In [0]:

```
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.svm import SVC
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import SGDClassifier
from gensim.models import Word2Vec
from sklearn.calibration import CalibratedClassifierCV
```

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')
```

In [0]:

```
# 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 [0]:

```
# 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[0]:

```
(364173, 10)
```

In [0]:

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

In [0]:

```
final_filtered_data.shape
```

Out[0]:

```
(364171, 10)
```

In [0]:

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

Out[0]:

```
69.25852107399194
```

In [0]:

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

In [0]:

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

Out[0]:

```
(68356, 10)
```

In [0]:

```
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 [0]:

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

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[0]:

```
True
```

In [0]:

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

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[0]:

```
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 f
eature
# 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 vector
ization

# 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 sea
rch 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 differentl
y
        # 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
```

In [0]:

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

In [0]:

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


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

```
100%|██████████| 68356/68356 [00:27<00:00, 2520.18it/s]
```

```
68356
tried several times get good coconut flavored coffee little success boyer trick great coffee good amount co
conut flavor highly recommend
```

Stemming

In [0]:

```
def do_stemming(processed_data):
    # Before applying BoW or Tfidf featurization techinque on our corpus we need to apply stemmming for each word
in each 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))

    return stemmed_reviews
```

In [0]:

```
stemmed_reviews=do_stemming(processed_data)
```

```
100%|██████████| 68356/68356 [01:06<00:00, 1021.30it/s]
```

Splitting Data In Train ,CV and Test Dataset

In [0]:

```
# 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_hype(train_data,tr_label,penalty='l2',use_kernelization=False):
    para_li = [10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1, 10 ** 0, 10 ** 1, 10 ** 2, 10 ** 3, 10 ** 4]
    parameters=list()
    if use_kernelization:
        model=SVC(cache_size=5000)
        parameters = [{'C': para_li,'class_weight':['balanced']}]
    else:
        model = SGDClassifier()
        parameters = [{'alpha': para_li, 'penalty': [penalty],'class_weight':['balanced']}]

    tbs_cv = TimeSeriesSplit(n_splits=5).split(train_data)
    gsearch = GridSearchCV(estimator=model, cv=tbs_cv,
                    param_grid=parameters, scoring = 'roc_auc',return_train_score=True,n_jobs=-1)
    gsearch.fit(train_data, tr_label)

    if use_kernelization:
        # Caliberating the model
        cccv_gsearch=CalibratedClassifierCV(gsearch,cv='prefit')
        model=cccv_gsearch.fit(train_data, tr_label)
        print("Best C       : ",model.base_estimator.best_estimator_.C)
        print("Best AUC     : ",model.score(train_data, tr_label))
        best_hype=model.base_estimator.best_estimator_.C

    else:
        print("Best alpha   : ",gsearch.best_estimator_.alpha)
        print("Best AUC     : ",gsearch.score(train_data, tr_label))
        best_hype=gsearch.best_estimator_.alpha

    test_auc=gsearch.cv_results_['mean_test_score']
    train_auc=gsearch.cv_results_['mean_train_score']

    para_li=np.log(para_li)
    plt.plot(para_li, test_auc,'bo',linestyle="solid",label='CV AUC')
    plt.plot(para_li, train_auc,'yo',linestyle="solid",label='Train AUC')
    plt.xlabel('log(alpha) values')
    plt.ylabel('AUC')
    plt.legend(loc="upper right")
    plt.grid()
    plt.show()
    return best_hype
```

In [0]:

```
def testing_on_test_data(train_rev,train_label,test_rev,test_label,best_hype,penalty='l2',use_kernelization=False)
:
    plt.figure(1)
    if use_kernelization:
        model=SVC(C=best_hype,class_weight='balanced',probability=True)
        model.fit(train_rev,train_label)
    else:
        sgd = SGDClassifier(alpha=best_hype,penalty=penalty,class_weight='balanced')
        sgd.fit(train_rev,train_label)
        model=CalibratedClassifierCV(sgd,cv='prefit')
        model.fit(train_rev,train_label)

    train_pred = model.predict_proba(train_rev)[:,1]
    test_pred= model.predict_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_hype,penalty='l2',use_kernelization=False)
:
    plt.figure(1,figsize=(15,7))
    np.set_printoptions(precision=5)
    if use_kernelization:
        model=SVC(C=best_hype,class_weight='balanced')
        model.fit(train_rev,train_label)
    else:
        sgd = SGDClassifier(alpha=best_hype,penalty=penalty,class_weight='balanced')
        sgd.fit(train_rev,train_label)
        model=CalibratedClassifierCV(sgd,cv='prefit')
        model.fit(train_rev,train_label)

    train_pred=model.predict(train_rev)
    test_pred=model.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,best_hype,penalty='l2'):
    sgd = SGDClassifier(alpha=best_hype,penalty=penalty)
    sgd.fit(train_rev,labels)
    model=CalibratedClassifierCV(sgd,cv='prefit')
    model.fit(train_rev,labels)
    #this sorts the  features probabilities return index of sorted values

    class_prob_pos= model.calibrated_classifiers_[0].base_estimator.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)
```

SVM using Linear Kernel

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)
bow_fit=bow_count.fit(x_tr)
print("Some Feature names: ",bow_fit.get_feature_names()[:5])
```

```
Some Feature names:  ['abl', 'absolut', 'absorb', 'accept', 'accord']
```

In [0]:

```
#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)
```

```
Shape of transformed train text reviews (54684, 2000)
Shape of transformed test text reviews (13672, 2000)
```

In [0]:

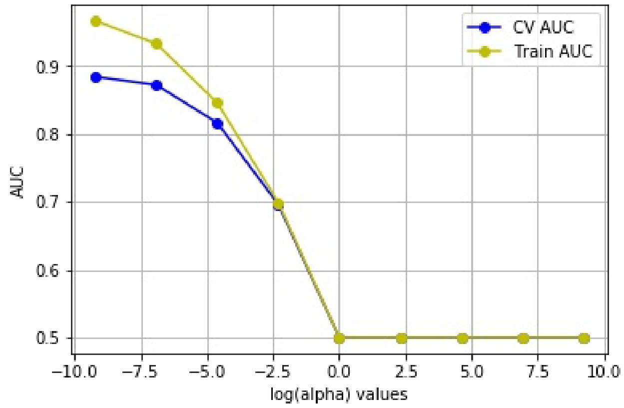
```
# converting sparse matrix to dense matrix before doing standardization
bow_dense_train_reviews=bow_train.toarray()
bow_dense_test_reviews=bow_test.toarray()
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()
std_train_data=std_data.fit_transform(bow_dense_train_reviews*1.0)
std_test_data=std_data.transform(bow_dense_test_reviews*1.0)
```

Applying L1 regularization on BOW

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=False,penalty='l1')
```

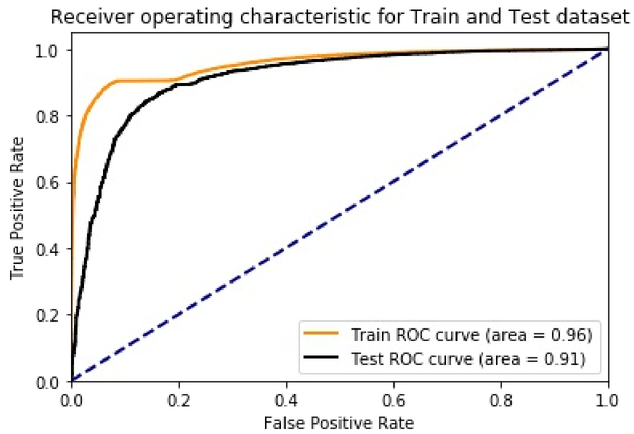
```
Best alpha    :  0.0001
Best AUC      :  0.9570288195807101
```



Testing with Test Data

In [0]:

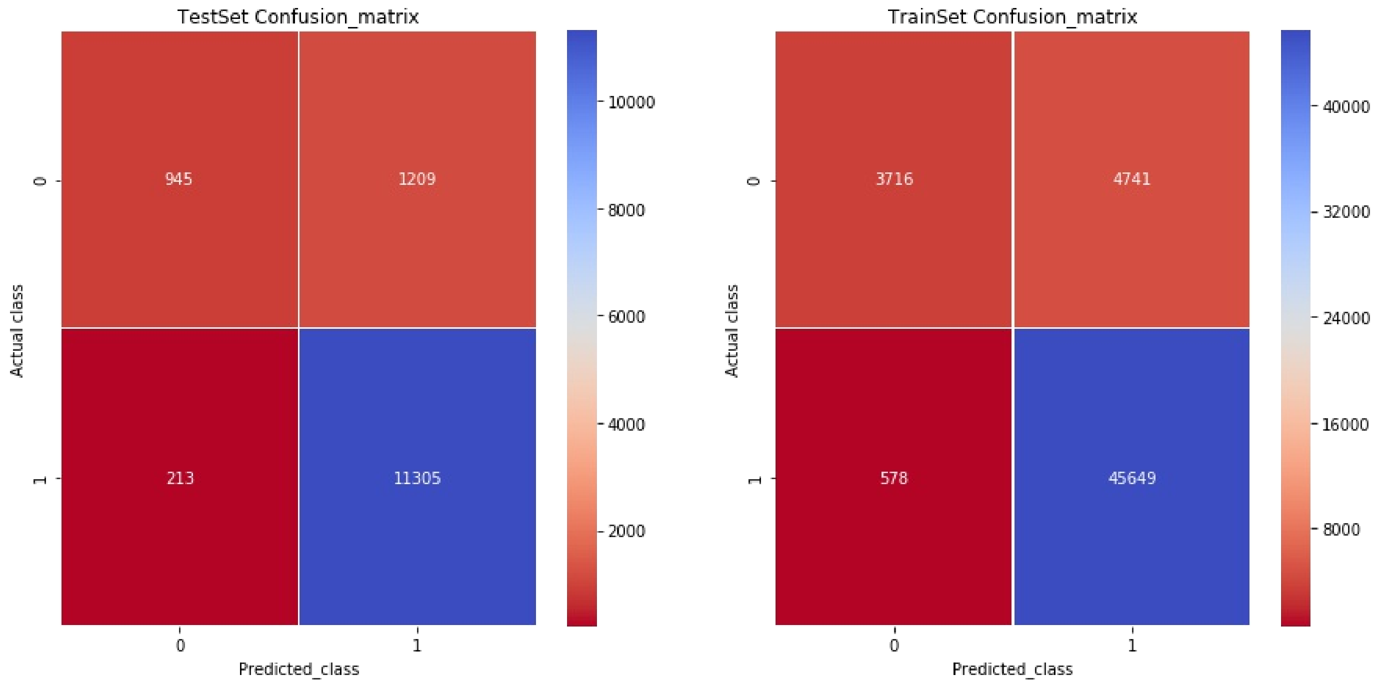
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,0.0001,use_kernelization=False,penalty='l1')
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,0.0001,use_kernelization=False,penalty='l1')
```



Top 10 positive and negative features using l1 regularizer

In [0]:

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

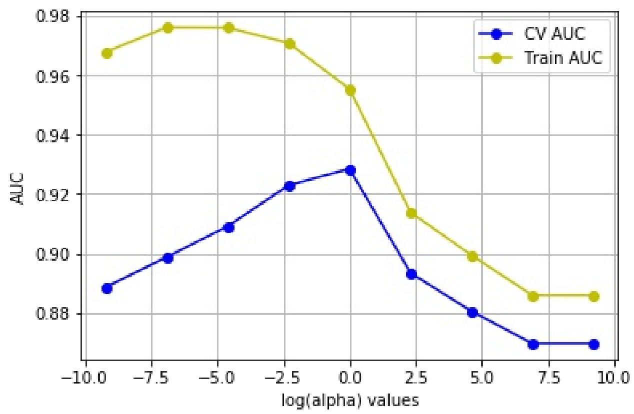
```
Top Ten Positive Features:  ['perfect' 'amaz' 'excel' 'good' 'film' 'nabisco' 'love' 'delici' 'best'
 'great']
Top Ten Negative Features:  ['not' 'disappoint' 'worst' 'jasmin' 'tast' 'aw' 'unfortun' 'terribl'
 'old' 'thought']
```

Applying L2 regularization on BOW

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=False,penalty='l2')
```

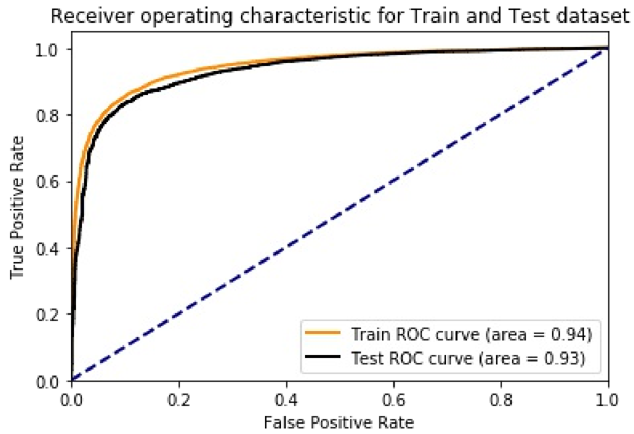
```
Best alpha    :  1
Best AUC      :  0.9448689744535055
```



Testing with Test Data

In [0]:

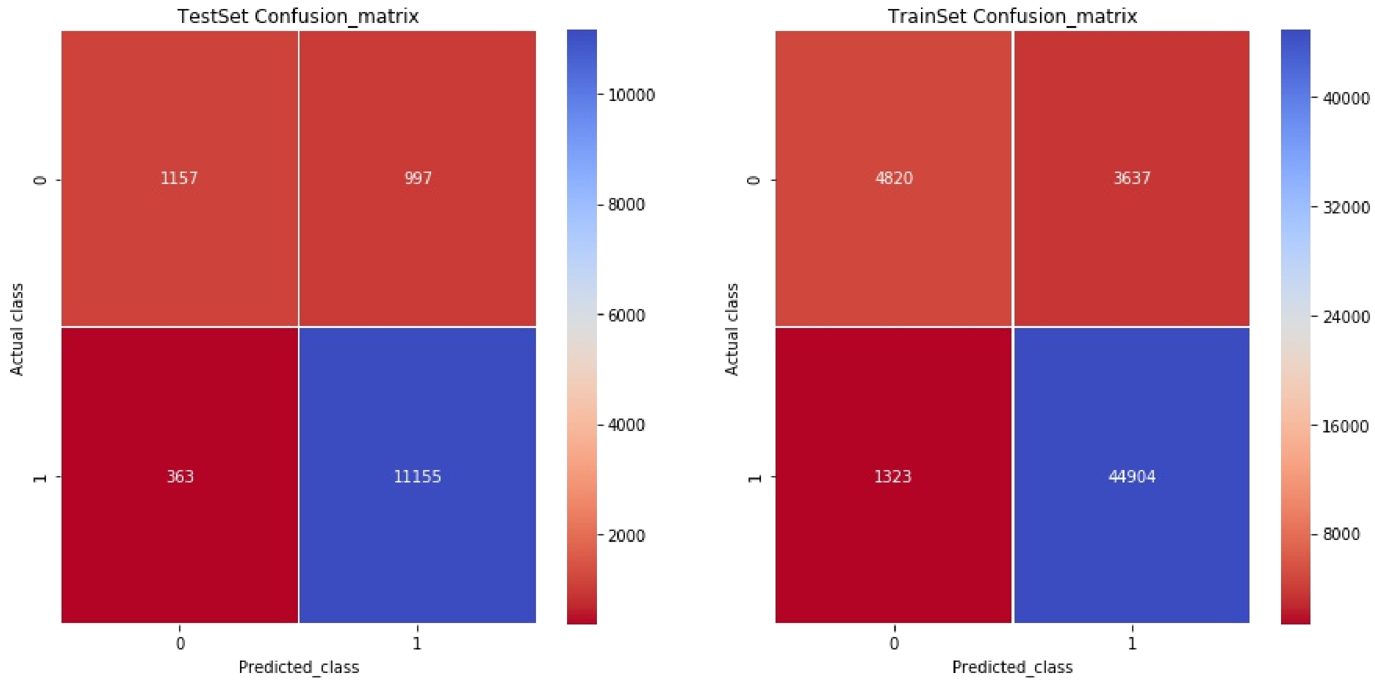
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=False,penalty='l2')
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=False,penalty='l2')
```



Top 10 positive and negative features using l2 regularizer

In [0]:

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

```
Top Ten Positive Features:  ['easi' 'keep' 'favorit' 'nice' 'good' 'perfect' 'delici' 'best' 'love'
 'great']
Top Ten Negative Features:  ['disappoint' 'worst' 'return' 'not' 'terribl' 'money' 'aw' 'wast' 'bad'
 'horribl']
```

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]:

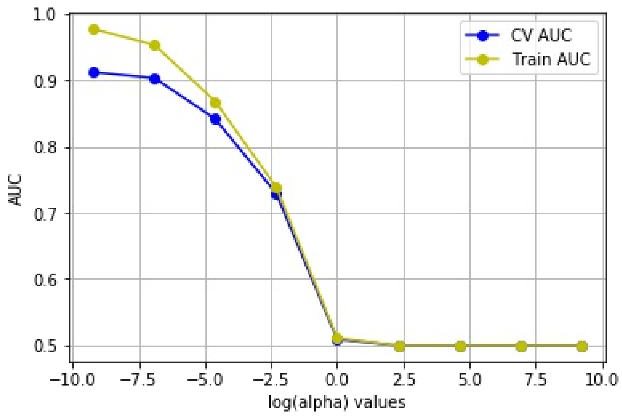
```
# converting sparse matrix to dense matrix before doing standardization
tfidf_dense_train_reviews=tfidf_tr.toarray()
tfidf_dense_test_reviews=tfidf_test.toarray()

# 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)
```

Applying L1 regularization on TFIDF

In [0]:

```
best_hype= find_best_hype(std_train_data,y_tr,penalty='l1',use_kernelization=False)
```

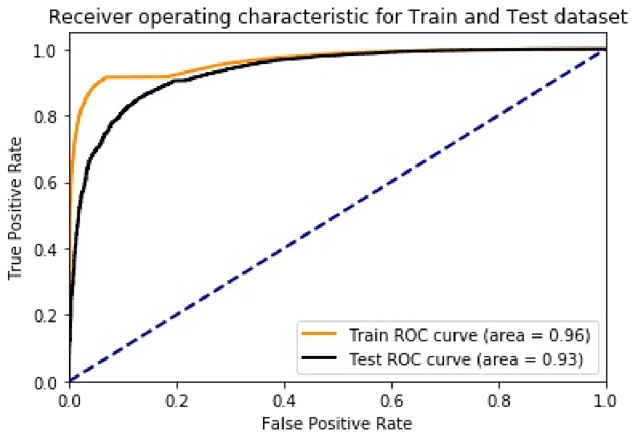
```
Best alpha    :  0.0001
Best AUC      :  0.9637526884792418
```



Testing with Test Data

In [0]:

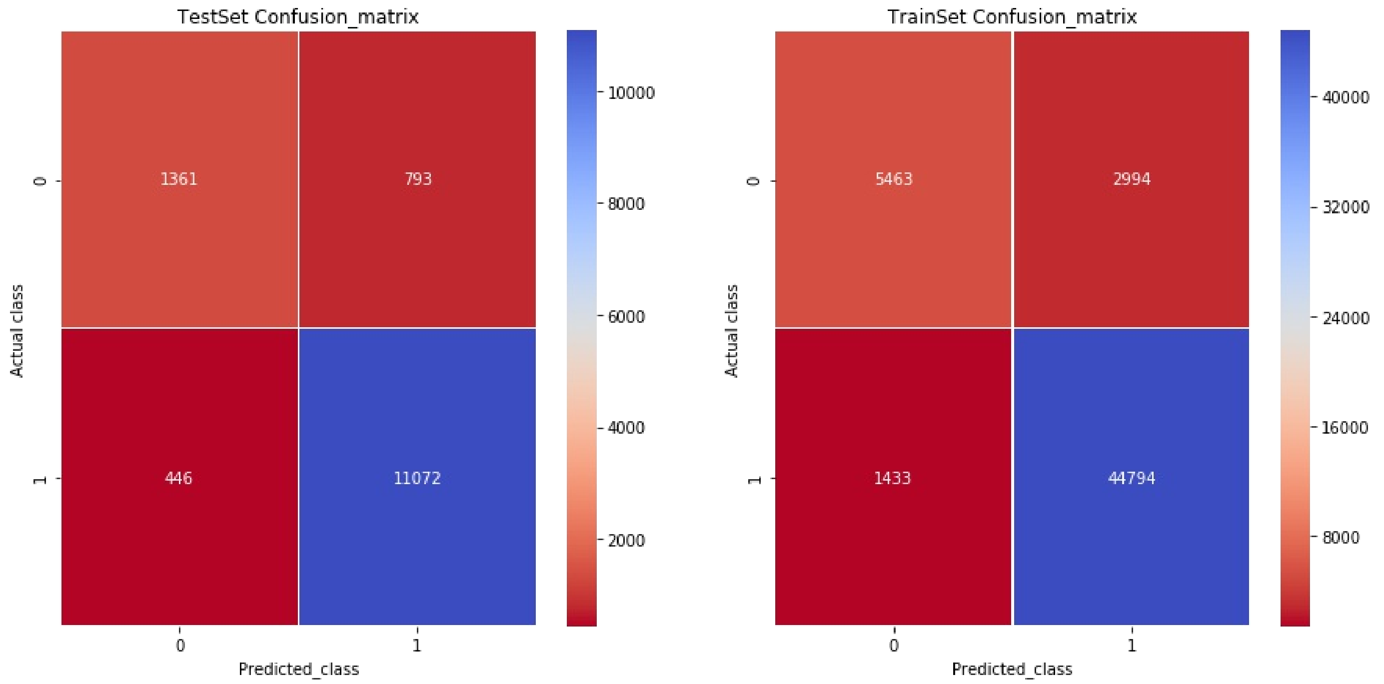
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=False,penalty='l1')
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=False,penalty='l1')
```



Top 10 positive and negative features using l1 regularizer

In [0]:

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

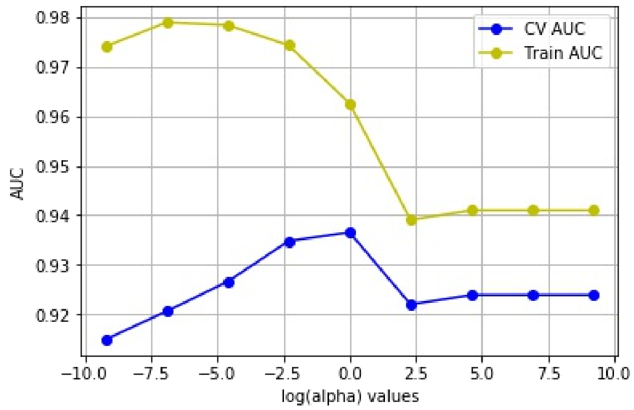
```
Top Ten Positive Features:  ['high recommend' 'even better' 'good' 'perfect' 'not disappoint' 'delici'
 'excel' 'love' 'best' 'great']
Top Ten Negative Features:  ['disappoint' 'not' 'worst' 'not worth' 'not recommend' 'aw' 'unfortun'
 'not good' 'old' 'return']
```

Applying L2 regularization on TFIDF

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=False,penalty='l2')
```

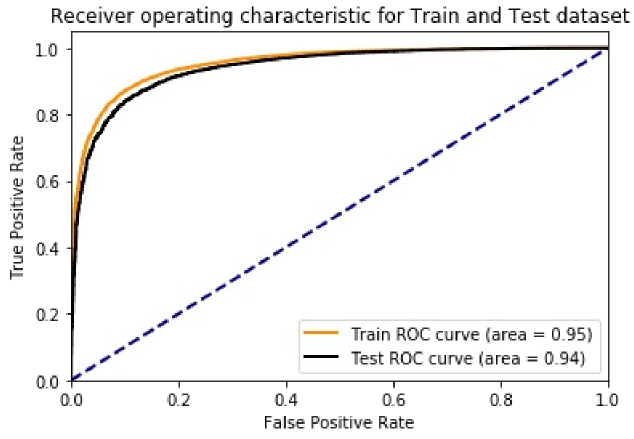
```
Best alpha    :  1
Best AUC      :  0.9531899943791879
```



Testing with Test Data

In [0]:

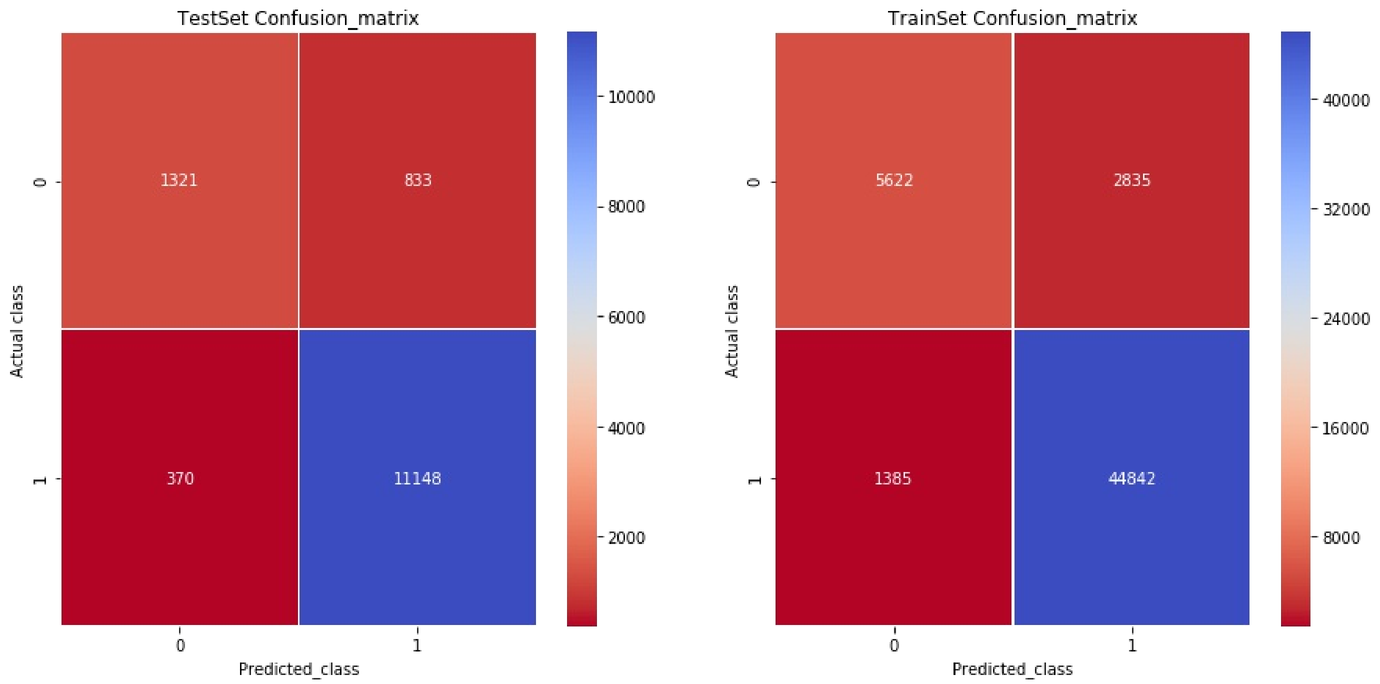
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=False,penalty='l2')
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=False,penalty='l2')
```



Top 10 positive and negative features using l2 regularizer

In [0]:

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

```
Top Ten Positive Features:  ['keep' 'high recommend' 'nice' 'favorit' 'good' 'perfect' 'delici' 'best'
 'love' 'great']
Top Ten Negative Features:  ['disappoint' 'not' 'return' 'worst' 'terribl' 'not buy' 'aw' 'horribl'
 'wast money' 'not recommend']
```

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())
# 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)
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(300) # 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 [0]:

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

```
100%|██████████| 54684/54684 [01:08<00:00, 798.81it/s]
```

In [0]:

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

```
100%|██████████| 13672/13672 [00:17<00:00, 798.73it/s]
```

In [0]:

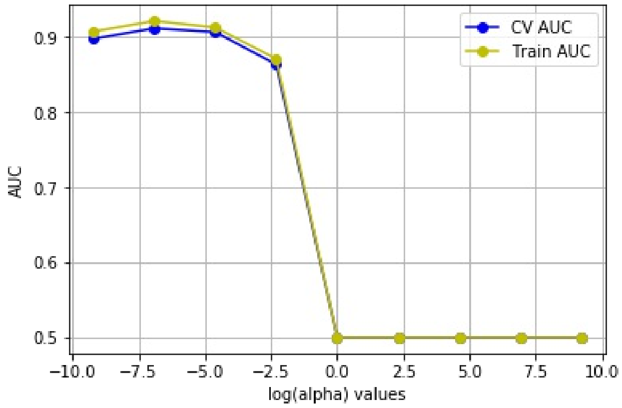
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()
std_train_data=std_data.fit_transform(train_avgw2v)
std_test_data=std_data.transform(test_avgw2v)
```

Applying L1 regularization on Avg Word2Vec

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,penalty='l1')
```

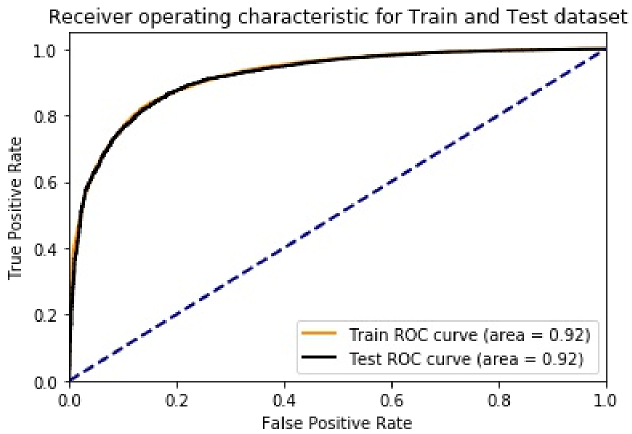
```
Best alpha   :  0.001
Best AUC     :  0.9199968527791298
```



Testing with Test Data

In [0]:

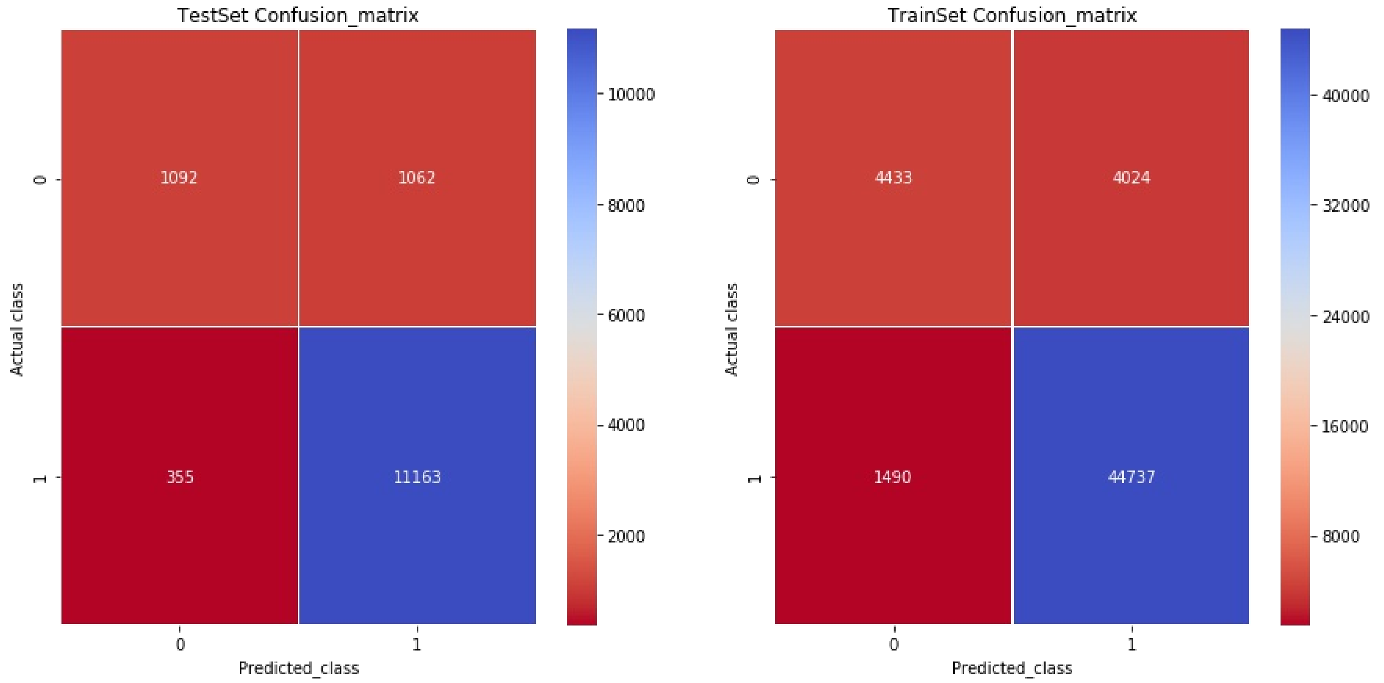
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype=best_hype,penalty='l1')
```



Confusion Matrix

In [0]:

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

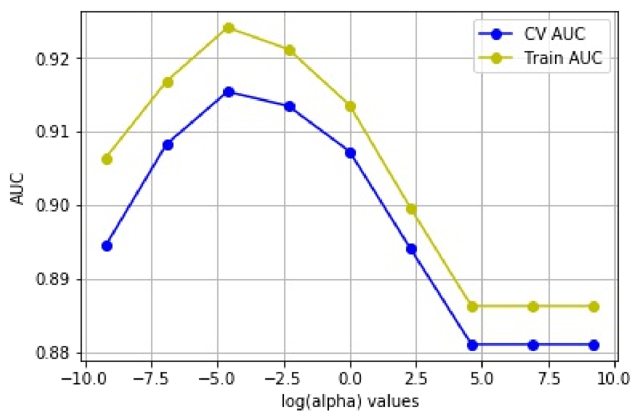


Applying L2 regularization on Avg Word2Vec

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,penalty='l2')
```

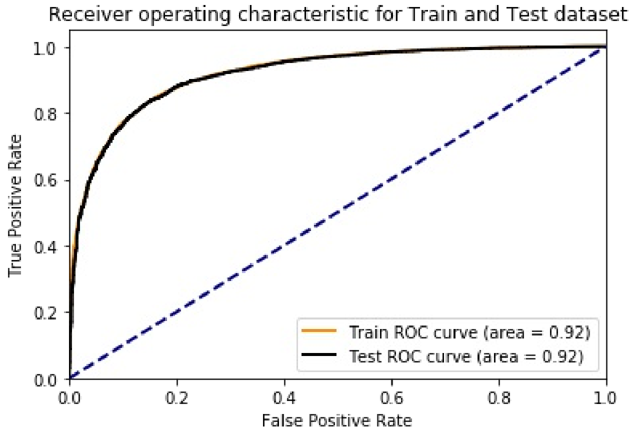
```
Best alpha   :  0.01
Best AUC     :  0.9214860567753294
```



Testing with Test Data

In [0]:

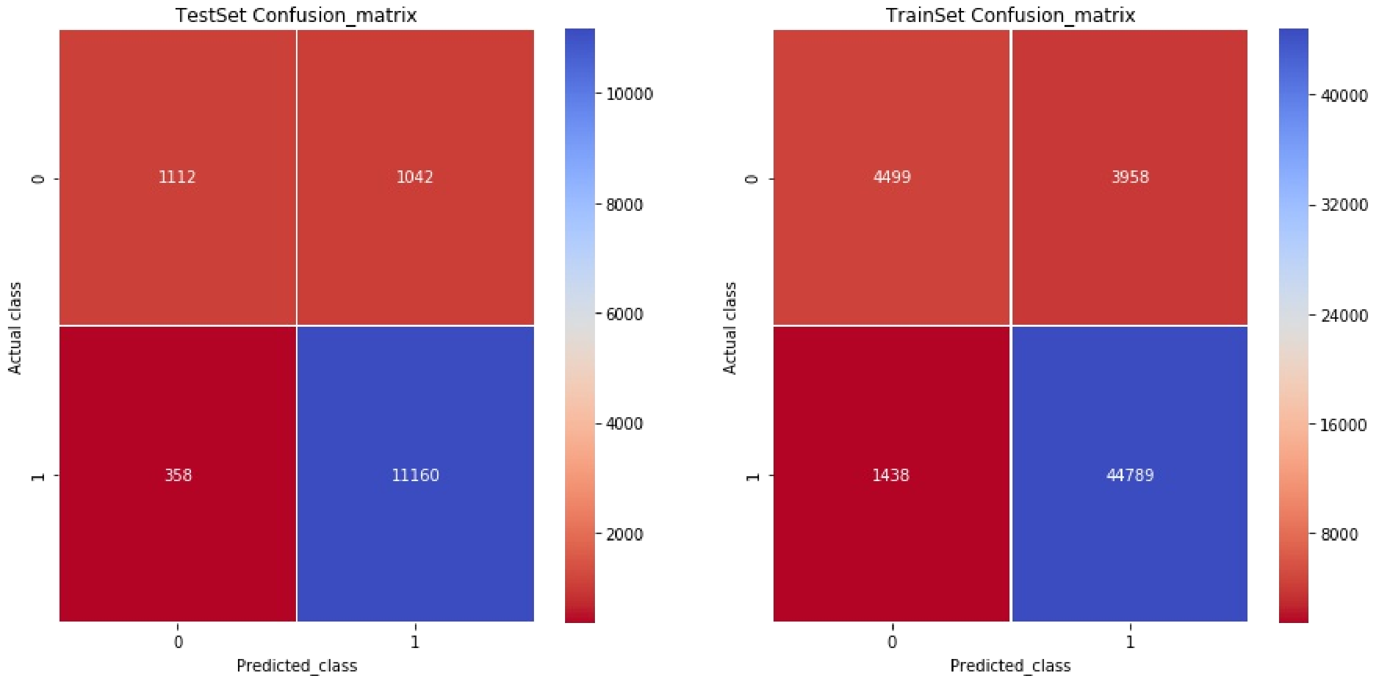
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype=best_hype,penalty='l2')
```



Confusion Matrix

In [0]:

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



4. TFIDF weighted W2Vec

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 [0]:

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

```
100%|██████████| 54684/54684 [01:23<00:00, 655.62it/s]
```

In [0]:

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

```
100%|██████████| 13672/13672 [00:21<00:00, 650.12it/s]
```

In [0]:

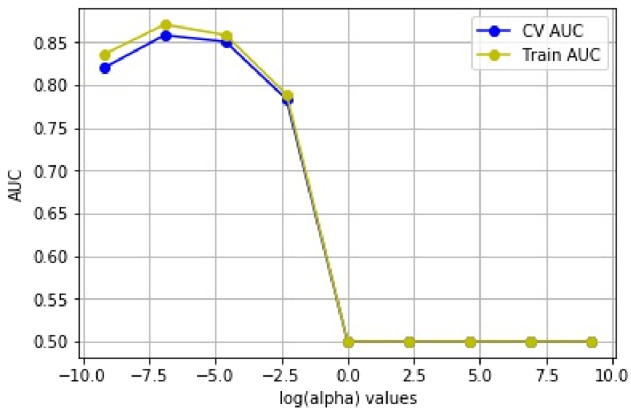
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()
std_train_data=std_data.fit_transform(train_tfidf_w2v)
std_test_data=std_data.transform(test_tfidf_w2v)
```

Applying Logistic Regression with L1 regularization on TFIDF weighted W2Vec

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,penalty='l1')
```

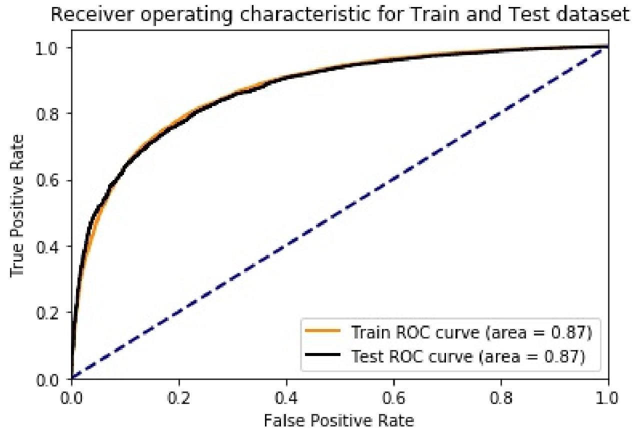
```
Best alpha    :  0.001
Best AUC      :  0.8687439447339237
```



Testing with Test Data

In [0]:

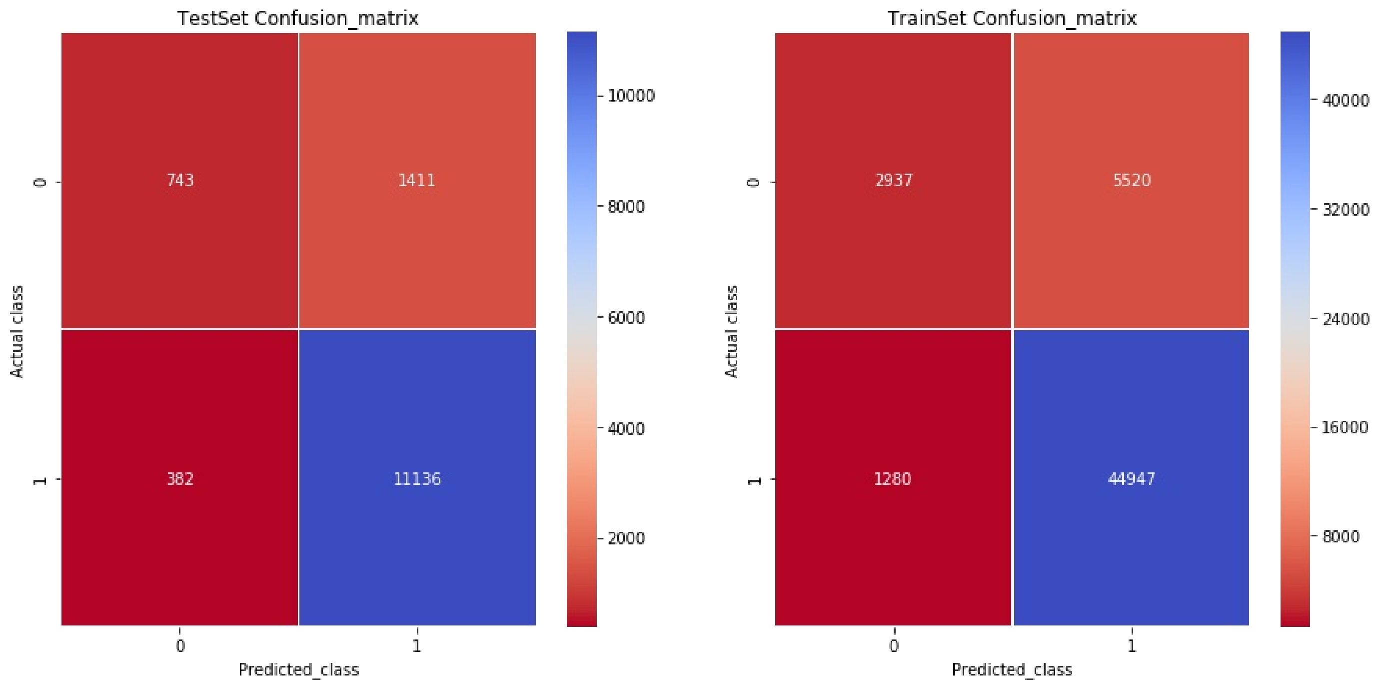
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype=best_hype,penalty='l1')
```



Confusion Matrix

In [0]:

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

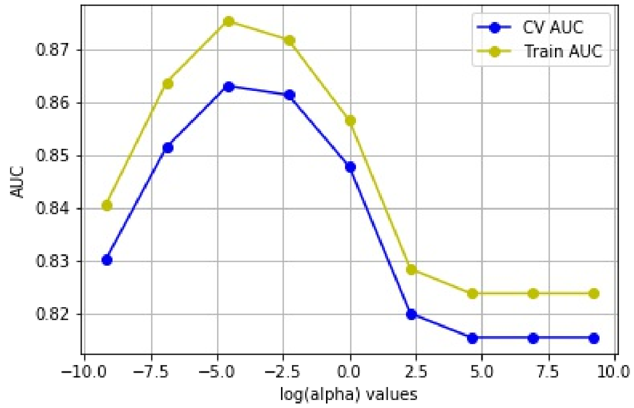


Applying Linear SVM with L2 regularization on TFIDF weighted W2Vec

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,penalty='l2')
```

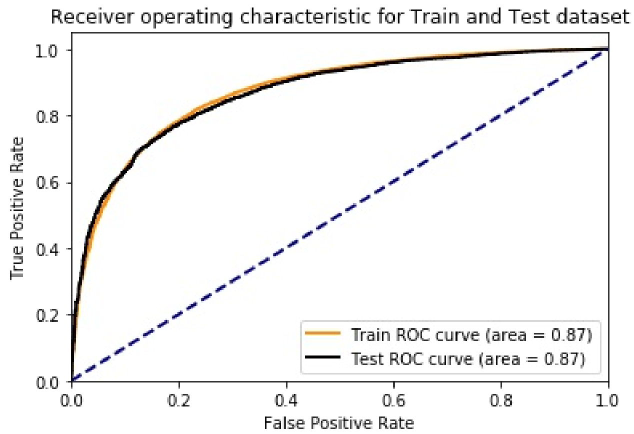
```
Best alpha    :  0.01
Best AUC      :  0.8720065408518582
```



Testing with Test Data

In [0]:

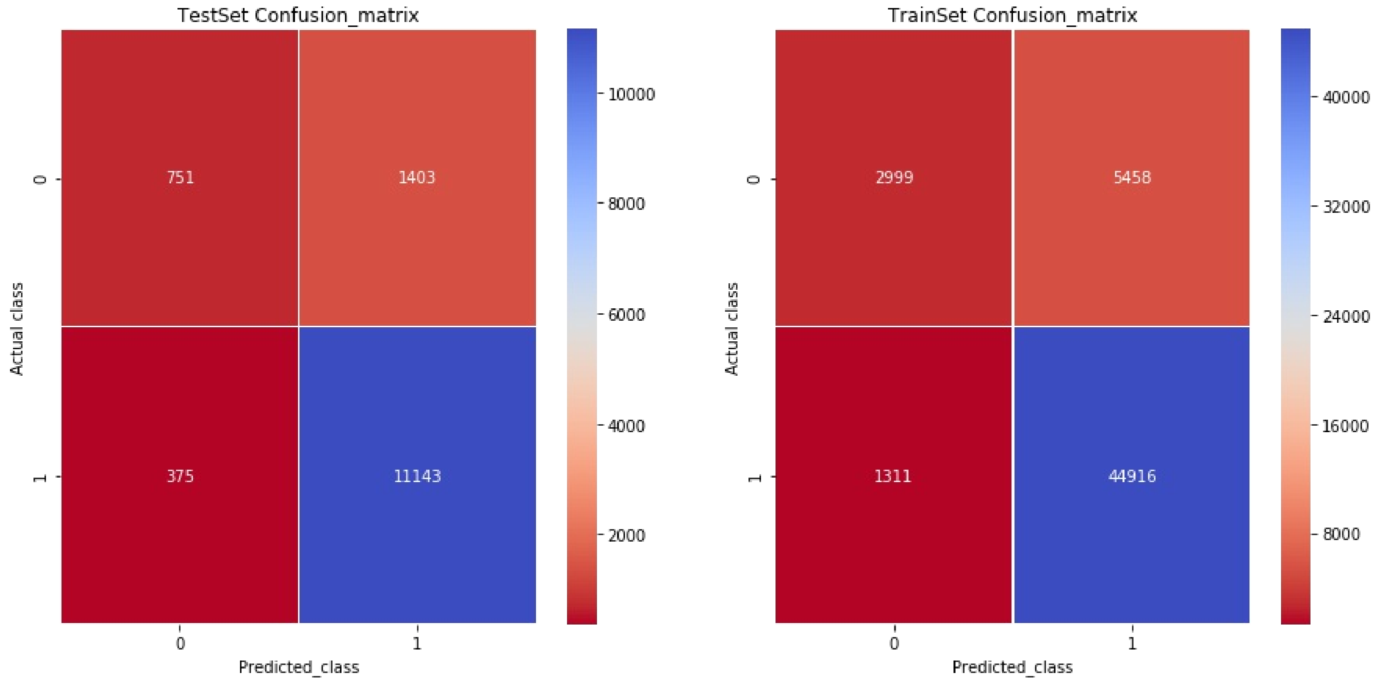
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype=best_hype,penalty='l2')
```



Confusion Matrix

In [0]:

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



SVM RBF Kernel

In [0]:

```
final=filtered_data.sample(frac=0.03,random_state=2)
final.shape
```

Out[0]:

```
(15774, 10)
```

In [0]:

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

```
Positive Reviews:  13265
Positive Reviews:  2509
```

Text Preprocessing

In [0]:

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

In [0]:

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


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

```
100%|██████████| 15774/15774 [00:06<00:00, 2523.10it/s]

15774
tried several times get good coconut flavored coffee little success boyer trick great coffee good amount co
conut flavor highly recommend
```

Stemming

In [0]:

```
stemmed_reviews=do_stemming(processed_data)
```

```
100%|██████████| 15774/15774 [00:15<00:00, 1025.30it/s]
```

Splitting Data In Train ,CV and Test Dataset

In [0]:

```
# 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: 12619 , 3155
```

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=500)
bow_fit=bow_count.fit(x_tr)
print("Some Feature names: ",bow_fit.get_feature_names()[:5])
```

```
Some Feature names:  ['abl', 'absolut', 'acid', 'actual', 'ad']
```

In [0]:

```
#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)
```

```
Shape of transformed train text reviews (12619, 500)
Shape of transformed test text reviews (3155, 500)
```

In [0]:

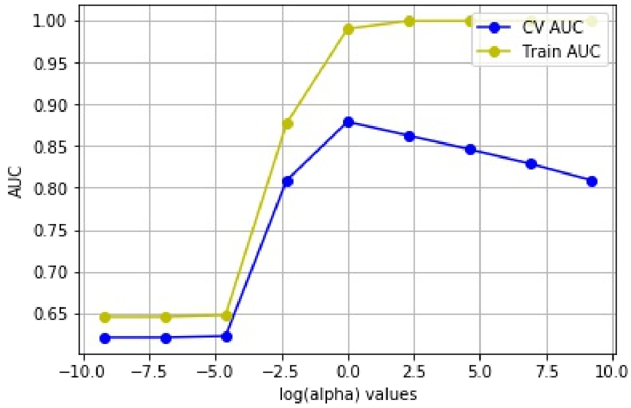
```
# converting sparse matrix to dense matrix before doing standardization
bow_dense_train_reviews=bow_train.toarray()
bow_dense_test_reviews=bow_test.toarray()
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()
std_train_data=std_data.fit_transform(bow_dense_train_reviews*1.0)
std_test_data=std_data.transform(bow_dense_test_reviews*1.0)
```

Finding Best Hyperparameters

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=True)
```

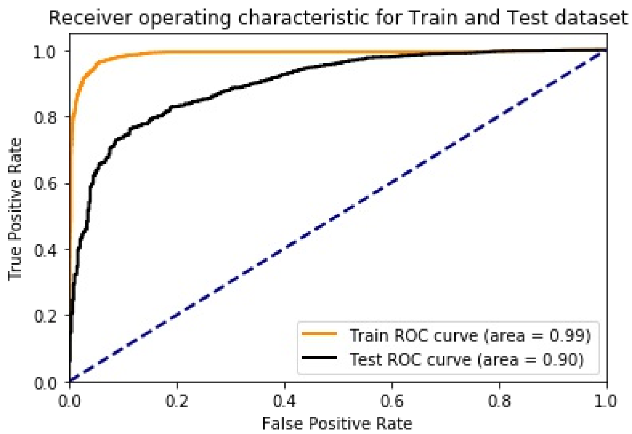
```
Best C       :  1
Best AUC     :  0.966716855535304
```



Testing with Test Data

In [0]:

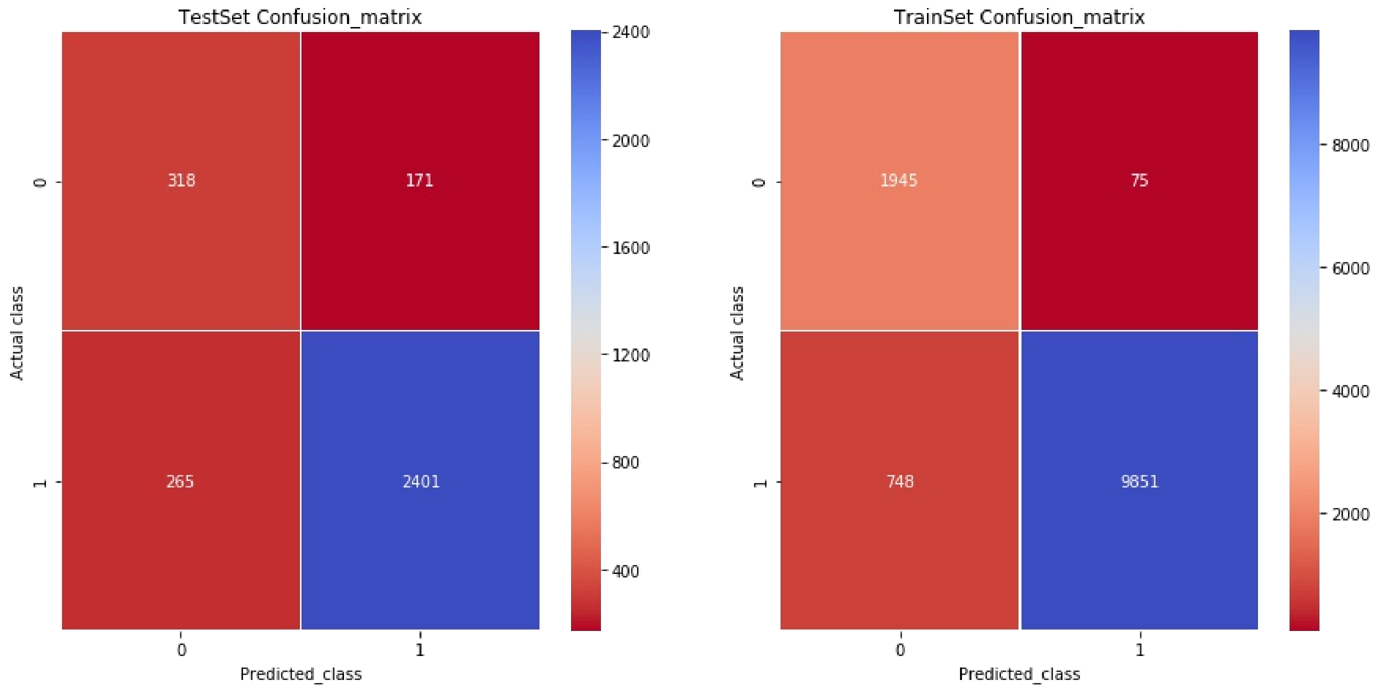
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



2.TFIDF

In [0]:

```
tfidf_count= TfidfVectorizer(min_df=10,max_features=300,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 : (12619, 300)
Shape of tfidf vector representation of test review text  : (3155, 300)
```

In [0]:

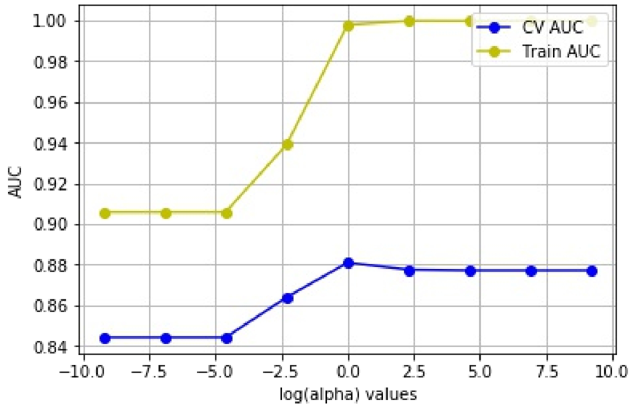
```
# converting sparse matrix to dense matrix before doing standardization
tfidf_dense_train_reviews=tfidf_tr.toarray()
tfidf_dense_test_reviews=tfidf_test.toarray()

# 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)
```

Finding Best Hyperparameters

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=True)
```

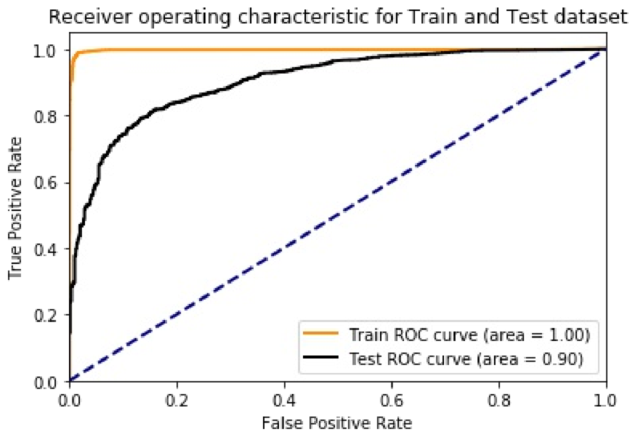
```
Best C       :  1
Best AUC     :  0.9878754259450035
```



Testing with Test Data

In [0]:

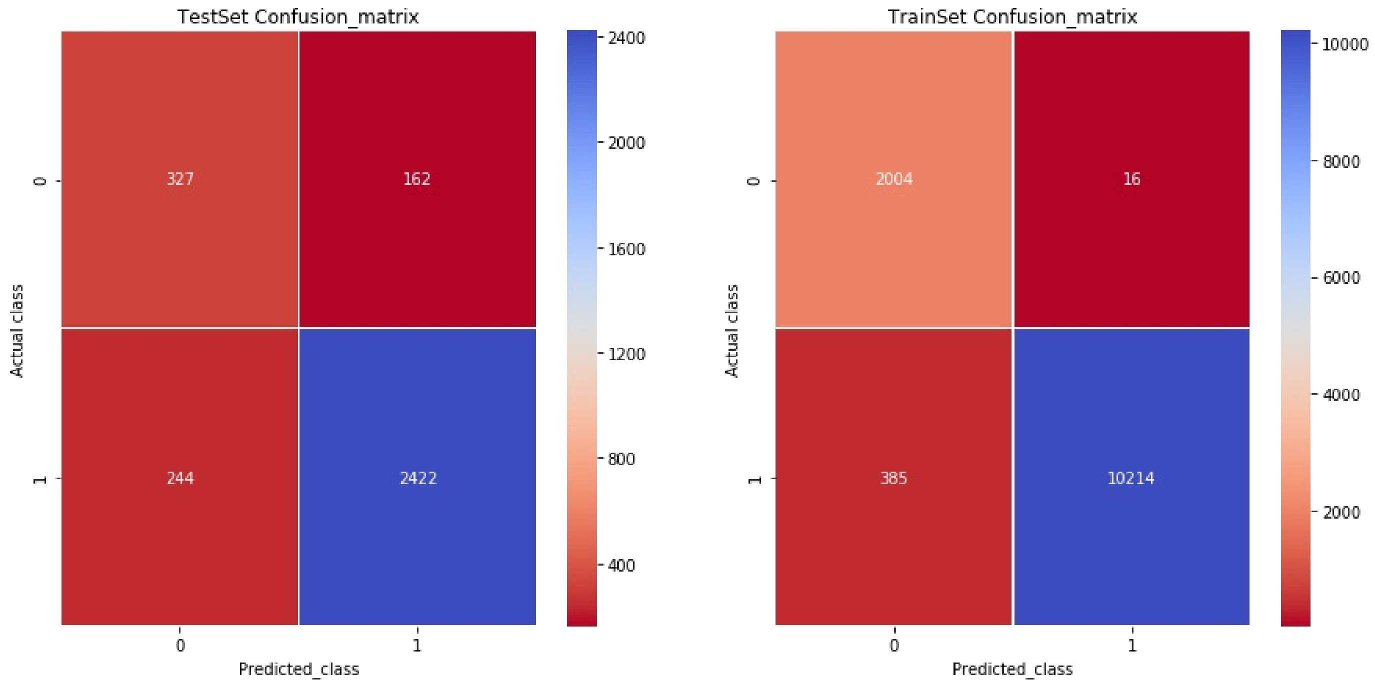
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



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())
# 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)
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(300) # 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 [0]:

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

```
100%|██████████| 12619/12619 [00:10<00:00, 1252.64it/s]
```

In [0]:

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

```
100%|██████████| 3155/3155 [00:02<00:00, 1211.41it/s]
```

In [0]:

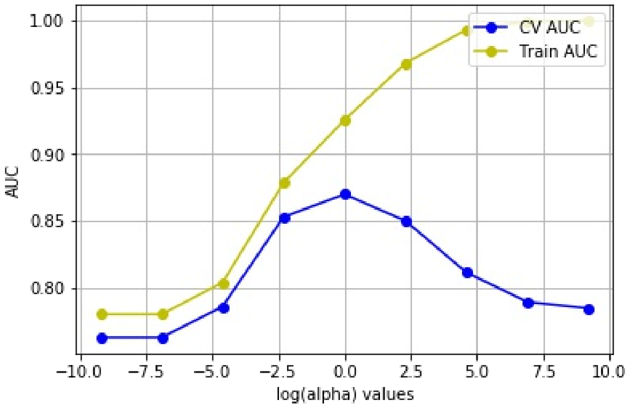
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()
std_train_data=std_data.fit_transform(train_avgw2v)
std_test_data=std_data.transform(test_avgw2v)
```

Finding Best Hyperparameter

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=True)
```

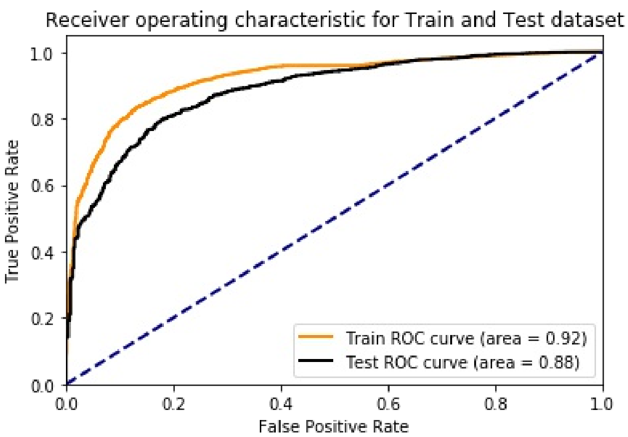
```
Best C       :  1
Best AUC     :  0.9007052856803234
```



Testing with Test Data

In [0]:

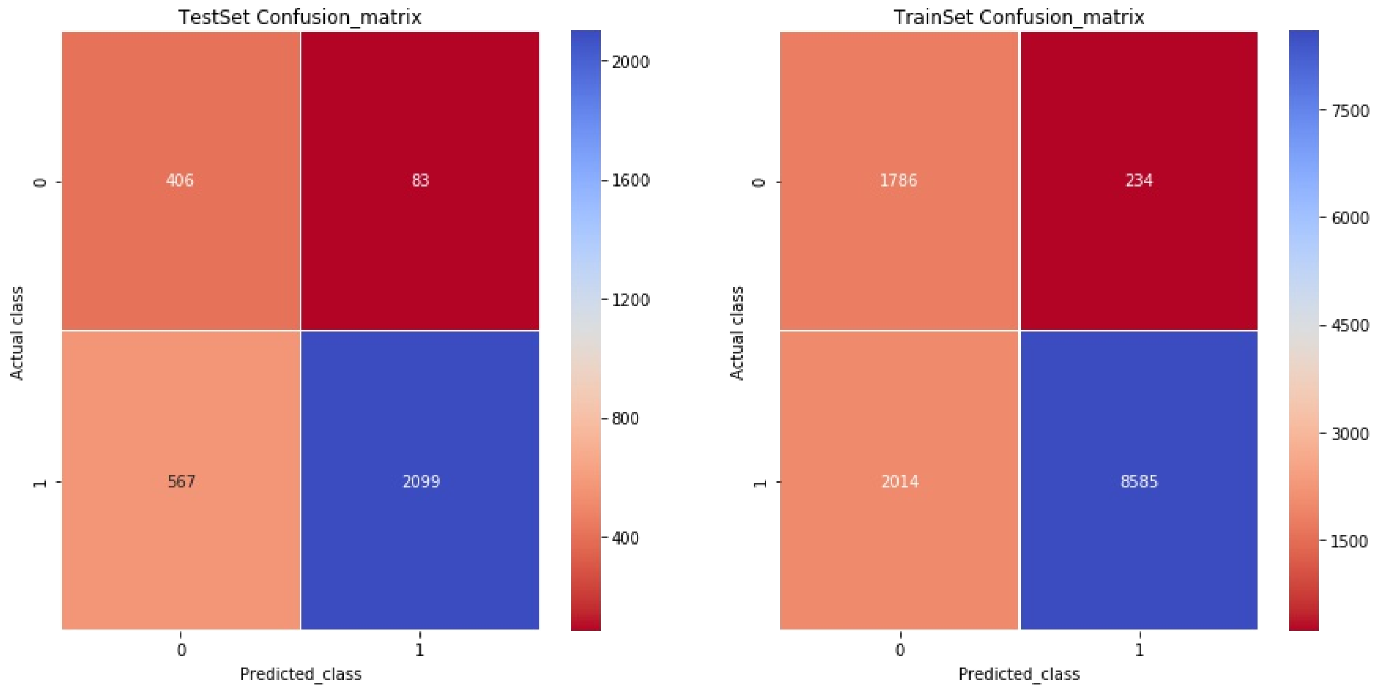
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



4. TFIDF weighted W2Vec

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 [0]:

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

```
100%|██████████| 12619/12619 [00:13<00:00, 906.70it/s]
```

In [0]:

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

```
100%|██████████| 3155/3155 [00:03<00:00, 831.84it/s]
```

In [0]:

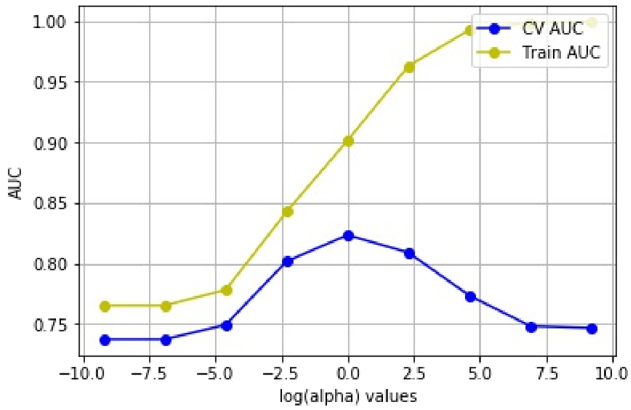
```
# Apply standardization on train,test and cv dataset
std_data=StandardScaler()
std_train_data=std_data.fit_transform(train_tfidf_w2v)
std_test_data=std_data.transform(test_tfidf_w2v)
```

Finding Best Hyperparameters

In [0]:

```
best_hype=find_best_hype(std_train_data,y_tr,use_kernelization=True)
```

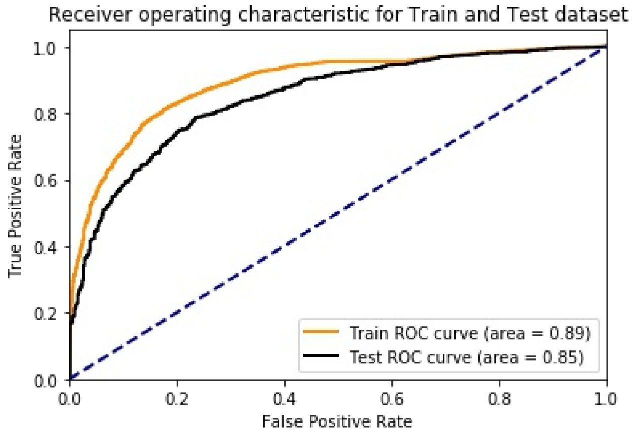
```
Best C      :  1
Best AUC    :  0.8624296695459228
```



Testing with Test Data

In [0]:

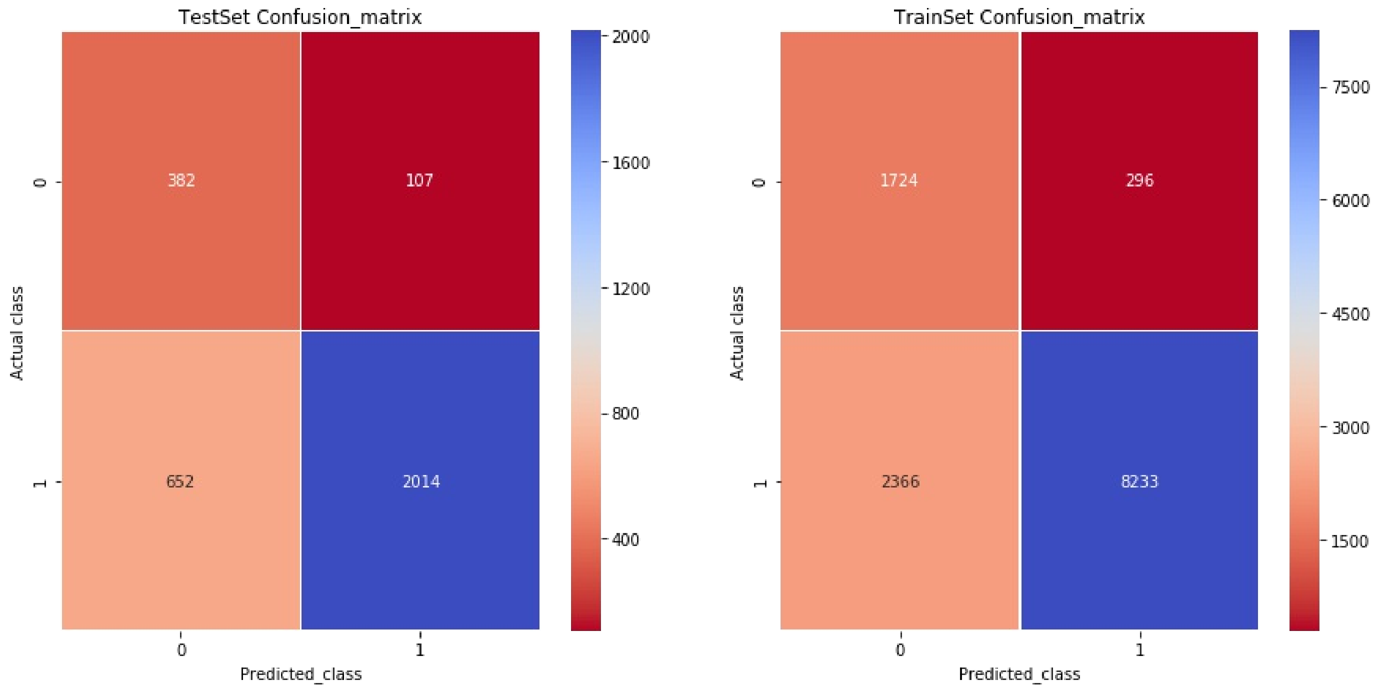
```
testing_on_test_data(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



Confusion Matrix

In [0]:

```
get_confusion_matrix(std_train_data,y_tr,std_test_data,y_test,best_hype,use_kernelization=True)
```



In [0]:

```
print("""

+++++++++++++++++++++++++++++++++++ Linear SVM +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

          |  Vectorizer     | HyperParameter'('C')'     |   Regularization      |         AUC         |
          ______________________________________________________________________________________________

              BoW           |       0.0001              |      L1               |  0.9570288195807101 |
              BoW           |       1                   |      L2               |  0.9448689744535055 |

            tfidf           |       0.0001              |      L1               |  0.9637526884792418 |
            tfidf           |       1                   |      L2               |  0.9531899943791879 |

         Avg tfidf          |       0.001               |      L1               |  0.9199968527791298 |
         Avg tfidf          |       0.01                |      L2               |  0.9214860567753294 |

     tfidf Weighted W2v     |       0.001               |      L1               |  0.8687439447339237 |
     tfidf Weighted W2v     |       0.01                |      L2               |  0.8720065408518582 |

++++++++++++++++++++++++++++++++++++ RBF Kernel ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

          |  Vectorizer     |   HyperParameter'('C')'   |   Kernelization       |         AUC         |
          ______________________________________________________________________________________________

              BoW           |          1                |      rbf              |  0.966716855535304  |

            tfidf           |          1                |      rbf              |  0.9878754259450035 |

         Avg tfidf          |          1                |      rbf              |  0.9007052856803234 |

     tfidf Weighted W2v     |          1                |      rbf              |  0.8624296695459228 |


             """)
```

++++++++++++++++++++++++++++++++++ Linear SVM ++

Vectorizer	HyperParameter'('C')'	Regularization	AUC
BoW	0.0001	L1	0.9570288195807101
BoW	1	L2	0.9448689744535055
tfidf	0.0001	L1	0.9637526884792418
tfidf	1	L2	0.9531899943791879
Avg tfidf	0.001	L1	0.9199968527791298
Avg tfidf	0.01	L2	0.9214860567753294
tfidf Weighted W2v	0.001	L1	0.8687439447339237
tfidf Weighted W2v	0.01	L2	0.8720065408518582

++++++++++++++++++++++++++++++++++ RBF Kernel ++
++

Vectorizer	HyperParameter'('C')'	Kernelization	AUC
BoW	1	rbf	0.966716855535304
tfidf	1	rbf	0.9878754259450035
Avg tfidf	1	rbf	0.9007052856803234
tfidf Weighted W2v	1	rbf	0.8624296695459228

Conclusion

- Support Vector Machines (SVM) defines a decision boundry in such a way that the distance between the datapoints and decision surface is maximum
- Two type of classification:

1. Hard SVM:

```
Here it is necessary to have perfectly linearly seperable class label, thus no bias variance tradeoff
```

2. Soft SVM Classification:

```
Can work with almost linearly seperable data and can be achieved solving Primal Form of SVM optimization problem
```

- Primal form of SVM optimizaiton problem enables us to minimize the misclassification error for almost linearly seperable data by minimizing hinge loss; it is also called as linear SVM it is as same as logistic regression withoout feature engineering.
- For non-linearly seperable data, dual form of SVM optimization problem gives excellent solution by enabling us to use Kernel Trick.
- In Logistic regression when we do feature engineering we actually convert features to high dimension but in SVM when we use Kernel Trick it implicitly converts features to higher dimensions and using that it construts decision boundry for our data.