Amazon Fine Food Reviews Analysis

Data Source: https://www.kaggle.com/snap/amazon-fine-food-reviews

EDA: https://nycdatascience.com/blog/student-works/amazon-fine-foods-visualization/

The Amazon Fine Food Reviews dataset consists of reviews of fine foods from Amazon.

Number of reviews: 568,454
Number of users: 256,059
Number of products: 74,258
Timespan: Oct 1999 - Oct 2012
Number of Attributes/Columns in data: 10

Attribute Information:

1. Id
2. ProductId - unique identifier for the product
3. UserId - unqiue identifier for the user
4. ProfileName
5. HelpfulnessNumerator - number of users who found the review helpful
6. HelpfulnessDenominator - number of users who indicated whether they found the review helpful or not
7. Score - rating between 1 and 5
8. Time - timestamp for the review
9. Summary - brief summary of the review
10. Text - text of the review

Objective:

Given a review, determine whether the review is positive (rating of 4 or 5) or negative (rating of 1 or 2).

[Q] How to determine if a review is positive or negative?

[Ans] We could use Score/Rating. A rating of 4 or 5 can be cosnidered as a positive review. A rating of 1 or 2 can be considered as negative one. A review of rating 3 is considered nuetral and such reviews are ignored from our analysis. This is an approximate and proxy way of determining the polarity (positivity/negativity) of a review.

[1]. Reading Data

[1.1] Loading the data

The dataset is available in two forms

1. .csv file
2. SQLite Database

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

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

In [1]:

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


import sqlite3
import pandas as pd
import numpy as np
import nltk
import string
import matplotlib.pyplot as plt
```

```
import seaborn as sns
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

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

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

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

from tqdm import tqdm
import os
```

In [2]:

```
# using SQLite Table to read data.

db_path = "/home/monodeepdas112/Datasets/amazon-fine-food-reviews/database.sqlite"
con = sqlite3.connect(db_path)

# filtering only positive and negative reviews i.e.
# not taking into consideration those reviews with Score=3
# SELECT * FROM Reviews WHERE Score != 3 LIMIT 500000, will give top 500000 data points
# you can change the number to any other number based on your computing power

# filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 LIMIT 500000""", co
n)
# for tsne assignment you can take 5k data points

filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 LIMIT 100000""", con)

# Give reviews with Score>3 a positive rating(1), and reviews with a score<3 a negative rating(0).
def partition(x):
    if x < 3:
        return 0
    return 1

#changing reviews with score less than 3 to be positive and vice-versa
actualScore = filtered_data['Score']
positiveNegative = actualScore.map(partition)
filtered_data['Score'] = positiveNegative
print("Number of data points in our data", filtered_data.shape)
filtered_data.head(3)
```

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

Out[2]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1	1	1303862400	Good Quality Dog Food
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	0	0	1346976000	Not as Advertised
				Natalia					

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	1	1	1219017600	"Delight" says it all

In [3]:

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

In [4]:

```
print(display.shape)
display.head()
```

```
(80668, 7)
```

Out[4]:

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

In [5]:

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

Out[5]:

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

In [6]:

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

Out[6]:

```
393063
```

[2] Exploratory Data Analysis

[2.1] Data Cleaning: Deduplication

It is observed (as shown in the table below) that the reviews data had many duplicate entries. Hence it was necessary to remove duplicates in order to get unbiased results for the analysis of the data. Following is an example:

In [7]:

```
display= pd.read_sql_query("""
```

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

Out[7]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summ
0	78445	B000HDL1RQ	AR5J8UI46CURR	Geetha Krishnan	2	2	5	1199577600	LOACH QUADRAT VANII WAFE
1	138317	B000HDOPYC	AR5J8UI46CURR	Geetha Krishnan	2	2	5	1199577600	LOACH QUADRAT VANII WAFE
2	138277	B000HDOPYM	AR5J8UI46CURR	Geetha Krishnan	2	2	5	1199577600	LOACH QUADRAT VANII WAFE
3	73791	B000HDOPZG	AR5J8UI46CURR	Geetha Krishnan	2	2	5	1199577600	LOACH QUADRAT VANII WAFE
4	155049	B000PAQ75C	AR5J8UI46CURR	Geetha Krishnan	2	2	5	1199577600	LOACH QUADRAT VANII WAFE

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

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

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

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

The method used for the same was that we first sort the data according to ProductId and then just keep the first similar product review and delelte the others. for eg. in the above just the review for ProductId=B000HDL1RQ remains. This method ensures that there is only one representative for each product and deduplication without sorting would lead to possibility of different representatives still existing for the same product.

In [8]:

```
#Sorting data according to ProductId in ascending order
sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')
```

In [9]:

```
#Deduplication of entries
final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"}, keep='first', inplace=False)
final.shape
```

Out[9]:

```
(87775, 10)
```

In [10]:

```
#Checking to see how much % of data still remains
```

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

Out[10]:

```
87.775
```

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

In [11]:

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

display.head()
```

Out[11]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary
0	64422	B000MIDROQ	A161DK06JJMCYF	J. E. Stephens "Jeanne"	3	1	5	1224892800	Bought This for My Son at College
1	44737	B001EQ55RW	A2V0I904FH7ABY	Ram	3	2	4	1212883200	Pure cocoa taste with crunchy almonds inside

In [12]:

```
final=final[final.HelpfulnessNumerator<=final.HelpfulnessDenominator]
```

In [13]:

```
#Before starting the next phase of preprocessing lets see the number of entries left
print(final.shape)

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

```
(87773, 10)
```

Out[13]:

```
1    73592
0    14181
Name: Score, dtype: int64
```

[3] Preprocessing

[3.1]. Preprocessing Review Text

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

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

1. Begin by removing the html tags

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

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

In [14]:

```
# printing some random reviews
sent_0 = final['Text'].values[0]
print(sent_0)
print("="*50)

sent_1000 = final['Text'].values[1000]
print(sent_1000)
print("="*50)

sent_1500 = final['Text'].values[1500]
print(sent_1500)
print("="*50)

sent_4900 = final['Text'].values[4900]
print(sent_4900)
print("="*50)
```

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

In [15]:

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

print(sent_0)
```

```
My dogs loves this chicken but its a product from China, so we wont be buying it anymore.  Its ver
y hard to find any chicken products made in the USA but they are out there, but this one isnt.  It
s too bad too because its a good product but I wont take any chances till they know what is going
on with the china imports.
```

In [16]:

```
# https://stackoverflow.com/questions/16206380/python-beautifulsoup-how-to-remove-all-tags-from-an
-element
from bs4 import BeautifulSoup

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

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

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

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

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

In [17]:

```
# https://stackoverflow.com/a/47091490/4084039
import re

def decontracted(phrase):
    # specific
    phrase = re.sub(r"won't", "will not", phrase)
    phrase = re.sub(r"can\'t", "can not", phrase)

    # general
    phrase = re.sub(r"n\'t", " not", phrase)
    phrase = re.sub(r"\'re", " are", phrase)
    phrase = re.sub(r"\'s", " is", phrase)
    phrase = re.sub(r"\'d", " would", phrase)
    phrase = re.sub(r"\'ll", " will", phrase)
    phrase = re.sub(r"\'t", " not", phrase)
    phrase = re.sub(r"\'ve", " have", phrase)
    phrase = re.sub(r"\'m", " am", phrase)
    return phrase
```

In [18]:

```
sent_1500 = decontracted(sent_1500)
print(sent_1500)
print("="*50)
```

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

In [19]:

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

```
My dogs loves this chicken but its a product from China, so we wont be buying it anymore.  Its ver
y hard to find any chicken products made in the USA but they are out there, but this one isnt.  It
s too bad too because its a good product but I wont take any chances till they know what is going
```

on with the china imports.

In [20]:

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

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

In [21]:

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

stopwords= set(['br', 'the', 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've",\
            "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', \
            'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their',\
            'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', \
            'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', \
            'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', \
            'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after',\
            'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further',\
            'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',\
            'most', 'other', 'some', 'such', 'only', 'own', 'same', 'so', 'than', 'too', 'very', \
            's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', \
            've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn',\
            "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn',\
            "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", \
            'won', "won't", 'wouldn', "wouldn't"])
```

In [22]:

```
# Combining all the above stundents
from tqdm import tqdm
preprocessed_reviews = []
# tqdm is for printing the status bar
for sentance in tqdm(final['Text'].values):
    sentance = re.sub(r"http\S+", "", sentance)
    sentance = BeautifulSoup(sentance, 'lxml').get_text()
    sentance = decontracted(sentance)
    sentance = re.sub("\S*\d\S*", "", sentance).strip()
    sentance = re.sub('[^A-Za-z]+', ' ', sentance)
    # https://gist.github.com/sebleier/554280
    sentance = ' '.join(e.lower() for e in sentance.split() if e.lower() not in stopwords)
    preprocessed_reviews.append(sentance.strip())
```

```
100%|██████████| 87773/87773 [00:40<00:00, 2189.47it/s]
```

In [23]:

```
preprocessed_reviews[1500]
```

Out[23]:

```
'way hot blood took bite jig lol'
```

[3.2] Preprocessing Review Summary

In [24]:

```
## Similartly you can do preprocessing for review summary also.
```

[4] Featurization

[4.1] BAG OF WORDS

In [25]:

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

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

[4.2] Bi-Grams and n-Grams.

In [26]:

```
# #bi-gram, tri-gram and n-gram

# #removing stop words like "not" should be avoided before building n-grams
# # count_vect = CountVectorizer(ngram_range=(1,2))
# # please do read the CountVectorizer documentation http://scikit-
learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

# # you can choose these numebrs min_df=10, max_features=5000, of your choice
# count_vect = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
# final_bigram_counts = count_vect.fit_transform(preprocessed_reviews)
# print("the type of count vectorizer ",type(final_bigram_counts))
# print("the shape of out text BOW vectorizer ",final_bigram_counts.get_shape())
# print("the number of unique words including both unigrams and bigrams ",
final_bigram_counts.get_shape()[1])
```

[4.3] TF-IDF

In [27]:

```
# tf_idf_vect = TfidfVectorizer(ngram_range=(1,2), min_df=10)
# tf_idf_vect.fit(preprocessed_reviews)
# print("some sample features(unique words in the corpus)",tf_idf_vect.get_feature_names()[0:10])
# print('='*50)

# final_tf_idf = tf_idf_vect.transform(preprocessed_reviews)
# print("the type of count vectorizer ",type(final_tf_idf))
# print("the shape of out text TFIDF vectorizer ",final_tf_idf.get_shape())
# print("the number of unique words including both unigrams and bigrams ",
final_tf_idf.get_shape()[1])
```

[4.4] Word2Vec

In [28]:

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

In [29]:

```
# # Using Google News Word2Vectors

# # in this project we are using a pretrained model by google
# # its 3.3G file, once you load this into your memory
# # it occupies ~9Gb, so please do this step only if you have >12G of ram
# # we will provide a pickle file wich contains a dict ,
# # and it contains all our courpus words as keys and  model[word] as values
# # To use this code-snippet, download "GoogleNews-vectors-negative300.bin"
# # from https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
# # it's 1.9GB in size.


# # http://kavita-ganesan.com/gensim-word2vec-tutorial-starter-code/#.W17SRFAzZPY
# # you can comment this whole cell
# # or change these varible according to your need

# is_your_ram_gt_16g=False
# want_to_use_google_w2v = False
# want_to_train_w2v = True

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

# elif want_to_use_google_w2v and is_your_ram_gt_16g:
#     if os.path.isfile('GoogleNews-vectors-negative300.bin'):
#         w2v_model=KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin',
binary=True)
#         print(w2v_model.wv.most_similar('great'))
#         print(w2v_model.wv.most_similar('worst'))
#     else:
#         print("you don't have gogole's word2vec file, keep want_to_train_w2v = True, to train
your own w2v ")
```

In [30]:

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

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

[4.4.1.1] Avg W2v

In [31]:

```
# # average Word2Vec
# # compute average word2vec for each review.
# sent_vectors = []; # the avg-w2v for each sentence/review is stored in this list
# for sent in tqdm(list_of_sentance): # for each review/sentence
#     sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might need to change th
is to 300 if you use google's w2v
#     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_vec /= cnt_words
#     sent_vectors.append(sent_vec)
# print(len(sent_vectors))
# print(len(sent_vectors[0]))
```

[4.4.1.2] TFIDF weighted W2v

In [32]:

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

In [33]:

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

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

[5] Assignment 8: Decision Trees

1. **Apply Decision Trees on these feature sets**

    - SET 1:Review text, preprocessed one converted into vectors using (BOW)
    - SET 2:Review text, preprocessed one converted into vectors using (TFIDF)
    - SET 3:Review text, preprocessed one converted into vectors using (AVG W2v)
    - SET 4:Review text, preprocessed one converted into vectors using (TFIDF W2v)

2. **The hyper paramter tuning (best `depth` in range [1, 5, 10, 50, 100, 500, 100], and the best `min_samples_split` in range [5, 10, 100, 500])**

    - Find the best hyper parameter which will give the maximum AUC value
    - Find the best hyper paramter using k-fold cross validation or simple cross validation data
    - Use gridsearch cv or randomsearch cv or you can also write your own for loops to do this task of hyperparameter tuning

3. **Graphviz**

    - Visualize your decision tree with Graphviz. It helps you to understand how a decision is being made, given a new vector.
    - Since feature names are not obtained from word2vec related models, visualize only BOW & TFIDF decision trees using Graphviz
    - Make sure to print the words in each node of the decision tree instead of printing its index.
    - Just for visualization purpose, limit max_depth to 2 or 3 and either embed the generated images of graphviz in your notebook, or directly upload them as .png files.

4. **Feature importance**

- Find the top 20 important features from both feature sets Set 1 and Set 2 using `feature_importances_` method of Decision Tree Classifier and print their corresponding feature names

5. **Feature engineering**

   - To increase the performance of your model, you can also experiment with with feature engineering like :
     - Taking length of reviews as another feature.
     - Considering some features from review summary as well.

6. **Representation of results**

   - You need to plot the performance of model both on train data and cross validation data for each hyper parameter, like shown in the figure.
   - Once after you found the best hyper parameter, you need to train your model with it, and find the AUC on test data and plot the ROC curve on both train and test.
   - Along with plotting ROC curve, you need to print the confusion matrix with predicted and original labels of test data points. Please visualize your confusion matrices using seaborn heatmaps.

7. **Conclusion**

   - You need to summarize the results at the end of the notebook, summarize it in the table format. To print out a table please refer to this prettytable library link

Note: Data Leakage

1. There will be an issue of data-leakage if you vectorize the entire data and then split it into train/cv/test.
2. To avoid the issue of data-leakag, make sure to split your data first and then vectorize it.
3. While vectorizing your data, apply the method fit_transform() on you train data, and apply the method transform() on cv/test data.
4. For more details please go through this link.

Applying Decision Trees

In [34]:

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
import pprint
import os.path
import pickle
import math

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

[5.0.0] Splitting up the Dataset into D_train and D_test

In [35]:

```
num_data_points = 100000
```

In [36]:

```
Dx_train, Dx_test, Dy_train, Dy_test = train_test_split(preprocessed_reviews[:num_data_points],
final['Score'].tolist()[:num_data_points], test_size=0.30, random_state=42)
```

In [37]:

```
prettytable_data = []
```

[5.0.1] Defining some functions to increase code reusability and readability

In [38]:

```
'''Creating Custom Vectorizers for TFIDF - W2Vec and Avg - W2Vec'''
class Tfidf_W2Vec_Vectorizer(object):
    def __init__(self, w2vec_model):
        if(w2v_model is None):
            raise Exception('Word 2 Vector model passed to Tfidf_W2Vec Vectorizer is None !')
        self.tfidf = TfidfVectorizer(max_features=300)
        self.dictionary = None
        self.tfidf_feat = None

        self.word2vec = w2vec_model

    def fit(self, X):
        '''X : list'''
        #Initializing the TFIDF Vectorizer
        self.tfidf.fit_transform(X)
        # we are converting a dictionary with word as a key, and the idf as a value
        self.dictionary = dict(zip(self.tfidf.get_feature_names(), list(self.tfidf.idf_)))
        self.tfidf_feat = self.tfidf.get_feature_names()

        return self

    def transform(self, X):
        '''X : list'''
        return np.array([
                np.mean([self.word2vec[w] * self.dictionary[word]*(X.cout(word)/len(X))
                         for w in words if w in self.word2vec and w in self.tfidf_feat] or
                        [np.zeros(300)], axis=0)
                for words in X
            ])

class Avg_W2Vec_Vectorizer(object):
    def __init__(self, w2vec_model):
        if(w2v_model is None):
            raise Exception('Word 2 Vector model passed to Avg_W2Vec Vectorizer is None !')
        self.word2vec = w2vec_model

    def fit(self, X):
        return self

    def transform(self, X):
        '''X : list'''
        return np.array([
            np.mean([self.word2vec[w] for w in words if w in self.word2vec]
                    or [np.zeros(300)], axis=0)
            for words in X
        ])
```

In [39]:

```
def get_vectorizer(vectorizer, train, W2V_model=None):
    if(vectorizer=='BOW'):
        vectorizer = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
    if(vectorizer=='TFIDF'):
        vectorizer = TfidfVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
    if(vectorizer=='TFIDF-W2Vec'):
        vectorizer = Tfidf_W2Vec_Vectorizer(W2V_model)
    if(vectorizer=='Avg-W2Vec'):
        vectorizer = Avg_W2Vec_Vectorizer(W2V_model)

    vectorizer.fit(train)
    return vectorizer
```

In [40]:

```
'''Perform Simple Cross Validation'''
def perform_hyperparameter_tuning(X, Y, vectorizer, results_path, retrain=False, W2V_model=None):
    #If the pandas dataframe with the hyperparameter info exists then return it

    if(retrain==False):
```

```
        # If Cross Validation results exists then return them
        if(os.path.exists(results_path)):
            return pd.read_csv(results_path)
        else:
            # If no data exists but retrain=False then mention accordingly
            print('Retrain is set to be False but no Cross Validation Results DataFrame was found !
\nPlease set retrain to True.')
    else:
        # else perform hyperparameter tuning
        print('Performing Hyperparameter Tuning...\n')
        # regularization parameter
        hyperparameters = {
            'dt_max_depth' : [1, 5, 10, 50, 100, 500, 1000],
            'dt_min_samples_split' : [5, 10, 100, 500]
        }

        max_depth = []
        min_samples_split = []

        train_scores = []
        test_scores = []

        train_mean_score = []
        test_mean_score = []

        # Initializing KFold
        skf = StratifiedKFold(n_splits=3)
        X = np.array(X)
        Y = np.array(Y)

        for depth in hyperparameters['dt_max_depth']:
            for min_samples in hyperparameters['dt_min_samples_split']:

                #Performing Cross Validation
                for train_index, test_index in skf.split(X, Y):
                    Dx_train, Dx_cv = X[train_index], X[test_index]
                    Dy_train, Dy_cv = Y[train_index], Y[test_index]

                    #Initializing the Vectorizer
                    vectorizer = get_vectorizer(vectorizer, Dx_train.tolist(), W2V_model)

                    #Transforming the data to features
                    x_train = vectorizer.transform(Dx_train.tolist())
                    x_cv = vectorizer.transform(Dx_cv.tolist())

                    #Initializing the LR model
                    dt_clf = DecisionTreeClassifier(max_depth=depth, min_samples_split=min_samples,
class_weight='balanced')

                    # Fit the model
                    dt_clf.fit(x_train, Dy_train)

                    #Prediction
                    train_results = dt_clf.predict_proba(x_train)
                    cv_results = dt_clf.predict_proba(x_cv)

                    try:
                        train_score = roc_auc_score(Dy_train, train_results[:, 1])
                        test_score = roc_auc_score(Dy_cv, cv_results[:, 1])

                        #storing the results to form a dataframe
                        train_scores.append(train_score)
                        test_scores.append(test_score)

                    except Exception as e:
                        print('Error Case : ', e)
                        print(('Actual, Predicted'))
                        [print((Dy_cv[i], cv_results[i, 1])) for i in range(len(Dy_cv))]

                    print('CV iteration : depth={0}, min_samples={1}, train_score={2}, test_score={
3}'
                      .format(depth, min_samples, train_score, test_score))

                train_mean_score.append(sum(train_scores)/len(train_scores))
                test_mean_score.append(sum(test_scores)/len(test_scores))

                max_depth.append(depth)
```

```
                min_samples_split.append(min_samples)

                print('CV : depth={0}, min_samples_split={1}, train_score={2}, test_score={3}'
                      .format(depth, min_samples, sum(train_scores)/len(train_scores), sum(test_sco
es)/len(test_scores)))

                train_scores = []
                test_scores = []

        # Creating a DataFrame from the saved data for visualization
        results_df = pd.DataFrame({
            'max_depth' : max_depth,
            'min_samples_split' : min_samples_split,
            'train_score' : train_mean_score,
            'test_score' : test_mean_score
        })

        #writing the results to csv after performing hyperparameter tuning
        try:
            results_df.to_csv(results_path)
        except Exception as ex:
            print(str(ex), "\nError occured while converting DataFrame to CSV after cross validatio
n.")
        return results_df
```

In [41]:

```
def analyse_results(df):
    # plotting error curves
    fig = plt.figure(figsize=(15, 15))
    ax = fig.gca()

    unique_min_samples = np.unique(df['min_samples_split'].values)
    c = 1
    for i in unique_min_samples:
        mini = df.loc[df['min_samples_split'] == i]
        plt.subplot(len(unique_min_samples)//2, len(unique_min_samples)//2, c)
        plt.plot([math.log10(i) for i in mini.max_depth.tolist()], mini.train_score.tolist(), '-o',
c='r', label='Train AUC')
        plt.plot([math.log10(i) for i in mini.max_depth.tolist()], mini.test_score.tolist(), '-o',
c='b', label='Validation AUC')
        plt.grid(True)
        plt.xlabel('log10 of Hyperparameter : max_depth')
        plt.ylabel('Area Under ROC Curve')
        plt.title('AUC ROC : min_samples_split = {0}'.format(i))
        plt.legend(loc='best')
        c = c + 1

    plt.show()

    # return the best parameters
    mmax = 0
    ind_max = 0
    for index, row in df.iterrows():
        if(row['test_score']>mmax):
            mmax=row['test_score']
            ind_max = index


    best_params = {
        'max_depth': df.loc[ind_max, 'max_depth'],
        'min_samples_split':df.loc[ind_max, 'min_samples_split']
    }

    return best_params
```

In [42]:

```
def retrain_with_best_params(data, labels, best_params, vec_name, model_path, word2vec):
    if(os.path.exists(model_path)):
        print('Loading Model....')
        with open(model_path, 'rb') as input_file:
            dt_clf = pickle.load(input_file)
    else:
        dt_clf = DecisionTreeClassifier(max_depth=best_params['max_depth'], min_samples_split=best
```

```
params['min_samples_split'])

        print('Initializing Vectorizer')
        vectorizer = get_vectorizer(vectorizer=vec_name, train=data, W2V_model=word2vec)
        print('Training Model....')
        dt_clf.fit(vectorizer.transform(data), np.array(labels))

        print('Saving Trained Model....')
        with open(model_path,'wb') as file:
            pickle.dump(dt_clf, file)
    return dt_clf
```

In [43]:

```
def plot_confusion_matrix(model, data, labels, dataset_label):
    pred = model.predict(data)
    conf_mat = confusion_matrix(labels, pred)

    strings = strings = np.asarray([['TN = ', 'FP = '],
                                    ['FN = ', 'TP = ']])

    labels = (np.asarray(["{0}{1}".format(string, value)
                          for string, value in zip(strings.flatten(),
                                                   conf_mat.flatten())])
             ).reshape(2, 2)

    fig, ax = plt.subplots()
    ax.set(xlabel='Predicted', ylabel='Actual', title='Confusion Matrix : {0}'.format(dataset_label
))
    sns.heatmap(conf_mat, annot=labels, fmt="", cmap='YlGnBu', ax=ax)
    ax.set_xlabel('Predicted')
    ax.set_ylabel('Actual')
    ax.set_xticklabels(['False', 'True'])
    ax.set_yticklabels(['False', 'True'])
    plt.show()
```

In [44]:

```
def plot_AUC_ROC(model, vectorizer, Dx_train, Dx_test, Dy_train, Dy_test):

    #predicting probability of Dx_test, Dx_train
    test_score = model.predict_proba(vectorizer.transform(Dx_test))
    train_score = model.predict_proba(vectorizer.transform(Dx_train))

    #Finding out the ROC_AUC_SCORE
    train_roc_auc_score = roc_auc_score(np.array(Dy_train), train_score[:, 1])
    print('Area Under the Curve for Train : ', train_roc_auc_score)
    test_roc_auc_score = roc_auc_score(np.array(Dy_test), test_score[:, 1])
    print('Area Under the Curve for Test : ', test_roc_auc_score)

    #Plotting with matplotlib.pyplot
    #ROC Curve for D-train
    train_fpr, train_tpr, thresholds = roc_curve(np.array(Dy_train), train_score[:, 1])
    plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, train_tpr)))

    # #ROC Curve for D-test
    test_fpr, test_tpr, thresholds = roc_curve(np.array(Dy_test), test_score[:, 1])
    plt.plot(test_fpr, test_tpr, label="train AUC ="+str(auc(test_fpr, test_tpr)))

    plt.legend()
    plt.xlabel("FPR : False Positive Ratio")
    plt.ylabel("TPF : True Positive Ratio")
    plt.title("Area Under ROC Curve")
    plt.show()

    plot_confusion_matrix(model, vectorizer.transform(Dx_train), np.array(Dy_train), 'Training')
    plot_confusion_matrix(model, vectorizer.transform(Dx_test), np.array(Dy_test), 'Testing')
    return train_roc_auc_score, test_roc_auc_score
```

[5.1] Applying Decision Trees on BOW, SET 1

In [45]:

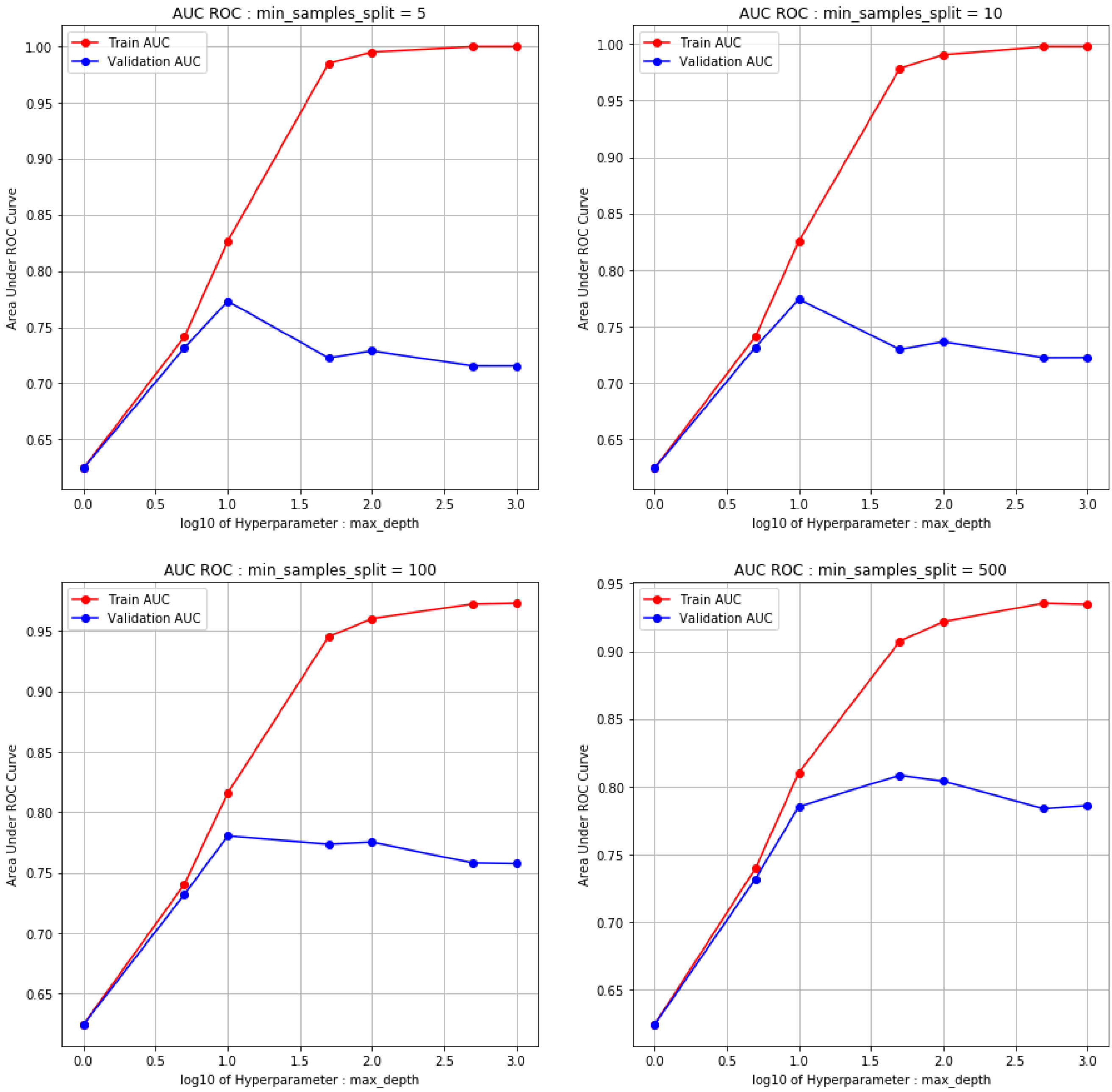
```python
# Please write all the code with proper documentation
csv_path = 'saved_models/Assignment8/BOW_dtree_results.csv'
cv_results = perform_hyperparameter_tuning(X=Dx_train, Y=Dy_train, vectorizer='BOW',
                                           results_path=csv_path, retrain=False, W2V_model=None)
# Analysing best parameters
best_parameters = analyse_results(cv_results)
pprint.pprint(best_parameters)

# retraining the model with best parameters
model_path = 'saved_models/Assignment8/{0}_dtree.pkl'.format('BOW')
clf = retrain_with_best_params(Dx_train, Dy_train, best_parameters, 'BOW', model_path, None)

print('Retraining Vectorizer with Dx_train')
vectorizer_obj = get_vectorizer(W2V_model = None, train=Dx_train, vectorizer='BOW')

# plotting AUC ROC
train_score, test_score = plot_AUC_ROC(clf, vectorizer_obj, Dx_train, Dx_test, Dy_train, Dy_test)

# appending the data results
prettytable_data.append(['BOW', 'Decision Trees', best_parameters['max_depth'], best_parameters['m
in_samples_split'], train_score, test_score])
```

```
Performing Hyperparameter Tuning...

CV iteration : depth=1, min_samples=5, train_score=0.6250930906943508,
test_score=0.6231161498667342
CV iteration : depth=1, min_samples=5, train_score=0.6257707226233483,
test_score=0.6217613166133636
CV iteration : depth=1, min_samples=5, train_score=0.622438733240049,
test_score=0.6284257248067614
CV : depth=1, min_samples_split=5, train_score=0.6244341821859161, test_score=0.6244343970956198
CV iteration : depth=1, min_samples=10, train_score=0.6250930906943508,
test_score=0.6231161498667342
CV iteration : depth=1, min_samples=10, train_score=0.6257707226233483,
test_score=0.6217613166133636
CV iteration : depth=1, min_samples=10, train_score=0.622438733240049,
test_score=0.6284257248067614
CV : depth=1, min_samples_split=10, train_score=0.6244341821859161, test_score=0.6244343970956198
CV iteration : depth=1, min_samples=100, train_score=0.6250930906943508,
test_score=0.6231161498667342
CV iteration : depth=1, min_samples=100, train_score=0.6257707226233483,
test_score=0.6217613166133636
CV iteration : depth=1, min_samples=100, train_score=0.622438733240049,
test_score=0.6284257248067614
CV : depth=1, min_samples_split=100, train_score=0.6244341821859161, test_score=0.6244343970956198
CV iteration : depth=1, min_samples=500, train_score=0.6250930906943508,
test_score=0.6231161498667342
CV iteration : depth=1, min_samples=500, train_score=0.6257707226233483,
test_score=0.6217613166133636
CV iteration : depth=1, min_samples=500, train_score=0.622438733240049,
test_score=0.6284257248067614
CV : depth=1, min_samples_split=500, train_score=0.6244341821859161, test_score=0.6244343970956198
CV iteration : depth=5, min_samples=5, train_score=0.7404648534009576,
test_score=0.7306307286868274
CV iteration : depth=5, min_samples=5, train_score=0.7445013094563706,
test_score=0.729824284737596
CV iteration : depth=5, min_samples=5, train_score=0.7384259457973434,
test_score=0.7338285663133953
CV : depth=5, min_samples_split=5, train_score=0.7411307028848905, test_score=0.7314278599126062
CV iteration : depth=5, min_samples=10, train_score=0.7404067609475136,
test_score=0.7310502982143388
CV iteration : depth=5, min_samples=10, train_score=0.7445013094563706,
test_score=0.729824284737596
CV iteration : depth=5, min_samples=10, train_score=0.7384259457973434,
test_score=0.7339612228952521
CV : depth=5, min_samples_split=10, train_score=0.7411113387337425, test_score=0.7316119352823955
CV iteration : depth=5, min_samples=100, train_score=0.7396865368665517,
test_score=0.731308679633853
CV iteration : depth=5, min_samples=100, train_score=0.7438928469878299,
test_score=0.7296387651826904
CV iteration : depth=5, min_samples=100, train_score=0.7376276005547604,
test_score=0.7348619094016773
CV : depth=5, min_samples_split=100, train_score=0.7404023281363807, test_score=0.7319364514060736
CV iteration : depth=5, min_samples=500, train_score=0.7395141078823898,
test_score=0.7311015753035408
CV iteration : depth=5, min_samples=500, train_score=0.7427654148157379
```

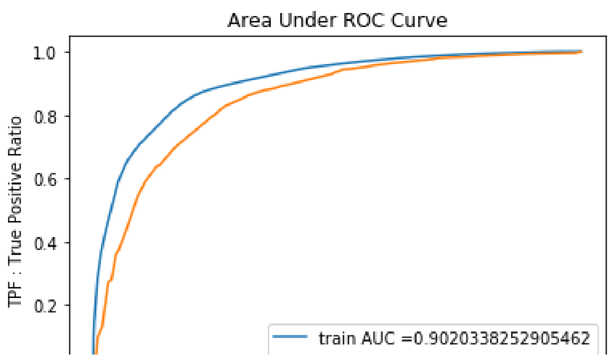
```
CV iteration : depth=5, min_samples=500, train_score=0.7427654148157579,
test_score=0.7293313499362296
CV iteration : depth=5, min_samples=500, train_score=0.7367444573493848,
test_score=0.735307146061196
CV : depth=5, min_samples_split=500, train_score=0.7396746600158375, test_score=0.7319133571003222
CV iteration : depth=10, min_samples=5, train_score=0.8278626308560338,
test_score=0.7757935946490105
CV iteration : depth=10, min_samples=5, train_score=0.8261891415465756,
test_score=0.7706868257870874
CV iteration : depth=10, min_samples=5, train_score=0.8252784784881217,
test_score=0.7727523398535996
CV : depth=10, min_samples_split=5, train_score=0.8264434169635771, test_score=0.7730775867632325
CV iteration : depth=10, min_samples=10, train_score=0.8266814632749232,
test_score=0.7757917576468484
CV iteration : depth=10, min_samples=10, train_score=0.825217167306235,
test_score=0.7704108250439494
CV iteration : depth=10, min_samples=10, train_score=0.8242765726513689,
test_score=0.775953714682913
CV : depth=10, min_samples_split=10, train_score=0.8253917344108422, test_score=0.7740520991245704
CV iteration : depth=10, min_samples=100, train_score=0.8167120134217637,
test_score=0.7812891279252625
CV iteration : depth=10, min_samples=100, train_score=0.8162972130758274,
test_score=0.7768412069542261
CV iteration : depth=10, min_samples=100, train_score=0.8147362432295431,
test_score=0.7831747473888255
CV : depth=10, min_samples_split=100, train_score=0.8159151565757113,
test_score=0.7804350274227714
CV iteration : depth=10, min_samples=500, train_score=0.8115280160981948,
test_score=0.7856792451503762
CV iteration : depth=10, min_samples=500, train_score=0.8099830253700869,
test_score=0.7809519673745591
CV iteration : depth=10, min_samples=500, train_score=0.8097111061512052,
test_score=0.7893037288488512
CV : depth=10, min_samples_split=500, train_score=0.8104073825398289,
test_score=0.7853116471245954
CV iteration : depth=50, min_samples=5, train_score=0.984495281850888,
test_score=0.7148181231850551
CV iteration : depth=50, min_samples=5, train_score=0.9849798788243449,
test_score=0.7376041213708742
CV iteration : depth=50, min_samples=5, train_score=0.9845696565431243,
test_score=0.7153338356996416
CV : depth=50, min_samples_split=5, train_score=0.9846816057394524, test_score=0.7225853600851903
CV iteration : depth=50, min_samples=10, train_score=0.9770264906225207,
test_score=0.7234866714365135
CV iteration : depth=50, min_samples=10, train_score=0.979464386165912,
test_score=0.7455669075223791
CV iteration : depth=50, min_samples=10, train_score=0.9786693093709118,
test_score=0.719677444218062
CV : depth=50, min_samples_split=10, train_score=0.9783867287197815, test_score=0.7295770077256515
CV iteration : depth=50, min_samples=100, train_score=0.9442862565202244,
test_score=0.7597558415697268
CV iteration : depth=50, min_samples=100, train_score=0.9460285111648832,
test_score=0.7869458669729938
CV iteration : depth=50, min_samples=100, train_score=0.9454442063760296,
test_score=0.773961507026832
CV : depth=50, min_samples_split=100, train_score=0.9452529913537123,
test_score=0.7735544051898509
CV iteration : depth=50, min_samples=500, train_score=0.9084090321402849,
test_score=0.8018522121868913
CV iteration : depth=50, min_samples=500, train_score=0.9073983413754334,
test_score=0.8142068907434838
CV iteration : depth=50, min_samples=500, train_score=0.9060628653404289,
test_score=0.8093342277588144
CV : depth=50, min_samples_split=500, train_score=0.9072900796187158,
test_score=0.8084644435630631
CV iteration : depth=100, min_samples=5, train_score=0.9945468927977287,
test_score=0.7282509718712931
CV iteration : depth=100, min_samples=5, train_score=0.995786132957271,
test_score=0.7306781021464346
CV iteration : depth=100, min_samples=5, train_score=0.9941060403845849,
test_score=0.727721575012027
CV : depth=100, min_samples_split=5, train_score=0.9948130220465282, test_score=0.7288835496765849
CV iteration : depth=100, min_samples=10, train_score=0.9904294328160335,
test_score=0.732613869670137
CV iteration : depth=100, min_samples=10, train_score=0.9913282140100368,
test_score=0.7378120293896324
CV iteration : depth=100, min_samples=10, train_score=0.9900144183473558,
test_score=0.7387040645552605
```

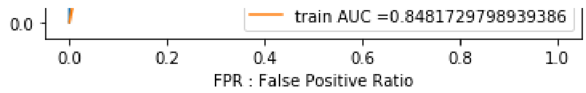
```
test_score=0.7387040645552605
CV : depth=100, min_samples_split=10, train_score=0.9905906883911421,
test_score=0.7363766545383433
CV iteration : depth=100, min_samples=100, train_score=0.9588681314600995,
test_score=0.7683152913225777
CV iteration : depth=100, min_samples=100, train_score=0.9613826672603436,
test_score=0.7788932796773264
CV iteration : depth=100, min_samples=100, train_score=0.9601620535303138,
test_score=0.7789433160678634
CV : depth=100, min_samples_split=100, train_score=0.9601376174169189,
test_score=0.7753839623559226
CV iteration : depth=100, min_samples=500, train_score=0.9230035785048489,
test_score=0.7984690724258946
CV iteration : depth=100, min_samples=500, train_score=0.9222500726111502,
test_score=0.8066058880370498
CV iteration : depth=100, min_samples=500, train_score=0.9196538110252561,
test_score=0.8072125264283894
CV : depth=100, min_samples_split=500, train_score=0.9216358207137517,
test_score=0.8040958289637778
CV iteration : depth=500, min_samples=5, train_score=0.9995856832793696,
test_score=0.7131904597836245
CV iteration : depth=500, min_samples=5, train_score=0.9995738731052263,
test_score=0.7153249768104972
CV iteration : depth=500, min_samples=5, train_score=0.9996003548805561,
test_score=0.7177614820699014
CV : depth=500, min_samples_split=5, train_score=0.999586637088384, test_score=0.7154256395546744
CV iteration : depth=500, min_samples=10, train_score=0.9974039943148454,
test_score=0.7233277795812194
CV iteration : depth=500, min_samples=10, train_score=0.9978542185582082,
test_score=0.7194865285742684
CV iteration : depth=500, min_samples=10, train_score=0.9978328585212649,
test_score=0.7241794101506148
CV : depth=500, min_samples_split=10, train_score=0.9976970237981062,
test_score=0.7223312394353676
CV iteration : depth=500, min_samples=100, train_score=0.9725195974359485,
test_score=0.748433293276088
CV iteration : depth=500, min_samples=100, train_score=0.9715866643453241,
test_score=0.7618716882957774
CV iteration : depth=500, min_samples=100, train_score=0.9730726438265829,
test_score=0.763953714258836
CV : depth=500, min_samples_split=100, train_score=0.9723929685359519,
test_score=0.7580862319435671
CV iteration : depth=500, min_samples=500, train_score=0.9382730537966992,
test_score=0.7781629565051937
CV iteration : depth=500, min_samples=500, train_score=0.9353552539911554,
test_score=0.7841063650393901
CV iteration : depth=500, min_samples=500, train_score=0.9328414000514148,
test_score=0.7889863248954854
CV : depth=500, min_samples_split=500, train_score=0.9354899026130897,
test_score=0.7837518821466899
CV iteration : depth=1000, min_samples=5, train_score=0.9995685112684983,
test_score=0.7110081453735684
CV iteration : depth=1000, min_samples=5, train_score=0.9995760328565935,
test_score=0.7177232449555127
CV iteration : depth=1000, min_samples=5, train_score=0.9996200739506417,
test_score=0.7176176404575663
CV : depth=1000, min_samples_split=5, train_score=0.9995882060252446,
test_score=0.7154496769288824
CV iteration : depth=1000, min_samples=10, train_score=0.9973247128896536,
test_score=0.7207047082965977
CV iteration : depth=1000, min_samples=10, train_score=0.9977591364980134,
test_score=0.7226325714119577
CV iteration : depth=1000, min_samples=10, train_score=0.9978789425466623,
test_score=0.7239590138075579
CV : depth=1000, min_samples_split=10, train_score=0.9976542639781097,
test_score=0.7224320978387045
CV iteration : depth=1000, min_samples=100, train_score=0.9726961449643845,
test_score=0.7498571642501433
CV iteration : depth=1000, min_samples=100, train_score=0.9731039537225613,
test_score=0.7620144622234459
CV iteration : depth=1000, min_samples=100, train_score=0.973466407006411,
test_score=0.760624612742615
CV : depth=1000, min_samples_split=100, train_score=0.973088835231119,
test_score=0.7574987464054016
CV iteration : depth=1000, min_samples=500, train_score=0.9359806102203914,
test_score=0.7798915490446546
CV iteration : depth=1000, min_samples=500, train_score=0.935624845286829,
test_score=0.7856067408206234
```

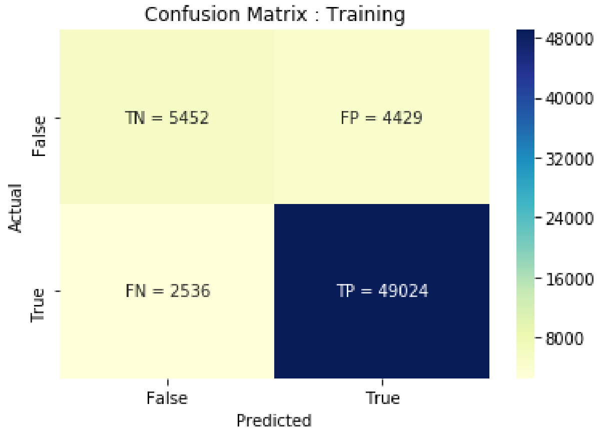
```
test_score=0.7856967408296234
CV iteration : depth=1000, min_samples=500, train_score=0.9323087709893838,
test_score=0.7929144086481454
CV : depth=1000, min_samples_split=500, train_score=0.9346380754988681,
test_score=0.7861675661741412
```

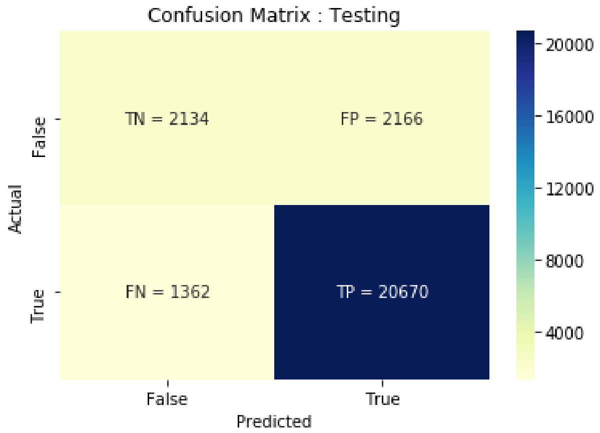



```
{'max_depth': 50, 'min_samples_split': 500}
Initializing Vectorizer
Training Model....
Saving Trained Model....
Retraining Vectorizer with Dx_train
Area Under the Curve for Train :  0.9020338252905462
Area Under the Curve for Test :  0.8481729798939386
```









[5.1.1] Top 20 important features from SET 1

In [46]:

```
clf1 = DecisionTreeClassifier(random_state=0)
vectorizer_obj1 = get_vectorizer(W2V_model = None, train=Dx_train, vectorizer='BOW')
clf1.fit(vectorizer_obj1.transform(Dx_train), np.array(Dy_train))
```

Out[46]:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=0,
            splitter='best')
```

In [47]:

```
feature_importance = clf1.feature_importances_
features = vectorizer_obj1.get_feature_names()
features_with_names = [(features[i], feature_importance[i]) for i in range(feature_importance.shape
[0]) if feature_importance[i]>0]
features_with_names.sort(key=lambda x: x[1], reverse=True)
features_with_names[:20]
```

Out[47]:

```
[('not', 0.03789421537218232),
 ('great', 0.024372247031116144),
 ('disappointed', 0.01590932693459837),
 ('not buy', 0.015736499956928187),
 ('money', 0.013322794154553919),
```

```
 ('worst', 0.011989099703382 8),
 ('horrible', 0.011930031773261929),
 ('good', 0.010272861322536 95),
 ('return', 0.009826070015016357),
 ('best', 0.009476224457512164),
 ('delicious', 0.009163800137051908),
 ('not disappointed', 0.00906987831120911),
 ('not recommend', 0.008992242315023737),
 ('awful', 0.008884596965140311),
 ('love', 0.00831565566432052),
 ('not worth', 0.008000935963592432),
 ('not good', 0.006321453599875989),
 ('loves', 0.005971119116773436),
 ('bad', 0.00585598714676059),
 ('terrible', 0.005721904233097538)]
```

In [48]:

```
del clf1
del vectorizer_obj1
del feature_importance
del features
del features_with_names
```

[5.1.2] Graphviz visualization of Decision Tree on BOW, SET 2

In [49]:

```
from sklearn import tree
from graphviz import Source
```

In [50]:

```
dot_data = tree.export_graphviz(clf, feature_names = vectorizer_obj.get_feature_names(),filled=True
)
graph = Source(dot_data)
graph
```

Out[50]:

[5.2] Applying Decision Trees on TFIDF, SET 2

In [51]:

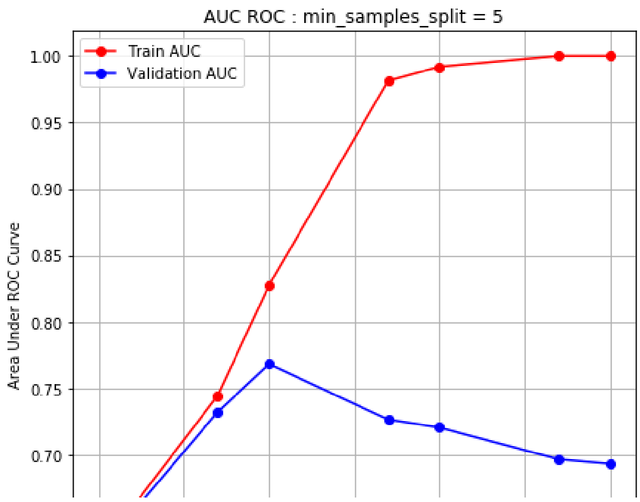
```
# Please write all the code with proper documentation
csv_path = 'saved_models/Assignment8/TFIDF_dtree_results.csv'
cv_results = perform_hyperparameter_tuning(X=Dx_train, Y=Dy_train, vectorizer='TFIDF',
                                            results_path=csv_path, retrain=False, W2V_model=None)
# Analysing best parameters
best_parameters = analyse_results(cv_results)
pprint.pprint(best_parameters)

# retraining the model with best parameters
model_path = 'saved_models/Assignment8/{0}_dtree.pkl'.format('TFIDF')
clf = retrain_with_best_params(Dx_train, Dy_train, best_parameters, 'TFIDF', model_path, None)

print('Retraining Vectorizer with Dx_train')
vectorizer_obj = get_vectorizer(W2V_model = None, train=Dx_train, vectorizer='TFIDF')

# plotting AUC ROC
train_score, test_score = plot_AUC_ROC(clf, vectorizer_obj, Dx_train, Dx_test, Dy_train, Dy_test)

# appending the data results
prettytable_data.append(['TFIDF', 'Decision Trees', best_parameters['max_depth'], best_parameters[
'min_samples_split'], train_score, test_score])
```

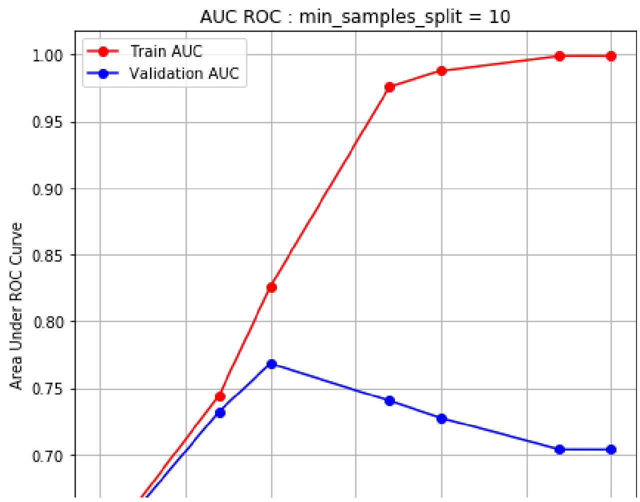
```
Performing Hyperparameter Tuning...
```

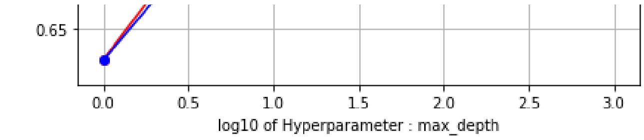
```
CV iteration : depth=1, min_samples=5, train_score=0.628082562003602,
test_score=0.6255606786719704
CV iteration : depth=1, min_samples=5, train_score=0.6286244630550037,
test_score=0.6245612788417729
CV iteration : depth=1, min_samples=5, train_score=0.6256104631969158,
test_score=0.630557259422275
CV : depth=1, min_samples_split=5, train_score=0.6274391627518406, test_score=0.6268930723120062
CV iteration : depth=1, min_samples=10, train_score=0.628082562003602,
test_score=0.6255606786719704
CV iteration : depth=1, min_samples=10, train_score=0.6286244630550037,
test_score=0.6245612788417729
CV iteration : depth=1, min_samples=10, train_score=0.6256104631969158,
test_score=0.630557259422275
CV : depth=1, min_samples_split=10, train_score=0.6274391627518406, test_score=0.6268930723120062
CV iteration : depth=1, min_samples=100, train_score=0.628082562003602,
test_score=0.6255606786719704
CV iteration : depth=1, min_samples=100, train_score=0.6286244630550037,
test_score=0.6245612788417729
CV iteration : depth=1, min_samples=100, train_score=0.6256104631969158,
test_score=0.630557259422275
CV : depth=1, min_samples_split=100, train_score=0.6274391627518406, test_score=0.6268930723120062
CV iteration : depth=1, min_samples=500, train_score=0.628082562003602,
test_score=0.6255606786719704
CV iteration : depth=1, min_samples=500, train_score=0.6286244630550037,
test_score=0.6245612788417729
CV iteration : depth=1, min_samples=500, train_score=0.6256104631969158,
test_score=0.630557259422275
CV : depth=1, min_samples_split=500, train_score=0.6274391627518406, test_score=0.6268930723120062
CV iteration : depth=5, min_samples=5, train_score=0.7436590815065035,
test_score=0.735021817403469
CV iteration : depth=5, min_samples=5, train_score=0.7476589896221121,
test_score=0.7297856723652238
CV iteration : depth=5, min_samples=5, train_score=0.7429283453637545,
test_score=0.7323844693254338
CV : depth=5, min_samples_split=5, train_score=0.7447488054974567, test_score=0.7323973196980421
CV iteration : depth=5, min_samples=10, train_score=0.7436590815065035,
test_score=0.735021817403469
CV iteration : depth=5, min_samples=10, train_score=0.7475978827917596,
test_score=0.7307574818360228
CV iteration : depth=5, min_samples=10, train_score=0.7428207328055979,
test_score=0.731889182746753
CV : depth=5, min_samples_split=10, train_score=0.744692565701287, test_score=0.7325561606620816
CV iteration : depth=5, min_samples=100, train_score=0.7431747340748711,
test_score=0.7358739037910391
CV iteration : depth=5, min_samples=100, train_score=0.7471635585584627,
test_score=0.7307333358556787
CV iteration : depth=5, min_samples=100, train_score=0.7421101839019332,
test_score=0.7343299136589861
CV : depth=5, min_samples_split=100, train_score=0.7441494921784223, test_score=0.7336457177685679
CV iteration : depth=5, min_samples=500, train_score=0.7423184589077778,
test_score=0.7379673002310491
CV iteration : depth=5, min_samples=500, train_score=0.7456324207141249,
test_score=0.7305178325395189
CV iteration : depth=5, min_samples=500, train_score=0.7410257344043197,
test_score=0.733895455622835
CV : depth=5, min_samples_split=500, train_score=0.7429922046754075, test_score=0.734126862797801
CV iteration : depth=10, min_samples=5, train_score=0.8300455763630203,
test_score=0.7664054361274525
CV iteration : depth=10, min_samples=5, train_score=0.8280412122607573,
test_score=0.7663511827414777
CV iteration : depth=10, min_samples=5, train_score=0.8245373430568684,
test_score=0.7723015460185904
CV : depth=10, min_samples_split=5, train_score=0.8275413772268821, test_score=0.7683527216291736
CV iteration : depth=10, min_samples=10, train_score=0.8287038783052801,
test_score=0.7689877312631167
CV iteration : depth=10, min_samples=10, train_score=0.8269508403621276,
test_score=0.764879832326921l
CV iteration : depth=10, min_samples=10, train_score=0.8227745010251709,
test_score=0.7713237481804005
CV : depth=10, min_samples_split=10, train_score=0.8261430732308596, test_score=0.7683971039234795
CV iteration : depth=10, min_samples=100, train_score=0.8191634261820456,
test_score=0.7740320332197819
CV iteration : depth=10, min_samples=100, train_score=0.8181985053628371,
test_score=0.7747071845048585
CV iteration : depth=10, min_samples=100, train_score=0.814666512217177,
test_score=0.7787875296204523
```

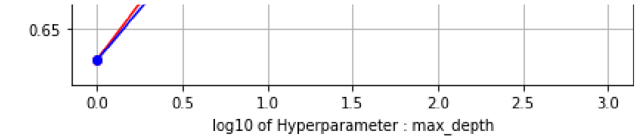
```
CV : depth=10, min_samples_split=100, train_score=0.8173428145873533,
test_score=0.7758422491150309
CV iteration : depth=10, min_samples=500, train_score=0.8135801928557271,
test_score=0.7777370987779733
CV iteration : depth=10, min_samples=500, train_score=0.8131322100721166,
test_score=0.7785932124395145
CV iteration : depth=10, min_samples=500, train_score=0.8092522221278994,
test_score=0.7822099369083001
CV : depth=10, min_samples_split=500, train_score=0.8119882083519143,
test_score=0.7795134160419294
CV iteration : depth=50, min_samples=5, train_score=0.9808480381668787,
test_score=0.7144647387258319
CV iteration : depth=50, min_samples=5, train_score=0.9840773964613317,
test_score=0.7391807108131493
CV iteration : depth=50, min_samples=5, train_score=0.9787203192999439,
test_score=0.7266034872062512
CV : depth=50, min_samples_split=5, train_score=0.9812152513093847, test_score=0.726749645581744
CV iteration : depth=50, min_samples=10, train_score=0.974350105969813,
test_score=0.7408860052194177
CV iteration : depth=50, min_samples=10, train_score=0.9795042598578219,
test_score=0.7480577411465414
CV iteration : depth=50, min_samples=10, train_score=0.9733358164656792,
test_score=0.7328322239420507
CV : depth=50, min_samples_split=10, train_score=0.975730060764438, test_score=0.74059199010267
CV iteration : depth=50, min_samples=100, train_score=0.9468715009902328,
test_score=0.7608623969154755
CV iteration : depth=50, min_samples=100, train_score=0.9477646575556755,
test_score=0.7870764530978551
CV iteration : depth=50, min_samples=100, train_score=0.9441621091349561,
test_score=0.7736504818981149
CV : depth=50, min_samples_split=100, train_score=0.9462660892269548,
test_score=0.7738631106371484
CV iteration : depth=50, min_samples=500, train_score=0.9133437327643614,
test_score=0.8029573986127596
CV iteration : depth=50, min_samples=500, train_score=0.9216568432355865,
test_score=0.8070365767266875
CV iteration : depth=50, min_samples=500, train_score=0.911396285383797,
test_score=0.8008820642258233
CV : depth=50, min_samples_split=500, train_score=0.9154656204612484,
test_score=0.8036253465217569
CV iteration : depth=100, min_samples=5, train_score=0.9919511278210599,
test_score=0.7233244235195767
CV iteration : depth=100, min_samples=5, train_score=0.9931043158011239,
test_score=0.722907035432133
CV iteration : depth=100, min_samples=5, train_score=0.9903078838586471,
test_score=0.7166756241149822
CV : depth=100, min_samples_split=5, train_score=0.9917877758269437, test_score=0.7207635837259242
CV iteration : depth=100, min_samples=10, train_score=0.9879643504637643,
test_score=0.7259001478398144
CV iteration : depth=100, min_samples=10, train_score=0.98943339214379,
test_score=0.7294425062305283
CV iteration : depth=100, min_samples=10, train_score=0.9858131767387907,
test_score=0.7279738654783275
CV : depth=100, min_samples_split=10, train_score=0.9877369731154483,
test_score=0.7277721731828901
CV iteration : depth=100, min_samples=100, train_score=0.9618268671665592,
test_score=0.7663875253563703
CV iteration : depth=100, min_samples=100, train_score=0.963419756674991,
test_score=0.7770286076699996
CV iteration : depth=100, min_samples=100, train_score=0.9603542233863164,
test_score=0.7651896689616182
CV : depth=100, min_samples_split=100, train_score=0.9618669490759556,
test_score=0.7695352673293293
CV iteration : depth=100, min_samples=500, train_score=0.9297718815397488,
test_score=0.7981216405602165
CV iteration : depth=100, min_samples=500, train_score=0.9382658347504623,
test_score=0.7933482964224842
CV iteration : depth=100, min_samples=500, train_score=0.9278560433432181,
test_score=0.7909588836512633
CV : depth=100, min_samples_split=500, train_score=0.9319645865444764,
test_score=0.7941429402113215
CV iteration : depth=500, min_samples=5, train_score=0.9998731775447107,
test_score=0.6976060223148425
CV iteration : depth=500, min_samples=5, train_score=0.9998957953090298,
test_score=0.6997553060129426
CV iteration : depth=500, min_samples=5, train_score=0.9998561662633918,
test_score=0.6934868825390507
```

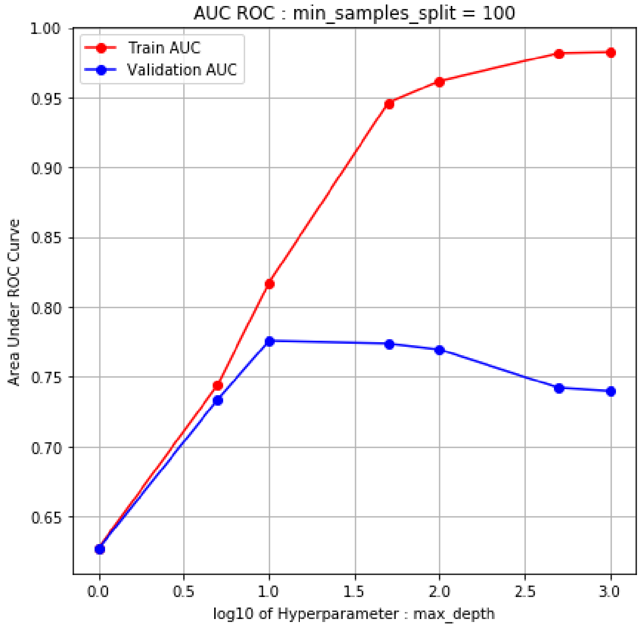
```
CV : depth=500, min_samples_split=5, train_score=0.9998750463723775, test_score=0.6969494036222786
CV iteration : depth=500, min_samples=10, train_score=0.9989137709373265,
test_score=0.7054013409903822
CV iteration : depth=500, min_samples=10, train_score=0.9989637830939884,
test_score=0.7094466317134613
CV iteration : depth=500, min_samples=10, train_score=0.9986862210954334,
test_score=0.6976448248524946
CV : depth=500, min_samples_split=10, train_score=0.9988545917089161,
test_score=0.7041642658521128
CV iteration : depth=500, min_samples=100, train_score=0.9834412017252342,
test_score=0.7408045235754321
CV iteration : depth=500, min_samples=100, train_score=0.9807308308010101,
test_score=0.7531610303024459
CV iteration : depth=500, min_samples=100, train_score=0.9815042386705277,
test_score=0.7324018564818171
CV : depth=500, min_samples_split=100, train_score=0.9818920903989241,
test_score=0.7421224701198984
CV iteration : depth=500, min_samples=500, train_score=0.9524481137290266,
test_score=0.7635454180591231
CV iteration : depth=500, min_samples=500, train_score=0.9569957396497195,
test_score=0.7629411591603756
CV iteration : depth=500, min_samples=500, train_score=0.9511601843276231,
test_score=0.7518829813276431
CV : depth=500, min_samples_split=500, train_score=0.9535346792354563,
test_score=0.7594565195157138
CV iteration : depth=1000, min_samples=5, train_score=0.9998808912579277,
test_score=0.6943815889425754
CV iteration : depth=1000, min_samples=5, train_score=0.9998911555977591,
test_score=0.6961910095771755
CV iteration : depth=1000, min_samples=5, train_score=0.9998472550365565,
test_score=0.6901680560547787
CV : depth=1000, min_samples_split=5, train_score=0.9998731006307477,
test_score=0.6935802181915097
CV iteration : depth=1000, min_samples=10, train_score=0.998905290931958,
test_score=0.7051700023623142
CV iteration : depth=1000, min_samples=10, train_score=0.9989548923383598,
test_score=0.7079821258983072
CV iteration : depth=1000, min_samples=10, train_score=0.9987030124433227,
test_score=0.6992234779338079
CV : depth=1000, min_samples_split=10, train_score=0.9988543985712135,
test_score=0.704125202064809 8
CV iteration : depth=1000, min_samples=100, train_score=0.9836327681381782,
test_score=0.7383540598401335
CV iteration : depth=1000, min_samples=100, train_score=0.981275322229052,
test_score=0.7487698356049101
CV iteration : depth=1000, min_samples=100, train_score=0.9831522172845724,
test_score=0.7320029502329048
CV : depth=1000, min_samples_split=100, train_score=0.9826867692172675,
test_score=0.7397089485593161
CV iteration : depth=1000, min_samples=500, train_score=0.9506316214957024,
test_score=0.770388392421391
CV iteration : depth=1000, min_samples=500, train_score=0.9569622811702041,
test_score=0.7649871556455545
CV iteration : depth=1000, min_samples=500, train_score=0.9504895681416345,
test_score=0.7529746968459168
CV : depth=1000, min_samples_split=500, train_score=0.9526944902691804,
test_score=0.7627834149709541
```

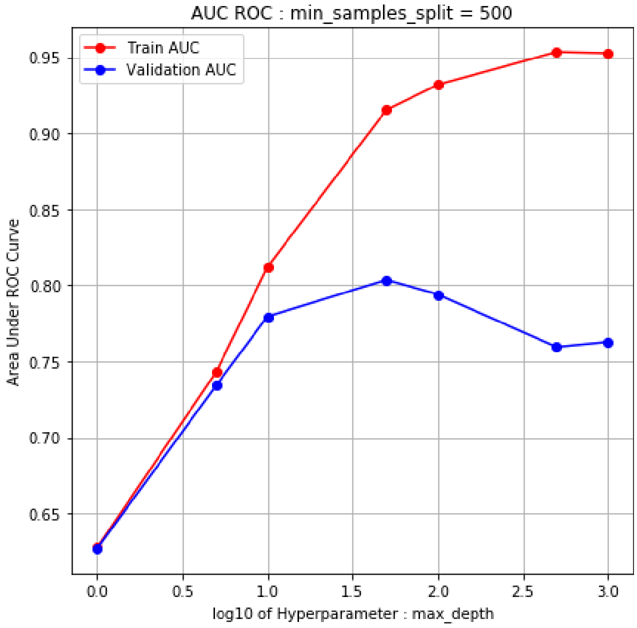




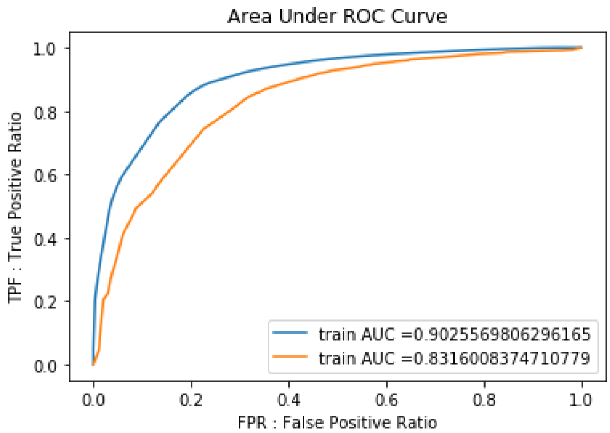


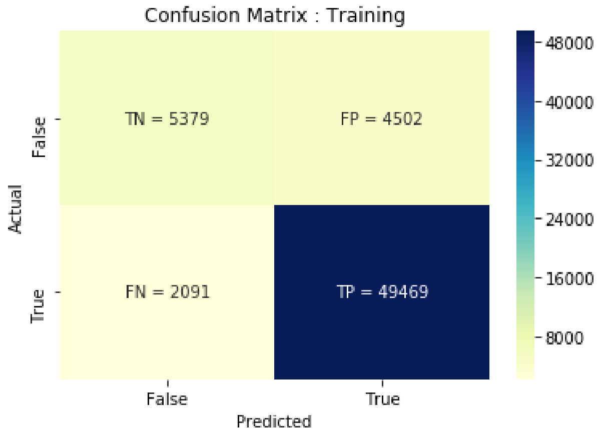


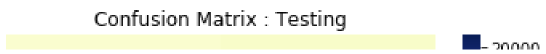


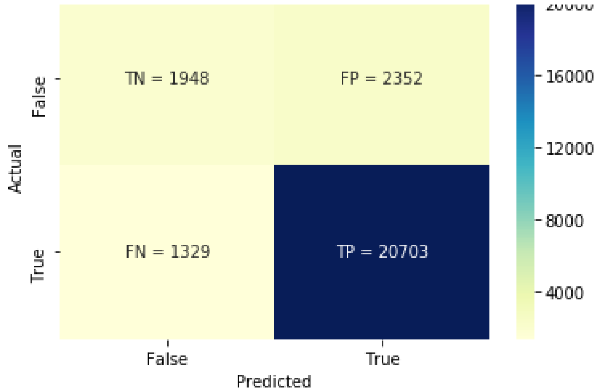



```
{'max_depth': 50, 'min_samples_split': 500}
Initializing Vectorizer
Training Model....
Saving Trained Model....
Retraining Vectorizer with Dx_train
Area Under the Curve for Train :  0.9025569806296165
Area Under the Curve for Test :  0.8316008374710779
```









[5.2.1] Top 20 important features from SET 2

In [52]:

```
clf1 = DecisionTreeClassifier(random_state=0)
vectorizer_obj1 = get_vectorizer(W2V_model = None, train=Dx_train, vectorizer='TFIDF')
clf1.fit(vectorizer_obj1.transform(Dx_train), np.array(Dy_train))
```

Out[52]:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=0,
            splitter='best')
```

In [53]:

```
feature_importance = clf1.feature_importances_
features = vectorizer_obj1.get_feature_names()
features_with_names = [(features[i], feature_importance[i]) for i in range(feature_importance.shape
[0]) if feature_importance[i]>0]
features_with_names.sort(key=lambda x: x[1], reverse=True)
features_with_names[:20]
```

Out[53]:

```
[('not', 0.05285774017768727 5),
 ('great', 0.02464312151077595),
 ('disappointed', 0.018193155360967064),
 ('worst', 0.014167732198852845),
 ('money', 0.013095772156989174),
 ('not buy', 0.012989791164653671),
 ('awful', 0.01196214528199168),
 ('return', 0.011809157283450666),
 ('good', 0.011741300079850681),
 ('horrible', 0.011704491465894641),
 ('love', 0.010272241989765755),
 ('best', 0.0100960915665325),
 ('delicious', 0.00884606501470341),
 ('bad', 0.008365333667113348),
 ('not worth', 0.008049574196986922),
 ('waste money', 0.008007015390045853),
 ('like', 0.007860762110616981),
 ('not recommend', 0.007835141539032498),
 ('not disappointed', 0.007081061149136909),
 ('taste', 0.006445505143929829)]
```

In [54]:

```
del clf1
del vectorizer_obj1
del feature_importance
del features
del features_with_names
```

[5.2.2] Graphviz visualization of Decision Tree on TFIDF, SET 2

In [55]:

```
from sklearn import tree
from graphviz import Source
```

In [56]:

```
dot_data = tree.export_graphviz(clf, feature_names = vectorizer_obj.get_feature_names(),filled=True
)
graph = Source(dot_data)
graph
```

Out[56]:

Preparing/Training Google Word2Vec

In [47]:

```
is_your_ram_gt_16g=True
want_to_use_google_w2v = True
want_to_train_w2v = False

path_to_word2vec = '/home/monodeepdas112/Datasets/google_w2v_for_amazon.pkl'

if want_to_train_w2v:

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

    # min_count = 5 considers only words that occured atleast 5 times
    w2v_model=Word2Vec(list_of_sentences,min_count=5,size=300, workers=4)
    print(w2v_model.wv.most_similar('great'))
    print('='*50)
    print(w2v_model.wv.most_similar('worst'))

elif want_to_use_google_w2v and is_your_ram_gt_16g:
    if os.path.isfile(path_to_word2vec):
        print('Preparing to load pre-trained Word2Vec model !')
        w2v_model=KeyedVectors.load_word2vec_format(path_to_word2vec, binary=True)
        print('Successfully loaded model into memory !!')
        print('Words similar to "similar" : ', w2v_model.wv.most_similar('great'))
        print('Words similar to "worst" : ',w2v_model.wv.most_similar('worst'))
    else:
        print("you don't have google's word2vec file, keep want_to_train_w2v = True, to train your
own w2v ")
```

```
Preparing to load pre-trained Word2Vec model !


---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-47-ea129842bf5a> in <module>()
     22     if os.path.isfile(path_to_word2vec):
     23         print('Preparing to load pre-trained Word2Vec model !')
---> 24         w2v_model=KeyedVectors.load_word2vec_format(path_to_word2vec, binary=True, unicode_
errors='ignore')
     25         print('Successfully loaded model into memory !!')
     26         print('Words similar to "similar" : ', w2v_model.wv.most_similar('great'))

~/anaconda3/envs/AppliedAI/lib/python3.7/site-packages/gensim/models/keyedvectors.py in
load_word2vec_format(cls, fname, fvocab, binary, encoding, unicode_errors, limit, datatype)
   1117         return _load_word2vec_format(
   1118             Word2VecKeyedVectors, fname, fvocab=fvocab, binary=binary, encoding=encoding, u
```

```
nicode_errors=unicode_errors,
-> 1119                limit=limit, datatype=datatype)
   1120
   1121     def get_keras_embedding(self, train_embeddings=False):

~/anaconda3/envs/AppliedAI/lib/python3.7/site-packages/gensim/models/utils_any2vec.py in
_load_word2vec_format(cls, fname, fvocab, binary, encoding, unicode_errors, limit, datatype)
    172     logger.info("loading projection weights from %s", fname)
    173     with utils.smart_open(fname) as fin:
--> 174         header = utils.to_unicode(fin.readline(), encoding=encoding)
    175         vocab_size, vector_size = (int(x) for x in header.split())  # throws for invalid
file format
    176         if limit:

~/anaconda3/envs/AppliedAI/lib/python3.7/site-packages/gensim/utils.py in any2unicode(text,
encoding, errors)
    357     if isinstance(text, unicode):
    358         return text
--> 359     return unicode(text, encoding, errors=errors)
    360
    361

UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0: invalid start byte
```

[5.3] Applying Decision Trees on AVG W2V, SET 3

In [58]:

```
# Please write all the code with proper documentation
csv_path = 'saved_models/Assignment8/Avg-W2Vec_dtree_results.csv'
cv_results = perform_hyperparameter_tuning(X=Dx_train, Y=Dy_train, vectorizer='Avg-W2Vec',
                                            results_path=csv_path, retrain=True, W2V_model=w2v_model
# Analysing best parameters
best_parameters = analyse_results(cv_results)
pprint.pprint(best_parameters)

# retraining the model with best parameters
model_path = 'saved_models/Assignment8/{0}_dtree.pkl'.format('Avg-W2Vec')
clf = retrain_with_best_params(Dx_train, Dy_train, best_parameters, 'Avg-W2Vec', model_path, w2v_mo
del)

print('Retraining Vectorizer with Dx_train')
vectorizer_obj = get_vectorizer(W2V_model = w2v_model, train=Dx_train, vectorizer='Avg-W2Vec')

# plotting AUC ROC
train_score, test_score = plot_AUC_ROC(clf, vectorizer_obj, Dx_train, Dx_test, Dy_train, Dy_test)

# appending the data results
prettytable_data.append(['Avg-W2Vec', 'Decision Trees', best_parameters['max_depth'],
best_parameters['min_samples_split'], train_score, test_score])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-58-b2a83dc029dd> in <module>()
      2 csv_path = 'saved_models/Assignment8/Avg-W2Vec_dtree_results.csv'
      3 cv_results = perform_hyperparameter_tuning(X=Dx_train, Y=Dy_train, vectorizer='Avg-W2Vec',
----> 4                                             results_path=csv_path, retrain=True, W2V_model=w
_model)
      5 # Analysing best parameters
      6 best_parameters = analyse_results(cv_results)

NameError: name 'w2v_model' is not defined
```

[5.4] Applying Decision Trees on TFIDF W2V, SET 4

In [ ]:

```
# Please write all the code with proper documentation
csv_path = 'saved_models/Assignment8/TFIDF-W2Vec_dtree_results.csv'
cv_results = perform_hyperparameter_tuning(X=Dx_train, Y=Dy_train, vectorizer='TFIDF-W2Vec',
                                            results_path=csv_path, retrain=True, W2V_model=w2v_model
```

```
# Analysing best parameters
best_parameters = analyse_results(cv_results)
pprint.pprint(best_parameters)

# retraining the model with best parameters
model_path = 'saved_models/Assignment8/{0}_dtree.pkl'.format('TFIDF-W2Vec')
clf = retrain_with_best_params(Dx_train, Dy_train, best_parameters, 'TFIDF-W2Vec', model_path, w2v_
model)

print('Retraining Vectorizer with Dx_train')
vectorizer_obj = get_vectorizer(W2V_model = w2v_model, train=Dx_train, vectorizer='TFIDF-W2Vec')

# plotting AUC ROC
train_score, test_score = plot_AUC_ROC(clf, vectorizer_obj, Dx_train, Dx_test, Dy_train, Dy_test)

# appending the data results
prettytable_data.append(['TFIDF-W2Vec', 'Decision Trees', best_parameters['max_depth'],
best_parameters['min_samples_split'], train_score, test_score])
```

[6] Conclusions

In [ ]:

```
from prettytable import PrettyTable
```

In [ ]:

```
# Please compare all your models using Prettytable library
x = PrettyTable()

x.field_names = ["Vectorizer", "Model", "max_depth", "min_samples_split", "Train AUC", "Test AUC"]
[x.add_row(i) for i in prettytable_data]
print(x)
```

Save to PDF