Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Machine Learning – I (DJ19DSC402)

AY: 2021-22

Experiment 7 (AdaBoost)

Name: Dev Patel **SAP ID: 60009200016**

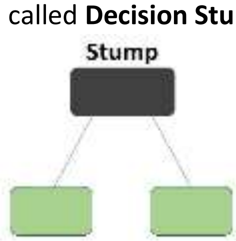
Batch: K/K1 **Date: 09/05/2022**

Aim: Evaluate the performance of boosting algorithm (AdaBoost) with different base learners and hyperparameter tuning.

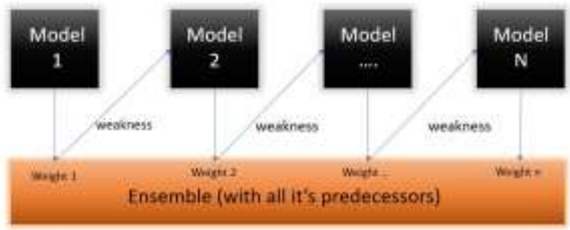
Theory:

Boosting algorithms improve the prediction power by **converting a number of weak learners to strong learners.** The principle behind boosting algorithms is first built a model on the training dataset, then a second model is built to rectify the errors present in the first model. This procedure is continued until and unless the errors are minimized, and the dataset is predicted correctly. Let's take an example to understand this, suppose you built a decision tree algorithm on the Titanic dataset and from there you get an accuracy of 80%. After this, you apply a different algorithm and check the accuracy and it comes out to be 75% for KNN and 70% for Linear Regression. The accuracy differs when we built a different model on the same dataset. But what if we use combinations of all these algorithms for making the final prediction? We'll get more accurate results by taking the average of results from these models. We can increase the prediction power in this way.

AdaBoost also called Adaptive Boosting is a technique in Machine Learning used as an Ensemble Method. The most common algorithm used with AdaBoost is decision trees with one level that means with Decision trees with only 1 split. These trees are also called **Decision Stumps.**



Algorithm builds a model and gives equal weights to all the data points. It then assigns higher weights to points that are wrongly classified. Now all the points which have higher weights are given more importance in the next model. It will keep training models until and unless a lower error is received.



Step 1 – The Image is shown below is the actual representation of our dataset. Since the target column is binary it is a classification problem. First of all, these data points will be assigned some weights. Initially, all the weights will be equal.

Row No.	Gender	Age	Income	Illness	Sample Weights
1	Male	41	40000	Yes	1/5
2	Male	54	30000	No	1/5
3	Female	42	25000	No	1/5
4	Female	40	60000	Yes	1/5
5	Male	46	50000	Yes	1/5

The formula to calculate the sample weights is:

$$w(x_i, y_i) = \frac{1}{N}, \; i = 1, 2, \ldots . n$$

Where N is the total number of datapoints. since we have 5 data points so the sample weights assigned will be 1/5.

Step 2 – We start by seeing how well "*Gender*" classifies the samples and will see how the variables (Age, Income) classifies the samples.

We'll create a decision stump for each of the features and then calculate the ***Gini Index*** of each tree. The tree with the lowest Gini Index will be our first stump.

Here in our dataset let's say ***Gender*** has the lowest Gini index so it will be our first stump.

Step 3 – Calculate the **"Amount of Say"** or **"Importance"** or **"Influence"** for this classifier in classifying the datapoints using this formula:

$$\frac{1}{2}\log\frac{1 - Total\ Error}{Total\ Error}$$

The total error is nothing, but the summation of all the sample weights of misclassified data points.

Here in our dataset let's assume there is 1 wrong output, so our total error will be 1/5, and alpha (performance of the stump) will be:

$$Performance\ of\ the\ stump = \frac{1}{2}\log_e\left(\frac{1-Total\ Error}{Total\ Error}\right)$$

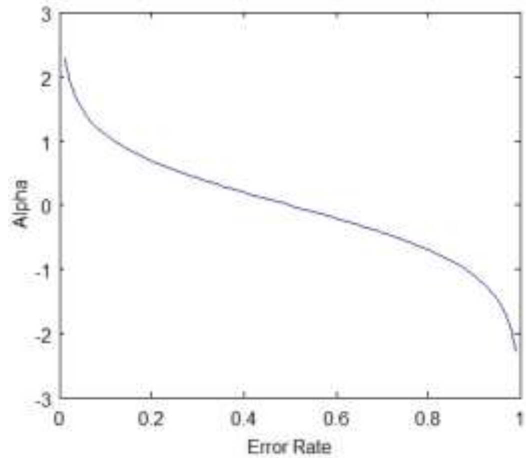
$$\alpha = \frac{1}{2}\log_e\left(\frac{1-\frac{1}{5}}{\frac{1}{5}}\right)$$

$$\alpha = \frac{1}{2}\log_e\left(\frac{0.8}{0.2}\right)$$

$$\alpha = \frac{1}{2}\log_e(4) = \frac{1}{2}*(1.38)$$

$$\alpha = 0.69$$

Note: Total error will always be between 0 and 1.
0 Indicates perfect stump and 1 indicates horrible stump.



From the graph above we can see that when there is no misclassification then we have no error (Total Error = 0), so the "amount of say (alpha)" will be a large number.
When the classifier predicts half right and half wrong then the Total Error = 0.5 and the importance (amount of say) of the classifier will be 0. If all the samples have been incorrectly classified then the error will be very high (approx. to 1) and hence our alpha value will be a negative integer.
Step 4 –We need to update the weights because if the same weights are applied to the next model, then the output received will be the same as what was received in the first model. The wrong predictions will be given more weight whereas the correct predictions weights will be decreased. Now when we build our next model after updating the weights, more preference will be given to the points with higher weights. After finding the importance of the classifier and total error we need to finally update the weights and for this, we use the following formula:

$$New\ sample\ weight = old\ weight * e^{\pm Amount\ of\ say\ (\alpha)}$$

The amount of say (alpha) will be ***negative*** when the sample is **correctly classified**.
The amount of say (alpha) will be ***positive*** when the sample is **miss-classified.**
There are four correctly classified samples and 1 wrong, here the ***sample weight*** of that datapoint is *1/5* and the ***amount of say/performance of the stump*** of ***Gender*** is *0.69*.

$$New\ sample\ weight = \frac{1}{5} * \exp(-0.69)$$

$$New\ sample\ weight = 0.2 * 0.502 = 0.1004$$

For *wrongly classified* samples the updated weights will be:

$$New\ sample\ weight = \frac{1}{5} * \exp(0.69)$$

$$New\ sample\ weight = 0.2 * 1.994 = 0.3988$$

Note: See the sign of alpha when I am putting the values, the **alpha is negative** when the data point is correctly classified, and this *decreases the sample weight* from 0.2 to 0.1004. It is **positive** when there is **misclassification**, and this will *increase the sample weight* from 0.2 to 0.3988

Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004
2	Male	54	30000	No	1/5	0.1004
3	Female	42	25000	No	1/5	0.1004
4	Female	40	60000	Yes	1/5	0.3988
5	Male	46	50000	Yes	1/5	0.1004

We know that the total sum of the sample weights must be equal to 1 but here if we sum up all the new sample weights, we will get 0.8004. To bring this sum equal to 1 we will normalize these weights by dividing all the weights by the total sum of updated weights that is 0.8004. So, after normalizing the sample weights we get this dataset and now the sum is equal to 1.

Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004/0.8004 =0.1254
2	Male	54	30000	No	1/5	0.1004/0.8004 =0.1254
3	Female	42	25000	No	1/5	0.1004/0.8004 =0.1254
4	[illegible]	[illegible]	[illegible]	[illegible]	1/5	[illegible]
5	Male	46	50000	Yes	1/5	0.1004/0.8004 =0.1254

Step 5 – Now we need to make a new dataset to see if the errors decreased or not. For this we will remove the "sample weights" and "new sample weights" column and then based on the "new sample weights" we will divide our data points into buckets.

Row No.	Gender	Age	Income	Illness	New Sample Weights	Buckets
1	Male	41	40000	Yes	0.1004/0.8004= 0.1254	0 to 0.1254
2	Male	54	30000	No	0.1004/0.8004= 0.1254	0.1254 to 0.2508
3	Female	42	25000	No	0.1004/0.8004= 0.1254	0.2508 to 0.3762
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
5	Male	46	50000	Yes	0.1004/0.8004= 0.1254	0.8744 to 0.9998

Step 6 – We are almost done, now what the algorithm does is selects random numbers from 0-1. Since incorrectly classified records have higher sample weights, the probability to select those records is very high. Suppose the 5 random numbers our algorithm take is 0.38,0.26,0.98,0.40,0.55.

Now we will see where these random numbers fall in the bucket and according to it, we'll make our new dataset shown below.

Row No.	Gender	Age	Income	Illness
1	Female	40	60000	Yes
2	Male	54	30000	No
3	Female	42	25000	No
4	Female	40	60000	Yes
5	Female	40	60000	Yes

This comes out to be our new dataset and we see the datapoint which was wrongly classified has been selected 3 times because it has a higher weight.

Step 9 – Now this act as our new dataset and we need to repeat all the above steps i.e.

1. Assign ***equal weights*** to all the datapoints
2. Find the stump that does the ***best job classifying*** the new collection of samples by finding their Gini Index and selecting the one with the lowest Gini index
3. Calculate the ***"Amount of Say"*** and ***"Total error"*** to update the previous sample weights.
4. Normalize the new sample weights.

Iterate through these steps until and unless a low training error is achieved.

Suppose with respect to our dataset we have constructed 3 decision trees (DT1, DT2, DT3) in a ***sequential manner.*** If we send our **test data** now it will pass through all the decision trees and finally, we will see which class has the majority, and based on that we will do predictions for our test dataset.

Lab Assignments to complete in this session:

Use the given dataset and perform the following tasks:

Dataset 1: Synthetic dataset

Dataset 2: CreditcardFraud.csv: The dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, the original features and more background information about the data are not provided. Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

1. Implement Decision Tree classifier and Logistic Regression on Dataset 1 using K fold cross validation and compare the results with AdaBoost classifier with base learner as Decision tree and Logistic Regression.
2. Check if there is class imbalance problem in Dataset 2. Compare the results of decision tree classifier and AdaBoost classifier on Dataset 2 and write your analysis.

3. Implement AdaBoost with base learner as decision tree on dataset 2 using K fold cross validation. Perform Hyperparameter tuning using (a) different depth, (b) different learning rate and (c) grid search CV. Show your results using Boxplot.

Code and Output:

1. Implement Decision Tree classifier and Logistic Regression on Dataset 1 using K fold cross validation and compare the results with AdaBoost classifier with base learner as Decision tree and Logistic Regression.

In [15]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold, RepeatedKFo
ld, GridSearchCV
from sklearn.ensemble import AdaBoostClassifier, AdaBoostRegressor
from sklearn import linear_model
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import make_scorer
from collections import Counter
```

In [16]:

```
x,y = make_classification(n_features = 20,n_samples=1000, n_informative=15, n_redundant=
5, random_state=6)
print(x.shape,y.shape)
```

```
(1000, 20) (1000,)
```

In [17]:

```
x, y
```

Out[17]:

```
(array([[-3.47224758,  1.95378146,  0.04875169, ...,  2.07283886,
          0.08385173,  0.91461126],
        [-2.42264447,  1.49687583, -0.80110683, ...,  1.14726175,
         -2.86306705, -0.27575018],
        [-4.01744369, -2.26537329,  2.72577799, ...,  1.34014025,
         -0.78634498, -1.17749558],
        ...,
        [ 2.39019744, -0.28042398, -0.01286339, ..., -0.95516099,
         -0.76710459,  1.70412285],
        [ 0.60081099,  1.84539674, -1.58163928, ..., -1.55912569,
         -2.26992832,  0.42082267],
        [-1.27669747,  1.6527396 , -1.39187956, ...,  0.72869505,
         -0.49441791,  2.75397458]]),
 array([0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0,
        1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
        0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1,
        1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1,
        0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0,
        0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0,
        0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1,
        0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1,
        0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
        1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
        0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
        1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0,
        1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1,
        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1,
        0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1,
        1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0,
        1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0,
        1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0,
```

```
       1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1,
       1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0,
       1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0,
       1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0,
       1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1,
       0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
       1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1,
       0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0,
       0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
       0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
       1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1,
       0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0,
       0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0,
       0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0,
       0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0,
       0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0,
       1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0,
       1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1,
       0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1,
       1, 1, 1, 1, 1, 0, 1, 1, 0, 1]))
```

WIth base learner as Decision Tree

In [18]:

```
abc = AdaBoostClassifier()
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(abc, X, y, scoring='accuracy', cv=cv, n_jobs=-1, error_score=
'raise')
```

In [19]:

```
print('Accuracy: %.3f (%.3f)' % (np.mean(n_scores), np.std(n_scores)))
```

```
Accuracy: 0.806 (0.041)
```

In [20]:

```
abc.fit(X, y)
row = [[-1.27,-2.236,-4.372,2.29,3.82,4.21,1.71,0.587,3.413,2.521,3.594,-2.109,4.5123,-0
.2332,-1.982,-2.3123,2.5432,-1.8673,-0.394,0.898]]
ypred = abc.predict(row)
print('Predicted Class: %d' % ypred[0])
```

```
Predicted Class: 0
```

With base learner as Logistic Regression

In [21]:

```
abb = AdaBoostClassifier(base_estimator=linear_model.LogisticRegression())
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(abb, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-
1, error_score='raise')
```

In [22]:

```
print('MAE: %.3f (%.3f)' % (np.mean(n_scores), np.std(n_scores)))
```

```
MAE: -0.204 (0.033)
```

In [23]:

```
abb.fit(X, y)
```

```
row = [[-1.27,-2.236,-4.372,2.29,3.82,4.21,1.71,0.587,3.413,2.521,3.594,-2.109,4.5123,-0
.2332,-1.982,-2.3123,2.5432,-1.8673,-0.394,0.898]]
ypred = abb.predict(row)
print('Predicted Class: %d' % ypred[0])
```

```
Predicted Class: 0
```

We can see that there is slight change in metrics with Decision Tree and Logistic Regession as base learner

AdaBoost ensemble depth effect on performance

In [24]:

```
def get_models():
  models = {}
  # explore depths from 1 - 10
  for i in range(1,10):
    # define base model
    base = DecisionTreeClassifier(max_depth=i)
    # define ensemble model
    models[str(i)] = AdaBoostClassifier(base_estimator=base)
  return models
```

In [25]:

```
def evaluate(model, x, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  n_scores = cross_val_score(model, x, y, scoring='accuracy', cv=cv, n_jobs=1, error_sco
re='raise')
  return [np.mean(n_scores), np.std(n_scores)]
  # return n_scores

models = get_models()

results, names = [], []

#evaluating models
for name, model in models.items():
  scores = evaluate(model, x, y)
  results.append(scores)
  names.append(name)
  print(f'Name: {name} Accuracy: {scores[0]:.16f}  Std. Dev: {scores[1]:.16f}')
```

```
Name: 1 Accuracy: 0.8063333333333333  Std. Dev: 0.0407008053428376
Name: 2 Accuracy: 0.8633333333333333  Std. Dev: 0.0272437556556034
Name: 3 Accuracy: 0.8713333333333335  Std. Dev: 0.0282528268005561
Name: 4 Accuracy: 0.8930000000000001  Std. Dev: 0.0313209195267317
Name: 5 Accuracy: 0.9050000000000000  Std. Dev: 0.0312783205005213
Name: 6 Accuracy: 0.9256666666666667  Std. Dev: 0.0218606699094261
Name: 7 Accuracy: 0.9329999999999999  Std. Dev: 0.0270985854489369
Name: 8 Accuracy: 0.9320000000000002  Std. Dev: 0.0198997487421324
Name: 9 Accuracy: 0.9320000000000001  Std. Dev: 0.0210396451174126
```

2. Check if there is class imbalance problem in Dataset 2. Compare the results of decision tree classifier and AdaBoost classifier on Dataset 2 and write your analysis.

In [ ]:

```
data = pd.read_csv('/creditcard 1.csv')
```

In [ ]:

```
data.head()
```

Out[ ]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	0.
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	0.
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	0.

5 rows × 31 columns

In [ ]:

```
data.isna().sum()
```

Out[ ]:

```
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

In [ ]:

```
data.dropna(inplace=True)
```

In [ ]:

```
data.isna().sum()
```

Out[ ]:

```
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
```

```
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

In [ ]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 69.5 MB
```

In [ ]:

```
data.Class.value_counts()
```

Out[ ]:

```
0    284315
1       492
Name: Class, dtype: int64
```

There is class imbalance in the dataset given to us as we can there are 65283 '0.0' values but only 169 '1.0' values

In [ ]:

```
x = data.iloc[:,:-1].values
y = data.iloc[:,-1].values.reshape(-1,1)
```

In [ ]:

```
t = [(d) for d in y if d==0]
s = [(d) for d in y if d==1]
print('Before Over-Sampling: ')
print('Samples in class 0: ',len(t))
print('Samples in class 1: ',len(s))
```

```
Before Over-Sampling:
Samples in class 0:  284315
Samples in class 1:  492
```

In [ ]:

```
x, y
```

Out[ ]:

```
(array([[ 0.00000000e+00, -1.35980713e+00, -7.27811733e-02, ...,
           1.33558377e-01, -2.10530535e-02,  1.49620000e+02],
        [ 0.00000000e+00,  1.19185711e+00,  2.66150712e-01, ...,
          -8.98309914e-03,  1.47241692e-02,  2.69000000e+00],
        [ 1.00000000e+00, -1.35835406e+00, -1.34016307e+00, ...,
          -5.53527940e-02, -5.97518406e-02,  3.78660000e+02],
        ...,
        [ 1.72788000e+05,  1.91956501e+00, -3.01253846e-01, ...,
           4.45477214e-03, -2.65608286e-02,  6.78800000e+01],
        [ 1.72788000e+05, -2.40440050e-01,  5.30482513e-01, ...,
           1.08820735e-01,  1.04532821e-01,  1.00000000e+01],
        [ 1.72792000e+05, -5.33412522e-01, -1.89733337e-01, ...,
          -2.41530880e-03,  1.36489143e-02,  2.17000000e+02]]), array([[0],
        [0],
        [0],
        ...,
        [0],
        [0],
        [0]]))
```

In [ ]:

```
from imblearn.over_sampling import RandomOverSampler
Over = RandomOverSampler()
x_Over, y_Over = Over.fit_resample(x, y)

t = [(d) for d in y_Over if d==0]
s = [(d) for d in y_Over if d==1]
print('After Over-Sampling: ')
print('Samples in class 0: ',len(t))
print('Samples in class 1: ',len(s))
```

```
After Over-Sampling:
Samples in class 0:  284315
Samples in class 1:  284315
```

In [ ]:

In [ ]:

```
dt = DecisionTreeClassifier()
dt.fit(x,y)
scores = evaluate(dt,x_Over,y_Over)
print(f'Accuracy: {np.mean(scores)}, STD of all accuracies: {np.std(scores)}')
```

```
Accuracy: 0.49992191494855764, STD of all accuracies: 0.4998647066976212
```

In [ ]:

```
model_ada = AdaBoostClassifier()
model_ada.fit(x,y)
scores = evaluate(model_ada,x_Over,y_Over)
print(f'Accuracy: {np.mean(scores)}, STD of all accuracies: {np.std(scores)}')
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: DataConversionWar
ning: A column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

```
Accuracy: 0.4849726345554646, STD of all accuracies: 0.482659510541816
```

3. Implement AdaBoost with base learner as decision tree on dataset 1 using K fold cross validation. Perform Hyperparameter tuning using (a) different depth, (b) different learning rate and (c) grid search CV. Show your results using Boxplot.

a. Different depth

In [ ]:

```
a. Different depthfrom sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
# Set a list of models to evaluate
def get_models():
  models = {}
  # explore depths from 1 - 10
  for i in range(1,10):
    # define base model
    base = DecisionTreeClassifier(max_depth=i)
    # define ensemble model
    models[str(i)] = AdaBoostClassifier(base_estimator=base)
  return models
```

In [ ]:

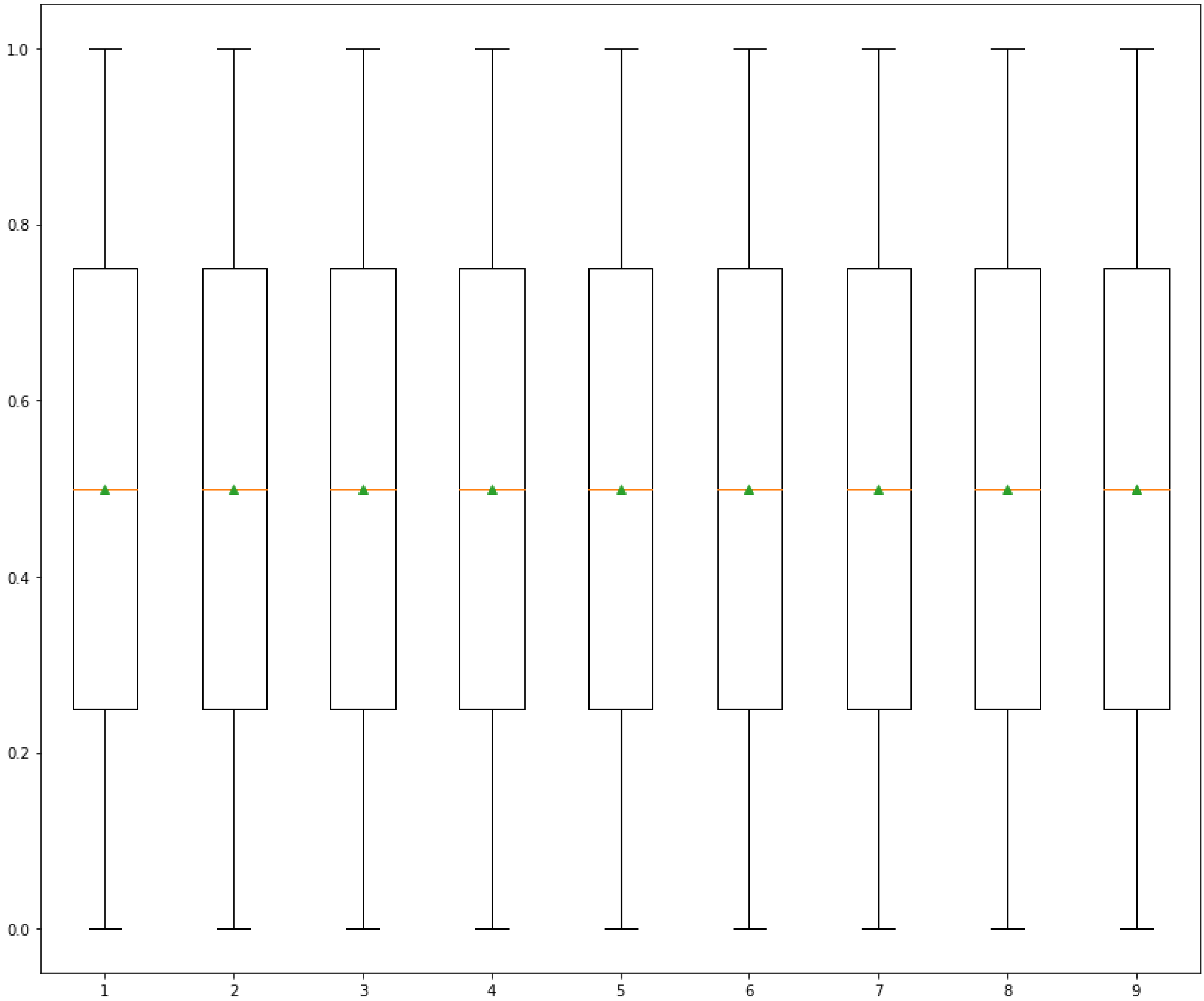
```
def evaluate(model, x, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  n_scores = cross_val_score(model, x, y, scoring='accuracy', cv=cv, n_jobs=1, error_sco
re='raise')
  return [np.mean(n_scores), np.std(n_scores)]
  # return n_scores

models = get_models()

results, names = [], []

#evaluating models
for name, model in models.items():
  scores = evaluate(model, x_Over, y_Over)
  results.append(scores)
  names.append(name)
  print(f'Name: {name}  Acc: {scores[0]:.16f}  STD: {scores[1]:.16f}')
```

```
Name: 1  Acc: 1.0000000000000000  STD: 0.0000000000000000
Name: 2  Acc: 1.0000000000000000  STD: 0.0000000000000000
Name: 3  Acc: 1.0000000000000000  STD: 0.0000000000000000
Name: 4  Acc: 0.9998045784477944  STD: 0.0003542316454620
Name: 5  Acc: 0.9996929089893913  STD: 0.0004578788237537
```

```
Name: 6  Acc: 0.9997766610831936  STD: 0.0004288746927900
Name: 7  Acc: 0.9998045784477945  STD: 0.0004150214613992
Name: 8  Acc: 0.9997766610831937  STD: 0.0004288746927900
Name: 9  Acc: 0.9998045784477944  STD: 0.0003542316454620
```

In [ ]:

```
plt.figure(figsize=(14,12))
plt.boxplot(results, labels=names, showmeans=True)
plt.show()
```



b. Different Learning Rate

In [ ]:

```
def get_models_lr():
  models = {}
  # explore depths from 0.1 - 2.1
  for i in np.arange(0.1,2.1,0.1):
    # define base model
    key = '%3f'%i
    # define ensemble model
    models[key] = AdaBoostClassifier(learning_rate=i)
  return models
```

In [ ]:

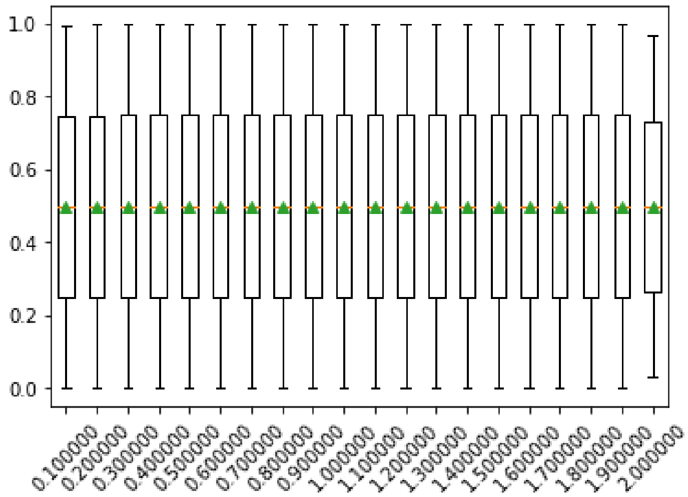
```
models = get_models_lr()

results, names = [], []
```

```
#evaluating models
for name, model in models.items():
  scores = evaluate(model, x_Over, y_Over)
  results.append(scores)
  names.append(name)
  print(f'Name: {name}  Acc: {scores[0]:.16f}  STD: {scores[1]:.16f}')
```

```
Name: 0.100000  Acc: 0.9959933703782617  STD: 0.0009224391344081
Name: 0.200000  Acc: 0.9974173713939490  STD: 0.0007506785199739
Name: 0.300000  Acc: 0.9979157758976669  STD: 0.0006982341325385
Name: 0.400000  Acc: 0.9983623916668395  STD: 0.0006296156455777
Name: 0.500000  Acc: 0.9986666188991988  STD: 0.0006052599731034
Name: 0.600000  Acc: 0.9990485084922386  STD: 0.0003990686575834
Name: 0.700000  Acc: 0.9993786156359264  STD: 0.0003831556496386
Name: 0.800000  Acc: 0.9994692368078509  STD: 0.0003128452778720
Name: 0.900000  Acc: 0.9996504753805842  STD: 0.0003225930864181
Name: 1.000000  Acc: 0.9997346202894835  STD: 0.0002155489749261
Name: 1.100000  Acc: 0.9996698890851963  STD: 0.0002842917529439
Name: 1.200000  Acc: 0.9997216690204714  STD: 0.0002491086126143
Name: 1.300000  Acc: 0.9997863964536428  STD: 0.0002027425807952
Name: 1.400000  Acc: 0.9997152002996780  STD: 0.0002227317523988
Name: 1.500000  Acc: 0.9997475615021862  STD: 0.0002076333723833
Name: 1.600000  Acc: 0.9998122901923961  STD: 0.0002035567587148
Name: 1.700000  Acc: 0.9997540339940958  STD: 0.0002181590120879
Name: 1.800000  Acc: 0.9997540377652115  STD: 0.0002238466754530
Name: 1.900000  Acc: 0.9995792729414263  STD: 0.0002927086109185
Name: 2.000000  Acc: 0.9661081191496638  STD: 0.0332473848510627
```

In [ ]:

```
plt.boxplot(results, labels=names, showmeans=True)
plt.xticks(rotation=45)
plt.show()
```



c. Grid Search

In [ ]:

```
def evaluate_grid(model, x, y, grid):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  n_scores = GridSearchCV(model, x, y,grid, scoring='accuracy', cv=cv, n_jobs=1)
  return [np.mean(n_scores), np.std(n_scores)]
```

In [ ]:

```
model = AdaBoostClassifier()
grid = {}
grid['n_estimators'] = [20, 50, 70, 100]
grid['learning_rate'] = [0.0001,0.001,0.01,0.1,1]
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='
```

```python
accuracy')
# execute the grid search
grid_result = grid_search.fit(x_Over, y_Over)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    # print("%f (%f) with: %r" % (mean, stdev, param))
    print(f"{mean:.4f} {stdev:.4f} {param}")
```

```
Best: 1.000000 using {'learning_rate': 0.1, 'n_estimators': 20}
0.9847 0.0035 {'learning_rate': 0.0001, 'n_estimators': 20}
0.9847 0.0035 {'learning_rate': 0.0001, 'n_estimators': 50}
0.9847 0.0035 {'learning_rate': 0.0001, 'n_estimators': 70}
0.9847 0.0035 {'learning_rate': 0.0001, 'n_estimators': 100}
0.9847 0.0035 {'learning_rate': 0.001, 'n_estimators': 20}
0.9853 0.0046 {'learning_rate': 0.001, 'n_estimators': 50}
0.9997 0.0005 {'learning_rate': 0.001, 'n_estimators': 70}
0.9997 0.0005 {'learning_rate': 0.001, 'n_estimators': 100}
0.9997 0.0005 {'learning_rate': 0.01, 'n_estimators': 20}
0.9999 0.0003 {'learning_rate': 0.01, 'n_estimators': 50}
0.9999 0.0003 {'learning_rate': 0.01, 'n_estimators': 70}
0.9999 0.0003 {'learning_rate': 0.01, 'n_estimators': 100}
1.0000 0.0000 {'learning_rate': 0.1, 'n_estimators': 20}
1.0000 0.0000 {'learning_rate': 0.1, 'n_estimators': 50}
1.0000 0.0000 {'learning_rate': 0.1, 'n_estimators': 70}
1.0000 0.0000 {'learning_rate': 0.1, 'n_estimators': 100}
1.0000 0.0002 {'learning_rate': 1, 'n_estimators': 20}
1.0000 0.0000 {'learning_rate': 1, 'n_estimators': 50}
1.0000 0.0000 {'learning_rate': 1, 'n_estimators': 70}
1.0000 0.0000 {'learning_rate': 1, 'n_estimators': 100}
```

In [ ]: