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

[1]. Reading Data

Applying Logistic Regression

[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".

Mounting Google Drive locally

In [2]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&scope=email%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdocs.test%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive.photos.readonly%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fpeopleapi.readonly&response_type=code (https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&scope=email%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdocs.test%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive.photos.readonly%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fpeopleapi.readonly&response_type=code)

Enter your authorization code:
··········
Mounted at /content/gdrive
```

In [0]:
```
%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
```

In [4]:
```
# using SQLite Table to read data.
con = sqlite3.connect("/content/gdrive/My Drive/Dataset/database.sqlite")



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

# 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 [0]:
```
display = pd.read_sql_query("""
SELECT UserId, ProductId, ProfileName, Time, Score, Text, COUNT(*)
FROM Reviews
GROUP BY UserId
HAVING COUNT(*)>1
""", con)
```

In [0]:
```
print(display.shape)
display.head()
```

```
(80668, 7)
```

Out[5]:

	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 [0]:
```
display[display['UserId']=='AZY10LLTJ71NX']
```

Out[7]:

	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 [0]:
```
display['COUNT(*)'].sum()
```

Out[8]:
```
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 [0]:
```
display= pd.read_sql_query("""
SELECT *
FROM Reviews
WHERE Score != 3 AND UserId="AR5J8UI46CURR"
ORDER BY ProductID
""", con)
display.head()
```

Out[6]:

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

In [7]:
```
#Deduplication of entries
final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"},
final.shape
```

Out[7]:
```
(87775, 10)
```

In [8]:
```
#Checking to see how much % of data still remains
(final['Id'].size*1.0)/(filtered_data['Id'].size*1.0)*100
```

Out[8]:
```
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 [0]:
```
display= pd.read_sql_query("""
SELECT *
FROM Reviews
WHERE Score != 3 AND Id=44737 OR Id=64422
ORDER BY ProductID
""", con)

display.head()
```

Out[10]:

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

In [9]:
```
final=final[final.HelpfulnessNumerator<=final.HelpfulnessDenominator]
final.shape
```

Out[9]:
```
(87773, 10)
```

In [10]:
```
#Before starting the next phase of preprocessing lets see the number of entries le
print(final.shape)

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

```
(87773, 10)
```

Out[10]:
```
1    73592
0    14181
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 [0]:
```
# 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 [0]:
```
# https://gist.github.com/sebleier/554280
# we are removing the words from the stop words list: 'no', 'nor', 'not'
# <br /><br /> ==> after the above steps, we are getting "br br"
# we are including them into stop words list
# instead of <br /> if we have <br/> these tags would have revmoved in the 1st st

stopwords= set(['br', 'the', 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'our
            "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', '
            'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itsel
            'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that
            'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has
            'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because'
            'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'th
            'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off
            'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all'
            'most', 'other', 'some', 'such', 'only', 'own', 'same', 'so', 'than',
            's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've
            've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "di
            "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma',
            "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn'
            'won', "won't", 'wouldn', "wouldn't"])
```

In [13]:
```
# Combining all the above stundents
from bs4 import BeautifulSoup
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 [00:39<00:00, 2216.35it/s]
```

In [0]:
```
final["CleanText"] = [preprocessed_reviews[i] for i in range(len(final))]
```

In [15]:
```
final.head(2)
```

Out[15]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenon
22620	24750	2734888454	A13ISQV0U9GZIC	Sandikaye	1	
22621	24751	2734888454	A1C298ITT645B6	Hugh G. Pritchard	0	

In [0]:
```
final['totalwords'] = final['Text'].str.split().str.len()
```

In [17]:
```
final.head(2)
```

Out[17]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenon
22620	24750	2734888454	A13ISQV0U9GZIC	Sandikaye	1	
22621	24751	2734888454	A1C298ITT645B6	Hugh G. Pritchard	0	

[4] Featurization

In [0]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

from sklearn.metrics import roc_auc_score
import seaborn as sns

from sklearn.metrics import confusion_matrix

# https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.htm
from sklearn.metrics import roc_curve, auc
```

In [0]:
```
Total_X = final['CleanText'].values
Total_y = final['Score'].values
```

In [0]:
```
# split the data set into train and test
X_train, X_test, y_train, y_test = train_test_split(Total_X, Total_y, test_size=0

# split the train data set into cross validation train and cross validation test
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_size=0.33)
```

In [100]:
```
print(f"Train Data : ({len(X_train)} , {len(y_train)})")
print(f"CV Data : ({len(X_cv)} , {len(y_cv)})")
print(f"Test Data : ({len(X_test)} , {len( y_test)})")
```

```
Train Data : (39400 , 39400)
CV Data : (19407 , 19407)
Test Data : (28966 , 28966)
```

L1 and L2 Regularizer Function

In [0]:
```
def logistic_l1(X_train_reg,X_cv_reg, y_train=y_train,  y_cv=y_cv):
    train_auc = []
    cv_auc = []
    max_C=0
    max_roc_auc=-1
    all_C = [1000,500,100,50,10,5,1,0.5,0.1,0.05,0.01,0.005,0.001,0.0005,0.0001]
    for i in tqdm(all_C):

        clf =  LogisticRegression(penalty='l1',C = i)
        clf.fit(X_train_reg, y_train)
        # roc_auc_score(y_true, y_score) the 2nd parameter should be probability
        # not the predicted outputs

        y_train_pred =  clf.predict_proba(X_train_reg)[:,1]
        y_cv_pred =  clf.predict_proba(X_cv_reg)[:,1]
        #proba1 =roc_auc_score(y_train,y_train_pred) * float(100)
        proba2 = roc_auc_score(y_cv, y_cv_pred) * float(100)

        if(max_roc_auc<proba2):
            max_roc_auc=proba2
            max_C=i
        train_auc.append(roc_auc_score(y_train,y_train_pred))
        cv_auc.append(roc_auc_score(y_cv, y_cv_pred))

    print(f"\nThe 'C' value {max_C} with highest roc_auc Score is {proba2} %" )
    plt.plot(all_C, train_auc, label='Train AUC')
    plt.plot(all_C, cv_auc, label='CV AUC')
    plt.legend()
    plt.xlabel("C: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()
```

In [0]:
```
def logistic_l2(X_train_reg,X_cv_reg, y_train=y_train,  y_cv=y_cv):
    train_auc = []
    cv_auc = []
    max_C=0
    max_roc_auc=-1
    all_C = [1000,500,100,50,10,5,1,0.5,0.1,0.05,0.01,0.005,0.001,0.0005,0.0001]
    for i in tqdm(all_C):

        clf =  LogisticRegression(penalty='l2',C = i)
        clf.fit(X_train_reg, y_train)
        # roc_auc_score(y_true, y_score) the 2nd parameter should be probability 
        # not the predicted outputs

        y_train_pred =  clf.predict_proba(X_train_reg)[:,1]
        y_cv_pred =  clf.predict_proba(X_cv_reg)[:,1]
        #proba1 =roc_auc_score(y_train,y_train_pred) * float(100)
        proba2 = roc_auc_score(y_cv, y_cv_pred) * float(100)

        if(max_roc_auc<proba2):
            max_roc_auc=proba2
            max_C=i
        train_auc.append(roc_auc_score(y_train,y_train_pred))
        cv_auc.append(roc_auc_score(y_cv, y_cv_pred))

    print(f"\nThe 'C' value {max_C} with highest roc_auc Score is {proba2} %" )
    plt.plot(all_C, train_auc, label='Train AUC')
    plt.plot(all_C, cv_auc, label='CV AUC')
    plt.legend()
    plt.xlabel("C: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()
```

Testing the Best C with Test datapoints and Confusion Matrix

In [0]:
```
def testing_l1(X_train_reg,X_test_reg, y_train=y_train,  y_test=y_test):
    clf =  LogisticRegression(penalty='l1',C = max_C)
    clf.fit(X_test_reg, y_test)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probability esti
    # not the predicted outputs

    train_fpr, train_tpr, thresholds = roc_curve(y_train, clf.predict_proba(X_tra
    test_fpr, test_tpr, thresholds = roc_curve(y_test, clf.predict_proba(X_test_r

    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_t
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.legend()
    plt.xlabel("C: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()
    print("\nConfusion Matrix of test set:\n [ [TN  FP]\n   [FN TP] ]\n")
    confusionMatrix=confusion_matrix(y_test, clf.predict(X_test_reg))
    df_cm = pd.DataFrame(confusionMatrix, range(2),range(2))
    plt.figure(figsize = (7,5))
    plt.ylabel("Predicted label")
    plt.xlabel("Actual label")
    plt.title("Confusion Matrix")
    sns.set(font_scale=1.4)#for label size
    sns.heatmap(df_cm, annot=True,annot_kws={"size": 12},fmt="d")
```

In [0]:
```
def testing_l2(X_train_reg,X_test_reg, y_train=y_train,  y_test=y_test):
    clf =  LogisticRegression(penalty='l2',C = max_C)
    clf.fit(X_test_reg, y_test)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probability esti
    # not the predicted outputs

    train_fpr, train_tpr, thresholds = roc_curve(y_train, clf.predict_proba(X_tra
    test_fpr, test_tpr, thresholds = roc_curve(y_test, clf.predict_proba(X_test_r

    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_t
    plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_tpr)))
    plt.legend()
    plt.xlabel("C: hyperparameter")
    plt.ylabel("AUC")
    plt.title("ERROR PLOTS")
    plt.show()
    print("\nConfusion Matrix of test set:\n [ [TN  FP]\n   [FN TP] ]\n")
    confusionMatrix=confusion_matrix(y_test, clf.predict(X_test_reg))
    df_cm = pd.DataFrame(confusionMatrix, range(2),range(2))
    plt.figure(figsize = (7,5))
    plt.ylabel("Predicted label")
    plt.xlabel("Actual label")
    plt.title("Confusion Matrix")
    sns.set(font_scale=1.4)#for label size
    sns.heatmap(df_cm, annot=True,annot_kws={"size": 12},fmt="d")
```

[4.1] BAG OF WORDS

[5.1] Logistic Regression on BOW, SET 1

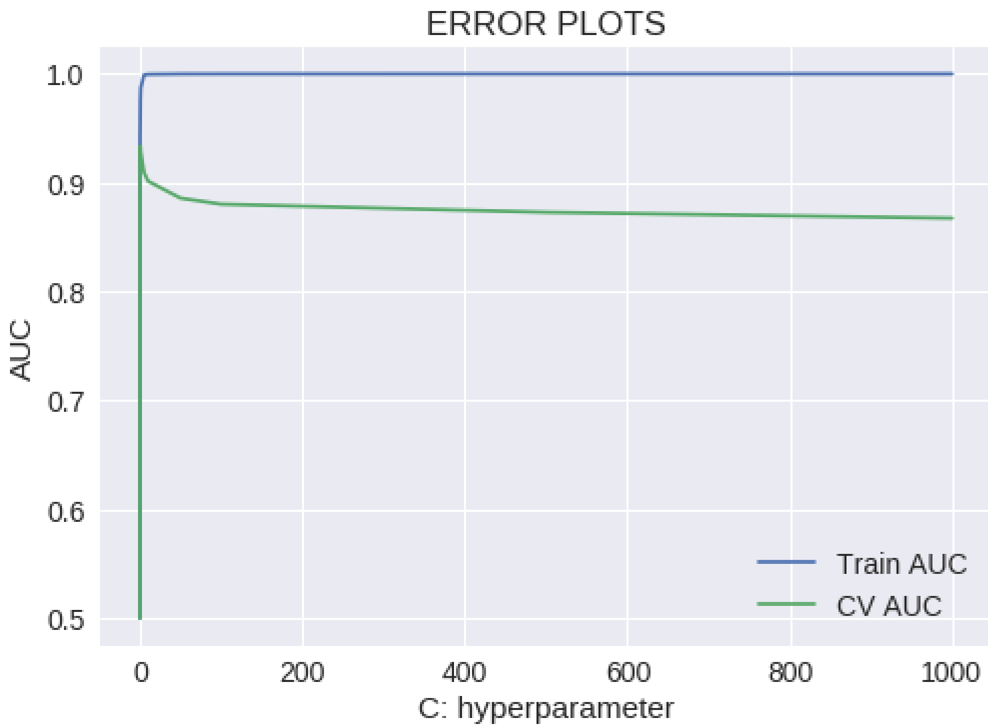
In [102]:
```
#BoW
count_vect = CountVectorizer() #in scikit-learn
count_vect.fit(X_train)
print("some feature names ", count_vect.get_feature_names()[1000:1010])
print('='*50)

# we use the fitted CountVectorizer to convert the text to vector
X_train_bow = count_vect.transform(X_train)
X_cv_bow = count_vect.transform(X_cv)
X_test_bow = count_vect.transform(X_test)


print("After vectorizations")
print(X_train_bow.shape, y_train.shape)
print(X_cv_bow.shape, y_cv.shape)
print(X_test_bow.shape, y_test.shape)
print("="*100)
```

```
some feature names  ['alternataly', 'alternatative', 'alternate', 'alternated',
'alternately', 'alternates', 'alternating', 'alternative', 'alternatively', 'al
ternatives']
==================================================
After vectorizations
(39400, 37210) (39400,)
(19407, 37210) (19407,)
(28966, 37210) (28966,)
================================================================================
====================
```

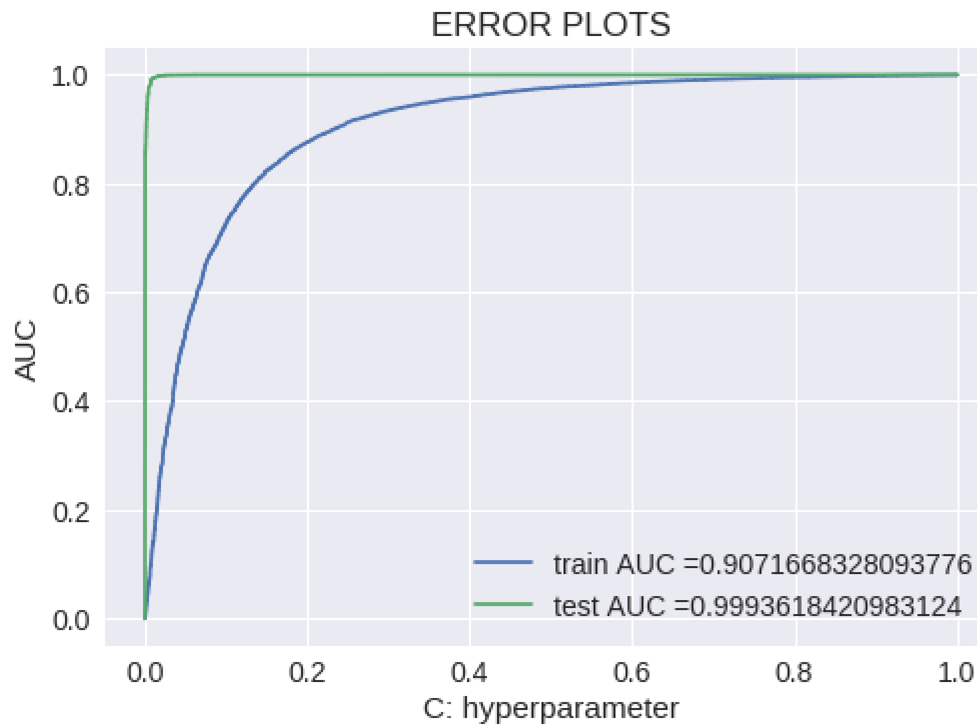
[5.1.1] Applying Logistic Regression with L1 regularization on BOW

In [103]:
```
logistic_l1(X_train_bow,X_cv_bow)
```

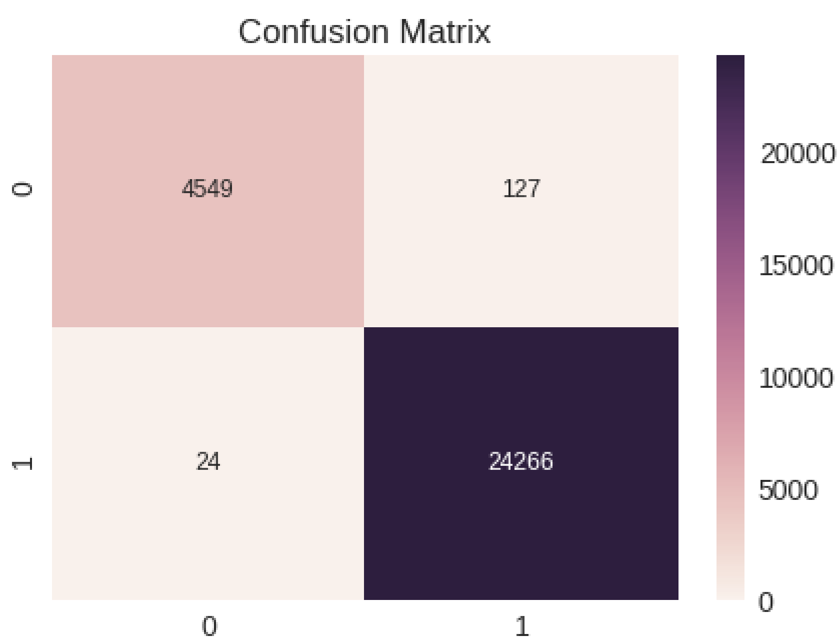
```
100%|██████████| 15/15 [00:44<00:00,  1.78it/s]


The 'C' value 0.5 with highest roc_auc Score is 50.0 %
```



In [110]:
```
testing_l1(X_train_bow,X_test_bow)
```




```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



[5.1.1.1] Calculating sparsity on weight vector obtained using L1 regularization on BOW

In [69]:
```
clf = LogisticRegression(C= 1000, penalty= 'l1')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 88.121%
Non Zero weights: 9157
```

In [24]:
```
clf = LogisticRegression(C= 100, penalty= 'l1')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 88.649%
Non Zero weights: 7483
```

In [25]:
```
clf = LogisticRegression(C= 10, penalty= 'l1')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 89.936%
Non Zero weights: 6538
```

In [26]:
```
clf = LogisticRegression(C= 1, penalty= 'l1')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))

print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 91.459%
Non Zero weights: 3492
```

In [27]:
```
clf = LogisticRegression(C= 0.1, penalty= 'l1')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 90.717%
Non Zero weights: 684
```

In [28]:
```
clf = LogisticRegression(C= 0.01, penalty= 'l1')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 85.528%
Non Zero weights: 79
```

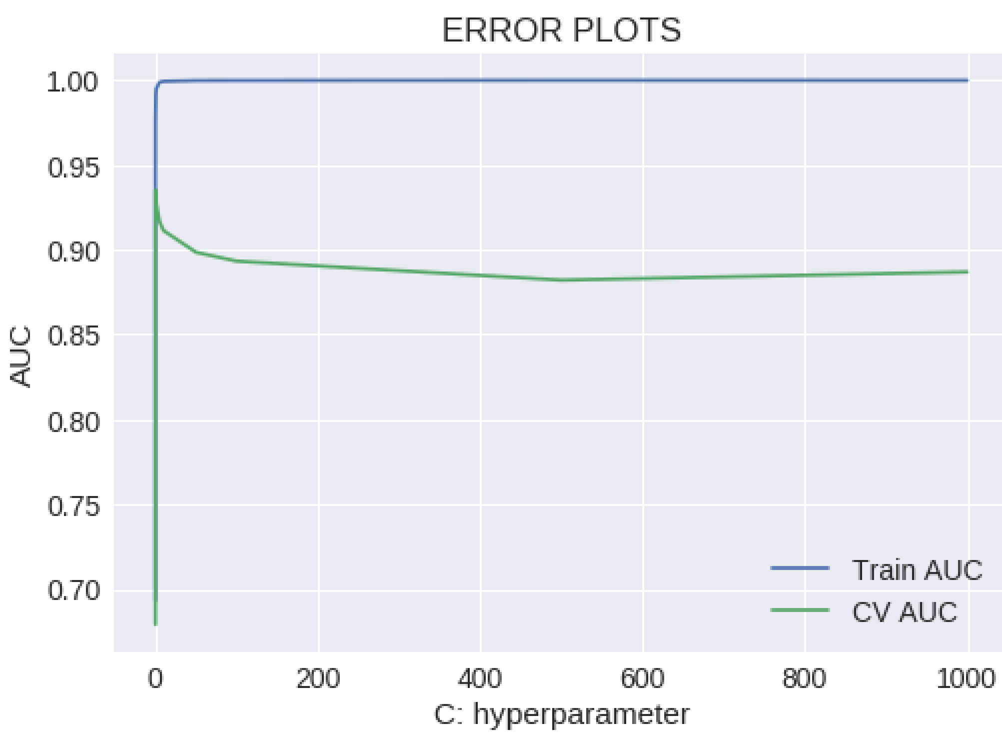
Observation

We can see how drastically the sparsity increases from 9157 non-zero weights at C=1000 to only 79 non-zero weights at C=0.01 when we use L1 Regularization

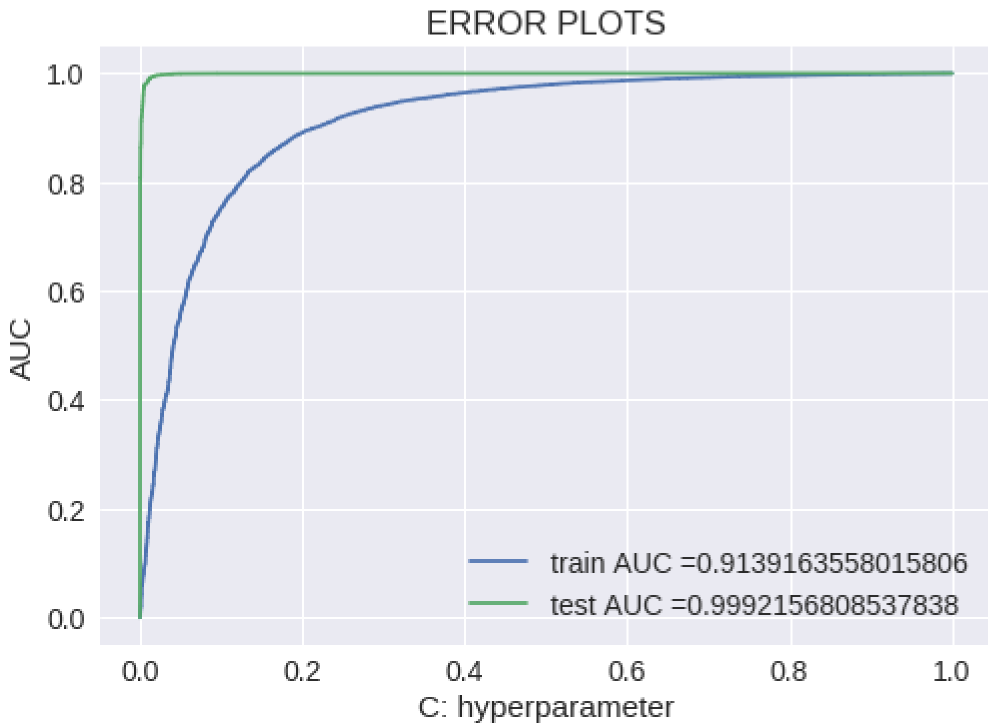
[5.1.2] Applying Logistic Regression with L2 regularization on BOW

In [113]:
```
logistic_l2(X_train_bow,X_cv_bow)
```

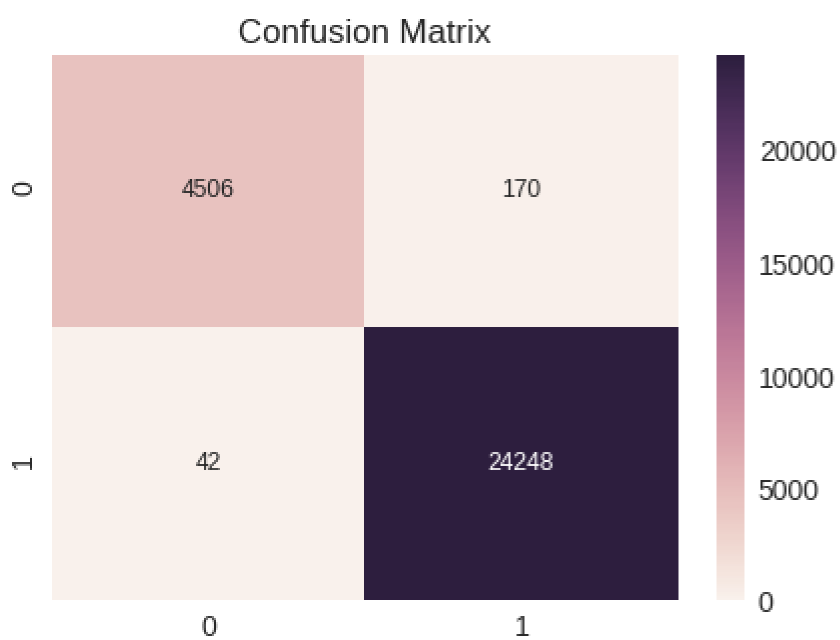
```
100%|██████████| 15/15 [01:32<00:00,  1.12s/it]


The 'C' value 0.1 with highest roc_auc Score is 67.91806791523072 %
```



In [114]:
```
testing_l2(X_train_bow,X_test_bow)
```




```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



Pertubation Testing

In [29]:
```
clf = LogisticRegression(C= 10, penalty= 'l2')
clf.fit(X_train_bow,y_train)
y_pred = clf.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf.coef_))
```

```
Accuracy on test set: 90.586%
Non Zero weights: 37166
```

In [0]:
```
#Weights before adding random noise
weights1 = np.count_nonzero(clf.coef_)
```

In [0]:
```
#Adding some random noise
import copy
train_noise = copy.deepcopy(X_train_bow)
e = np.random.normal(0,0.1)
train_noise.data = train_noise.data + e
```

In [0]:
```
##train_noise = X_train_bow
#Random noise
##epsilon = np.random.uniform(low=-0.0001, high=0.0001, size=(find(train_noise)[0
#Getting the postions(row and column) and value of non-zero datapoints
##a,b,c = find(train_noise)

#Introducing random noise to non-zero datapoints
##train_noise[a,b] = epsilon + train_noise[a,b]
```

In [32]:
```
clf_noise = LogisticRegression(C= 10, penalty= 'l2')
clf_noise.fit(train_noise,y_train)
y_pred = clf_noise.predict(X_test_bow)
print("Accuracy on test set: %0.3f%%"%(accuracy_score(y_test, y_pred)*100))
print("Non Zero weights:",np.count_nonzero(clf_noise.coef_))
```

```
Accuracy on test set: 90.630%
Non Zero weights: 37166
```

In [0]:
```
#Weights after adding random noise
weights_noise = np.count_nonzero(clf_noise.coef_)
```

In [0]:
```
weights_diff = (abs(weights1 - weights_noise)/weights1) * 100
```

In [35]:
```
weights_diff
```

Out[35]:
```
0.0
```

Observation:

As we can see, Before Perturbation test my weights1(W) is 37512 and after adding some noise (i.e 0.1) my weight didn't differ significantly then our feature --> **not co_linear**.

[5.1.1] Top 10 important features of positive and negative class from

SET 1

Reference : [https://stackoverflow.com/questions/11116697/how-to-get-most-informative-features-for-scikit-learn-classifiers (https://stackoverflow.com/questions/11116697/how-to-get-most-informative-features-for-scikit-learn-classifiers)](https://stackoverflow.com/questions/11116697/how-to-get-most-informative-features-for-scikit-learn-classifiers)

In [36]:
```
def important_features(vectorizer, clf, n=10):
    feature_names = vectorizer.get_feature_names()
    coefs_with_fns = sorted(zip(clf.coef_[0], feature_names))
    top = zip(coefs_with_fns[:n], coefs_with_fns[:-(n + 1):-1])
    print("\t\t\tNegative\t\t\t\t\t\tPositive")
    print("________________________________________________________________________
    for (coef_1, fn_1), (coef_2, fn_2) in top:
        print("\t%.4f\t%-15s\t\t\t\t%.4f\t%-15s" % (coef_1, fn_1, coef_2, fn_2))


important_features(count_vect,clf)
```

```
                        Negative
Positive
_______________________________________________________________________________
_________________
        -5.8760 died                                    6.5893  repackage
        -5.8324 ozs                                     4.6500  partmner
        -5.2265 worst                                   4.5656  complaint
        -5.1657 canceled                                4.4368  worried
        -5.1535 revolting                               4.3389  resist
        -5.0925 undamamaged                             4.2750  haha
        -4.8437 sends                                   4.1978  addictive
        -4.6808 pucker                                  4.1176  bright
        -4.5289 multigrain                              4.0689  skeptical
        -4.2369 loading                                 3.9733  satisfied
```

[4.2] Bi-Grams and n-Grams.

In [0]:
```
#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))
# please do read the CountVectorizer documentation http://scikit-learn.org/stable/

# you can choose these numebrs min_df=10, max_features=5000, of your choice
count_vect = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
final_bigram_counts = count_vect.fit_transform(preprocessed_reviews)
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
```

```
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text BOW vectorizer  (4986, 3144)
the number of unique words including both unigrams and bigrams  3144
```

[4.3] TF-IDF

[5.2] Logistic Regression on TFIDF, SET 2

[5.2.1] Applying Logistic Regression with L1 regularization on TFIDF

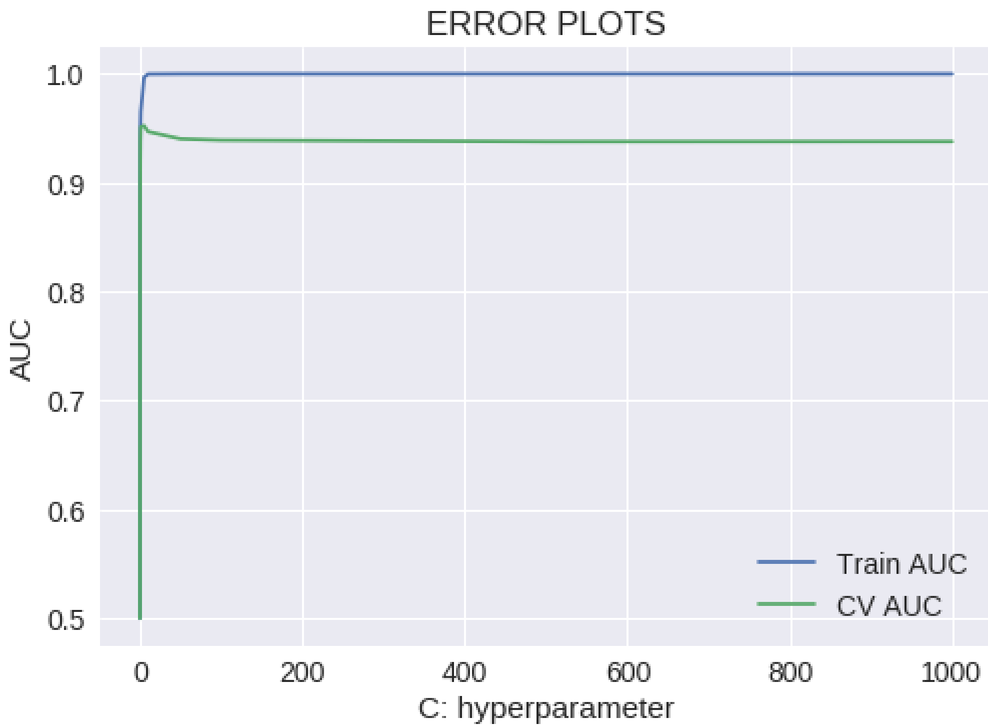
In [115]:

```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2), min_df=10)
tf_idf_vect.fit(X_train)
print("some sample features(unique words in the corpus)",tf_idf_vect.get_feature_
print('='*50)


# we use the fitted CountVectorizer to convert the text to vector
X_train_tf_idf = tf_idf_vect.transform(X_train)
X_cv_tf_idf = tf_idf_vect.transform(X_cv)
X_test_tf_idf = tf_idf_vect.transform(X_test)


print("After vectorizations")
print(X_train_tf_idf.shape, y_train.shape)
print(X_cv_tf_idf.shape, y_cv.shape)
print(X_test_tf_idf.shape, y_test.shape)
print("="*100)
```

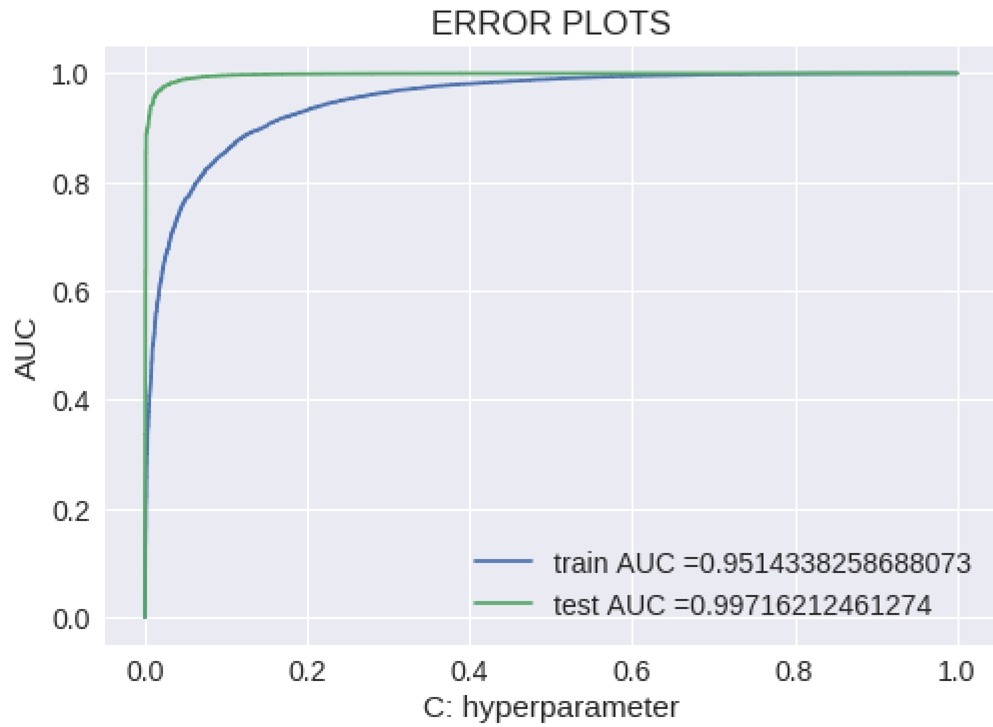
```
some sample features(unique words in the corpus) ['ability', 'able', 'able bu
y', 'able drink', 'able eat', 'able enjoy', 'able find', 'able get', 'able giv
e', 'able make']
==================================================
After vectorizations
(39400, 23476) (39400,)
(19407, 23476) (19407,)
(28966, 23476) (28966,)
================================================================================
====================
```

In [116]:
```
logistic_l1(X_train_tf_idf, X_cv_tf_idf)
```

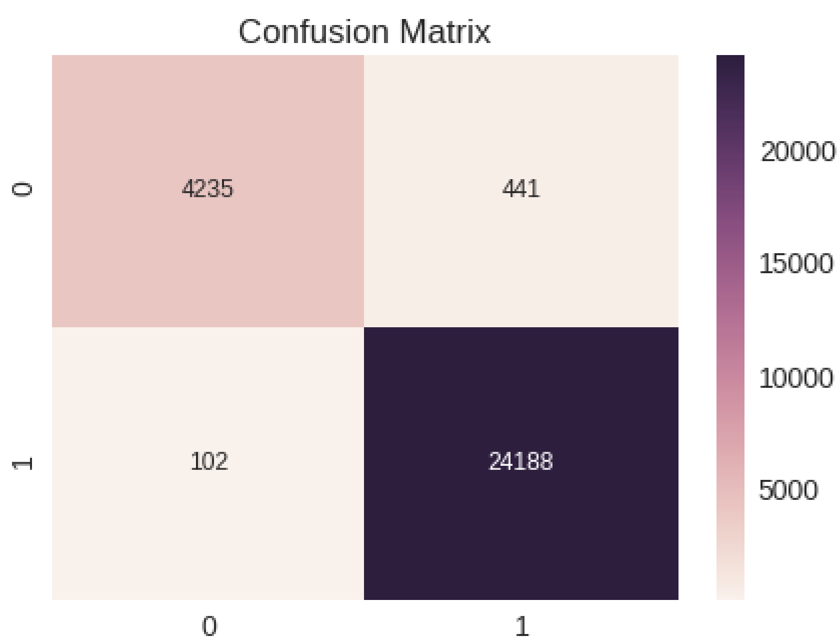
```
100%|██████████| 15/15 [00:12<00:00,  2.97it/s]


The 'C' value 5 with highest roc_auc Score is 50.0 %
```



In [117]:
```
testing_l1(X_train_tf_idf, X_test_tf_idf)
```



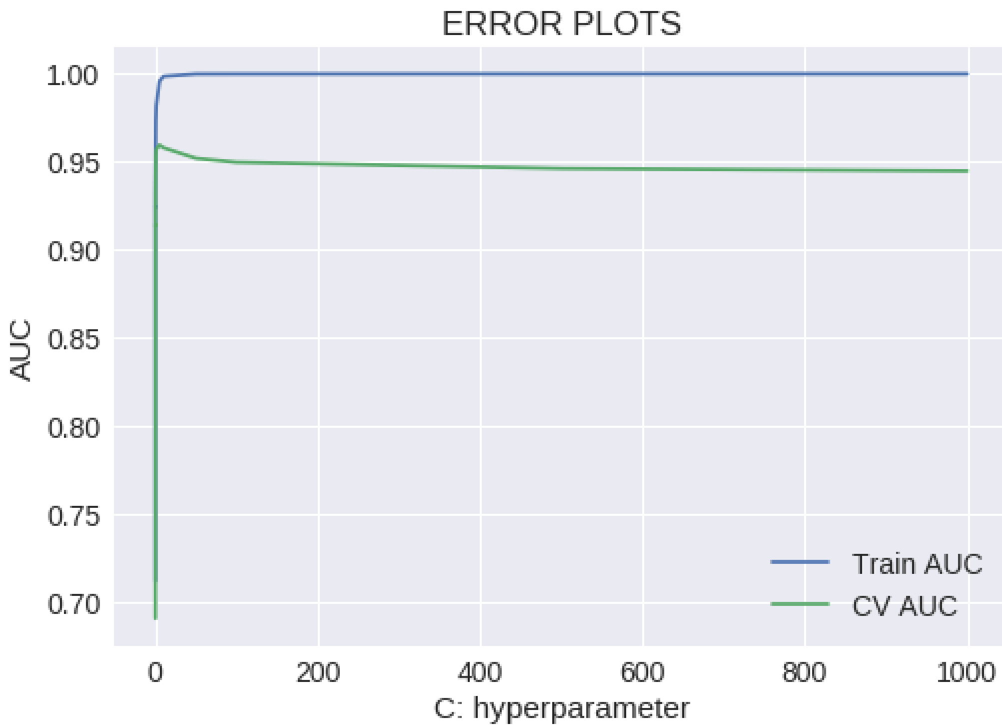

```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



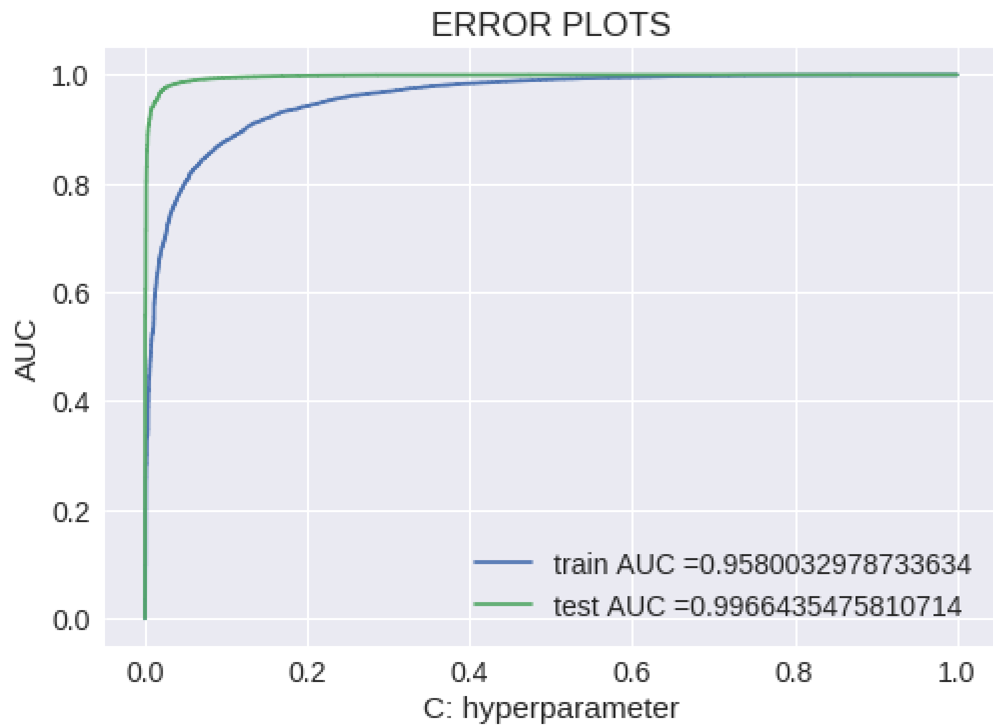
[5.2.2] Applying Logistic Regression with L2 regularization on TFIDF,

In [118]:
```
logistic_l2(X_train_tf_idf, X_cv_tf_idf)
```

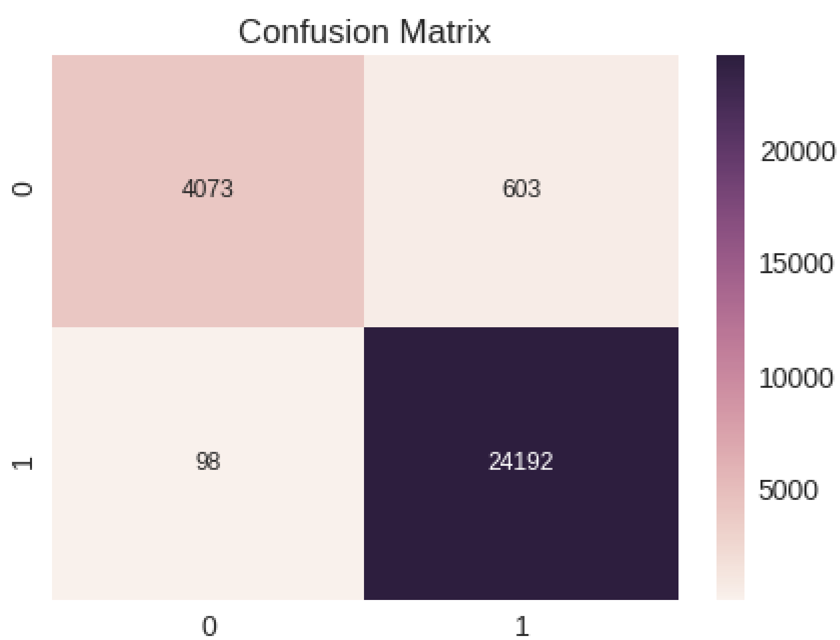
```
100%|██████████| 15/15 [00:13<00:00,  2.88it/s]


The 'C' value 5 with highest roc_auc Score is 69.09296885597479 %
```



In [119]:
```
testing_l2(X_train_tf_idf, X_test_tf_idf)
```




```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



[5.2.2] Top 10 important features of Positive and Negative class from SET 2

In [0]:
```
important_features(tf_idf_vect,clf)
```

```
                        Negative
Positive

______________________________________________________________________________

_________________
        -10.6751        disappointed                            13.5841 great

        -10.1761        worst                                   11.3394 delicio
us
        -9.4964 not worth                               9.9795  best
        -9.1345 terrible                                9.4711  perfect
        -8.8966 not good                                8.7920  good
        -8.3154 disappointing                           8.5621  not disappointe
d
        -8.1905 not                                     8.3348  loves
        -7.9029 awful                                   8.2665  excellent
        -7.7972 unfortunately                           7.9957  wonderful
        -7.5983 not recommend                           7.7493  nice
```

[4.4] Word2Vec

In [0]:
```
i=0

w2v_train=[]
w2v_cv=[]
w2v_test=[]

for sentance in X_train:
    w2v_train.append(sentance.split())

for sentance in X_cv:
    w2v_cv.append(sentance.split())


for sentance in X_test:
    w2v_test.append(sentance.split())
```

In [121]:
```
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)
    w2v_model_train = Word2Vec(w2v_train,min_count=5,size=50, workers=4)
    print(w2v_model_train.wv.most_similar('great'))
    print('='*50)
else:
    pass
```

```
[('awesome', 0.8368321657180786), ('wonderful', 0.8265479207038879), ('fantasti
c', 0.8049709796905518), ('terrific', 0.8012307286262512), ('good', 0.793755531
3110352), ('excellent', 0.7647097706794739), ('perfect', 0.7548678517341614),
('amazing', 0.7389508485794067), ('decent', 0.7073011994361877), ('fabulous',
0.6987826824188232)]
==================================================
```

In [122]:
```
w2v_words_train = list(w2v_model_train.wv.vocab)

print("number of words that occured minimum 5 times ",len(w2v_words_train ))
print("sample words ", w2v_words_train[0:50])
```

```
number of words that occured minimum 5 times  11958
sample words  ['product', 'fishy', 'smell', 'first', 'open', 'package', 'rins
e', 'gone', 'noodles', 'take', 'flavor', 'whatever', 'cook', 'tried', 'olive',
'oil', 'different', 'spices', 'turned', 'great', 'texture', 'regular', 'pasta',
'not', 'bad', 'way', 'overall', 'really', 'liked', 'ordering', 'soon', 'ordere
d', 'disposakups', 'came', 'fast', 'advertised', 'definitely', 'thanks', 'alo
t', 'organic', 'mac', 'n', 'cheese', 'best', 'far', 'could', 'bit', 'cheesier',
'flavorful', 'though']
```

#Converting text into vectors using Avg W2V, TFIDF-W2V

[5.1.3] Applying Logistic Regression on AVG W2V, SET 3

[4.4.1.1] Avg W2v

In [123]:
```
train_vectors = []; # the avg-w2v for each sentence/review is stored in this list
for sent in tqdm(w2v_train): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might nee
    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_train:
            vec = w2v_model_train.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    train_vectors.append(sent_vec)
print()
print(len(train_vectors))
print(len(train_vectors[0]))
```

```
100%|██████████| 39400/39400 [01:12<00:00, 541.59it/s]


39400
50
```

In [125]:
```
# compute average word2vec for each review.
cv_vectors = [] # the avg-w2v for each sentence/review is stored in this list
for sent in tqdm(w2v_cv): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might ne
    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_train:
            vec = w2v_model_train.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    cv_vectors.append(sent_vec)
print()
print(len(cv_vectors))
print(len(cv_vectors[0]))
```

```
100%|██████████| 19407/19407 [00:37<00:00, 523.86it/s]


19407
50
```

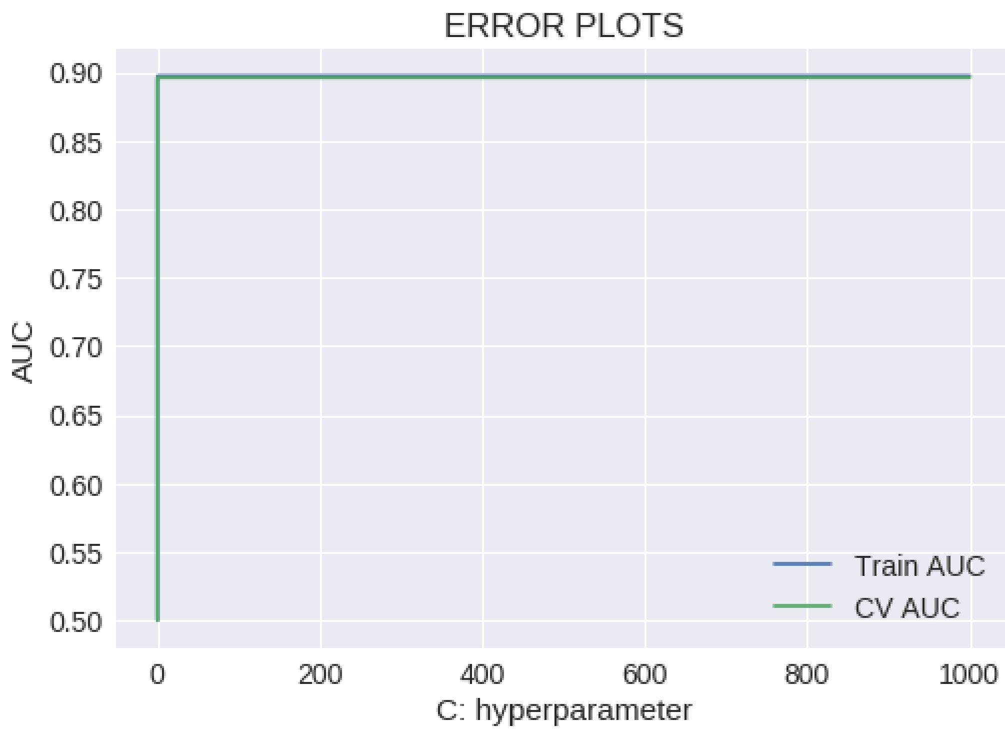
In [124]:
```
test_vectors = []; # the avg-w2v for each sentence/review is stored in this list
for sent in tqdm(w2v_test): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might ne
    cnt_words =0; # num of words with a valid vector in the sentence/review
    for word in sent: # for each word in a review/sentence
        if word in w2v_words_train:
            vec =  w2v_model_train.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    test_vectors.append(sent_vec)
print()
print(len(test_vectors))
print(len(test_vectors[0]))
```

```
100%|██████████| 28966/28966 [00:54<00:00, 534.34it/s]


28966
50
```

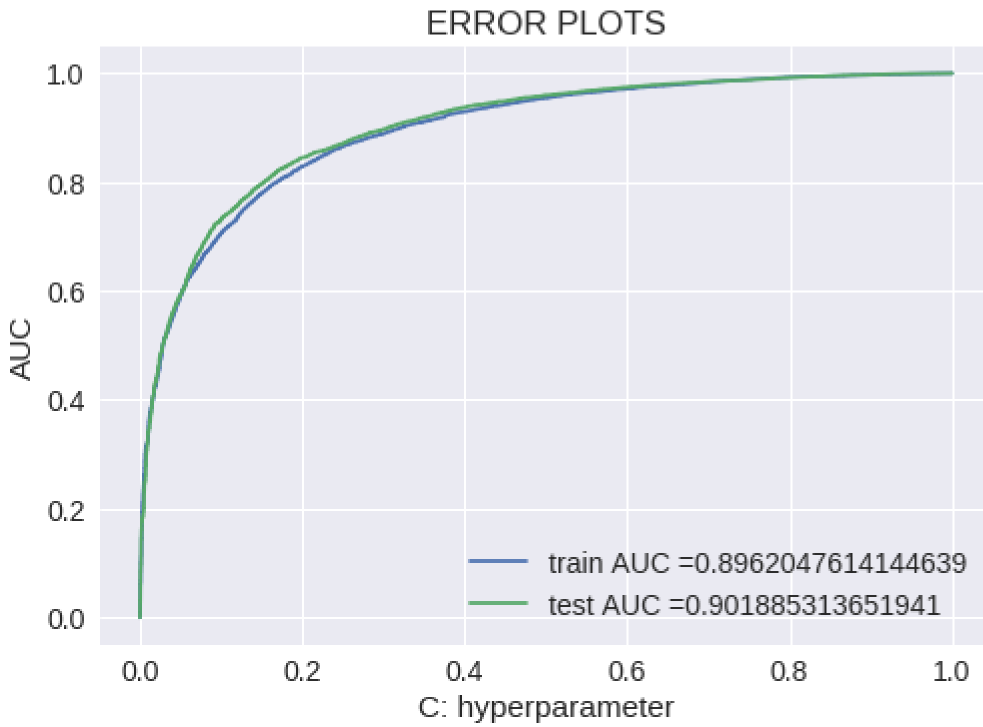
[5.3.1] Applying Logistic Regression with L1 regularization on AVG W2V

In [129]:
```
logistic_l1(train_vectors, cv_vectors)
```

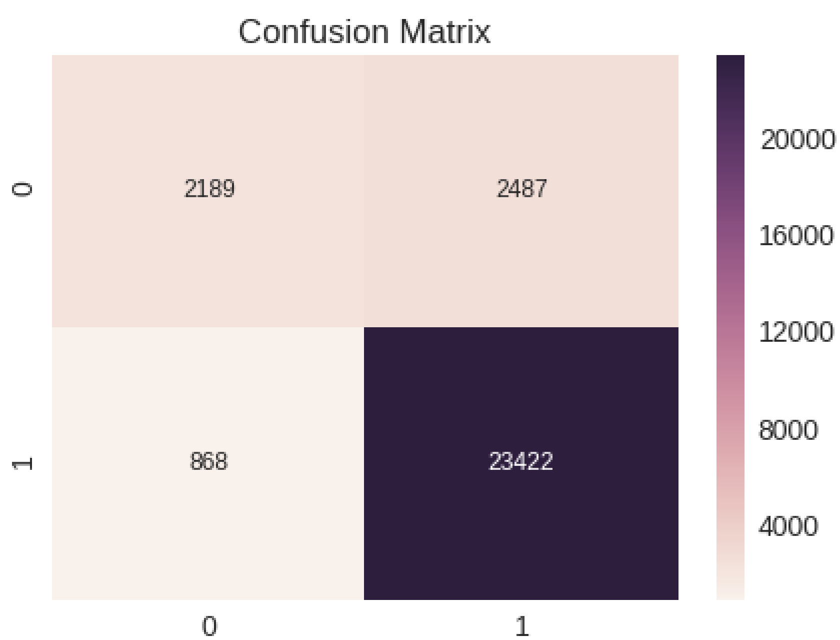
```
100%|██████████| 15/15 [00:42<00:00,  1.03s/it]

The 'C' value 500 with highest roc_auc Score is 50.0 %
```



In [130]:
```
testing_l1(train_vectors, test_vectors)
```



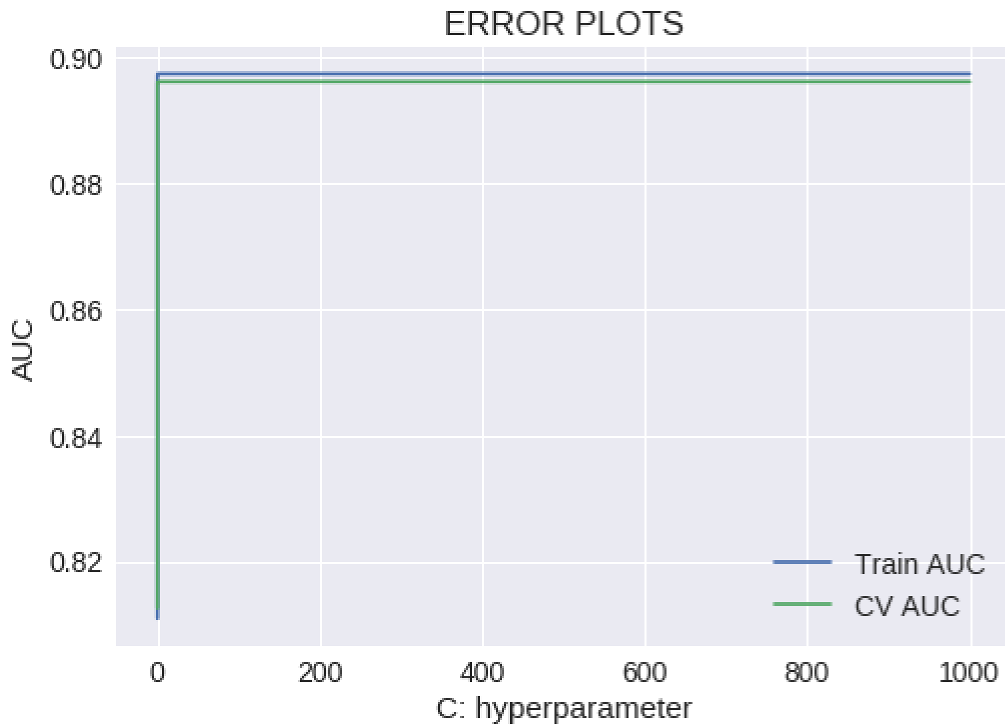

```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



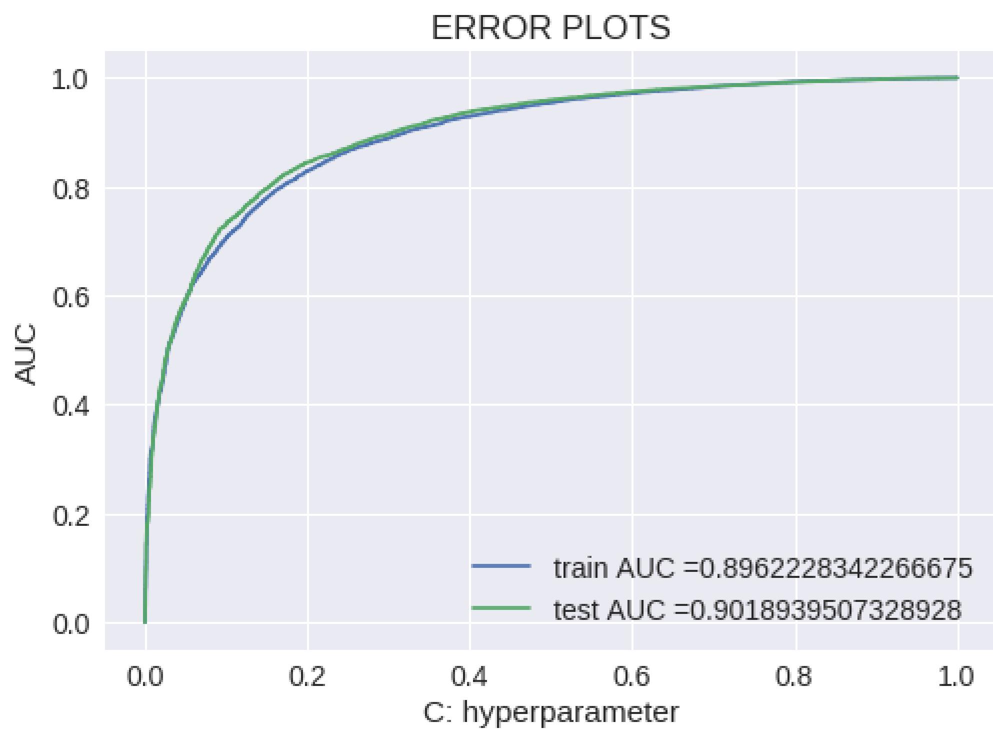
[5.3.2] Applying Logistic Regression with L2 regularization on AVG W2V

In [131]:
```
logistic_l2(train_vectors, cv_vectors)
```

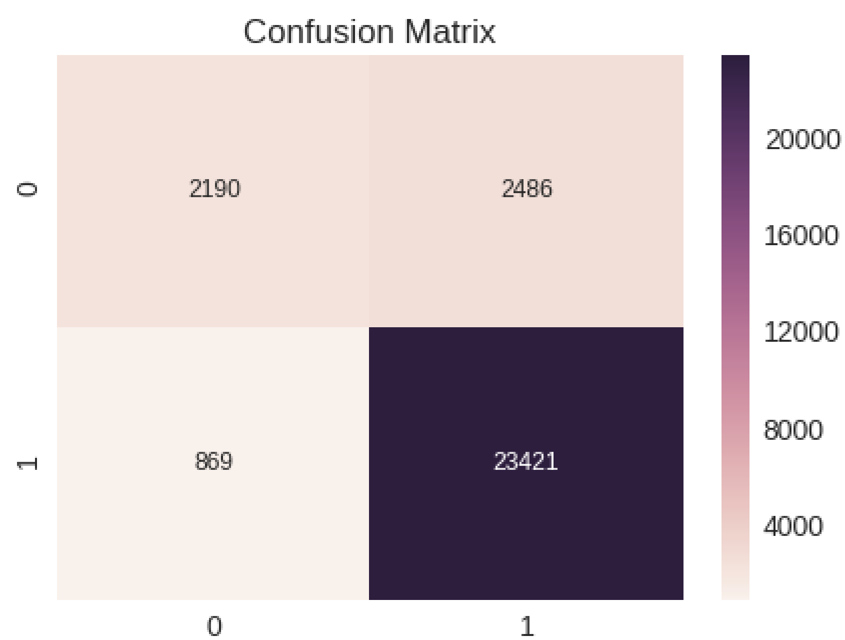
```
100%|██████████| 15/15 [00:11<00:00,  1.96it/s]

The 'C' value 1 with highest roc_auc Score is 81.25339241026228 %
```



In [132]:
```
testing_l2(train_vectors, test_vectors)
```




```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



[5.4] Logistic Regression on TFIDF W2V

In [0]:
```
model = TfidfVectorizer()
tf_idf_matrix = model.fit_transform(X_train)
# 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 [134]:
```
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

train_tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored
row=0;
for sent in tqdm(w2v_train): # 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_train and word in tfidf_feat:
            vec = w2v_model_train.wv[word]
            tf_idf = dictionary[word]*(sent.count(word)/len(sent))
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum
    train_tfidf_sent_vectors.append(sent_vec)
    row += 1
```

```
100%|██████████| 39400/39400 [12:08<00:00, 54.12it/s]
```

In [135]:
```
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

cv_tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored in
row=0;
for sent in tqdm(w2v_cv): # 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_train and word in tfidf_feat:
            vec = w2v_model_train.wv[word]
#             tf_idf = tf_idf_matrix[row, tfidf_feat.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 = dictionary[word]*(sent.count(word)/len(sent))
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum
    cv_tfidf_sent_vectors.append(sent_vec)
    row += 1
```

```
100%|██████████| 19407/19407 [05:58<00:00, 50.89it/s]
```

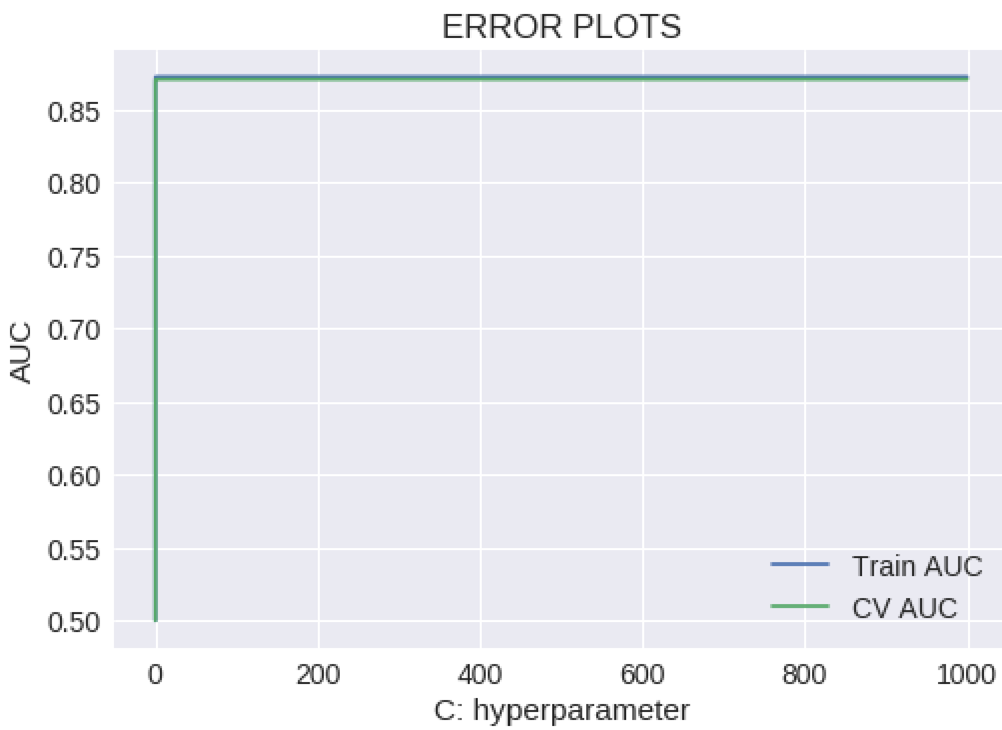
In [136]:
```
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and cell_val = t

test_tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored
row=0;
for sent in tqdm(w2v_test): # 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_train and word in tfidf_feat:
            vec = w2v_model_train.wv[word]
#             tf_idf = tf_idf_matrix[row, tfidf_feat.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 = dictionary[word]*(sent.count(word)/len(sent))
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum
    test_tfidf_sent_vectors.append(sent_vec)
    row += 1
```

```
100%|██████████| 28966/28966 [08:51<00:00, 54.50it/s]
```

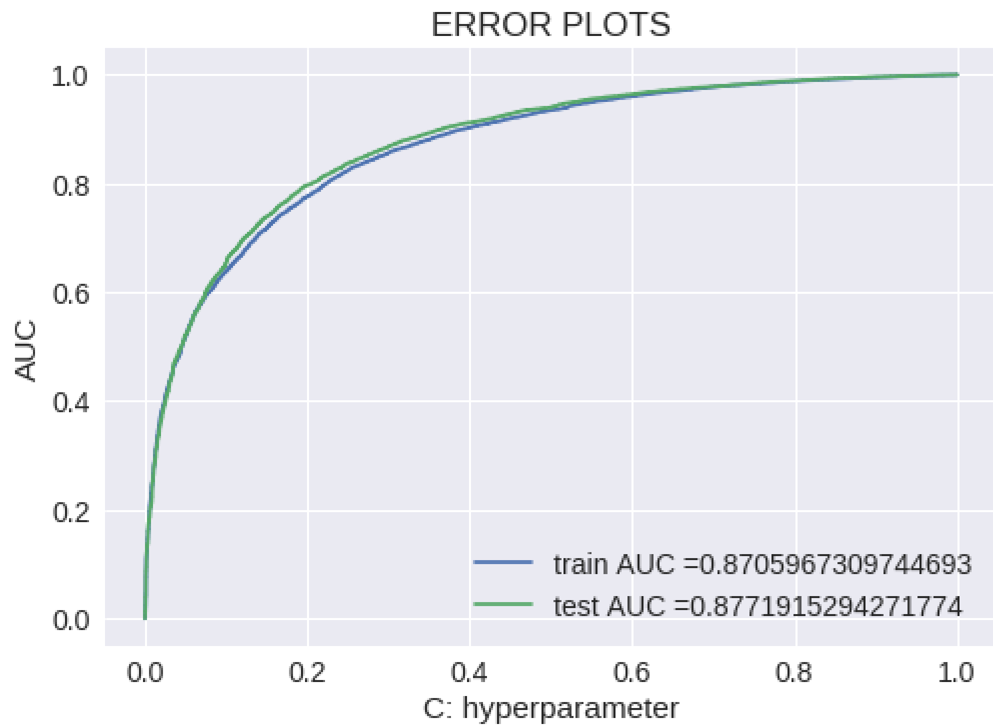
[5.4.1] Applying Logistic Regression with L1 regularization on TFIDF W2V

In [140]:
```
logistic_l1(train_tfidf_sent_vectors, cv_tfidf_sent_vectors)
```

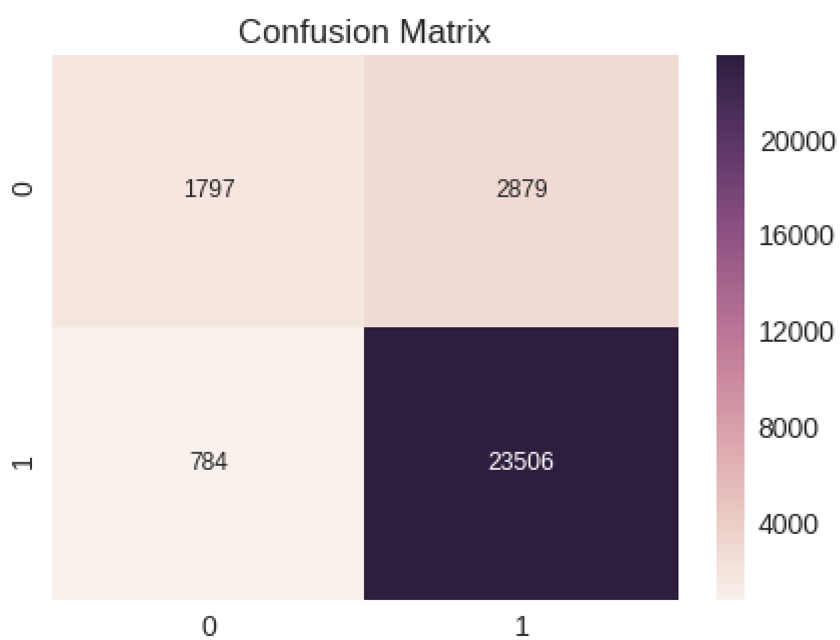
```
100%|██████████| 15/15 [00:33<00:00,  1.13it/s]

The 'C' value 500 with highest roc_auc Score is 50.0 %
```



In [141]:
```
testing_l1(train_tfidf_sent_vectors, test_tfidf_sent_vectors)
```



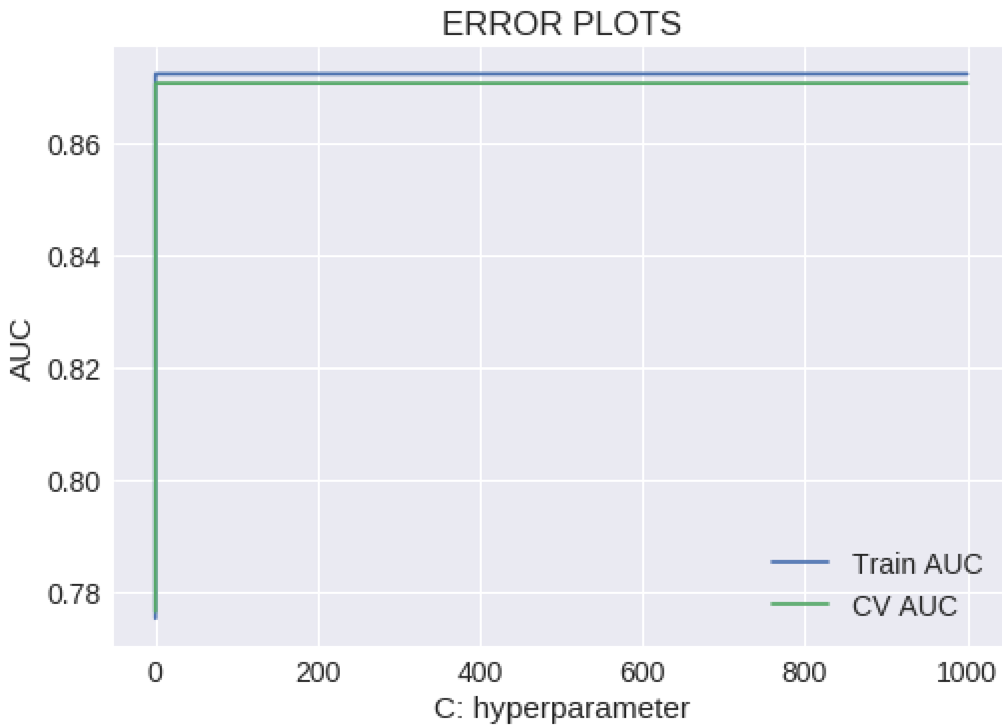

```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



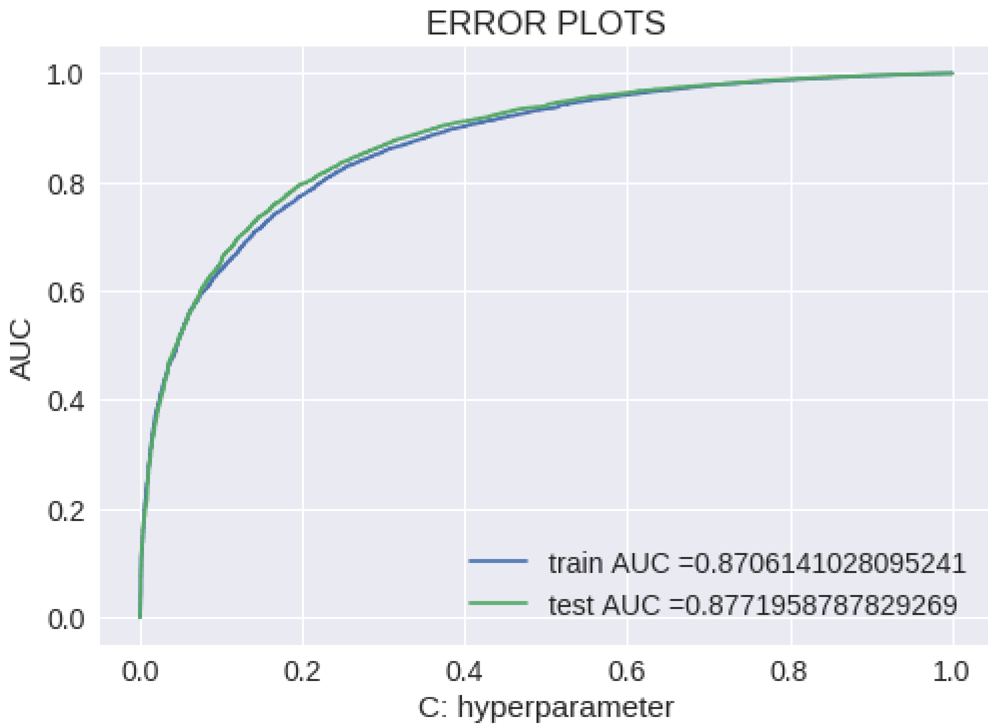
[5.4.2] Applying Logistic Regression with L2 regularization on TFIDF W2V

In [142]:
```
logistic_l2(train_tfidf_sent_vectors, cv_tfidf_sent_vectors)
```

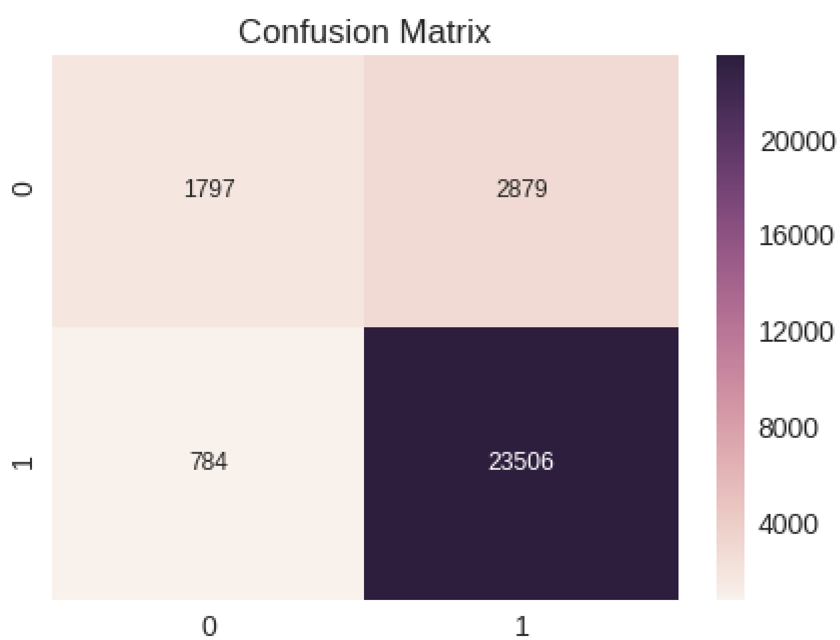
```
100%|██████████| 15/15 [00:13<00:00,  1.75it/s]

The 'C' value 5 with highest roc_auc Score is 77.6487146783811 %
```



In [143]:
```
testing_l2(train_tfidf_sent_vectors, test_tfidf_sent_vectors)
```




```
Confusion Matrix of test set:
 [ [TN  FP]
   [FN TP] ]
```



[5.4.2] Applying Logistic Regression with L2 regularization on TFIDF W2V,

[6] Conclusions

In [144]:

```
from prettytable import PrettyTable
x = PrettyTable()
x.field_names = ["Vectorizer", "L1 (C)", "Test AUC (L1)", "L2 (C)", "Test AUC(L2)

x.add_row(["BoW","0.5" ,"0.99", "0.1", 0.99])
x.add_row(["Tf-Idf","5" ,"0.99",  "5", 0.99])
x.add_row(["AVG_W2V","500" ,"0.90", "1", 0.90])
x.add_row(["TFIDF_W2V","500" ,"0.87", "5", 0.87])

from IPython.display import Markdown, display
def printmd(string):
    display(Markdown(string))
printmd('**Conclusion:**')
print(x)
```

```
<IPython.core.display.Markdown object>

+------------+--------+---------------+--------+--------------+
| Vectorizer | L1 (C) | Test AUC (L1) | L2 (C) | Test AUC(L2) |
+------------+--------+---------------+--------+--------------+
|    BoW     |  0.5   |      0.99     |  0.1   |     0.99     |
|   Tf-Idf   |   5    |      0.99     |   5    |     0.99     |
|  AVG_W2V   |  500   |      0.90     |   1    |     0.9      |
| TFIDF_W2V  |  500   |      0.87     |   5    |     0.87     |
+------------+--------+---------------+--------+--------------+
```

- **Test Prob.(unseen data) using:L1 and L2 regularization**
- BOW and Tf-idf has predicted 99% accurate on test data using both L1 regularization and L2 regularization.

In [0]: