Aim: Implementation of AdaBoost Classifier, Gradient Descent Algorithm, Voting Ensemble Algorithm.

THEORY:

A) ADABOOST CLASSIFIER:

- AdaBoost or Adaptive Boosting is one of the ensemble boosting classifier proposed by Yoav Freund and Robert Schapire in 1996.
- It combines multiple weak 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.
- The basic concept behind Adaboost is to set the weights of classifiers and training the data sample in each iteration such that it ensures the accurate predictions of unusual observations.
- Any machine learning algorithm can be used as base classifier if it accepts weights on the training set.

AdaBoost should meet two conditions :-

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

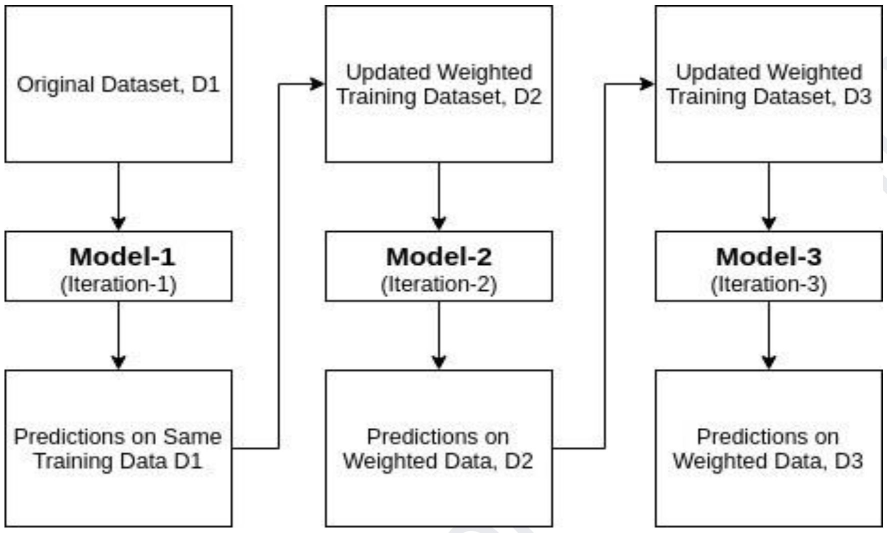
- To build a AdaBoost classifier, imagine that as a first base classifier we train a Decision Tree algorithm to make predictions on our training data.
- Now, following the methodology of AdaBoost, the weight of the misclassified training instances is increased.
- The second classifier is trained and acknowledges the updated weights and it repeats the procedure over and over again.
- At the end of every model prediction we end up boosting the weights of the misclassified instances so that the next model does a better job on them, and so on.
- AdaBoost adds predictors to the ensemble gradually making it better. The great disadvantage of this algorithm is that the model cannot be parallelized since each predictor can only be trained after the previous one has been trained and evaluated.

Below are the steps for performing the AdaBoost algorithm :-

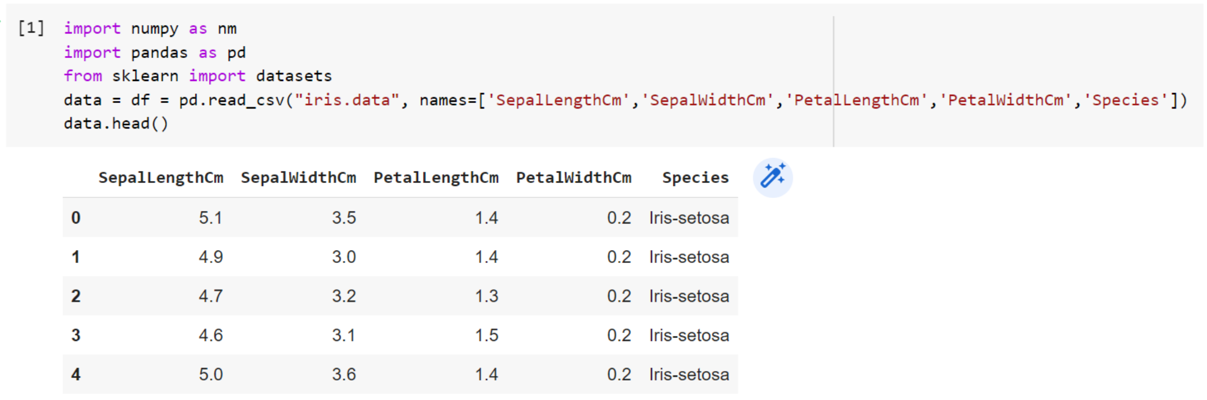
1. Initially, all observations are given equal weights.
2. A model is built on a subset of data.
3. Using this model, predictions are made on the whole dataset.
4. Errors are calculated by comparing the predictions and actual values.
5. While creating the next model, higher weights are given to the data points which were predicted incorrectly.
6. Weights can be determined using the error value. For instance, the higher the error the more is the weight assigned to the observation.
7. This process is repeated until the error function does not change, or the maximum limit of the number of estimators is reached. **AdaBoost algorithm intuition :-**

It works in the following steps:

1. Initially, Adaboost selects a training subset randomly.
2. It iteratively trains the AdaBoost machine learning model by selecting the training set based on the accurate prediction of the last training.
3. It assigns the higher weight to wrong classified observations so that in the next iteration these observations will get the high probability for classification.
4. 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.
5. This process iterate until the complete training data fits without any error or until reached to the specified maximum number of estimators.
6. To classify, perform a "vote" across all of the learning algorithms you built. The intuition can be depicted with the following diagram

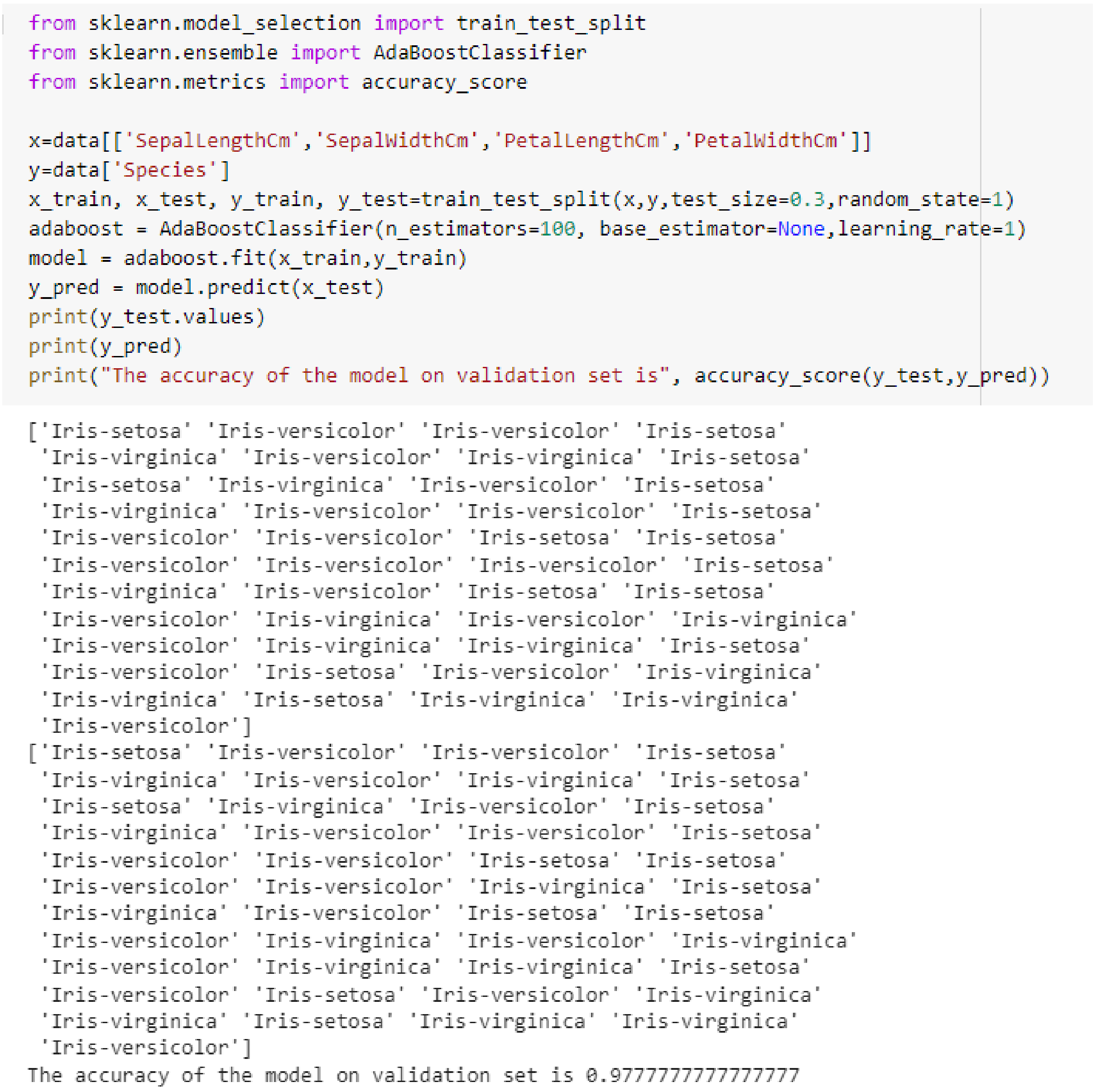

Original Dataset, D1
Model-1 (Iteration-1)
Predictions on Same Training Data D1
Updated Weighted Training Dataset, D2
Model-2 (Iteration-2)
Predictions on Weighted Data, D2
Updated Weighted Training Dataset, D3
Model-3 (Iteration-3)
Predictions on Weighted Data, D3

SOURCE CODE:



```
[1] import numpy as nm
    import pandas as pd
    from sklearn import datasets
    data = df = pd.read_csv("iris.data", names=['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm','Species'])
    data.head()
```

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa



```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

x=data[['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm']]
y=data['Species']
x_train, x_test, y_train, y_test=train_test_split(x,y,test_size=0.3,random_state=1)
adaboost = AdaBoostClassifier(n_estimators=100, base_estimator=None,learning_rate=1)
model = adaboost.fit(x_train,y_train)
y_pred = model.predict(x_test)
print(y_test.values)
print(y_pred)
print("The accuracy of the model on validation set is", accuracy_score(y_test,y_pred))
```

```
['Iris-setosa' 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa'
 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor' 'Iris-setosa'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa' 'Iris-setosa'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa'
 'Iris-virginica' 'Iris-versicolor' 'Iris-setosa' 'Iris-setosa'
 'Iris-versicolor' 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica'
 'Iris-versicolor' 'Iris-virginica' 'Iris-virginica' 'Iris-setosa'
 'Iris-versicolor' 'Iris-setosa' 'Iris-versicolor' 'Iris-virginica'
 'Iris-virginica' 'Iris-setosa' 'Iris-virginica' 'Iris-virginica'
 'Iris-versicolor']
['Iris-setosa' 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa'
 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor' 'Iris-setosa'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-setosa' 'Iris-setosa'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-virginica' 'Iris-versicolor' 'Iris-setosa' 'Iris-setosa'
 'Iris-versicolor' 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica'
 'Iris-versicolor' 'Iris-virginica' 'Iris-virginica' 'Iris-setosa'
 'Iris-versicolor' 'Iris-setosa' 'Iris-versicolor' 'Iris-virginica'
 'Iris-virginica' 'Iris-setosa' 'Iris-virginica' 'Iris-virginica'
 'Iris-versicolor']
The accuracy of the model on validation set is 0.9777777777777777
```

B) GRADIENT DESCENT ALGORITHM TO OPTIMIZE VARIOUS MODELS.

Gradient Descent is the most common optimization algorithm in *machine learning* and *deep learning*. It is a first-order optimization algorithm. This means it only takes into account the first derivative when performing the updates on the parameters. On each iteration, we update the parameters in the opposite direction of the gradient of the objective function *J(w)* w.r.t the parameters where the gradient gives the direction of the steepest ascent. The size of the step we take on each iteration to reach the local minimum is determined by the learning rate α. Therefore, we follow the direction of the slope downhill until we reach a local minimum.

Types of Gradient Descent

There are three popular types of gradient descent that mainly differ in the amount of data they use:

- **Batch gradient descent**

Batch gradient descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle and it's called a training epoch.

- **Stochastic gradient descent**

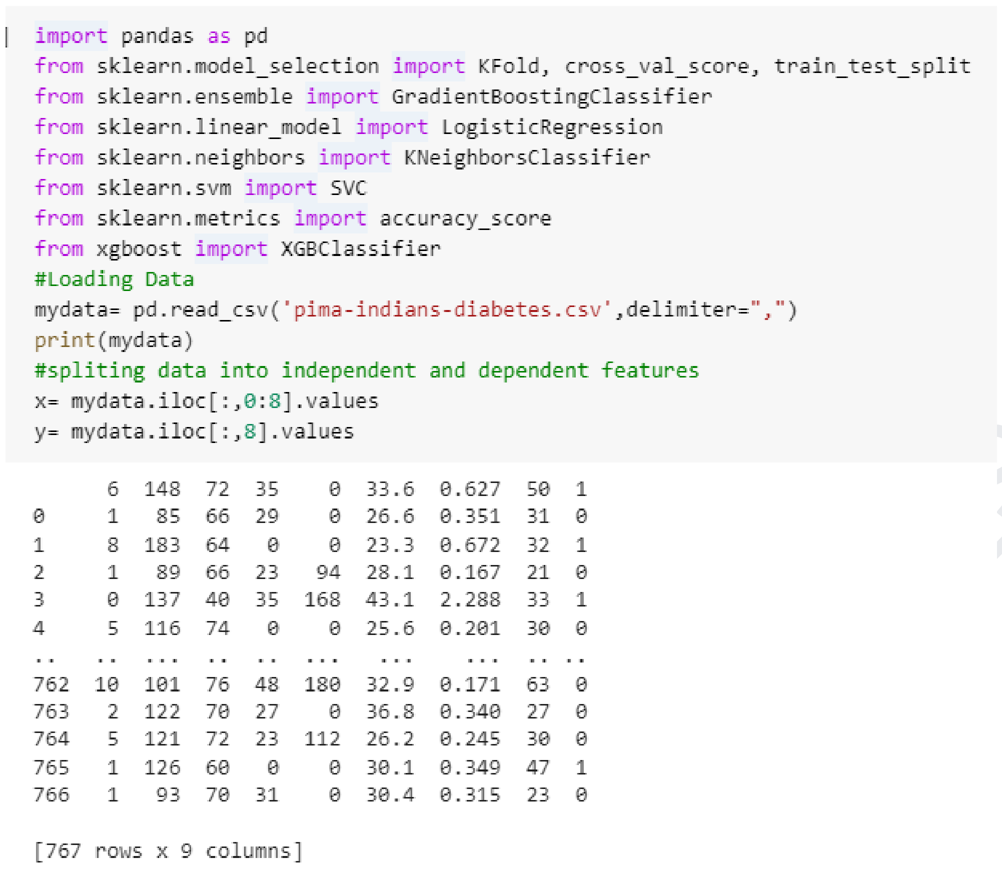
By contrast, stochastic gradient descent (SGD) does this for each training example within the dataset, meaning it updates the parameters for each training example one by one. Depending on the problem, this can make SGD faster than batch gradient descent. One advantage is the frequent updates allow us to have a pretty detailed rate of improvement.
The frequent updates, however, are more computationally expensive than the batch gradient descent approach. Additionally, the frequency of those updates can result in noisy gradients, which may cause the error rate to jump around instead of slowly decreasing.

- **Mini-batch gradient descent**

Mini-batch gradient descent is the go-to method since it’s a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

Common mini-batch sizes range between 50 and 256, but like any other machine learning technique, there is no clear rule because it varies for different applications. This is the go-to algorithm when training a neural network and it is the most common type of gradient descent within deep learning.

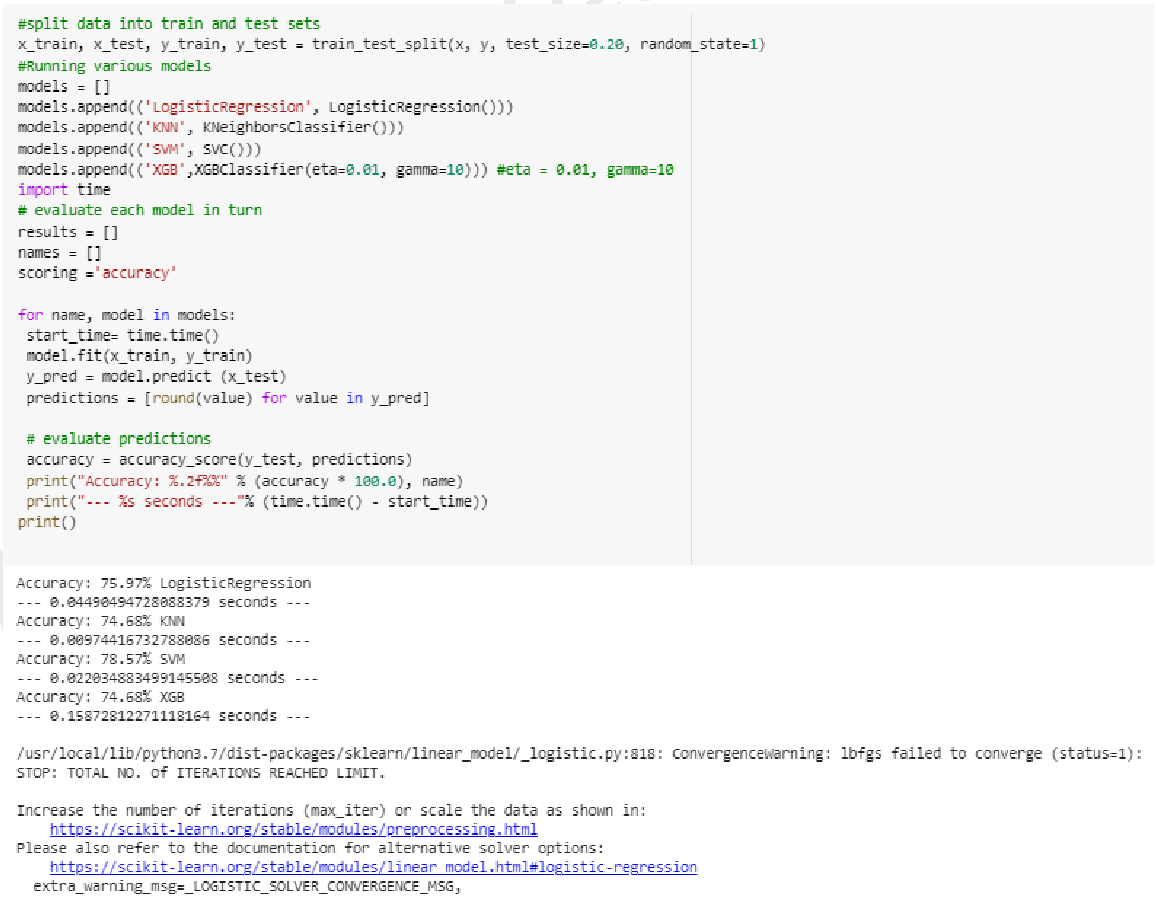
SOURCE CODE:



```
import pandas as pd
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from xgboost import XGBClassifier
#Loading Data
mydata= pd.read_csv('pima-indians-diabetes.csv',delimiter=",")
print(mydata)
#spliting data into independent and dependent features
x= mydata.iloc[:,0:8].values
y= mydata.iloc[:,8].values
```

```
      6  148  72  35    0  33.6  0.627  50  1
0     1   85  66  29    0  26.6  0.351  31  0
1     8  183  64   0    0  23.3  0.672  32  1
2     1   89  66  23   94  28.1  0.167  21  0
3     0  137  40  35  168  43.1  2.288  33  1
4     5  116  74   0    0  25.6  0.201  30  0
..   ..  ...  ..  ..  ...   ...    ...  .. ..
762  10  101  76  48  180  32.9  0.171  63  0
763   2  122  70  27    0  36.8  0.340  27  0
764   5  121  72  23  112  26.2  0.245  30  0
765   1  126  60   0    0  30.1  0.349  47  1
766   1   93  70  31    0  30.4  0.315  23  0

[767 rows x 9 columns]
```



```
#split data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=1)
#Running various models
models = []
models.append(('LogisticRegression', LogisticRegression()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('SVM', SVC()))
models.append(('XGB',XGBClassifier(eta=0.01, gamma=10))) #eta = 0.01, gamma=10
import time
# evaluate each model in turn
results = []
names = []
scoring ='accuracy'

for name, model in models:
 start_time= time.time()
 model.fit(x_train, y_train)
 y_pred = model.predict (x_test)
 predictions = [round(value) for value in y_pred]

 # evaluate predictions
 accuracy = accuracy_score(y_test, predictions)
 print("Accuracy: %.2f%%" % (accuracy * 100.0), name)
 print("--- %s seconds ---"% (time.time() - start_time))
print()
```

```
Accuracy: 75.97% LogisticRegression
--- 0.04490494728088379 seconds ---
Accuracy: 74.68% KNN
--- 0.00974416732788086 seconds ---
Accuracy: 78.57% SVM
--- 0.022034883499145508 seconds ---
Accuracy: 74.68% XGB
--- 0.15872812271118164 seconds ---

/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG,
```

C) VOTING ENSEMBLE ALGORITHM USING IRIS DATASET:

- Voting is an ensemble machine learning algorithm.
- For regression, a voting ensemble involves making a prediction that is the average of multiple other regression models.
- A voting ensemble (or a “majority voting ensemble”) is an ensemble machine learning model that combines the predictions from multiple other models.
- It is a technique that may be used to improve model performance, ideally achieving better performance than any single model used in the ensemble.
- A voting ensemble works by combining the predictions from multiple models. It can be used for classification or regression. In the case of regression, this involves calculating the average of the predictions from the models. In the case of classification, the predictions for each label are summed and the label with the majority vote is predicted.

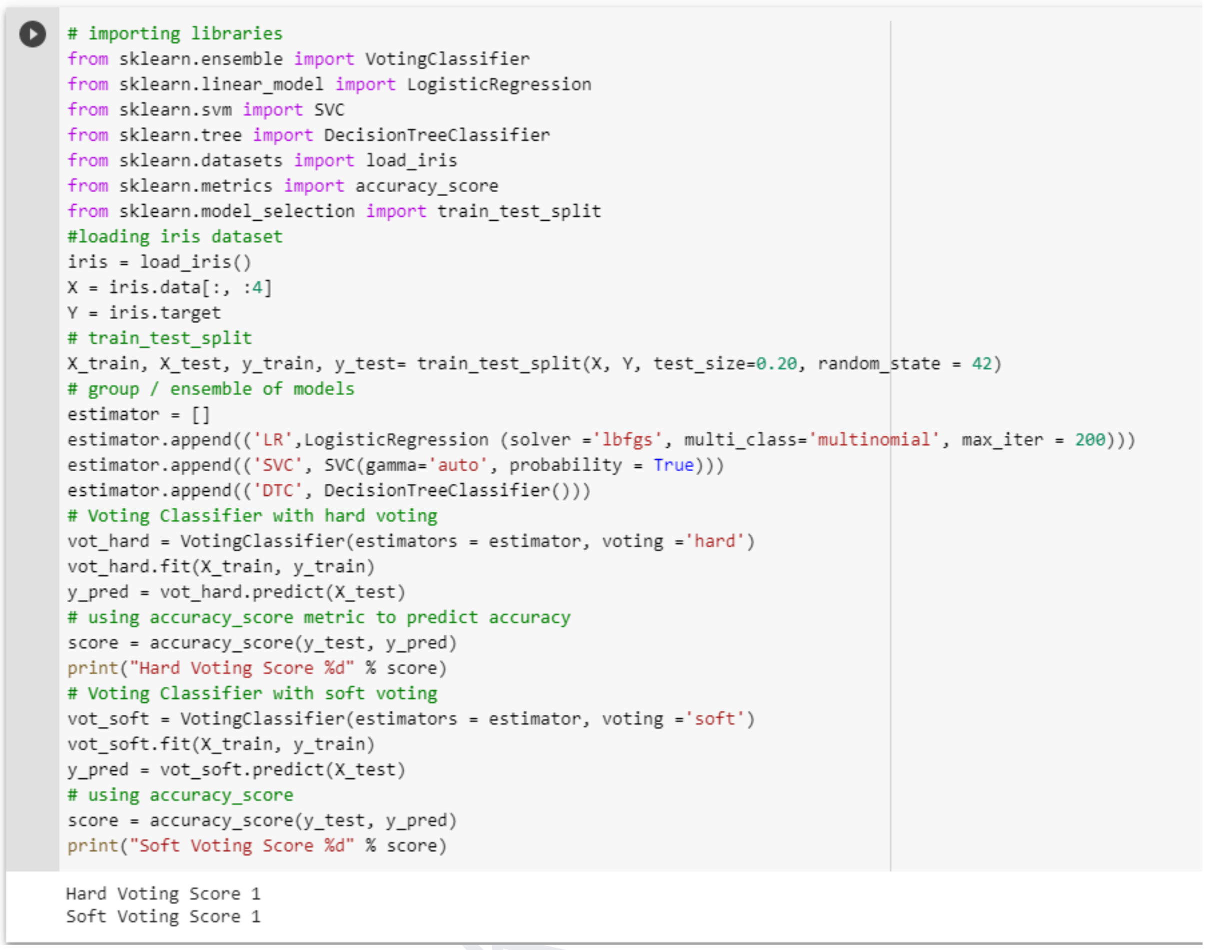
- **Regression Voting Ensemble**: Predictions are the average of contributing models.
- **Classification Voting Ensemble**: Predictions are the majority vote of contributing models.

There are two approaches to the majority vote prediction for classification; they are hard voting and soft voting.

Hard voting involves summing the predictions for each class label and predicting the class label with the most votes. Soft voting involves summing the predicted probabilities (or probability-like scores) for each class label and predicting the class label with the largest probability.

- **Hard Voting**. Predict the class with the largest sum of votes from models
- **Soft Voting**. Predict the class with the largest summed probability from models.

SOURCE CODE:



```
# importing libraries
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
#loading iris dataset
iris = load_iris()
X = iris.data[:, :4]
Y = iris.target
# train_test_split
X_train, X_test, y_train, y_test= train_test_split(X, Y, test_size=0.20, random_state = 42)
# group / ensemble of models
estimator = []
estimator.append(('LR',LogisticRegression (solver ='lbfgs', multi_class='multinomial', max_iter = 200)))
estimator.append(('SVC', SVC(gamma='auto', probability = True)))
estimator.append(('DTC', DecisionTreeClassifier()))
# Voting Classifier with hard voting
vot_hard = VotingClassifier(estimators = estimator, voting ='hard')
vot_hard.fit(X_train, y_train)
y_pred = vot_hard.predict(X_test)
# using accuracy_score metric to predict accuracy
score = accuracy_score(y_test, y_pred)
print("Hard Voting Score %d" % score)
# Voting Classifier with soft voting
vot_soft = VotingClassifier(estimators = estimator, voting ='soft')
vot_soft.fit(X_train, y_train)
y_pred = vot_soft.predict(X_test)
# using accuracy_score
score = accuracy_score(y_test, y_pred)
print("Soft Voting Score %d" % score)
```

```
Hard Voting Score 1
Soft Voting Score 1
```

CONCLUSION:

From this practical, I have successfully learned & implemented boosting algorithms such as: AdaBoost Classifier, Gradient Descent Algorithm, Voting Ensemble Algorithm