In [0]:

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

import sqlite3
import pandas as pd
import numpy as np
import seaborn as sns
import nltk
from tqdm import tqdm
from bs4 import BeautifulSoup
import re
import datetime
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
from nltk.stem import SnowballStemmer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Normalizer
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import roc_auc_score,classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from gensim.models import Word2Vec
```

In [0]:

```
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
# Authenticate and create the PyDrive client.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

In [0]:

```
link ="https://drive.google.com/open?id=18yHOyLnrSgzAabvXoev4C2yjzSThegLB"
fluff, id = link.split('=')
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('database.sqlite')
```

Data Read and Cleanup

In [6]:

```
# Load the data from .sqlite file

db=sqlite3.connect('database.sqlite')

# select all reviews from given dataset
# we are considering a review is positive or negative on the basis of the Score column which is nothing but a rati
ng given
# by a customer for a product. If a score >3 it is considered as positive elseif score<3 it is negative and score=
3 is neutral
# Therefore all reviews which are having score other than 3 are taken into account.

filtered_data=pd.read_sql_query("""
SELECT *
FROM Reviews WHERE Score!=3""",db)

# Replace this numbers in Score column as per our assumptions i.e replace 3+ with positive 1 and 3- with negative
0
def partition(x):
    if x < 3:
        return -1
    return 1

# changing reviews with score less than 3 to be positive (1) and vice-versa
actualScore = filtered_data['Score']
positiveNegative = actualScore.map(partition)
filtered_data['Score'] = positiveNegative
print(filtered_data.shape)
```

```
(525814, 10)
```

In [0]:

```
# converting datestamp into string representable form as YYYY-MM-DD
filtered_data["Time"] = filtered_data["Time"].map(lambda t: datetime.datetime.fromtimestamp(t).strftime('%Y-%m-%d'
))
```

In [8]:

```
# There is lot of duplicate data present as we can see above productId B007OSBE1U
# have multiple duplicate reviews this is what we need to avoid.

# so first step is to sort the data and then remove duplicate entries so that only
# one copy of them should be remain in our data.
dup_free=filtered_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"})
# dup_free.head()
# This is shape of our dataset of 100k datapoints after removal of dups
dup_free.shape
```

Out[8]:

```
(364173, 10)
```

In [0]:

```
final_filtered_data=dup_free[dup_free.HelpfulnessNumerator<=dup_free.HelpfulnessDenominator]
```

In [10]:

```
final_filtered_data.shape
```

Out[10]:

```
(364171, 10)
```

In [11]:

```
((final_filtered_data['Id'].size*1.0)/(filtered_data['Id'].size*(1.0)))*100
```

Out[11]:

```
69.25852107399194
```

so after data cleanup we left with 69.25% data of 525k datapoints

In [0]:

```
filtered_data=filtered_data.sort_values(by='Time').reset_index(drop=True)
```

In [13]:

```
final=filtered_data.sample(frac=0.13,random_state=2)
final.shape
```

Out[13]:

```
(68356, 10)
```

In [14]:

```
print("Positive Reviews: ",final[final.Score ==1].shape[0])
print("Positive Reviews: ",final[final.Score ==-1].shape[0])
```

```
Positive Reviews:  57745
Positive Reviews:  10611
```

In [15]:

```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Out[15]:

```
True
```

Text Preprocessing

In [0]:

```
# Now we have already done with data cleanup part. As in our dataset most cruicial or I can say most determinant feature
# from which we can say it is positive or negative review is review Text.
# So we are need to perform some Text Preprocessing on it before we actually convert it into word vector or vectorization

# I am creating some precompiled objects for our regular expressions cause it will be used for over ~64K times (in our case)
# as it seems fast but using regular expression is CPU expensive task so it would be faster to use precompiled search objects.

_wont  = re.compile(r"won't")
_cant  = re.compile(r"can\'t")
_not   = re.compile(r"n\'t")
_are   = re.compile(r"\'re")
_is    = re.compile(r"\'s")
_would = re.compile(r"\'d")
_will  = re.compile(r"\'ll")
_have  = re.compile(r"\'ve")
_am    = re.compile(r"\'m")

# we are ignoring "not" from stopwords as "not" plays important role for semantic analysis as it can alone change the
# meaning of whole sentence
stopWords = set(stopwords.words('english'))
sw=stopWords.copy()
sw.discard('not')

def expand_abbrevated_words(phrase):
    phrase = re.sub(_wont, "will not", phrase)
    phrase = re.sub(_cant, "can not", phrase)
    phrase = re.sub(_not, " not", phrase)
    phrase = re.sub(_are, " are", phrase)
    phrase = re.sub(_is, " is", phrase)
    phrase = re.sub(_would, " would", phrase)
    phrase = re.sub(_will, " will", phrase)
    phrase = re.sub(_have, " have", phrase)
    phrase = re.sub(_am, " am", phrase)
    return phrase

# As this dataset is web scrapped from amazon.com while scrapping there might be a good chance that we are getting some garbage
# characters/words/sentences in our Text data like html tags,links, alphanumeric characters so we ought to remove them
def remove_unwanted_char(data):
    processed_data=[]
    for sentence in tqdm(data):
        sentence = re.sub(r"http\S+", "", sentence) # this will remove links
        sentence = BeautifulSoup(sentence, 'lxml').get_text()
        sentence = re.sub("\S*\d\S*", "", sentence).strip() #remove alphanumeric words
        sentence = re.sub('[^A-Za-z]+', ' ', sentence) #remove special characters
        sentence =  expand_abbrevated_words(sentence)
        # we need to convert everything into lower case because I dont want my model to treat same word differently
        # if it appears in the begining of sentence and somewhere middle of sentence.
        # Also remove stopword froms from sentences
        sentence =" ".join(j.lower() for j in sentence.split() if j.lower() not in sw)
        processed_data.append(sentence)
    return processed_data

def preprocess_my_data(data):
    return remove_unwanted_char(data)
```

In [17]:

```
data_to_be_processed=final['Text'].values
processed_data=preprocess_my_data(data_to_be_processed)
label=final['Score']
print(len(processed_data))
```

```
100%|██████████| 68356/68356 [00:30<00:00, 2239.65it/s]

68356
```

In [18]:

```
final['CleanedText']=processed_data
print(processed_data[0])
```

```
tried several times get good coconut flavored coffee little success boyer trick great coffee good amount co
conut flavor highly recommend
```

In [19]:

```
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

Out[19]:

```
True
```

Stemming

In [20]:

```
# Before applying BoW or Tfidf featurization techinque on our corpus we need to apply stemmming for each word in e
ach document.
stemmed_data=processed_data.copy()
bow_stem=SnowballStemmer('english')
stemmed_reviews=[]
def stemSentence(review):
    token_words=word_tokenize(review)
    stem_sentence=[]
    for word in token_words:
        stem_sentence.append(bow_stem.stem(word))
        stem_sentence.append(" ")
    return "".join(stem_sentence)

for review in tqdm(stemmed_data):
    stemmed_reviews.append(stemSentence(review))
```

```
100%|██████████| 68356/68356 [01:05<00:00, 1041.06it/s]
```

Splitting Data In Train ,CV and Test Dataset

In [21]:

```
# To avoid data leakage we are splitting our dataset before any featurization.
x_tr, x_test, y_tr, y_test = train_test_split(stemmed_reviews, label, test_size=0.2, random_state=0)

print("Sizes of Train,test dataset after split: {0} , {1}".format(len(x_tr),len(x_test)))
```

```
Sizes of Train,test dataset after split: 54684 , 13672
```

HyperParameter Tuning Using Simple Cross-Validation

In [0]:

```
def find_best_c(train_data,tr_label,penalty):
    lr = LogisticRegression()
    c_val=[10**-4, 10**-2, 10**0, 10**2, 10**4]
    parameters = [{'C': c_val,'penalty':[penalty],'class_weight':['balanced']}]

    tbs_cv = TimeSeriesSplit(n_splits=5).split(train_data)
    gsearch = GridSearchCV(estimator=lr, cv=tbs_cv,
                    param_grid=parameters, scoring = 'roc_auc',return_train_score=True)
    gsearch.fit(train_data, tr_label)

    print("Best C   : ",gsearch.best_estimator_.C)
    print("Best AUC : ",gsearch.score(train_data, tr_label))

    test_AUC=gsearch.cv_results_['mean_test_score']
    train_AUC=gsearch.cv_results_['mean_train_score']
    c_val=np.log(c_val)
    plt.plot(c_val, test_AUC,'bo',linestyle="solid",label='Test AUC')
    plt.plot(c_val, train_AUC,'yo',linestyle="solid",label='Train AUC')
    plt.xlabel('log(c) values')
    plt.ylabel('AUC')
    plt.legend(loc="upper right")
    plt.grid()
    plt.show()
    return gsearch.best_estimator_.C
```

In [0]:

```
def testing_on_test_data(train_rev,train_label,test_rev,test_label,best_C,penalty):
    plt.figure(1)
    lr=LogisticRegression(C=best_C,penalty=penalty,class_weight='balanced')
    lr.fit(train_rev,train_label)
    train_pred = lr.predict_log_proba(train_rev)[:,1]
    test_pred= lr.predict_log_proba(test_rev)[:,1]
    # Train data AUC value
    fpr_tr,tpr_tr, _  = roc_curve(train_label, train_pred)
    roc_auc_tr = auc(fpr_tr, tpr_tr)

    # Test data AUC value
    fpr_t,tpr_t, _  = roc_curve(test_label, test_pred)
    roc_auc_t= auc(fpr_t, tpr_t)

    plt.plot(fpr_tr, tpr_tr, color='darkorange',
             lw=2, label='Train ROC curve (area = %0.2f)' % roc_auc_tr)
    plt.plot(fpr_t, tpr_t, color='black',
             lw=2, label='Test ROC curve (area = %0.2f)' % roc_auc_t)

    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic for Train and Test dataset')
    plt.legend(loc="lower right")
    plt.show()
```

In [0]:

```
def get_confusion_matrix(train_rev,train_label,test_rev,test_label,best_C,penalty):
    plt.figure(1,figsize=(15,7))
    np.set_printoptions(precision=5)
    lr=LogisticRegression(C=best_C,penalty=penalty,class_weight='balanced')
    lr.fit(train_rev,train_label)
    train_pred=lr.predict(train_rev)
    test_pred=lr.predict(test_rev)

    test_cnf_matrix=confusion_matrix(test_label,test_pred)
    train_cnf_matrix=confusion_matrix(train_label,train_pred)

    plt.subplot(121)
    sns.heatmap(test_cnf_matrix,cmap="coolwarm_r",fmt='.8g',annot=True,linewidths=0.5)
    plt.title("TestSet Confusion_matrix")
    plt.xlabel("Predicted_class")
    plt.ylabel("Actual class")

    plt.subplot(122)
    sns.heatmap(train_cnf_matrix,cmap="coolwarm_r",fmt='.8g',annot=True,linewidths=0.5)
    plt.title("TrainSet Confusion_matrix")
    plt.xlabel("Predicted_class")
    plt.ylabel("Actual class")
    plt.show()
```

In [0]:

```
def get_top_imp_features(train_rev,labels,vectorizer,penalty,best_C):
    lr=LogisticRegression(C=best_C,penalty=penalty,class_weight='balanced')
    lr.fit(train_rev,labels)
    #this sorts the  features probabilities return index of sorted values
    class_prob_pos=lr.coef_
    class_prob_pos=class_prob_pos.argsort()
    # As we are sorting in ascending order, last 10 weights of positive class
    # and first 10 will be of negative class
    # feature_log_prob_ stores
    positive_class_prob=class_prob_pos[0][-10:]
    positive_class_prob=np.take(vectorizer.get_feature_names(), class_prob_pos[0][-10:])
    negative_class_prob=np.take(vectorizer.get_feature_names(), class_prob_pos[0][:10])
    print("Top Ten Positive Features: ",positive_class_prob)
    print("Top Ten Negative Features: ",negative_class_prob)
```

BoW (Bag of Words)

In [0]:

```
# Applying fit_transform to only train dataset as we are only because we want our vocabulary to be built only on t
rain data
bow_count=CountVectorizer(min_df=10, max_features=2000,ngram_range=(1,2))
bow_fit=bow_count.fit(x_tr)
print("Some Feature names: ",bow_fit.get_feature_names()[:5])

#extract token count out of raw text document using vocab build using train dataset
bow_train=bow_count.transform(x_tr)
bow_test=bow_count.transform(x_test)

print("Shape of transformed train text reviews",bow_train.shape)
print("Shape of transformed test text reviews",bow_test.shape)
```

```
Some Feature names:  ['abl', 'abl find', 'absolut', 'absolut delici', 'absolut love']
Shape of transformed train text reviews (54684, 2000)
Shape of transformed test text reviews (13672, 2000)
```

In [0]:

```
# converting sparse matrix to dense matrix before doing standardization
bow_dense_train_reviews=bow_train.toarray()
bow_dense_test_reviews=bow_test.toarray()
```

In [0]:

```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()

std_train_data=std_data.fit_transform(bow_dense_train_reviews*1.0)
std_test_data=std_data.transform(bow_dense_test_reviews*1.0)
```

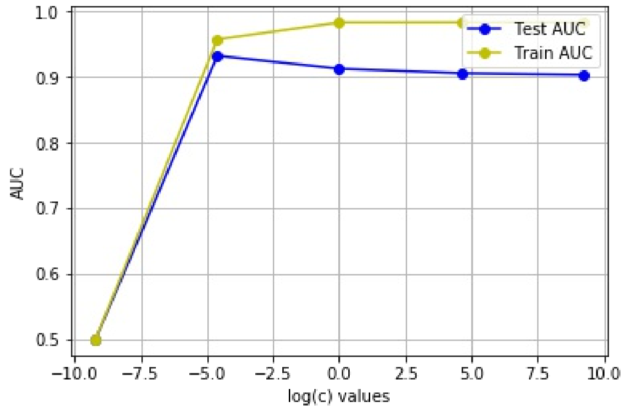
[1.1] Logistic Regression on BOW

[1.1.1] Applying Logistic Regression with L1 regularization on BOW

In [0]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l1')
```

```
Best C   :  0.01
Best AUC :  0.9606956587973842
```



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

In [0]:

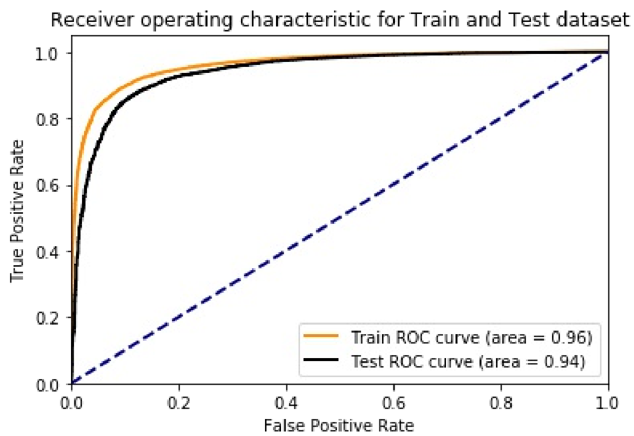
```
lr = LogisticRegression(C=best_c, penalty='l1');
lr.fit(std_train_data, y_tr);
w = lr.coef_
print("Number of non-zero features of weight vector: ",np.count_nonzero(w))
```

```
Number of non-zero features of weight vector:  784
```

[1.1.3] Testing with Test Data

In [0]:

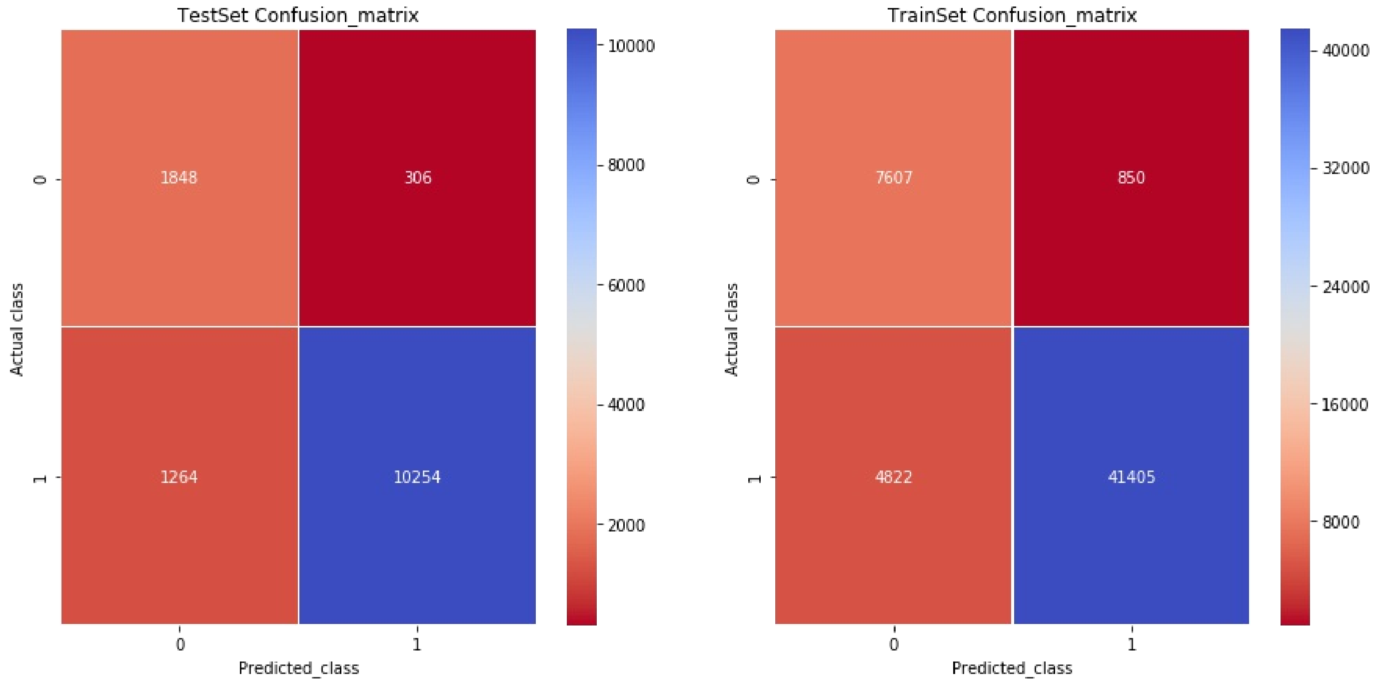
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l1')
```



[1.1.4] Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l1')
```



[1.1.5] Feature Importance on BOW

[1.1.5.1] Top 10 positive and negative features using l1 regularizer

In [0]:

```
get_top_imp_features(std_train_data,y_tr,bow_fit,'l1',best_c)
```

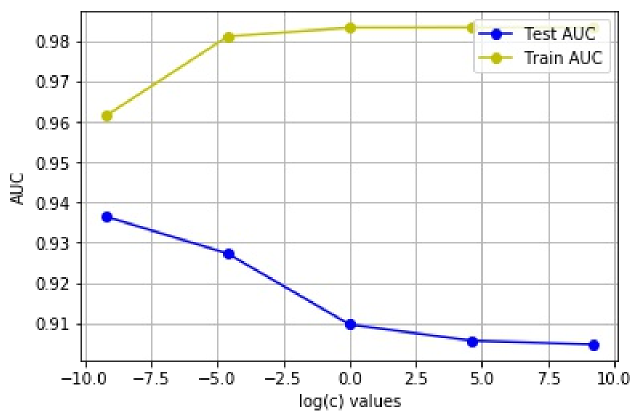
```
Top Ten Positive Features:  ['amaz' 'favorit' 'excel' 'high recommend' 'perfect' 'good' 'delici'
 'best' 'love' 'great']
Top Ten Negative Features:  ['disappoint' 'not' 'worst' 'return' 'aw' 'unfortun' 'terribl' 'bad'
 'bland' 'not buy']
```

[1.2] Applying Logistic Regression with L2 regularization on BOW

In [0]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l2')
```

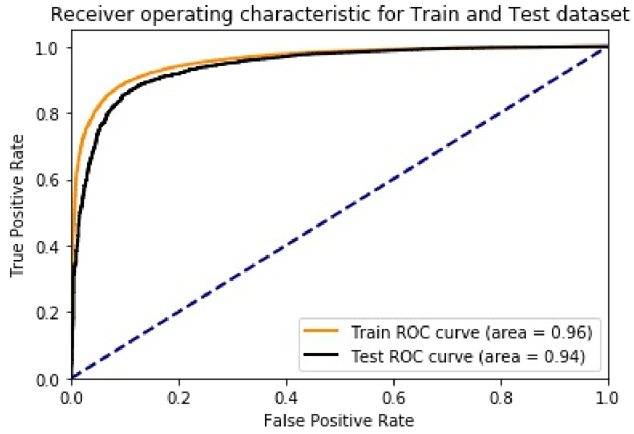
```
Best C   :  0.0001
Best AUC :  0.9571995918297177
```



[1.2.2] Testing with Test Data

In [0]:

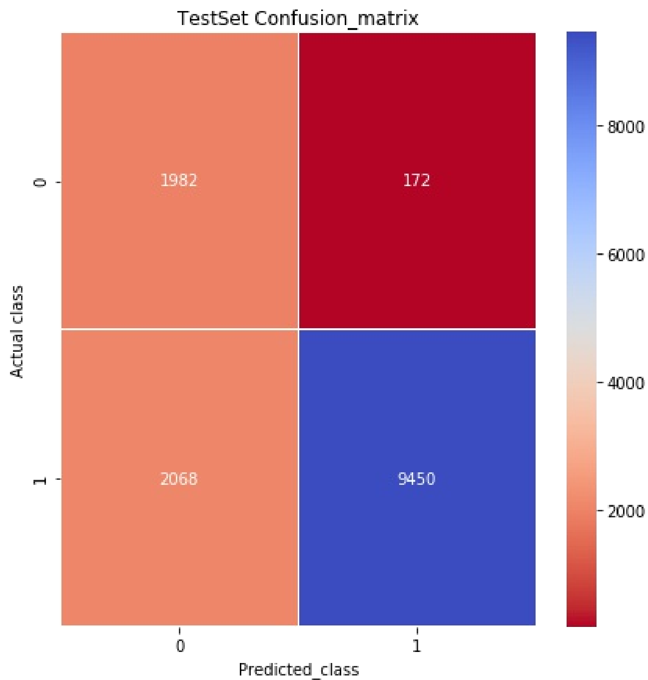
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```

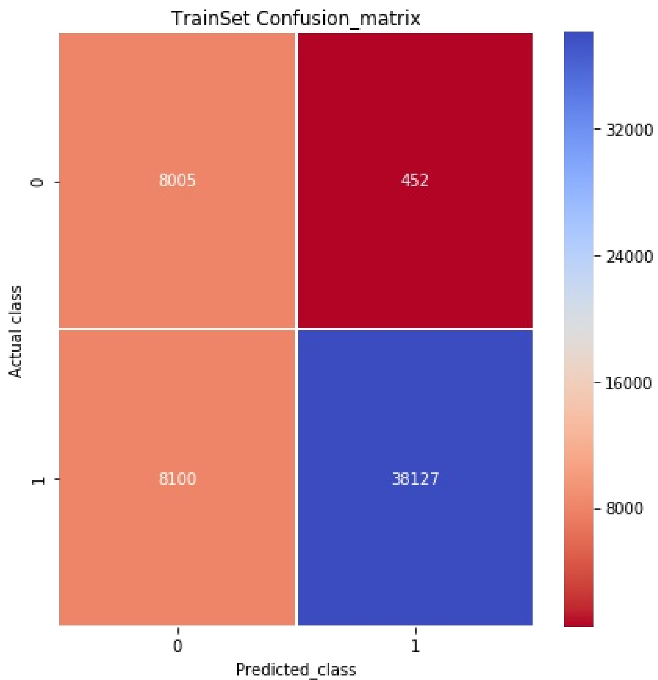


[1.2.3] Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```





[1.2.4] Feature Importance on BOW

[1.2.4.1] Top 10 positive and negative features using l1 regularizer

In [0]:

```
get_top_imp_features(std_train_data,y_tr,bow_fit,'l2',best_c)
```

```
Top Ten Positive Features:  ['high recommend' 'excel' 'favorit' 'nice' 'good' 'perfect' 'delici'
 'best' 'love' 'great']
Top Ten Negative Features:  ['disappoint' 'not' 'worst' 'return' 'not buy' 'aw' 'terribl' 'bad'
 'not recommend' 'unfortun']
```

[1.2.3] Performing pertubation test (multicollinearity check) on BOW

In [0]:

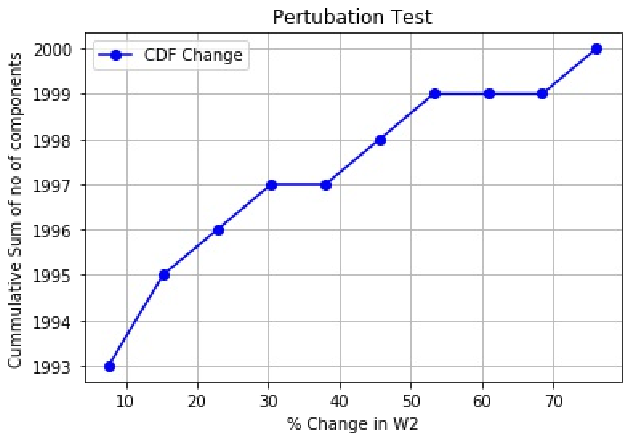
```
lr_before=LogisticRegression(penalty='l2',C=0.0001)
lr_before.fit(std_train_data,y_tr)
# Adding an small epsilon value in train data to see if there is significant change in new weight vector
epsilon = np.random.normal(0, 0.01)
std_train_data_1 = std_train_data + epsilon
# Training with new train data
lr_after = LogisticRegression(C=0.0001, penalty='l2')
lr_after.fit(std_train_data_1, y_tr)

W1=lr_before.coef_
W2=lr_after.coef_

#adding small value to W1 and W2 to avoid divide by zero
W1 = W1+10**-6
W2 = W2+10**-6
```

In [0]:

```
# calculating percentage change of weight for each feature
change=abs((W1-W2) / (W1))*100
```

In [0]:

```
#plotting CDF of %change in Weight vector after adding epsilon
num_bins = 10
counts, bin_edges = np.histogram(change, bins=num_bins)
cdf = np.cumsum(counts)
#plot the cdf
plt.plot(bin_edges[1:]/100, cdf,'bo',linestyle="solid",label='CDF Change')
plt.xlabel("% Change in W2")
plt.ylabel("Cummulative Sum of no of components")
plt.grid()
plt.legend()
plt.title("Pertubation Test")
plt.show()
```



we can observe a sudden rise in our CDF at ~39th percentile i.e there are ~1997 features which having change less than ~40% and 3 features are having large change.

In [0]:

```
feat=bow_fit.get_feature_names()
print("feature names whose % change is more than a 39%:")
for c in feat[1997:2000]:
    print(c,end=",")
```

```
feature names whose % change is more than a 39%:
yummi,zero,zico,
```

2.TFIDF

In [0]:

```
tfidf_count= TfidfVectorizer(min_df=10,max_features=2000,ngram_range=(1,2))
tfidf_tr=tfidf_count.fit_transform(x_tr)
tfidf_test=tfidf_count.transform(x_test)
print("Shape of tfidf vector representation of train review text :",tfidf_tr.shape)
print("Shape of tfidf vector representation of test review text  :",tfidf_test.shape)
```

```
Shape of tfidf vector representation of train review text : (54684, 2000)
Shape of tfidf vector representation of test review text  : (13672, 2000)
```

In [0]:

```
# converting sparse matrix to dense matrix before doing standardization
tfidf_dense_train_reviews=tfidf_tr.toarray()
tfidf_dense_test_reviews=tfidf_test.toarray()
```

In [0]:

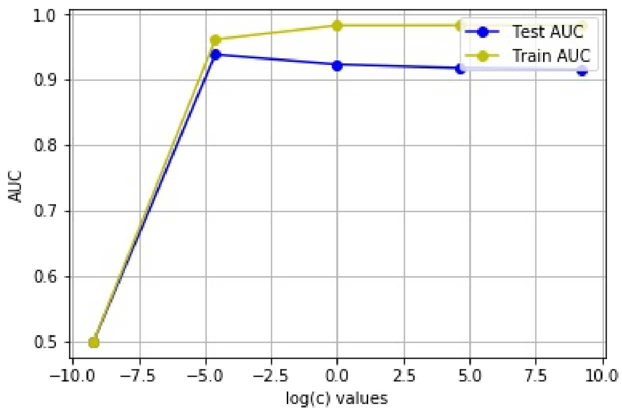
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()

std_train_data=std_data.fit_transform(tfidf_dense_train_reviews*1.0)
std_test_data=std_data.transform(tfidf_dense_test_reviews*1.0)
```

[2.1] Applying Logistic Regression with L1 regularization on TFIDF

In [0]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l1')
```

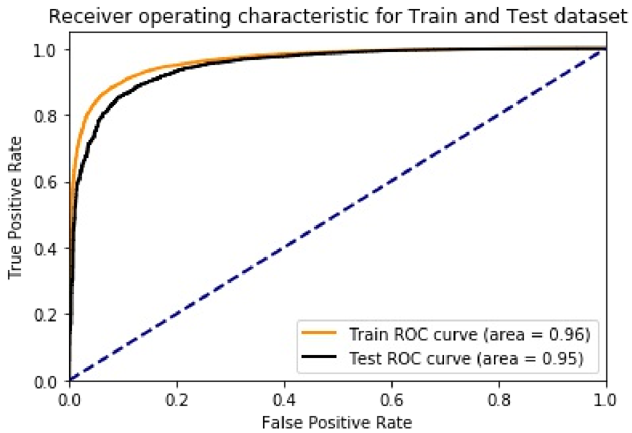
```
Best C    :  0.01
Best AUC :  0.9637926586293718
```



[2.1.1] Testing with Test Data

In [0]:

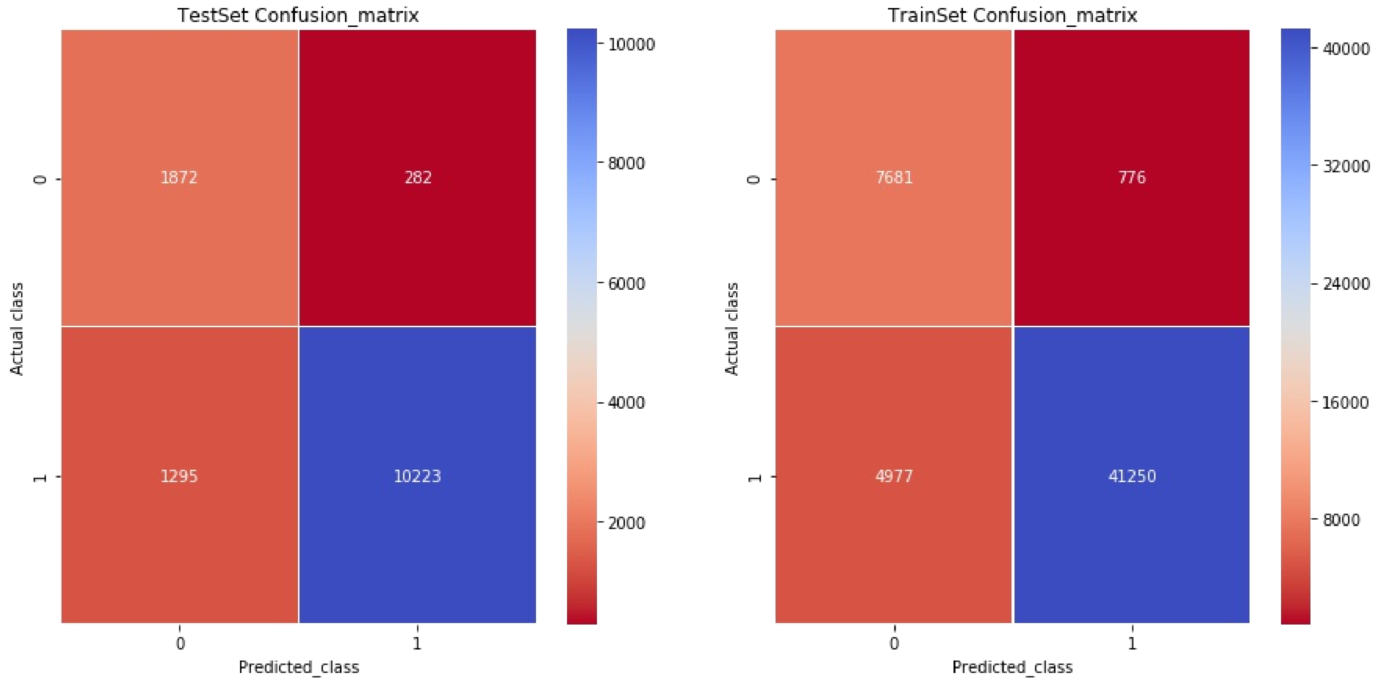
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l1')
```



[2.1.2] Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l1')
```



[2.1.3] Feature Importance on TFIDF

[2.1.3.1] Top 10 positive and negative features using l1 regularizer

In [0]:

```
get_top_imp_features(std_train_data,y_tr,tfidf_count,'l1',best_c)
```

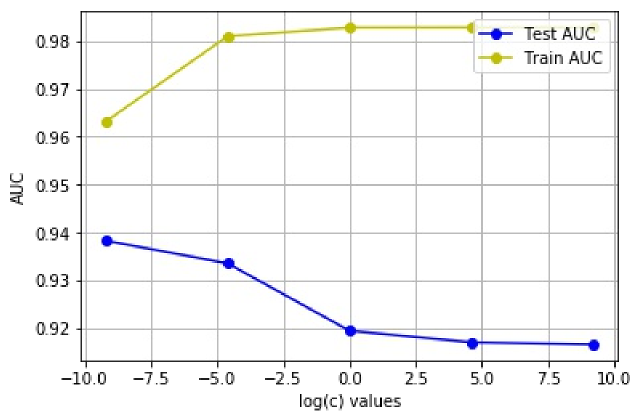
```
Top Ten Positive Features:  ['food' 'follow' 'folk' 'foil' 'flower' 'flour' 'floor' 'flax' 'forc'
 'zico']
Top Ten Negative Features:  ['abl' 'pork' 'popular' 'popcorn' 'popchip' 'pop' 'poor' 'poop' 'pomegran'
 'point']
```

[2.2] Applying Logistic Regression with L2 regularization on TFIDF

In [0]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l2')
```

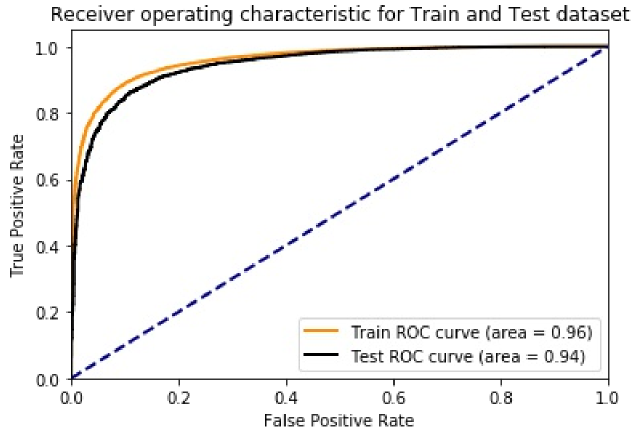
```
Best C   :  0.0001
Best AUC :  0.9590669851703914
```



[2.2.1] Testing with Test Data

In [0]:

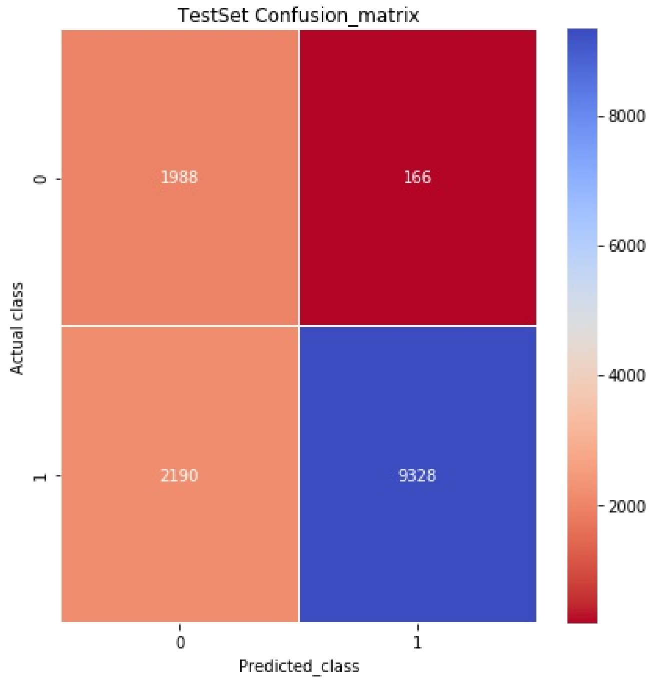
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```

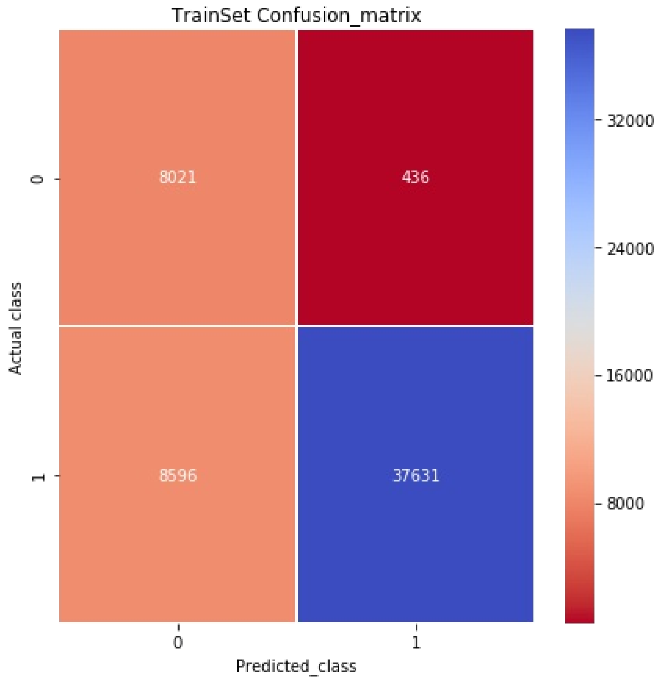


[2.2.2] Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```





[2.2.3] Feature Importance on TFIDF

[2.2.3.1] Top 10 positive and negative features using l2 regularizer

In [0]:

```
get_top_imp_features(std_train_data,y_tr,tfidf_count,'l2',best_c)
```

```
Top Ten Positive Features:  ['amaz' 'excel' 'favorit' 'nice' 'perfect' 'delici' 'good' 'best' 'love'
 'great']
Top Ten Negative Features:  ['disappoint' 'not' 'worst' 'return' 'terribl' 'aw' 'bad' 'not buy'
 'not recommend' 'horribl']
```

3. Avg Word2Vec

In [0]:

```
# As w2vec preserves semantic meaning of words I am not going to do stemming for this.
# split each sentence from train dataset into words
reviews=x_tr.copy()
train_sentences_set=[]
for s in reviews:
    train_sentences_set.append(s.split())
```

In [0]:

```
# min_count = 10 considers only words that occured atleast 10 times
# size = dimensionality of word vectors
# workers = no of threads to use while training our w2v model/featurization
w2v_model=Word2Vec(train_sentences_set,min_count=10,size=200, workers=4)
```

In [0]:

```
w2v_words = list(w2v_model.wv.vocab)
```

In [0]:

```
def compute_avgW2Vec(reviews):
    # average Word2Vec
    # compute average word2vec for each review.
    rev_words=[]
    for i in reviews:
        rev_words.append(i.split())

    sent_vectors = []; # the avg-w2v for each sentence/review is stored in this list
    for sent in tqdm(rev_words): # for each review/sentence
        sent_vec = np.zeros(200) # as our w2v model is trained with size=50 i.e 50 dimension so this value will be
 change as dim change
        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 sent_vectors #Average W2v repersentation of reviews in given dataset
```

In [29]:

```
train_avgw2v=compute_avgW2Vec(x_tr)
```

```
100%|██████████| 54684/54684 [00:45<00:00, 1212.80it/s]
```

In [30]:

```
test_avgw2v=compute_avgW2Vec(x_test)
```

```
100%|██████████| 13672/13672 [00:11<00:00, 1207.31it/s]
```

In [0]:

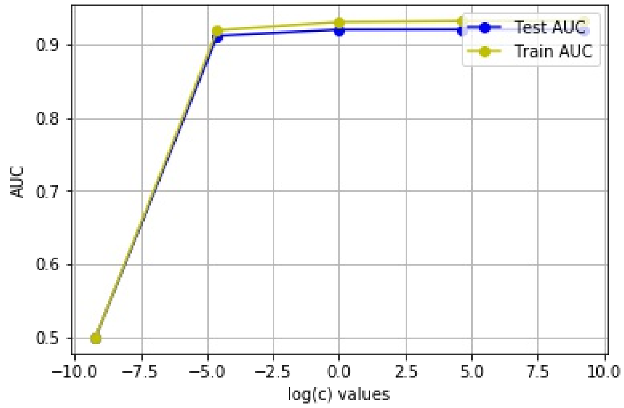
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()

std_train_data=std_data.fit_transform(train_avgw2v)
std_test_data=std_data.transform(test_avgw2v)
```

[3.1] Applying Logistic Regression with L1 regularization on Avg Word2Vec

In [0]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l1')
```

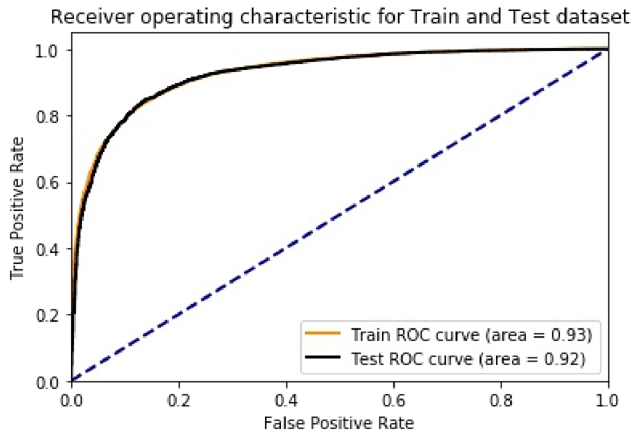
```
Best C   :  100
Best AUC :  0.9274151985598038
```



[3.1.1] Testing with Test Data

In [0]:

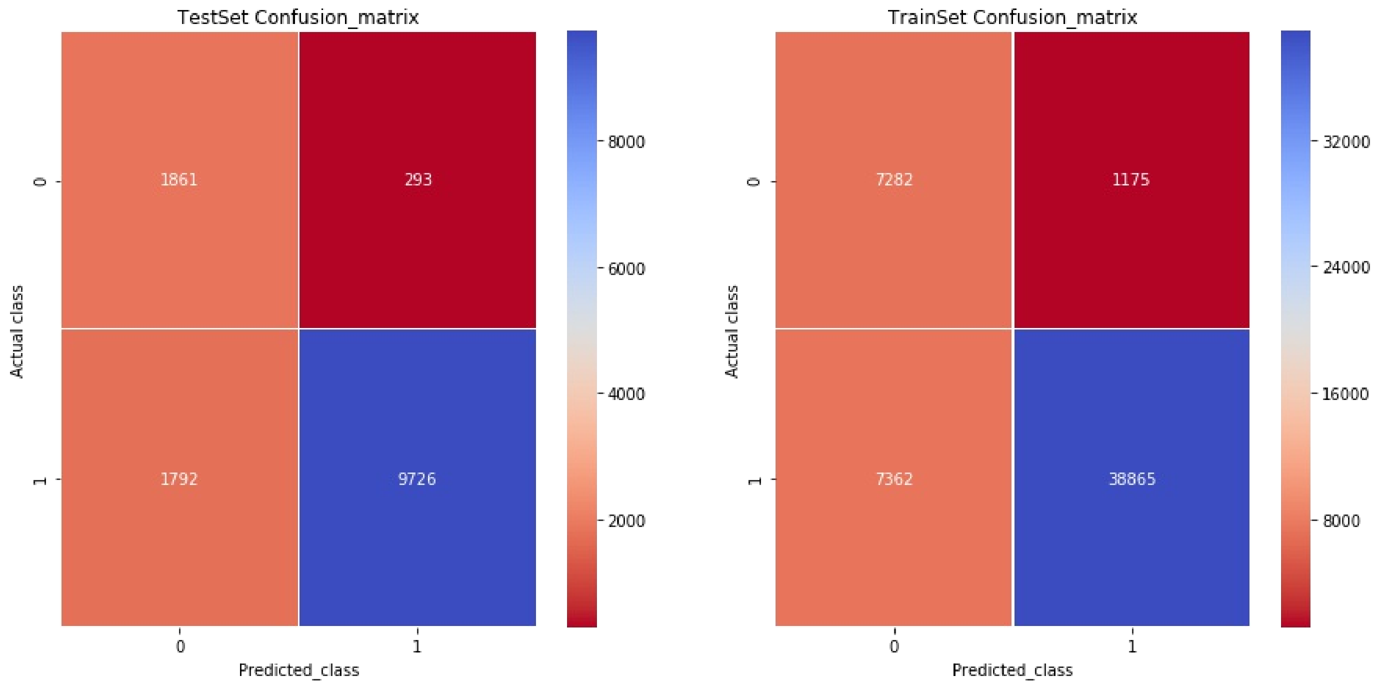
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l1')
```



[3.1.2] Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,100,'l1')
```

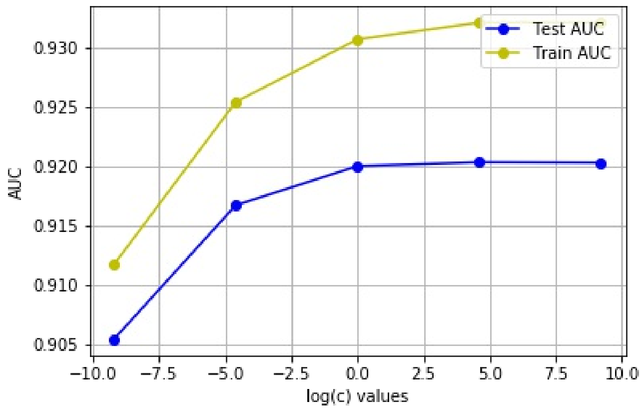


[3.2] Applying Logistic Regression with L2 regularization on Avg Word2Vec

In [32]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l2')
```

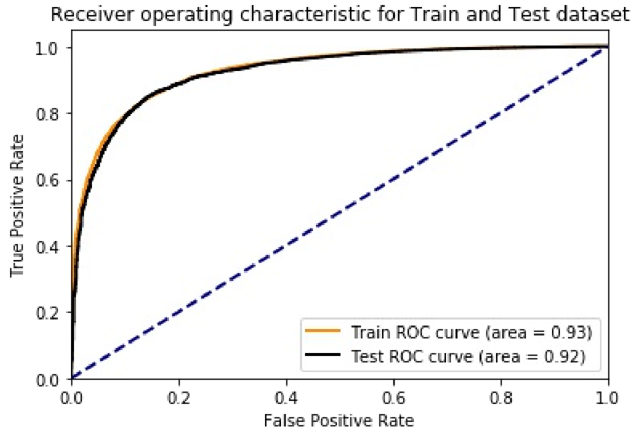
```
Best C   :  100
Best AUC :  0.9275968547835205
```



[3.2.1] Testing with Test Data

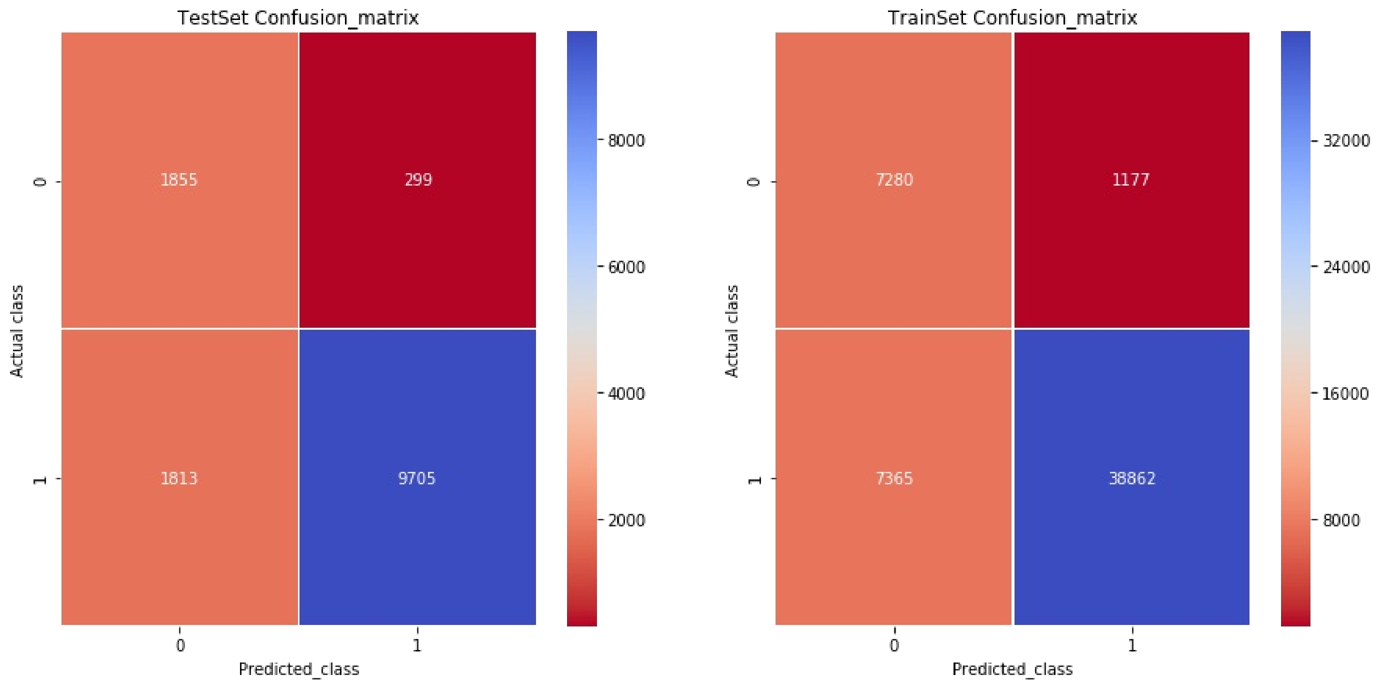
In [33]:

```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```



In [34]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```



[4.1] Applying Logistic Regression on TFIDF weighted W2Vec

In [0]:

```
# As w2vec preserves semantic meaning of words I am not going to do stemming for this.
# split each sentence from train dataset into words
reviews=x_tr.copy()
train_sentences_set=[]
for s in reviews:
    train_sentences_set.append(s.split())
```

In [0]:

```
# min_count = 10 considers only words that occured atleast 10 times
# size = dimensionality of word vectors
# workers = no of threads to use while training our w2v model/featurization
w2v_model=Word2Vec(train_sentences_set,min_count=10,size=300, workers=4)
```

In [0]:

```
w2v_words = list(w2v_model.wv.vocab)
```

In [0]:

```
tfidf_w2v = TfidfVectorizer(min_df=10,max_features=300)
tfidf_w2v.fit(x_tr)
# we are converting a dictionary with word as a key, and the idf as a value
dictionary = dict(zip(tfidf_w2v.get_feature_names(), list(tfidf_w2v.idf_)))
tfidf_feat = tfidf_w2v.get_feature_names() # tfidf words/col-names
```

In [0]:

```
def compute_tfidf_w2vec(reviews):
    tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored in this list
    row=0;
    rev_words=[]
    for i in reviews:
        rev_words.append(i.split())

    for sent in tqdm(rev_words): # for each review/sentence
        sent_vec = np.zeros(300) #as our w2v model is trained with size=50 i.e 500 dimension so this value will be
 change as dim change
        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 tfidf_feat:
                vec = w2v_model.wv[word]
                # dictionary[word] = idf value of word in whole courpus
                # sent.count(word) = tf values 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
        tfidf_sent_vectors.append(sent_vec)
        row += 1

    return tfidf_sent_vectors
```

In [31]:

```
train_tfidf_w2v=compute_tfidf_w2vec(x_tr)
```

```
100%|██████████| 54684/54684 [01:22<00:00, 664.01it/s]
```

In [32]:

```
test_tfidf_w2v=compute_tfidf_w2vec(x_test)
```

```
100%|██████████| 13672/13672 [00:21<00:00, 650.88it/s]
```

In [0]:

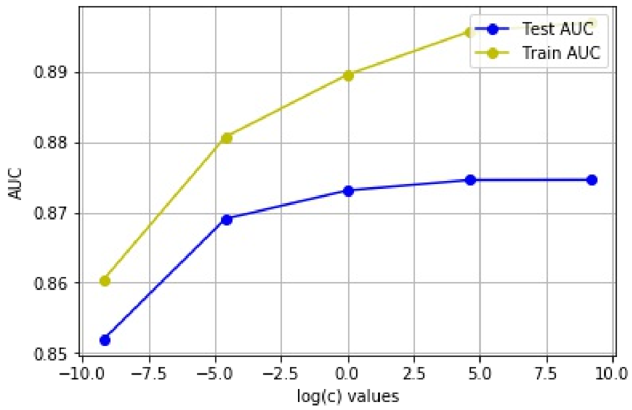
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()

std_train_data=std_data.fit_transform(train_tfidf_w2v)
std_test_data=std_data.transform(test_tfidf_w2v)
```

[4.1] Applying Logistic Regression with L2 regularization on TFIDF weighted W2Vec

In [43]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l2')
```

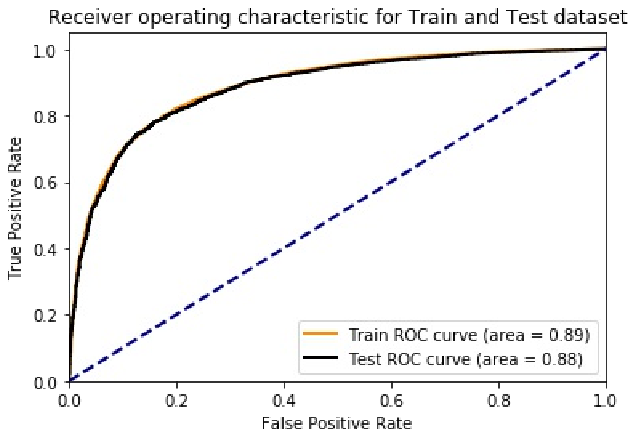
```
Best C    :  10000
Best AUC :  0.8881341664569615
```



[4.1.1] Testing with Test Data

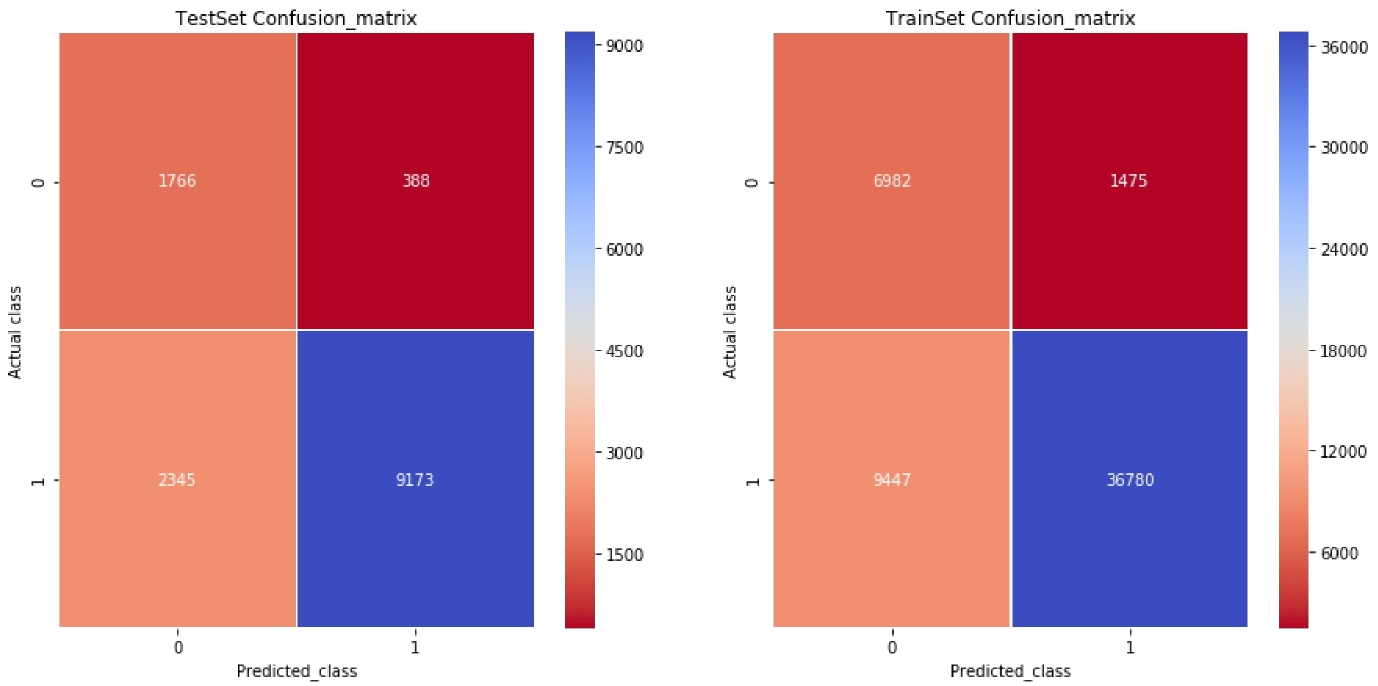
In [44]:

```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```



In [45]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=best_c,penalty='l2')
```

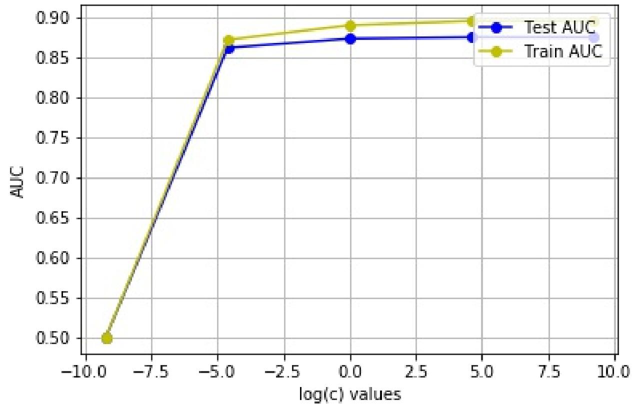


[4.2] Applying Logistic Regression with L1 regularization on TFIDF weighted W2Vec

In [37]:

```
best_c=find_best_c(std_train_data,y_tr,penalty='l1')
```

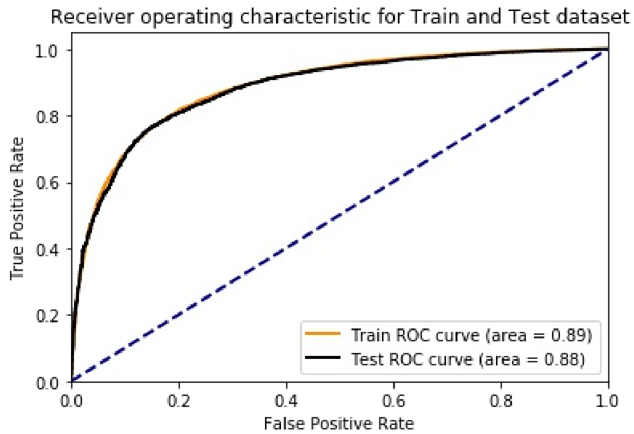
```
Best C   :  100
Best AUC :  0.8858661482037353
```



[4.2.1] Testing with Test Data

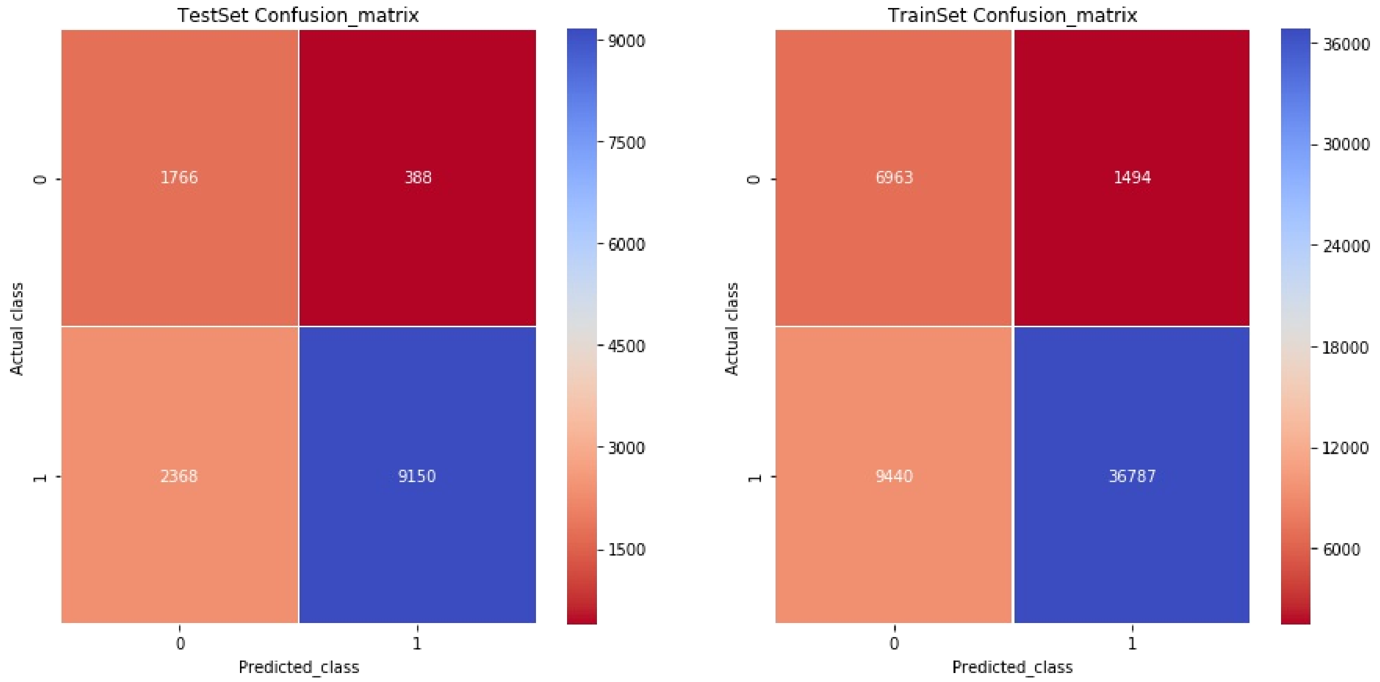
In [32]:

```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_C=100,penalty='l1')
```



In [35]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_C=100,penalty='l1')
```



Summary

In [1]:

```
print("""
          |  Vectorizer     | HyperParameter'('C')'  |   Regularization      |          AUC          |
          ___________________________________________________________________________________

             BoW            |       0.01             |     L1                |  0.9606956587973842 |
             BoW            |       0.0001           |     L2                |  0.9571995918297177 |

           tfidf            |       0.01             |     L1                |  0.9637926586293718 |
           tfidf            |       0.0001           |     L2                |  0.9590669851703914 |

        Avg tfidf           |       100              |     L1                |  0.9274151985598038 |
        Avg tfidf           |       100              |     L2                |  0.9275968547835205 |

    tfidf Weighted W2v      |       100              |     L1                |  0.8858661482037353 |
    tfidf Weighted W2v      |      10000             |     L2                |  0.8881341664569615 |
                """)
```

```
          |  Vectorizer     | HyperParameter'('C')'  |   Regularization      |          AUC          |
          ___________________________________________________________________________________

             BoW            |       0.01             |     L1                |  0.9606956587973842 |

             BoW            |       0.0001           |     L2                |  0.9571995918297177 |

           tfidf            |       0.01             |     L1                |  0.9637926586293718 |

           tfidf            |       0.0001           |     L2                |  0.9590669851703914 |

        Avg tfidf           |       100              |     L1                |  0.9274151985598038 |
        Avg tfidf           |       100              |     L2                |  0.9275968547835205 |

    tfidf Weighted W2v      |       100              |     L1                |  0.8858661482037353 |
    tfidf Weighted W2v      |      10000             |     L2                |  0.8881341664569615 |
```

Conclusion:

1. Logistic Regression is Linear Classification algorithm.
2. Basic assumption we made on training data is- it is linearly seperable.
3. Based on this assumption we try to find the line/plane/hyperplane which seperates our datapoints from each other and minimize the number of missclassified points.
4. Due to use of sigmoid function in optimization problem our Logistic Regression model becomes robust to outliers as well give probablistic outputs.
5. sigmoid function converts distance values to range of [0,1] because of that we get probablistic interpretation of our results and this technique is called as SQUASHING.
6. Our data varies on time so we used Time Based Splitting to Train,CV and Test dataset,in order to test our models performance on future unseen data.
7. ROCAUC value for Random Classifier is 0.5 and our models getting values greter than 0.5 so we can say our models are good classifiers.