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.	70
Name	MAYURI SHRIDATTA YERANDE
Class	D20B
Subject	AIDS - 2
Grade:	

EXPERIMENT - 9

AIM: Supervised learning algorithm AdaBoost

THEORY:

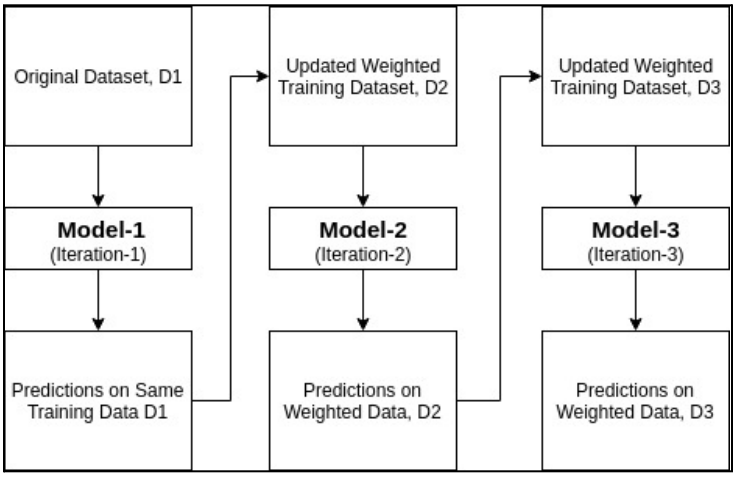
Supervised learning is a type of machine learning where an algorithm learns from labeled training data to make predictions or decisions without human intervention. It is called "supervised" because it involves a "teacher" who provides the algorithm with the correct answers during training, allowing the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

- **Import libraries and read the dataset**

```
[19]
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns


## Importing the datasets
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
```

- **Read the dataset**

```
[5] train.sample(5)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
439	440	0	2	Kvillner, Mr. Johan Henrik Johannesson	male	31.0	0	0	C.A. 18723	10.5000	NaN	S
526	527	1	2	Ridsdale, Miss. Lucy	female	50.0	0	0	W./C. 14258	10.5000	NaN	S
79	80	1	3	Dowdell, Miss. Elizabeth	female	30.0	0	0	364516	12.4750	NaN	S
832	833	0	3	Saad, Mr. Amin	male	NaN	0	0	2671	7.2292	NaN	C
124	125	0	1	White, Mr. Percival Wayland	male	54.0	0	1	35281	77.2875	D26	S

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)
```

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Create a base estimator (e.g., Decision Tree)
base_estimator = DecisionTreeClassifier(max_depth=1)  # You can adjust max_depth as needed

# Create the AdaBoost classifier with the base estimator
adaBoost = AdaBoostClassifier(base_estimator=base_estimator,
                              learning_rate=1.0,
                              n_estimators=100)

# Fit the AdaBoost classifier to your training data
adaBoost.fit(X_train, y_train)

# Make predictions on the test data
y_pred = adaBoost.predict(X_test)

# Print the predicted output

print("Predicted Output:", y_pred)
```

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.

```
Predicted Output: [1 0 1 0 1 0 0 1 1 1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 1 0 1 0 1 0 0 0 1 1 0 0 0
 0 0 0 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 1
 0 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 0 0 0 0 1
 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 0 1 0 1
 0 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 0 0
 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0
 1 1 0 1 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 1 0 0 1 1 0 1 0 0 0
 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1]
```

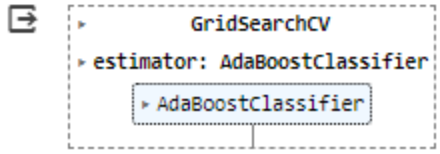
- **Calculating the accuracy of the model**

```
# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

```
Accuracy: 0.7857142857142857
```

- **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.**

```
from sklearn.model_selection import GridSearchCV, StratifiedShuffleSplit
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Create your dataset (X, y) here

n_estimators = [100, 140, 145, 150, 160, 170, 175, 180, 185]
cv = StratifiedShuffleSplit(n_splits=10, test_size=0.30, random_state=15)
learning_r = [0.1, 1, 0.01, 0.5]

# Specify the base estimator (Decision Tree in this case)
base_estimator = DecisionTreeClassifier()

parameters = {
    'base_estimator': [base_estimator],
    'n_estimators': n_estimators,
    'learning_rate': learning_r
}

grid = GridSearchCV(AdaBoostClassifier(),
                    param_grid=parameters,
                    cv=cv,
                    n_jobs=-1)

grid.fit(X, y)
```



- **grid.best_score_` shows the highest mean performance score.**
- **grid.best_params_` displays the optimal hyperparameters.**
- **grid.best_estimator_` is the best-performing model.**

```
[15] print (grid.best_score_)
     print (grid.best_params_)
     print (grid.best_estimator_)
```

```
0.7767790262172285
{'base_estimator': DecisionTreeClassifier(), 'learning_rate': 0.1, 'n_estimators': 175}
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(), learning_rate=0.1,
                   n_estimators=175)
```

- **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.**

```
[16] adaBoost_grid = grid.best_estimator_
     adaBoost_grid.score(X,y)
```

```
0.9887387387387387
```

<u>CONCLUSION</u>: 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.