Assignment No. 3

Problem Statement: Implement a Decision Tree model

Objective:

1. Understand the mechanics of Decision Trees for both classification and regression.
2. Learn how to control overfitting through pre-pruning (limiting tree growth) and post-pruning (pruning the fully grown tree).
3. Implement and evaluate Decision Tree models with performance metrics.
4. Use cross-validation to assess model generalization.

Prerequisite :

1. A Python environment with essential libraries like pandas, numpy, matplotlib, seaborn, and scikit-learn.
2. Basic knowledge of Python, statistics, and machine learning principles.

Theory :

1. Decision Trees
A **Decision Tree** is a supervised learning algorithm used for both classification and regression. It models decisions by recursively partitioning the input space into distinct regions, each represented by a node in a tree. At each decision node, the algorithm selects a feature and a threshold that best splits the data into two (or more) subsets that are more homogeneous regarding the target variable.

For Classification:

- **Impurity Measures:**
  - **Gini Impurity:**

$$Gini = 1 - \sum_{i=1}^{K} p_i^2$$

  - where pi is the proportion of samples of class i in the node. Gini impurity quantifies the likelihood of an incorrect classification of a randomly chosen element if it were randomly labeled according to the distribution in the node.
  - **Entropy (Information Gain):**

$$Entropy = -\sum_{i=1}^{K} p_i \log_2(p_i)$$

  - Lower entropy indicates a purer node. When a split significantly reduces entropy, it is considered a good split.

For Regression:

- **Variance Reduction (Mean Squared Error):**
  The goal is to split the data such that the variance (or the Mean Squared Error, MSE) within each subset is minimized.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2$$

- Here, y¯ is the mean target value of the samples in the node. The split is chosen to maximize the reduction in MSE.

Decision Trees are highly interpretable since each decision can be traced back through the tree, providing a clear understanding of how input features influence the final prediction.

2. Pre-Pruning (Early Stopping)
Pre-pruning (or early stopping) is the practice of halting the growth of the tree before it fully learns the training data. This is done by specifying constraints during the tree-building process to prevent overfitting. Overfitting happens when the tree captures noise along with the underlying pattern, resulting in poor performance on unseen data.

Key Parameters:

1. **max_depth:**
   Limits the maximum depth of the tree. A shallow tree may underfit, while a very deep tree might overfit.
2. **min_samples_split:**
   The minimum number of samples required to split an internal node. If a node has fewer samples than this threshold, no further split is attempted.
3. **min_samples_leaf:**
   The minimum number of samples that must be present in a leaf node. This ensures that leaves have enough data to make reliable predictions.
4. **max_leaf_nodes:**
   Limits the total number of leaf nodes in the tree, effectively reducing complexity.

By controlling these parameters, pre-pruning reduces the risk of developing overly complex trees and encourages better generalization on unseen data.

3. Post-Pruning (Cost Complexity Pruning)
Post-pruning is a method applied after a decision tree has been grown to its full depth. The idea is to remove parts of the tree that do not provide substantial predictive power, thereby simplifying the model.

Cost Complexity Pruning:

- **Concept:**
  Once the tree is fully grown, some branches may be capturing noise rather than signal. Cost complexity pruning introduces a penalty for complexity, quantified by the parameter α(often called ccp_alpha in scikit-learn). Higher values of α lead to simpler trees.
- **Process:**
  1. **Grow the full tree:**
     The tree is grown without constraints.
  2. **Compute the cost complexity measure:**
     For each subtree, calculate a cost that includes both the error and a penalty for the number of terminal nodes.
  3. **Prune subtrees:**
     Remove branches that result in a minimal increase in error relative to the reduction in complexity.
  4. **Select the optimal α\alphaα:**
     Use cross-validation to determine the value of α\alphaα that results in the best generalization performance.

Post-pruning helps in striking a balance between bias and variance by simplifying the model while retaining its predictive capability.

4. Decision Tree Regression
When applying decision trees to regression tasks, the goal is to predict a continuous output value. Instead of classifying samples into discrete categories, the decision tree regression model partitions the feature space into regions where the response variable is approximately constant.

Key Aspects:
1. **Splitting Criterion:**
   At each node, the model chooses splits that minimize the MSE. The optimal split is the one that results in the greatest reduction in variance.
2. **Prediction:**
   For a given leaf, the prediction is typically the mean of the target values of the training samples that fall into that leaf.
3. **Handling Overfitting:**
   Just like in classification, overfitting is a concern. Pre-pruning parameters (e.g., max_depth, min_samples_split) and post-pruning techniques can also be applied to regression trees to improve generalization.

Decision Tree Regression models are particularly useful for problems where the relationship between input features and the target is non-linear and complex, yet interpretable.

5. Cross-Validation
Cross-validation is a robust technique to evaluate the generalizability of a model. Instead of relying on a single train/test split, cross-validation divides the dataset into multiple folds and iteratively uses different folds for training and validation.

Key Methods:

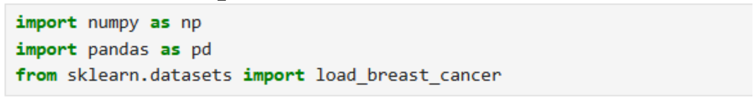
1. **k-fold Cross-Validation:**
   The dataset is partitioned into k folds. For each iteration, one fold is used as the test set and the remaining $k-1$ folds are used for training. The performance metric is averaged over all k iterations.
2. **GridSearchCV:**
   This technique combines exhaustive hyperparameter search with k-fold cross-validation. By specifying a grid of hyperparameter values (different values for max_depth, min_samples_split, or ccp_alpha), GridSearchCV trains and evaluates the model for every combination and selects the one with the best average performance.

Benefits:

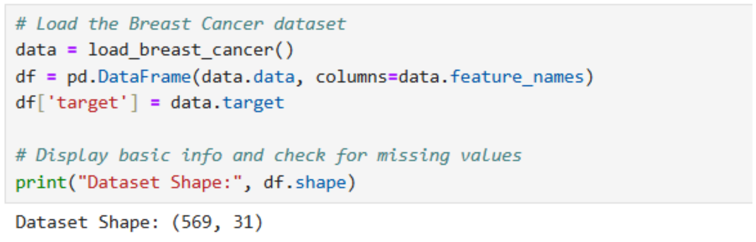
1. **Robust Performance Estimation:**
   Reduces the risk that the model's performance is an artifact of a particular split.
2. **Hyperparameter Tuning:**
   Helps in systematically finding the optimal parameters for the model, balancing bias and variance.
3. **Generalization:**
   Provides a better estimate of how the model will perform on unseen data.

Cross-validation is crucial when working with limited data or when tuning complex models like decision trees, as it ensures that the model's performance is not overestimated.

1. Code & Output

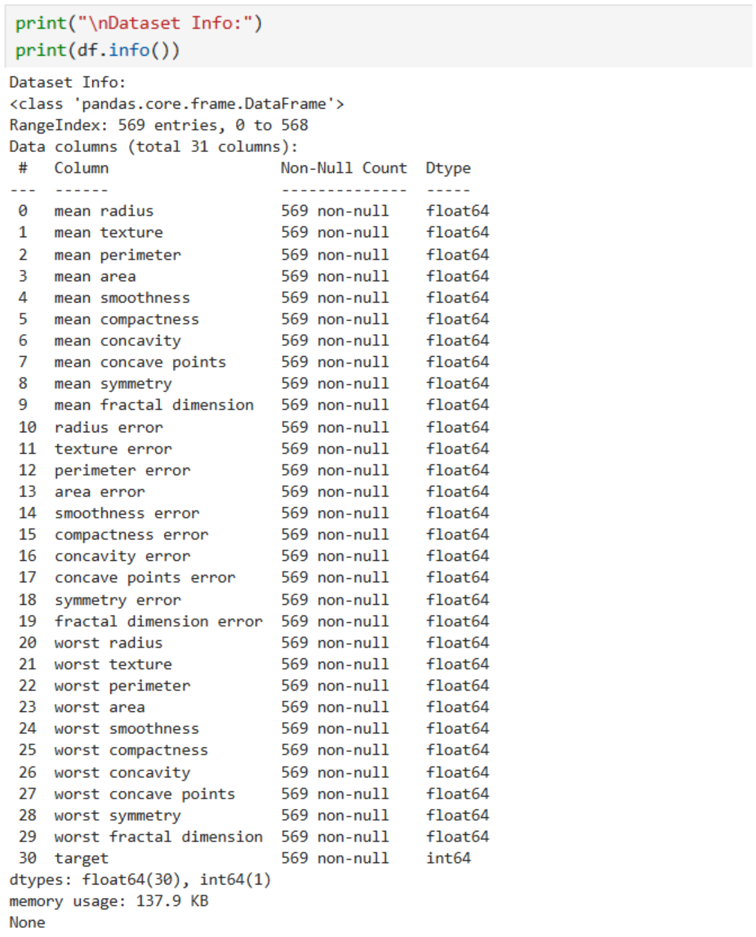


```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
```



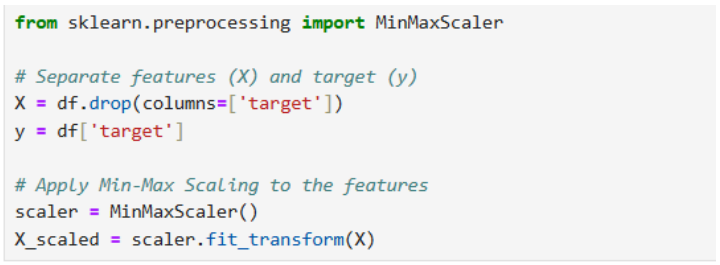
```python
# Load the Breast Cancer dataset
data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

# Display basic info and check for missing values
print("Dataset Shape:", df.shape)
```

```
Dataset Shape: (569, 31)
```

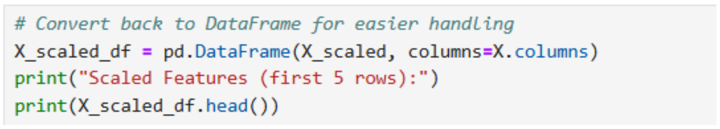


```python
print("\nDataset Info:")
print(df.info())
```

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
 30  target                   569 non-null    int64  
dtypes: float64(30), int64(1)
memory usage: 137.9 KB
None
```



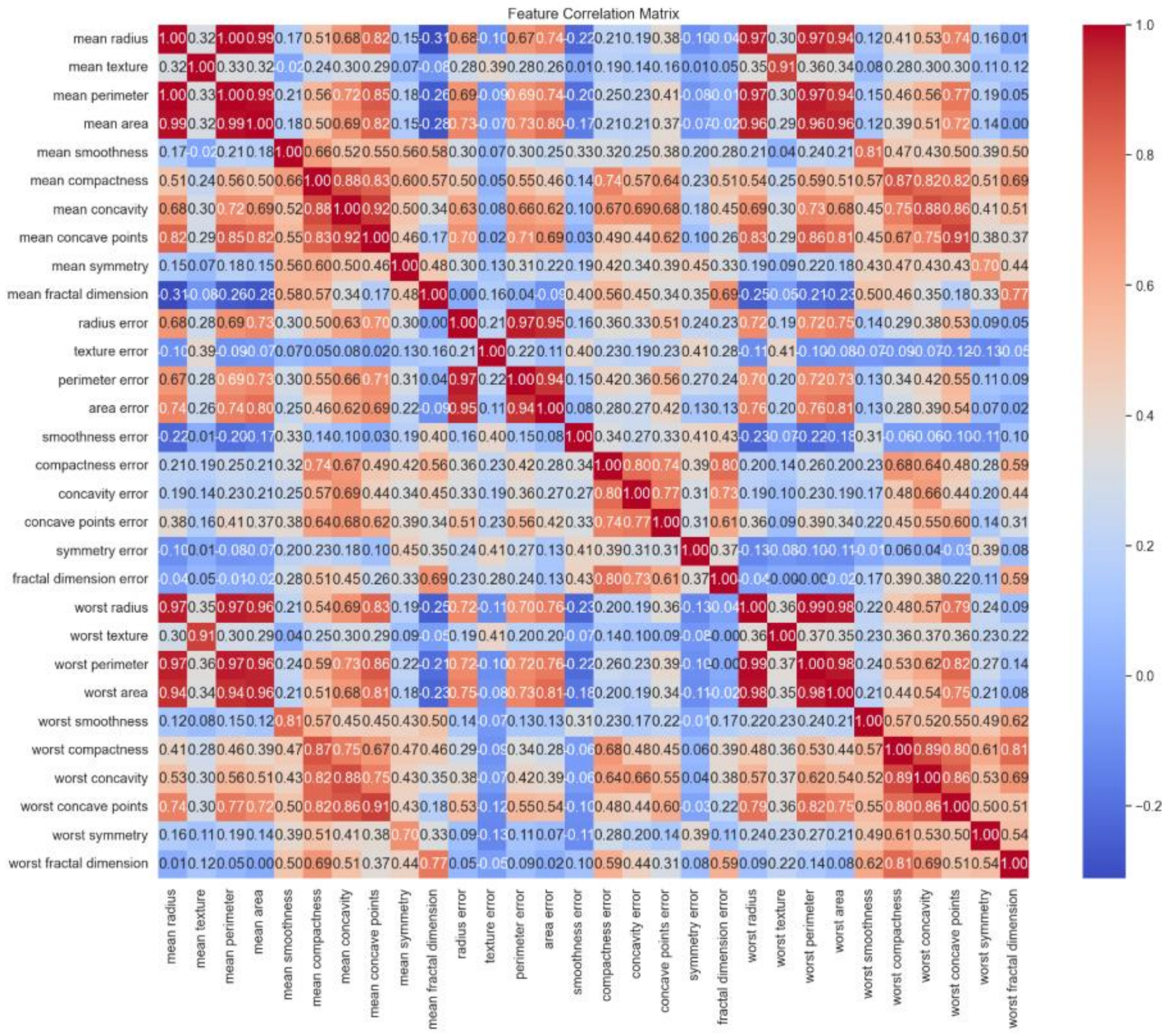
```python
from sklearn.preprocessing import MinMaxScaler

# Separate features (X) and target (y)
X = df.drop(columns=['target'])
y = df['target']

# Apply Min-Max Scaling to the features
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```



```python
# Convert back to DataFrame for easier handling
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
print("Scaled Features (first 5 rows):")
print(X_scaled_df.head())
```

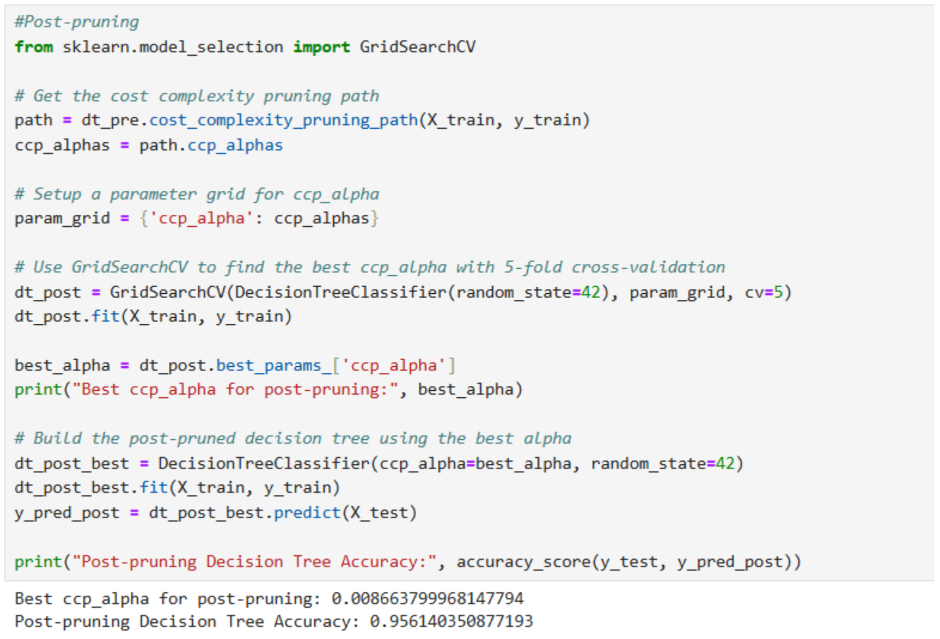
```
Scaled Features (first 5 rows):
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0     0.521037      0.022658        0.545989   0.363733         0.593753   
1     0.643144      0.272574        0.615783   0.501591         0.289880   
2     0.601496      0.390260        0.595743   0.449417         0.514309   
3     0.210090      0.360839        0.233501   0.102906         0.811321   
4     0.629893      0.156578        0.630986   0.489290         0.430351   

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0          0.792037        0.703140             0.731113       0.686364   
1          0.181768        0.203608             0.348757       0.379798   
2          0.431017        0.462512             0.635686       0.509596   
3          0.811361        0.565604             0.522863       0.776263   
4          0.347893        0.463918             0.518390       0.378283   

   mean fractal dimension  ...  worst radius  worst texture  worst perimeter  \
0                0.605518  ...      0.620776       0.141525         0.668310   
1                0.141323  ...      0.606901       0.303571         0.539818   
2                0.211247  ...      0.556386       0.360075         0.508442   
3                1.000000  ...      0.248310       0.385928         0.241347   
4                0.186816  ...      0.519744       0.123934         0.506948   

   worst area  worst smoothness  worst compactness  worst concavity  \
0    0.450698          0.601136           0.619292         0.568610   
1    0.435214          0.347553           0.154563         0.192971   
2    0.374508          0.483590           0.385375         0.359744   
3    0.094008          0.915472           0.814012         0.548642   
4    0.341575          0.437364           0.172415         0.319489   

   worst concave points  worst symmetry  worst fractal dimension  
0              0.912027        0.598462                 0.418864  
1              0.639175        0.233590                 0.222878  
2              0.835052        0.403706                 0.213433  
3              0.884880        1.000000                 0.773711  
4              0.558419        0.157500                 0.142595  

[5 rows x 30 columns]
```

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Calculate the correlation matrix
corr_matrix = X_scaled_df.corr()

# Plot the correlation matrix as a heatmap
plt.figure(figsize=(19, 15))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Feature Correlation Matrix")
plt.show()
```



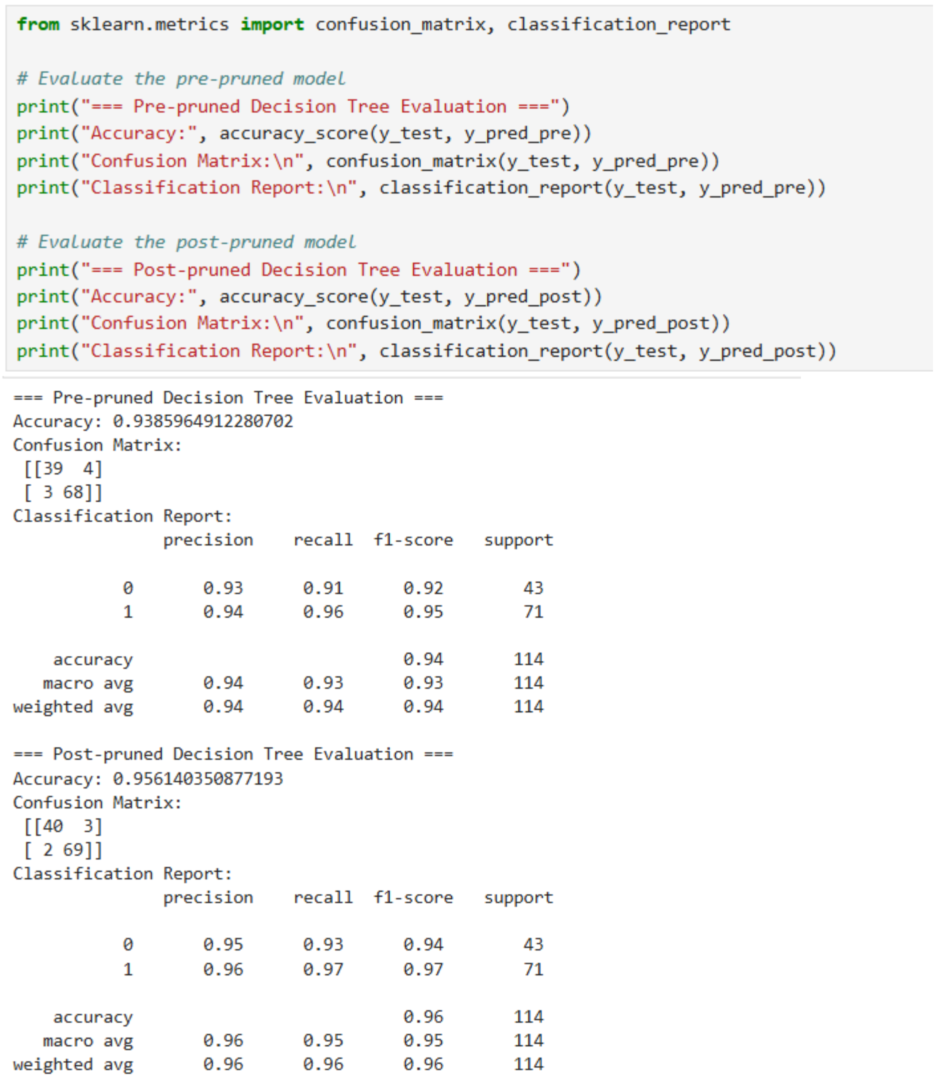
```python
# Pre-pruning
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

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

# Build a Decision Tree with pre-pruning parameters
dt_pre = DecisionTreeClassifier(max_depth=4, min_samples_split=10, random_state=42)
dt_pre.fit(X_train, y_train)
y_pred_pre = dt_pre.predict(X_test)

print("Pre-pruning Decision Tree Accuracy:", accuracy_score(y_test, y_pred_pre))
```

```
Pre-pruning Decision Tree Accuracy: 0.9385964912280702
```



```python
#Post-pruning
from sklearn.model_selection import GridSearchCV

# Get the cost complexity pruning path
path = dt_pre.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path.ccp_alphas

# Setup a parameter grid for ccp_alpha
param_grid = {'ccp_alpha': ccp_alphas}

# Use GridSearchCV to find the best ccp_alpha with 5-fold cross-validation
dt_post = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid, cv=5)
dt_post.fit(X_train, y_train)

best_alpha = dt_post.best_params_['ccp_alpha']
print("Best ccp_alpha for post-pruning:", best_alpha)

# Build the post-pruned decision tree using the best alpha
dt_post_best = DecisionTreeClassifier(ccp_alpha=best_alpha, random_state=42)
dt_post_best.fit(X_train, y_train)
y_pred_post = dt_post_best.predict(X_test)

print("Post-pruning Decision Tree Accuracy:", accuracy_score(y_test, y_pred_post))
```

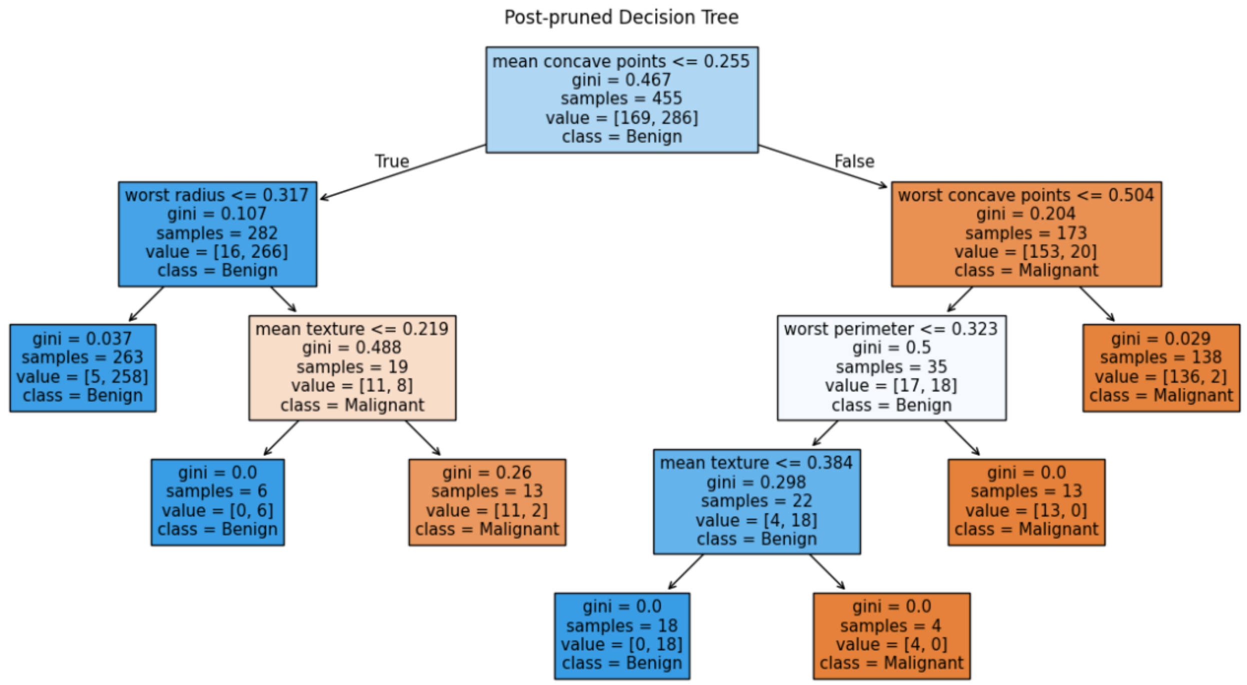
```
Best ccp_alpha for post-pruning: 0.008663799968147794
Post-pruning Decision Tree Accuracy: 0.956140350877193
```



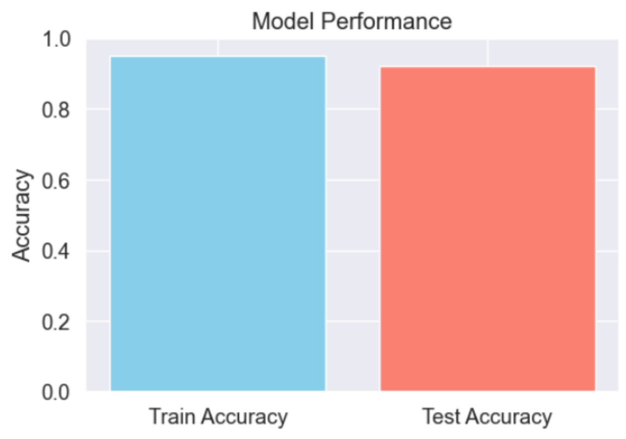
```python
from sklearn.metrics import confusion_matrix, classification_report

# Evaluate the pre-pruned model
print("=== Pre-pruned Decision Tree Evaluation ===")
print("Accuracy:", accuracy_score(y_test, y_pred_pre))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_pre))
print("Classification Report:\n", classification_report(y_test, y_pred_pre))

# Evaluate the post-pruned model
print("=== Post-pruned Decision Tree Evaluation ===")
print("Accuracy:", accuracy_score(y_test, y_pred_post))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_post))
print("Classification Report:\n", classification_report(y_test, y_pred_post))
```

```
=== Pre-pruned Decision Tree Evaluation ===
Accuracy: 0.9385964912280702
Confusion Matrix:
 [[39  4]
 [ 3 68]]
Classification Report:
               precision    recall  f1-score   support

           0       0.93      0.91      0.92        43
           1       0.94      0.96      0.95        71

    accuracy                           0.94       114
   macro avg       0.94      0.93      0.93       114
weighted avg       0.94      0.94      0.94       114

=== Post-pruned Decision Tree Evaluation ===
Accuracy: 0.956140350877193
Confusion Matrix:
 [[40  3]
 [ 2 69]]
Classification Report:
               precision    recall  f1-score   support

           0       0.95      0.93      0.94        43
           1       0.96      0.97      0.97        71

    accuracy                           0.96       114
   macro avg       0.96      0.95      0.95       114
weighted avg       0.96      0.96      0.96       114
```

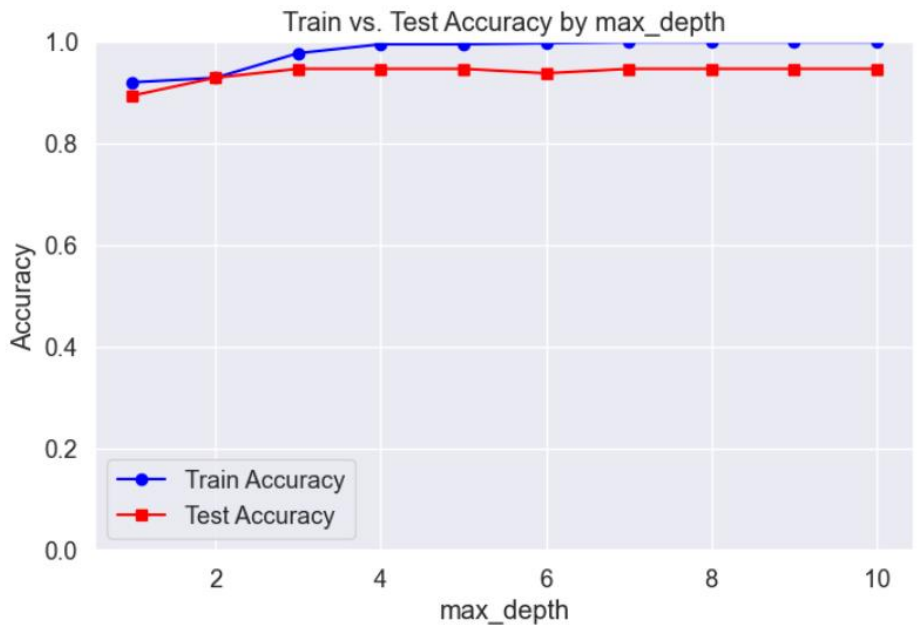
```python
from sklearn.tree import plot_tree

# Visualize the post-pruned Decision Tree
plt.figure(figsize=(16, 8))
plot_tree(dt_post_best, feature_names=X_scaled_df.columns, class_names=['Malignant', 'Benign'], filled=True)
plt.title("Post-pruned Decision Tree")
plt.show()
```


Post-pruned Decision Tree
mean concave points <= 0.255 / gini = 0.467 / samples = 455 / value = [169, 286] / class = Benign
True
False
worst radius <= 0.317 / gini = 0.107 / samples = 282 / value = [16, 266] / class = Benign
worst concave points <= 0.504 / gini = 0.204 / samples = 173 / value = [153, 20] / class = Malignant
gini = 0.037 / samples = 263 / value = [5, 258] / class = Benign
mean texture <= 0.219 / gini = 0.488 / samples = 19 / value = [11, 8] / class = Malignant
worst perimeter <= 0.323 / gini = 0.5 / samples = 35 / value = [17, 18] / class = Benign
gini = 0.029 / samples = 138 / value = [136, 2] / class = Malignant
gini = 0.0 / samples = 6 / value = [0, 6] / class = Benign
gini = 0.26 / samples = 13 / value = [11, 2] / class = Malignant
mean texture <= 0.384 / gini = 0.298 / samples = 22 / value = [4, 18] / class = Benign
gini = 0.0 / samples = 13 / value = [13, 0] / class = Malignant
gini = 0.0 / samples = 18 / value = [0, 18] / class = Benign
gini = 0.0 / samples = 4 / value = [4, 0] / class = Malignant

```python
import matplotlib.pyplot as plt

# Example: Suppose these are your final accuracy scores
train_accuracy = 0.95
test_accuracy = 0.92

# Create a bar chart
labels = ['Train Accuracy', 'Test Accuracy']
scores = [train_accuracy, test_accuracy]

plt.figure(figsize=(6, 4))
plt.bar(labels, scores, color=['skyblue', 'salmon'])
plt.ylim(0, 1)  # Accuracy ranges from 0 to 1
plt.title('Model Performance')
plt.ylabel('Accuracy')
plt.show()
```



```python
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Example data
train_accuracies = []
test_accuracies = []
max_depth_values = range(1, 11)  # Suppose we test depths from 1 to 10

for depth in max_depth_values:
    # Build and train the model
    dt = DecisionTreeClassifier(max_depth=depth, random_state=42)
    dt.fit(X_train, y_train)

    # Predict on train and test sets
    y_train_pred = dt.predict(X_train)
    y_test_pred = dt.predict(X_test)

    # Compute accuracy
    train_acc = accuracy_score(y_train, y_train_pred)
    test_acc = accuracy_score(y_test, y_test_pred)

    # Store the results
    train_accuracies.append(train_acc)
    test_accuracies.append(test_acc)

# Plot the results
plt.figure(figsize=(8, 5))
plt.plot(max_depth_values, train_accuracies, marker='o', label='Train Accuracy', color='blue')
plt.plot(max_depth_values, test_accuracies, marker='s', label='Test Accuracy', color='red')

plt.title('Train vs. Test Accuracy by max_depth')
plt.xlabel('max_depth')
plt.ylabel('Accuracy')
plt.ylim(0, 1)
plt.legend()
plt.grid(True)
plt.show()
```



```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score, classification_report

# Create a manual testing dataset
manual_data = {
    "mean radius": [14.12, 13.10],
    "mean texture": [14.82, 20.54],
    "mean perimeter": [90.16, 86.60],
    "mean area": [600.0, 515.0],
    "mean smoothness": [0.0847, 0.1029],
    "mean compactness": [0.0583, 0.0632],
    "mean concavity": [0.0447, 0.0517],
    "mean concave points": [0.0263, 0.0276],
    "mean symmetry": [0.1601, 0.1744],
    "mean fractal dimension": [0.0559, 0.0655],
    "radius error": [0.27, 0.39],
    "texture error": [1.20, 1.52],
    "perimeter error": [2.57, 2.85],
    "area error": [21.0, 22.5],
    "smoothness error": [0.0042, 0.0053],
    "compactness error": [0.0101, 0.0114],
    "concavity error": [0.0115, 0.0139],
    "concave points error": [0.0067, 0.0072],
    "symmetry error": [0.0121, 0.0154],
    "fractal dimension error": [0.0022, 0.0031],
    "worst radius": [15.50, 14.80],
    "worst texture": [16.60, 22.30],
    "worst perimeter": [98.20, 95.60],
    "worst area": [700.0, 653.0],
    "worst smoothness": [0.0932, 0.1078],
    "worst compactness": [0.0745, 0.0853],
    "worst concavity": [0.0653, 0.0742],
    "worst concave points": [0.0342, 0.0401],
    "worst symmetry": [0.1803, 0.1932],
    "worst fractal dimension": [0.0659, 0.0712],
    "target": [0, 1]  # 0: malignant, 1: benign
}
```

```python
# Create DataFrame for manual testing dataset
manual_df = pd.DataFrame(manual_data)

# Separate features and target for manual dataset
X_manual = manual_df.drop(columns=['target'])
y_manual = manual_df['target']

# Apply the same scaling (using the same scaler fitted on the training data)
X_manual_scaled = scaler.transform(X_manual)

# Use the trained model to predict on the manual testing dataset
y_manual_pred = model.predict(X_manual_scaled)

# Evaluate performance on the manual dataset
print("\nManual Test Data Accuracy:", accuracy_score(y_manual, y_manual_pred))
print("Manual Test Data Classification Report:\n", classification_report(y_manual, y_manual_pred))
```

```
Manual Test Data Accuracy: 0.5
Manual Test Data Classification Report:
               precision    recall  f1-score   support

           0       0.00      0.00      0.00         1
           1       0.50      1.00      0.67         1

    accuracy                           0.50         2
   macro avg       0.25      0.50      0.33         2
weighted avg       0.25      0.50      0.33         2
```

Github: https://github.com/dnyaneshwardhere/ML

Conclusion

In this assignment, we explored Decision Tree models for both classification and regression. We implemented pre-pruning techniques to control tree growth and reduce overfitting, and we applied post-pruning using cost complexity pruning to further refine the model. Cross-validation was used to assess model performance and ensure generalizability. This comprehensive approach provides valuable insights into model tuning and evaluation, laying a strong foundation for further work with Decision Trees.