Project 2

Team members:

Hari Priya Avarampalayam Manoharan - 002711275 -havarampalayammanoh1@student.gsu.edu
Rida Fathima - 002695685- rfathima1@student.gsu.edu
Sakshi Sachin Agarkar - 002712319 - sagarkar1@student.gsu.edu
Lilia Chebbah - 0027044185 - lchebbah1@student.gsu.edu
Mouayed Lajnef - 002704186 - mlajnef1@student.gsu.edu- (MANAGER)

"Statement of Academic Honesty:

The following code represents our own work. We have neither received nor given inappropriate assistance. We have not copied or modified code from any source other than the course webpage or the course textbook. We recognize that any unauthorized assistance or plagiarism will be handled in accordance with Georgia State University's Academic Honesty Policy and the policies of this course. We recognize that our work is based on an assignment created by the Institute for Insight at Georgia State University. Any publishing or posting of source code for this project is strictly prohibited unless you have written consent from the Institute for Insight at Georgia State University."

1. **Motivation and Problem:**

MARTA (The Metropolitan Atlanta Rapid Transit Authority) is a public transport operator in the Atlanta metropolitan area. MARTA being the 8th largest transit system in the US, operates bus routes linked to a rapid transit system consisting of 48 miles (77 km) of rail track with 38 train stations. According to sources, In 2021, the entire system (bus and rail) had 50,288,800 rides or about 179,600 per weekday in the second quarter of 2022 [1] . It has become the primary mode of transportation thanks to its affordability and connectivity to major areas in Atlanta. However, amidst this, the MARTA schedule suffers from a great problem reported by its daily users which is the frequent delays in the arrival of buses and trains. That led to the loss of trust and satisfaction by its users. Adding to that the fact that customers can sometimes be impacted in their livelihood because the planned bus does arrive on time. One user, in particular, had this to say: "I cannot afford to regularly miss lectures or more severely, exams. If I show up to work late more than six times per year, I will be fired." [2]. This comment along with many reviews on MARTA's trip advisor's page complains about the arrival times of their buses and trains. The problem here is twofold. From the customer's side, MARTA riders do not know if the scheduled trips arrive on time at a particular stop causing them to miss their appointments and commitments. From the organization's side, their scheduling is not accurate, causing them to lose the trust of their riders and lose customers.

2. Proposed Solution

Our proposed solution is to create a time series predictive model that predicts the arrival time of buses and trains at each particular stop. The prediction will include both late and early arrivals of MARTA's viceless. This model can be used by MARTA to update their scheduling times shared with their customers. It can also be used to determine the factors that affect the arrival timings and maintain all updates. That would allow MARTA riders to plan their trips in a more accurate manner. While MARTA can fulfill its duties & obligations as a public service by promoting the interest of the public and providing a more consistent method of transportation. Users will not be put at a major disadvantage in their livelihoods for not being able to afford a car.

3. Data Used

For this project, we used Public "The General Transit Feed Specification (GTFS)" data. Public transport agencies can publish their transit data in a format that can be used by a wide range of software applications according to the General Transit Feed Specification (GTFS), a data specification. Today, tens of thousands of public transportation companies employ the GTFS data format.

A real-time component of GTFS comprises arrival forecasts, vehicle positions, and service advisories in addition to a schedule component that contains schedule, fare, and geographic transit

4. Machine learning Algorthims:

Logistic Regression:

We first chose to use logistic regression and support vector machine because they are the most well-known machine learning techniques for binary classification, which the case of our study. We want to predict if the train is on time or not. But we found that the result of the accuracy is really low. This can be explained by the simplicity of this algorithm that can sometimes not work well with large datasets. So, we chose not to focus on finetuning them since we have other models that outperform significantly these two.

Gradient Boosting Trees:

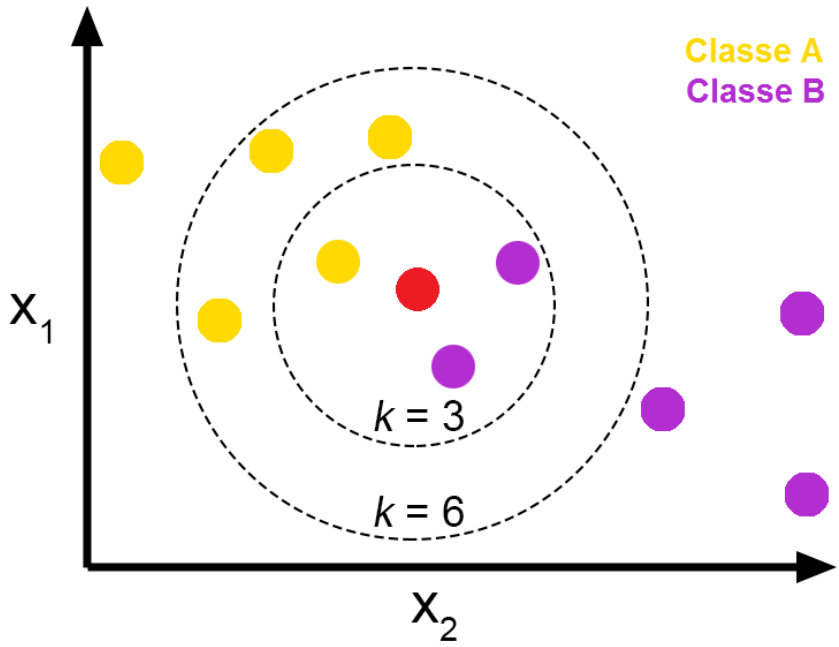
Like random forests, gradient boosting trees use a group of trees to predict the target value. This machine learning technique can be used for both classification and regression. Our case is a binary classification, that's why we used the classifier. In training, instead of creating the trees independently like the random forests, the trees are created sequentially, one after another, to measure each time the residuals and use it for the next tree in order to correct the previous errors. So, the most important parameter for this classifier is the loss function result. The output is then the class that is most predicted by the trees.

Since our classification is binary, we can use the AUC-ROC curve to visualize the performance of our models. The higher the area under the curve the better the model is performing. We used the AUC-ROC curve for the number of estimators, learning rate, minimum sample leaf finetuning for the boosting classifier. We tried to do it for the maximum sample leaf, but it took so long.

KNN Algorithm:

- The k-nearest neighbors (KNN) algorithm is a data classification method used in estimating the likelihood that a data point will become a member of one group or another based on which group the data points nearest to it belong to.
- Here k is the parameter that determines the number of nearest neighbours.

- Classification is a critical problem in data science and machine learning. The KNN is one of the oldest yet accurate algorithms used for pattern classification and regression models.



- k- Nearest Neighbors is one of the most basic algorithms used in supervised machine learning. It classifies new data points based on similarity index which is usually a distance metric The most commonly used metric is Euclidean distance. It uses a majority vote for classifying the new data.
- For example, if there are 2 purple dots and 1 yellow dot near the new data point, it will classify it as a purple one.

Hyperparameter tuning:

A hyperparameter is a parameter of the model that is set before the start of learning process. Different machine learning models have different hyperparameters.

Exhaustive Grid Search technique for hyperparameter optimization is used in our model for better results. An exhaustive grid search takes in as many hyperparameters as you would like, and tries every single possible combination of the hyperparameters as well as many cross-validations as you would like it to perform. It is a good way of determining the best hyperparameter values to use, but it can quickly become time consuming with every additional parameter value and cross-validation added.

The grid search provided by **GridSearchCV** exhaustively generates candidates from a grid of parameter values specified with the param_grid parameter. The **GridSearchCV** instance implements the usual estimator API: when "fitting" it on a dataset all the possible combinations of parameter values are evaluated and the best combination is retained.

In other words, the Sklearn function **GridSearchCV()** can be used to run multiple models in order to select the best combination of hyperparameters for a particular data set.

Why KNN on MARTA dataset:

- KNN was picked because, it is one of the accurate algorithms used for pattern classification and regression models.
- In KNN, the quality of the predictions depends on the distance measure. There is enough information in the MARTA Dataset, thereby allowing us to make a selection of an appropriate measure

Work Flow:

Steps Followed:

For without Tuning model:

1. Loaded the training and test dataset.
2. Chose the value of K randomly. K can be any integer.
3. Checked the results.

For Hyperparameter tuning model:

1. Checked for the best k values that can give a good accuracy score.
2. Applied various values of k, weights and metric on the model, to come up with best parameters to be used
3. Retrieved the best parameters and substituted them
4. Checked the results.

Implementation:

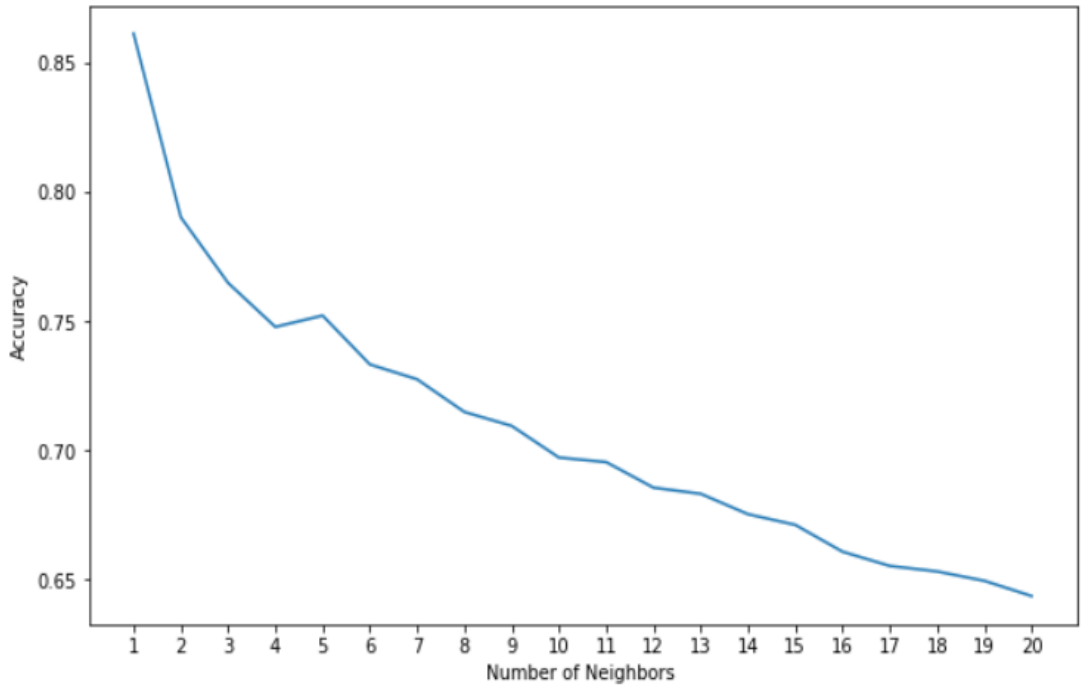
```
#training the model
neigh = KNeighborsClassifier(n_neighbors=5)
neigh.fit(X_train, y_train)

#predict
pred_y = neigh.predict(X_test)

#confusion matrix
cm =confusion_matrix(y_test, pred_y)
print(cm)
print(classification_report(y_test, pred_y))



print('Accuracy: ',round(.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
[[3064  922]
 [1084 2930]]
              precision    recall  f1-score   support

           0       0.74      0.77      0.75      3986
           1       0.76      0.73      0.74      4014

    accuracy                           0.75      8000
   macro avg       0.75      0.75      0.75      8000
weighted avg       0.75      0.75      0.75      8000
```

Number of Neighbors v/s Accuracy

```
#plot
loc = np.arange(1,21,step=1.0)
plt.figure(figsize = (10, 6))
plt.plot(range(1,21), mean_acc)
plt.xticks(loc)
plt.xlabel('Number of Neighbors ')
plt.ylabel('Accuracy')
plt.show()
```



After Hyperparameter tuning

```
# your code

knn_params ={
    "n_neighbors":[3,5,7],
    "weights" : ["uniform","distance"],
    "metric" : ["euclidean", "manhattan", "minkowski"],

}

knn = KNeighborsClassifier()

#grid search

gs = GridSearchCV(KNeighborsClassifier(), knn_params, verbose = 1, cv=3, n_jobs = -1)
g_res = gs.fit(X_train, y_train)
```

For n_neighbors, when dealing with a two-class problem, it's better to choose an odd value for K. Otherwise, a scenario can arise where the number of neighbors in each class is the same. Also, the value of K must not be a multiple of the number of classes present

```
#Retrieving the best parameters
g_res.best_params_
```

```
{'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'distance'}
```

So, GridSearchCV() has determined that **n_neighbors = 3**, **weights = 'distance'** and **metric = 'Manhattan'** is the best set of hyperparameters to use for this data.

```
# substituting the best parameters
knn = KNeighborsClassifier(n_neighbors = 3, weights = 'distance',algorithm = 'brute',metric = 'manhattan')
knn.fit(X_train, y_train)
```

```
# getting a prediction
y_hat = knn.predict(X_train)
y_pred = knn.predict(X_test)
```

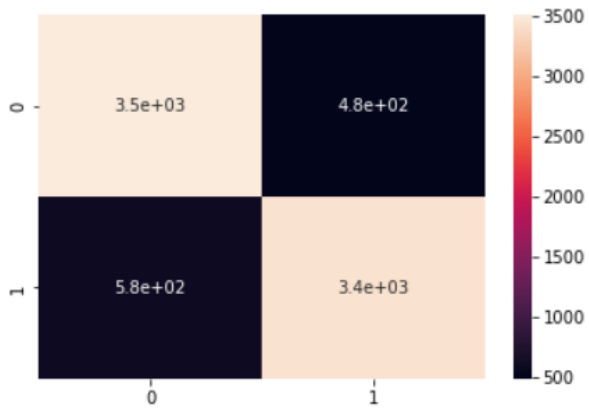
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.86      0.88      0.87      3986
           1       0.88      0.85      0.87      4014

    accuracy                           0.87      8000
   macro avg       0.87      0.87      0.87      8000
weighted avg       0.87      0.87      0.87      8000
```

Confusion Matrix plot:

```
#heatmap of confusion matrix
cm=confusion_matrix(y_test, y_pred)
print(cm)
import seaborn as sns
sns.heatmap(cm, annot=True)

print('Accuracy: ',round(.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

```
[[3504  482]
 [ 583 3431]]
Accuracy:  0.867

Precision:  0.879
```



Comparison:

Tuned model has given better result than the untuned model.

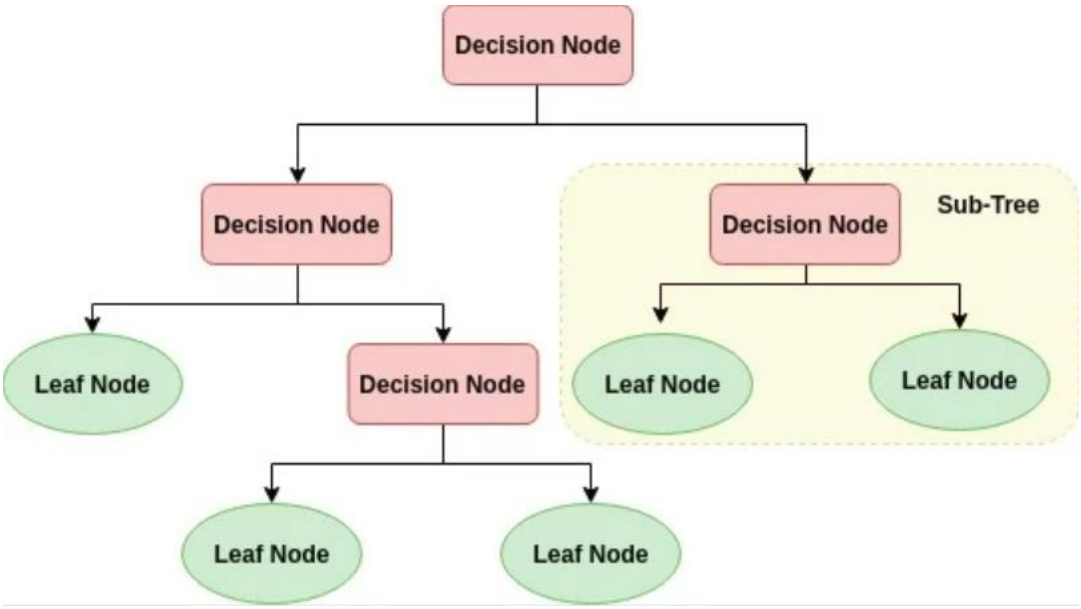
Challenges:

1. Time consuming
2. Sensitive to irrelevant features
3. Need to carefully choose the k value

Decision Trees:

A decision tree is a flowchart-like tree structure where the internal node represents an attribute, the branch represents a decision rule, and the leaf node represents an outcome. The node at the top of the decision tree is referred to as the root node. The root node basically learns to partition on the basis of the attribute value into a sub-tree as depicted in the figure below. This flowchart kind of structure aids in the process of making decisions. The flowchart like process flow makes the process for humans to visualize, hence making decisions easy to understand and interpret.

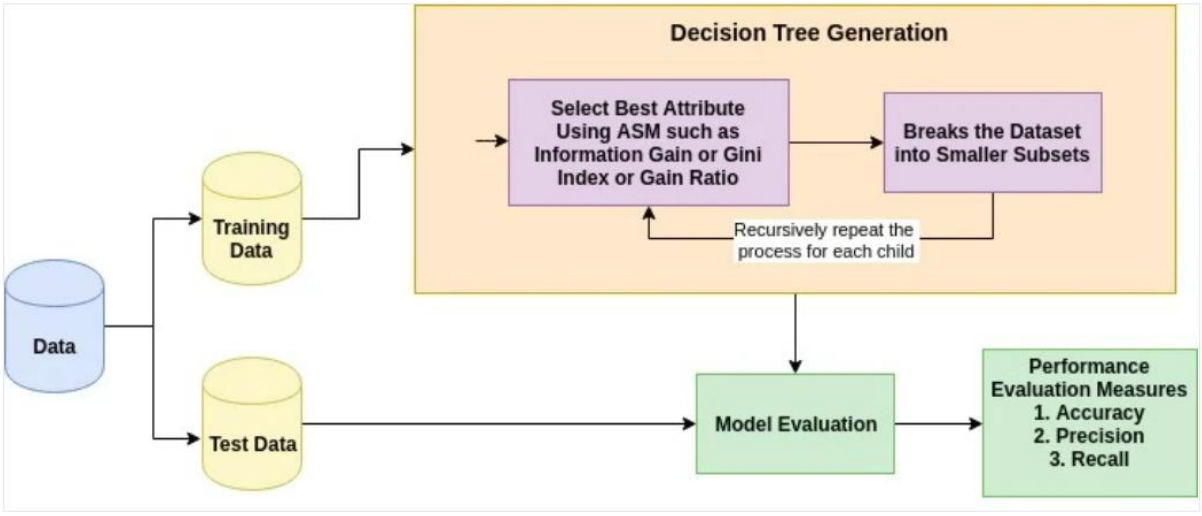
Decision Tree Model:



Work Flow:

The concept behind any decision tree algorithm can be summarised as follows:

1. Selection of the best attribute/feature using Attribute Selection Measures(ASM) to partition the records
2. The identified attribute is made into a decision node and the dataset is further broken down into smaller subsets.
3. The process is recursively repeated and new trees are built each time, this process keeps on occurring till either of the below conditions are satisfied:
    - All the tuples belong to the same attribute value.
    - There are no more remaining attributes.
    - There are no more instances.



Attribute Selection Measures:

During the process of partitioning the data in the best possible way, Attribute Selection Measure (ASM) is a heuristic that is employed for picking the best splitting criteria

GiniIndex:

The gini impurity basically measures the frequency at which any element of the dataset will be mislabelled when it is randomly labeled. For the split to be optimum, attributes with a low Gini index are selected. Moreover, it gets the maximum value when the probability of the two classes is the same.

The below-depicted formula can be used to calculate entropy:

$$GiniIndex = 1 - \sum_j p_j^2$$

Where p_j is the probability of class j.

Entopy:

Entropy can be described as a measure of information that basically depicts the disorder of the features with the target. Just like we saw in the case of the Gini index, a lower Gini index meant optimal split here also, the optimum split is chosen by the feature with less entropy. It obtains its maximum value when the probability of the two classes is the same and a node is considered to be pure when the entropy has its minimum value, i.e. 0:

Entropy can be obtained by using the below-depicted formula:

$$Entropy = -\sum_j p_j \cdot log_2 \cdot p_j$$

Where, as before, p_j is the probability of class j.

Information Gain:

Information gain can be simply described as a measure of the amount of information a feature provides about a class. Information gain is very useful in classifying the order of attributes in the nodes of a decision tree. The main node is referred to as the parent node, whereas sub-nodes are known as child nodes. The information gain can be very useful in understanding how well the splitting of nodes in a decision tree has been executed. In other words, information gain helps us understand the quality of splitting. The following method to calculate information gain further helps clarify the concept

$$\text{Gain} = E_{\text{parent}} - E_{\text{children}}$$

Gain here depicts information gain. E_{parent} is the entropy of the parent node while $E_{children}$ is the average entropy of the child nodes.

Why Decision Tree on MARTA dataset?

- Decision Trees are non-linear classifiers, the missing values in the data do not affect the process of building a decision tree and also it handles skewed classes which suits well for our dataset which is non-linear and has missing values.
- Decision Tree was picked because compared to other algorithms it requires less effort for data preparation during pre-processing and it is not necessary to normalize or scale the data.
- Moreover, the decision tree model is very intuitive and makes it easier to explain to all stakeholders including technical teams.

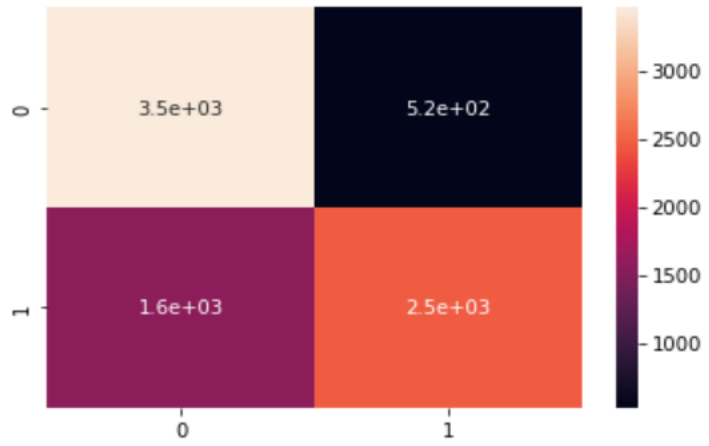
Implementation for entropy:

```
#Defining decision tree classifier
dtree = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=42)

#training the data
dtree.fit(X_train, y_train)

#testing the data
y_pred2 = dtree.predict(X_test)


# an evaluation of the model
print('Evaluation')
print('Report: \n',classification_report(y_test,y_pred), '\n')
#Confusion Matrix computation
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix: \n' , cm, '\n')
print('Accuracy: ',round(dtree.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
Evaluation
Report:
              precision    recall  f1-score   support

           0       0.69      0.87      0.77      3986
           1       0.82      0.61      0.70      4014

    accuracy                           0.74      8000
   macro avg       0.76      0.74      0.74      8000
weighted avg       0.76      0.74      0.74      8000


Confusion Matrix:
 [[3461  525]
 [1557 2457]]

Accuracy:  0.74

Precision:  0.868
```

Confusion Matrix plot:



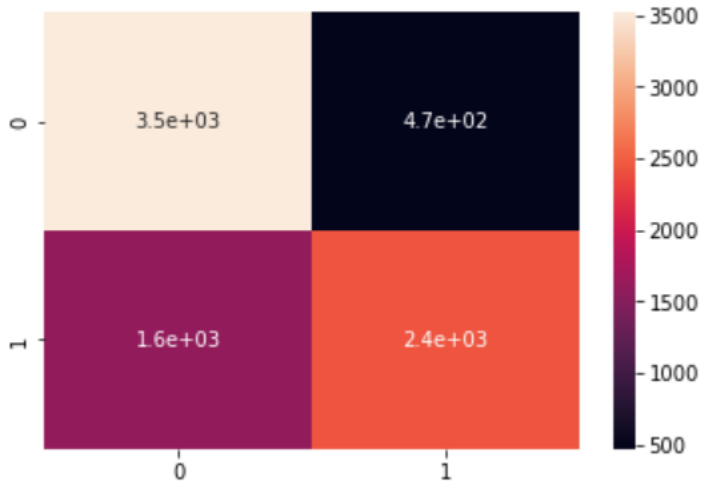
Implementation for GINI:

```
# your code

#Defining decision tree classifier
dtree = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)

#training the data
dtree.fit(X_train, y_train)

#testing the data
y_pred = dtree.predict(X_test)


# an evaluation of the model
print('Evaluation')
print('Report: \n',classification_report(y_test,y_pred), '\n')
#Confusion Matrix computation
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix: \n' , cm, '\n')
print('Accuracy: ',round(dtree.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
Evaluation
Report:
              precision    recall  f1-score   support

           0       0.75      0.83      0.79      3996
           1       0.81      0.72      0.76      4004

    accuracy                           0.78      8000
   macro avg       0.78      0.78      0.78      8000
weighted avg       0.78      0.78      0.78      8000


Confusion Matrix:
 [[3336  660]
 [1123 2881]]

Accuracy:  0.753

Precision:  0.835
```

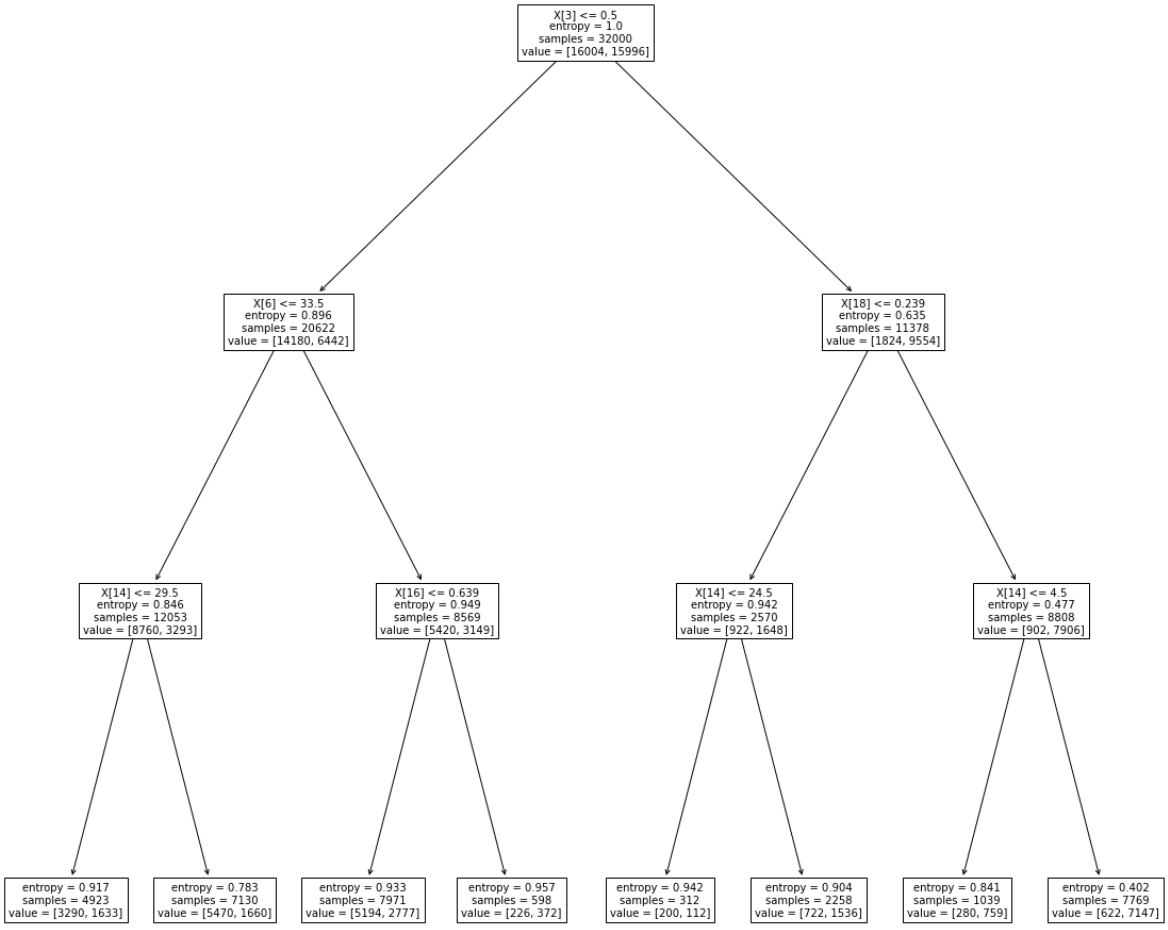
Confusion Matrix plot:



Entropy calculation and Feature Selection:

```
[Text(0.5, 0.875, 'X[3] <= 0.5\nentropy = 1.0\nsamples = 32000\nvalue = [16004, 15996]'),
 Text(0.25, 0.625, 'X[6] <= 33.5\nentropy = 0.896\nsamples = 20622\nvalue = [14180, 6442]'),
 Text(0.125, 0.375, 'X[14] <= 29.5\nentropy = 0.846\nsamples = 12053\nvalue = [8760, 3293]'),
 Text(0.0625, 0.125, 'entropy = 0.917\nsamples = 4923\nvalue = [3290, 1633]'),
 Text(0.1875, 0.125, 'entropy = 0.783\nsamples = 7130\nvalue = [5470, 1660]'),
 Text(0.375, 0.375, 'X[16] <= 0.639\nentropy = 0.949\nsamples = 8569\nvalue = [5420, 3149]'),
 Text(0.3125, 0.125, 'entropy = 0.933\nsamples = 7971\nvalue = [5194, 2777]'),
 Text(0.4375, 0.125, 'entropy = 0.957\nsamples = 598\nvalue = [226, 372]'),
 Text(0.75, 0.625, 'X[18] <= 0.239\nentropy = 0.635\nsamples = 11378\nvalue = [1824, 9554]'),
 Text(0.625, 0.375, 'X[14] <= 24.5\nentropy = 0.942\nsamples = 2570\nvalue = [922, 1648]'),
 Text(0.5625, 0.125, 'entropy = 0.942\nsamples = 312\nvalue = [200, 112]'),
 Text(0.6875, 0.125, 'entropy = 0.904\nsamples = 2258\nvalue = [722, 1536]'),
 Text(0.875, 0.375, 'X[14] <= 4.5\nentropy = 0.477\nsamples = 8808\nvalue = [902, 7906]'),
 Text(0.8125, 0.125, 'entropy = 0.841\nsamples = 1039\nvalue = [280, 759]'),
 Text(0.9375, 0.125, 'entropy = 0.402\nsamples = 7769\nvalue = [622, 7147]')]
```

Decision Tree for MARTA data:



Regularization:

Regularization can be described as the process of reducing overfitting. Regularization is carried out differently depending on different regression models. But for a decision tree classifier, regularization can be executed by providing additional arguments accepted by the Decision Tree Classifier. The arguments provided to revamp the model are called **Hyperparameters.**

Hyperparameter tuning in Decision Trees:

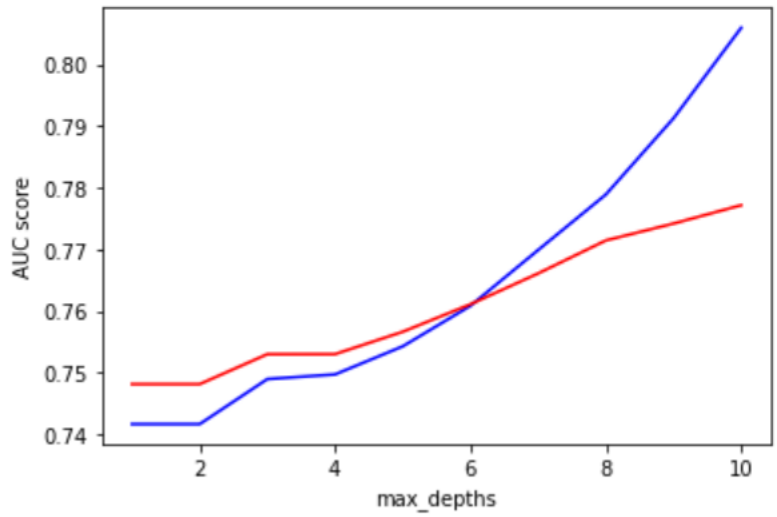
This method employed to calibrate our model by discovering the right hyperparameters to generalize our model is called Hyperparameter Tuning.

a. Max Depth

This represents tuning the maximum depth of a tree.

Implementation:

```
max_depths = np.linspace(1, 10, 10, endpoint=True)
train_results = []
test_results = []
for max_depth in max_depths:
   model = DecisionTreeClassifier(max_depth=int(max_depth))
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train, train_pred)
   roc_auc = auc(false_positive_rate, true_positive_rate)
   train_results.append(roc_auc)
   y_pred = model.predict(X_test)
   false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
   roc_auc = auc(false_positive_rate, true_positive_rate)
   test_results.append(roc_auc)

line1 = plt.plot(max_depths, train_results, 'b' , label='Train AUC')
line2 = plt.plot(max_depths, test_results, 'r' , label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('max_depths')
plt.show()
```



Implementation of Decision tree with hypertuning:

```
# your code

#Defining decision tree classifier
dtree = DecisionTreeClassifier(criterion='entropy', max_depth=10, random_state=0)

#training the data
dtree.fit(X_train, y_train)

#testing the data
y_pred2 = dtree.predict(X_test)


# an evaluation of the model
print('Evaluation')
print('Report: \n',classification_report(y_test,y_pred), '\n')
#Confusion Matrix computation
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix: \n' , cm, '\n')
print('Accuracy: ',round(dtree.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
Evaluation
Report:
              precision    recall  f1-score   support

           0       0.75      0.83      0.79      3996
           1       0.81      0.72      0.76      4004

    accuracy                           0.78      8000
   macro avg       0.78      0.78      0.78      8000
weighted avg       0.78      0.78      0.78      8000


Confusion Matrix:
 [[3336  660]
 [1123 2881]]

Accuracy:  0.779

Precision:  0.835
```

Comparison:

The tuned model provides better accuracy in comparison to the untuned model. However, precision in both cases remains the same.

Disadvantages:

- Decision Trees are unstable because even a minute change in the data can cause a significant change in the whole structure of the decision tree.
- For a Decision tree, sometimes it is possible that the calculation becomes far more complex compared to the other algorithms.
- A decision tree often involves an investment of a higher amount of time to train the model.
- The Decision Tree algorithm falls short when applied to regression and predicting continuous values.

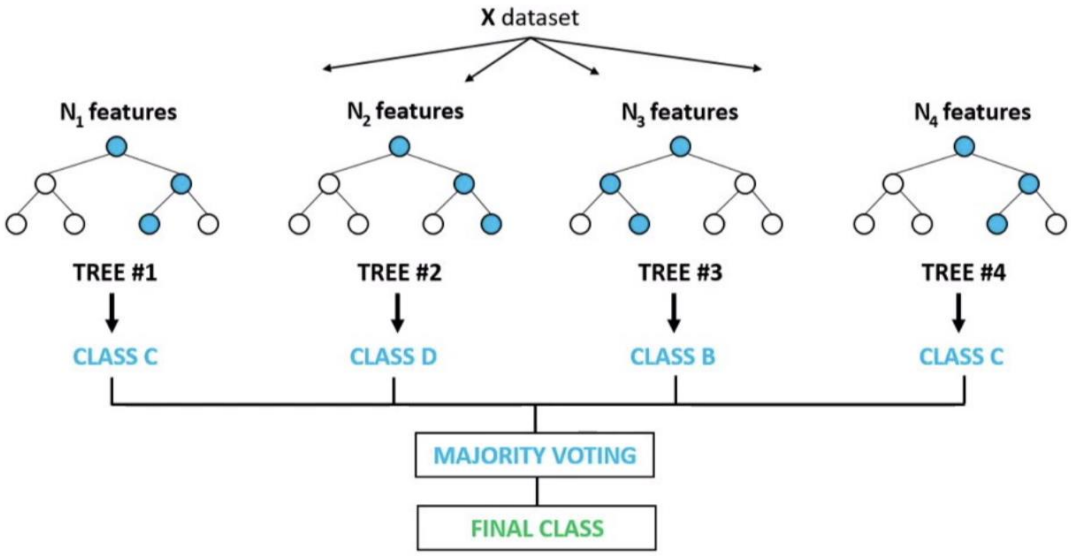
Random Forest Classifier:

Random forest is a supervised learning algorithm. It has two variations – one is used for classification problems and other is used for regression problems. It is one of the most flexible and easy to use algorithm. It creates decision trees on the given data samples, gets prediction from each tree and selects the best solution by means of voting. It is also a pretty good indicator of feature importance.

Random forest algorithm combines multiple decision-trees, resulting in a forest of trees, hence the name Random Forest. In the random forest classifier, the higher the number of trees in the forest results in higher accuracy.

Decision Tree Model:

Random Forest Classifier



Work Flow:

Step 1: In Random forest n number of random records are taken from the data set having k number of records.

Step 2: Individual decision trees are constructed for each sample.

Step 3: Each decision tree will generate an output.

Step 4: Final output is considered based on ***Majority Voting or Averaging*** for Classification and regression respectively.

Attribute Selection Measures:

Attribute selection measure is a heuristic for selecting the splitting criterion that partition data into the best possible manner.

GiniIndex:

The gini impurity measures the frequency at which any element of the dataset will be mislabelled when it is randomly labeled. The optimum split is chosen by the features with less Gini Index. Moreover, it gets the maximum value when the probability of the two classes are the same.

Gini is calculated using the following formula.

$$GiniIndex = 1 - \sum_j p_j^2$$

Where p_j is the probability of class j.

Entopy:

Entropy is a measure of information that indicates the disorder of the features with the target. Similar to the Gini Index, the optimum split is chosen by the feature with less entropy. It gets its maximum value when the probability of the two classes is the same and a node is pure when the entropy has its minimum value, which is 0:

Entropy is calculated using the following formula.

$$Entropy = -\sum_j p_j \cdot log_2 \cdot p_j$$

Where, as before, p_j is the probability of class j.

Information Gain:

We can define information gain as a measure of how much information a feature provides about a class. Information gain helps to determine the order of attributes in the nodes of a decision tree.The main node is referred to as the parent node, whereas sub-nodes are known as child nodes. We can use information gain to determine how good the splitting of nodes in a decision tree.It can help us determine the quality of splitting, as we shall soon see. The calculation of information gain should help us understand this concept better.

$$Gain = E_{parent} - E_{children}$$

The term Gain represents information gain. Eparent is the entropy of the parent node and E_{children} is the average entropy of the child nodes.

Why Random Forest on MARTA dataset?

- Random forests work well for a large range of data items than a single decision tree does.
- Random Forest algorithms maintains good accuracy even a large proportion of the data is missing.
- It overcomes the problem of overfitting by averaging or combining the results of different decision trees.
- It overcomes the problem of overfitting by averaging or combining the results of different decision trees.

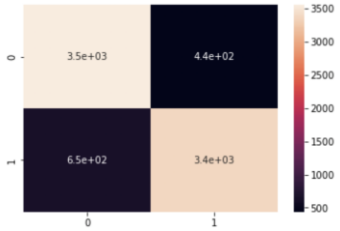
Implementation for entropy:

```
: # your code
#splitting data into target and data
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(criterion='entropy',n_estimators=100)

#Train the model using the training sets y_pred=clf.predict(X_test)
clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

# an evaluation of the model
print('Evaluation')
print('Report: \n',classification_report(y_test,y_pred), '\n')
#Confusion Matrix computation
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix: \n' , cm, '\n')
print('Accuracy: ',round(clf.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
Evaluation
Report:
              precision    recall  f1-score   support

           0       0.85      0.89      0.87      3986
           1       0.88      0.84      0.86      4014

    accuracy                           0.86      8000
   macro avg       0.87      0.86      0.86      8000
weighted avg       0.87      0.86      0.86      8000


Confusion Matrix:
 [[3548  438]
 [ 647 3367]]

Accuracy:  0.864

Precision:  0.89
```

Confusion Matrix plot:



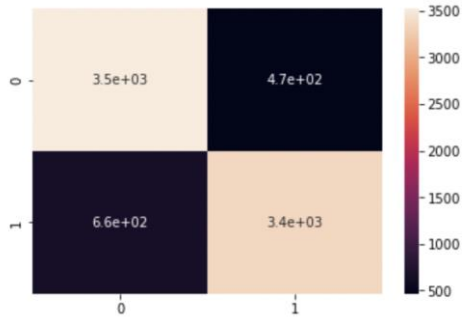
Implementation for GINI:

```python
# your code
#splitting data into target and data
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(criterion='gini',n_estimators=100)

#Train the model using the training sets y_pred=clf.predict(X_test)
clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

# an evaluation of the model
print('Evaluation')
print('Report: \n',classification_report(y_test,y_pred), '\n')
#Confusion Matrix computation
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix: \n' , cm, '\n')
print('Accuracy: ',round(clf.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
Evaluation
Report:
              precision    recall  f1-score   support

           0       0.84      0.88      0.86      3986
           1       0.88      0.83      0.86      4014

    accuracy                           0.86      8000
   macro avg       0.86      0.86      0.86      8000
weighted avg       0.86      0.86      0.86      8000


Confusion Matrix:
 [[3520  466]
 [ 663 3351]]

Accuracy:  0.859

Precision:  0.883
```

Confusion Matrix plot:



Hyperparameter tuning in Random Forest:

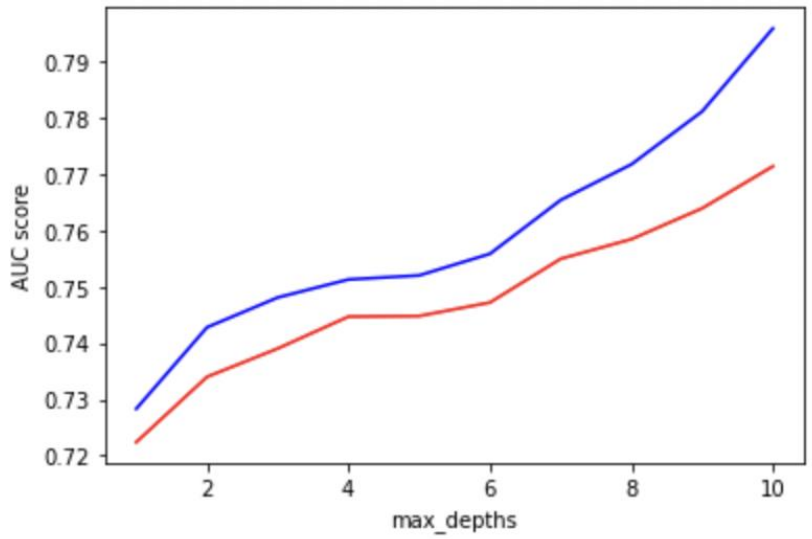
This process of calibrating our model by finding the right hyperparameters to generalize our model is called Hyperparameter Tuning.

a. Max Depth

This represents tuning the maximum depth of a tree.

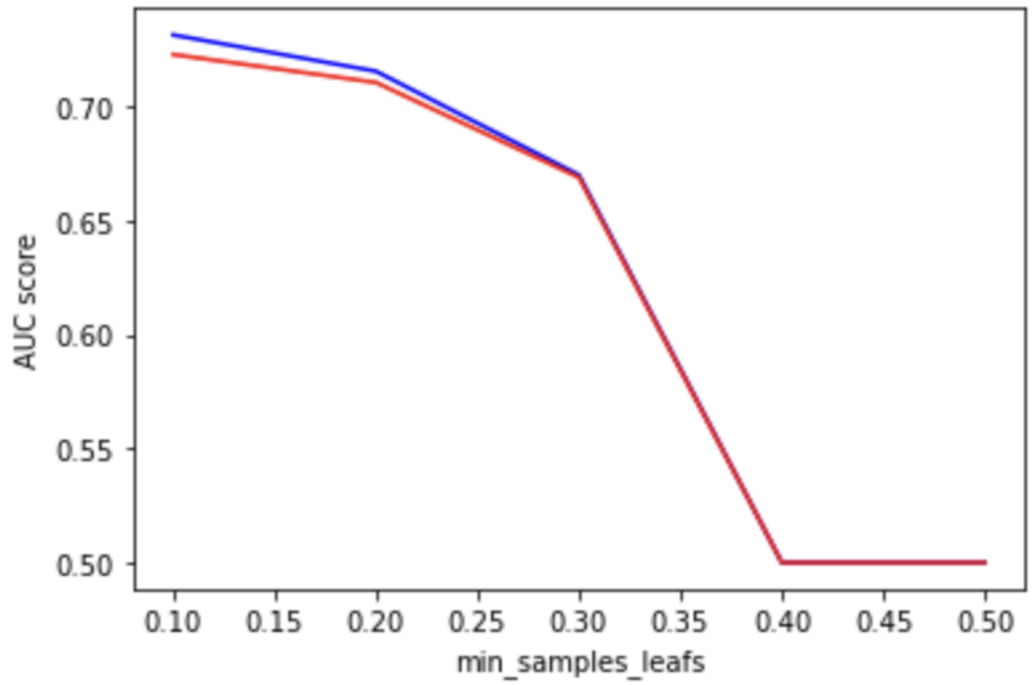
Implementation:

```
# your code
max_depths = np.linspace(1, 10, 10, endpoint=True)
train_results = []
test_results = []
for max_depth in max_depths:
   model = RandomForestClassifier(max_depth=int(max_depth))
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,
   roc_auc = auc(false_positive_rate, true_positive_rate)
   train_results.append(roc_auc)
   y_pred = model.predict(X_test)
   false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,
   roc_auc = auc(false_positive_rate, true_positive_rate)
   test_results.append(roc_auc)

line1 = plt.plot(max_depths, train_results, 'b' , label='Train AUC')
line2 = plt.plot(max_depths, test_results, 'r' , label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('max_depths')
plt.show()
```



b. min_samples_leaf

```
# your code
min_samples_leafs = np.linspace(0.1, 0.5, 5, endpoint=True)
train_results = []
test_results = []
for min_samples_leaf in min_samples_leafs:
   model = RandomForestClassifier(min_samples_leaf=min_samples_leaf)
   model.fit(X_train, y_train)
   train_pred = model.predict(X_train)
   false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,
   roc_auc = auc(false_positive_rate, true_positive_rate)
   train_results.append(roc_auc)
   y_pred = model.predict(X_test)
   false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,
   roc_auc = auc(false_positive_rate, true_positive_rate)
   test_results.append(roc_auc)

line1 = plt.plot(min_samples_leafs, train_results, 'b' , label='Train AUC')
line2 = plt.plot(min_samples_leafs, test_results, 'r' , label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('min_samples_leafs')
plt.show()
```



Implementation of Random Forest with hypertuning:

```
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(criterion='entropy',n_estimators=800,max_depth=32,
 min_samples_leaf=1,random_state=42)

#Train the model using the training sets y_pred=clf.predict(X_test)
clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

# an evaluation of the model
print('Evaluation')
print('Report: \n',classification_report(y_test,y_pred), '\n')
#Confusion Matrix computation
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix: \n' , cm, '\n')
print('Accuracy: ',round(clf.score(X_test, y_test),3), '\n')
print('Precision: ',round(cm[0,0]/(cm[0,1]+cm[0,0]),3), '\n')
```

Results:

```
Evaluation
Report:
               precision    recall  f1-score   support

           0       0.85      0.88      0.86      3986
           1       0.88      0.84      0.86      4014

    accuracy                           0.86      8000
   macro avg       0.86      0.86      0.86      8000
weighted avg       0.86      0.86      0.86      8000


Confusion Matrix:
 [[3523  463]
 [ 642 3372]]

Accuracy:  0.862

Precision:  0.884
```

Comparison:

Tuned model gives almost same accuracy compared to untuned model. However, precision for the untune model is more.

Disadvantages:

- Complexity is the main disadvantage of Random forest algorithms.
- Construction of Random forests are much harder and time-consuming than decision trees.
- More computational resources are required to implement Random Forest algorithm.
- It is less intuitive in case when we have a large collection of decision trees.
- The prediction process using random forests is very time-consuming in comparison with other algorithms.

5. Result and Discussion:

ML Model	Accuracy	Precision	F1-Score
Support Vector Machine	0.52	0.52	0.52
K Nearest Neighbors	0.87	0.87	0.87
Decision Trees	0.773	0.884	0.75
Gradient Boosting Classifier	0.83	0.83	0.83
Random forest	0.86	0.86	0.86

Hence from the above scores, the best model to be chosen is KNN.

References:

https://www.datacamp.com/tutorial/decision-tree-classification-python

https://www.section.io/engineering-education/entropy-information-gain-machine-learning/#:~:text=We%20can%20define%20information%20gain,are%20known%20as%20child%20nodes.

https://quantdare.com/decision-trees-gini-vs-entropy/

https://www.section.io/engineering-education/hyperparmeter-tuning/

https://corporatefinanceinstitute.com/resources/data-science/bagging-bootstrap-aggregation/

https://learn.g2.com/k-nearest-neighbor

https://www.tutorialspoint.com/machine_learning_with_python/machine_learning_with_python_knn_algorithm_finding_nearest_neighbors.htm

https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d

https://dev.to/balapriya/hyperparameter-tuning-understanding-grid-search-2648

https://www.geeksforgeeks.org/hyperparameters-of-random-forest-classifier/

https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

https://machinelearningmastery.com/types-of-classification-in-machine-learning/

https://ruslanmv.com/blog/The-best-binary-Machine-Learning-Model