Assignment 9: GBDT

Response Coding: Example

Train Data

State	class
A	0
B	1
C	1
A	0
A	1
B	1
A	0
A	1
C	1
C	0

Resonse table(only from train)

State	Class=0	Class=1
A	3	2
B	0	2
C	1	2

Encoded Train Data

State_0	State_1	class
3/5	2/5	0
0/2	2/2	1
1/3	2/3	1
3/5	2/5	0
3/5	2/5	1
0/2	2/2	1
3/5	2/5	0
3/5	2/5	1
1/3	2/3	1
1/3	2/3	0

Test Data

State
A
C
D
C
B
E

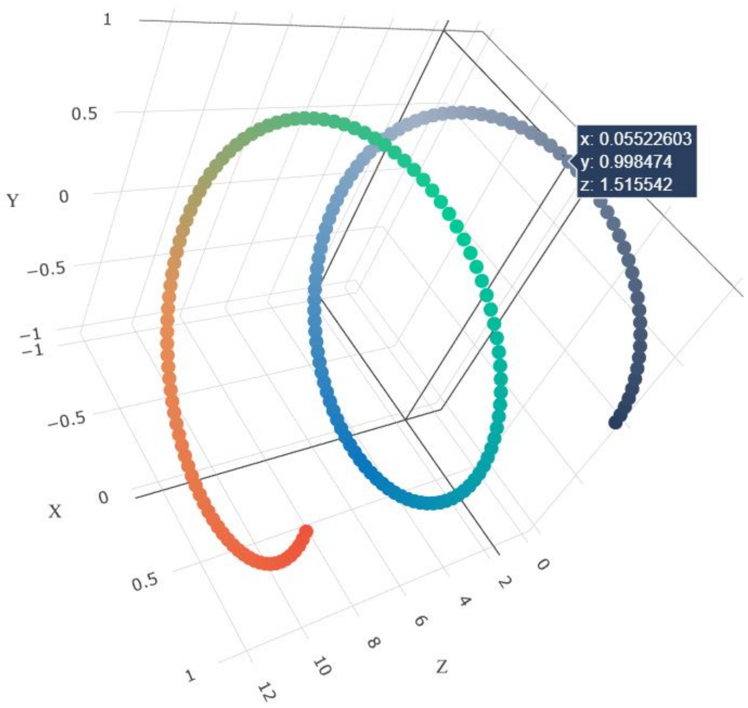
Encoded Test Data

State_0	State_1
3/5	2/5
1/3	2/3
1/2	1/2
1/3	2/3
0/2	2/2
1/2	1/2

> The response tabel is built only on train dataset. For a category which is not there in train data and present in test data, we will encode them with default values Ex: in our test data if have State: D then we encode it as [0.5, 0.05]

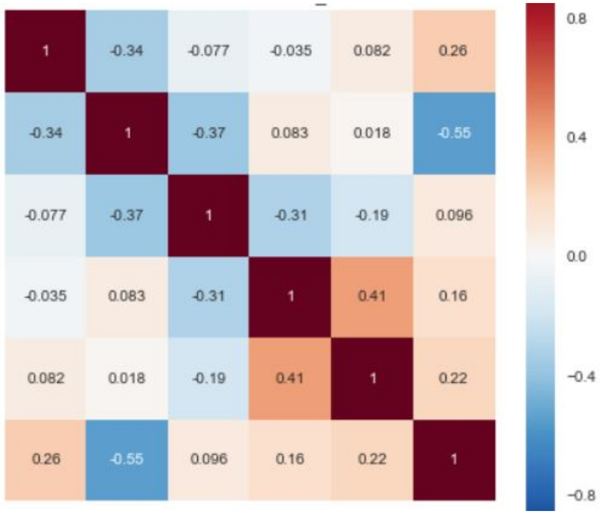
1. **Apply GBDT on these feature sets**
    - Set 1: categorical(instead of one hot encoding, try response coding: use probability values), numerical features + project_title(TFIDF)+ preprocessed_eassay (TFIDF)+sentiment Score of eassay(check the bellow example, include all 4 values as 4 features)
    - Set 2: categorical(instead of one hot encoding, try response coding: use probability values), numerical features + project_title(TFIDF W2V)+ preprocessed_eassay (TFIDF W2V)
2. **The hyper paramter tuning (Consider any two hyper parameters)**
    - Find the best hyper parameter which will give the maximum AUC value
    - find the best hyper paramter using k-fold cross validation/simple cross validation data
    - use gridsearch cv or randomsearch cv or you can write your own for loops to do this task
3. **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

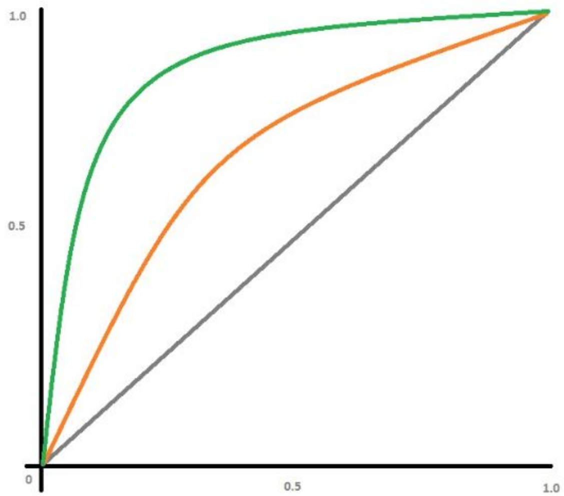


with X-axis as **n_estimators**, Y-axis as **max_depth**, and Z-axis as **AUC Score** , we have given the notebook which explains how to plot this 3d plot, you can find it in the same drive *3d_scatter_plot.ipynb*

or

- 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 

seaborn heat maps with rows as **n_estimators**, columns as **max_depth**, and values inside the cell representing **AUC Score**
- You choose either of the plotting techniques out of 3d plot or heat map
- 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

	Predicted: NO	Predicted: YES
Actual: NO	TN = ??	FP = ??
Actual: YES	FN = ??	TP = ??

4. You need to summarize the results at the end of the notebook, summarize it in the table format

```
+-------------+--------+-----------------+------+
|  Vectorizer | Model  | Hyper parameter | AUC  |
+-------------+--------+-----------------+------+
|     BOW     | Brute  |        7        | 0.78 |
+-------------+--------+-----------------+------+
|    TFIDF    | Brute  |        12       | 0.79 |
+-------------+--------+-----------------+------+
|     W2V     | Brute  |        10       | 0.78 |
+-------------+--------+-----------------+------+
|  TFIDFW2V   | Brute  |        6        | 0.78 |
+-------------+--------+-----------------+------+
```

REQUIRED LIBRARIES:-

In [1]:

```
# NECESSARY LIBRARIES:
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np
import nltk
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix
```

```python
from sklearn import metrics
from sklearn.metrics import roc_curve, auc

import re
# Tutorial about Python regular expressions: https://pymotw.com/2/re/

import pickle
from tqdm import tqdm
import os
```

In [2]:

```python
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# import nltk
# nltk.download('vader_lexicon')

sid = SentimentIntensityAnalyzer()

for_sentiment = 'a person is a person no matter how small dr seuss i teach the smallest students with the biggest ent
for learning my students learn in many different ways using all of our senses and multiple intelligences i use a wide
of techniques to help all my students succeed students in my class come from a variety of different backgrounds which
for wonderful sharing of experiences and cultures including native americans our school is a caring community of succ
learners which can be seen through collaborative student project based learning in and out of the classroom kindergar
in my class love to work with hands on materials and have many different opportunities to practice a skill before it
mastered having the social skills to work cooperatively with friends is a crucial aspect of the kindergarten curricul
montana is the perfect place to learn about agriculture and nutrition my students love to role play in our pretend ki
in the early childhood classroom i have had several kids ask me can we try cooking with real food i will take their i
and create common core cooking lessons where we learn important math and writing concepts while cooking delicious hea
food for snack time my students will have a grounded appreciation for the work that went into making the food and kno
of where the ingredients came from as well as how it is healthy for their bodies this project would expand our learni
nutrition and agricultural cooking recipes by having us peel our own apples to make homemade applesauce make our own
and mix up healthy plants from our classroom garden in the spring we will also create our own cookbooks to be printed
shared with families students will gain math and literature skills as well as a life long enjoyment for healthy cooki
nannan'
ss = sid.polarity_scores(for_sentiment)

for k in ss:
    print('{0}: {1}, '.format(k, ss[k]), end='')

# we can use these 4 things as features/attributes (neg, neu, pos, compound)
# neg: 0.0, neu: 0.753, pos: 0.247, compound: 0.93
```

```
neg: 0.01, neu: 0.745, pos: 0.245, compound: 0.9975,
```

GBDT (xgboost/lightgbm)

SET (1):-

Loading Data

In [3]:

```
import pandas as pd
data = pd.read_csv('final_preprocessed.csv')
print("The shape of the data : ",data.shape)
data = data.iloc[:,0:12]
data.head(3)
```

```
The shape of the data :  (50000, 11)
```

Out[3]:

	id	teacher_prefix	school_state	project_grade_category	project_subject_categories	project_subject_subcategories	teacher_number_of_pr
0	p036502	ms	nv	grades_prek_2	literacy_language	literacy	
1	p039565	mrs	ga	grades_3_5	music_arts_health_sports	performingarts_teamsports	
2	p233823	ms	ut	grades_3_5	math_science_literacy_language	appliedsciences_literature_writing	

CHECK DISTRIBUTION OF TARGET COLUMN:-

In [4]:

```
count_majority,count_minority = data["project_is_approved"].value_counts()
print("No of majority class points:",count_majority)
print("No of minority class points:",count_minority)
```

```
No of majority class points: 42466
No of minority class points: 7534
```

In [5]:
```python
# Take the dataset for splitting:
y = data["project_is_approved"].values # returns a numpy nd array--> Target variables
X = data.drop("project_is_approved",axis = 1) # Creates a dataframe X-->Input variables
```

Splitting data into Train and cross validation(or test): Stratified Sampling

In [6]:
```python
# Split data into train and test set:(stratified sampling):

from sklearn.model_selection import train_test_split # necessary library

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.30,stratify = y,
                                               random_state=17)
```

In [7]:
```python
# Check the shape of X_train,X_test,y_train & y_test :
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(35000, 10)
(15000, 10)
(35000,)
(15000,)
```

Make Data Model Ready: encoding essay, and project_title(SET 1)

In [8]:
```python
#TEXT FEATURE --> ESSAY ENCODING INTO NUMERIC VECTOR(tfidf):

from sklearn.feature_extraction.text import TfidfVectorizer

print("(i) Shape before vectorization of essay(feature) :")
print(X_train.shape,y_train.shape)
print(X_test.shape,y_test.shape)
print("\n")

# Use the tfidf vectorizer to encode the text data (essay)
vectorizer = TfidfVectorizer(min_df= 30,ngram_range=(1,2),max_features=7500)
vectorizer.fit(X_train['preprocessed_essay'].values) # fit on train data

# we use the fitted Vectorizer to convert the text to vector
```

```
X_train_essay_tfidf = vectorizer.transform(X_train['preprocessed_essay'].values)
X_test_essay_tfidf = vectorizer.transform(X_test['preprocessed_essay'].values)

print("(ii) Shape After vectorization of essay(feature) :")
print(X_train_essay_tfidf.shape, y_train.shape)
print(X_test_essay_tfidf.shape, y_test.shape)
print("\n")

#////////////////////////////////////////////////////////////////////////////////
#////////////////////////////////////////////////////////////////////////////////

print("(iii) Shape before vectorization of title(feature) :")
print(X_train.shape,y_train.shape)
print(X_test.shape,y_test.shape)
print("\n")

# Use the tfidf vectorizer to encode the text data(title)
vectorizer = TfidfVectorizer(min_df= 25,ngram_range=(1,3),max_features=1500)
vectorizer.fit(X_train['preprocessed_title'].values.astype('U')) # fit on train data

# we use the fitted Vectorizer to convert the text to vector
X_train_title_tfidf = vectorizer.transform(X_train['preprocessed_title'].values.astype('U'))
X_test_title_tfidf = vectorizer.transform(X_test['preprocessed_title'].values.astype('U'))

print("(iv) Shape After vectorization of title(feature) :")
print(X_train_title_tfidf.shape, y_train.shape)
print(X_test_title_tfidf.shape, y_test.shape)
```

```
(i) Shape before vectorization of essay(feature) :
(35000, 10) (35000,)
(15000, 10) (15000,)


(ii) Shape After vectorization of essay(feature) :
(35000, 7500) (35000,)
(15000, 7500) (15000,)


(iii) Shape before vectorization of title(feature) :
(35000, 10) (35000,)
(15000, 10) (15000,)
```

```
(iv) Shape After vectorization of title(feature) :
(35000, 1006) (35000,)
(15000, 1006) (15000,)
```

Response_encoder CUSTOM FUNCTION :-

In [140…

```
# define response_encoder() for train_data:-

def response_encoder(data):

    list_1 = [];list_2 = []
    counts = data.iloc[:,0].value_counts()
    # create a pivot table to get counts:-
    table = data.pivot_table(index = list(data.columns), aggfunc='size')
    t = dict(table) # convert table to a dictionary
    for i in range(data.shape[0]):
        if data.iloc[i,1] == 0:
            value_1 = t[(data.iloc[i,0],data.iloc[i,1])]/dict(counts)[data.iloc[i,0]]
            value_2 = 1 - value_1
            list_1.append(value_1)
            list_2.append(value_2)
        else:
            value_1 = t[(data.iloc[i,0],data.iloc[i,1])]/dict(counts)[data.iloc[i,0]]
            value_2 = 1 - value_1
            list_1.append(value_2)
            list_2.append(value_1)

    df = pd.DataFrame({"feature_0" : list_1,"feature_1" : list_2})
    return df # return a dataframe

# define response_encoder() for test_data:-
def response_encoder_test(test_data,train_data):

    list_1 = [];list_2 = []
    counts = train_data.iloc[:,0].value_counts()
    # create a pivot table to get counts:-
    table = train_data.pivot_table(index = list(train_data.columns), aggfunc='size')
    t = dict(table) # convert table to a dictionary
    for i in range(test_data.shape[0]):
        if test_data.iloc[i,0] in list(counts.index):
            if (test_data.iloc[i,0],0) in list(t.keys()):
```

```
                value_1 = t[(test_data.iloc[i,0],0)]/dict(counts)[test_data.iloc[i,0]]
                value_2 = 1 - value_1
                list_1.append(value_1)
                list_2.append(value_2)
            else:
                value_1 = t[(test_data.iloc[i,0],1)]/dict(counts)[test_data.iloc[i,0]]
                value_2 = 1 - value_1
                list_1.append(value_2)
                list_2.append(value_1)
        else:
            value_1 = 1/2
            value_2 = 1/2
            list_1.append(value_1)
            list_2.append(value_2)
    df_test = pd.DataFrame({"feature_0" : list_1,"feature_1" : list_2})
    return df_test # return a dataframe
```

GRADING Response_encoder() for train & test:-

In [10]:

```
# check the functions for sample train & test
train = pd.DataFrame({"feature" : ['A','B','C','A','A','B','A','A','C','C'],
                "class_col" :[0,1,1,0,1,1,0,1,1,0]})
print("The response coded data for sample_train data:")
response_encoder(train) #call the function
```

```
The response coded data for sample_train data:
```

Out[10]:

	feature_0	feature_1
0	0.600000	0.400000
1	0.000000	1.000000
2	0.333333	0.666667
3	0.600000	0.400000
4	0.600000	0.400000
5	0.000000	1.000000
6	0.600000	0.400000
7	0.600000	0.400000

	feature_0	feature_1
8	0.333333	0.666667
9	0.333333	0.666667

In [11]:

```
test = pd.DataFrame({"feature" : ['A','C','D','C','B','E']})

print("The response coded data for sample_test data:")
response_encoder_test(test,train) # call the function
```

```
The response coded data for sample_test data:
```

Out[11]:

	feature_0	feature_1
0	0.600000	0.400000
1	0.333333	0.666667
2	0.500000	0.500000
3	0.333333	0.666667
4	0.000000	1.000000
5	0.500000	0.500000

Make Data Model Ready: encoding numerical, categorical features

In [50]:

```
import numpy as np
# Normalizing: map all the values to range of (0,1) -- > [PRICE]

from sklearn.preprocessing import Normalizer
normalizer = Normalizer()

# fit and transform the train and test data:
# reshape of the data to single row allows the normalizer() to fit & transform

normalizer.fit(X_train['price'].values.reshape(1,-1))
X_train_price_norm = normalizer.transform(X_train['price'].values.reshape(1,-1))
X_test_price_norm = normalizer.transform(X_test['price'].values.reshape(1,-1))
```

```
print("(A) Shape After Nomalization of feature Price :")
print(X_train_price_norm.transpose().shape, y_train.shape)
print(X_test_price_norm.transpose().shape, y_test.shape)

# assign to another variable for readability purpose :
X_train_price_norm = X_train_price_norm.transpose()
X_test_price_norm  = X_test_price_norm.transpose()
print("="*100)


#Normalizing: map the values to range of (0,1)-- >[TEACHER_NO_OF_PREVIOUSLY_POSTED_PROJECTS]

from sklearn.preprocessing import Normalizer
normalizer = Normalizer()

# fit and transform the train and test data:
# reshape of the data to single row allows the normalizer() to fit & transform
normalizer.fit(X_train['teacher_number_of_previously_posted_projects'].
               values.reshape(1,-1))
X_train_previous_norm = normalizer.transform(X_train['teacher_number_of_previously_posted_projects'].
                                             values.reshape(1,-1))
X_test_previous_norm = normalizer.transform(X_test['teacher_number_of_previously_posted_projects'].
                                            values.reshape(1,-1))

print("(B) Shape After Normalization of Number of previously posted projects :")
print(X_train_previous_norm.transpose().shape, y_train.shape)
print(X_test_previous_norm.transpose().shape, y_test.shape)

# assign to another variable for readability purpose :
X_train_previous_norm = X_train_previous_norm.transpose()
X_test_previous_norm = X_test_previous_norm.transpose()
print("="*100)

#################################################################################
#################################################################################


# Response coding for categorical features:
# CATEGORICAL FEATURE--> SCHOOL STATE:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['school_state'],y_train))
t2 = pd.DataFrame(t1,columns = ['school_state','project_is_approved'])
```

```
t3 = pd.DataFrame(X_test['school_state'],columns = ['school_state'])

X_train_state_school = response_encoder(t2)
X_test_state_school = response_encoder_test(t3,t2)

print("(C) Shape After vectorization of school state :")
print(X_train_state_school.shape, y_train.shape)
print(X_test_state_school.shape, y_test.shape)
print("="*100)


# CATEGORICAL FEATURES TEACHER_PREFIX:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['teacher_prefix'],y_train))
t2 = pd.DataFrame(t1,columns = ['teacher_prefix','project_is_approved'])
t3 = pd.DataFrame(X_test['teacher_prefix'],columns = ['teacher_prefix'])

t4 = t2.fillna('unknown',inplace=False)
t5 = t3.fillna('unknown',inplace=False)
X_train_prefix_teacher = response_encoder(t4)
X_test_prefix_teacher = response_encoder_test(t5,t4)

print("(D) Shape After vectorization of teacher_prefix :")
print(X_train_prefix_teacher.shape, y_train.shape)
print(X_test_prefix_teacher.shape, y_test.shape)
print("="*100)




# CATEGORICAL FEATURES project_grade_category:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['project_grade_category'],y_train))
t2 = pd.DataFrame(t1,columns = ['project_grade_category','project_is_approved'])
t3 = pd.DataFrame(X_test['project_grade_category'],columns = ['project_grade_category'])

X_train_grade_cat = response_encoder(t2)
X_test_grade_cat = response_encoder_test(t3,t2)

print("(E) Shape After vectorization of project_grade_Category :")
print(X_train_grade_cat.shape, y_train.shape)
print(X_test_grade_cat.shape, y_test.shape)
```

```
print("="*100)


# CATEGORICAL FEATURES CLEANED_SUBJECT_CATEGORIES:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['project_subject_categories'],y_train))
t2 = pd.DataFrame(t1,columns = ['project_subject_categories','project_is_approved'])
t3 = pd.DataFrame(X_test['project_subject_categories'],
                  columns = ['project_subject_categories'])

X_train_categories = response_encoder(t2)
X_test_categories = response_encoder_test(t3,t2)
print("(F) Shape After vectorization of project Categories :")
print(X_train_categories.shape,y_train.shape)
print(X_test_categories.shape,y_test.shape)
print("="*100)


# CATEGORICAL FEATURES CLEAN_SUBCATEGORIES:

#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['project_subject_subcategories'],y_train))
t2 = pd.DataFrame(t1,columns = ['project_subject_subcategories','project_is_approved'])
t3 = pd.DataFrame(X_test['project_subject_subcategories'],
                  columns = ['project_subject_subcategories'])

X_train_subcategories = response_encoder(t2)
X_test_subcategories = response_encoder_test(t3,t2)

print("(G) Shape After vectorization of project Subcategories :")
print(X_train_subcategories.shape, y_train.shape)
print(X_test_subcategories.shape, y_test.shape)
```

SENTIMENT SCORES (ESSAY FEATURE):

In [13]:
```
# Define a function and call the function:
def sentiment_scores(lst):
    neg,neu,pos,compound = [],[],[],[]   # create empty lists to store scores
    for sent in lst:
        sentiment_dict = sia.polarity_scores(sent)
        neg.append(sentiment_dict['neg'])
```

```
            neu.append(sentiment_dict['neu'])
            pos.append(sentiment_dict['pos'])
            compound.append(sentiment_dict['compound'])
        negative = pd.Series(neg)
        neutral = pd.Series(neu)
        positive = pd.Series(pos)
        compound = pd.Series(compound)
        features = {'Negative':negative,"Neutral":neutral,"Positive":positive,"Compound":compound}
        result = pd.DataFrame(features)
        return result # return the scores dataframe.

sia = SentimentIntensityAnalyzer() # initialization

# create lists to store essay values:-
lst_xtrain = X_train['preprocessed_essay'].values
lst_xtest = X_test['preprocessed_essay'].values

# pass the lists to sentiment_scores() and return the scores as dataframe:-
df_Xtrain = sentiment_scores(lst_xtrain)
df_Xtest = sentiment_scores(lst_xtest)
print(df_Xtrain.head())
```

```
   Negative  Neutral  Positive  Compound
0     0.051    0.687     0.262    0.9909
1     0.050    0.777     0.174    0.9434
2     0.018    0.558     0.425    0.9937
3     0.060    0.725     0.215    0.9648
4     0.017    0.833     0.150    0.9501
```

CONCATENATING FEATURES FOR SET (1) :

In [14]:
```
# convert pandas dataframe to numpy arrays for fast computation:

X_Train_state_school = X_train_state_school.to_numpy()
X_Test_state_school = X_test_state_school.to_numpy()

X_Train_prefix_teacher = X_train_prefix_teacher.to_numpy()
X_Test_prefix_teacher = X_test_prefix_teacher.to_numpy()

X_Train_grade_cat = X_train_grade_cat.to_numpy()
X_Test_grade_cat = X_test_grade_cat.to_numpy()
```

```
X_Train_categories = X_train_categories.to_numpy()
X_Test_categories = X_test_categories.to_numpy()

X_Train_subcategories = X_train_subcategories.to_numpy()
X_Test_subcategories = X_test_subcategories.to_numpy()

sentimentscores_Xtrain = df_Xtrain.to_numpy()
sentimentscores_Xtest  = df_Xtest.to_numpy()

# concatenate all the features :
from scipy.sparse import hstack

# hstack() helps in concatenating "n" number of array like shapes into one dataframe.
# we store the concatenated outcome in a csr matrix format.

train_X = hstack((X_train_essay_tfidf,X_train_title_tfidf,X_Train_state_school,
                  X_Train_prefix_teacher,X_Train_grade_cat,
                 X_Train_categories,X_Train_subcategories,
                 X_train_price_norm,X_train_previous_norm,
                 sentimentscores_Xtrain)).tocsr()

test_X = hstack((X_test_essay_tfidf,X_test_title_tfidf,X_Test_state_school,
                  X_Test_prefix_teacher,X_Test_grade_cat,
                 X_Test_categories,X_Test_subcategories,
                 X_test_price_norm,X_test_previous_norm,
                sentimentscores_Xtest)).tocsr()

print("(H) Final Data matrix :")

print(train_X.shape, y_train.shape)#we totally have 35k rows & 8522 columns in train data

print(test_X.shape, y_test.shape)#we totally have 15k rows & 8522 columns in test data
```

```
(H) Final Data matrix :
(35000, 8522) (35000,)
(15000, 8522) (15000,)
```

Appling Models on different kind of featurization as mentioned in the instructions

Apply GBDT on different kind of featurization as mentioned in the instructions

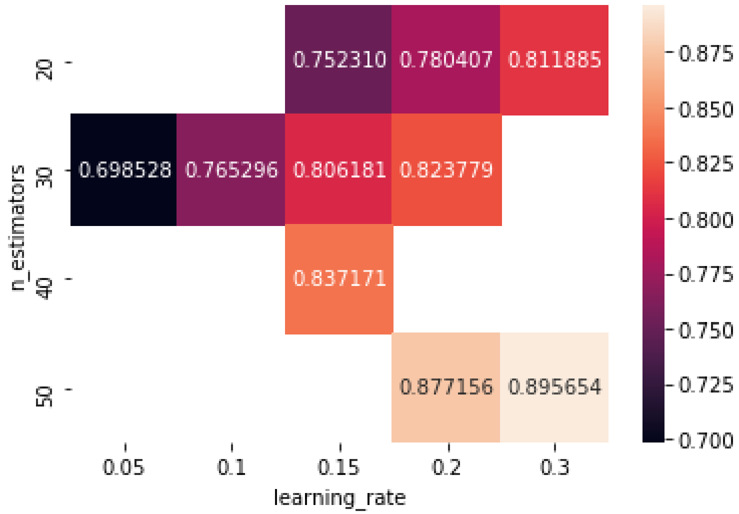
For Every model that you work on make sure you do the step 2 and step 3 of instrucations

APPLY XGBOOST FOR SET (1) FEATURES:-

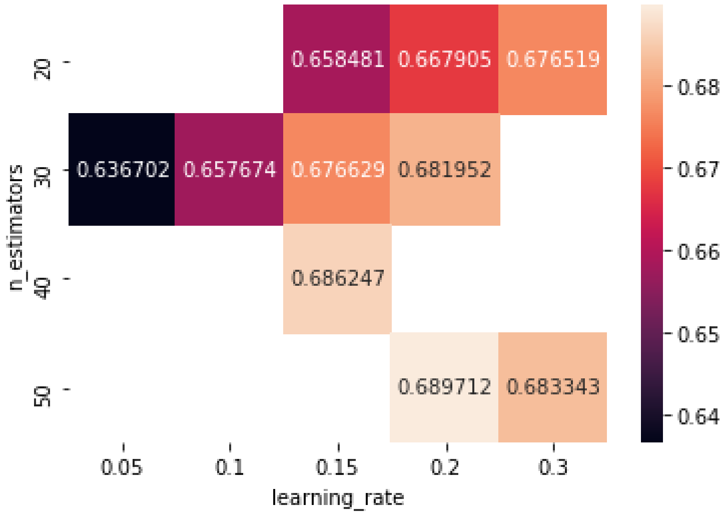
In [19]:
```
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV
from time import time

start = time()

# fit the model to the training data using randomsearchCv:

model = XGBClassifier(max_depth = 5,seed = 8,scale_pos_weight = 5)
param = {'learning_rate':[0.05,0.15,0.1,0.2,0.3],
         'n_estimators' : [20,30,40,50]}

# use "ROC_AUC" as a scoring and CV = 3
clf = RandomizedSearchCV(model,param,cv=3,scoring='roc_auc',return_train_score = True)

clf.fit(train_X,y_train)

print("The best paramters from randomsearchCv :",clf.best_params_)
print('\n')

# make a dataframe out of cv_results:
cv_results = pd.DataFrame.from_dict(clf.cv_results_)


# obtain mean train,Cv scores and their corresponding hyperparameters:
train_auc = cv_results['mean_train_score']
cv_auc = cv_results['mean_test_score']
lr =  cv_results['param_learning_rate']
no_of_estimators = cv_results['param_n_estimators']

# plot the Hyperparameters vs TRAIN_AUC plot (heatmap) for train data:

df1 = pd.DataFrame({'AUC_score':list(train_auc),
                   'learning_rate':list(lr),
                   'n_estimators':list(no_of_estimators)})

result = df1.pivot("n_estimators","learning_rate","AUC_score")
```

```
print("HEATMAP FOR TRAIN_AUC VS HYPERPARAMETERS :-")
sns.heatmap(result,annot=True,fmt="f")
plt.show()
print('\n')

# plot the Hyperparameters vs CV_AUC plot (heatmap) for CV data:

df2 = pd.DataFrame({'AUC_score':list(cv_auc),
                    'learning_rate':list(lr),
                    'n_estimators':list(no_of_estimators)})

result = df2.pivot("n_estimators","learning_rate","AUC_score")
print("HEATMAP FOR CV_AUC VS HYPERPARAMETERS :-")
sns.heatmap(result,annot=True,fmt="f")
plt.show()

end = time()
training_time = end - start
print("training & validation time: %0.2fs" % training_time)
```

```
The best paramters from randomsearchCv : {'n_estimators': 50, 'learning_rate': 0.2}


HEATMAP FOR TRAIN_AUC VS HYPERPARAMETERS :-
```




```
HEATMAP FOR CV_AUC VS HYPERPARAMETERS :-
```




```
training & validation time: 836.42s
```

NOTE:-

Even though randomized_searchCV returns best_params and best_score, we should pick the optimal hyperparameters only after comparing it with Train_auc to ensure optimal fitting.

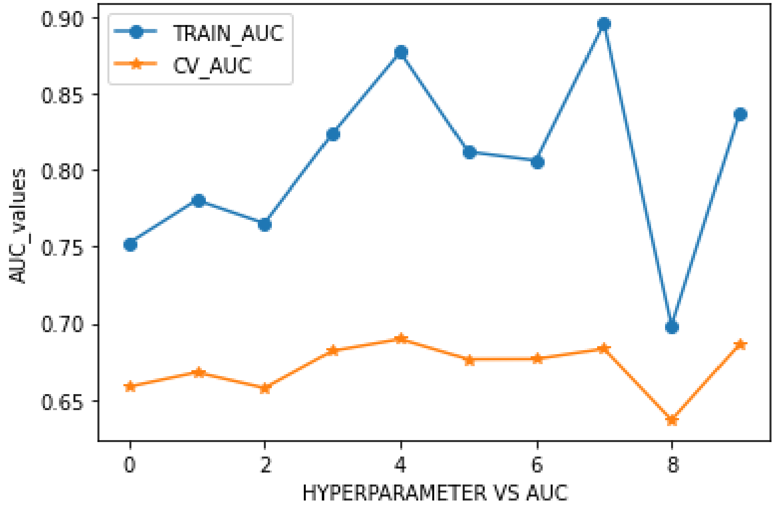
PICK THE OPTIMAL HYPERPARAMETERS:-

In [20]:

```
# find the difference between cv and train auc & get the optimal hyperparameters.
diff = train_auc - cv_auc
print("The train_auc :\n",tuple(train_auc))
print("The cv_auc :\n",tuple(cv_auc))
print("The difference :\n",tuple(diff))

# plot cv_auc vs train_auc curve for different hyperparameters:-
plt.plot(train_auc,marker = 'o',label = 'TRAIN_AUC')
plt.plot(cv_auc,marker = '*',label = 'CV_AUC')
plt.legend();plt.xlabel("HYPERPARAMETER VS AUC");plt.ylabel("AUC_values")
plt.show()
```

```
The train_auc :
 (0.7523101338433763, 0.7804067569544181, 0.7652963727168002, 0.8237785407577434, 0.8771560604120276, 0.8118849190502
86, 0.8061813128438211, 0.8956542753600427, 0.6985277104308141, 0.8371706263405598)
```

```
The cv_auc :
 (0.6584807051935391, 0.6679045404953271, 0.6576744778903224, 0.6819521009358168, 0.6897121003703758, 0.6765194388662
654, 0.6766285333905774, 0.6833433769615253, 0.6367023873858426, 0.6862470803084112)
The difference :
 (0.09382942864983723, 0.11250221645909098, 0.10762189482647777, 0.14182643982192655, 0.18744396004165187, 0.13536548
018402061, 0.12955277945324373, 0.21231089839851747, 0.06182532304497146, 0.15092354603214864)
```



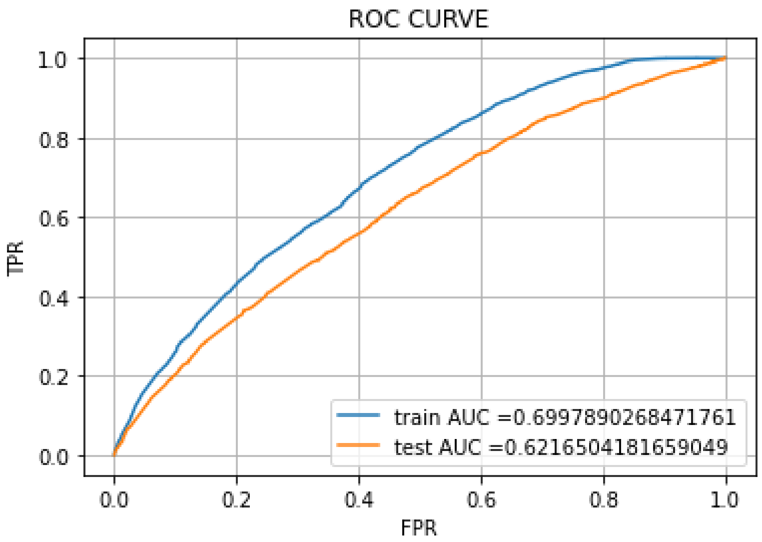
OBSERVATION:-

As per the values in the HEATMAP & the (TRAIN VS CV CURVE),we can clearly see that the difference between CV_AUC & TRAIN_AUC is small at n_estimator = 30,learning_rate = 0.05 and also greater than 60%. Hence, lets select these hyperparameters for test data and interpret the results.

TRAIN VS TEST AUC:-

In [36]:
```
from sklearn.metrics import roc_curve, auc # -->necessary libraries
from time import time
start = time()

# Fit the classifier with the optimal parameters:
clf = XGBClassifier(max_depth = 5,learning_rate = 0.05,
                    n_estimators = 30,seed = 8,scale_pos_weight = 5)

clf.fit(train_X,y_train)
```

```
# predict for train and test data :
y_pred_train = clf.predict_proba(train_X)
y_pred_test = clf.predict_proba(test_X)

# compute TPR,FPR values to construt ROC curve:
train_fpr,train_tpr,tr_thresholds = roc_curve(y_train,y_pred_train[:,1])
test_fpr,test_tpr,te_thresholds = roc_curve(y_test,y_pred_test[:,1])

#plot the ROC with Train AUC and Test AUC:
plt.plot(train_fpr,train_tpr, label="train AUC =" + str(auc(train_fpr, train_tpr)))
plt.plot(test_fpr,test_tpr, label="test AUC =" + str(auc(test_fpr, test_tpr)))
plt.legend()
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.title("ROC CURVE")
plt.grid()
plt.show()

end = time()
time_evaluation = end - start
print("training with best params & testing time: %0.2fs" % time_evaluation)
```




```
training with best params & testing time: 35.01s
```

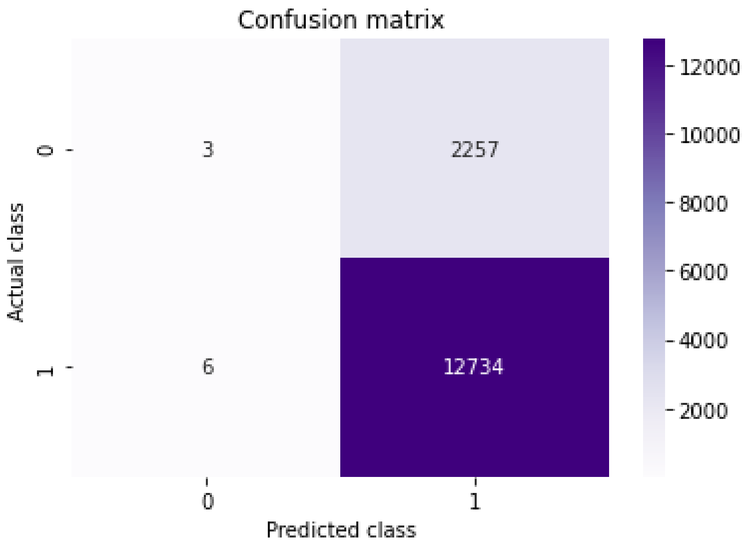
OBSERVATION:-

Hence the above test_AUC guarantee that for an unseen data in the future, the model can predict the instance correctly with atmost 62% confidence.

CONFUSION MATRIX FOR TEST DATA:-

In [37]:
```
# CONFUSION MATRIX :
from sklearn.metrics import confusion_matrix
y_pred_test = clf.predict(test_X)
confusion_mat = confusion_matrix(y_test,y_pred_test)
print(confusion_mat)

# Represent confusion matrix as a heatmap:
cm=np.array([[3,2257],[6,12734]])
sns.heatmap(cm,annot=True,fmt="d",cmap='Purples')
plt.xlabel("Predicted class")
plt.ylabel("Actual class")
plt.title("Confusion matrix")
plt.show()
```

```
[[    3  2257]
 [    6 12734]]
```



OBSERVATION:-

Here, we can clearly interpret that the dominant class is badly impacting the minority class points where most of the points are being predicted to be class (1)[project is approved].

SET (2):-

NOTE: TO HANDLE TIME COMPLEXITY OF W2V , WE USE ONLY 10K POINTS

LOADING DATA:-

In [145…

```
# necessary libraries
import pandas as pd
data = pd.read_csv('final_preprocessed.csv')
data = data.iloc[5000:15000,0:12]

print("The shape of the data : ",data.shape)
data.reset_index(inplace = True)

y = data["project_is_approved"].values # returns a numpy nd array--> Target variables
X = data.drop(["index","project_is_approved"],axis = 1) # Creates a dataframe X-->Input variables
X.head(3)
```

```
The shape of the data :  (10000, 11)
```

Out[145…

	id	teacher_prefix	school_state	project_grade_category	project_subject_categories	project_subject_subcategories	teacher_number_of_
0	p019015	mrs	ok	grades_prek_2	math_science	appliedsciences_mathematics	
1	p244340	mrs	nv	grades_prek_2	math_science_literacy_language	health_lifescience_literature_writing	
2	p038563	ms	mn	grades_prek_2	literacy_language	esl	

SPLITTING INTO Train-Test data (stratified sampling) :-

In [146…

```
# Split data into train and test set:(stratified sampling)

from sklearn.model_selection import train_test_split # necessary library

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.30,
                                               stratify = y,random_state=6)
```

Make Data model ready: ENCODING ESSAY & PROJECT_TITLE FEATURES (W2V):-

LOAD GLOVE VECTORS:-

In [186…

```
#please use below code to load glove vectors
with open('glove_vectors', 'rb') as f:
    glovemodel = pickle.load(f)
    glove_words = set(glovemodel.keys())

#check dim of word vector
len(glovemodel["student"])
```

Out[186… 300

FOR TRAIN & TEST DATA [ESSAY]:-

In [187…

```
# first lets create a tfidf model:

model = TfidfVectorizer(min_df= 10,ngram_range=(1,2),max_features=7500)
model.fit(X_train['preprocessed_essay'].values)

# 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_)))

#####################################################################################
#####################################################################################

# compute TFIDF word2vec for each essay in train data
import time
st = time.time()
```

```
tfidf_feat = model.get_feature_names() # tfidf words/col-names


tfidf_essay_vec_train = []; # the tfidf-w2v for each sentence/review is stored in this list
row=0; #--> To store number of iterations
list_of_sent = X_train['preprocessed_essay'].values

for sent in tqdm(list_of_sent):
    sent_vec = np.zeros(300)
    weight_sum =0;                                  # To store sum of tfidf values.
    for word in sent:                               # for each word in a review/sentence
        if word in glove_words and word in tfidf_feat:
            vec = glovemodel[word]
            tf_idf = dictionary[word]*(sent.count(word)/len(sent))

            sent_vec += (vec * tf_idf)   # (vector) x (tfidf value)

            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum

    tfidf_essay_vec_train.append(sent_vec) # first sentence transformed to a vector and appended
    row += 1

print("(A) Number of iterations to train w2v on train data :",row) # To know the number of iterations.
print("(B) Total essays transformed :",len(tfidf_essay_vec_train)) # list of each review vector having 50 as dimensio
print("(C) Dimension of each essay(train) :",len(tfidf_essay_vec_train[0])) # each review has been transformed to 50
et = time.time()
print("(D) Time taken to train tfidf-w2v on train data :",et-st) # # time taken by the program.

#####################################################################################
#####################################################################################

# compute TFIDF word2vec for each essay in test data
import time
st = time.time()

# TF-IDF weighted Word2Vec
tfidf_feat = model.get_feature_names() # tfidf words/col-names
tfidf_matrix = model.transform(X_test['preprocessed_essay'].values)# returns tfidf weighted values

df = pd.DataFrame(tfidf_matrix.toarray(),columns = tfidf_feat)
```

```
tfidf_essay_vec_test = []; # the tfidf-w2v for each sentence/review is stored in this list
row=0; #--> To store number of iterations

list_of_sent = X_test['preprocessed_essay'].values

for idx,sent in tqdm(enumerate(list_of_sent)):
    sent_vec = np.zeros(300)
    weight_sum =0; # To store sum of tfidf values.
    for w,word in enumerate(sent): # for each word in a review/sentence
        if word in glove_words and word in tfidf_feat:
            vec = glovemodel[word]
            tf_idf = df.iloc[idx,w]      # grab the value of tfidf from df.
            sent_vec += (vec * tf_idf)   # (vector) x (tfidf value).
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum

    tfidf_essay_vec_test.append(sent_vec) # first sentence transformed to a vector and appended
    row += 1

print('\n')
print("(A) Number of iterations for w2v on test data :",row) # To know the number of iterations.
print("(B) Total essays transformed :",len(tfidf_essay_vec_test)) # list of each review vector having 50 as dimensior
print("(C) Dimension of each essay(test) :",len(tfidf_essay_vec_test[0])) # each review has been transformed to 50 di
et = time.time()
print("(D) Time taken to train tfidf-w2v on test data :",et-st) # # time taken by the program.
```

```
100%|████████████████████████████████████████████████████████████| 7000/7000 [19:04<00:00,  6.12it/
s]
(A) Number of iterations to train w2v on train data : 7000
(B) Total essays transformed : 7000
(C) Dimension of each essay(train) : 300
(D) Time taken to train tfidf-w2v on train data : 1144.1195266246796
3000it [08:33,  5.84it/s]

(A) Number of iterations for w2v on test data : 3000
(B) Total essays transformed : 3000
(C) Dimension of each essay(test) : 300
(D) Time taken to train tfidf-w2v on test data : 514.9873979091644
```

FOR TRAIN & TEST DATA [PROJECT_TITLE]:-

In [188…

```
# first lets create a tfidf model:

model = TfidfVectorizer(min_df= 15,ngram_range=(1,3),max_features=2500)
model.fit(X_train['preprocessed_title'].values.astype('U'))

# 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_)))

#####################################################################################
#####################################################################################

# compute TFIDF word2vec for each essay in train data
import time
st = time.time()

tfidf_feat = model.get_feature_names() # tfidf words/col-names


tfidf_title_vec_train = []; # the tfidf-w2v for each sentence/review is stored in this list
row=0; #--> To store number of iterations
list_of_sent = X_train['preprocessed_title'].values.astype('U')

for sent in tqdm(list_of_sent):
    sent_vec = np.zeros(300)
    weight_sum =0;                                  # To store sum of tfidf values.
    for word in sent:                               # for each word in a review/sentence
        if word in glove_words and word in tfidf_feat:
            vec = glovemodel[word]
            tf_idf = dictionary[word]*(sent.count(word)/len(sent))

            sent_vec += (vec * tf_idf)   # (vector) x (tfidf value)

            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum

    tfidf_title_vec_train.append(sent_vec) # first sentence transformed to a vector and appended
    row += 1

print("(A) Number of iterations to train w2v on train data :",row) # To know the number of iterations.
print("(B) Total titles transformed :",len(tfidf_title_vec_train)) # list of each review vector having 50 as dimensio
```

```
print("(C) Dimension of each title(train) :",len(tfidf_title_vec_train[0])) # each review has been transformed to 50
et = time.time()
print("(D) Time taken to train tfidf-w2v on train data :",et-st) # # time taken by the program.

##########################################################################################
##########################################################################################

# compute TFIDF word2vec for each essay in test data
import time
st = time.time()

# TF-IDF weighted Word2Vec
tfidf_feat = model.get_feature_names() # tfidf words/col-names
tfidf_matrix = model.transform(X_test['preprocessed_title'].values.astype('U'))# returns tfidf weighted values

df = pd.DataFrame(tfidf_matrix.toarray(),columns = tfidf_feat)

tfidf_title_vec_test = []; # the tfidf-w2v for each sentence/review is stored in this list
row=0; #--> To store number of iterations

list_of_sent = X_test['preprocessed_title'].values.astype('U')

for idx,sent in tqdm(enumerate(list_of_sent)):
    sent_vec = np.zeros(300)
    weight_sum =0; # To store sum of tfidf values.
    for w,word in enumerate(sent): # for each word in a review/sentence
        if word in glove_words and word in tfidf_feat:
            vec = glovemodel[word]
            tf_idf = df.iloc[idx,w]      # grab the value of tfidf from df.
            sent_vec += (vec * tf_idf)   # (vector) x (tfidf value).
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum

    tfidf_title_vec_test.append(sent_vec) # first sentence transformed to a vector and appended
    row += 1

print('\n')
print("(A) Number of iterations for w2v on test data :",row) # To know the number of iterations.
print("(B) Total titles transformed :",len(tfidf_title_vec_test)) # list of each review vector having 50 as dimension
print("(C) Dimension of each title(test) :",len(tfidf_title_vec_test[0])) # each review has been transformed to 50 di
```

```
et = time.time()
print("(D) Time taken to train tfidf-w2v on test data :",et-st) # # time taken by the program.
```

```
100%|████████████████████████████████████████████████████████████████████████████| 7000/7000 [00:01<00:00, 6749.31it/
s]
0it [00:00, ?it/s]
(A) Number of iterations to train w2v on train data : 7000
(B) Total titles transformed : 7000
(C) Dimension of each title(train) : 300
(D) Time taken to train tfidf-w2v on train data : 1.0510706901550293
3000it [00:00, 5539.30it/s]

(A) Number of iterations for w2v on test data : 3000
(B) Total titles transformed : 3000
(C) Dimension of each title(test) : 300
(D) Time taken to train tfidf-w2v on test data : 0.6452767848968506
```

Make data model ready: ENCODING NUMERICAL & CATEGORICAL FEATURES:-

In [189…

```
import numpy as np
# Normalizing: map all the values to range of (0,1) -- > [PRICE]

from sklearn.preprocessing import Normalizer
normalizer = Normalizer()

# fit and transform the train and test data:
# reshape of the data to single row allows the normalizer() to fit & transform
normalizer.fit(X_train['price'].values.reshape(1,-1))
X_train_price_norm = normalizer.transform(X_train['price'].values.reshape(1,-1))
X_test_price_norm = normalizer.transform(X_test['price'].values.reshape(1,-1))

print("(A) Shape After Nomalization of feature Price :")
print(X_train_price_norm.transpose().shape, y_train.shape)
print(X_test_price_norm.transpose().shape, y_test.shape)

# assign to another variable for readability purpose :
X_train_price_norm = X_train_price_norm.transpose()
X_test_price_norm  = X_test_price_norm.transpose()
print("="*100)
```

```
#Normalizing: map the values to range of (0,1)-- >[TEACHER_NO_OF_PREVIOUSLY_POSTED_PROJECTS]

from sklearn.preprocessing import Normalizer
normalizer = Normalizer()

# fit and transform the train and test data:
# reshape of the data to single row allows the normalizer() to fit & transform
normalizer.fit(X_train['teacher_number_of_previously_posted_projects'].
               values.reshape(1,-1))
X_train_previous_norm = normalizer.transform(X_train['teacher_number_of_previously_posted_projects'].
                                             values.reshape(1,-1))
X_test_previous_norm = normalizer.transform(X_test['teacher_number_of_previously_posted_projects'].
                                            values.reshape(1,-1))

print("(B) Shape After Normalization of Number of previously posted projects :")
print(X_train_previous_norm.transpose().shape, y_train.shape)
print(X_test_previous_norm.transpose().shape, y_test.shape)

# assign to another variable for readability purpose :
X_train_previous_norm = X_train_previous_norm.transpose()
X_test_previous_norm = X_test_previous_norm.transpose()
print("="*100)

#################################################################################
#################################################################################


# Response coding for categorical features:

# CATEGORICAL FEATURE--> SCHOOL STATE:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['school_state'],y_train))
t2 = pd.DataFrame(t1,columns = ['school_state','project_is_approved'])
t3 = pd.DataFrame(X_test['school_state'],columns = ['school_state'])

X_train_state_school = response_encoder(t2)
X_test_state_school = response_encoder_test(t3,t2)

print("(C) Shape After vectorization of school state :")
print(X_train_state_school.shape, y_train.shape)
print(X_test_state_school.shape, y_test.shape)
print("="*100)
```

```
# CATEGORICAL FEATURES TEACHER_PREFIX:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['teacher_prefix'],y_train))
t2 = pd.DataFrame(t1,columns = ['teacher_prefix','project_is_approved'])
t3 = pd.DataFrame(X_test['teacher_prefix'],columns = ['teacher_prefix'])

t4 = t2.fillna('unknown',inplace=False)
t5 = t3.fillna('unknown',inplace=False)
X_train_prefix_teacher = response_encoder(t4)
X_test_prefix_teacher = response_encoder_test(t5,t4)

print("(D) Shape After vectorization of teacher_prefix :")
print(X_train_prefix_teacher.shape, y_train.shape)
print(X_test_prefix_teacher.shape, y_test.shape)
print("="*100)




# CATEGORICAL FEATURES project_grade_category:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['project_grade_category'],y_train))
t2 = pd.DataFrame(t1,columns = ['project_grade_category','project_is_approved'])
t3 = pd.DataFrame(X_test['project_grade_category'],columns = ['project_grade_category'])

X_train_grade_cat = response_encoder(t2)
X_test_grade_cat = response_encoder_test(t3,t2)

print("(E) Shape After vectorization of project_grade_Category :")
print(X_train_grade_cat.shape, y_train.shape)
print(X_test_grade_cat.shape, y_test.shape)
print("="*100)


# CATEGORICAL FEATURES CLEANED_SUBJECT_CATEGORIES:
#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['project_subject_categories'],y_train))
t2 = pd.DataFrame(t1,columns = ['project_subject_categories','project_is_approved'])
t3 = pd.DataFrame(X_test['project_subject_categories'],
                  columns = ['project_subject_categories'])
```

```
X_train_categories = response_encoder(t2)
X_test_categories = response_encoder_test(t3,t2)
print("(F) Shape After vectorization of project Categories :")
print(X_train_categories.shape,y_train.shape)
print(X_test_categories.shape,y_test.shape)
print("="*100)


# CATEGORICAL FEATURES CLEAN_SUBCATEGORIES:

#create a train dataframe and test dataframe with desired features to encode:
t1 = np.column_stack((X_train['project_subject_subcategories'],y_train))
t2 = pd.DataFrame(t1,columns = ['project_subject_subcategories','project_is_approved'])
t3 = pd.DataFrame(X_test['project_subject_subcategories'],
                  columns = ['project_subject_subcategories'])

X_train_subcategories = response_encoder(t2)
X_test_subcategories = response_encoder_test(t3,t2)

print("(G) Shape After vectorization of project Subcategories :")
print(X_train_subcategories.shape, y_train.shape)
print(X_test_subcategories.shape, y_test.shape)
```

```
(A) Shape After Nomalization of feature Price :
(7000, 1) (7000,)
(3000, 1) (3000,)
====================================================================================================
(B) Shape After Normalization of Number of previously posted projects :
(7000, 1) (7000,)
(3000, 1) (3000,)
====================================================================================================
(C) Shape After vectorization of school state :
(7000, 2) (7000,)
(3000, 2) (3000,)
====================================================================================================
(D) Shape After vectorization of teacher_prefix :
(7000, 2) (7000,)
(3000, 2) (3000,)
====================================================================================================
(E) Shape After vectorization of project_grade_Category :
(7000, 2) (7000,)
(3000, 2) (3000,)
====================================================================================================
```

```
(F) Shape After vectorization of project Categories :
(7000, 2) (7000,)
(3000, 2) (3000,)
====================================================================================================
(G) Shape After vectorization of project Subcategories :
(7000, 2) (7000,)
(3000, 2) (3000,)
```

CONCATENATING FEATURES FOR SET (2):-

In [191…

```
# convert pandas dataframe to numpy arrays for fast computation:

X_Train_state_school = X_train_state_school.to_numpy()
X_Test_state_school = X_test_state_school.to_numpy()

X_Train_prefix_teacher = X_train_prefix_teacher.to_numpy()
X_Test_prefix_teacher = X_test_prefix_teacher.to_numpy()

X_Train_grade_cat = X_train_grade_cat.to_numpy()
X_Test_grade_cat = X_test_grade_cat.to_numpy()

X_Train_categories = X_train_categories.to_numpy()
X_Test_categories = X_test_categories.to_numpy()

X_Train_subcategories = X_train_subcategories.to_numpy()
X_Test_subcategories = X_test_subcategories.to_numpy()


# concatenate all the features :

# np.hstack() helps in concatenating "n" number of array like shapes into one array.

tfidf_essay_w2v_train = np.asarray(tfidf_essay_vec_train)
tfidf_title_w2v_train = np.asarray(tfidf_title_vec_train)


train_X = np.hstack((tfidf_essay_w2v_train,tfidf_title_w2v_train,
                     X_Train_state_school,X_Train_prefix_teacher,X_Train_grade_cat,
                 X_Train_categories,X_Train_subcategories,
                 X_train_price_norm,X_train_previous_norm))

tfidf_essay_w2v_test= np.asarray(tfidf_essay_vec_test)
```

```
tfidf_title_w2v_test= np.asarray(tfidf_title_vec_test)

test_X = np.hstack((tfidf_essay_w2v_test,tfidf_title_w2v_test,
                    X_Test_state_school,X_Test_prefix_teacher,X_Test_grade_cat,
                 X_Test_categories,X_Test_subcategories,
                 X_test_price_norm,X_test_previous_norm))

print("(H) Final Data matrix :")

print(train_X.shape, y_train.shape)#we totally have 7k rows & 612columns in train data

print(test_X.shape, y_test.shape)#we totally have 3k rows & 612 columns in test data
```

```
(H) Final Data matrix :
(7000, 612) (7000,)
(3000, 612) (3000,)
```

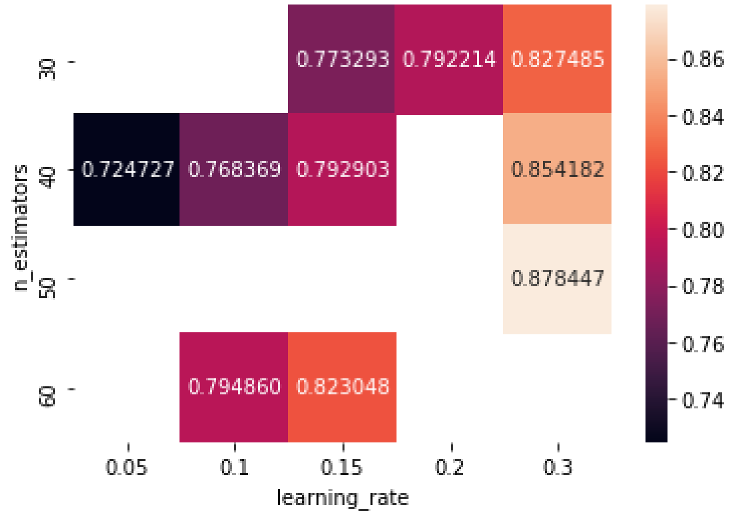
APPLY XGBOOST For SET (2) Features :-

In [201…

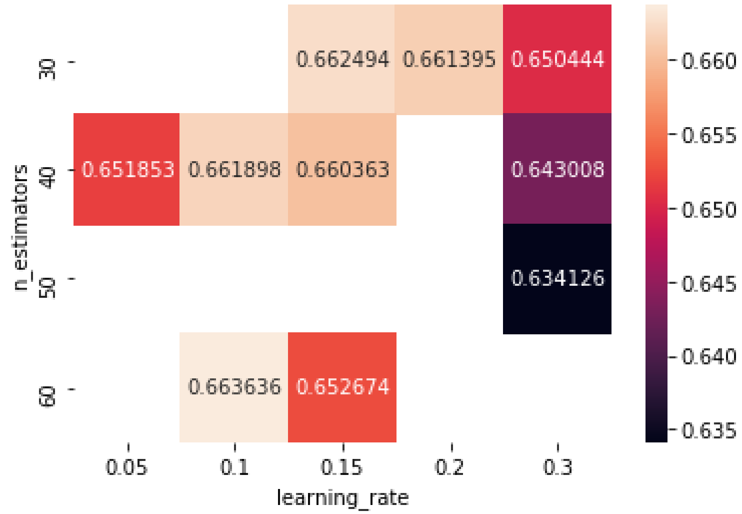
```
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV
from time import time

start = time()

# fit the model to the training data using randomsearchCv:

model = XGBClassifier(max_depth = 5,seed = 7,scale_pos_weight = 5)
param = {'learning_rate':[0.05,0.15,0.1,0.2,0.3],
         'n_estimators' : [30,40,50,60]}

# use "ROC_AUC" as a scoring and CV = 3
clf = RandomizedSearchCV(model,param,cv=3,scoring='roc_auc',return_train_score = True)

clf.fit(train_X,y_train)

print("The best paramters from randomsearchCv :",clf.best_params_)
print('\n')

# make a dataframe out of cv_results:
cv_results = pd.DataFrame.from_dict(clf.cv_results_)
```

```python
# obtain mean train,Cv scores and their corresponding hyperparameters:
train_auc = cv_results['mean_train_score']
cv_auc = cv_results['mean_test_score']
lr =  cv_results['param_learning_rate']
no_of_estimators = cv_results['param_n_estimators']


# plot the Hyperparameters vs TRAIN_AUC plot (heatmap) for train data:
df1 = pd.DataFrame({'AUC_score':list(train_auc),
                   'learning_rate':list(lr),
                   'n_estimators':list(no_of_estimators)})

result = df1.pivot("n_estimators","learning_rate","AUC_score")
print("HEATMAP FOR TRAIN_AUC VS HYPERPARAMETERS :-")
sns.heatmap(result,annot=True,fmt="f")
plt.show()
print('\n')

# plot the Hyperparameters vs CV_AUC plot (heatmap) for CV data:

df2 = pd.DataFrame({'AUC_score':list(cv_auc),
                   'learning_rate':list(lr),
                   'n_estimators':list(no_of_estimators)})

result = df2.pivot("n_estimators","learning_rate","AUC_score")
print("HEATMAP FOR CV_AUC VS HYPERPARAMETERS :-")
sns.heatmap(result,annot=True,fmt="f")
plt.show()

end = time()
training_time = end - start
print("training & validation time: %0.2fs" % training_time)
```

```
The best paramters from randomsearchCv : {'n_estimators': 60, 'learning_rate': 0.1}


HEATMAP FOR TRAIN_AUC VS HYPERPARAMETERS :-
```




```
HEATMAP FOR CV_AUC VS HYPERPARAMETERS :-
```




```
training & validation time: 166.14s
```

NOTE:-

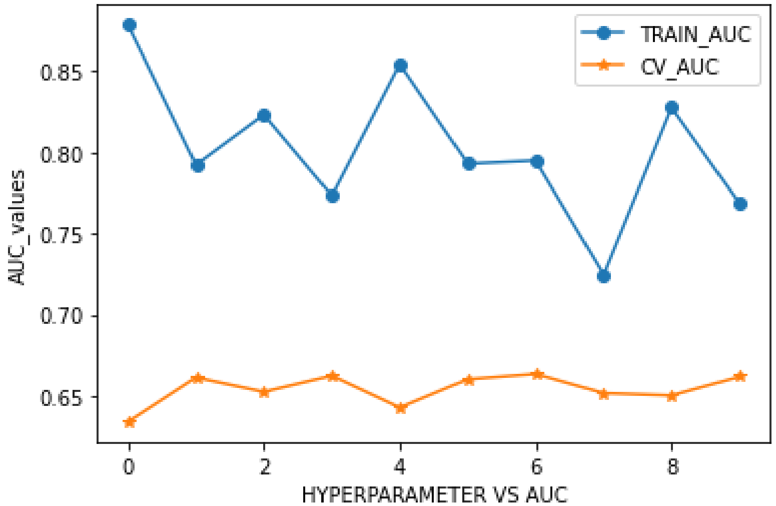
Even though randomized_searchCV returns best_params and best_score, we should pick the optimal hyperparameters only after comparing it with Train_auc to ensure optimal fitting.

PICK OPTIMAL HYPERPARAMETERS:-

In [202…

```
# find the difference btw mean scores of cv_auc & train_auc:-
diff = train_auc - cv_auc
print("The train_auc :\n",tuple(train_auc))
print("The cv_auc :\n",tuple(cv_auc))
print("The difference :\n",tuple(diff))

# plot cv_train auc for finding optimal hyperparameters:-
plt.plot(train_auc,marker = 'o',label = 'TRAIN_AUC')
plt.plot(cv_auc,marker = '*',label = 'CV_AUC')
plt.legend();plt.xlabel("HYPERPARAMETER VS AUC");plt.ylabel("AUC_values")
plt.show()
```

```
The train_auc :
 (0.8784474640566664, 0.7922143593477348, 0.8230477042340686, 0.7732929060804898, 0.8541816871195413, 0.7929030214314
344, 0.7948597374298917, 0.724726621603466, 0.8274849671784329, 0.7683693336759329)
The cv_auc :
 (0.634126340845217, 0.6613949445737951, 0.6526739554073802, 0.6624938644777134, 0.6430083749122453, 0.66036278821375
29, 0.6636355094420961, 0.651852612867433, 0.6504436311065774, 0.6618981734803752)
The difference :
 (0.2443211232114494, 0.13081941477393966, 0.17037374882668843, 0.11079904160277643, 0.21117331220729607, 0.132540233
2176815, 0.13122422798779554, 0.072874008736033, 0.17704133607185546, 0.10647116019555769)
```



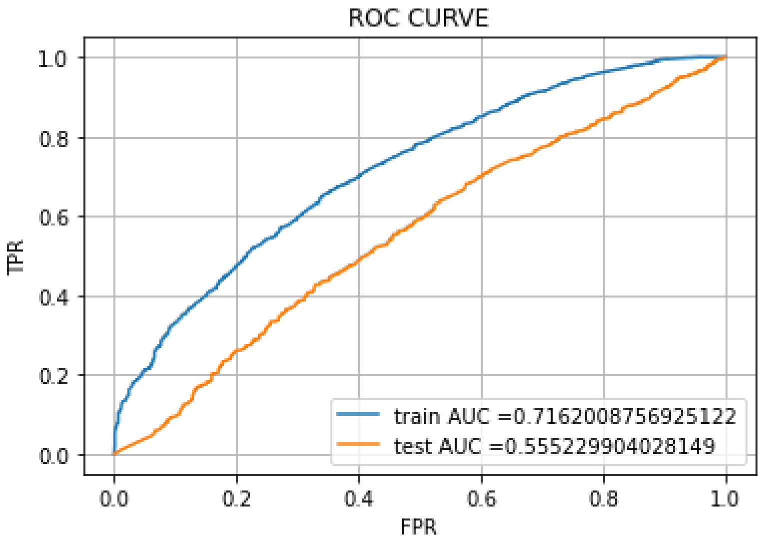
OBSERVATION:-

Here we can observe that the minimal gap between Cv_auc and Train_auc to be = 0.07 which occurs at a specific combination of hyperparameter i.e - - >n_estimators = 40 & learning_rate = 0.05 and also the both the auc value is typically greater than 65% ,thus lets interpret the model performance for the test data.

TRAIN VS TEST AUC:-

In [207…

```
from sklearn.metrics import roc_curve, auc # -->necessary libraries
from time import time
start = time()

# Fit the classifier with the optimal hyperparameters:
clf = XGBClassifier(max_depth = 5,learning_rate = 0.05,
                    n_estimators = 40,seed = 7,scale_pos_weight = 5)

clf.fit(train_X,y_train)

# predict for train and test data :
y_pred_train = clf.predict_proba(train_X)
y_pred_test = clf.predict_proba(test_X)

# compute TPR,FPR values to construct ROC curve:
train_fpr,train_tpr,tr_thresholds = roc_curve(y_train,y_pred_train[:,1])
test_fpr,test_tpr,te_thresholds = roc_curve(y_test,y_pred_test[:,1])

#plot the ROC with Train AUC and Test AUC:
plt.plot(train_fpr,train_tpr, label="train AUC =" + str(auc(train_fpr, train_tpr)))
plt.plot(test_fpr,test_tpr, label="test AUC =" + str(auc(test_fpr, test_tpr)))
plt.legend()
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.title("ROC CURVE")
plt.grid()
plt.show()

end = time()
time_evaluation = end - start
print("training with best params & testing time: %0.2fs" % time_evaluation)
```




```
training with best params & testing time: 6.10s
```

OBSERVATION:-

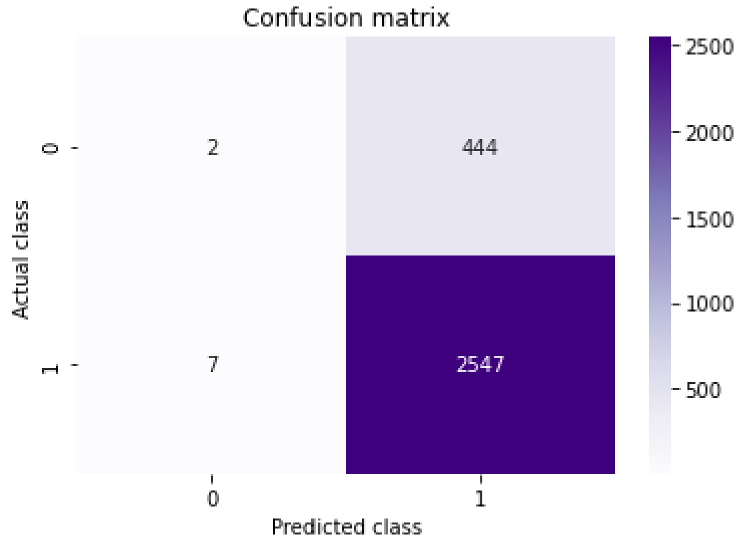
Thus, we can infer that the model can guarantee atmost 55% accurate prediction for an unseen data.

CONFUSION MATRIX FOR TEST DATA:-

In [209…

```
# CONFUSION MATRIX :
from sklearn.metrics import confusion_matrix
y_pred_test = clf.predict(test_X)
confusion_mat = confusion_matrix(y_test,y_pred_test)
print(confusion_mat)

# Represent confusion matrix as a heatmap:
cm=np.array([[2,444],[7,2547]])
sns.heatmap(cm,annot=True,fmt="d",cmap='Purples')
plt.xlabel("Predicted class")
plt.ylabel("Actual class")
plt.title("Confusion matrix")
plt.show()
```

```
[[   2  444]
 [   7 2547]]
```



OBSERVATION:-

We can clearly see that positive class points dominates with a high TPR & low TNR which is probably because of lack of points from minority class points.

3. Summary

as mentioned in the step 4 of instructions

In [210…

```
from prettytable import PrettyTable

# final results of the tasks:
# create a table with desired attributes:
summary = [["TFIDF","XGBOOST","(0.05[learning_rate,30[n_estimators])","0.62"],
           ["TFIDF-W2V","XGBOOST","(0.05[learning_rate,30[n_estimators])","0.55"]]

table = PrettyTable(["Vectorizer","Model","Hyperparameters","AUC"])

# add rows to the table:
```

```
for j in summary:
    table.add_row(j)

#print final summary of tasks:
print(table)
```

```
+------------+---------+---------------------------------------+------+
| Vectorizer |  Model  |            Hyperparameters            | AUC  |
+------------+---------+---------------------------------------+------+
|   TFIDF    | XGBOOST | (0.05[learning_rate,30[n_estimators]) | 0.62 |
| TFIDF-W2V  | XGBOOST | (0.05[learning_rate,30[n_estimators]) | 0.55 |
+------------+---------+---------------------------------------+------+
```