Import necessary libraries

In [3]:

```
import warnings
warnings.filterwarnings('ignore')
```

In [4]:

```
import xgboost as xgb
#IMPORTING XGBOOST WRAPPER OF SKLEARN
from xgboost.sklearn import XGBClassifier
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import TimeSeriesSplit
from scipy.sparse import *
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import *
import pickle
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from prettytable import PrettyTable
from wordcloud import WordCloud
```

Load preprocessed data

In [37]:

```
#Functions to save objects for later use and retireve it
def savetofile(obj,filename):
    pickle.dump(obj,open(filename+".pkl","wb"))
def openfromfile(filename):
    temp = pickle.load(open(filename+".pkl","rb"))
    return temp
y_train =openfromfile('y_train')
y_test =openfromfile('y_test')

count_vect =openfromfile('count_vect')
X_train_bigram = openfromfile('X_train_bigram')
X_test_bigram = openfromfile('X_test_bigram')

tf_idf_vect =openfromfile('tf_idf_vect')
X_train_tfidf =openfromfile('X_train_tfidf')
X_test_tfidf =openfromfile('X_test_tfidf')

avg_sent_vectors=openfromfile('avg_sent_vectors')
avg_sent_vectors_test=openfromfile('avg_sent_vectors_test')

tfidf_sent_vectors=openfromfile('tfidf_sent_vectors')
tfidf_sent_vectors_test=openfromfile('tfidf_sent_vectors_test')
```

Standardizing data

In [6]:

```
def std_data(train,test,mean):
    scaler=StandardScaler(with_mean=mean)
    std_train=scaler.fit_transform(train)
    std_test=scaler.transform(test)
    return std_train, std_test
```

Observation:

1. In Decision Tree based algorithm we are not dealing with distance at all.
2. So, Data Standardization is not required for DecisionTree, GBDT and Random-Forest.

Ensemble Models

Function for hyperparameter tunning using corss validation and error plot using heatmap:

In [7]:

```
# find Optimal value of hyperparam by TimeSeriesSplit and 10_fold_cross _validation
# using RandomizedSearchCV and GridSearchCV.
def Ensemble_Classifier(x_train,y_train,TBS,params,searchMethod,vect,classifier):
    if classifier=='random_forest':
        #INITIALIZE RANDOM-FOREST CLASSIFIER
        clf=RandomForestClassifier(class_weight='balanced',\
                                   criterion='gini',\
                                   oob_score=True)
    elif classifier =='gbdt':
        #INITIALIZE GBDT CLASSIFIER
        clf=xgb.XGBClassifier(nthread=8,\
                              learning_rate=.1,\
                              gamma=0,\
                              subsample=.8,\
                              colsample_bytree=.8,\
                              booster='gbtree',\
                              objective='binary:logistic')
    # APPLY RANDOM OR GRID SEARCH FOR HYPERPARAMETER TUNNING
    if searchMethod=='grid':
        model=GridSearchCV(clf,\
                           cv=TBS,\
                           n_jobs=-1,\
                           param_grid=params,\
                           return_train_score=False,\
                           scoring=make_scorer(roc_auc_score,average='weighted'))
        model.fit(x_train,y_train)
    elif searchMethod=='random':
        model=RandomizedSearchCV(clf,\
                                 n_jobs=-1,\
                                 cv=TBS,\
                                 param_distributions=params,\
                                 n_iter=len(params['max_depth']),\
                                 return_train_score=False,\
                                 scoring=make_scorer(roc_auc_score,average='weighted'))
        model.fit(x_train,y_train)
    #PLOT THE PERFORMANCE OF MODEL ON CROSSVALIDATION DATA FOR EACH HYPERPARAM VALUE
    cv_auc=model.cv_results_['mean_test_score']
    cv_auc=np.array(cv_auc).reshape(len(params['max_depth']),len(params['n_estimators']))
    cv_auc_df=pd.DataFrame(cv_auc, np.array(params['max_depth']),np.array(params['n_estimators']))
    cv_auc_df=cv_auc_df.round(4)
    plt.figure(figsize= (10,8))
```

```
    plt.title('Hyperparam Tunning Results(%s)' %vect)
    sns.set(font_scale=1.4)#for label size
    ax=sns.heatmap(cv_auc_df, annot=True,annot_kws={"size": 15}, fmt='g',)
    ax.set(xlabel='No. of Estimators', ylabel='Depth-Values')
    plt.show()
    return model
```

Function which calculate performance on test data with optimal hyperparam :

In [8]:

```
def test_performance(x_train,y_train,x_test,y_test,optimal,vect,summarize,classifier):
    '''FUNCTION FOR TEST PERFORMANCE(PLOT ROC CURVE FOR BOTH TRAIN AND TEST) WITH OPTIMAL HYPERPARAM'''
    if classifier=='random_forest':
        #INITIALIZE RANDOM FOREST WITH OPTIMAL VALUE OF HYPERPARAMS
        clf=RandomForestClassifier(n_estimators=optimal['n_estimators'],\
                                   max_depth=optimal['max_depth'],\
                                   class_weight='balanced',\
                                   criterion='gini',\
                                   oob_score=True,\
                                   n_jobs=-1)
    elif classifier=='gbdt':
        #INITIALIZE GBDT WITH OPTIMAL VALUE OF HYPERPARAMS
        clf=xgb.XGBClassifier(n_estimators=optimal['n_estimators'],\
                              max_depth=optimal['max_depth'],\
                              nthread=8,\
                              learning_rate=.1,\
                              gamma=0,\
                              subsample=.8,\
                              colsample_bytree=.8,\
                              booster='gbtree',\
                              objective='binary:logistic',\
                              n_jobs=-1)
    clf.fit(x_train,y_train)
    train_prob=clf.predict_proba(x_train)[:,1]
    test_prob=clf.predict_proba(x_test)[:,1]
    fpr_test, tpr_test, threshold_test = roc_curve(y_test, test_prob,pos_label=1)
    fpr_train, tpr_train, threshold_train = roc_curve(y_train, train_prob,pos_label=1)
    auc_score_test=auc(fpr_test, tpr_test)
    auc_score_train=auc(fpr_train, tpr_train)
    y_pred=clf.predict(x_test)
    f1=f1_score(y_test,y_pred,average='weighted')

    #ADD RESULTS TO PRETTY TABLE
    summarize.add_row([vect, optimal['max_depth'],optimal['n_estimators'], '%.3f' %auc_score_test,'%.3f' %f1])

    plt.figure(1,figsize=(14,5))
    plt.subplot(121)
    plt.title('ROC Curve (%s)' %vect)
    #IDEAL ROC CURVE
    plt.plot([0,1],[0,1],'k--')
    #ROC CURVE OF TEST DATA
```

```
    plt.plot(fpr_test, tpr_test , 'b', label='Test_AUC= %.2f' %auc_score_test)
    #ROC CURVE OF TRAIN DATA
    plt.plot(fpr_train, tpr_train , 'g', label='Train_AUC= %.2f' %auc_score_train)
    plt.xlim([-0.1,1.1])
    plt.ylim([-0.1,1.1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.grid(True)
    plt.legend(loc='lower right')

    #PLOT CONFUSION MATRIX USING HEATMAP
    plt.subplot(122)
    plt.title('Confusion-Matrix(Test Data)')
    df_cm = pd.DataFrame(confusion_matrix(y_test, y_pred), ['Negative','Positive'],['Negative','Positive'])
    sns.set(font_scale=1.4)#for label size
    sns.heatmap(df_cm,cmap='gist_earth', annot=True,annot_kws={"size": 16}, fmt='g')
    plt.show()
    return clf
```

Function which print top important features (feature importance):

In [9]:

```
def feature_importance(vectorizer,clf,n):
    '''FUNCTION FOR FEATURE IMPORTANCE AND PLOT CORRESPONDING IMPORTANT FEATURES IN WORDCLOUD AND BARCHART'''
    #CALCULATE FEATURE IMPORTANCES FROM ENSEMBLE MODEL
    importances = clf.feature_importances_

    # SORT FEATURE IMPORTANCES IN DECENDING ORDER
    indices = np.argsort(importances)[::-1][:n]

    # Rearrange feature names so they match the sorted feature importances
    names = vectorizer.get_feature_names()
    names=np.array(names)

    #wordcloud plot
    wordcloud = WordCloud(max_font_size=50, max_words=100,collocations=False).\
    generate(str(names[indices]))
    plt.figure(1,figsize=(14,13))
    plt.title("WordCloud(Important Feature)")
    plt.imshow(wordcloud, interpolation="bilinear")
    plt.axis("off")

    #bar chart
    plt.figure(2,figsize=(13,8))
    sns.set(rc={'figure.figsize':(11.7,8.27)})
    # Create plot title
    plt.title("Feature Importance")
    # Add bars
    plt.bar(range(n), importances[indices])
    # Add feature names as x-axis labels
    plt.xticks(range(n), names[indices], rotation=70)
    # Show plot
    plt.show()
```

Initialization of common objects required for all vectorization:

In [10]:
```
#ENSEMBLE MODEL TO USE
classifier=['random_forest','gbdt']
#VECTORIZER
vect=['BoW','TF-IDF','AVG-W2V','TFIDF-W2V']
#OBJECT FOR TIMESERIES CROSS VALIDATION
TBS=TimeSeriesSplit(n_splits=10)
#METHOD USE FOR HYPER PARAMETER TUNNING
searchMethod='grid'
#RANGE OF VALUES FOR HYPERPARAM
estimators=[16,32,64,128,256,512]
depth=[9,16,18,20]
params={'max_depth':depth,'n_estimators':estimators}
#INITIALIZE PRETTY TABLE OBJECT
summarize = PrettyTable()
summarize.field_names = ['Vectorizer', 'Optimal-Depth', 'Optimal #Estimators', 'Test(AUC)','Test(f1-score)']
```

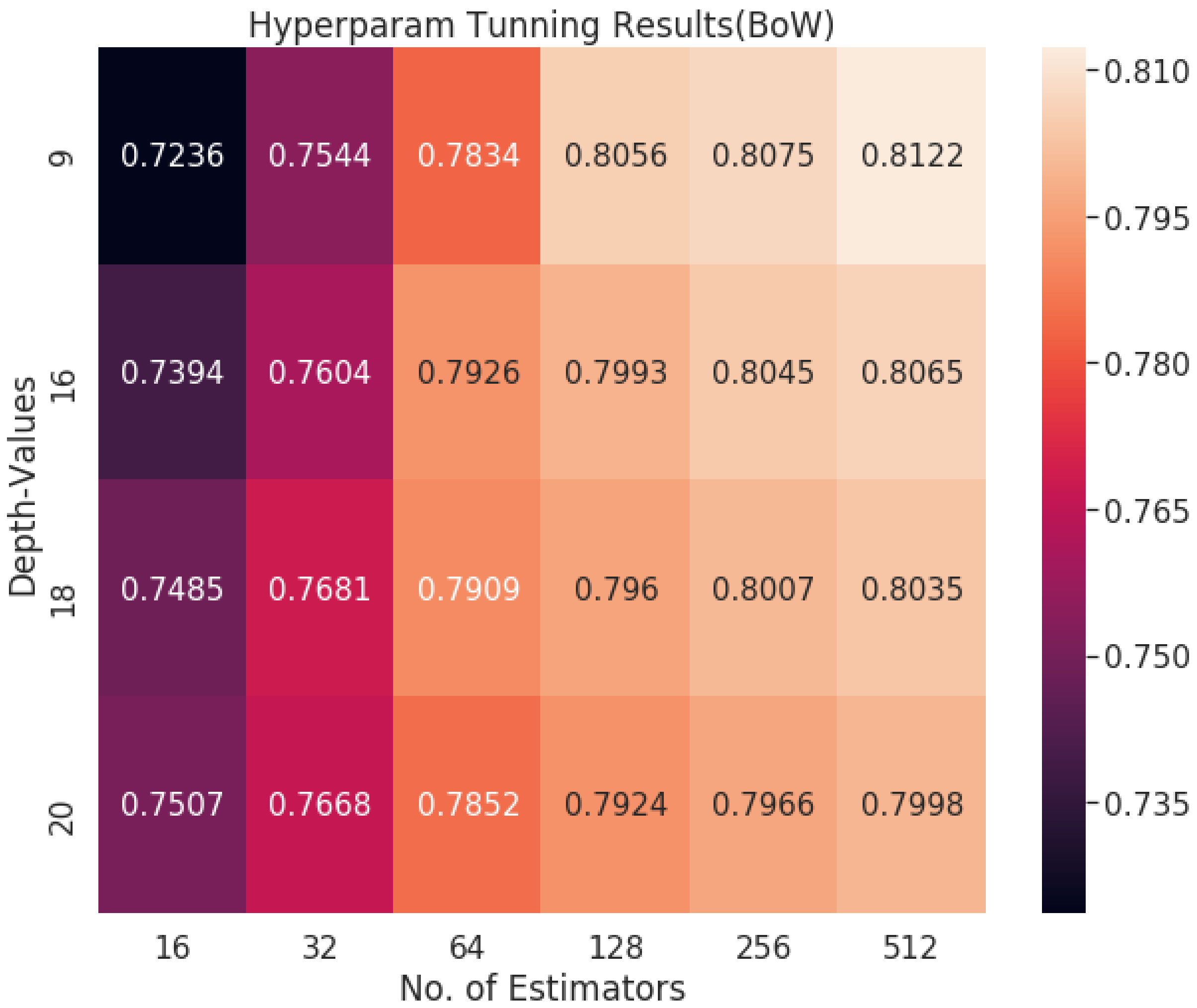
Observation:

1. In case of Random Forest, base learners are high variance(low train error) models.
2. Here the base learners are Decision Tree and Decision Tree is having high variance when the tree is of significant depth(high depth).
3. So here we are taking high depth values in our hyperparam as well.

[1.1] Applying Random Forests on BOW, SET 1

[1.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

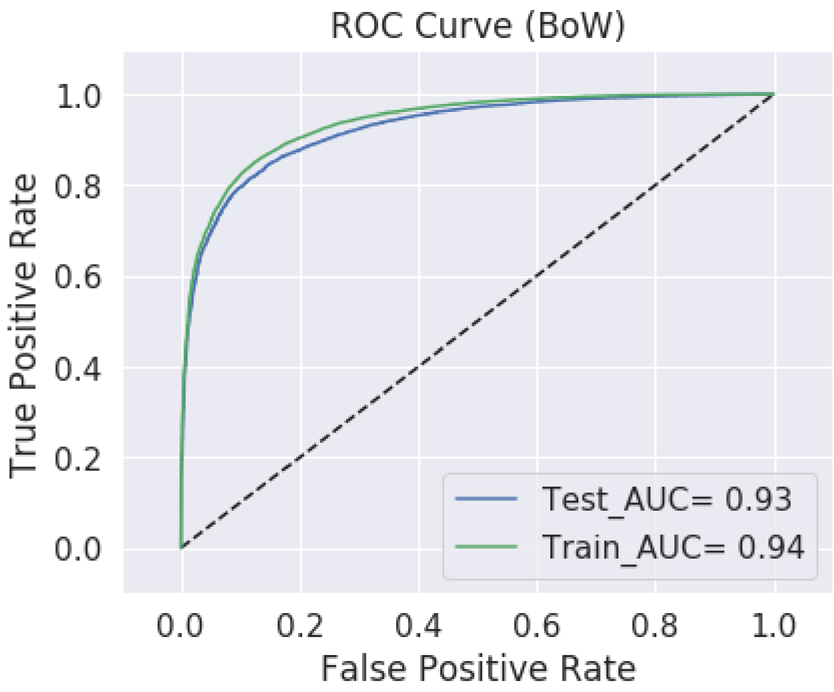
In [82]:
```
#TRAIN AND TEST DATA
train=X_train_bigram;test=X_test_bigram;
#defaulat search tech='GridSearch'
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[0],classifier[0])
print(model.best_params_)
```

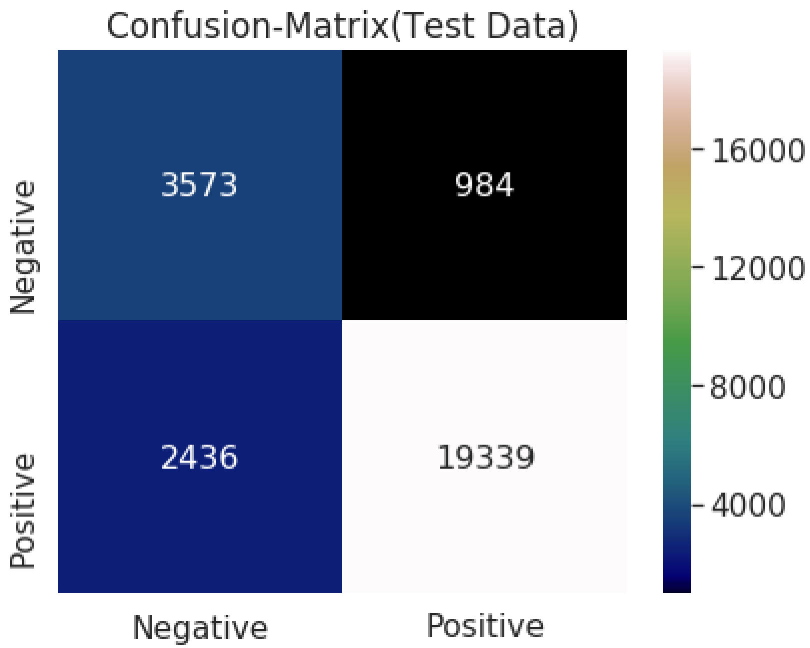



```
CPU times: user 49.3 s, sys: 788 ms, total: 50 s
Wall time: 5min 47s
{'n_estimators': 512, 'max_depth': 9}
```

[1.1.2] Performance on test data with optimal value of hyperparam:

In [83]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[0],summarize,classifier[0])
```

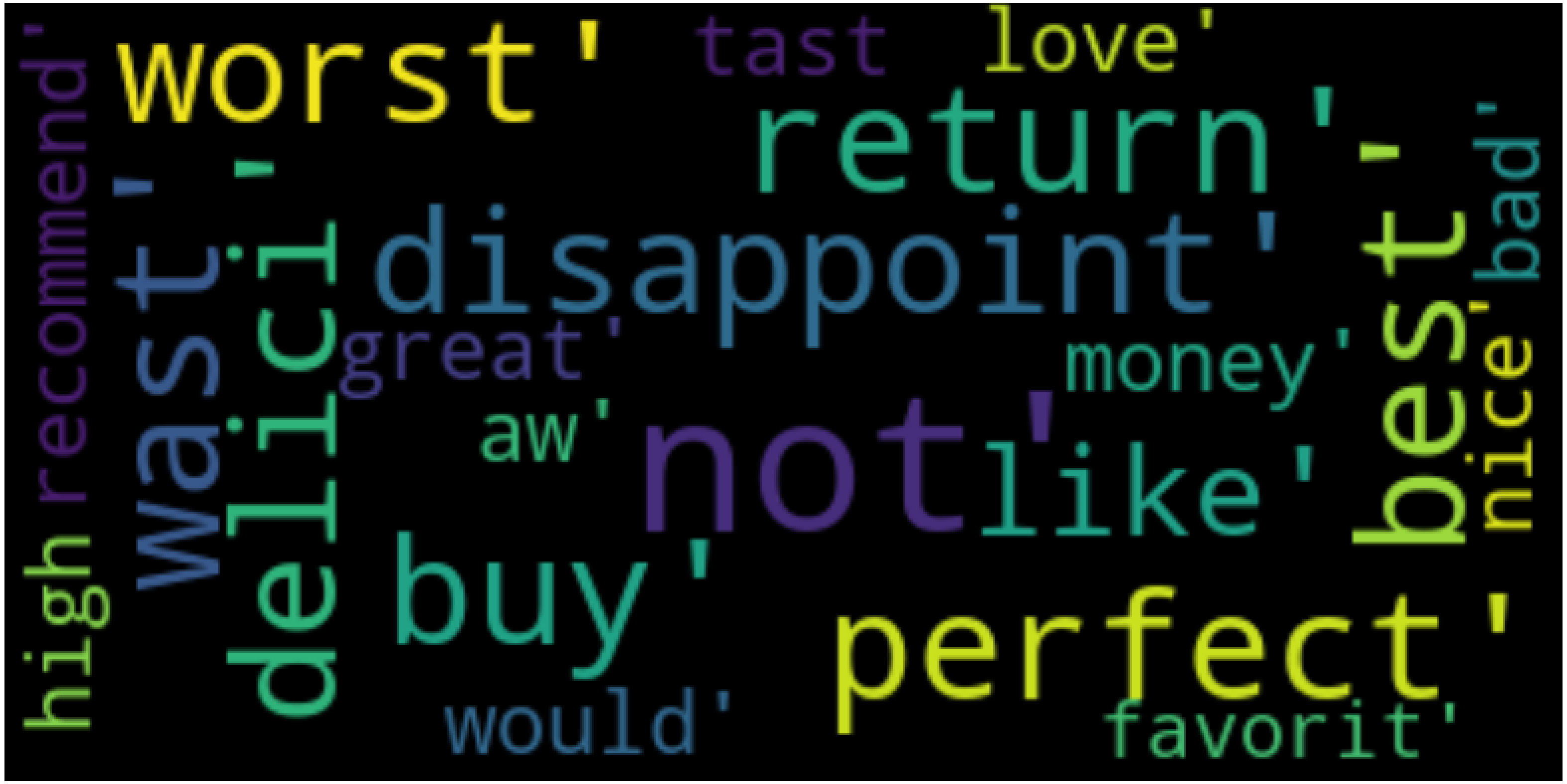


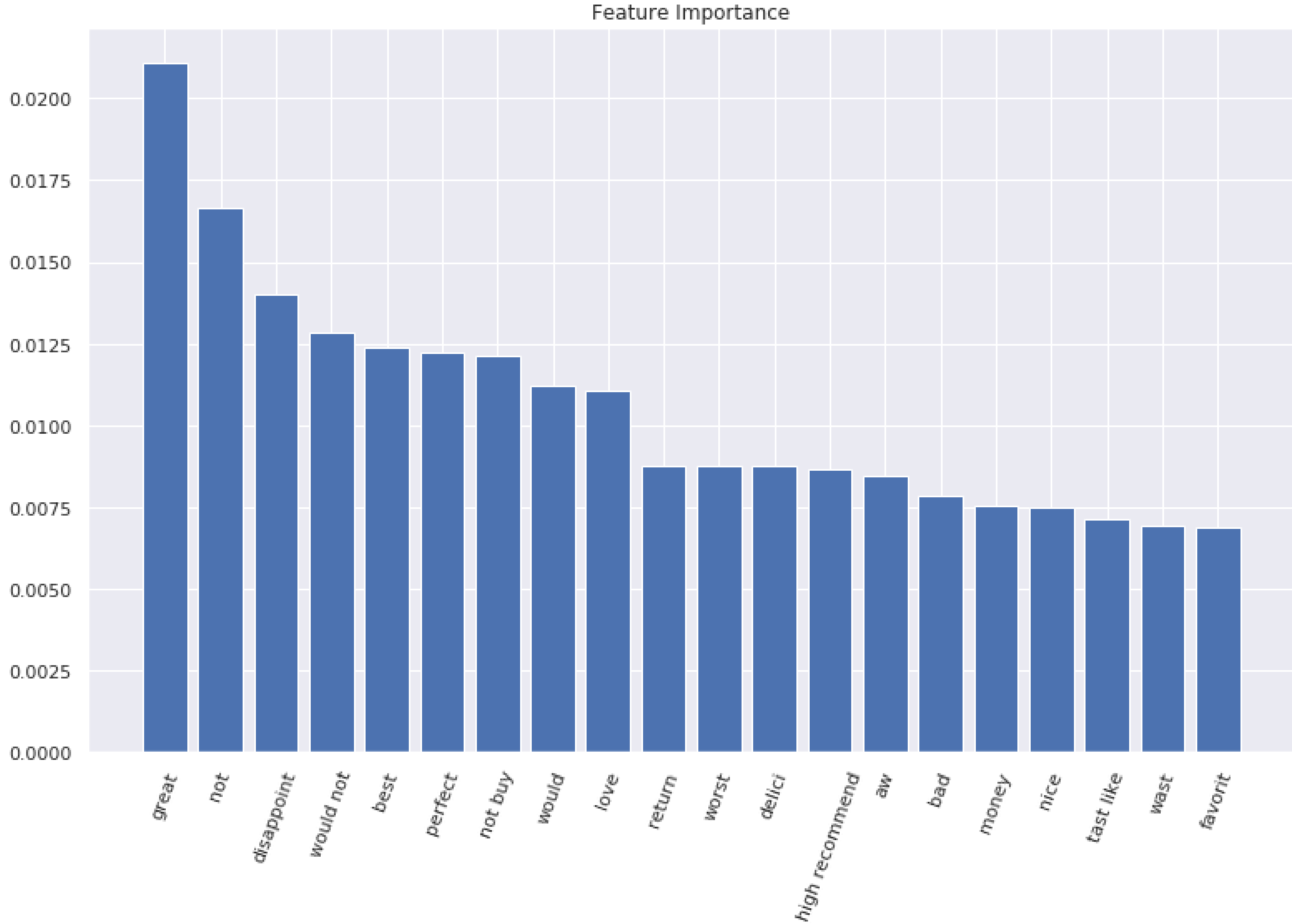


[1.1.3] Top 20 important features:

In [84]:
```
no_of_imp_features=20
feature_importance(count_vect,clf,no_of_imp_features)
```

WordCloud(Important Feature)





Observation:

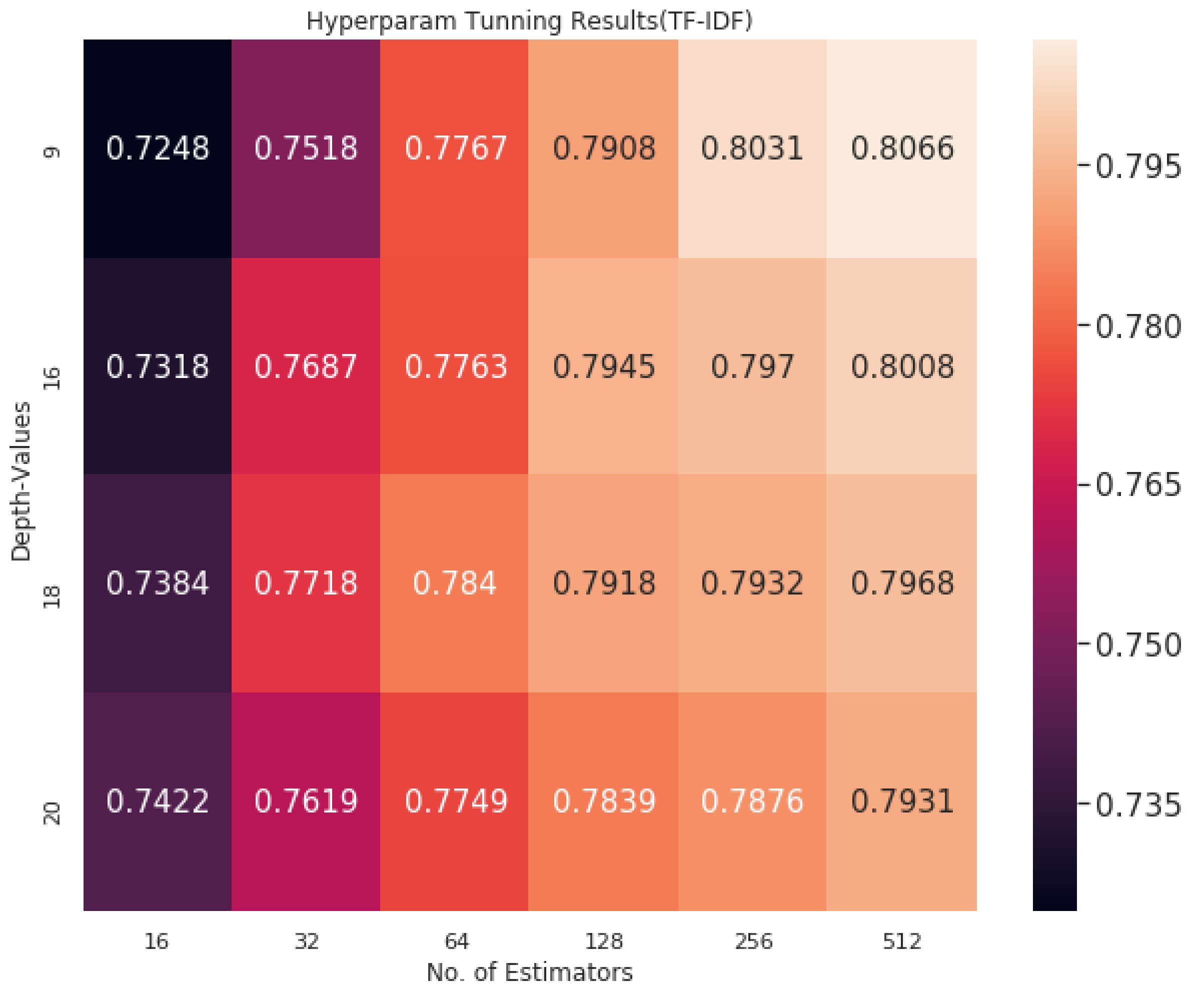
1. Random Forest is good at overall interpretation.
2. feature_importances_ provides the overall important feature.
3. We can't get class based feature importance in Random Forest.

[2.1] Applying Random Forests on TFIDF, SET 2

[2.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

In [86]:

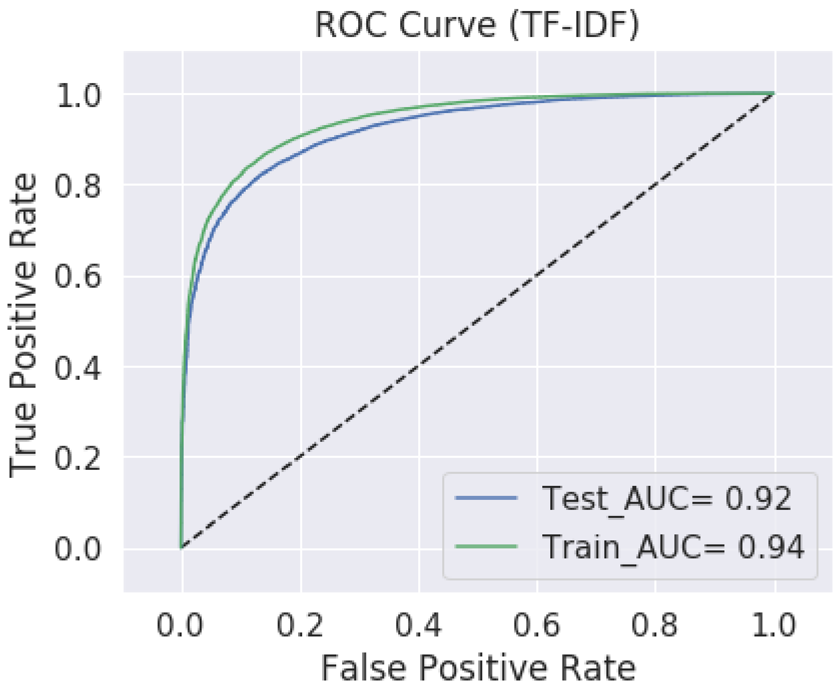
```
#TRAIN AND TEST DATA
#train, test=std_data(train=X_train_tfidf,test=X_test_tfidf,mean=False)
train=X_train_tfidf;test=X_test_tfidf;
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[1],classifier[0])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```

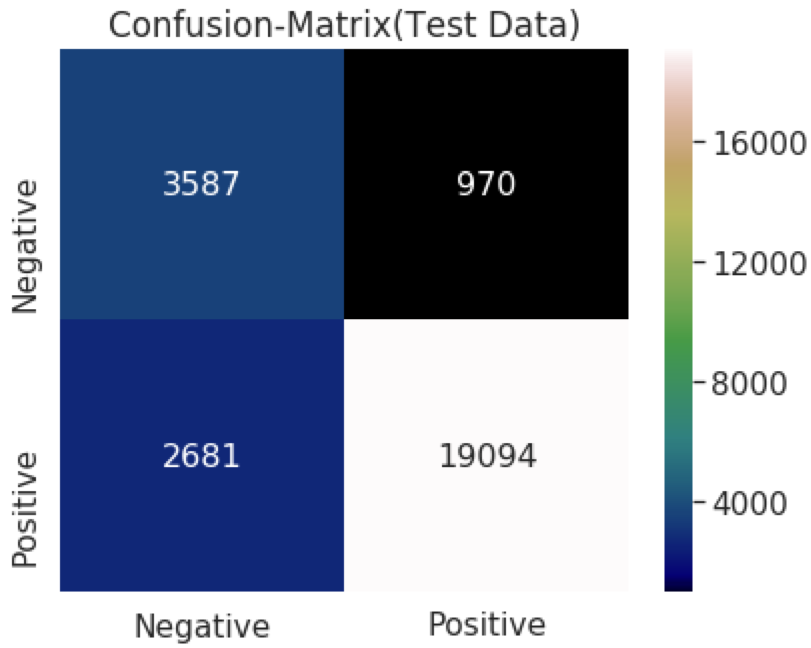



```
CPU times: user 50.7 s, sys: 884 ms, total: 51.6 s
Wall time: 5min 59s
Optimal value of hyperparam:  {'n_estimators': 512, 'max_depth': 9}
```

[2.1.2] Performance on test data with optimal value of hyperparam:

In [87]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[1],summarize,classifier[0])
```

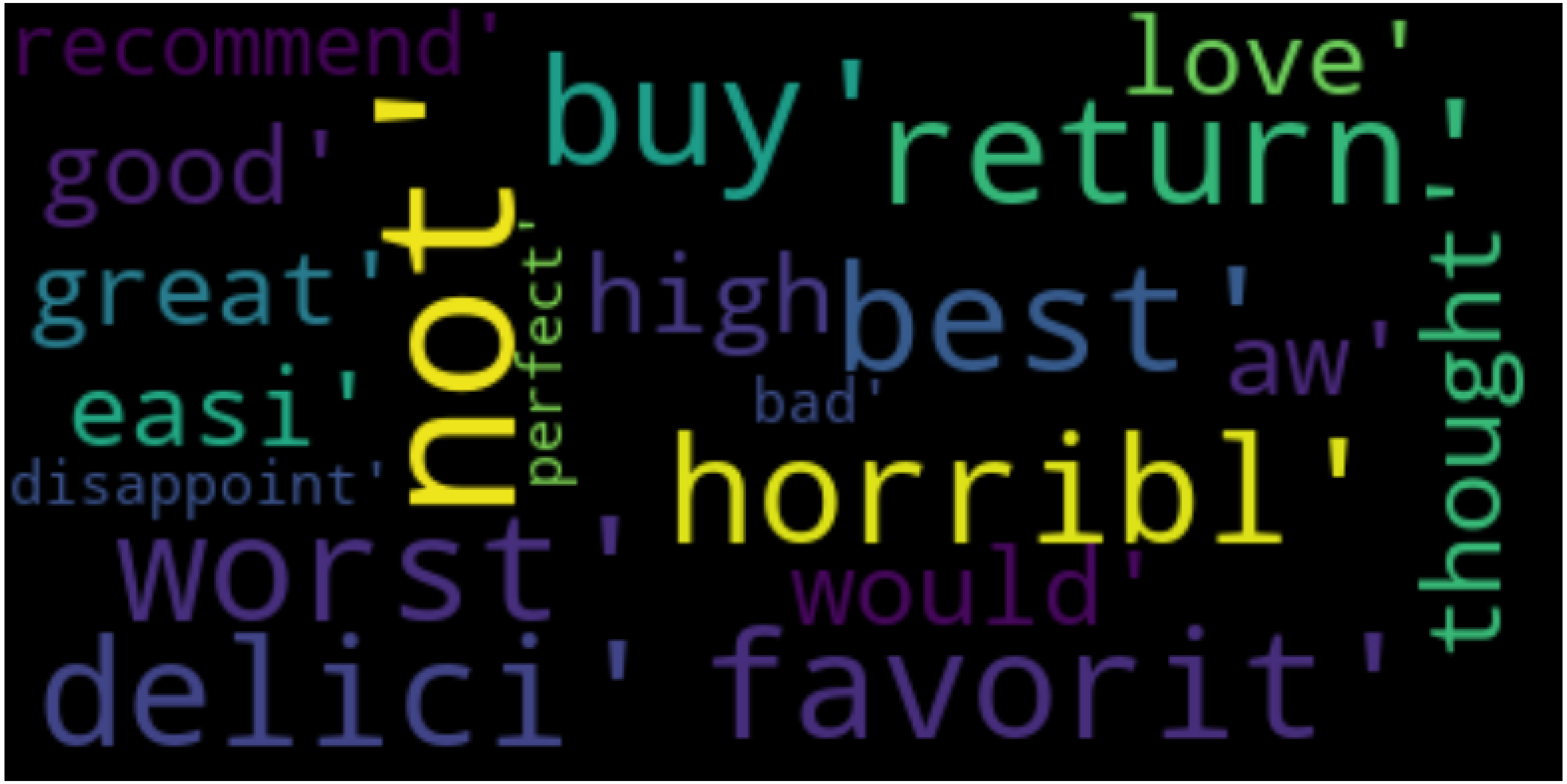


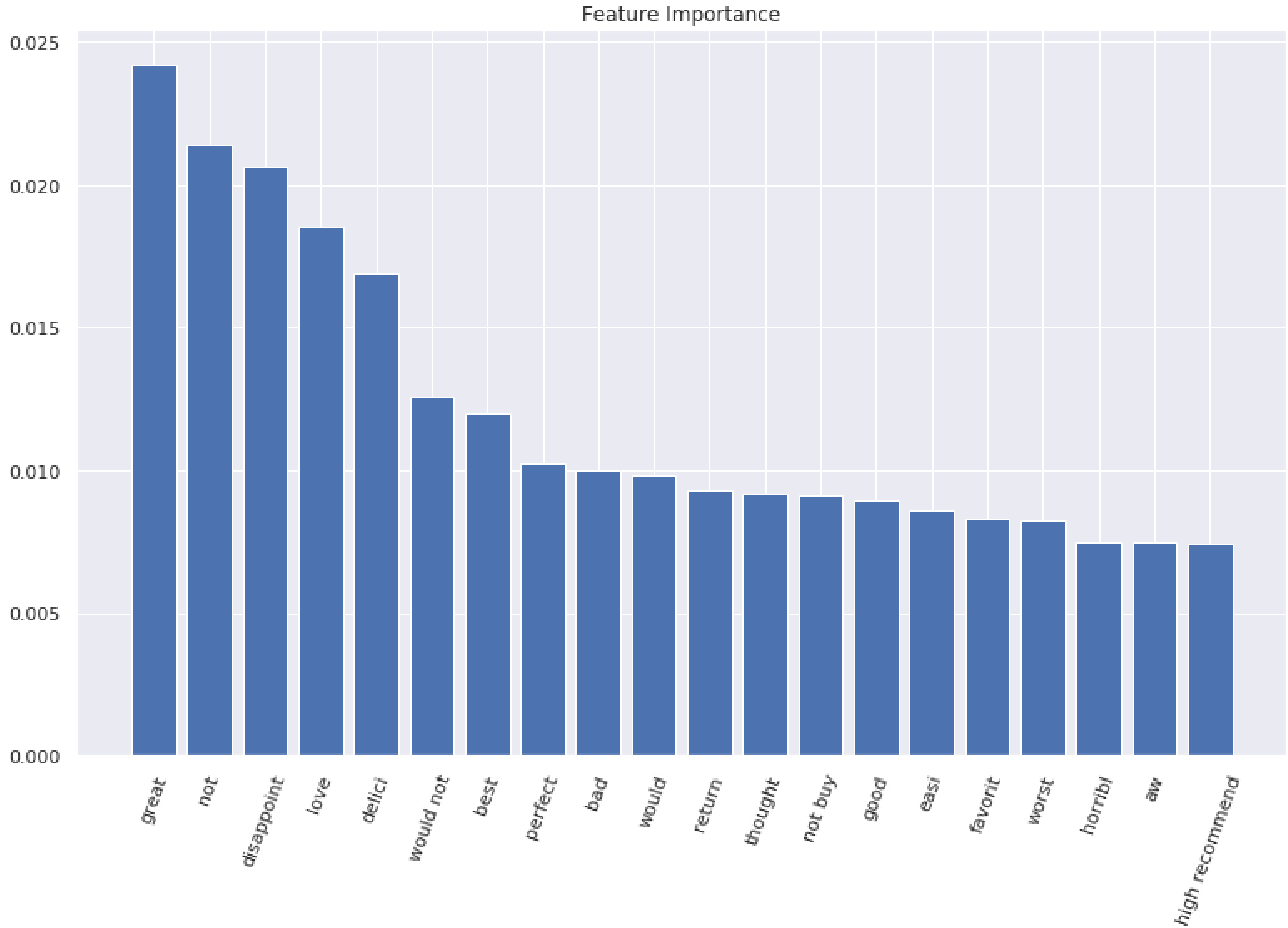


[2.1.3] Top 20 important features:

In [88]:
```
no_of_imp_features=20
feature_importance(tf_idf_vect,clf,no_of_imp_features)
```

WordCloud(Important Feature)





Observation:

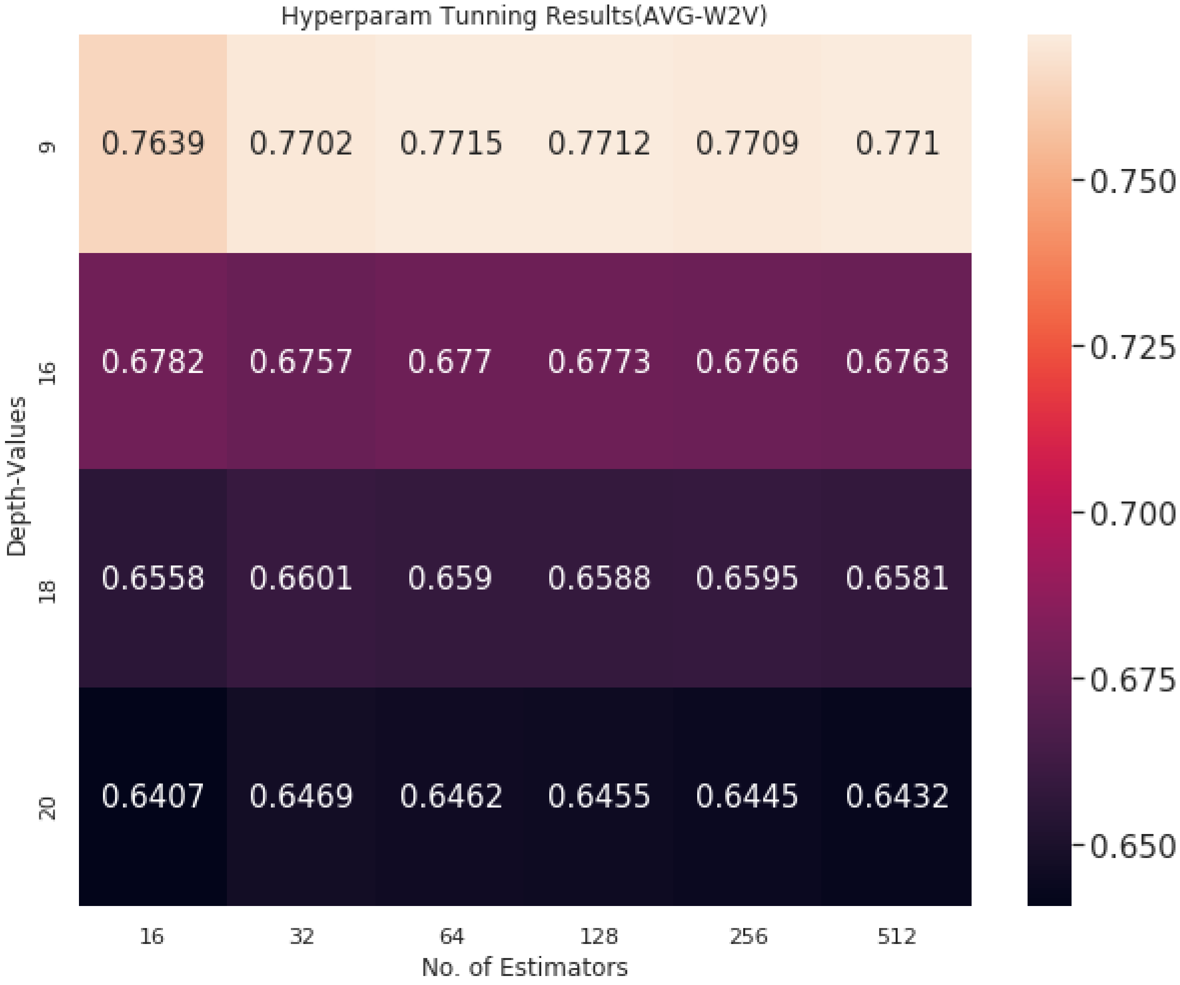
1. Random Forest is good at overall interpretation.
2. feature_importances_ provides the overall important feature.
3. We can't get class based feature importance in Random Forest.

[3.1] Applying Random Forests on AVG-W2V, SET 3

[3.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

In [89]:

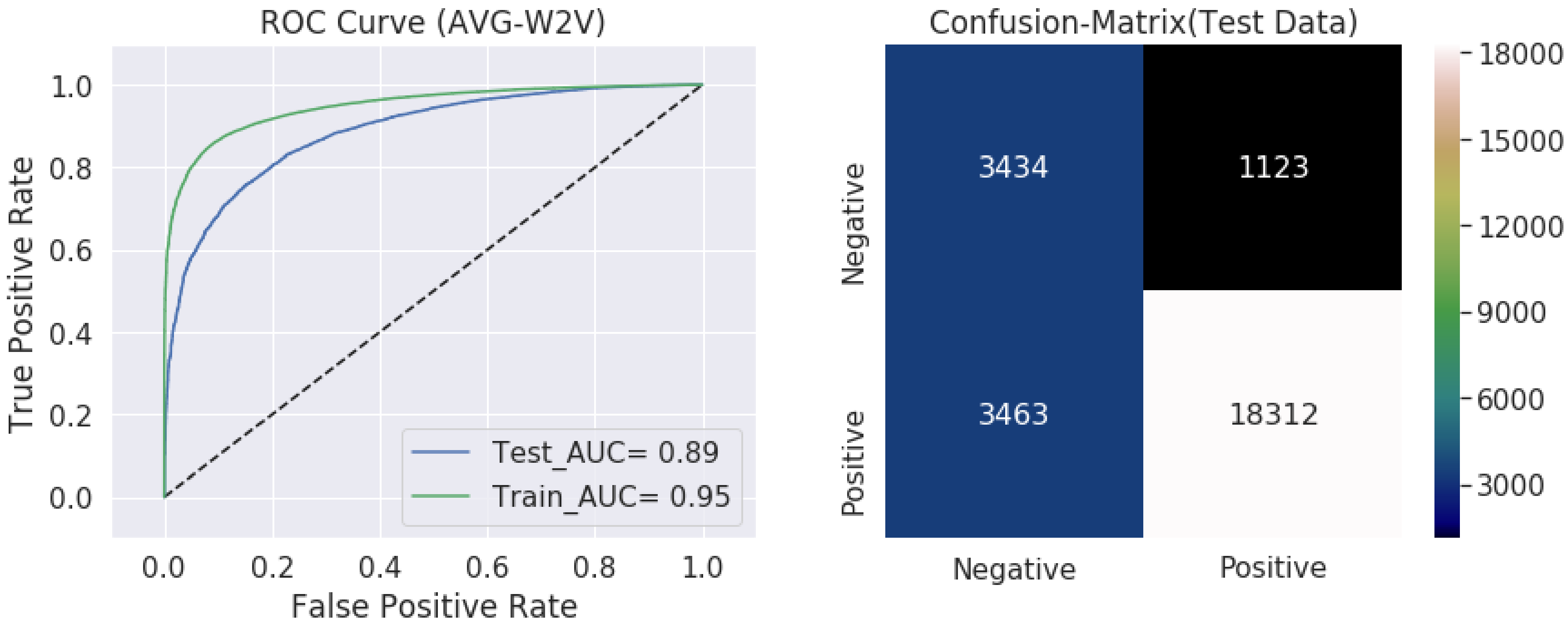
```
#TRAIN AND TEST DATA
#train, test=std_data(train=avg_sent_vectors,test=avg_sent_vectors_test,mean=True)
train=avg_sent_vectors;test=avg_sent_vectors_test;
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[2],classifier[0])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```




```
CPU times: user 6min 7s, sys: 29.9 s, total: 6min 37s
Wall time: 22min 2s
Optimal value of hyperparam:  {'n_estimators': 64, 'max_depth': 9}
```

[3.1.2] Performance on test data with optimal value of hyperparam:

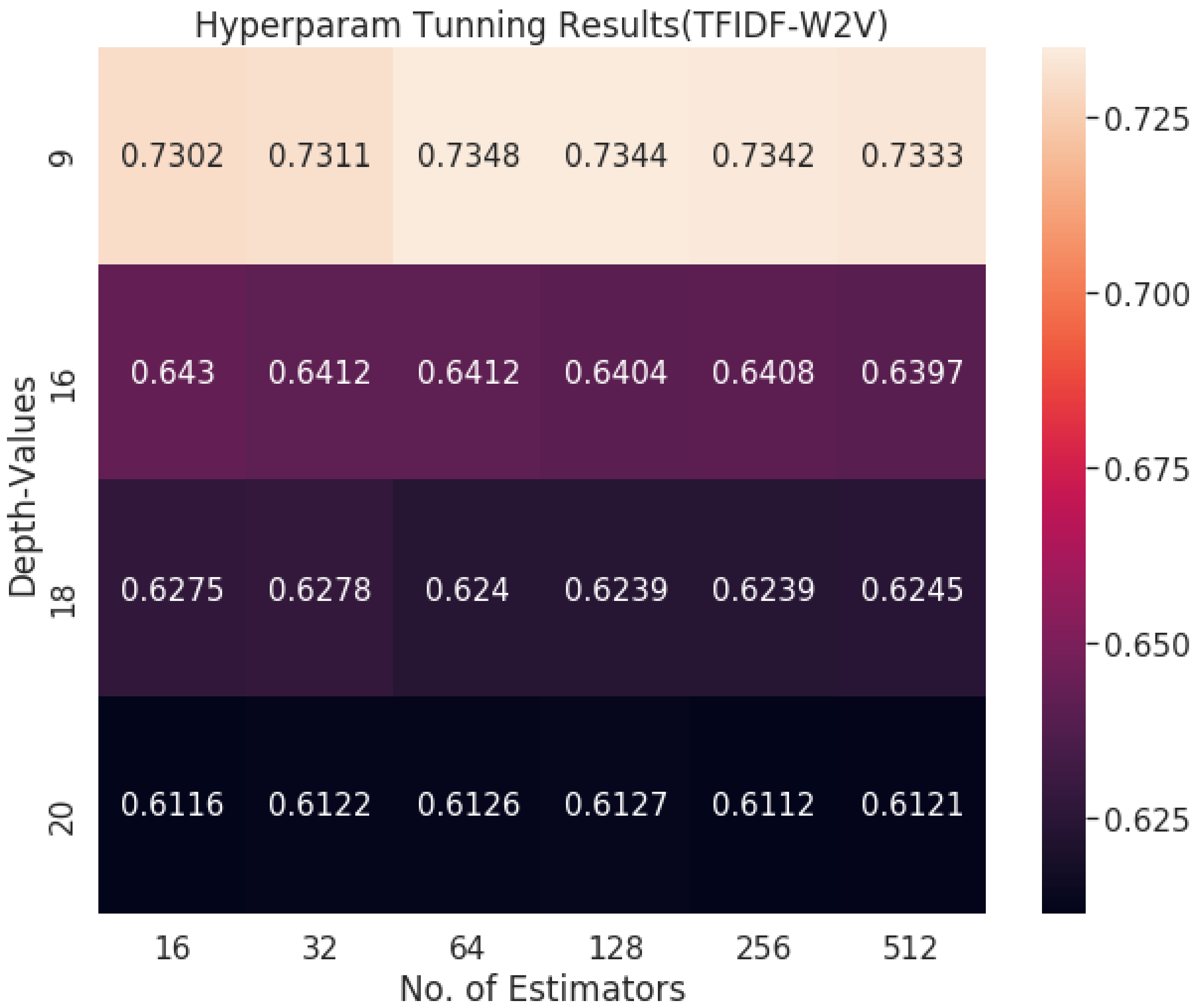
In [90]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[2],summarize,classifier[0])
```



[4.1] Applying Random Forests on TFIDF-W2V, SET 4

[4.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

In [95]:
```
#TRAIN AND TEST DATA
#train, test=std_data(train=avg_sent_vectors,test=avg_sent_vectors_test,mean=True)
train=tfidf_sent_vectors;test=tfidf_sent_vectors_test;
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[3],classifier[0])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```

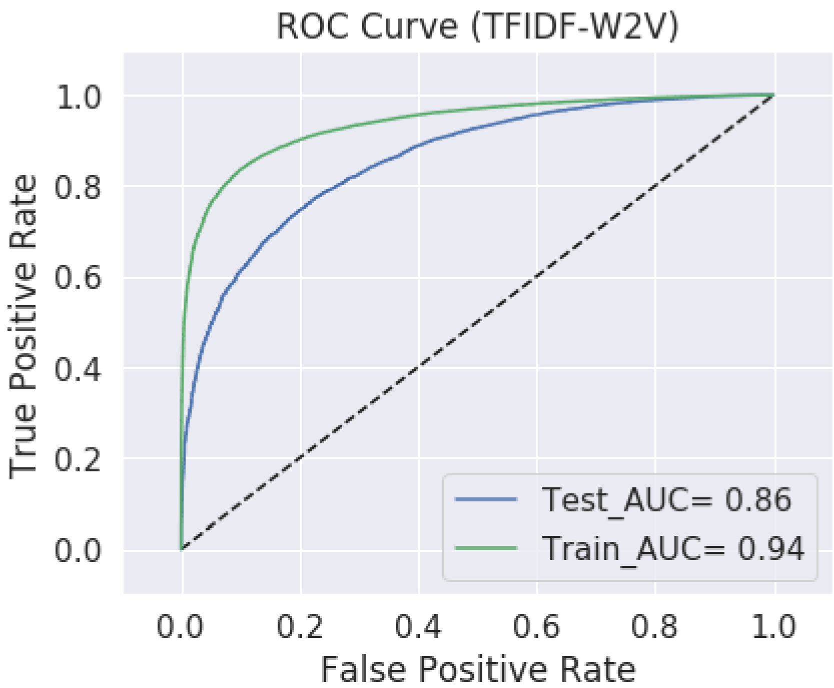


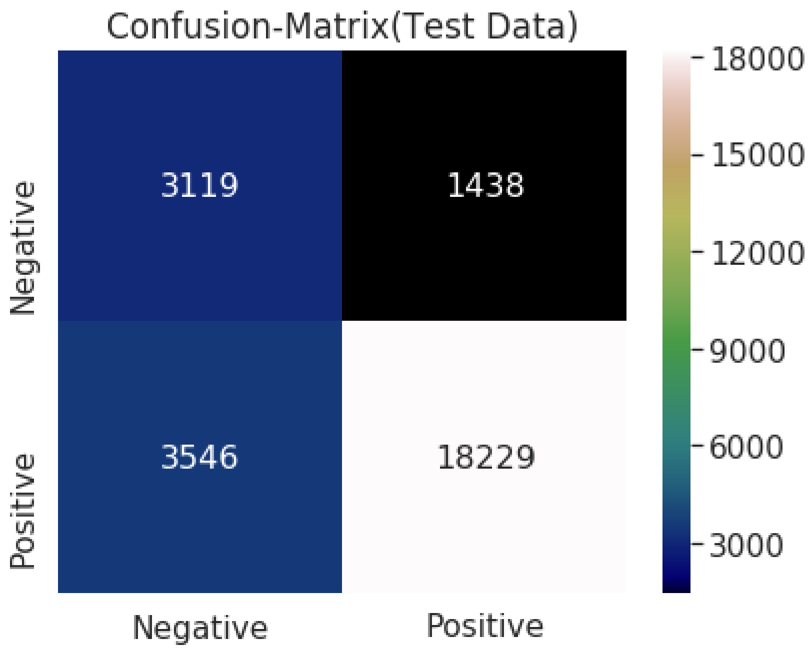

```
CPU times: user 6min 12s, sys: 30.6 s, total: 6min 42s
Wall time: 22min 19s
```

```
Optimal value of hyperparam:  {'n_estimators': 64, 'max_depth': 9}
```

[4.1.2] Performance on test data with optimal value of hyperparam:

In [96]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[3],summarize,classifier[0])
```





Conclusion:

In [97]:
```
#summarize.del_row(2)
print(summarize)
```

```
+------------+---------------+---------------------+-----------+----------------+
| Vectorizer | Optimal-Depth | Optimal #Estimators | Test(AUC) | Test(f1-score) |
+------------+---------------+---------------------+-----------+----------------+
|    BoW     |       9       |         512         |   0.926   |     0.877      |
|   TF-IDF   |       9       |         512         |   0.921   |     0.869      |
|  AVG-W2V   |       9       |          64         |   0.886   |     0.839      |
| TFIDF-W2V  |       9       |          64         |   0.858   |     0.824      |
+------------+---------------+---------------------+-----------+----------------+
```

1. from the above table we can observe that the optimal performance is give by:

```
a. Bag of word vectorizer
b. f1-score=.877 and auc=.926
```

--

[2] GBDT(Gradient Booted Decision Tree)

In [49]:
```
#INITIALIZE PRETTY TABLE OBJECT
summarize_gbdt = PrettyTable()
summarize_gbdt.field_names = ['Vectorizer', 'Optimal-Depth', 'Optimal #Estimators', 'Test(AUC)','Test(f1-score)']
#IN GBDT THE BASE LEARNERS ARE HIGH BIAS MODELS SO WE TAKE LOWER VALUES OF DEPTH IN HYPERPARAM
params['max_depth']=[3,5,7,9,12]
```

Observation:

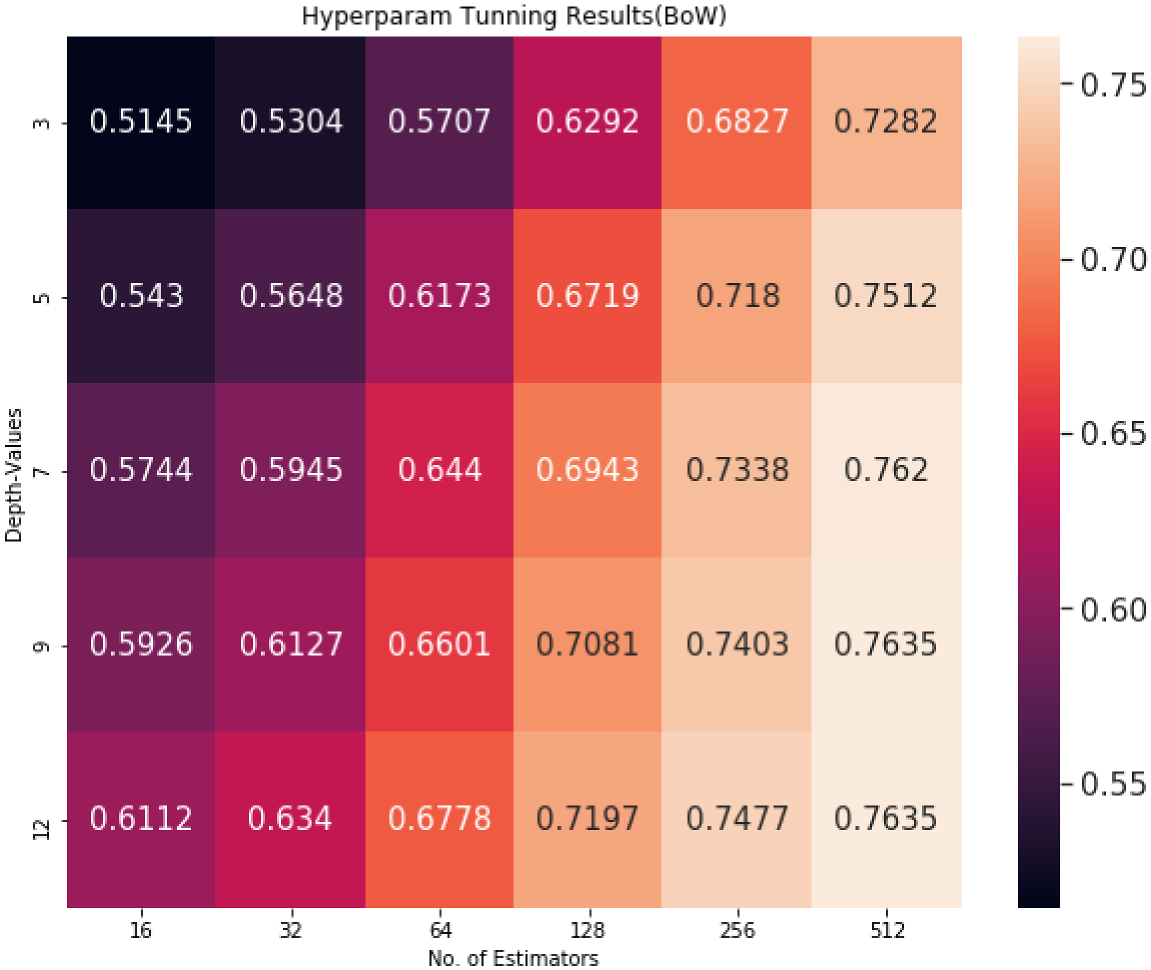
1. In case of GBDT, base learners are high bias(high train error) models.
2. Here the base learners are Decision Tree and Decision Tree is highly biased when the tree is shallow(low depth).

3. So here we are taking low depth values in our hyperparam as well.

[1.1] Applying GBDT on BOW, SET 1

[1.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

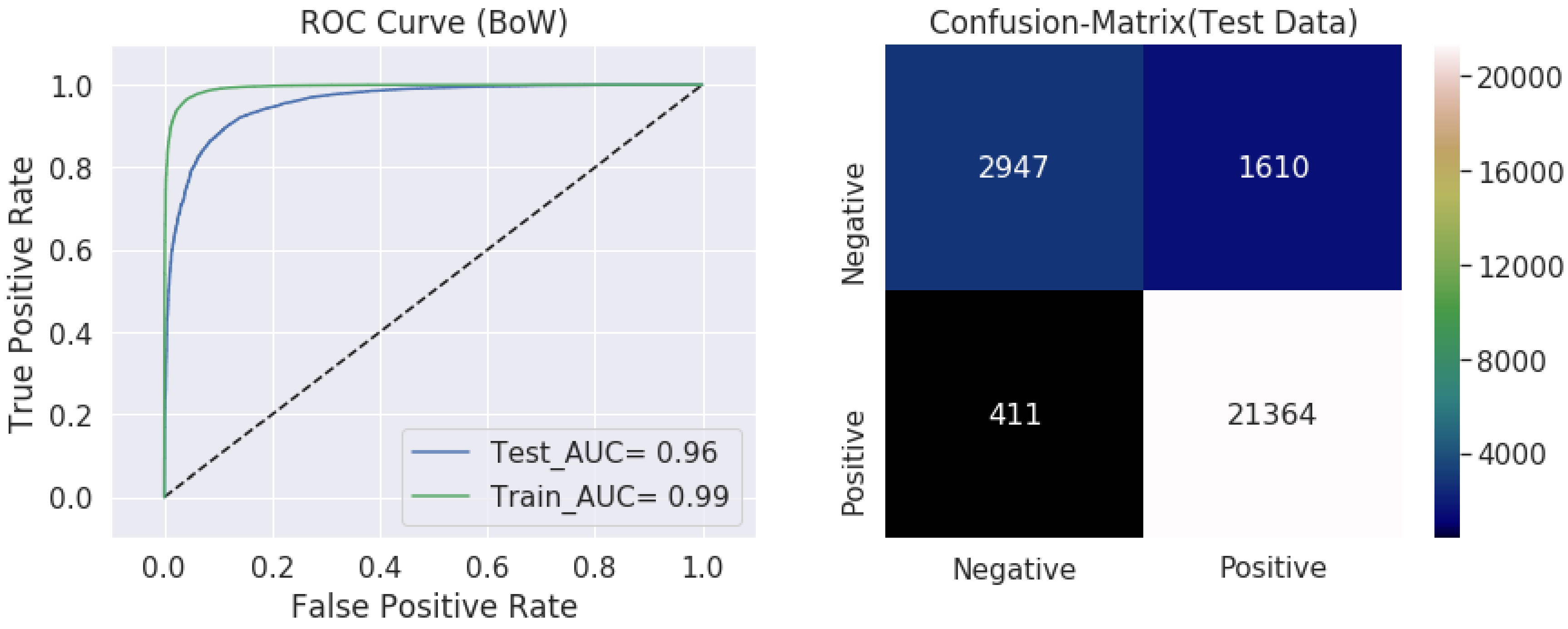
In [20]:
```
#TRAIN AND TEST DATA
#train, test=std_data(train=X_train_bigram,test=X_test_bigram,mean=False)
train=X_train_bigram;test=X_test_bigram;
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[0],classifier[1])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```




```
CPU times: user 7min 43s, sys: 2 s, total: 7min 45s
Wall time: 1h 3min 50s
Optimal value of hyperparam:  {'n_estimators': 512, 'max_depth': 9}
```

[1.1.2] Performance on test data with optimal value of hyperparam:

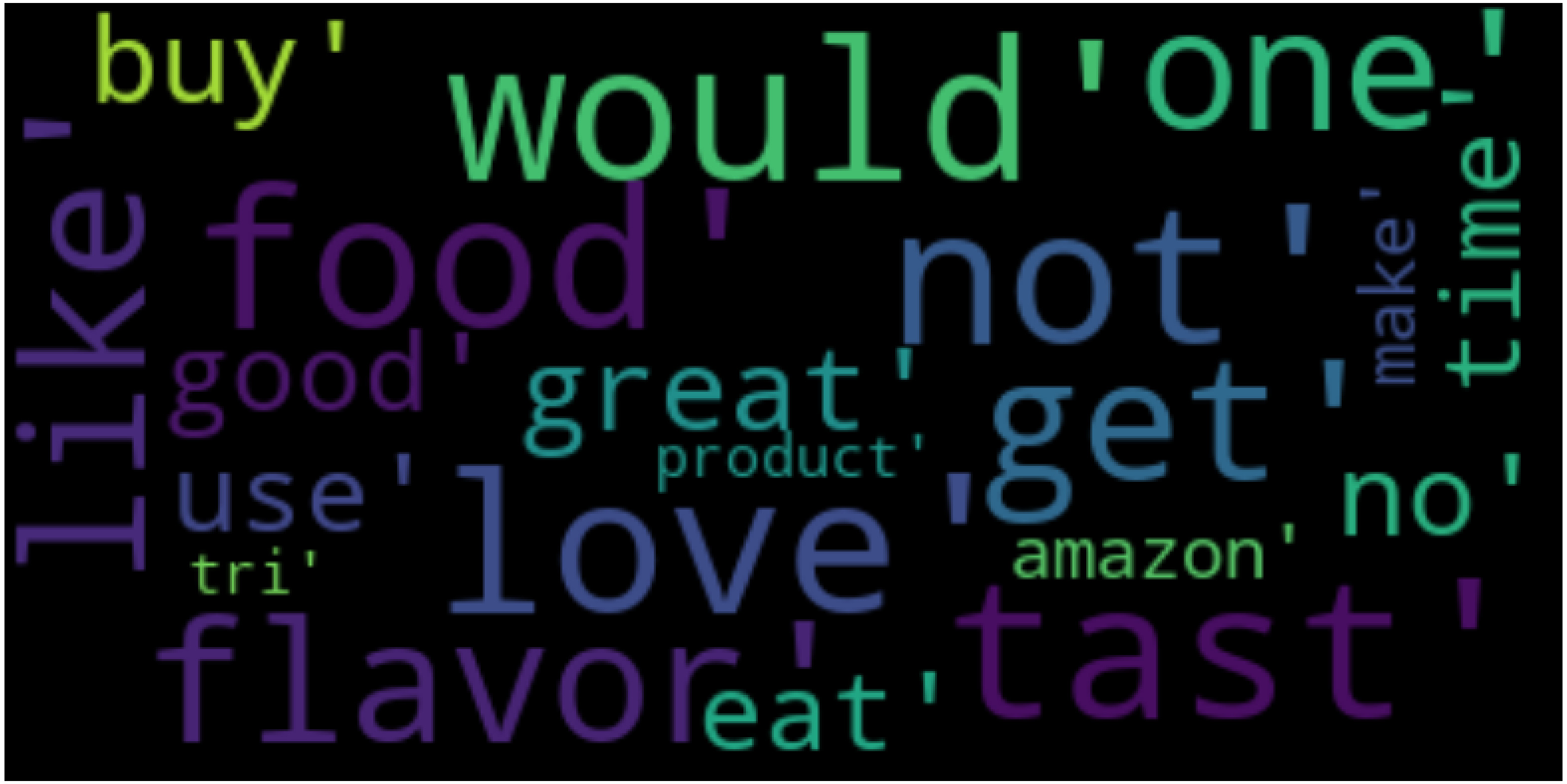
In [21]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[0],summarize_gbdt,classifier[1])
```

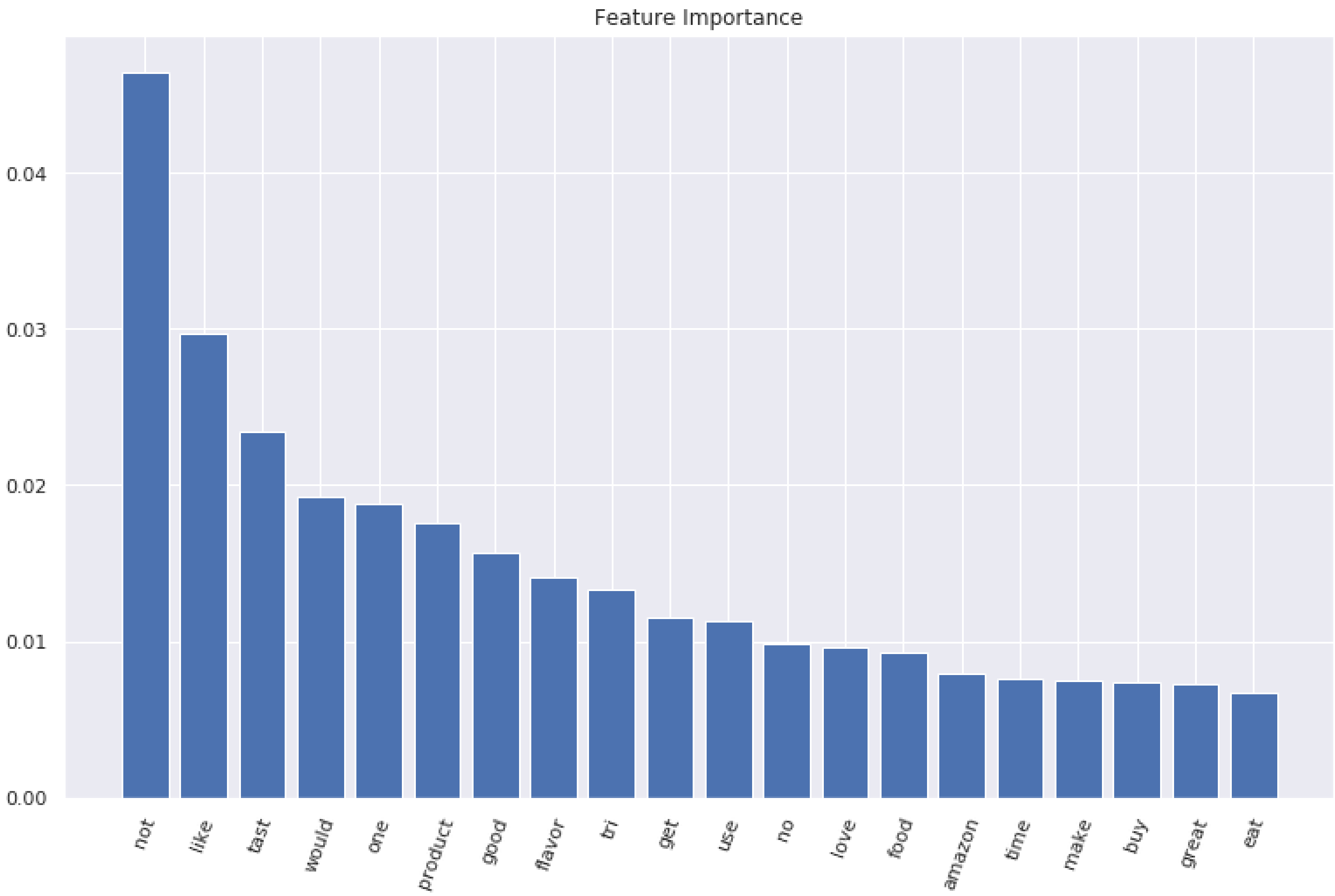


[1.1.3] Top 20 important features:

In [22]:
```
no_of_imp_features=20
feature_importance(count_vect,clf,no_of_imp_features)
```

WordCloud(Important Feature)





Observation:

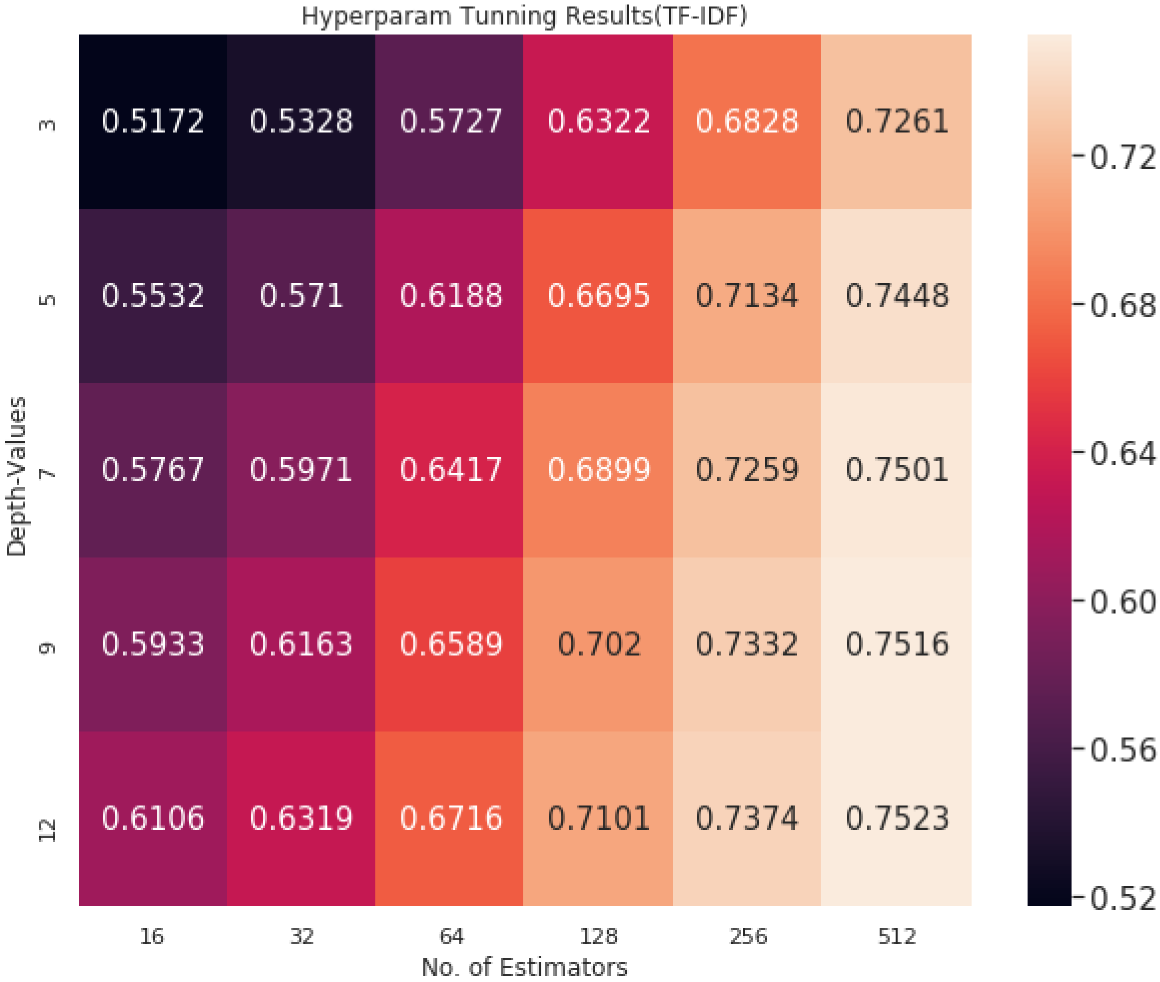
1. GBDT is good at overall interpretation.
2. feature_importances_ provides the overall important feature.
3. We can't get class based feature importance in GBDT.

[2.1] Applying GBDT on TFIDF, SET 2

[2.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

In [23]:

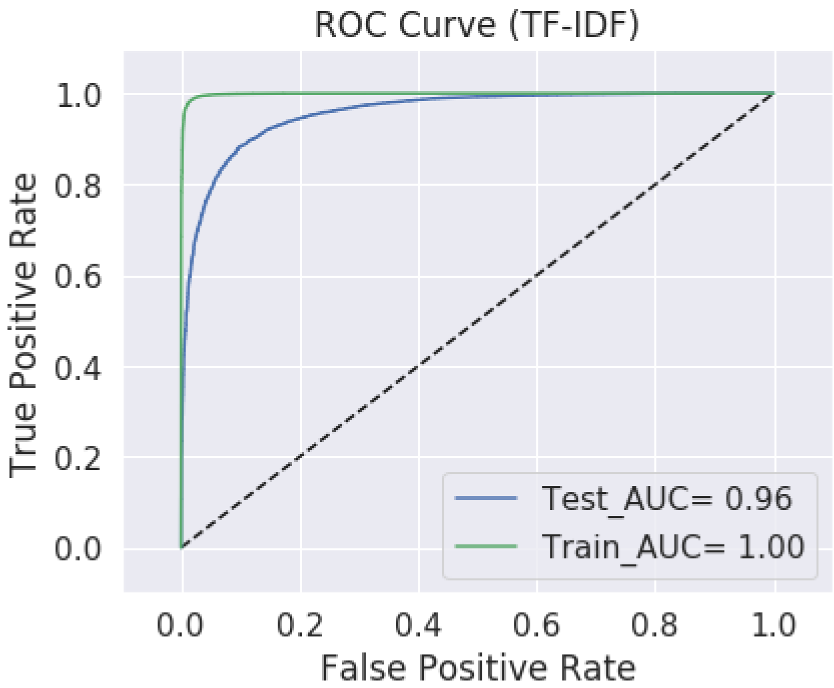
```
# TRAIN AND TEST DATA
#train, test=std_data(train=X_train_tfidf,test=X_test_tfidf,mean=False)
train=X_train_tfidf;test=X_test_tfidf;
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[1],classifier[1])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```

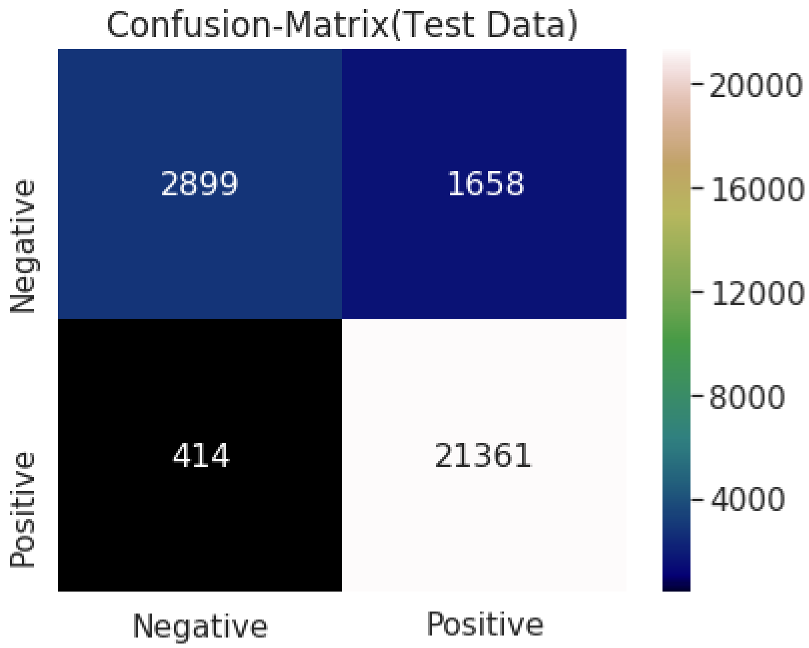



```
CPU times: user 14min 26s, sys: 2.77 s, total: 14min 29s
Wall time: 1h 21min 23s
Optimal value of hyperparam:  {'n_estimators': 512, 'max_depth': 12}
```

[2.1.2] Performance on test data with optimal value of hyperparam:

In [24]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[1],summarize_gbdt,classifier[1])
```

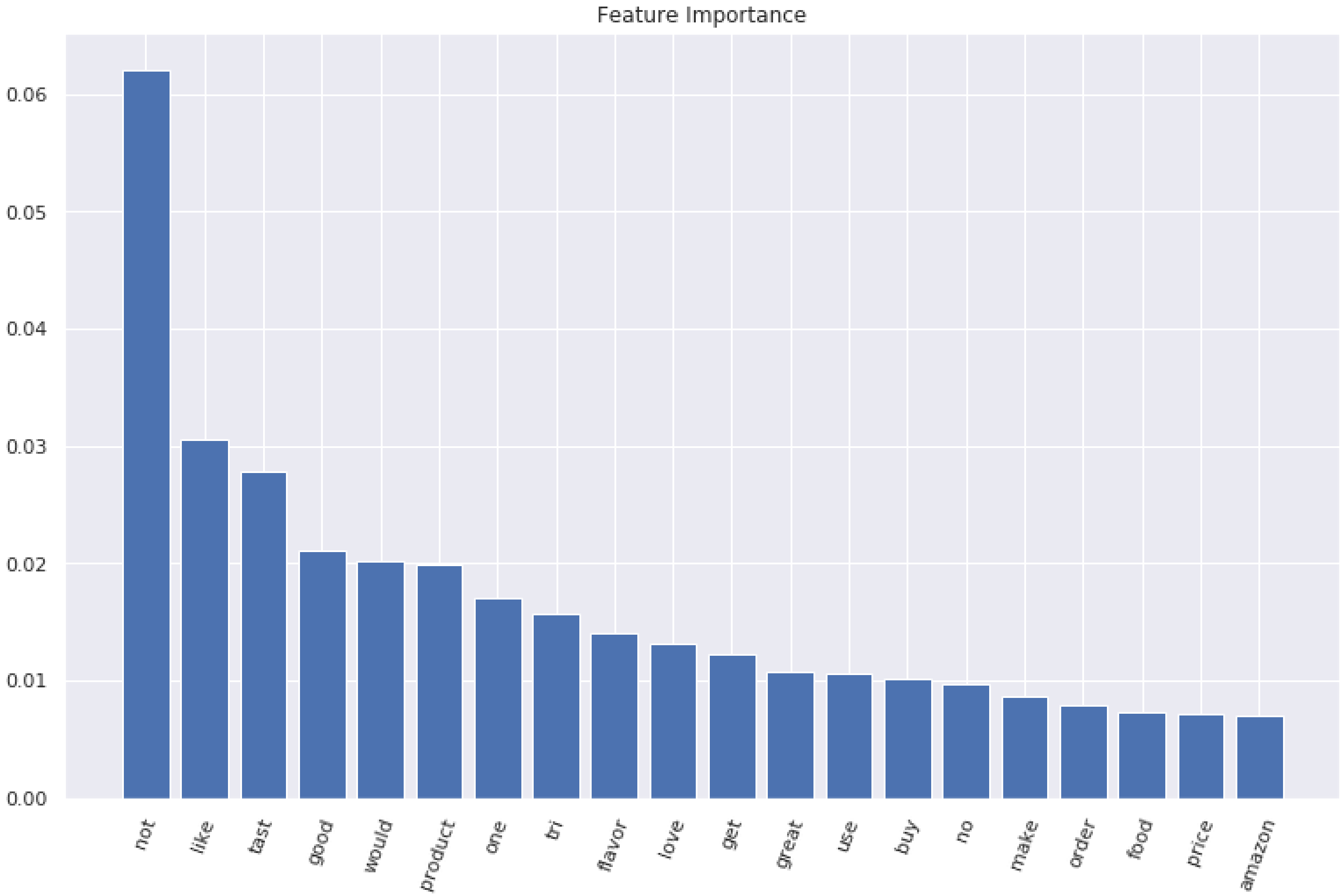




[2.1.3] Top 20 important features:

In [25]:
```
no_of_imp_features=20
feature_importance(tf_idf_vect,clf,no_of_imp_features)
```

WordCloud(Important Feature)



Observation:

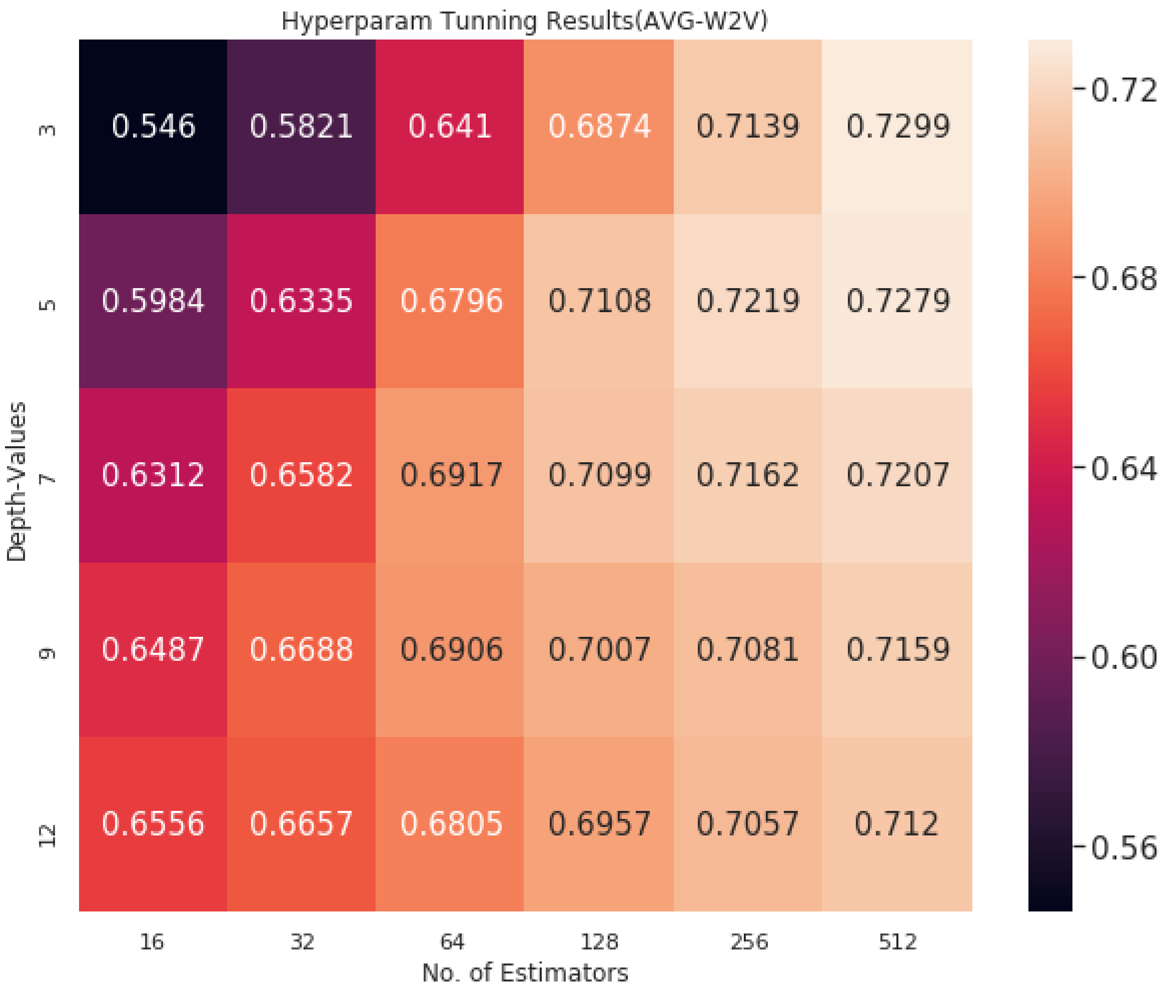
1. GBDT is good at overall interpretation.
2. feature_importances_ provides the overall important feature.
3. We can't get class based feature importance in GBDT.

[3.1] Applying GBDT on AVG W2V, SET 3

[3.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

In [38]:

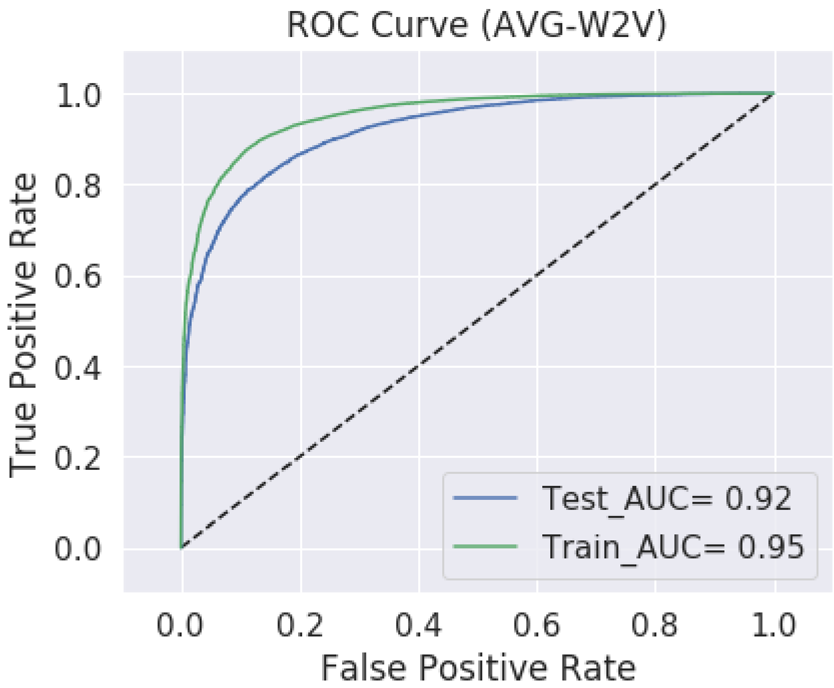
```
#TRAIN AND TEST DATA(convert the avgw2v to array)
train=np.array(avg_sent_vectors);test=np.array(avg_sent_vectors_test);
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[2],classifier[1])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```

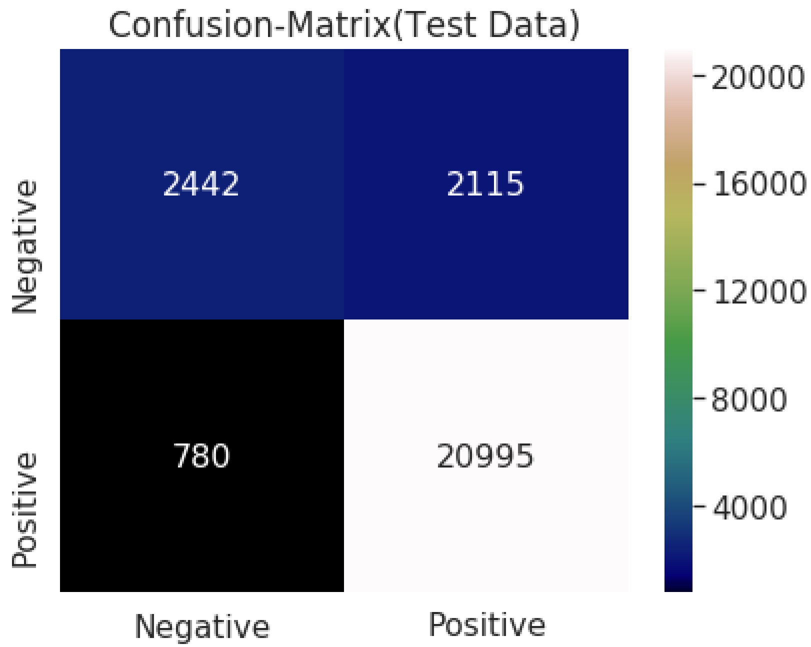



```
CPU times: user 3min 24s, sys: 1.69 s, total: 3min 25s
Wall time: 1h 2min 19s
Optimal value of hyperparam:  {'n_estimators': 512, 'max_depth': 3}
```

[3.1.2] Performance on test data with optimal value of hyperparam:

In [39]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[2],summarize_gbdt,classifier[1])
```



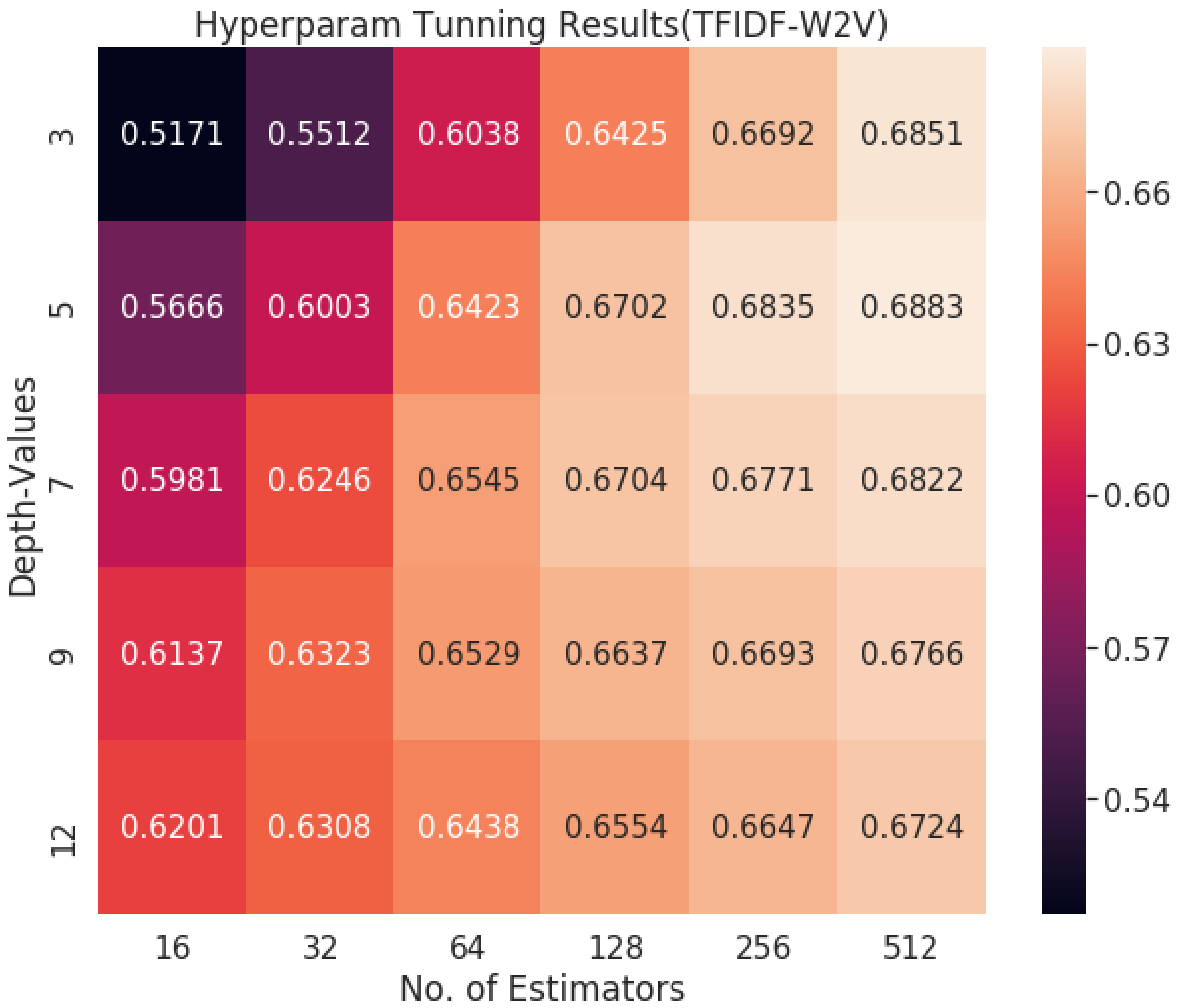


[4.1] Applying GBDT on TFIDF W2V, SET 4

[4.1.1] Hyperparam tunning and plot Heatmap for hyperparam:

In [41]:

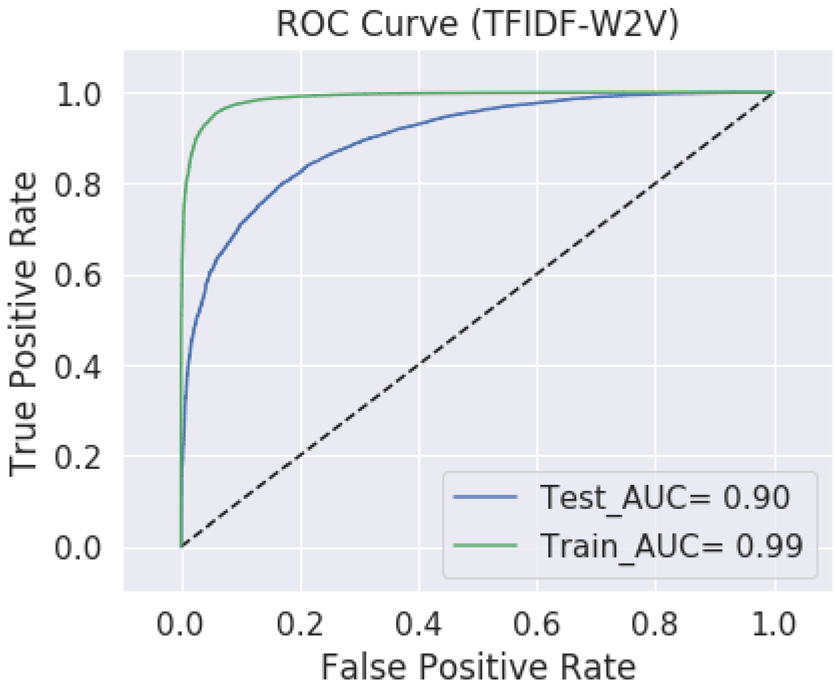
```
#TRAIN AND TEST DATA(convert tfidfw2v to array)
train=np.array(tfidf_sent_vectors);test=np.array(tfidf_sent_vectors_test);
#HYPERPARAM TUNNING
%time model=Ensemble_Classifier(train,y_train,TBS,params,searchMethod,vect[3],classifier[1])
#PRINT OPTIMAL VALUE OF HYPERPARAM
print('Optimal value of hyperparam: ',model.best_params_)
```

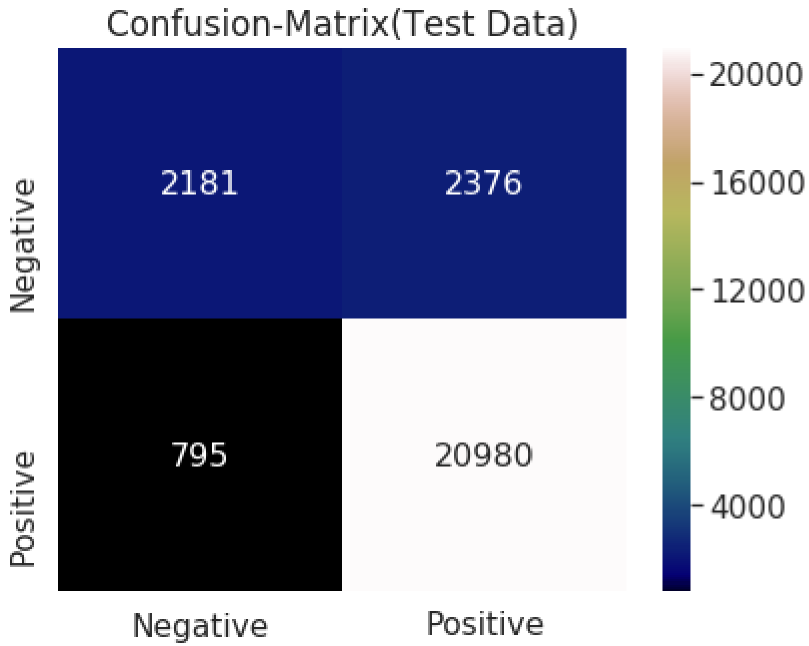



```
CPU times: user 5min 32s, sys: 2.06 s, total: 5min 34s
Wall time: 1h 4min 9s
Optimal value of hyperparam:  {'n_estimators': 512, 'max_depth': 5}
```

[4.1.2] Performance on test data with optimal value of hyperparam:

In [42]:
```
clf=test_performance(train,y_train,test,y_test,model.best_params_,vect[3],summarize_gbdt,classifier[1])
```





Conclusion:

In [47]:
```
print("Summarry of GBDT model with optimal Hyperparam and Scores:" )
print(summarize_gbdt)
```

Summarry of GBDT model with optimal Hyperparam and Scores:

Vectorizer	Optimal-Depth	Optimal #Estimators	Test(AUC)	Test(f1-score)
BoW	9	512	0.958	0.918
TF-IDF	12	512	0.957	0.916
AVG-W2V	3	512	0.919	0.882
TFIDF-W2V	5	512	0.899	0.869

1. from the above table we can observe that the optimal performance is give by:

```
a. Bag of word vectorizer
b. f1-score=.918 and auc=.958
```

In [ ]:
```

```