In [46]:
```
#converting array to 1D
check=clf.coef_.ravel()
```

In [47]:
```
#creating dataframe for feature importance
feature_importance=pd.DataFrame({"word":all_feat,"coef_":check})
feature_importance.head(4)
```

Out[47]:

	word	coef_
0	aa	-0.079120
1	aa caffene	0.000000
2	aa coffee	0.000000
3	aa currently	0.005264

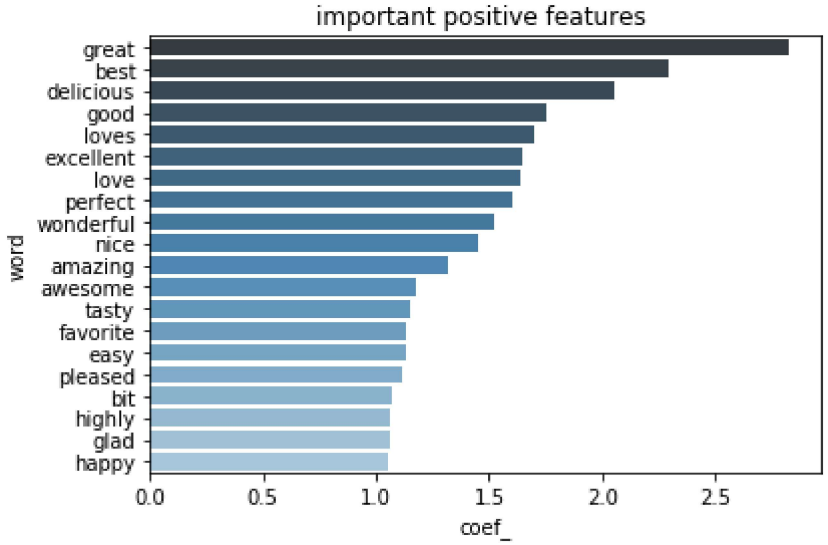
In [48]:
```
#sorting in descending order
feature_importance.sort_values(by="coef_",ascending=False,inplace=True)
feature_importance.head(10)
```

Out[48]:

	word	coef_
389855	great	2.825967
68838	best	2.291503
234813	delicious	2.057964
378813	good	1.750837
523945	loves	1.702213
298988	excellent	1.650491
519807	love	1.641037
644421	perfect	1.598821
995889	wonderful	1.521407
585467	nice	1.455520

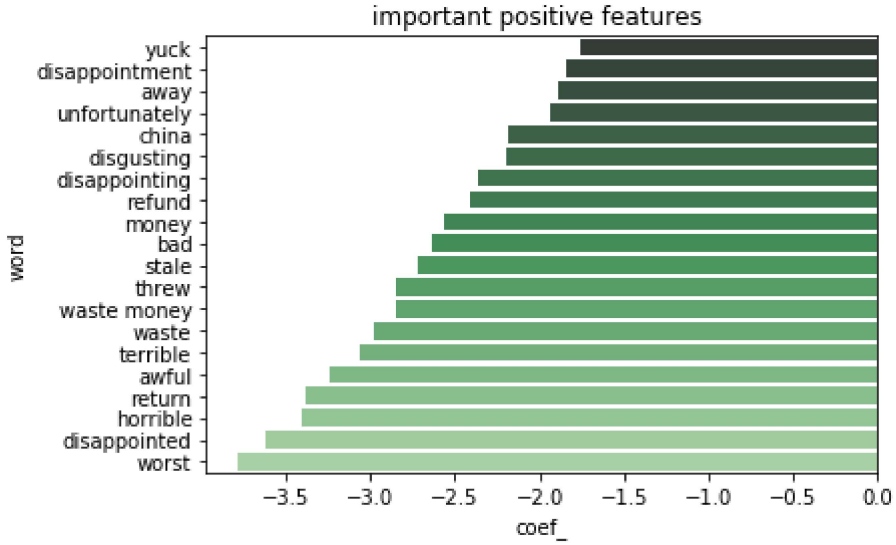
In [49]:
```
#visualising positive features
sns.barplot(y='word', x='coef_', data=feature_importance.head(20), palette="Blues_
```

Out[49]: Text(0.5,1,'important positive features ')



In [50]:
```
#visualising negative features
sns.barplot(y='word', x='coef_', data=feature_importance.tail(20), palette="Green
```

Out[50]: Text(0.5,1,'important positive features ')



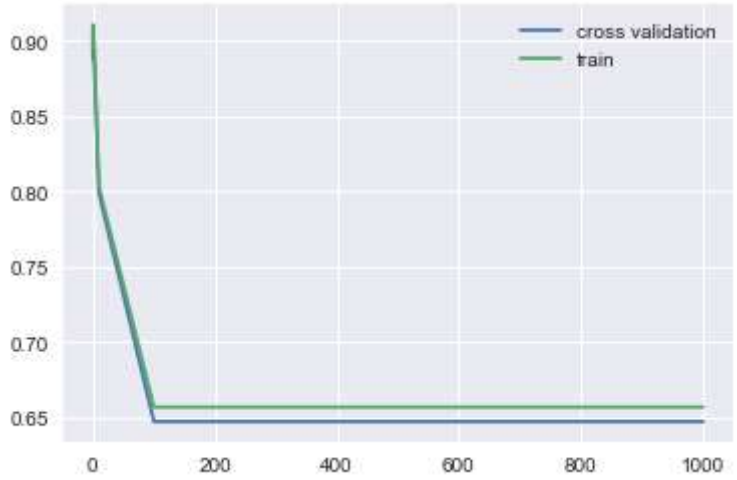
In [ ]:
```

```

[5.1.3] Applying Linear SVM on AVG W2V

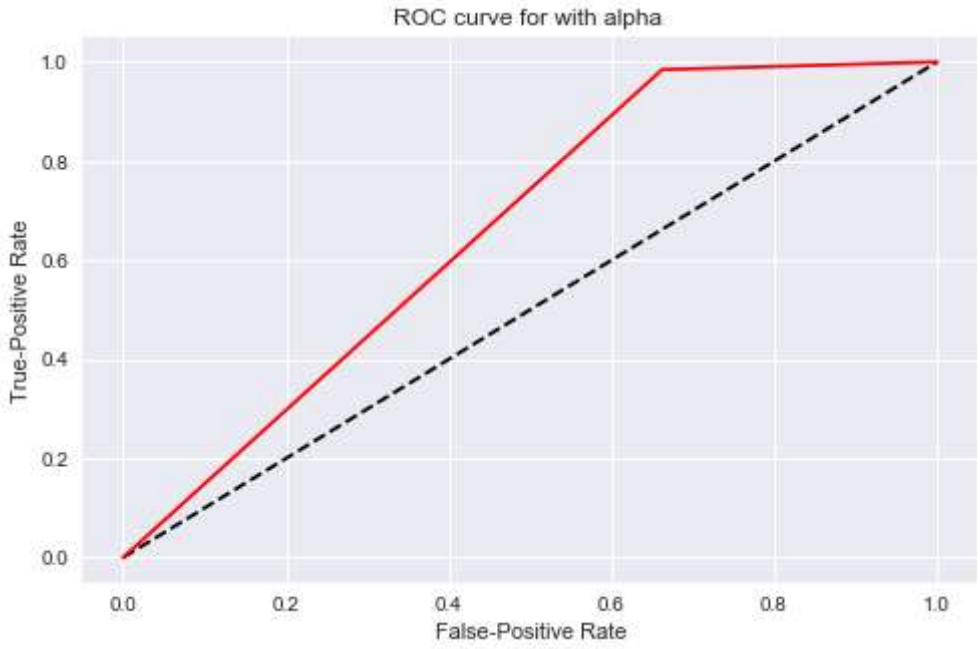
In [3]:
```
# Please write all the code with proper documentation
```

In [62]:
```
%%time
clfg(atrain,y_1,atest,y_test)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits

[Parallel(n_jobs=-1)]: Done  24 out of  24 | elapsed:    6.7s finished

Best HyperParameter:  {'alpha': 0.01}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```




```
the accuracy is 87.32%

the recall is 98.44%
the precision is 87.74%
the f1 score is 92.78%
```




```
Area under the ROC curve is  0.6611290971345745
Wall time: 7.89 s
```

In [63]:
```
%%time
clfr(atrain,y_1,atest,y_test)
```

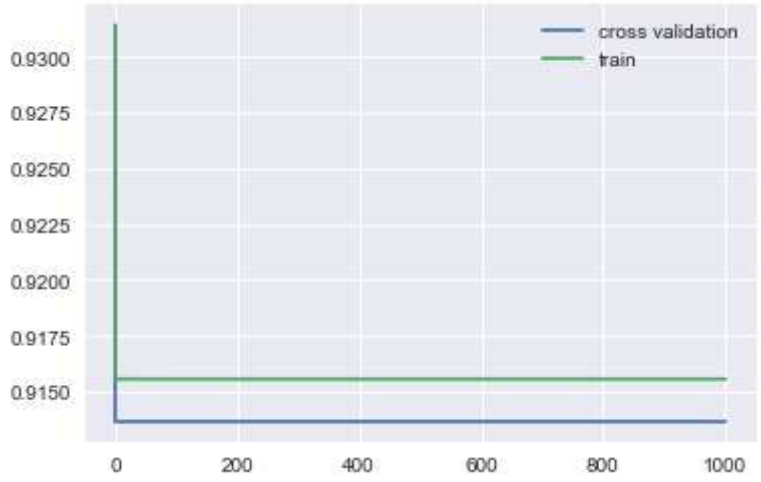
```
Fitting 3 folds for each of 10 candidates, totalling 30 fits

[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    5.9s finished

Best HyperParameter:  {'alpha': 8.45230874211559e-05}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)

the accuracy is 86.80%

the recall is 91.33%
the precision is 92.62%
the f1 score is 91.97%
Wall time: 6.73 s
```

[5.1.4] Applying Linear SVM on TFIDF W2V

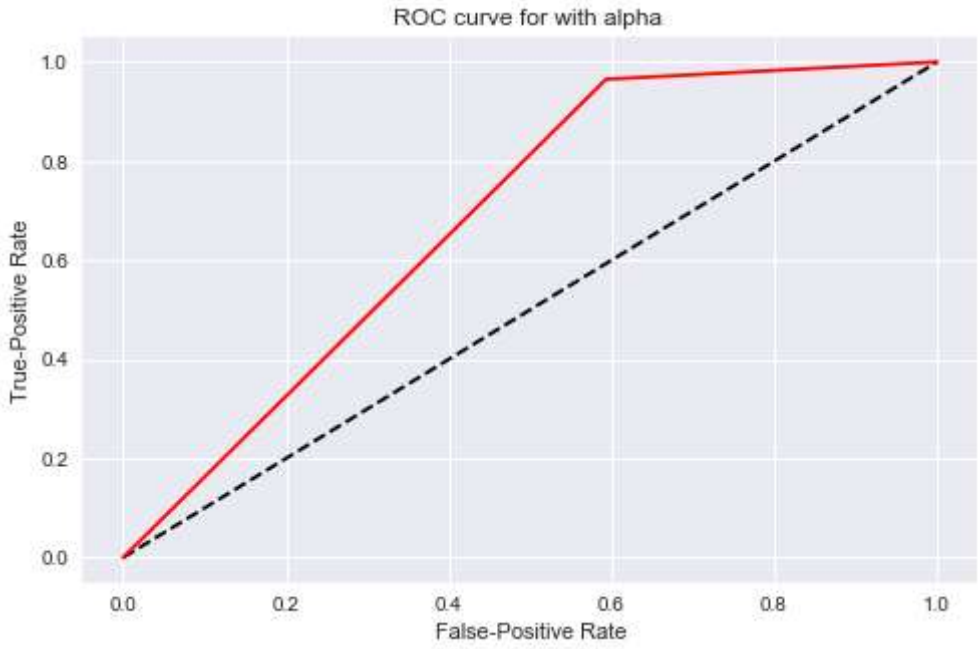
In [3]:
```
# Please write all the code with proper documentation
```

In [57]:
```
%%time
clfg(train,y_1,test,y_test)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits

[Parallel(n_jobs=-1)]: Done  24 out of  24 | elapsed:    6.2s finished

Best HyperParameter:  {'alpha': 0.001}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```



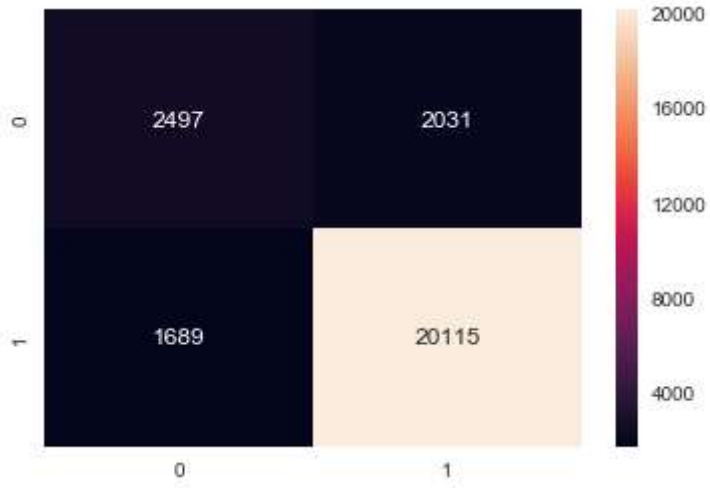

```
the accuracy is 86.89%

the recall is 96.49%
the precision is 88.68%
the f1 score is 92.42%
```




```
Area under the ROC curve is  0.6858584073464006
Wall time: 7.13 s
```

In [54]:
```
%%time
clfr(train,y_1,test,y_test)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits

[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    5.5s finished

Best HyperParameter:  {'alpha': 8.667571504004117e-05}
best estimator:  SGDClassifier(alpha=0.0001, average=False, class_weight=None,
epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)

the accuracy is 85.87%

the recall is 92.25%
the precision is 90.83%
the f1 score is 91.54%
Wall time: 6.19 s
```



In [ ]:
```
```

Featurization for RBF

In [58]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(preprocessed_reviews_20, score_20,
```

BOW

In [79]:
```
count_vect = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
final_bigram_counts = count_vect.fit_transform(x_train)
print("the type of count vectorizer ",type(final_bigram_counts))
print("the shape of out text BOW vectorizer ",final_bigram_counts.get_shape())
print("the number of unique words including both unigrams and bigrams ", final_bi
final_bigram_counts=preprocessing.normalize(final_bigram_counts)
bdata=count_vect.transform(x_test)
bdata=preprocessing.normalize(bdata)
```

```
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text BOW vectorizer  (13547, 5000)
the number of unique words including both unigrams and bigrams  5000
```

TFIDF

In [86]:
```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2), min_df=10)
tdata=tf_idf_vect.fit_transform(x_train)
print("some sample features(unique words in the corpus)",tf_idf_vect.get_feature_
print('='*50)
tdata=preprocessing.normalize(tdata)

tfdata = tf_idf_vect.transform(x_test)
tfdata=preprocessing.normalize(tfdata)
print("the type of count vectorizer ",type(tdata))
print("the shape of out text TFIDF vectorizer ",tdata.get_shape())
print("the number of unique words including both unigrams and bigrams ", tdata.ge
```

```
some sample features(unique words in the corpus) ['ability', 'able', 'able bu
y', 'able eat', 'able find', 'able get', 'able order', 'able purchase', 'able u
se', 'absolute']
==================================================
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text TFIDF vectorizer  (13547, 8350)
the number of unique words including both unigrams and bigrams  8350
```

Word2Vec

In [105]:
```
# Train your own Word2Vec model using your own text corpus
i=0
list_of_sentance=[]
for sentance in preprocessed_reviews_20:
    list_of_sentance.append(sentance.split())
```

In [106]:
```
is_your_ram_gt_16g=False
want_to_use_google_w2v = False
want_to_train_w2v = True

if want_to_train_w2v:
    # min_count = 5 considers only words that occured atleast 5 times
    w2v_model=Word2Vec(list_of_sentance,min_count=5,size=50, workers=4)
    print(w2v_model.wv.most_similar('great'))
    print('='*50)
    print(w2v_model.wv.most_similar('worst'))

elif want_to_use_google_w2v and is_your_ram_gt_16g:
    if os.path.isfile('GoogleNews-vectors-negative300.bin'):
        w2v_model=KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative3
        print(w2v_model.wv.most_similar('great'))
        print(w2v_model.wv.most_similar('worst'))
    else:
        print("you don't have gogole's word2vec file, keep want_to_train_w2v = Tr
```

```
[('awesome', 0.8440372943878174), ('good', 0.810171365737915), ('excellent', 0.
8020244240760803), ('fantastic', 0.7967700958251953), ('wonderful', 0.792942643
1655884), ('amazing', 0.7725126147270203), ('perfect', 0.7068961262702942), ('d
elicious', 0.691575288772583), ('decent', 0.680519163608551), ('quick', 0.67233
34789276123)]
==================================================
[('surpasses', 0.8000553846359253), ('closest', 0.7882041931152344), ('designe
r', 0.7808443307876587), ('best', 0.7718047499656677), ('superior', 0.771187782
2875977), ('disappointing', 0.7683774828910828), ('fanatic', 0.766479253768920
9), ('tastiest', 0.7655558586120605), ('greatest', 0.7589340209960938), ('worl
d', 0.7552046179771423)]
```

In [107]:
```
w2v_words = list(w2v_model.wv.vocab)
print("number of words that occured minimum 5 times ",len(w2v_words))
print("sample words ", w2v_words[0:50])
```

```
number of words that occured minimum 5 times  8370
sample words  ['used', 'fly', 'bait', 'seasons', 'ca', 'not', 'beat', 'great',
'product', 'available', 'traps', 'course', 'total', 'pretty', 'stinky', 'righ
t', 'nearby', 'really', 'good', 'idea', 'final', 'outstanding', 'use', 'car',
'window', 'everybody', 'asks', 'bought', 'made', 'two', 'thumbs', 'received',
'shipment', 'could', 'hardly', 'wait', 'try', 'love', 'call', 'instead', 'stick
ers', 'removed', 'easily', 'daughter', 'designed', 'signs', 'printed', 'revers
e', 'windows', 'beautifully']
```

avg word2vec

In [108]:
```
# average Word2Vec
# compute average word2vec for each review.
def avg(l):

    sent_vectors = []; # the avg-w2v for each sentence/review is stored in this l
    for sent in tqdm(l): # for each review/sentence
        sent_vec = np.zeros(50) # as word vectors are of zero length 50, you migh
        cnt_words =0; # num of words with a valid vector in the sentence/review
        for word in sent: # for each word in a review/sentence
            if word in w2v_words:
                vec = w2v_model.wv[word]
                sent_vec += vec
                cnt_words += 1
        if cnt_words != 0:
            sent_vec /= cnt_words
        sent_vectors.append(sent_vec)
    return np.matrix(sent_vectors)
```

In [109]:
```
x_1, x_test, y_1, y_test = train_test_split(list_of_sentance, score_20, test_size
```

In [110]:
```
x_1=avg(x_1)
```

```
100%|
| 13547/13547 [00:16<00:00, 810.60it/s]
```

In [111]:
```
np.asarray(x_1).shape
```

Out[111]:
```
(13547, 50)
```

In [112]:
```
x_test=avg(x_test)
```

```
100%|
| 5807/5807 [00:07<00:00, 804.05it/s]
```

In [113]:
```
np.asarray(x_test).shape
```

Out[113]:
```
(5807, 50)
```

In [114]:
```
#changing nan values if any
atrain=np.nan_to_num(x_1)
atest=np.nan_to_num(x_test)
```

tfidf word2vec

In [117]:
```
# S = ["abc def pqr", "def def def abc", "pqr pqr def"]
model = TfidfVectorizer()
tf_idf_matrix = model.fit_transform(preprocessed_reviews_20)
# we are converting a dictionary with word as a key, and the idf as a value
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

In [118]:
```
# TF-IDF weighted Word2Vec
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and cell_val = t

def tw2v(l,d,t=tfidf_feat):
    tfidf_sent_vectors = [] # the tfidf-w2v for each sentence/review is stored in
    row=0
    for sent in tqdm(l): # for each review/sentence
        sent_vec = np.zeros(50) # as word vectors are of zero length
        weight_sum =0; # num of words with a valid vector in the sentence/review
        for word in sent: # for each word in a review/sentence
            if word in w2v_words and word in t:
                vec = w2v_model.wv[word]
                tf_idf = tf_idf_matrix[row, t.index(word)]
                # to reduce the computation we are
                # dictionary[word] = idf value of word in whole courpus
                # sent.count(word) = tf valeus of word in this review
                tf_idf = d[word]*(sent.count(word)/len(sent))
                sent_vec += (vec * tf_idf)
                weight_sum += tf_idf
        if weight_sum != 0:
            sent_vec /= weight_sum
        tfidf_sent_vectors.append(sent_vec)
        row += 1
    return np.matrix(tfidf_sent_vectors)
```

In [119]:
```
x_1, x_test, y_1, y_test = train_test_split(list_of_sentance, score_20, test_size
```

In [120]:
```
x_1=tw2v(x_1,dictionary)
```

```
100%|
| 13547/13547 [04:25<00:00, 51.01it/s]
```

In [121]:
```
x_test=tw2v(x_test,dictionary)
```

```
100%|
| 5807/5807 [02:00<00:00, 48.31it/s]
```

In [122]:
```
# changing 'NaN' to numeric value
train=np.nan_to_num(x_1)
test=np.nan_to_num(x_test)
```

In [64]:
```
#funtion finding optimal parameters
```

In [54]:
```
from sklearn.svm import SVC
```

In [83]:
```
def clfsg(xtrain,ytrain,xtest,ytest):

    #using the gridsearchcv for finding the best c
    lr = SVC()
    param_grid = {'C':[0.0001,0.001,0.01,0.1,1,10,100],"gamma":[0.0001,0.001,0.01
    #For time based splitting
    t = TimeSeriesSplit(n_splits=3)

    gsv = GridSearchCV(lr,param_grid,cv=t,n_jobs=-1,verbose=1,scoring="f1")
    gsv.fit(xtrain,ytrain)
    print("Best HyperParameter: ",gsv.best_params_)
    #assinging best alpha to optimal print("Best Accuracy: %.2f%%"%(gsv.best_scor
    print("best estimator: ",gsv.estimator)

    oc=gsv.best_params_['C']
    og=gsv.best_params_['gamma']

    lr=SVC(C=oc,gamma=og)
    lr.fit(xtrain,ytrain)
    pred=lr.predict(xtest)
    #different metrics
    acc=accuracy_score(ytest,pred)*100
    print("\nthe accuracy is %.2f%%"%acc)

    re=recall_score(ytest,pred,) * 100
    print("\nthe recall is %.2f%%"%re)

    pre=precision_score(ytest,pred) * 100
    print("the precision is %.2f%%"%pre)

    f1=f1_score(ytest,pred) * 100
    print("the f1 score is %.2f%%"%f1)

    df_cm=pd.DataFrame(confusion_matrix(ytest,pred))
    #sns.set(font_scale=1.4)
    sns.heatmap(df_cm,annot=True,fmt="d")

    fpr0, tpr0, thresholds0 = roc_curve(ytest,pred)

    sns.set()
    plt.figure(figsize=(8,5))
    plt.plot([0,1],[0,1],'k--')
    plt.plot(fpr0, tpr0, color = 'r')
    plt.xlabel('False-Positive Rate')
    plt.ylabel('True-Positive Rate')
    plt.title('ROC curve for with alpha ')
    plt.show()
    print('Area under the ROC curve is ', roc(ytest,pred))
```

In [ ]:

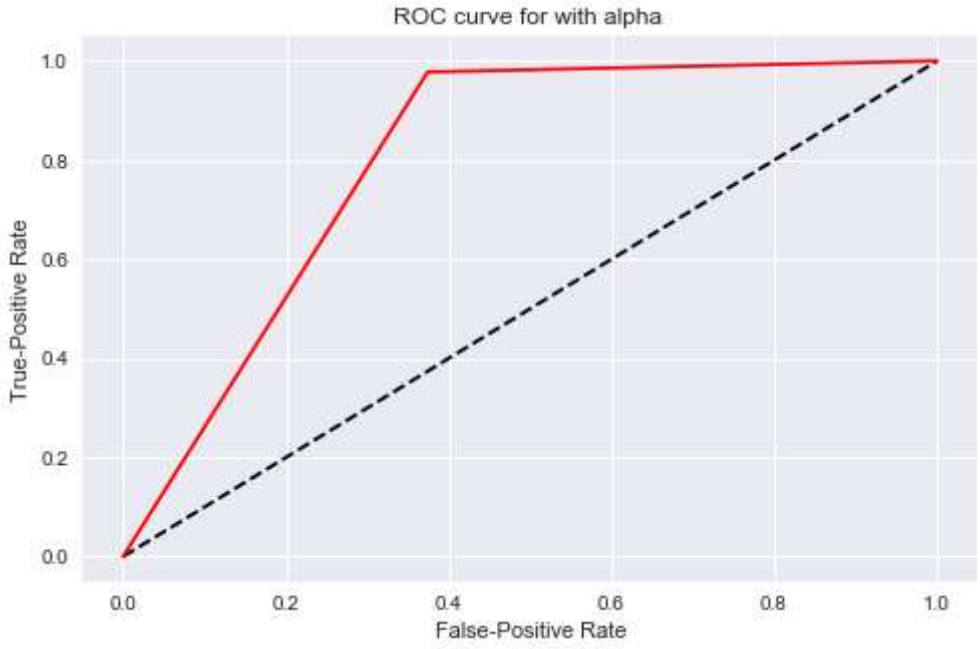
[5.2] RBF SVM

[5.2.1] Applying RBF SVM on BOW

In [84]:
```
# Please write all the code with proper documentation
```

In [85]:
```
clfsg(final_bigram_counts,y_train,bdata,y_test)
```

```
Fitting 3 folds for each of 49 candidates, totalling 147 fits

[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 147 out of 147 | elapsed: 16.3min finished

Best HyperParameter:  {'C': 10, 'gamma': 0.1}
best estimator:  SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

the accuracy is 92.30%

the recall is 97.76%
the precision is 93.43%
the f1 score is 95.55%
```




```
Area under the ROC curve is  0.8017748092552869
```

The model performs well

In [ ]:

```
```

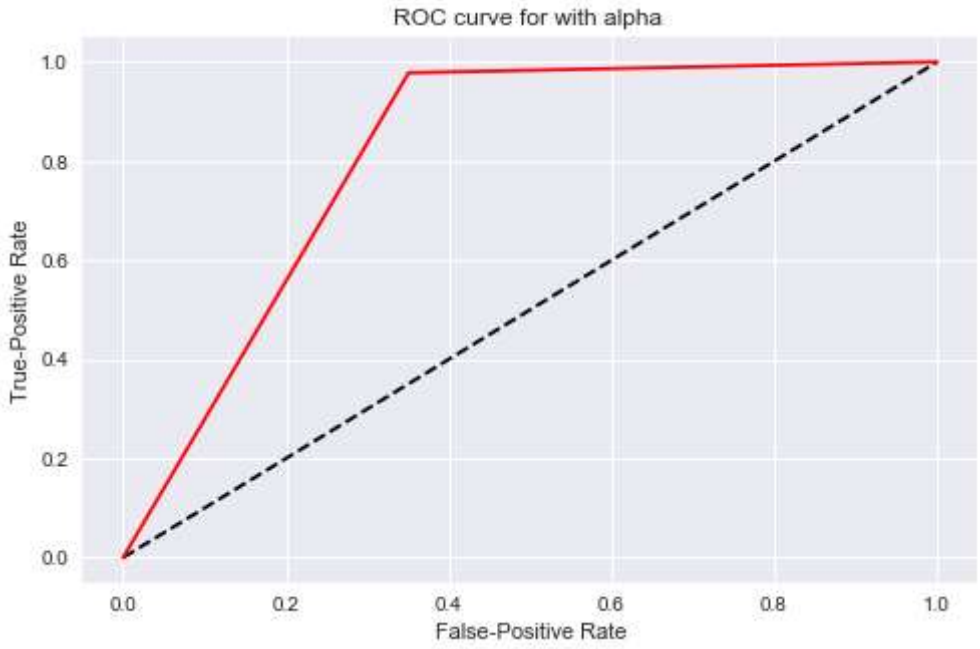
[5.2.2] Applying RBF SVM on TFIDF

In [3]:

```
# Please write all the code with proper documentation
```

In [87]:
```
%%time
clfsg(tdata,y_train,tfdata,y_test)
```

```
Fitting 3 folds for each of 49 candidates, totalling 147 fits

[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  2.0min
[Parallel(n_jobs=-1)]: Done 147 out of 147 | elapsed: 18.7min finished

Best HyperParameter:  {'C': 10, 'gamma': 0.1}
best estimator:  SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

the accuracy is 92.70%

the recall is 97.80%
the precision is 93.82%
the f1 score is 95.77%
```




```
Area under the ROC curve is  0.8136323593886938
Wall time: 19min 35s
```

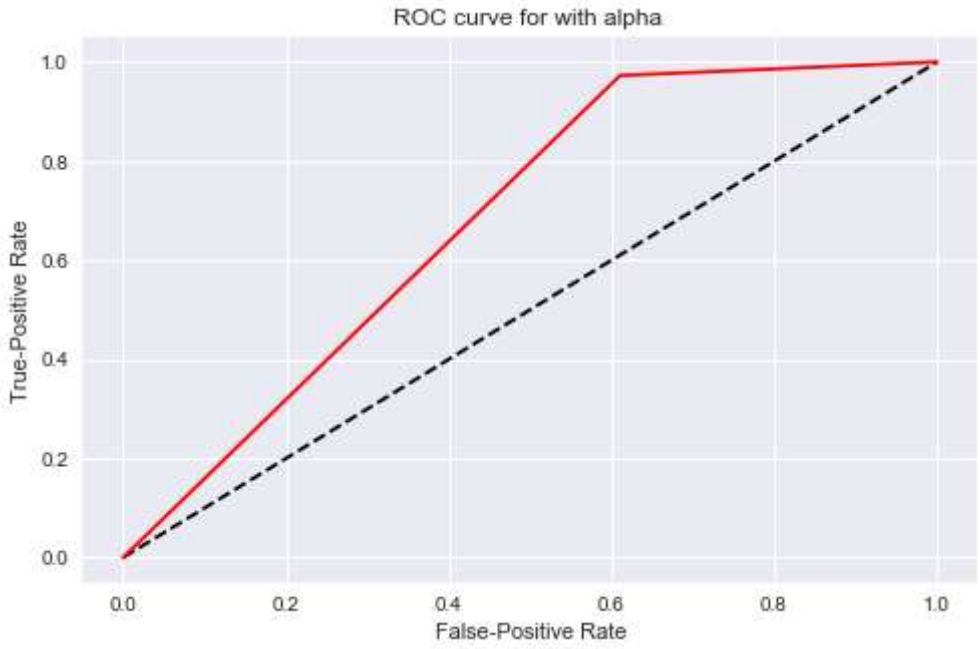
The model has a good performance

[5.2.3] Applying RBF SVM on AVG W2V

In [3]:
```
# Please write all the code with proper documentation
```

In [115]:
```
%%time
clfsg(atrain,y_1,atest,y_test)
```

```
Fitting 3 folds for each of 49 candidates, totalling 147 fits

[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   33.2s
[Parallel(n_jobs=-1)]: Done 147 out of 147 | elapsed:  5.5min finished

Best HyperParameter:  {'C': 10, 'gamma': 0.1}
best estimator:  SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

the accuracy is 88.20%

the recall is 97.25%
the precision is 89.66%
the f1 score is 93.30%
```




```
Area under the ROC curve is  0.6810249109450793
Wall time: 5min 45s
```

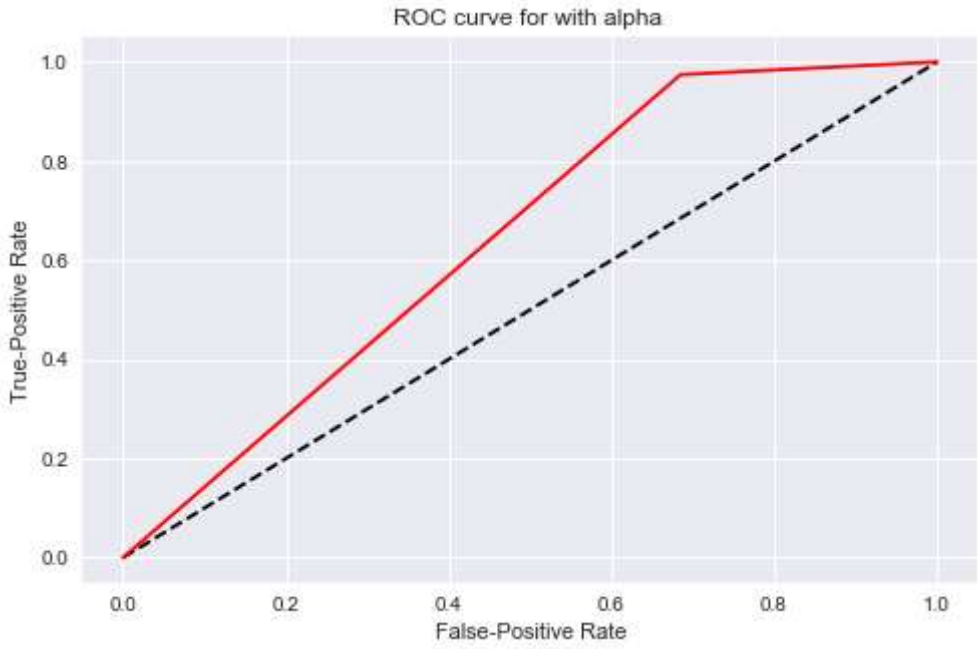
The model comparitively performs less than the BOW and TFIDF

[5.2.4] Applying RBF SVM on TFIDF W2V

In [3]:
```
# Please write all the code with proper documentation
```

In [123]:
```
%%time
clfsg(train,y_1,test,y_test)
```

```
Fitting 3 folds for each of 49 candidates, totalling 147 fits

[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   34.3s
[Parallel(n_jobs=-1)]: Done 147 out of 147 | elapsed:  5.8min finished

Best HyperParameter:  {'C': 10, 'gamma': 0.1}
best estimator:  SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

the accuracy is 87.21%

the recall is 97.43%
the precision is 88.57%
the f1 score is 92.79%
```




```
Area under the ROC curve is  0.6447612450359772
Wall time: 6min 3s
```

The comparitively performs less than all the above 3.

[6] Conclusions

In [4]:
```
# Please compare all your models using Prettytable library
```

In [65]:
```
from prettytable import PrettyTable
```

GridSearchCV results

In [75]:
```
lk=PrettyTable()
```

In [76]:
```
lk.field_names=['feature_set','roc']
```

In [77]:
```
lk.add_row(["BOW",'0.779'])
lk.add_row(["TFIDF",'0.748'])
lk.add_row(["AVG_W2v",'0.661'])
lk.add_row(["TFIDF_w2V",'0.685'])
```

In [78]:
```
print(lk)
```

```
+-------------+-------+
| feature_set |  roc  |
+-------------+-------+
|     BOW     | 0.779 |
|    TFIDF    | 0.748 |
|   AVG_W2v   | 0.661 |
|  TFIDF_w2V  | 0.685 |
+-------------+-------+
```

RandomizedSearchCV results

In [79]:
```
lkr=PrettyTable()
```

In [81]:
```
lkr.field_names=['feature_set','accuracy']
```

In [82]:
```
lkr.add_row(["BOW",'91.4%'])
lkr.add_row(["TFIDF",'91.24%'])
lkr.add_row(["AVG_W2v",'86.80%'])
lkr.add_row(["TFIDF_w2V",'85.87%'])
```

In [83]:
```
print(lkr)
```

```
+-------------+----------+
| feature_set | accuracy |
+-------------+----------+
|     BOW     |  91.4%   |
|    TFIDF    |  91.24%  |
|   AVG_W2v   |  86.80%  |
|  TFIDF_w2V  |  85.87%  |
+-------------+----------+
```

RBF kernel

In [66]:
```
x=PrettyTable()
```

In [67]:
```
x.field_names = ['feature_set','ROC']
```

In [68]:
```
x.add_row(["BOW",'0.801'])
x.add_row(["TFIDF",'0.813'])
x.add_row(["AVG_W2v",'0.681'])
x.add_row(["TFIDF_w2V",'0.644'])
```

In [69]:
```
print(x)
```

```
+-------------+-------+
| feature_set |  ROC  |
+-------------+-------+
|     BOW     | 0.801 |
|    TFIDF    | 0.813 |
|   AVG_W2v   | 0.681 |
|  TFIDF_w2V  | 0.644 |
+-------------+-------+
```

In [ ]:
```

```