**Vivekanand Education Society’s**

**Institute of Technology**

**(Affiliated to University of Mumbai, Approved by AICTE & Recognized by Govt. of Maharashtra)**

**Department of Information Technology**

AIDS - 2 Lab

Experiment - 9

Aim: Supervised learning algorithm AdaBoost

| Roll No. | 53 |
| --- | --- |
| Name | SAKSHI SANTOSH PATIL |
| Class | D20B |
| Subject | AIDS - 2 |
| Grade: |  |

**EXPERIMENT 09**

**AIM:** Supervised learning algorithm AdaBoost

**THEORY**:

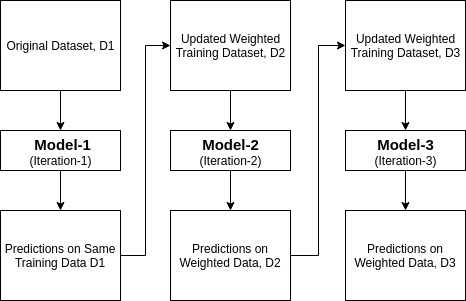
**Supervised learning** is a type of machine learning where an algorithm learns from labeled training data to make predictions or decisions without human intervention. It is called "supervised" because it involves a "teacher" who provides the algorithm with the correct answers during training, allowing the algorithm to learn the relationship between input data and output labels.

**AdaBoost (Adaptive Boosting)** is a supervised learning algorithm that combines multiple weak learners to create a strong predictive model, with a focus on correcting errors from previous models. It is used for classification tasks and aims to improve model accuracy by assigning more weight to misclassified samples in each iteration.

**AdaBoost Classifier**

Ada-boost or Adaptive Boosting is an ensemble boosting classifier proposed by Yoav Freund and Robert Schapire in 1996. It combines multiple classifiers to increase the accuracy of classifiers. AdaBoost is an iterative ensemble method. AdaBoost classifier builds a strong classifier by combining multiple poorly performing classifiers so that you will get high accuracy strong classifier. Any machine learning algorithm can be used as base classifier if it accepts weights on the training set. Adaboost should meet two conditions:

* The classifier should be trained interactively on various weighed training examples.
* In each iteration, it tries to provide an excellent fit for these examples by minimizing training error.



**How does the AdaBoost algorithm work?**

* Initially, Adaboost selects a training subset randomly.
* It iteratively trains the AdaBoost machine learning model by selecting the training set based on the accurate prediction of the last training.
* It assigns the higher weight to wrong classified observations so that in the next iteration these observations will get the high probability for classification.
* Also, It assigns the weight to the trained classifier in each iteration according to the accuracy of the classifier. The more accurate classifier will get high weight.
* This process iterates until the complete training data fits without any error or until reached the specified maximum number of estimators.
* To classify, perform a "vote" across all of the learning algorithms you built.

**Pros:** AdaBoost is easy to implement. It iteratively corrects the mistakes of the weak classifier and improves accuracy by combining weak learners. You can use many base classifiers with AdaBoost. AdaBoost is not prone to overfitting. This can be found out via experiment results, but there is no concrete reason available.

**Cons:** AdaBoost is sensitive to noise data. It is highly affected by outliers because it tries to fit each point perfectly. AdaBoost is slower compared to XGBoost.

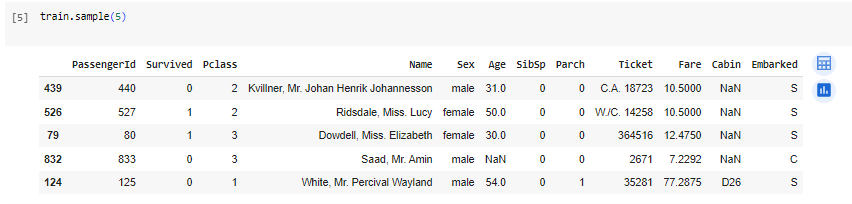
**IMPLEMENTATION**:

**Dataset Link:** [**https://www.kaggle.com/competitions/titanic/data?select=test.csv**](https://www.kaggle.com/competitions/titanic/data?select=test.csv)

* **Import libraries and read the dataset**

****

* **Read the dataset**

****

**1. Data Loading and Preprocessing:**

* Load the Titanic dataset into `train` and `test`.
* Save passenger IDs from the test dataset.
* Fill missing values in the "Embarked" column with "C" and in the "Fare" column in the test dataset with the mean fare.
* Perform feature engineering, such as converting "Sex" to numerical values, calculating name length, grouping names by length, and extracting titles from names.

**2. Family and Fare Features:**

* Create new features like "family\_size" based on the number of family members and "is\_alone" to indicate if a passenger is alone.
* Calculate "calculated\_fare" based on the fare divided by family size.
* Group fares into "fare\_group" categories.

**3. Missing Age Values:** Use a Random Forest Regressor to predict missing "Age" values based on features like "Sex," "SibSp" (siblings/spouses), and "Parch" (parents/children).

**4. Age Group:** Create age groups based on passengers' ages.

**5. Convert Categorical Features:** Convert categorical variables like "title," "Pclass," "Cabin," "Embarked," "nLength\_group," "family\_group," "fare\_group," and "age\_group" into dummy variables.

**6. Separate Features and Target:**

**7. Standardize Features:**

**8. AdaBoost Classifier:**

* Create an AdaBoost classifier using a Decision Tree base estimator with a specified maximum depth.
* Fit the AdaBoost classifier to the training data.
* Make predictions on the test data.
* Calculate the accuracy of the model's predictions.

**This code performs data preprocessing, feature engineering, and builds an AdaBoost classifier to predict passenger survival on the Titanic.**

## saving passenger id in advance in order to submit later.

passengerid = test.PassengerId

## Replacing the null values in the Embarked column with the mode.

train.Embarked.fillna("C", inplace=True)

## Concat train and test into a variable "all\_data"

survivers = train.Survived

train.drop(["Survived"],axis=1, inplace=True)

all\_data = pd.concat([train,test], ignore\_index=False)

## Assign all the null values to N

all\_data.Cabin.fillna("N", inplace=True)

all\_data.Cabin = [i[0] for i in all\_data.Cabin]

with\_N = all\_data[all\_data.Cabin == "N"]

without\_N = all\_data[all\_data.Cabin != "N"]

all\_data.groupby("Cabin")['Fare'].mean().sort\_values()

def cabin\_estimator(i):

a = 0

if i<16:

a = "G"

elif i>=16 and i<27:

a = "F"

elif i>=27 and i<38:

a = "T"

elif i>=38 and i<47:

a = "A"

elif i>= 47 and i<53:

a = "E"

elif i>= 53 and i<54:

a = "D"

elif i>=54 and i<116:

a = 'C'

else:

a = "B"

return a

##applying cabin estimator function.

with\_N['Cabin'] = with\_N.Fare.apply(lambda x: cabin\_estimator(x))

## getting back on the train.

all\_data = pd.concat([with\_N, without\_N], axis=0)

## PassengerId helps us separate train and test.

all\_data.sort\_values(by = 'PassengerId', inplace=True)

## Separating train and test from all\_data.

train = all\_data[:891]

test = all\_data[891:]

train['Survived'] = survivers

missing\_value = test[(test.Pclass == 3) & (test.Embarked == "S") & (test.Sex == "male")].Fare.mean()

## replace the test.fare null values with test.fare mean

test.Fare.fillna(missing\_value, inplace=True)

## dropping the three outliers where Fare is over $500

train = train[train.Fare < 500]

# Placing 0 for female and

# 1 for male in the "Sex" column.

train['Sex'] = train.Sex.apply(lambda x: 0 if x == "female" else 1)

test['Sex'] = test.Sex.apply(lambda x: 0 if x == "female" else 1)

# Creating a new colomn with a

train['name\_length'] = [len(i) for i in train.Name]

test['name\_length'] = [len(i) for i in test.Name]

def name\_length\_group(size):

a = ''

if (size <=20):

a = 'short'

elif (size <=35):

a = 'medium'

elif (size <=45):

a = 'good'

else:

a = 'long'

return a

train['nLength\_group'] = train['name\_length'].map(name\_length\_group)

test['nLength\_group'] = test['name\_length'].map(name\_length\_group)

train["title"] = [i.split('.')[0] for i in train.Name]

train["title"] = [i.split(',')[1] for i in train.title]

test["title"] = [i.split('.')[0] for i in test.Name]

test["title"]= [i.split(',')[1] for i in test.title]

train["title"] = [i.replace('Ms', 'Miss') for i in train.title]

train["title"] = [i.replace('Mlle', 'Miss') for i in train.title]

train["title"] = [i.replace('Mme', 'Mrs') for i in train.title]

train["title"] = [i.replace('Dr', 'rare') for i in train.title]

train["title"] = [i.replace('Col', 'rare') for i in train.title]

train["title"] = [i.replace('Major', 'rare') for i in train.title]

train["title"] = [i.replace('Don', 'rare') for i in train.title]

train["title"] = [i.replace('Jonkheer', 'rare') for i in train.title]

train["title"] = [i.replace('Sir', 'rare') for i in train.title]

train["title"] = [i.replace('Lady', 'rare') for i in train.title]

train["title"] = [i.replace('Capt', 'rare') for i in train.title]

train["title"] = [i.replace('the Countess', 'rare') for i in train.title]

train["title"] = [i.replace('Rev', 'rare') for i in train.title]

test['title'] = [i.replace('Ms', 'Miss') for i in test.title]

test['title'] = [i.replace('Dr', 'rare') for i in test.title]

test['title'] = [i.replace('Col', 'rare') for i in test.title]

test['title'] = [i.replace('Dona', 'rare') for i in test.title]

test['title'] = [i.replace('Rev', 'rare') for i in test.title]

## Family\_size seems like a good feature to create

train['family\_size'] = train.SibSp + train.Parch+1

test['family\_size'] = test.SibSp + test.Parch+1

def family\_group(size):

a = ''

if (size <= 1):

a = 'loner'

elif (size <= 4):

a = 'small'

else:

a = 'large'

return a

train['family\_group'] = train['family\_size'].map(family\_group)

test['family\_group'] = test['family\_size'].map(family\_group)

train['is\_alone'] = [1 if i<2 else 0 for i in train.family\_size]

test['is\_alone'] = [1 if i<2 else 0 for i in test.family\_size]

train.drop(['Ticket'], axis=1, inplace=True)

test.drop(['Ticket'], axis=1, inplace=True)

## Calculating fare based on family size.

train['calculated\_fare'] = train.Fare/train.family\_size

test['calculated\_fare'] = test.Fare/test.family\_size

def fare\_group(fare):

a= ''

if fare <= 4:

a = 'Very\_low'

elif fare <= 10:

a = 'low'

elif fare <= 20:

a = 'mid'

elif fare <= 45:

a = 'high'

else:

a = "very\_high"

return a

train['fare\_group'] = train['calculated\_fare'].map(fare\_group)

test['fare\_group'] = test['calculated\_fare'].map(fare\_group)

train.drop(['PassengerId'], axis=1, inplace=True)

test.drop(['PassengerId'], axis=1, inplace=True)

train = pd.get\_dummies(train,

columns=['title',"Pclass", 'Cabin','Embarked','nLength\_group', 'family\_group', 'fare\_group'], drop\_first=False)

test=pd.get\_dummies(test, columns=['title',"Pclass",'Cabin','Embarked','nLength\_group', 'family\_group', 'fare\_group'], drop\_first=False)

train.drop(['family\_size','Name', 'Fare','name\_length'], axis=1, inplace=True)

test.drop(['Name','family\_size',"Fare",'name\_length'], axis=1, inplace=True)

## rearranging the columns so that I can easily use the dataframe to predict the missing age values.

train = pd.concat([train[["Survived", "Age", "Sex","SibSp","Parch"]], train.loc[:,"is\_alone":]], axis=1)

test = pd.concat([test[["Age", "Sex"]], test.loc[:,"SibSp":]], axis=1)

## Importing RandomForestRegressor

from sklearn.ensemble import RandomForestRegressor

## writing a function that takes a dataframe with missing values and outputs it by filling the missing values.

def completing\_age(df):

## gettting all the features except survived

age\_df = df.loc[:,"Age":]

temp\_train = age\_df.loc[age\_df.Age.notnull()] ## df with age values

temp\_test = age\_df.loc[age\_df.Age.isnull()] ## df without age values

y = temp\_train.Age.values ## setting target variables(age) in y

x = temp\_train.loc[:, "Sex":].values

rfr = RandomForestRegressor(n\_estimators=1500, n\_jobs=-1)

rfr.fit(x, y)

predicted\_age = rfr.predict(temp\_test.loc[:, "Sex":])

df.loc[df.Age.isnull(), "Age"] = predicted\_age

return df

## Implementing the completing\_age function in both train and test dataset.

completing\_age(train)

completing\_age(test);

## create bins for age

def age\_group\_fun(age):

a = ''

if age <= 1:

a = 'infant'

elif age <= 4:

a = 'toddler'

elif age <= 13:

a = 'child'

elif age <= 18:

a = 'teenager'

elif age <= 35:

a = 'Young\_Adult'

elif age <= 45:

a = 'adult'

elif age <= 55:

a = 'middle\_aged'

elif age <= 65:

a = 'senior\_citizen'

else:

a = 'old'

return a

## Applying "age\_group\_fun" function to the "Age" column.

train['age\_group'] = train['Age'].map(age\_group\_fun)

test['age\_group'] = test['Age'].map(age\_group\_fun)

## Creating dummies for "age\_group" feature.

train = pd.get\_dummies(train,columns=['age\_group'], drop\_first=True)

test = pd.get\_dummies(test,columns=['age\_group'], drop\_first=True);

"""train.drop('Age', axis=1, inplace=True)

test.drop('Age', axis=1, inplace=True)"""

# separating our independent and dependent variable

X = train.drop(['Survived'], axis = 1)

y = train["Survived"]

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size = .33, random\_state = 0)

# Feature Scaling

## We will be using standardscaler to transform

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

## transforming "X\_train"

X\_train = sc.fit\_transform(X\_train)

## transforming "X\_test"

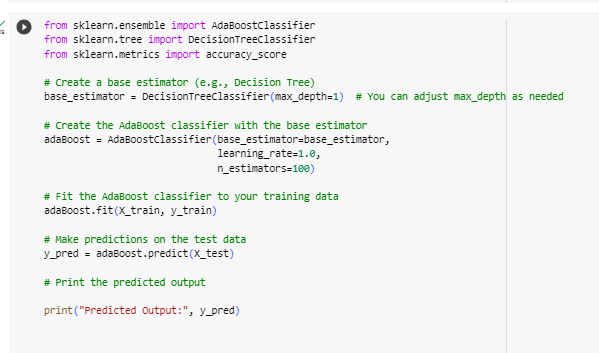
X\_test = sc.transform(X\_test)

from sklearn.model\_selection import StratifiedShuffleSplit, cross\_val\_score

cv = StratifiedShuffleSplit(n\_splits = 10, test\_size = .25, random\_state = 0 )

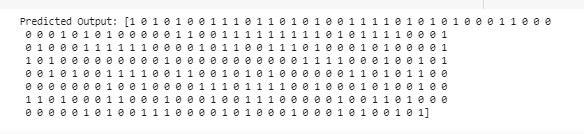
column\_names = X.columns

X = sc.fit\_transform(X)

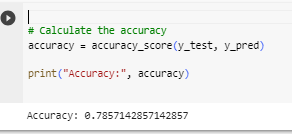


**0: This label typically represents the negative class or the "no" class. In the context of the Titanic dataset, it often means that the algorithm predicts that a passenger did not survive.**

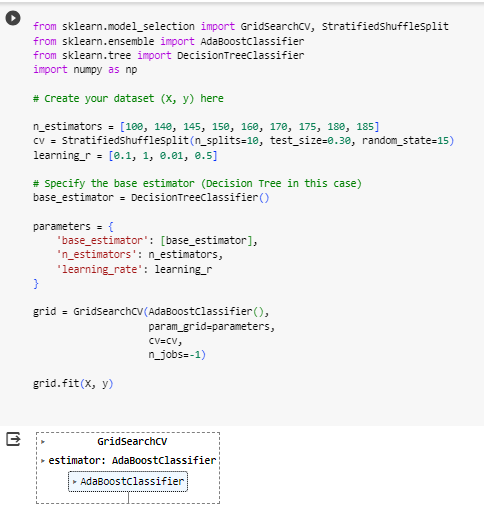
**1: This label typically represents the positive class or the "yes" class. In the context of the Titanic dataset, it often means that the algorithm predicts that a passenger survived.**



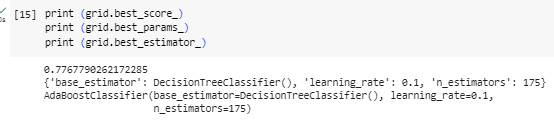
* **Calculating the accuracy of the model**



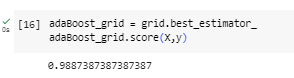
* **The code essentially performs a grid search to find the best combination of n\_estimators and learning\_rate for the AdaBoost classifier with a Decision Tree base estimator. It does this by fitting the classifier multiple times with different hyperparameters and selecting the combination that produces the best results based on cross-validation.**



* **grid.best\_score\_` shows the highest mean performance score.**
* **grid.best\_params\_` displays the optimal hyperparameters.**
* **grid.best\_estimator\_` is the best-performing model.**



* **These terms represent the process of evaluating the best-performing AdaBoost model on the training data and storing it in the adaBoost\_grid variable for future use. The score method helps measure the model's performance on the training set by calculating its accuracy.**



**CONCLUSION:**

AdaBoost (Adaptive Boosting) is primarily used for making predictions in machine learning, particularly in classification tasks. It is a boosting algorithm that combines the predictions of multiple weak learners (typically decision trees) to create a strong ensemble model. This ensemble model is then used for predicting the class labels or values of unseen or new data points. Thus we used adaboosting to predict if the people in the Titanic survived or not.