Import Necessary Libraries

Cricket T20 dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Loading Dataset

```
df = pd.read_csv('/content/Cricket T20.csv')
```

```
df.head()
```

	Mat	Inns	NO	Runs	HS	Ave	BF	SR	100	50	0	4s	6s
0	75	70	20	2633	94	52.66	1907	138.07	0	24	2	247	71
1	104	96	14	2633	118	32.10	1905	138.21	4	19	6	234	120
2	83	80	7	2436	105	33.36	1810	134.58	2	15	2	215	113
3	111	104	30	2263	75	30.58	1824	124.06	0	7	1	186	61
4	71	70	10	2140	123	35.66	1571	136.21	2	13	3	199	91

Next steps: Generate code with df View recommended plots

Understanding dataset

```
format(df.shape) # Displaying the shape of the DataFrame
```

```
'(99, 13)'
```

```
df.info() # Displaying information about the DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 99 entries, 0 to 98
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Mat     99 non-null     int64  
 1   Inns    99 non-null     int64  
 2   NO      99 non-null     int64  
 3   Runs    99 non-null     int64  
 4   HS      99 non-null     int64  
 5   Ave     99 non-null     float64
 6   BF      99 non-null     int64  
 7   SR      99 non-null     float64
 8   100     99 non-null     int64  
 9   50      99 non-null     int64  
 10  0       99 non-null     int64  
 11  4s      99 non-null     int64  
 12  6s      99 non-null     int64  
dtypes: float64(2), int64(11)
memory usage: 10.2 KB
```

```
df.describe() # Generating descriptive statistics for the DataFrame
```

	Mat	Inns	NO	Runs	HS	Ave	B
count	99.000000	99.000000	99.000000	99.000000	99.000000	99.000000	99.00000
mean	56.969697	53.111111	8.363636	1238.737374	87.010101	28.104242	963.19191
std	20.017571	18.432939	6.576791	475.251096	21.837757	6.458072	353.80407
min	23.000000	23.000000	0.000000	629.000000	50.000000	17.190000	476.00000
25%	38.000000	36.000000	4.000000	815.000000	72.500000	23.515000	656.00000
50%	58.000000	53.000000	7.000000	1167.000000	85.000000	27.410000	893.00000
75%	72.000000	65.500000	11.000000	1556.500000	100.000000	31.285000	1187.50000
max	111.000000	104.000000	42.000000	2633.000000	172.000000	52.660000	1907.00000

```
df.loc[:,:] # Accessing all rows and columns of the DataFrame using the .loc indexer
```

	Mat	Inns	NO	Runs	HS	Ave	BF	SR	100	50	0	4s	6s
0	75	70	20	2633	94	52.66	1907	138.07	0	24	2	247	71
1	104	96	14	2633	118	32.10	1905	138.21	4	19	6	234	120
2	83	80	7	2436	105	33.36	1810	134.58	2	15	2	215	113
3	111	104	30	2263	75	30.58	1824	124.06	0	7	1	186	61
4	71	70	10	2140	123	35.66	1571	136.21	2	13	3	199	91
...	...	...	...	...	...	...	...	...	...	...	...	...	...
94	23	23	5	671	107	37.27	476	140.96	1	4	0	54	31
95	25	23	4	666	73	35.05	558	119.35	0	5	0	56	20
96	43	38	5	638	57	19.33	524	121.75	0	1	2	60	18
97	71	50	16	636	55	18.70	551	115.42	0	1	3	51	19
98	31	30	3	629	88	23.29	487	129.15	0	4	4	76	23

99 rows × 13 columns

```
df.columns # Accessing the column names of the DataFrame
```

```
Index(['Mat', 'Inns', 'NO', 'Runs', 'HS', 'Ave', 'BF', 'SR', '100', '50', '0',
       '4s', '6s'],
      dtype='object')
```

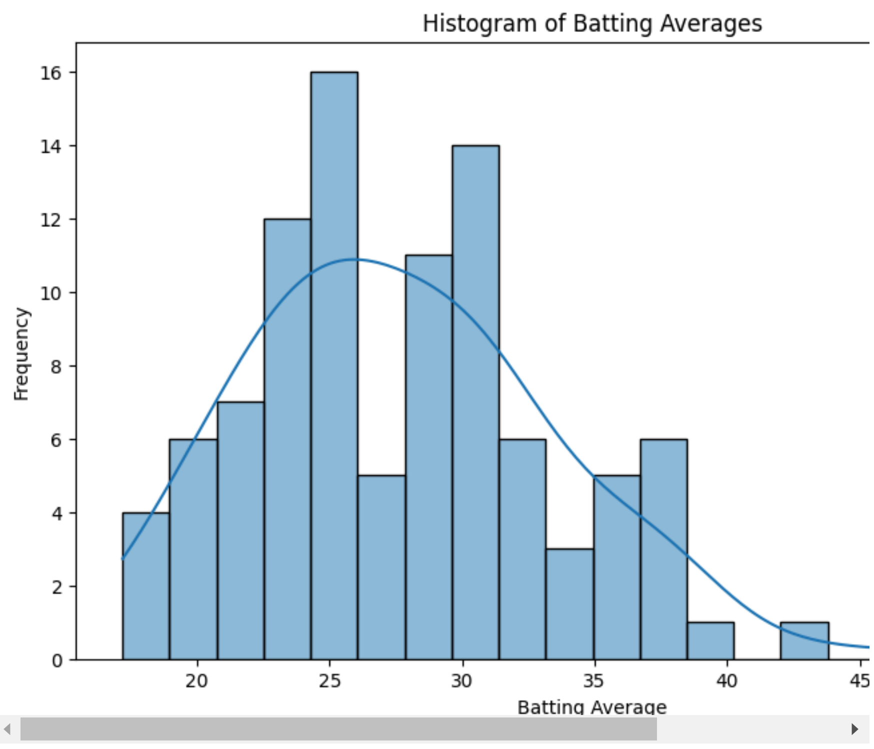
Cleaning DataSet

```
#finding NULL instances
df.isnull().sum()
```

```
Mat     0
Inns    0
NO      0
Runs    0
HS      0
Ave     0
BF      0
SR      0
100     0
50      0
0       0
4s      0
6s      0
dtype: int64
```

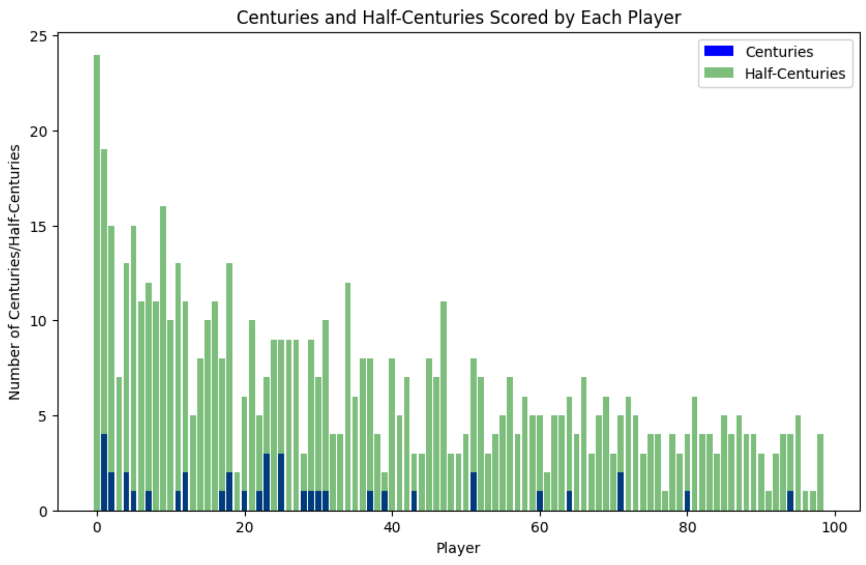
Data Visualization

```
# Histogram of Batting Averages
plt.figure(figsize=(10, 6))
sns.histplot(data=df, x='Ave', bins=20, kde=True)
plt.title('Histogram of Batting Averages')
plt.xlabel('Batting Average')
plt.ylabel('Frequency')
plt.show()
```



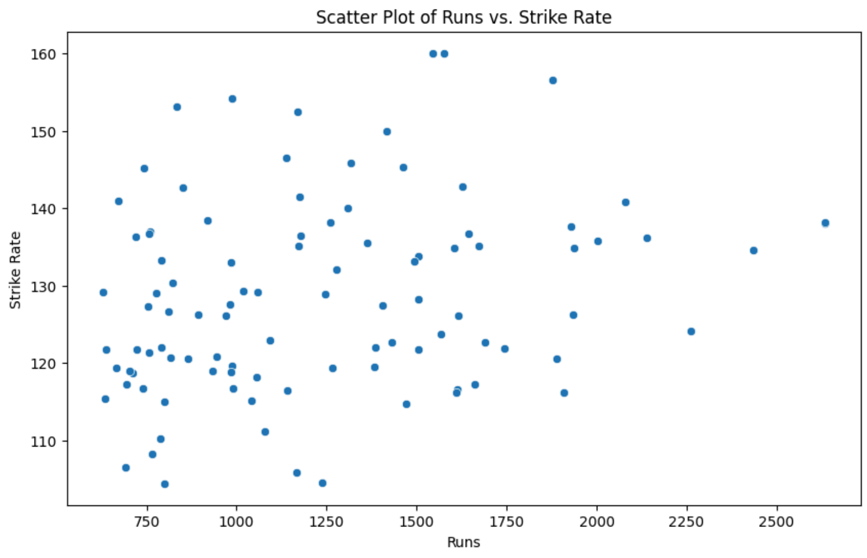
Histogram of Batting Averages: This will give us an overview of the distribution of batting averages across the players.

```
# Bar Chart of Centuries and Half-Centuries
plt.figure(figsize=(10, 6))
centuries = df['100']
half_centuries = df['50']
index = range(len(centuries))
plt.bar(index, centuries, color='blue', label='Centuries')
plt.bar(index, half_centuries, color='green', label='Half-Centuries', alpha=0.5)
plt.xlabel('Player')
plt.ylabel('Number of Centuries/Half-Centuries')
plt.title('Centuries and Half-Centuries Scored by Each Player')
plt.legend()
plt.show()
```



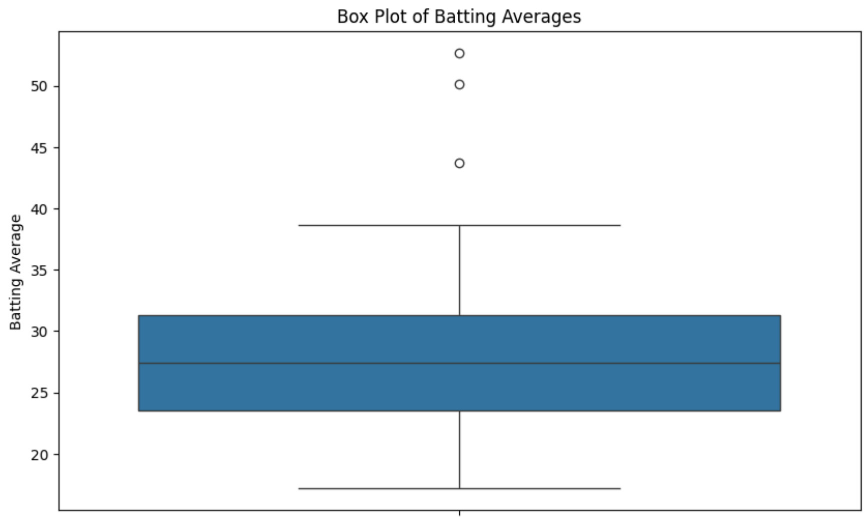
Bar Chart of Centuries and Half-Centuries: We can visualize the number of centuries and half-centuries scored by each player.

```
# Scatter Plot of Runs vs. Strike Rate
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Runs', y='SR')
plt.title('Scatter Plot of Runs vs. Strike Rate')
plt.xlabel('Runs')
plt.ylabel('Strike Rate')
plt.show()
```



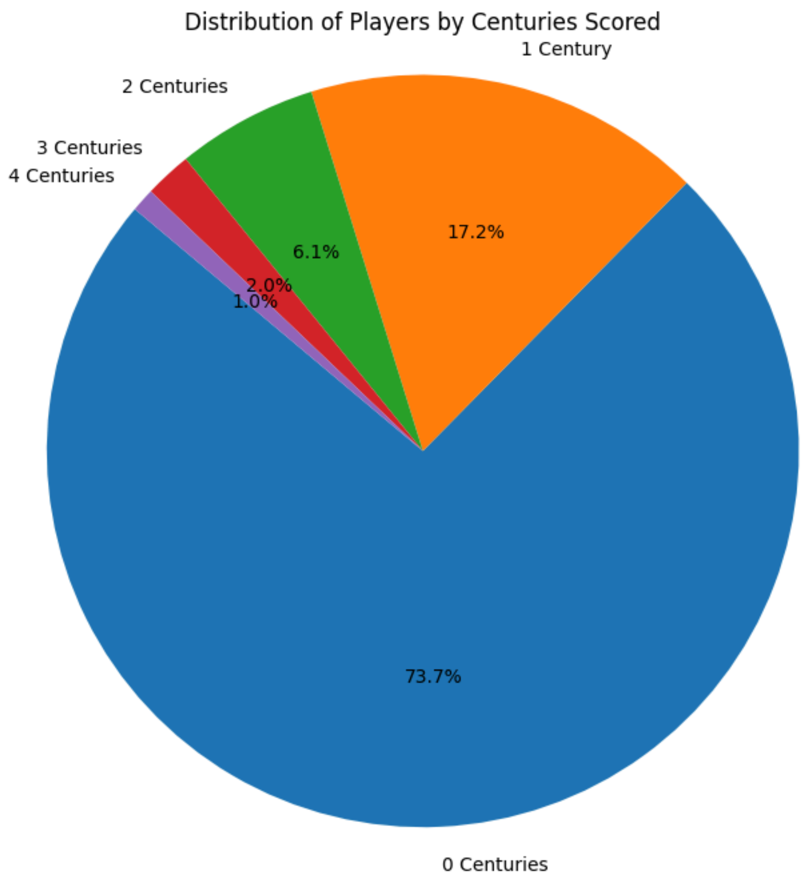
Scatter Plot of Runs vs. Strike Rate: This will help us understand the relationship between the number of runs scored by a player and their strike rate.

```
# Box Plot of Batting Averages
plt.figure(figsize=(10, 6))
sns.boxplot(data=df, y='Ave')
plt.title('Box Plot of Batting Averages')
plt.ylabel('Batting Average')
plt.show()
```



Box Plot of Batting Averages: This will show us the distribution of batting averages along with any outliers.

```
# Pie Chart of Centuries
plt.figure(figsize=(8, 8))
century_counts = df['100'].value_counts()
labels = ['0 Centuries', '1 Century', '2 Centuries', '3 Centuries', '4 Centuries']
plt.pie(century_counts, labels=labels, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Players by Centuries Scored')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```



It will display a pie chart showing the distribution of players based on the number of centuries they've scored.

Doing train_test_split

```
from sklearn.model_selection import train_test_split

X = df.drop(columns=['100'])  # Features: all columns except '100' (centuries)
y = df['100']  # Target variable: '100' (centuries)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


print("Training set:")
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("\nTesting set:")
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
Training set:
X_train shape: (79, 12)
y_train shape: (79,)

Testing set:
X_test shape: (20, 12)
y_test shape: (20,)
```

Training The Model

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=0,criterion='gini')
clf.fit(X_train,y_train)
```

```
▾       DecisionTreeClassifier
DecisionTreeClassifier(random_state=0)
```

Checking Accuracy of Dataset

```
from sklearn.metrics import accuracy_score
import math
predictions_test=clf.predict(X_test)
print("Accuracy : ",accuracy_score(y_test, predictions_test)*100)
```

```
Accuracy :  80.0
```

Checking accuracy of training dataset

```
predictions_train = clf.predict(X_train)
accuracy_score(y_train,predictions_train)
```

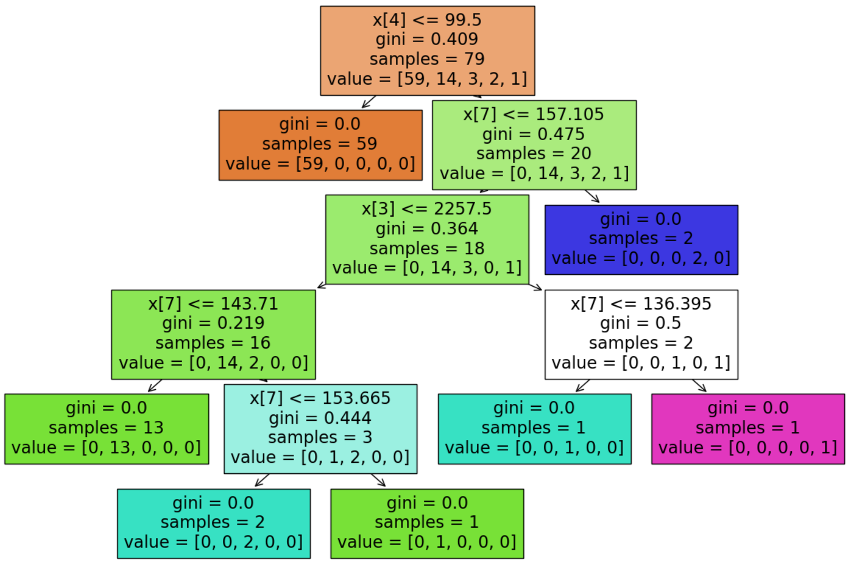
```
1.0
```

An accuracy of 80.0% on the test data suggests that our model correctly predicted 80.0% of the instances in the test set. Similarly, an accuracy of 100% on the training data indicates that our model achieved perfect accuracy on the data it was trained on.

Here we can clearly see that for training dataset our accuracy is 1.0 whereas for test dataset it is low, hence our model is overfitted and to avoid this we will use Pruning method later.

Visualizing our final decision tree

```
from sklearn import tree
plt.figure(figsize=(15,10))
tree.plot_tree(clf,filled=True)
plt.show()
```



Evaluating our test dataset

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

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        14
           1       0.40      0.67      0.50         3
           2       0.00      0.00      0.00         3

    accuracy                           0.80        20
   macro avg       0.47      0.56      0.50        20
weighted avg       0.76      0.80      0.78        20

[[14  0  0]
 [ 0  2  1]
 [ 0  3  0]]
```

Test Dataset:

While the precision, recall, and F1-score for class 0 remain perfect (1.00), indicating no misclassifications, the performance metrics for classes 1 and 2 are lower compared to the training dataset. In the test dataset, the model still performs well for class 0, with perfect precision, recall, and F1-score. However, for class 1, while precision is lower (0.40), recall is higher (0.67), suggesting that the model correctly identifies a significant portion of class 1 instances but also misclassifies some instances. The F1-score for class 1 is also lower (0.50) compared to the training dataset, indicating a slight drop in overall performance for this class. For class 2, the model's performance remains poor in both datasets, with precision, recall, and F1-score being 0.00.

Evaluating our training dataset

```
print(classification_report(y_train,predictions_train))
print(confusion_matrix(y_train,predictions_train))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        59
           1       1.00      1.00      1.00        14
           2       1.00      1.00      1.00         3
           3       1.00      1.00      1.00         2
           4       1.00      1.00      1.00         1

    accuracy                           1.00        79
   macro avg       1.00      1.00      1.00        79
weighted avg       1.00      1.00      1.00        79

[[59  0  0  0  0]
 [ 0 14  0  0  0]
 [ 0  0  3  0  0]
 [ 0  0  0  2  0]
 [ 0  0  0  0  1]]
```

Training Dataset:

For all classes (0 to 4), the precision, recall, and F1-score are perfect, indicating that the model achieved 100% accuracy on the training dataset. The confusion matrix shows that the model correctly predicted all instances for each class, resulting in diagonal values equal to the support values.

Conclusion:

The model achieves perfect accuracy on the training dataset, indicating potential overfitting. When evaluated on the test dataset, the model's performance is still excellent for class 0 but shows some degradation for classes 1 and 2. Further analysis is needed to determine the cause of decreased performance on the test dataset and to improve the model's generalization capabilities. This may involve regularization techniques, hyperparameter tuning, or exploring different model architectures.

Finding false positive rate and true positive rate

```
from sklearn.metrics import roc_curve,auc
from sklearn.metrics import roc_auc_score
# Convert multiclass target variable to binary
y_test_binary = (y_test > 0).astype(int)

# Calculate ROC curve and AUC for binary classification
fpr_dt, tpr_dt, thresholds_dt = roc_curve(y_test_binary, dt_probs)
roc_auc = auc(fpr_dt, tpr_dt)

print("FPR :", fpr_dt)
print("TPR :", tpr_dt)
print("Threshold :", thresholds_dt)
print("AUC Score :", roc_auc)
```

```
FPR : [0. 0. 1.]
TPR : [0.         0.83333333 1.        ]
Threshold : [2. 1. 0.]
AUC Score : 0.9166666666666667
```

FPR (False Positive Rate): The false positive rate is the ratio of false positive predictions to the total actual negative instances. In this case, the FPR values are [0. 0. 1.], indicating that for different thresholds, the model correctly identifies all negative instances (FPR of 0.0) up to a certain threshold and then starts incorrectly classifying some negative instances as positive (FPR of 1.0).

TPR (True Positive Rate, or Recall): The true positive rate is the ratio of true positive predictions to the total actual positive instances. The TPR values are [0. 0.83333333 1.], indicating that the model correctly identifies a fraction of positive instances, with the TPR increasing as the threshold decreases.

Thresholds: Thresholds represent the decision thresholds used by the model to classify instances. The threshold values are [2. 1. 0.], with a higher threshold corresponding to a more conservative prediction (lower positive rate) and a lower threshold corresponding to a more aggressive prediction (higher positive rate).

Plotting ROC curve for our Decision Tree

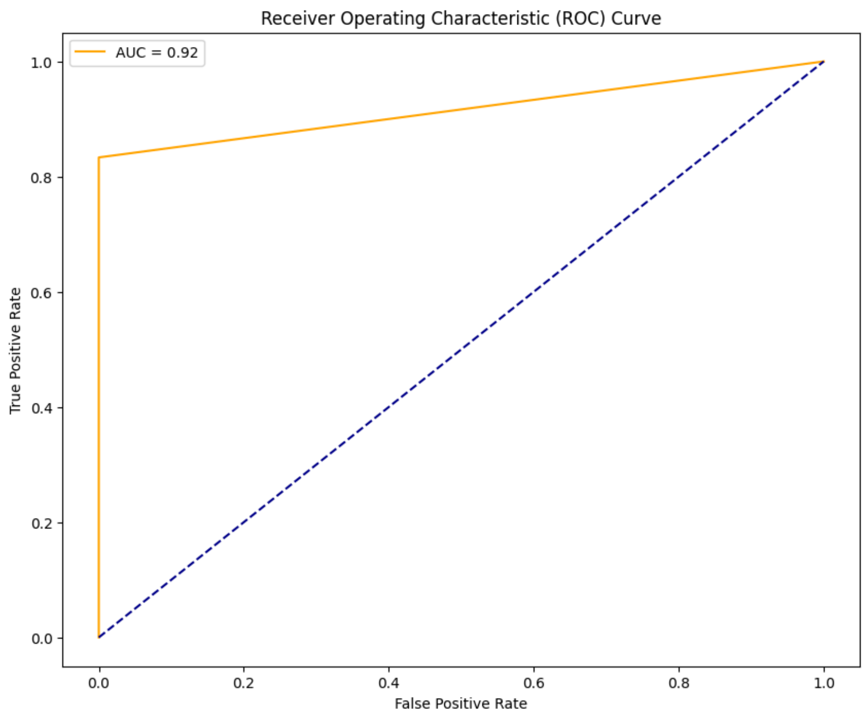
```
auc_score_dt = auc(fpr_dt,tpr_dt)
auc_score_dt
```

```
0.9166666666666667
```

AUC Score (Area Under the ROC Curve): The AUC score is a measure of the model's ability to discriminate between positive and negative instances across different threshold values. In this case, the AUC score is 0.9167, indicating that the model performs well in distinguishing between positive and negative instances.

```
def plot_roc_curve(fpr, tpr):
    plt.figure(figsize=(10,8))
    plt.plot(fpr_dt, tpr_dt, color='orange', label='AUC = %0.2f' % auc_score_dt)
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()
```

```
plot_roc_curve(fpr_dt,tpr_dt)
```



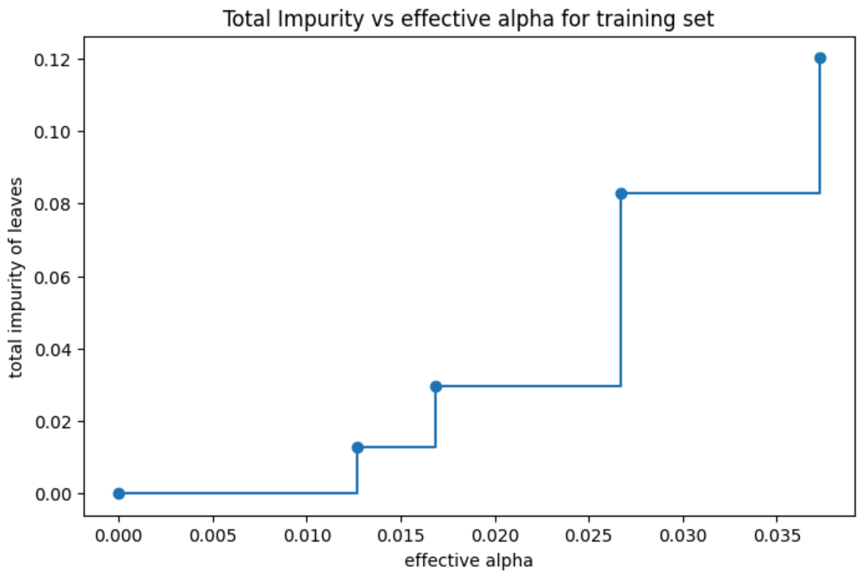
Pruning of our decision tree

```
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

Visualizing alpha w.r.t impurity of leaves

```
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-post")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```

```
Text(0.5, 1.0, 'Total Impurity vs effective alpha for training set')
```

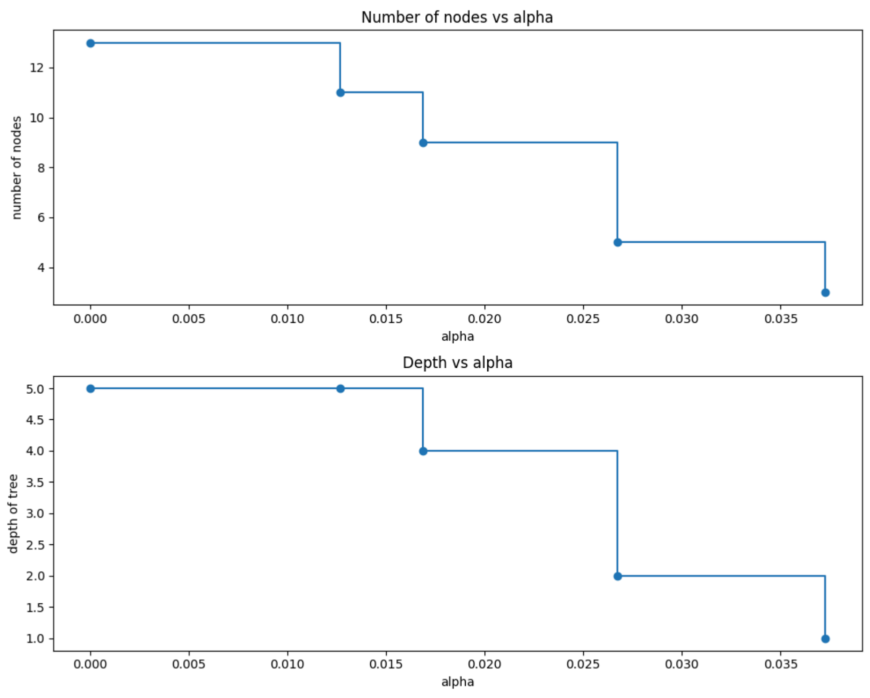


```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
    clfs[-1].tree_.node_count, ccp_alphas[-1]))
```

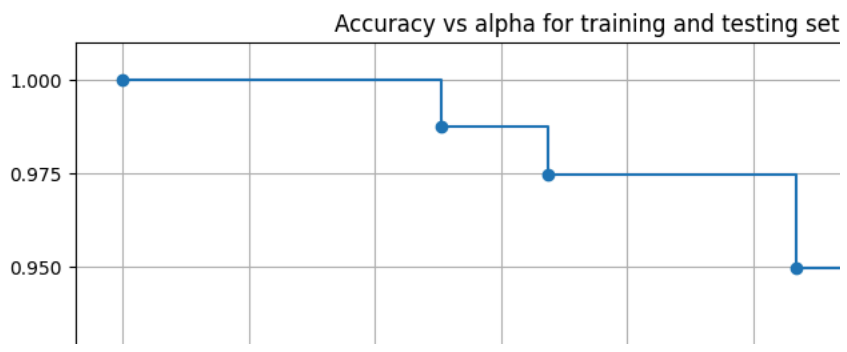
```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.2883352026918763
```

With a value of ccp_alpha equal to 0.2883352026918763, the pruning process has resulted in a highly simplified decision tree where only one node is present in the last tree.

```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]
node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
fig, ax = plt.subplots(2, 1,figsize=(10,8))
ax[0].plot(ccp_alphas, node_counts, marker='o', drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker='o', drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```



```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]
fig, ax = plt.subplots(figsize=(10,8))
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker='o', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker='o', label="test",
        drawstyle="steps-post")
ax.legend()
plt.grid()
plt.show()
```



Accuracy after pruning

```
clf = DecisionTreeClassifier(random_state=0, ccp_alpha=0.016)
clf.fit(X_train,y_train)
DecisionTreeClassifier(ccp_alpha=0.016, random_state=0)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(ccp_alpha=0.016, random_state=0)
```

Accuracy of test dataset

```
from sklearn.metrics import accuracy_score
pred=clf.predict(X_test)
accuracy_score(y_test, pred)
```

```
0.8
```

Accuracy of training dataset

```
pred_1 = clf.predict(X_train)
accuracy_score(y_train,pred_1)
```