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

# 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 50
0000 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 Sco
re != 3 LIMIT 500000""", con)
# for tsne assignment you can take 5k data points
```

```
filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score
 != 3 LIMIT 500000""", con)

# Give reviews with Score>3 a positive rating(1), and reviews with a sc
ore<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 (500000, 10)
```

Out[2]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenomin
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	

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(*)

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

Out[7]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenon
0	78445	B000HDL1RQ	AR5J8UI46CURR	Geetha Krishnan	2	

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenon
1	138317	B000HDOPYC	AR5J8UI46CURR	Geetha Krishnan	2	
2	138277	B000HDOPYM	AR5J8UI46CURR	Geetha Krishnan	2	
3	73791	B000HDOPZG	AR5J8UI46CURR	Geetha Krishnan	2	
4	155049	B000PAQ75C	AR5J8UI46CURR	Geetha Krishnan	2	

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

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

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

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

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

In [8]:
```
#Sorting data according to ProductId in ascending order
sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=Tr
ue, 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]:
```
(348262, 10)
```

In [10]:
```
#Checking to see how much % of data still remains
(final['Id'].size*1.0)/(filtered_data['Id'].size*1.0)*100
```

Out[10]:
```
69.6524
```

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	HelpfulnessDenom
0	64422	B000MIDROQ	A161DK06JJMCYF	J. E. Stephens "Jeanne"	3	
1	44737	B001EQ55RW	A2V0I904FH7ABY	Ram	3	

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()
```

```
(348260, 10)
```

Out[13]:
```
1    293516
0     54744
Name: Score, dtype: int64
```

[3] Preprocessing

[3.1]. Preprocessing Review Text

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

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

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

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

In [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)
```

```
This book was purchased as a birthday gift for a 4 year old boy. He squ
ealed with delight and hugged it when told it was his to keep and he di
d not have to return it to the library.
==================================================
I've purchased both the Espressione Espresso (classic) and the 100% Ara
```

```
bica.  My vote is definitely with the 100% Arabica.  The flavor has mor
e bite and flavor (much more like European coffee than American).
==================================================
This is a great product. It is very healthy for all of our dogs, and it
is the first food that they all love to eat. It helped my older dog los
e weight and my 10 year old lab gain the weight he needed to be health
y.
==================================================
I find everything I need at Amazon so I always look there first. Chocol
ate tennis balls for a tennis party, perfect! They were the size of mal
ted milk balls. Unfortunately, they arrived 3 days after the party. The
caveat here is, not everything from Amazon may arrive at an impressive
2 or 3 days. This shipment took 8 days from the Candy/Cosmetic Depot ba
ck east to southern California.
==================================================
```

In [15]:

```
# remove urls from text python: https://stackoverflow.com/a/40823105/40
84039
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)
```

```
This book was purchased as a birthday gift for a 4 year old boy. He squ
ealed with delight and hugged it when told it was his to keep and he di
d not have to return it to the library.
```

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)
```

```
This book was purchased as a birthday gift for a 4 year old boy. He squ
ealed with delight and hugged it when told it was his to keep and he di
d not have to return it to the library.
==================================================
I've purchased both the Espressione Espresso (classic) and the 100% Ara
bica.  My vote is definitely with the 100% Arabica.  The flavor has mor
e bite and flavor (much more like European coffee than American).
==================================================
This is a great product. It is very healthy for all of our dogs, and it
is the first food that they all love to eat. It helped my older dog los
e weight and my 10 year old lab gain the weight he needed to be health
y.
==================================================
I find everything I need at Amazon so I always look there first. Chocol
ate tennis balls for a tennis party, perfect! They were the size of mal
ted milk balls. Unfortunately, they arrived 3 days after the party. The
caveat here is, not everything from Amazon may arrive at an impressive
2 or 3 days. This shipment took 8 days from the Candy/Cosmetic Depot ba
ck east to southern California.
```

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)
```

```
This is a great product. It is very healthy for all of our dogs, and it
is the first food that they all love to eat. It helped my older dog los
e weight and my 10 year old lab gain the weight he needed to be health
y.
==================================================
```

In [19]:

```
#remove words with numbers python: https://stackoverflow.com/a/1808237
0/4084039
sent_0 = re.sub("\S*\d\S*", "", sent_0).strip()
print(sent_0)
```

```
This book was purchased as a birthday gift for a  year old boy. He sque
aled with delight and hugged it when told it was his to keep and he did
not have to return it to the library.
```

In [20]:

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

```
This is a great product It is very healthy for all of our dogs and it i
s the first food that they all love to eat It helped my older dog lose
weight and my 10 year old lab gain the weight he needed to be healthy
```

In [21]:

```
# https://gist.github.com/sebleier/554280
# we are removing the words from the stop words list: 'no', 'nor', 'no
t'
# <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', 'o
urs', 'ourselves', 'you', "you're", "you've",\
            "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselve
s', 'he', 'him', 'his', 'himself', \
            'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'it
s', 'itself', 'they', 'them', 'their',\
            'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'th
is', 'that', "that'll", 'these', 'those', \
            'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'h
ave', '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', 'h
ow', 'all', 'any', 'both', 'each', 'few', 'more',\
            'most', 'other', 'some', 'such', 'only', 'own', 'same', 's
o', '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', "is
n'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]:
```python
# 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())
preprocessed_reviews[1500]
```

```
100%|██████████████████████████████████████████████████
██████| 348260/348260 [01:56<00:00, 3001.64it/s]
```

Out[22]:
```
'great product healthy dogs first food love eat helped older dog lose w
eight year old lab gain weight needed healthy'
```

In [23]:
```python
# Adding new column into dataframe to store cleaned text
final.loc[:,'CleanedText'] = preprocessed_reviews
final.head(3)
print(final['Score'].value_counts())
final.head(3)
```

```
1    293516
0     54744
Name: Score, dtype: int64
```

Out[23]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessD
138702	150520	0006641040	ADBFSA9KTQANE	James L. Hammock "Pucks Buddy"	0	

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessD
138707	150525	0006641040	A2QID6VCFTY51R	Rick	1	
138708	150526	0006641040	A3E9QZFE9KXH8J	R. Mitchell	11	

[3.2] Preprocessing Review Summary

In [ ]:
```
## Similartly you can do preprocessing for review summary also.
```

[4] Featurization

[4.1] BAG OF WORDS

In [ ]:
```
#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 [ ]:
```
#bi-gram, tri-gram and n-gram

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

# you can choose these numebrs min_df=10, max_features=5000, of your ch
oice
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_s
hape())
print("the number of unique words including both unigrams and bigrams "
, final_bigram_counts.get_shape()[1])
```

[4.3] TF-IDF

In [ ]:
```
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.ge
t_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 [ ]:
```
i=0
list_of_sentance_train=[]
for sentance in X_train:
    list_of_sentance_train.append(sentance.split())

w2v_model=Word2Vec(list_of_sentance_train,min_count=5,size=50, workers=
4)

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])
```

In [ ]:
```
# 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 val
ues
# To use this code-snippet, download "GoogleNews-vectors-negative300.bi
n"
# from https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edi
t
# it's 1.9GB in size.


# http://kavita-ganesan.com/gensim-word2vec-tutorial-starter-code/#.W17
SRFAzZPY
# 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_trai
n_w2v = True, to train your own w2v ")
```

In [ ]:

```
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 [ ]:

```
# 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, yo
u might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/re
view
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors.append(sent_vec)
print(len(sent_vectors))
print(len(sent_vectors[0]))
```

[4.4.1.2] TFIDF weighted W2v

In [ ]:
```
# 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 v
alue
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

In [ ]:
```
# 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 ce
ll_val = tfidf

tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is st
ored 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/r
eview
    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 3: KNN

1. **Apply Knn(brute force version) 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. **Apply Knn(kd tree version) on these feature sets**
    NOTE: sklearn implementation of kd-tree accepts only dense matrices, you need to convert the sparse matrices of CountVectorizer/TfidfVectorizer into dense matices. You can convert sparse matrices to dense using .toarray() attribute. For more information please visit this link

    - SET 5:Review text, preprocessed one converted into vectors using (BOW) but with restriction on maximum features generated.

    ```
                count_vect = CountVectorizer(min_df=10, max_fe
    atures=500)
                count_vect.fit(preprocessed_reviews)
    ```

- SET 6:Review text, preprocessed one converted into vectors using (TFIDF) but with restriction on maximum features generated.

```
tf_idf_vect = TfidfVectorizer(min_df=10, max_features=500)
tf_idf_vect.fit(preprocessed_reviews)
```

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

3. **The hyper paramter tuning(find best K)**

- 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

4. **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

5. **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.

[5.1] Applying KNN brute force

[5.1.1] Applying KNN brute force on BOW, SET 1

In [24]:
```
#RANDOM Sampling

final['Score'].value_counts()

count_class_1, count_class_0 = final['Score'].value_counts()
print(count_class_0)
print(count_class_1)


df_class_0 = final[final['Score'] == 0].sample(
    n=25000, random_state=42)
df_class_1 = final[final['Score'] == 1].sample(
    n=25000, random_state=42)
final_bow = pd.concat([df_class_1, df_class_0], axis=0)
print('After Under Sampling')
print(final_bow.Score.value_counts())
```

```
54744
293516

After Under Sampling
```

```
1    25000
0    25000
Name: Score, dtype: int64
```

In [116]:

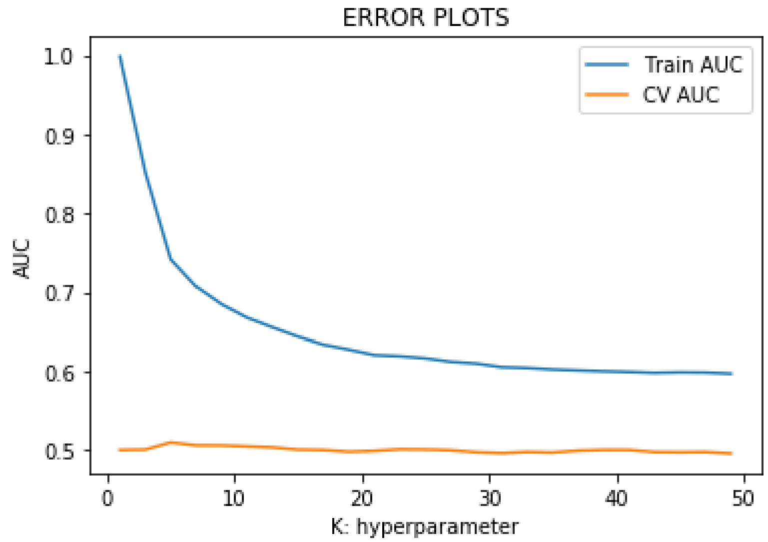
```
# Sorting based on time
final_bow['Time'] = pd.to_datetime(final['Time'])
total_points = final_bow.sort_values(by='Time' , ascending=True)

sample_points = final_bow['CleanedText']
labels = total_points['Score']

final.head(2)

# Splitting the Data into train and test

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(sample_points,label
s, test_size=0.30,shuffle=False)# this is for time series split
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_
size=0.30,shuffle=False)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
print(X_cv.shape,y_cv.shape)


vectorizer = CountVectorizer()

X_train_bow= vectorizer.fit_transform(X_train)
X_cv_bow=vectorizer.transform(X_cv)
X_test_bow = vectorizer.transform(X_test)

print(X_train_bow.shape, y_train.shape)
print(X_cv_bow.shape, y_cv.shape)
print(X_test_bow.shape, y_test.shape)
```

```
(24500,) (24500,)

(15000,) (15000,)
```

```
(10500,) (10500,)
(24500, 30486) (24500,)
(10500, 30486) (10500,)
(15000, 30486) (15000,)
```

In [26]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt

train_auc_k_bow = []
cv_auc_k_bow = []
myList=list(range(1,50,2))
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='brute')
    neigh.fit(X_train_bow, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(X_train_bow)[:,1]
    y_cv_pred =  neigh.predict_proba(X_cv_bow)[:,1]

    train_auc_k_bow.append(roc_auc_score(y_train,y_train_pred))
    cv_auc_k_bow.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc_k_bow, label='Train AUC')
plt.plot(myList, cv_auc_k_bow, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

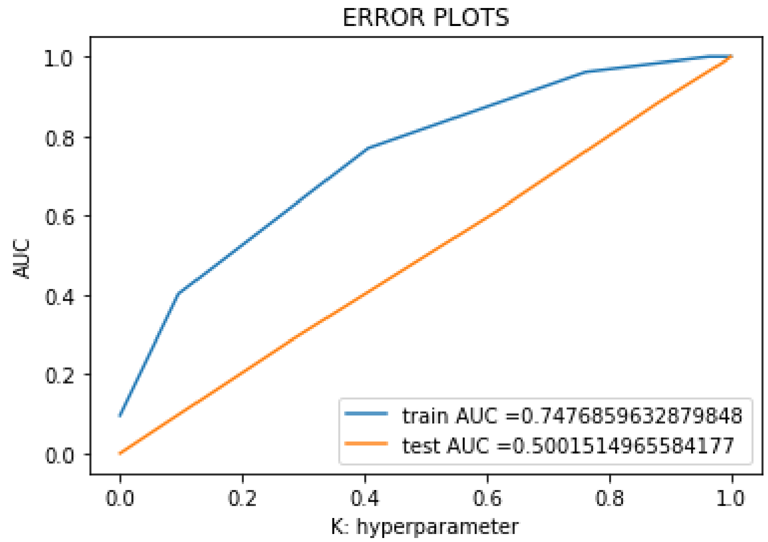


In [27]:
```
k_bow_train_auc = dict(zip(myList, train_auc_k_bow))
k_bow_cv_auc = dict(zip(myList, np.round(cv_auc_k_bow,3)))
print(k_bow_cv_auc)
print(k_bow_train_auc)

best_k_bow=5
```

```
{1: 0.5, 3: 0.5, 5: 0.509, 7: 0.506, 9: 0.506, 11: 0.504, 13: 0.503, 1
5: 0.5, 17: 0.5, 19: 0.498, 21: 0.499, 23: 0.501, 25: 0.501, 27: 0.5, 2
9: 0.497, 31: 0.496, 33: 0.497, 35: 0.497, 37: 0.499, 39: 0.5, 41: 0.5,
43: 0.497, 45: 0.497, 47: 0.497, 49: 0.496}
{1: 0.9982282734565668, 3: 0.8513379572344572, 5: 0.7415334875840621,
7: 0.7068494434888867, 9: 0.684469601434734, 11: 0.6676293703700602, 1
3: 0.6554092335226134, 15: 0.6437031962846054, 17: 0.6334626096399144,
19: 0.6271162042163572, 21: 0.6202839784241967, 23: 0.6188816879234078,
25: 0.6162571663596037, 27: 0.6118688809856244, 29: 0.6096053110233576,
31: 0.6050884815174025, 33: 0.6040666765639549, 35: 0.6021830971954297,
37: 0.6010194846912322, 39: 0.5998533773533472, 41: 0.5990306005999747,
43: 0.5978967923287865, 45: 0.5985179925398988, 47: 0.5982650946601435,
49: 0.5968281105874026}
```

Based on the above values we would say the the would be 5 let's train the model using optimal value

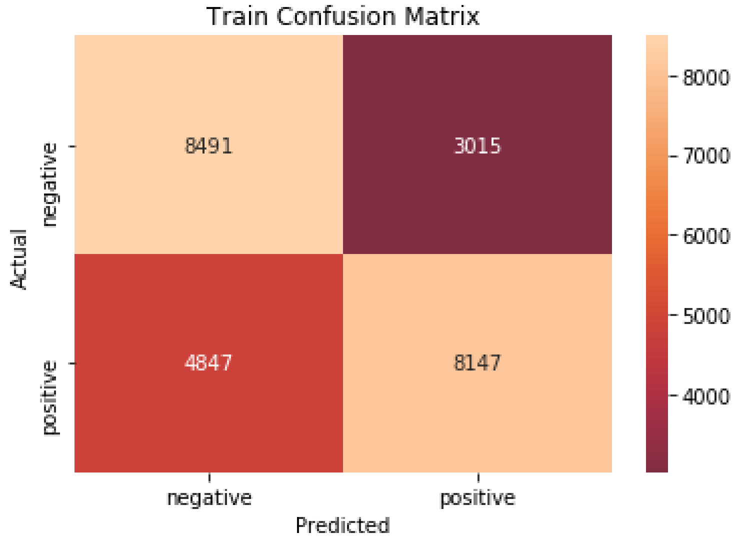
In [107]:
```
from sklearn.metrics import roc_curve, auc


neigh = KNeighborsClassifier(n_neighbors=best_k_bow,algorithm='brute')
neigh.fit(X_train_bow, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(X_train_bow)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
X_test_bow)[:,1])

train_auc_k_bow=auc(train_fpr, train_tpr)
test_auc_k_bow=auc(test_fpr, test_tpr)

plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```



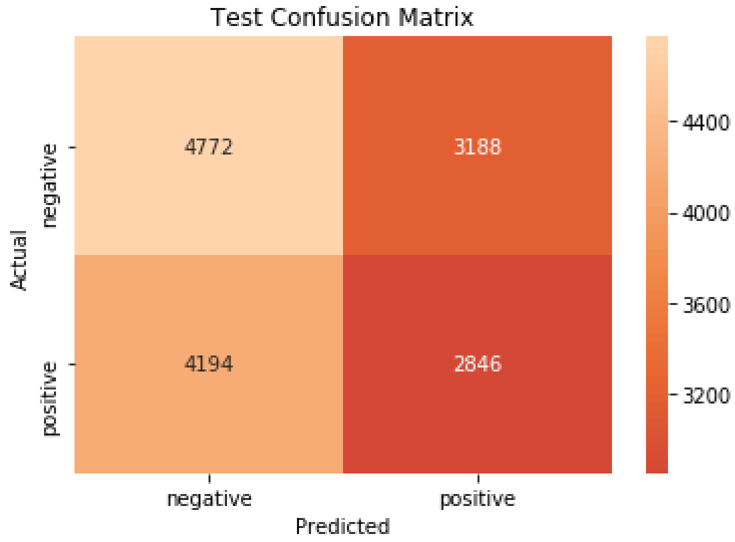
In [29]:

```
#source: https://tryolabs.com/blog/2017/03/16/pandas-seaborn-a-guide-to
-handle-visualize-data-elegantly/

from sklearn.metrics import confusion_matrix
import seaborn as sb
from sklearn.metrics import classification_report


conf_matrix = confusion_matrix(y_train,neigh.predict(X_train_bow))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d',center=0)
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(X_test_bow))
class_label = ['negative', 'positive']
```

```
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d',center=0)
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(X_test_bow)))
print("_" * 101)
```




```
====================================================================================================
```




```
______________________________________________________________________
______________________________
Classification Report on Test:

             precision    recall  f1-score   support

          0       0.53      0.60      0.56      7960
          1       0.47      0.40      0.44      7040

avg / total       0.50      0.51      0.50     15000

______________________________________________________________________
______________________________
```

Conclusion: The model is avg model hence it only captures 50% accuracy

[5.1.2] Applying KNN brute force on TFIDF, SET 2

In [30]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
X_train, X_test, y_train, y_test = train_test_split(sample_points,label
```

```
s, test_size=0.30,shuffle=False)# this is for time series split
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_
size=0.30,shuffle=False)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
print(X_cv.shape,y_cv.shape)


tf_idf_vectorizer = TfidfVectorizer()

X_train_tfidf= tf_idf_vectorizer.fit_transform(X_train)
X_cv_tfidf=tf_idf_vectorizer.transform(X_cv)
X_test_tfidf = tf_idf_vectorizer.transform(X_test)

print(X_train_tfidf.shape, y_train.shape)
print(X_cv_tfidf, y_cv.shape)
print(X_test_tfidf, y_test.shape)
```

```
(24500,) (24500,)
(15000,) (15000,)
(10500,) (10500,)
(24500, 30486) (24500,)
  (0, 30197)	0.1380477562423408
  (0, 28178)	0.25250307646764514
  (0, 27329)	0.11207574514386134
  (0, 26970)	0.17690816578528384
  (0, 26713)	0.12403504389301619
  (0, 25416)	0.20520333928959808
  (0, 23689)	0.3422393384326451
  (0, 23688)	0.21686718378144135
  (0, 23511)	0.2883634537083655
  (0, 18955)	0.2047650619504105
  (0, 17479)	0.20564690783106393
  (0, 16465)	0.3039744332481464
  (0, 16083)	0.13638473784642097
  (0, 12811)	0.2945013239154105
  (0, 11404)	0.22703559126825956
  (0, 11099)	0.10653703531089619
  (0, 9885)	0.11284110463959009
```

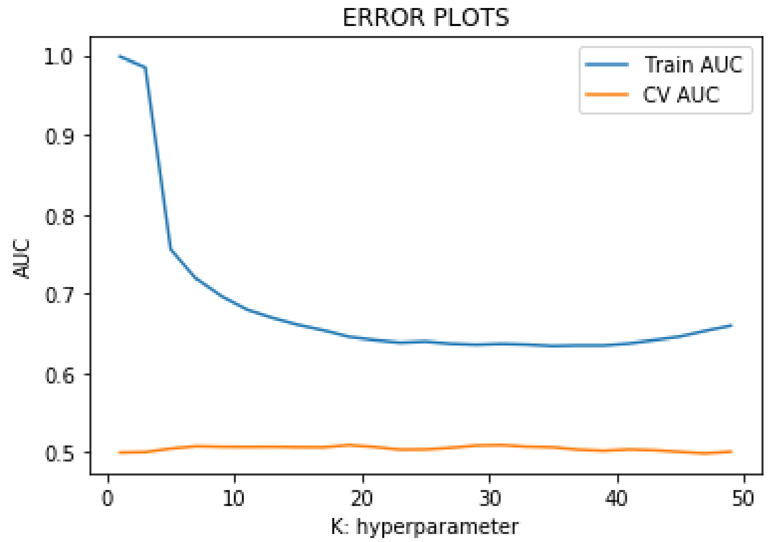
```
(0, 9562)	0.18848985556038728
(0, 5974)	0.1297122687719197
(0, 5349)	0.17479294496304823
(0, 2351)	0.10958668547960573
(0, 1427)	0.22372786709480413
(0, 1350)	0.1644969970798053
(0, 813)	0.11072580061286111
(0, 28)	0.16325624648232862
:	:
(10499, 18501)	0.1132405550238328
(10499, 18438)	0.059140268846050205
(10499, 17945)	0.07877345745603437
(10499, 17836)	0.06954618217910752
(10499, 17274)	0.07075203279003302
(10499, 16071)	0.1575248527079617
(10499, 15584)	0.11131766601690096
(10499, 15269)	0.14894950011210603
(10499, 12744)	0.1462411237086974
(10499, 10301)	0.07925467293469375
(10499, 9885)	0.07181048923023686
(10499, 9854)	0.13988277686790734
(10499, 8749)	0.11553153030377708
(10499, 7522)	0.11790790209557687
(10499, 7368)	0.09428435555102772
(10499, 6952)	0.15155017492861478
(10499, 6709)	0.21106971883714498
(10499, 3067)	0.09509001572258134
(10499, 2935)	0.3276038097618849
(10499, 1932)	0.15447673617965185
(10499, 1645)	0.10496322626496196
(10499, 1326)	0.12312543862655818
(10499, 1027)	0.09984593414551039
(10499, 813)	0.0704643395490966
(10499, 543)	0.19971758811233176 (10500,)
(0, 29352)	0.08361353700433503
(0, 28779)	0.13207101231960103
(0, 27845)	0.14225403022992267
(0, 26713)	0.1493475769725066
(0, 26653)	0.08028287105461072
```

```
(0, 26632)	0.05648809816599709
(0, 25836)	0.08507219402526359
(0, 24885)	0.1427292526259958
(0, 24745)	0.10654640884617218
(0, 24483)	0.08927052948292327
(0, 23522)	0.09190174197196671
(0, 22941)	0.32815361128429293
(0, 22438)	0.10585952377721272
(0, 22358)	0.12490885438802983
(0, 21433)	0.152016403852257
(0, 21313)	0.0924590573231849
(0, 20806)	0.09146163346246045
(0, 20795)	0.11832481305721071
(0, 20754)	0.10260942816704885
(0, 20072)	0.1562425384825512
(0, 19335)	0.1427292526259958
(0, 17945)	0.14904352976865168
(0, 16861)	0.503015154696521
(0, 15431)	0.12656985233947685
(0, 15321)	0.050150213052292045
:	:
(14999, 11389)	0.038757044314268074
(14999, 11365)	0.1099035756604461
(14999, 11201)	0.06486461010496483
(14999, 11099)	0.0492414407765521
(14999, 10065)	0.10433149133038411
(14999, 10058)	0.09183952500477063
(14999, 9891)	0.08277099183584638
(14999, 8572)	0.16995417345876596
(14999, 7697)	0.17683981219667153
(14999, 6756)	0.080465859120021
(14999, 5896)	0.14049704883090272
(14999, 5356)	0.08893805098692004
(14999, 4321)	0.20987179883467416
(14999, 4319)	0.09135713335148239
(14999, 4034)	0.10759476689544216
(14999, 3793)	0.0773115029836517
(14999, 3569)	0.054012998082465714
(14999, 2898)	0.13328165070921813
```

```
  (14999, 2000) 0.10469154880366283
  (14999, 1792) 0.07632471485896852
  (14999, 1755) 0.06898859679848597
  (14999, 1696) 0.0798649142473028
  (14999, 1132) 0.11238453141504107
  (14999, 778)  0.06379478537207278
  (14999, 766)  0.14641201911753282 (15000,)
```

In [31]:

```
train_auc_k_tfidf = []
cv_auc_k_tfidf = []
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='brute')
    neigh.fit(X_train_tfidf, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(X_train_tfidf)[:,1]
    y_cv_pred =  neigh.predict_proba(X_cv_tfidf)[:,1]

    train_auc_k_tfidf.append(roc_auc_score(y_train,y_train_pred))
    cv_auc_k_tfidf.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc_k_tfidf, label='Train AUC')
plt.plot(myList, cv_auc_k_tfidf, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

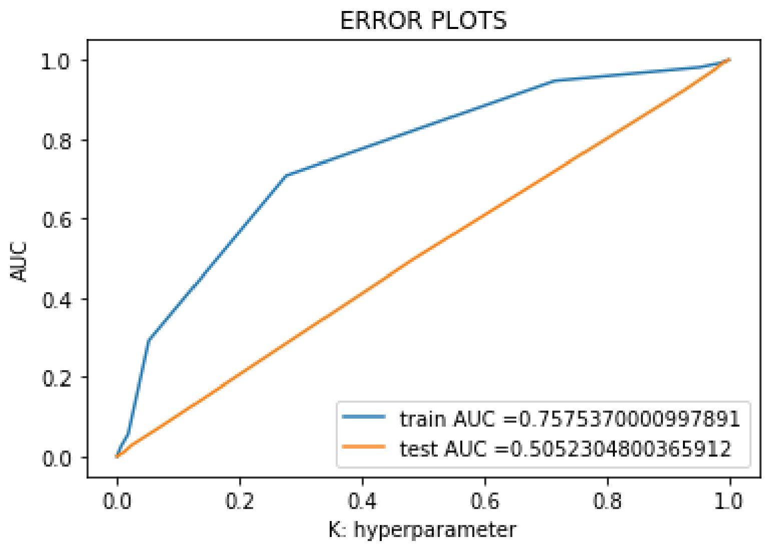


In [32]:
```
k_train_auc_tfidf = dict(zip(myList, np.round(train_auc_k_tfidf,3)))
k_cv_auc_tfidf = dict(zip(myList, np.round(cv_auc_k_tfidf,3)))
print(k_cv_auc_tfidf)
print(k_train_auc_tfidf)
best_k_tfidf=29
```

```
{1: 0.5, 3: 0.5, 5: 0.505, 7: 0.508, 9: 0.507, 11: 0.507, 13: 0.507, 1
5: 0.507, 17: 0.507, 19: 0.509, 21: 0.507, 23: 0.504, 25: 0.504, 27: 0.
506, 29: 0.509, 31: 0.509, 33: 0.507, 35: 0.507, 37: 0.504, 39: 0.502,
41: 0.504, 43: 0.503, 45: 0.501, 47: 0.499, 49: 0.501}
{1: 0.998, 3: 0.985, 5: 0.756, 7: 0.719, 9: 0.696, 11: 0.68, 13: 0.669,
15: 0.661, 17: 0.654, 19: 0.646, 21: 0.642, 23: 0.638, 25: 0.64, 27: 0.
637, 29: 0.636, 31: 0.637, 33: 0.636, 35: 0.634, 37: 0.635, 39: 0.635,
41: 0.637, 43: 0.642, 45: 0.646, 47: 0.653, 49: 0.66}
```

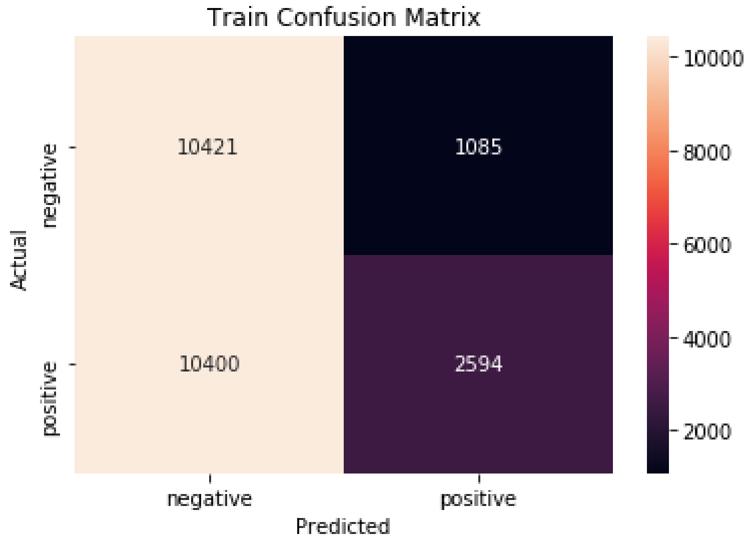
In [110]:
```
from sklearn.metrics import roc_curve, auc


neigh = KNeighborsClassifier(n_neighbors=best_k_tfidf,algorithm='brute'
)
neigh.fit(X_train_tfidf, y_train)
```

```python
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(X_train_tfidf)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
X_test_tfidf)[:,1])

train_k_auc_tfidf=auc(train_fpr, train_tpr)
test_k_auc_tfidf=auc(test_fpr, test_tpr)


plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

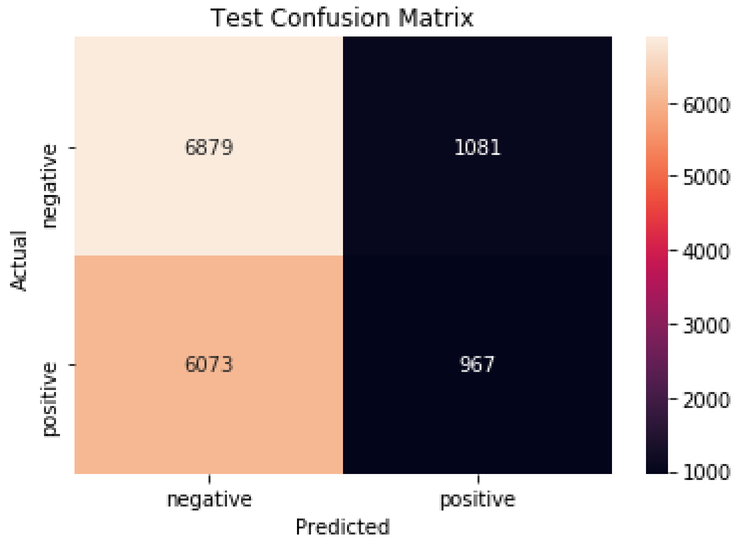


In [34]:

```python
from sklearn.metrics import confusion_matrix
import seaborn as sb
from sklearn.metrics import classification_report


conf_matrix = confusion_matrix(y_train,neigh.predict(X_train_tfidf))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(X_test_tfidf))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(X_test_tfidf)))
print("_" * 101)
```




```
==============================================================================================================
```



```
Classification Report on Test:
```

```
              precision    recall  f1-score   support

          0       0.53      0.86      0.66      7960
          1       0.47      0.14      0.21      7040

avg / total       0.50      0.52      0.45     15000

```

[5.1.3] Applying KNN brute force on AVG W2V, SET 3

In [121]:
```
i=0
list_of_sentance_train=[]
for sentance in X_train:
    list_of_sentance_train.append(sentance.split())
w2v_model=Word2Vec(list_of_sentance_train,min_count=5,size=50, workers=4)
w2v_words = list(w2v_model.wv.vocab)
```

Training W2V model on Train Data

In [122]:
```
sent_vectors_train = []; # the avg-w2v for each sentence/review is stored in this list
for sent in tqdm(list_of_sentance_train): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might need to change this 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_vectors_train.append(sent_vec)
sent_vectors_train = np.array(sent_vectors_train)
print(sent_vectors_train.shape)
print(sent_vectors_train[0])
```

```
100%|██████████████████████████████████████████████████████████████████████
█████████| 24500/24500 [00:48<00:00, 648.92it/s]
```

```
(24500, 50)
[ 0.19264201  0.997602   -0.73982878  0.34566013 -0.30177539  0.0962433
1
  0.20634034 -0.33893895  0.14981144  0.1068354  -0.53528658 -0.4589625
3
  0.03290013 -0.61915606 -0.08654531  0.47429021  0.17087841 -0.4500000
6
 -0.85042399  0.19483005 -0.94428199  0.90886559 -0.03958788 -1.1084436
4
  0.05675411 -0.13973631 -0.83668128  0.59301265 -0.67348322  0.5519673
4
 -0.96322306  0.54090708 -1.02503414 -0.45494577 -0.54934807 -0.3689269
4
  0.98864671  0.19866709  0.66509121  0.47306288 -0.04872625  0.3256414
 -1.22825715 -0.05654036  0.80779135  0.55871361  0.67620215 -0.8712255
8
  0.60101495 -1.23098091]
```

In [118]:

```
i=0
list_of_sentance_cv=[]
for sentance in X_cv:
    list_of_sentance_cv.append(sentance.split())

# average Word2Vec
# compute average word2vec for each review.
sent_vectors_cv = []; # the avg-w2v for each sentence/review is stored
 in this list
```

```
for sent in tqdm(list_of_sentance_cv): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, yo
u might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/re
view
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors_cv.append(sent_vec)
sent_vectors_cv = np.array(sent_vectors_cv)
print(sent_vectors_cv.shape)
print(sent_vectors_cv[0])
```

```
100%|██████████████████████████████████████████████████████████████████
█████████| 10500/10500 [00:17<00:00, 600.61it/s]
```

```
(10500, 50)
[-0.22272074  0.88077097 -0.5410872   0.40431074  0.00425091  0.0858636
8
  0.06575399 -0.14892975 -0.0851201   0.77006266 -0.04586707 -0.2443928
3
  0.33099695  0.14401136 -0.71115574  0.48575423  0.06450029  0.1414698
4
 -0.54609048  0.24432285 -0.66048661 -0.10162649 -0.05661974 -0.4685986
8
  0.52001499 -0.167588   -0.64922376  0.25735485  0.12726955 -0.0570445
2
 -0.44801857  0.35252933 -0.03972348  0.09060007 -0.10584066 -0.0019978
9
  0.75431736 -0.08563774 -0.11740405 -0.38904848 -0.31087601  0.5253288
4
 -1.20761307  0.09844579  0.35626311 -0.24025475  0.76640924 -0.0341810
8
  0.05425235 -0.46984297]
```

In [119]:
```python
i=0
list_of_sentance_test=[]
for sentance in X_test:
    list_of_sentance_test.append(sentance.split())

# average Word2Vec
# compute average word2vec for each review.
sent_vectors_test = []; # the avg-w2v for each sentence/review is store
d in this list
for sent in tqdm(list_of_sentance_test): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, yo
u might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/re
view
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors_test.append(sent_vec)
sent_vectors_test = np.array(sent_vectors_test)
print(sent_vectors_test.shape)
print(sent_vectors_test[0])
```

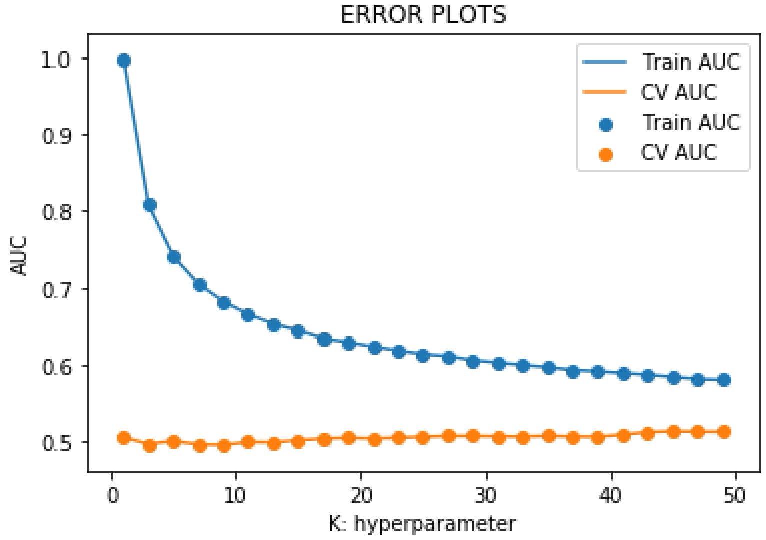
```
100%|██████████████████████████████████████████████████████████████████
█████████| 15000/15000 [00:20<00:00, 725.64it/s]
```

```
(15000, 50)
[-0.17258617  0.9468003  -0.53151082  0.61346548 -0.20389864  0.1512659
4
 -0.03972375  0.25862435 -0.67679614  0.82068429 -0.35056388 -0.1636255
8
  0.12088782  0.1877013  -0.39907618  0.30645338  0.20141172  0.0727169
8
 -0.19699983  0.31420609 -0.38835723 -0.20281383  0.26386319 -0.2424041
8
 -0.09676482  0.09529951 -0.3197415  -0.078828   -0.03124102 -0.0713425
7
```

```
 -0.5328514   0.18408195 -0.15312529 -0.17389104 -0.05850158 -0.4623257
7
  0.39776747 -0.38935703  0.02459788 -0.2698218  -0.29540478  0.3025436
4
 -0.98497744  0.54074459  0.2489831  -0.25022783  0.69866167 -0.2881594
9
  0.20974709 -0.34127963]
```

In [39]:

```
train_auc = []
cv_auc = []
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='brute')
    neigh.fit(sent_vectors_train, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(sent_vectors_train)[:,1]
    y_cv_pred =  neigh.predict_proba(sent_vectors_cv)[:,1]

    train_auc.append(roc_auc_score(y_train,y_train_pred))
    cv_auc.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc, label='Train AUC')
plt.scatter(myList, train_auc, label='Train AUC')
plt.plot(myList, cv_auc, label='CV AUC')
plt.scatter(myList, cv_auc, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

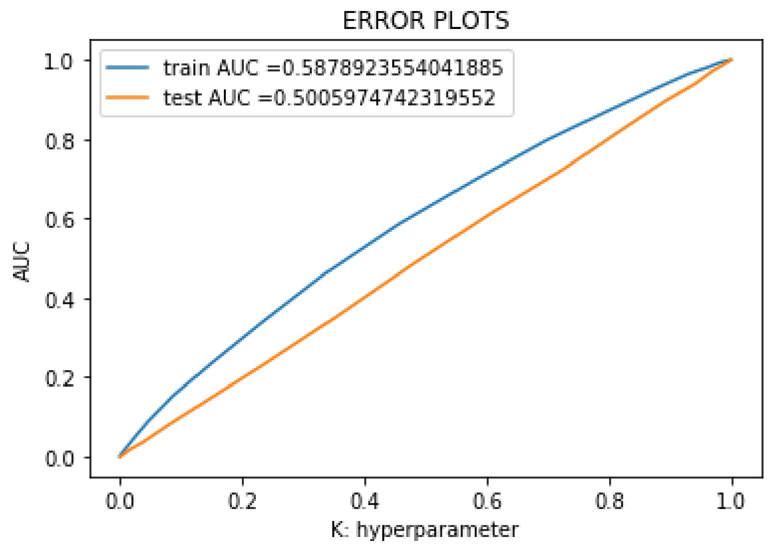


In [125]:
```
k_train_auc = dict(zip(myList, np.round(train_auc,3)))
k_cv_auc = dict(zip(myList, np.round(cv_auc,3)))
print(k_cv_auc)
print(k_train_auc)

best_k_avgw2v=49
```

```
{1: 0.504, 3: 0.502, 5: 0.499, 7: 0.507, 9: 0.505, 11: 0.506, 13: 0.50
7, 15: 0.51, 17: 0.506, 19: 0.506, 21: 0.503, 23: 0.503, 25: 0.503, 27:
0.502, 29: 0.501, 31: 0.502, 33: 0.502, 35: 0.502, 37: 0.502, 39: 0.50
3, 41: 0.503, 43: 0.502, 45: 0.503, 47: 0.503, 49: 0.505}
{1: 0.998, 3: 0.811, 5: 0.745, 7: 0.709, 9: 0.683, 11: 0.665, 13: 0.65
3, 15: 0.642, 17: 0.63, 19: 0.621, 21: 0.614, 23: 0.61, 25: 0.605, 27:
0.598, 29: 0.594, 31: 0.591, 33: 0.589, 35: 0.587, 37: 0.586, 39: 0.58
3, 41: 0.581, 43: 0.58, 45: 0.579, 47: 0.578, 49: 0.576}
```

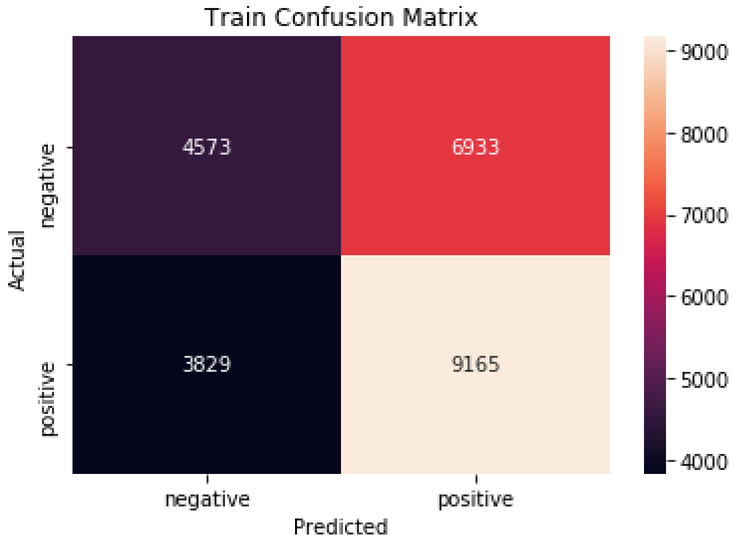
In [123]:
```
from sklearn.metrics import roc_curve, auc


neigh = KNeighborsClassifier(n_neighbors=best_k,algorithm='brute')
neigh.fit(sent_vectors_train, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
```

```
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(sent_vectors_train)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
sent_vectors_test)[:,1])

train_auc_k_avgw2v=auc(train_fpr, train_tpr)
test_auc_k_avgw2v=auc(test_fpr, test_tpr)


plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

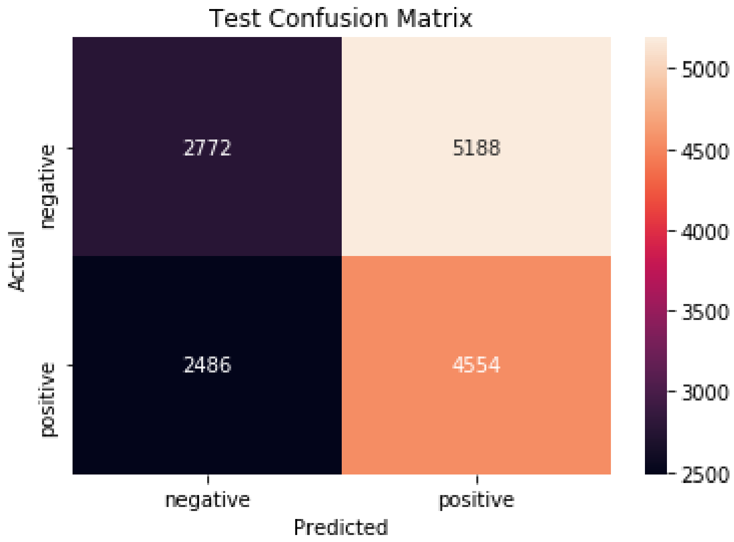


In [42]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sb
from sklearn.metrics import classification_report


conf_matrix = confusion_matrix(y_train,neigh.predict(sent_vectors_train
))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(sent_vectors_test))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(sent_vectors_test)))
print("_" * 101)
```




```
==========================================================================================================
```




```
Classification Report on Test:
```

```
classification Report on Test:

             precision    recall  f1-score   support

          0       0.53      0.35      0.42      7960
          1       0.47      0.65      0.54      7040

avg / total       0.50      0.49      0.48     15000

___________________________________________________________________________
______________________________
```

[5.1.4] Applying KNN brute force on TFIDF W2V, SET 4

TFIFD on Train Data

In [43]:
```
#Source:https://medium.com/@mohithsai504/sentiment-analysis-for-amazon-
fine-food-reviews-using-k-nn-1ae8be11908b

X_train, X_test, y_train, y_test = train_test_split(sample_points,label
s, test_size=0.30,shuffle=False)# this is for time series split
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_
size=0.30,shuffle=False)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
print(X_cv.shape,y_cv.shape)


model = TfidfVectorizer()
tf_idf_train_matrix = model.fit_transform(X_train)
# we are converting a dictionary with word as a key, and the idf as a v
alue
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

```
(24500,) (24500,)
(15000,) (15000,)
```

```
(10500,) (10500,)
```

In [127]:
```python
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and ce
ll_val = tfidf

tfidf_sent_vectors_train = []; # the tfidf-w2v for each sentence/review
 is stored in this list
row=0;
for sent in tqdm(list_of_sentance_train): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/r
eview
    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_train.append(sent_vec)
    row += 1
```

```
100%|██████████████████████████████████████████████████████████████████
██████████| 24500/24500 [04:35<00:00, 89.02it/s]
```

TFIFD on CV Data

In [128]:
```python
tf_idf_train_matrix = model.transform(X_cv)
# we are converting a dictionary with word as a key, and the idf as a v
alue
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

In [129]:
```
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and ce
ll_val = tfidf

tfidf_sent_vectors_cv = []; # the tfidf-w2v for each sentence/review is
 stored in this list
row=0;
for sent in tqdm(list_of_sentance_cv): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/r
eview
    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_cv.append(sent_vec)
    row += 1
```

```
100%|██████████████████████████████████████████████████
██████████| 10500/10500 [02:03<00:00, 84.90it/s]
```

TFIDF on test Data

In [130]:
```
tf_idf_train_matrix = model.transform(X_test)
# we are converting a dictionary with word as a key, and the idf as a v
alue
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

In [131]:
```
tfidf_feat = model.get_feature_names() # tfidf words/col-names
```

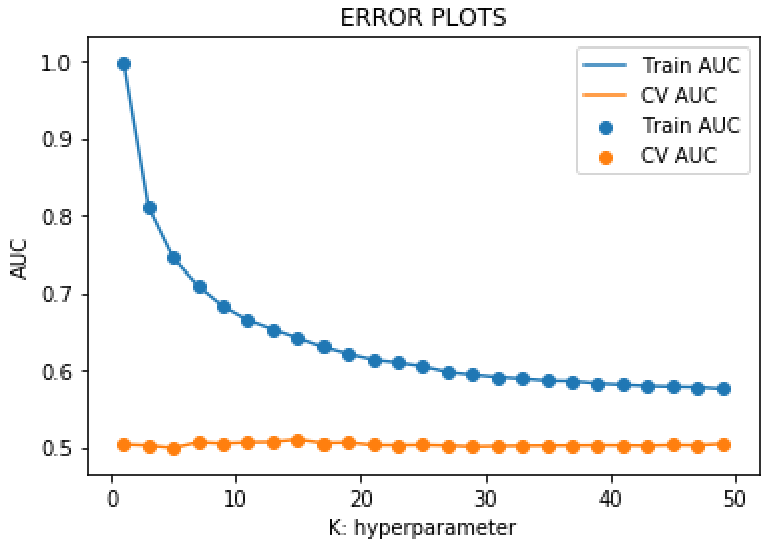
```
# final_tf_idf is the sparse matrix with row= sentence, col=word and ce
ll_val = tfidf

tfidf_sent_vectors_test = []; # the tfidf-w2v for each sentence/review
 is stored in this list
row=0;
for sent in tqdm(list_of_sentance_test): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/r
eview
    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_test.append(sent_vec)
    row += 1
```

```
100%|██████████████████████████████████████████████████████████████████
██████████| 15000/15000 [03:20<00:00, 74.84it/s]
```

In [49]:

```
train_auc = []
cv_auc = []
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='brute')
    neigh.fit(tfidf_sent_vectors_train, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(tfidf_sent_vectors_train)[:,1]
    y_cv_pred =  neigh.predict_proba(tfidf_sent_vectors_cv)[:,1]
```

```
    train_auc.append(roc_auc_score(y_train,y_train_pred))
    cv_auc.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc, label='Train AUC')
plt.scatter(myList, train_auc, label='Train AUC')
plt.plot(myList, cv_auc, label='CV AUC')
plt.scatter(myList, cv_auc, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```



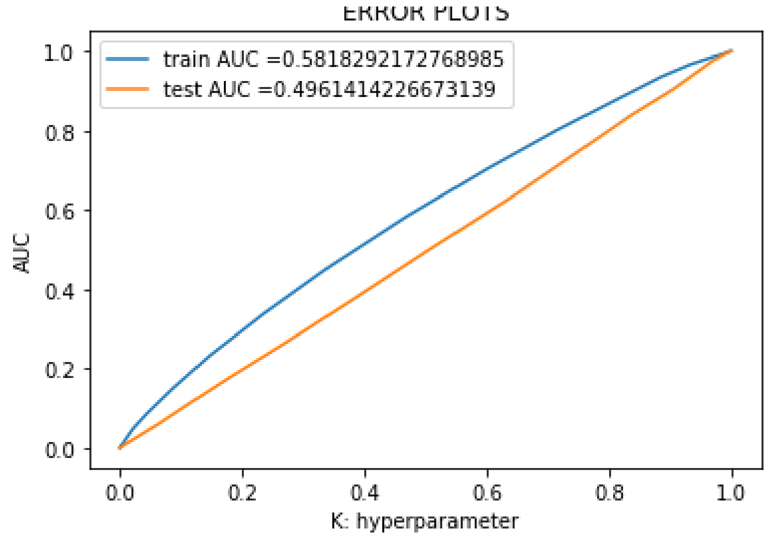
In [132]:
```
k_train_auc = dict(zip(myList, np.round(train_auc,3)))
k_cv_auc = dict(zip(myList, np.round(cv_auc,3)))
print(k_cv_auc)
print(k_train_auc)

best_k_tfidfw2v=13
```

```
{1: 0.504, 3: 0.502, 5: 0.499, 7: 0.507, 9: 0.505, 11: 0.506, 13: 0.50
7, 15: 0.51, 17: 0.506, 19: 0.506, 21: 0.503, 23: 0.503, 25: 0.503, 27:
```

```
0.502, 29: 0.501, 31: 0.502, 33: 0.502, 35: 0.502, 37: 0.502, 39: 0.50
3, 41: 0.503, 43: 0.502, 45: 0.503, 47: 0.503, 49: 0.505}
{1: 0.998, 3: 0.811, 5: 0.745, 7: 0.709, 9: 0.683, 11: 0.665, 13: 0.65
3, 15: 0.642, 17: 0.63, 19: 0.621, 21: 0.614, 23: 0.61, 25: 0.605, 27:
0.598, 29: 0.594, 31: 0.591, 33: 0.589, 35: 0.587, 37: 0.586, 39: 0.58
3, 41: 0.581, 43: 0.58, 45: 0.579, 47: 0.578, 49: 0.576}
```

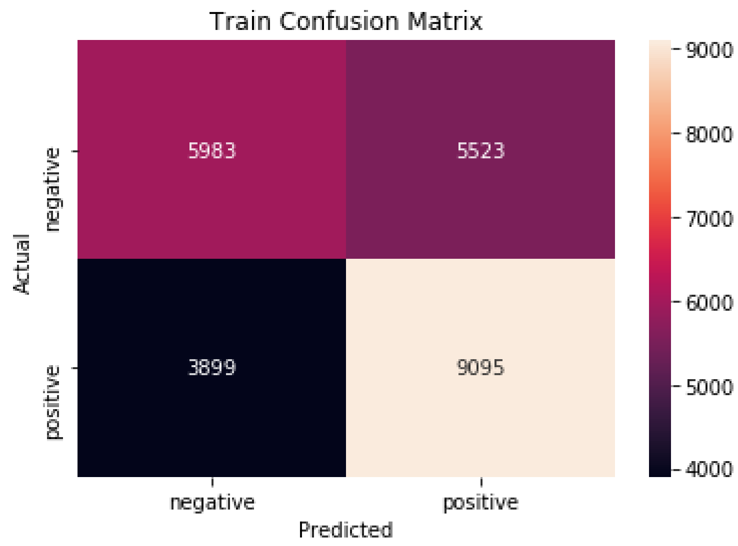
In [133]:

```
from sklearn.metrics import roc_curve, auc


neigh = KNeighborsClassifier(n_neighbors=best_k,algorithm='brute')
neigh.fit(tfidf_sent_vectors_train, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(tfidf_sent_vectors_train)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
tfidf_sent_vectors_test)[:,1])

train_auc_k_tfidfw2v=auc(train_fpr, train_tpr)
test_auc_k_tfidfw2v=auc(test_fpr, test_tpr)

plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

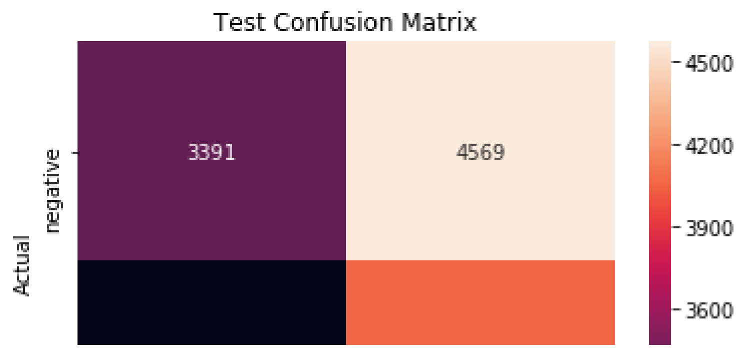


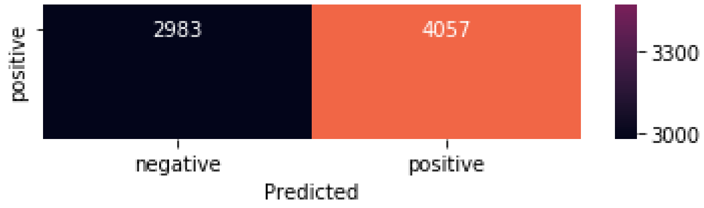
In [52]:
```
conf_matrix = confusion_matrix(y_train,neigh.predict(tfidf_sent_vectors
_train))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(tfidf_sent_vectors_
test))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
```

```
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(tfidf_sent_vectors_te
st)))
print("_" * 101)
```



==






```
______________________________________________________________________
______________________________
Classification Report on Test:
             precision    recall  f1-score   support

          0       0.53      0.43      0.47      7960
          1       0.47      0.58      0.52      7040

avg / total       0.50      0.50      0.49     15000

______________________________________________________________________
______________________________
```

[5.2] Applying KNN kd-tree

In [137]:
```
#RANDOM Sampling

final['Score'].value_counts()

count_class_1, count_class_0 = final['Score'].value_counts()
print(count_class_0)
print(count_class_1)


df_class_0 = final[final['Score'] == 0].sample(
    n=10000, random_state=42)
df_class_1 = final[final['Score'] == 1].sample(
    n=10000, random_state=42)
final_bow_kd = pd.concat([df_class_1, df_class_0], axis=0)
```

```
print('After Under Sampling')
print(final_bow_kd.Score.value_counts())
```

```
54744
293516
After Under Sampling
1    10000
0    10000
Name: Score, dtype: int64
```

[5.2.1] Applying KNN kd-tree on BOW, SET 5

In [138]:

```
# Sorting based on time
final_bow_kd['Time'] = pd.to_datetime(final['Time'])
total_points = final_bow_kd.sort_values(by='Time' , ascending=True)

sample_points = final_bow_kd['CleanedText']
labels = total_points['Score']

final.head(2)

# Splitting the Data into train and test

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(sample_points,label
s, test_size=0.30,shuffle=False)# this is for time series split
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_
size=0.30,shuffle=False)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
print(X_cv.shape,y_cv.shape)
```

```
(9800,) (9800,)
(6000,) (6000,)
(4200,) (4200,)
```

In [55]:
```
vectorizer = CountVectorizer(min_df=10, max_features=500)

X_train_bow= vectorizer.fit_transform(X_train)
X_cv_bow=vectorizer.transform(X_cv)
X_test_bow = vectorizer.transform(X_test)

print(X_train_bow.shape, y_train.shape)
print(X_cv_bow.shape, y_cv.shape)
print(X_test_bow.shape, y_test.shape)

from sklearn.preprocessing import StandardScaler

bow_train = StandardScaler().fit_transform(X_train_bow.todense())
bow_cv = StandardScaler().fit_transform(X_cv_bow.todense())
bow_test = StandardScaler().fit_transform(X_test_bow.todense())

type(bow_train)
```

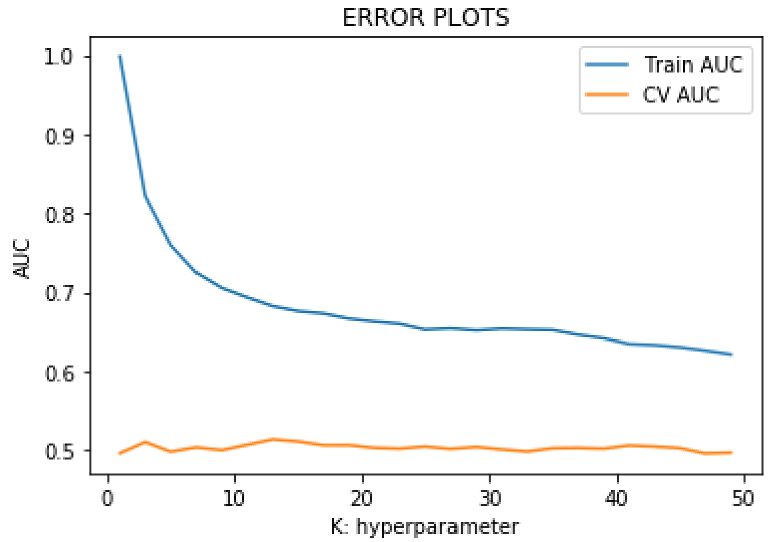
```
(9800, 500) (9800,)
(4200, 500) (4200,)
(6000, 500) (6000,)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:
475: DataConversionWarning: Data with input dtype int64 was converted t
o float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:
475: DataConversionWarning: Data with input dtype int64 was converted t
o float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:
475: DataConversionWarning: Data with input dtype int64 was converted t
o float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:
475: DataConversionWarning: Data with input dtype int64 was converted t
o float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:

475: DataConversionWarning: Data with input dtype int64 was converted t
```

```
o float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:
475: DataConversionWarning: Data with input dtype int64 was converted t
o float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
```

Out[55]: numpy.ndarray

In [56]:
```
train_auc_kd_tree = []
cv_auc_kd_tree = []
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='kd_tree')
    neigh.fit(bow_train, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(bow_train)[:,1]
    y_cv_pred =  neigh.predict_proba(bow_cv)[:,1]

    train_auc_kd_tree.append(roc_auc_score(y_train,y_train_pred))
    cv_auc_kd_tree.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc_kd_tree, label='Train AUC')
plt.plot(myList, cv_auc_kd_tree, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

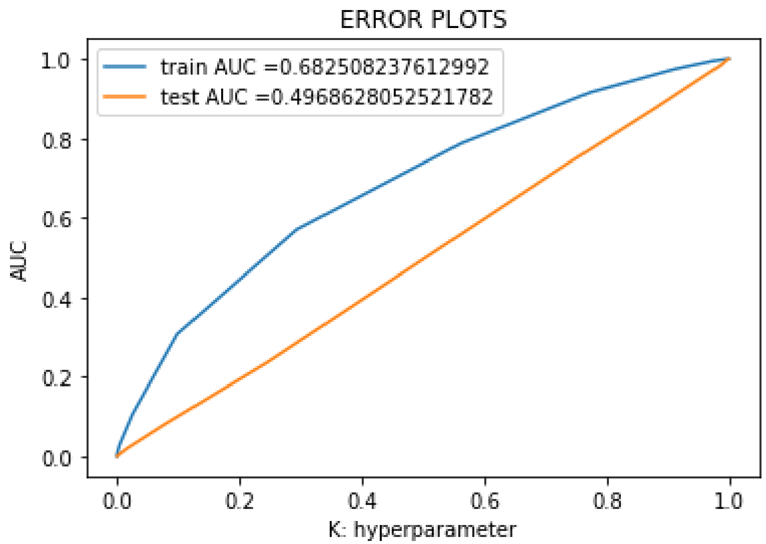


In [57]:
```
k_train_auc = dict(zip(myList, np.round(train_auc_kd_tree,3)))
k_cv_auc = dict(zip(myList, np.round(train_auc_kd_tree,3)))
print(k_cv_auc)
print(k_train_auc)

best_k_kd_bow=13
```

```
{1: 0.998, 3: 0.822, 5: 0.76, 7: 0.725, 9: 0.705, 11: 0.694, 13: 0.683,
15: 0.676, 17: 0.673, 19: 0.667, 21: 0.663, 23: 0.66, 25: 0.653, 27: 0.
654, 29: 0.652, 31: 0.654, 33: 0.653, 35: 0.653, 37: 0.646, 39: 0.642,
41: 0.635, 43: 0.633, 45: 0.631, 47: 0.626, 49: 0.622}
{1: 0.998, 3: 0.822, 5: 0.76, 7: 0.725, 9: 0.705, 11: 0.694, 13: 0.683,
15: 0.676, 17: 0.673, 19: 0.667, 21: 0.663, 23: 0.66, 25: 0.653, 27: 0.
654, 29: 0.652, 31: 0.654, 33: 0.653, 35: 0.653, 37: 0.646, 39: 0.642,
41: 0.635, 43: 0.633, 45: 0.631, 47: 0.626, 49: 0.622}
```

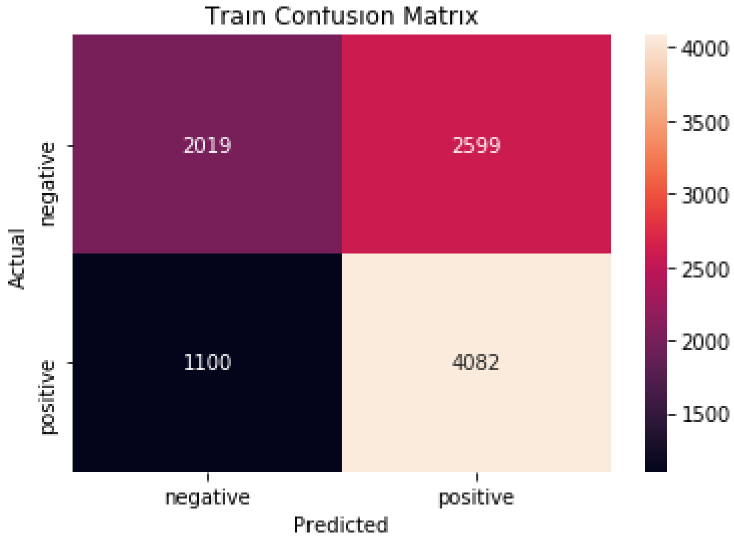
In [139]:
```
from sklearn.metrics import roc_curve, auc


neigh = KNeighborsClassifier(n_neighbors=best_k_kd_bow,algorithm='kd_tr
ee')
```

```
neigh.fit(bow_train, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(bow_train)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
bow_test)[:,1])

train_auc_kd_bow=auc(train_fpr, train_tpr)
test_auc_kd_bow=auc(test_fpr, test_tpr)

plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

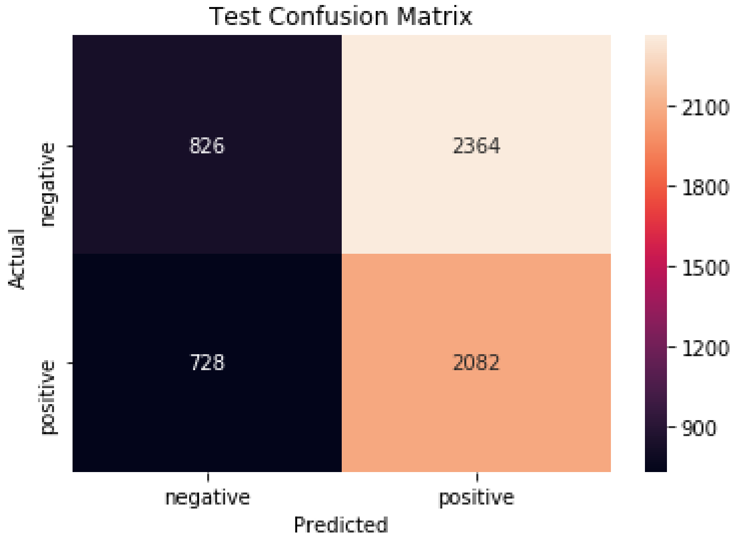


In [59]:

```
conf_matrix = confusion_matrix(y_train,neigh.predict(bow_train))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(bow_test))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(bow_test)))
print("_" * 101)
```



==



```
Classification Report on Test:
```

```
             precision    recall  f1-score   support

          0       0.53      0.26      0.35      3190
          1       0.47      0.74      0.57      2810

avg / total       0.50      0.48      0.45      6000
```

[5.2.2] Applying KNN kd-tree on TFIDF, SET 6

In [60]:
```
tf_idf_vectorizer = TfidfVectorizer(min_df=10, max_features=500)

X_train_tfidf= tf_idf_vectorizer.fit_transform(X_train)
X_cv_tfidf=tf_idf_vectorizer.transform(X_cv)
X_test_tfidf = tf_idf_vectorizer.transform(X_test)

print(X_train_tfidf.shape, y_train.shape)
print(X_cv_tfidf, y_cv.shape)
print(X_test_tfidf, y_test.shape)

from sklearn.preprocessing import StandardScaler

tfidf_train = StandardScaler().fit_transform(X_train_tfidf.todense())
tfidf_cv = StandardScaler().fit_transform(X_cv_tfidf.todense())
tfidf_test = StandardScaler().fit_transform(X_test_tfidf.todense())
```

```
(9800, 500) (9800,)
  (0, 493)	0.30927455257008246
  (0, 460)	0.31544112998182533
  (0, 438)	0.18875387055523463
  (0, 264)	0.2810640387268658
  (0, 226)	0.3303163216182056
  (0, 204)	0.16893942434086587
  (0, 187)	0.3496185786280846
  (0, 176)	0.24900044964384999
```

```
(0, 102)	0.3518924600687054
(0, 29)	0.3333298671331527
(0, 26)	0.3652399861575235
(1, 468)	0.29675268833221347
(1, 443)	0.332619127067219
(1, 438)	0.25279951728841316
(1, 406)	0.3947340373590649
(1, 201)	0.22681471196607483
(1, 182)	0.37676926131274413
(1, 135)	0.3351387416851761
(1, 57)	0.5252255097316536
(2, 494)	0.08878278769243407
(2, 470)	0.1308383215451572
(2, 469)	0.11241969376905739
(2, 468)	0.09871458183400601
(2, 462)	0.1042583863207774
(2, 451)	0.1450402300146749
:	:
(4197, 371)	0.2703482722768837
(4197, 358)	0.28563390676824396
(4197, 345)	0.27084684947777715
(4197, 313)	0.21166801150776693
(4197, 301)	0.19227129034151832
(4197, 249)	0.12950689417546418
(4197, 238)	0.2188173323514484
(4197, 138)	0.27948332402898907
(4197, 135)	0.5799580540416978
(4197, 84)	0.29872425274885606
(4197, 2)	0.24186292966338693
(4197, 0)	0.2516868720527706
(4198, 433)	0.4918508328215744
(4198, 345)	0.46153907678685197
(4198, 301)	0.16382083455639798
(4198, 271)	0.36437911775378423
(4198, 176)	0.3278052381515627
(4198, 24)	0.4388236111015016
(4198, 15)	0.29228854876973626
(4199, 322)	0.4773910983589288
(4199, 301)	0.18337571964848778
```

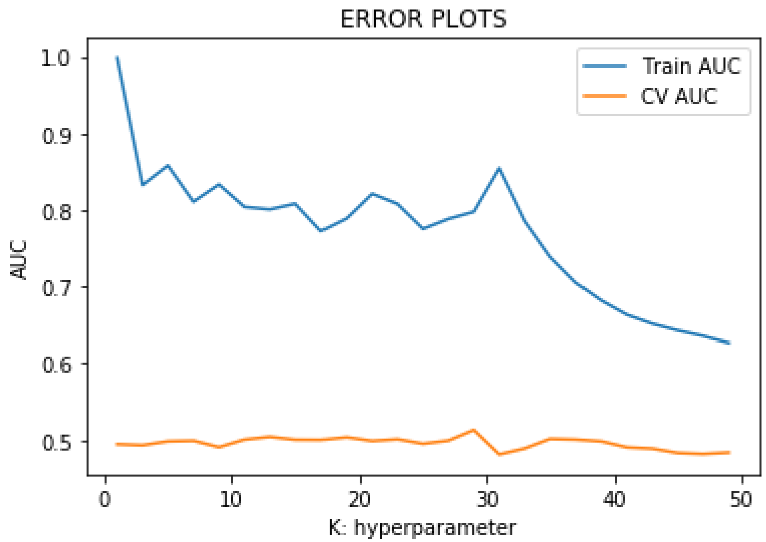
```
(4199, 234)	0.4145563236144401
(4199, 231)	0.5190465487613926
(4199, 182)	0.4145563236144401
(4199, 88)	0.35404501687285655 (4200,)
(0, 489)	0.10187039537084538
(0, 482)	0.16961233470269566
(0, 471)	0.11726345743701803
(0, 462)	0.08458907723727759
(0, 455)	0.1287953088666657
(0, 452)	0.09692998705048422
(0, 432)	0.10882455619765609
(0, 411)	0.10463142302559778
(0, 401)	0.1070336622346854
(0, 391)	0.11685619999403311
(0, 374)	0.10177853555138179
(0, 370)	0.1286689742286306
(0, 358)	0.13364385736756326
(0, 347)	0.07136729923340565
(0, 307)	0.10930698922459901
(0, 301)	0.17992175503959473
(0, 298)	0.27584225918695093
(0, 290)	0.12626068796096127
(0, 287)	0.1175938143721798
(0, 282)	0.10159576238317553
(0, 275)	0.12696055443881318
(0, 261)	0.1390141625377629
(0, 256)	0.10427650119105873
(0, 253)	0.16551184877357328
(0, 250)	0.12342428007661617
:	:
(5998, 49)	0.19903691260582262
(5998, 15)	0.16704272252016453
(5998, 10)	0.16934126418751758
(5999, 468)	0.12881533725809374
(5999, 463)	0.13925797850058588
(5999, 394)	0.22182668115769452
(5999, 378)	0.2000352416402707
(5999, 347)	0.22956846683023927
(5999, 311)	0.20197573484367115
```

```
(5999, 308)    0.1080568631977767
(5999, 301)    0.14468938119249952
(5999, 299)    0.12715160517860344
(5999, 289)    0.1290771389538596
(5999, 257)    0.20574660877747936
(5999, 176)    0.14476161468790777
(5999, 169)    0.1315431478916087
(5999, 142)    0.2139806585050739
(5999, 124)    0.2132697548515046
(5999, 123)    0.17205878474396277
(5999, 53)     0.17342540957674502
(5999, 47)     0.5887254441905742
(5999, 24)     0.19378828375177087
(5999, 20)     0.22388482410264998
(5999, 17)     0.18269280217178235
(5999, 15)     0.1290771389538596 (6000,)
```

In [61]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt
train_auc_kd_tfifd = []
cv_auc_kd_tfifd = []
myList = list(range(1,50,2))
parameters ={'n_neighbors':list(filter(lambda x: x % 2 != 0, myList))}
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='kd_tree')
    neigh.fit(tfidf_train, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(tfidf_train)[:,1]
    y_cv_pred =  neigh.predict_proba(tfidf_cv)[:,1]

    train_auc_kd_tfifd.append(roc_auc_score(y_train,y_train_pred))
    cv_auc_kd_tfifd.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc_kd_tfifd, label='Train AUC')
plt.plot(myList, cv_auc_kd_tfifd, label='CV AUC')
plt.legend()
```

```
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

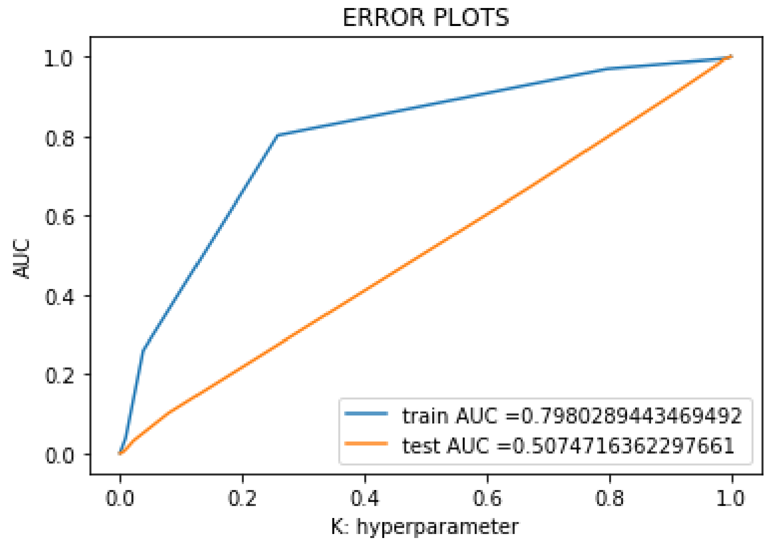


In [62]:
```
k_train_auc = dict(zip(myList, np.round(train_auc_kd_tfifd,3)))
k_cv_auc = dict(zip(myList, np.round(cv_auc_kd_tfifd,3)))
print(k_cv_auc)
print(k_train_auc)

best_k_kd_tfidf=29
```

```
{1: 0.494, 3: 0.493, 5: 0.498, 7: 0.499, 9: 0.491, 11: 0.5, 13: 0.504,
15: 0.5, 17: 0.5, 19: 0.503, 21: 0.499, 23: 0.501, 25: 0.495, 27: 0.49
9, 29: 0.513, 31: 0.481, 33: 0.489, 35: 0.501, 37: 0.501, 39: 0.498, 4
1: 0.49, 43: 0.489, 45: 0.483, 47: 0.482, 49: 0.484}
{1: 0.998, 3: 0.833, 5: 0.858, 7: 0.811, 9: 0.834, 11: 0.804, 13: 0.80
1, 15: 0.808, 17: 0.773, 19: 0.789, 21: 0.821, 23: 0.808, 25: 0.776, 2
7: 0.789, 29: 0.798, 31: 0.855, 33: 0.786, 35: 0.739, 37: 0.705, 39: 0.
682, 41: 0.663, 43: 0.652, 45: 0.643, 47: 0.636, 49: 0.627}
```

In [142]:
```
from sklearn.metrics import roc_curve, auc
```

```
neigh = KNeighborsClassifier(n_neighbors=best_k_kd_tfidf,algorithm='kd_
tree')
neigh.fit(tfidf_train, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(tfidf_train)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
tfidf_test)[:,1])

train_auc_kd_tfidf=auc(train_fpr, train_tpr)
test_auc_kd_tfidf=auc(test_fpr, test_tpr)

plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```



[5.2.3] Applying KNN kd-tree on AVG W2V, SET 3

In [144]:
```
i=0
list_of_sentance_train=[]
for sentance in X_train:
    list_of_sentance_train.append(sentance.split())
w2v_model=Word2Vec(list_of_sentance_train,min_count=5,size=50, workers=
4)
w2v_words = list(w2v_model.wv.vocab)

sent_vectors_kd_train = []; # the avg-w2v for each sentence/review is s
tored in this list
for sent in tqdm(list_of_sentance_train): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, yo
u might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/re
view
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
```

```
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors_kd_train.append(sent_vec)
sent_vectors_kd_train = np.array(sent_vectors_kd_train)
print(sent_vectors_train.shape)
print(sent_vectors_train[0])
```

```
100%|██████████████████████████████████████████████████████████████████
███████████| 9800/9800 [00:11<00:00, 854.22it/s]
```

```
(24500, 50)
[ 0.19264201  0.997602   -0.73982878  0.34566013 -0.30177539  0.0962433
1
  0.20634034 -0.33893895  0.14981144  0.1068354  -0.53528658 -0.4589625
3
  0.03290013 -0.61915606 -0.08654531  0.47429021  0.17087841 -0.4500000
6
 -0.85042399  0.19483005 -0.94428199  0.90886559 -0.03958788 -1.1084436
4
  0.05675411 -0.13973631 -0.83668128  0.59301265 -0.67348322  0.5519673
4
 -0.96322306  0.54090708 -1.02503414 -0.45494577 -0.54934807 -0.3689269
4
  0.98864671  0.19866709  0.66509121  0.47306288 -0.04872625  0.3256414
 -1.22825715 -0.05654036  0.80779135  0.55871361  0.67620215 -0.8712255
8
  0.60101495 -1.23098091]
```

In [145]:

```
i=0
list_of_sentance_cv=[]
for sentance in X_cv:
    list_of_sentance_cv.append(sentance.split())

# average Word2Vec
# compute average word2vec for each review.
sent_vectors_kd_cv = []; # the avg-w2v for each sentence/review is stor
ed in this list
```

```
for sent in tqdm(list_of_sentance_cv): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, yo
u might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/re
view
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors_kd_cv.append(sent_vec)
sent_vectors_kd_cv = np.array(sent_vectors_kd_cv)
print(sent_vectors_cv.shape)
print(sent_vectors_cv[0])
```

```
100%|██████████████████████████████████████████████████████████████████
███████████| 4200/4200 [00:05<00:00, 813.16it/s]
```

```
(10500, 50)
[-0.22272074  0.88077097 -0.5410872   0.40431074  0.00425091  0.0858636
8
  0.06575399 -0.14892975 -0.0851201   0.77006266 -0.04586707 -0.2443928
3
  0.33099695  0.14401136 -0.71115574  0.48575423  0.06450029  0.1414698
4
 -0.54609048  0.24432285 -0.66048661 -0.10162649 -0.05661974 -0.4685986
8
  0.52001499 -0.167588   -0.64922376  0.25735485  0.12726955 -0.0570445
2
 -0.44801857  0.35252933 -0.03972348  0.09060007 -0.10584066 -0.0019978
9
  0.75431736 -0.08563774 -0.11740405 -0.38904848 -0.31087601  0.5253288
4
 -1.20761307  0.09844579  0.35626311 -0.24025475  0.76640924 -0.0341810
8
  0.05425235 -0.46984297]
```

In [146]:
```python
i=0
list_of_sentance_test=[]
for sentance in X_test:
    list_of_sentance_test.append(sentance.split())

# average Word2Vec
# compute average word2vec for each review.
sent_vectors_kd_test = []; # the avg-w2v for each sentence/review is st
ored in this list
for sent in tqdm(list_of_sentance_test): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, yo
u might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/re
view
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors_kd_test.append(sent_vec)
sent_vectors_kd_test = np.array(sent_vectors_kd_test)
print(sent_vectors_kd_test.shape)
print(sent_vectors_kd_test[0])
```

```
100%|██████████████████████████████████████████████████████████████████
███████████| 6000/6000 [00:07<00:00, 839.24it/s]
```

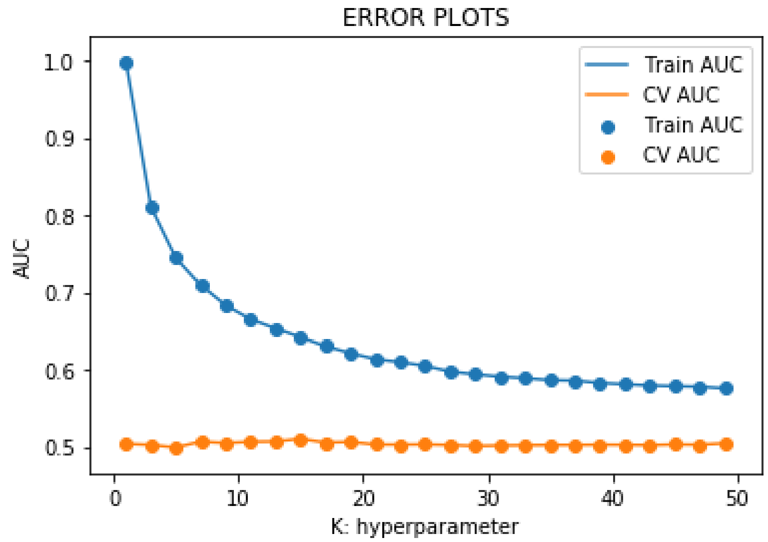
```
(6000, 50)
[-5.35536310e-01  9.70677045e-01 -8.80839754e-01  5.01268298e-01
 -2.03002271e-02  3.42551637e-01  6.30191199e-03  1.59570527e-01
 -6.56855075e-01  7.81928413e-01 -2.57449601e-01 -2.00287821e-01
  1.48178656e-01  1.72455452e-03 -3.33263941e-01  5.67379698e-01
  3.81932240e-01  8.09690594e-02 -8.81691185e-04  5.68689590e-02
 -4.57144687e-01 -3.86628588e-01 -1.27151693e-01  1.07977253e-01
  4.18637028e-01 -1.78533796e-02 -4.65203301e-01  3.06498173e-01
  1.37269768e-01 -1.74603312e-01 -2.34143366e-01  1.75003137e-01
 -5.41006165e-02 -2.40151707e-01 -5.80836074e-02 -1.79744987e-01
  5.60775316e-01 -3.39562531e-02 -3.45295588e-01 -6.45069900e-01
  6.06350086e-01  4.97571370e-02  7.13356188e-01  2.87369984e-01
```

```
-6.06350086e-01  4.97571270e-02 -7.13356188e-01  2.87369984e-01
 2.26524363e-01 -2.94855656e-01  4.14948794e-01 -1.53534245e-01
 4.54516139e-01  3.47491843e-02]
```

In [ ]:

In [68]:

```
train_auc_kd_avgw2v = []
cv_auc_kd_avgw2v = []
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='kd_tree')
    neigh.fit(sent_vectors_kd_train, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(sent_vectors_kd_train)[:,1]
    y_cv_pred =  neigh.predict_proba(sent_vectors_kd_cv)[:,1]

    train_auc_kd_avgw2v.append(roc_auc_score(y_train,y_train_pred))
    cv_auc_kd_avgw2v.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc, label='Train AUC')
plt.scatter(myList, train_auc, label='Train AUC')
plt.plot(myList, cv_auc, label='CV AUC')
plt.scatter(myList, cv_auc, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```



In [70]:

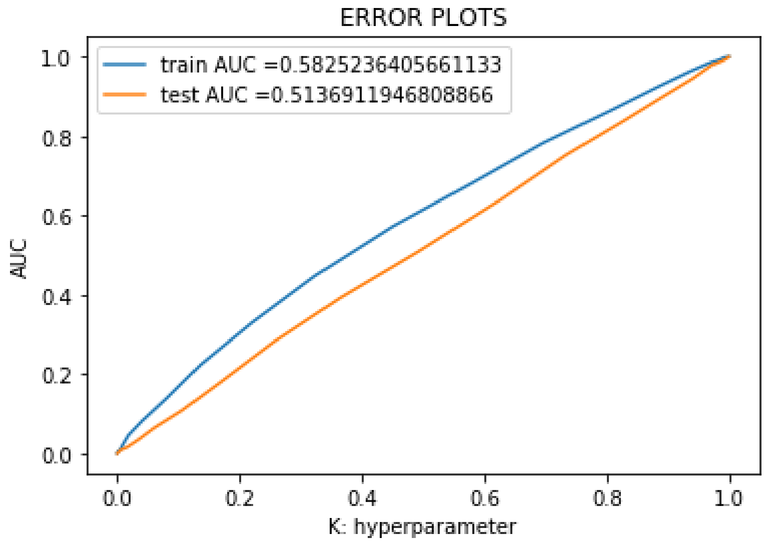
```
k_train_auc_kd_avgw2v = dict(zip(myList, np.round(train_auc_kd_avgw2v,3
)))
k_cv_auc_kd_avgw2v = dict(zip(myList, np.round(cv_auc_kd_avgw2v,3)))
print(k_train_auc_kd_avgw2v)
print(k_cv_auc_kd_avgw2v)

best_k_kd_avgw2v=41
```

```
{1: 0.999, 3: 0.819, 5: 0.755, 7: 0.712, 9: 0.689, 11: 0.671, 13: 0.65
4, 15: 0.642, 17: 0.635, 19: 0.629, 21: 0.621, 23: 0.615, 25: 0.608, 2
7: 0.606, 29: 0.6, 31: 0.598, 33: 0.592, 35: 0.589, 37: 0.586, 39: 0.58
4, 41: 0.581, 43: 0.578, 45: 0.575, 47: 0.573, 49: 0.572}
{1: 0.502, 3: 0.506, 5: 0.508, 7: 0.505, 9: 0.502, 11: 0.502, 13: 0.50
9, 15: 0.506, 17: 0.512, 19: 0.518, 21: 0.516, 23: 0.515, 25: 0.516, 2
7: 0.521, 29: 0.521, 31: 0.523, 33: 0.527, 35: 0.522, 37: 0.521, 39: 0.
523, 41: 0.525, 43: 0.522, 45: 0.524, 47: 0.52, 49: 0.516}
```

In [147]:

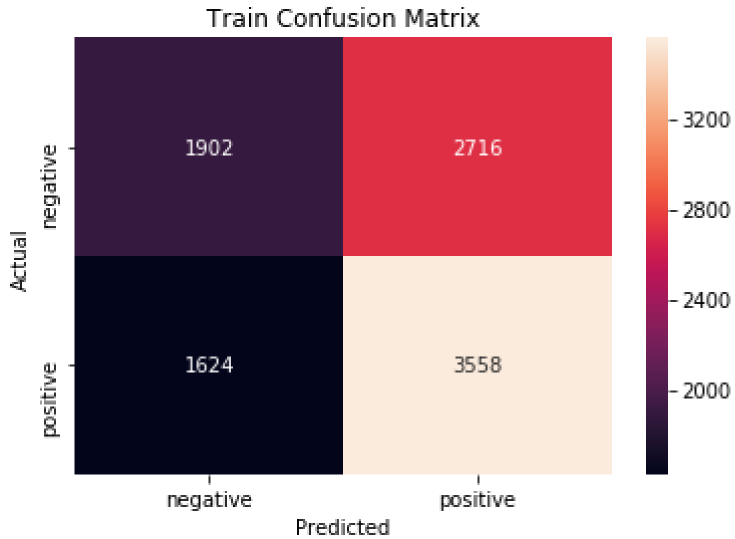
```
from sklearn.metrics import roc_curve, auc


neigh = KNeighborsClassifier(n_neighbors=best_k_kd_avgw2v,algorithm='kd
```

```
_tree')
neigh.fit(sent_vectors_kd_train, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs

train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(sent_vectors_kd_train)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
sent_vectors_kd_test)[:,1])

train_auc_kd_avgw2v=auc(train_fpr, train_tpr)
test_auc_kd_avgw2v=auc(test_fpr, test_tpr)

plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

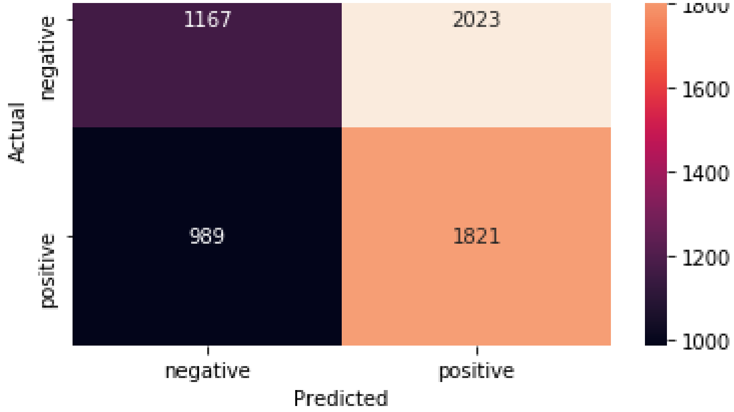


In [78]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sb
from sklearn.metrics import classification_report


conf_matrix = confusion_matrix(y_train,neigh.predict(sent_vectors_kd_tr
ain))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(sent_vectors_kd_tes
t))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
```

```
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(sent_vectors_kd_test
)))
print("_" * 101)
```




```
================================================================================
==============================
```




```
____________________________________________________________________________________________

Classification Report on Test:

             precision    recall  f1-score   support

          0       0.54      0.37      0.44      3190
          1       0.47      0.65      0.55      2810

avg / total       0.51      0.50      0.49      6000

____________________________________________________________________________________________
```

[5.2.4] Applying KNN kd-tree on TFIDF W2V, SET 4

In [79]:
```
#Source:https://medium.com/@mohithsai504/sentiment-analysis-for-amazon-fine-food-reviews-using-k-nn-1ae8be11908b

X_train, X_test, y_train, y_test = train_test_split(sample_points,labels, test_size=0.30,shuffle=False)# this is for time series split
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_size=0.30,shuffle=False)

print(X_train.shape, y_train.shape)
```

```
print(X_test.shape, y_test.shape)
print(X_cv.shape,y_cv.shape)


model = TfidfVectorizer()
tf_idf_train_matrix = model.fit_transform(X_train)
# we are converting a dictionary with word as a key, and the idf as a v
alue
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

```
(9800,) (9800,)
(6000,) (6000,)
(4200,) (4200,)
```

TFIFD on Train Data

In [149]:

```
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and ce
ll_val = tfidf

tfidf_sent_vectors_kd_train = []; # the tfidf-w2v for each sentence/rev
iew is stored in this list
row=0;
for sent in tqdm(list_of_sentance_train): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/r
eview
    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_kd_train.append(sent_vec)
    row += 1
tfidf_sent_vectors_kd_train=np.array(tfidf_sent_vectors_kd_train)
print(tfidf_sent_vectors_kd_train.shape)
```

```
100%|██████████████████████████████████████████████████████████
██████████| 9800/9800 [01:26<00:00, 112.83it/s]
```

```
(9800, 50)
```

TFIFD on CV Data

In [150]:

```
tf_idf_train_matrix = model.transform(X_cv)
# we are converting a dictionary with word as a key, and the idf as a v
alue
dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))
```

In [151]:

```
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and ce
ll_val = tfidf

tfidf_sent_vectors_kd_cv = []; # the tfidf-w2v for each sentence/review
 is stored in this list
row=0;
for sent in tqdm(list_of_sentance_cv): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/r
eview
    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_kd_cv.append(sent_vec)
    row += 1
```

```
100%|██████████████████████████████████████████████████| 4200/4200 [00:38<00:00, 108.51it/s]
```

TFIDF on test Data

In [152]:

```
tf_idf_train_matrix = model.transform(X_test)
# 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 [153]:

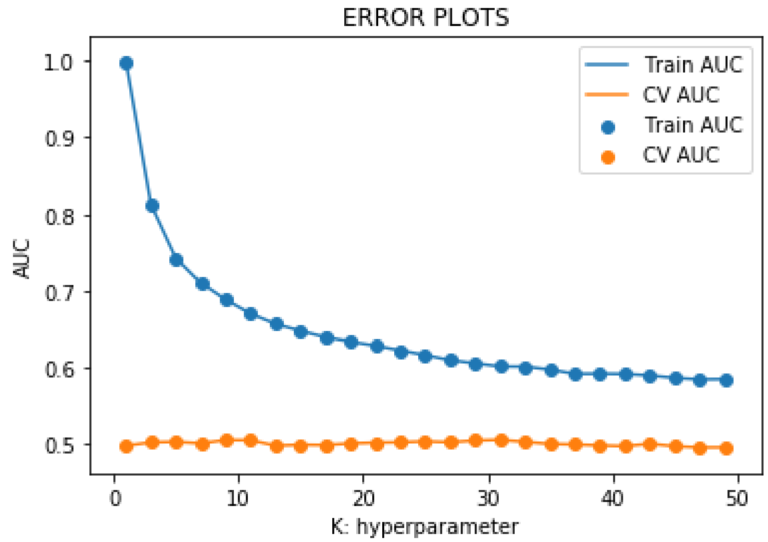
```
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and cell_val = tfidf

tfidf_sent_vectors_kd_test = []; # the tfidf-w2v for each sentence/review is stored in this list
row=0;
for sent in tqdm(list_of_sentance_test): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/review
    for word in sent: # for each word in a review/sentence
        if word in w2v_words 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_kd_test.append(sent_vec)
    row += 1
```

```
100%|██████████████████████████████████████████████████
█████████| 6000/6000 [01:05<00:00, 91.64it/s]
```

In [91]:

```
train_auc_kd_tfidfw2v = []
cv_auc_kd_tfidfw2v = []
for i in myList:
    neigh = KNeighborsClassifier(n_neighbors=i,algorithm='kd_tree')
    neigh.fit(tfidf_sent_vectors_kd_train, y_train)
    # roc_auc_score(y_true, y_score) the 2nd parameter should be probab
ility estimates of the positive class
    # not the predicted outputs
    y_train_pred =  neigh.predict_proba(tfidf_sent_vectors_kd_train)[:,
1]
    y_cv_pred =  neigh.predict_proba(tfidf_sent_vectors_kd_cv)[:,1]

    train_auc_kd_tfidfw2v.append(roc_auc_score(y_train,y_train_pred))
    cv_auc_kd_tfidfw2v.append(roc_auc_score(y_cv, y_cv_pred))

plt.plot(myList, train_auc_kd_tfidfw2v, label='Train AUC')
plt.scatter(myList, train_auc_kd_tfidfw2v, label='Train AUC')
plt.plot(myList, cv_auc_kd_tfidfw2v, label='CV AUC')
plt.scatter(myList, cv_auc_kd_tfidfw2v, label='CV AUC')
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

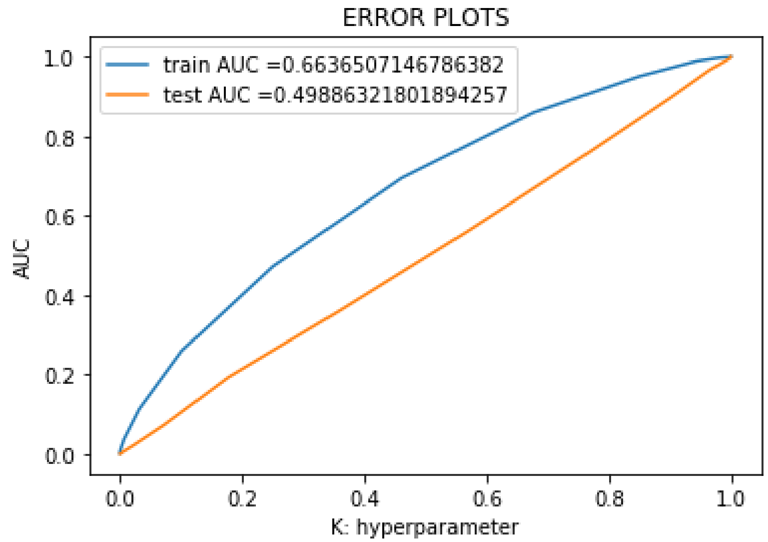


In [93]:
```
k_train_auc_kd_tfidfw2v  = dict(zip(myList, np.round(train_auc_kd_tfidf
w2v,3)))
k_cv_auc_kd_tfidfw2v  = dict(zip(myList, np.round(cv_auc_kd_tfidfw2v,3
)))
print(k_train_auc_kd_tfidfw2v)
print(k_cv_auc_kd_tfidfw2v)

best_k_kd_tfidfw2v=11
```

```
{1: 0.999, 3: 0.813, 5: 0.743, 7: 0.711, 9: 0.689, 11: 0.67, 13: 0.657,
15: 0.648, 17: 0.64, 19: 0.634, 21: 0.629, 23: 0.623, 25: 0.617, 27: 0.
61, 29: 0.606, 31: 0.602, 33: 0.602, 35: 0.598, 37: 0.592, 39: 0.592, 4
1: 0.592, 43: 0.59, 45: 0.587, 47: 0.585, 49: 0.585}
{1: 0.498, 3: 0.503, 5: 0.503, 7: 0.501, 9: 0.506, 11: 0.506, 13: 0.49
8, 15: 0.5, 17: 0.499, 19: 0.501, 21: 0.502, 23: 0.503, 25: 0.504, 27:
0.503, 29: 0.505, 31: 0.506, 33: 0.503, 35: 0.501, 37: 0.5, 39: 0.499,
41: 0.498, 43: 0.501, 45: 0.498, 47: 0.496, 49: 0.496}
```

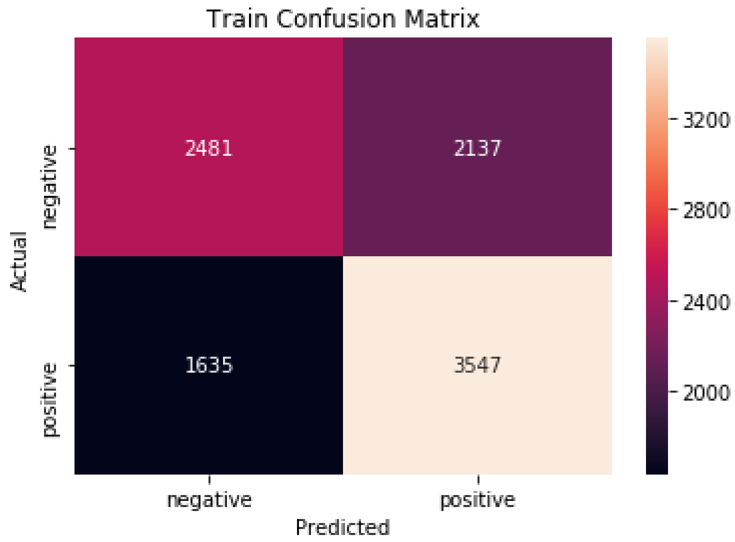
In [154]:
```
from sklearn.metrics import roc_curve, auc
```

```
neigh = KNeighborsClassifier(n_neighbors=best_k_kd_tfidfw2v,algorithm=
'kd_tree')
neigh.fit(tfidf_sent_vectors_kd_train, y_train)
# roc_auc_score(y_true, y_score) the 2nd parameter should be probabilit
y estimates of the positive class
# not the predicted outputs



train_fpr, train_tpr, thresholds = roc_curve(y_train, neigh.predict_pro
ba(tfidf_sent_vectors_kd_train)[:,1])
test_fpr, test_tpr, thresholds = roc_curve(y_test, neigh.predict_proba(
tfidf_sent_vectors_kd_test)[:,1])

train_auc_kd_tfidfw2v=auc(train_fpr, train_tpr)
test_auc_kd_tfidfw2v=auc(test_fpr, test_tpr)


plt.plot(train_fpr, train_tpr, label="train AUC ="+str(auc(train_fpr, t
rain_tpr)))
plt.plot(test_fpr, test_tpr, label="test AUC ="+str(auc(test_fpr, test_
tpr)))
plt.legend()
plt.xlabel("K: hyperparameter")
plt.ylabel("AUC")
plt.title("ERROR PLOTS")
plt.show()
```

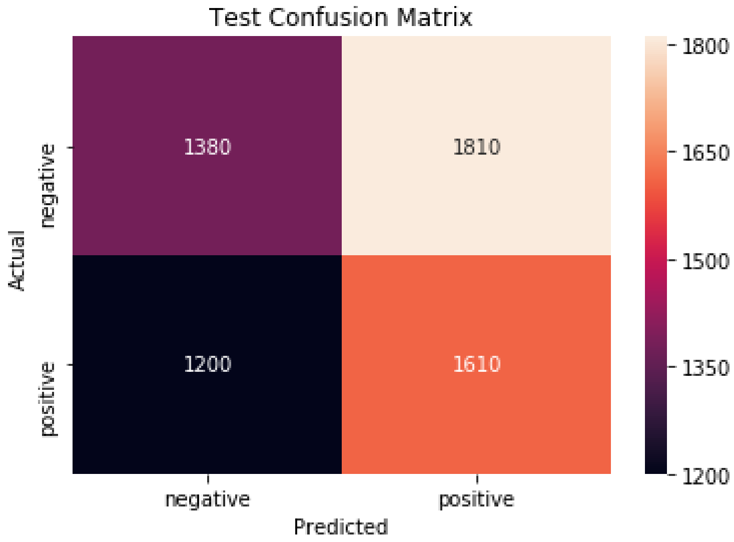


In [95]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sb
from sklearn.metrics import classification_report


conf_matrix = confusion_matrix(y_train,neigh.predict(tfidf_sent_vectors
_kd_train))
class_label = ['negative', 'positive']
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Train Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
print("="*101)

#Printing Confusion Matrix for Train & Test
conf_matrix = confusion_matrix(y_test,neigh.predict(tfidf_sent_vectors_
kd_test))
class_label = ['negative', 'positive']
```

```
df_conf_matrix = pd.DataFrame(
    conf_matrix, index=class_label, columns=class_label)
sb.heatmap(df_conf_matrix, annot=True, fmt='d')
plt.title("Test Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#Printing Classification Report

print("_" * 101)
print("Classification Report on Test: \n")
print(classification_report(y_test, neigh.predict(sent_vectors_kd_test
)))
print("_" * 101)
```




```
=============================================================================
================================
```




```
____________________________________________________________________________
________________________________
Classification Report on Test:

             precision    recall  f1-score   support

          0       0.53      0.43      0.47      3190
          1       0.47      0.57      0.51      2810

avg / total       0.50      0.49      0.49      6000

____________________________________________________________________________
________________________________
```

[6] Conclusions

In [136]:

```
from prettytable import PrettyTable
X = PrettyTable()
print("--------------------Brute Force Models--------------------------
--")
```

```
X.field_names = (["Model Name","K-Value","Train AUC Score","Test AUC Sc
ore"])
X.add_row(["BOW",best_k_bow,train_auc_k_bow,test_auc_k_bow])
X.add_row(["TF-IDF",best_k_tfidf,train_k_auc_tfidf,test_k_auc_tfidf])
X.add_row(["AVG-W2V",best_k_avgw2v,train_auc_k_avgw2v,test_auc_k_avgw2v
])
X.add_row(["TFIDF W2V",best_k_tfidfw2v,train_auc_k_tfidfw2v,test_auc_k_
tfidfw2v])
print(X)
```

```
--------------------Brute Force Models--------------------------
+------------+---------+--------------------+--------------------+
| Model Name | K-Value |  Train AUC Score   |   Test AUC Score   |
+------------+---------+--------------------+--------------------+
|    BOW     |    5    | 0.7476859632879848 | 0.5001514965584177 |
|   TF-IDF   |    29   | 0.7575370000997891 | 0.5052304800365912 |
|  AVG-W2V   |    49   | 0.5878923554041885 | 0.5005974742319552 |
| TFIDF W2V  |    13   | 0.5818292172768985 | 0.4961414226673139 |
+------------+---------+--------------------+--------------------+
```

Based on the above brute force models we can say that all the models performs average and Avg W2V model performs best among all. We need to look into some other model for better performance.

In [156]:

```
Y = PrettyTable()
print("--------------------KD Tree Models--------------------------")
Y.field_names = (["Model Name","K-Value","Train AUC Score","Test AUC Sc
ore"])
Y.add_row(["BOW",best_k_kd_bow,train_auc_kd_bow,test_auc_kd_bow])
Y.add_row(["TF-IDF",best_k_kd_tfidf,train_auc_kd_tfidf,test_auc_kd_tfid
f])
Y.add_row(["AVG-W2V",best_k_kd_avgw2v,train_auc_kd_avgw2v,test_auc_kd_a
vgw2v])
Y.add_row(["TFIDF W2V",best_k_kd_tfidfw2v,train_auc_kd_tfidfw2v,test_au
c_kd_tfidfw2v])
print(Y)
```

```
--------------------KD Tree Models--------------------------
+------------+---------+--------------------+--------------------+
```

Model Name	K-Value	Train AUC Score	Test AUC Score
BOW	13	0.682508237612992	0.4968628052521782
TF-IDF	29	0.7980289443469492	0.5074716362297661
AVG-W2V	41	0.5825236405661133	0.5136911946808866
TFIDF W2V	13	0.6636507146786382	0.49886321801894257

Based on the above Kd tree models we can say that all the models performs average and Avg W2V model performs best among all. We need to look into some other model for better performance.