NJALA UNIVERSITY

School of Technology

MSc. In Computer Science

IRIS DATASET DOCUMENTATION

SUPERVISOR: A. J FOFANAH

GROUP MEMBERS

Abu Junior Vandi – 82937

Joseph Prince Conteh – 54992

Table of Contents

Machine Learning Workflow on the Iris Dataset

1. Introduction

1.1 Project Overview

This project leverages machine learning techniques to classify the **Iris dataset** using **Random Forest** and **Decision Tree** models. The workflow follows a structured approach, including **data preprocessing, feature scaling, model training, performance evaluation, and visualization**.

The primary objective is to develop and compare classification models to determine which algorithm provides the highest accuracy and generalization capability. The project also incorporates **hyperparameter tuning, cross-validation, and learning curve analysis** to enhance model performance and interpretability.

Additionally, visualizations such as **confusion matrices, feature importance plots, and decision tree structures** are utilized to gain deeper insights into model behavior and feature significance. Logging mechanisms are implemented to ensure traceability and debugging throughout the machine learning pipeline.

1.2 Objective

- Load and explore the dataset.
- Handle missing values and preprocess data.
- Train and evaluate **Random Forest** and **Decision Tree** models.
- Compare model performances using accuracy, confusion matrix, and feature importance.
- Perform **hyperparameter tuning** using GridSearchCV.
- Implement **cross-validation** and **learning curves** for model evaluation.

2. Dataset Description

2.1 Overview

The dataset used in this analysis is the **Iris dataset**, a well-known dataset in machine learning and statistical classification. It consists of **measurements of iris flowers** from three different species, making it an ideal dataset for classification tasks.

2.2 Dataset Source

- **File Name:** `iris.csv`
- **Dataset Type:** Multiclass classification
- **Number of Instances:** 150
- **Number of Features:** 4 (Numerical)
- **Number of Classes:** 3 (Categorical Target Variable)

2.3 Features and Data Types

The dataset contains the following attributes:

Feature Name	**Data Type**	**Description**
Sepal Length	float64	Length of the sepal (in cm)
Sepal Width	float64	Width of the sepal (in cm)
Petal Length	float64	Length of the petal (in cm)
Petal Width	float64	Width of the petal (in cm)

2.4 Target Variable

The target variable **`species`** represents the class label of the iris flower species. It is a **categorical variable** with the following classes:

Class Label	Species Name
0	Setosa
1	Versicolor
2	Virginica

2.5 Dataset Characteristics

- **Balanced Dataset:** Each class has an equal number of observations (50 samples per species).
- **No Missing Values (by default):** The original dataset does not contain missing values, but verification is performed during preprocessing.
- **Well-Suited for Classification Tasks:** The dataset is commonly used for supervised learning and algorithm benchmarking.

3. Data Preprocessing

3.1 Data Loading

The dataset is imported into a **Pandas DataFrame** using the `pd.read_csv()` function. This step ensures the data is structured in a tabular format for further processing and analysis.

3.2 Initial Exploration

To understand the dataset's structure and key characteristics, the following exploratory steps are performed:

- **Preview Data:** The first few rows are displayed using `df.head()` to examine the format and content.
- **Dataset Summary:** The `df.info()` function provides insights into data types, non-null counts, and memory usage.
- **Descriptive Statistics:** The `df.describe()` function generates summary statistics for numerical columns, including mean, standard deviation, and percentiles.

3.3 Handling Missing Values

To ensure data completeness and prevent model biases, missing values are identified and handled systematically:

- **Detection:** The presence of missing values is assessed using `df.isnull().sum()`.
- **Imputation Strategies:**
    - **Numerical Features:** Missing values are replaced with the **median** of the respective column to preserve central tendency while minimizing outlier influence.
    - **Categorical Features:** Missing values are filled with the **mode** (most frequent value) to maintain consistency within categorical variables.

3.4 Encoding Categorical Variables

Machine learning models require numerical inputs. Therefore, categorical variables, such as the `species` column, are converted into numerical representations using **Label Encoding**:

- `LabelEncoder()` from `sklearn.preprocessing` is applied to transform category labels into integer values.
- This encoding ensures that the target variable is compatible with classification models.

3.5 Splitting Data

To evaluate model performance, the dataset is divided into **training and testing sets** using `train_test_split()` from `sklearn.model_selection`:

- **Training Set (80%)**: Used to train machine learning models.
- **Testing Set (20%)**: Reserved for evaluating model performance on unseen data.
- The `random_state` parameter is set to ensure reproducibility of results.

4. Model Training and Evaluation

4.1 Random Forest Classifier

Model Training

- The **Random Forest Classifier** is a powerful ensemble model that combines multiple decision trees to improve classification performance by reducing overfitting and increasing generalization. It operates by aggregating predictions from various decision trees (trained on different subsets of the data) and averaging their outputs for regression tasks or voting for classification tasks.
    - **Training the Model**:
      The `RandomForestClassifier` is initialized and trained on the training dataset, `X_train` and `y_train`. The model is configured with a `random_state` for reproducibility. The following command is used for training:
      python
      CopyEdit
      `rf_model = RandomForestClassifier(random_state=42)`
    - `rf_model.fit(X_train, y_train)`

Model Evaluation

Once the model has been trained, it is evaluated using various metrics to assess its performance on the test data, `X_test`and `y_test`. The following evaluation techniques are used:

- **Accuracy Score**:
  Accuracy measures the proportion of correct predictions out of the total number of predictions made. It is calculated using the `accuracy_score` function:
  python
  CopyEdit
  `test_accuracy = accuracy_score(y_test, y_test_pred)`

- **Classification Report**:
  The classification report provides a comprehensive summary of key metrics such as

precision, recall, and F1-score for each class. It is generated using the `classification_report` function, which includes:

a. **Precision**: The proportion of positive predictions that are actually correct.
b. **Recall**: The proportion of actual positives that were correctly identified.
c. **F1-Score**: The harmonic mean of precision and recall.
d. **Support**: The number of actual occurrences of each class in the dataset. Example:
e. python
   CopyEdit
   ```
   classification_report(y_test, y_test_pred)
   ```

- **Confusion Matrix**:
  A confusion matrix is used to evaluate the performance of the classification model by visualizing the true positives, false positives, true negatives, and false negatives. The matrix is visualized as a heatmap for better interpretation using `sns.heatmap()`. Example:
  python
  CopyEdit
  ```
  cm = confusion_matrix(y_test, y_test_pred)
  sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
  ```

Feature Importance

- **Feature Importance Extraction**:
  Random Forest assigns a score to each feature based on its importance in predicting the target variable. This is calculated through various techniques such as Gini impurity or information gain. The feature importances are accessible through the `feature_importances_` attribute of the trained model.
  python
  CopyEdit
  ```
  feature_importances = rf_model.feature_importances_
  ```

- **Feature Importance Visualization**:
  The feature importances are visualized using a **bar plot** to understand the relative significance of each feature. Features with higher scores are considered more important for classification tasks.
  python
  CopyEdit
  ```
  sns.barplot(x=feature_importances, y=feature_names)
  plt.title("Feature Importance for Random Forest")
  plt.xlabel("Importance")
  plt.ylabel("Feature")
  plt.show()
  ```

4.2 Decision Tree Classifier

Model Training

- **Decision Tree Classifier**:
  A decision tree is a supervised learning algorithm that recursively splits the dataset into subsets based on the feature that results in the greatest information gain or Gini impurity reduction. The Decision Tree model is initialized and trained on the training data as follows:
  python
  CopyEdit
  `dt_model = DecisionTreeClassifier(random_state=42)`
- `dt_model.fit(X_train, y_train)`

Model Evaluation

Similar to the Random Forest Classifier, the Decision Tree model is evaluated using a range of metrics:

- **Accuracy Score**:
  The accuracy score for the Decision Tree model is calculated by comparing the predicted values (`y_pred_dt`) with the true values (`y_test`).
  python
  CopyEdit
  `accuracy = accuracy_score(y_test, y_pred_dt)`

- **Classification Report**:
  The classification report for the Decision Tree model is computed in the same manner as for the Random Forest model, offering insights into precision, recall, and F1-score for each class.
  python
  CopyEdit
  `classification_report(y_test, y_pred_dt)`

- **Confusion Matrix**:
  A confusion matrix is generated to assess the performance of the Decision Tree classifier, allowing for a deeper understanding of where the model's predictions might be failing. The confusion matrix is visualized using a heatmap, similar to the Random Forest evaluation:
  python
  CopyEdit
  `cm_dt = confusion_matrix(y_test, y_pred_dt)`
- `sns.heatmap(cm_dt, annot=True, fmt='d', cmap='viridis', cbar=False)`

Decision Tree Visualization

- **Visualizing the Decision Tree**:
  Decision Trees are often visualized to interpret the decision-making process. The `plot_tree()` function is used to visualize the tree's structure, showing how the model splits the data at each node based on different features. This helps in understanding the model's decisions, especially for smaller datasets.
  Example:
  python
  CopyEdit
  ```
  plt.figure(figsize=(20, 10))
  plot_tree(dt_model, filled=True, feature_names=X.columns,
  class_names=np.unique(y).astype(str), rounded=True)
  plt.title("Decision Tree Visualization")
  plt.show()
  ```

Feature Importance

- **Feature Importance Extraction**:
  Like the Random Forest model, the Decision Tree model also computes feature importance using the Gini impurity criterion. The importance score for each feature indicates how useful that feature is for making predictions.
  python
  CopyEdit
  ```
  dt_feature_importances = dt_model.feature_importances_
  ```

- **Feature Importance Visualization**:
  Feature importances are plotted similarly to the Random Forest, where higher importance features are visualized to understand which features the model relies on the most for predictions.
  Example:
  python
  CopyEdit
  ```
  sns.barplot(x=dt_feature_importances, y=feature_names)
  plt.title("Feature Importance for Decision Tree")
  plt.xlabel("Importance")
  plt.ylabel("Feature")
  plt.show()
  ```

6. Model Performance Comparison

- Compare **Random Forest vs. Decision Tree** using test accuracy.

7. Advanced Model Analysis

7.1 Precision-Recall Curve

The **Precision-Recall Curve** is a performance evaluation tool primarily used for binary classification problems, particularly when dealing with imbalanced datasets. It illustrates the trade-off between precision (positive predictive value) and recall (sensitivity).

- Only for **binary classification** cases (`precision_recall_curve()`).

7.2 Hyperparameter Tuning (GridSearchCV)

Hyperparameter tuning refers to the process of optimizing the settings or parameters of a machine learning model to achieve the best performance. The **GridSearchCV** method exhaustively searches through a specified parameter grid to find the optimal combination of hyperparameters for the model.

In the case of the **Random Forest Classifier**, key hyperparameters that can be tuned include:

- **`n_estimators`:** The number of trees in the forest. A larger number typically increases model performance but also computational cost.
- **`max_depth`:** The maximum depth of each tree. Limiting this can prevent overfitting.
- **`min_samples_split`:** The minimum number of samples required to split an internal node. Increasing this value helps prevent overfitting by making the model more conservative.

- Perform GridSearchCV to optimize **Random Forest hyperparameters** (`n_estimators, max_depth, min_samples_split`).

7.3 Cross-Validation (Stratified K-Fold)

Cross-validation is a technique used to evaluate the generalization performance of a machine learning model. It involves splitting the dataset into multiple subsets or "folds" and training the model on some folds while testing it on the remaining folds.

Stratified K-Fold Cross-Validation ensures that each fold maintains the same distribution of target labels as the entire dataset, which is particularly important in imbalanced classification tasks. By preserving the distribution of classes, Stratified K-Fold provides a more accurate representation of model performance across different subsets of data.

Use `StratifiedKFold` to check model stability across multiple splits.

7.4 Learning Curve Analysis

A **Learning Curve** plots the performance of a model (in terms of accuracy or another evaluation metric) against the size of the training data. It is a valuable tool for diagnosing model performance issues like overfitting or underfitting.

- **Overfitting:** Occurs when the model performs well on training data but poorly on validation or test data, indicating that the model is too complex.
- **Underfitting:** Occurs when the model performs poorly on both training and test data, indicating that the model is too simple.

The **Learning Curve** allows for the visual identification of such issues:

- If the training score is high but the test score is low and converges slowly, the model may be overfitting.
- If both scores are low and converging, the model may be underfitting.

- Plot learning curves to diagnose **overfitting/underfitting** using `learning_curve()`.

8. Visualizations

Visualization plays a key role in understanding model performance, interpreting results, and communicating insights effectively. This section describes the visualizations used to evaluate the models and their results.

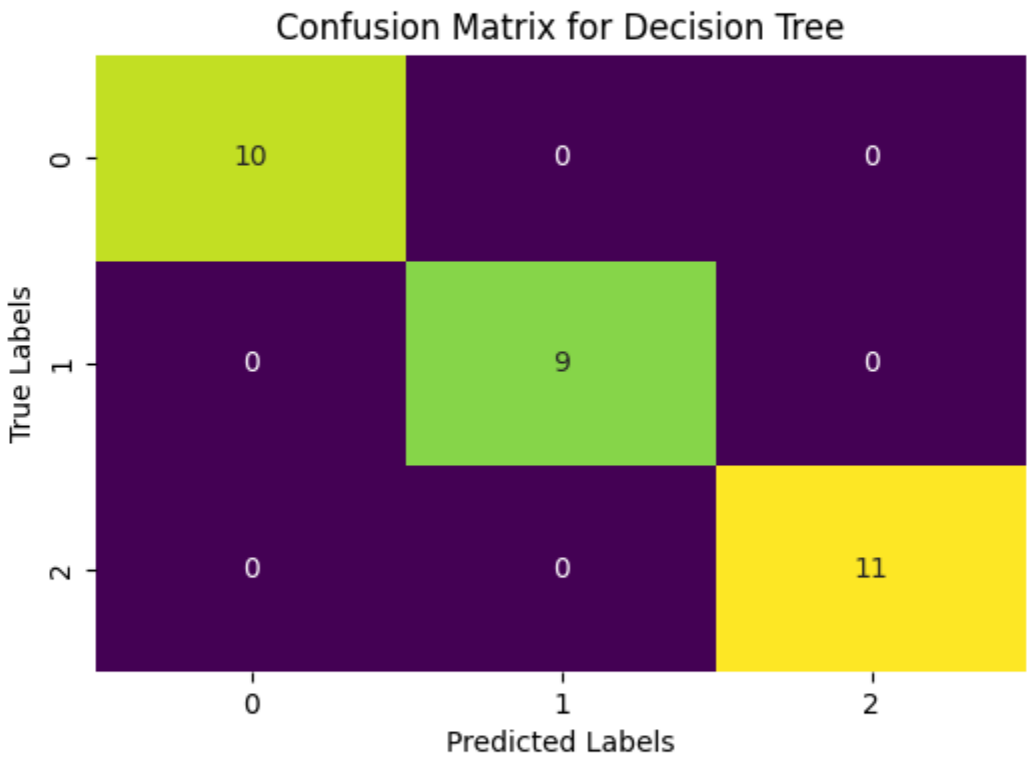
8.1 Confusion Matrix Heatmap

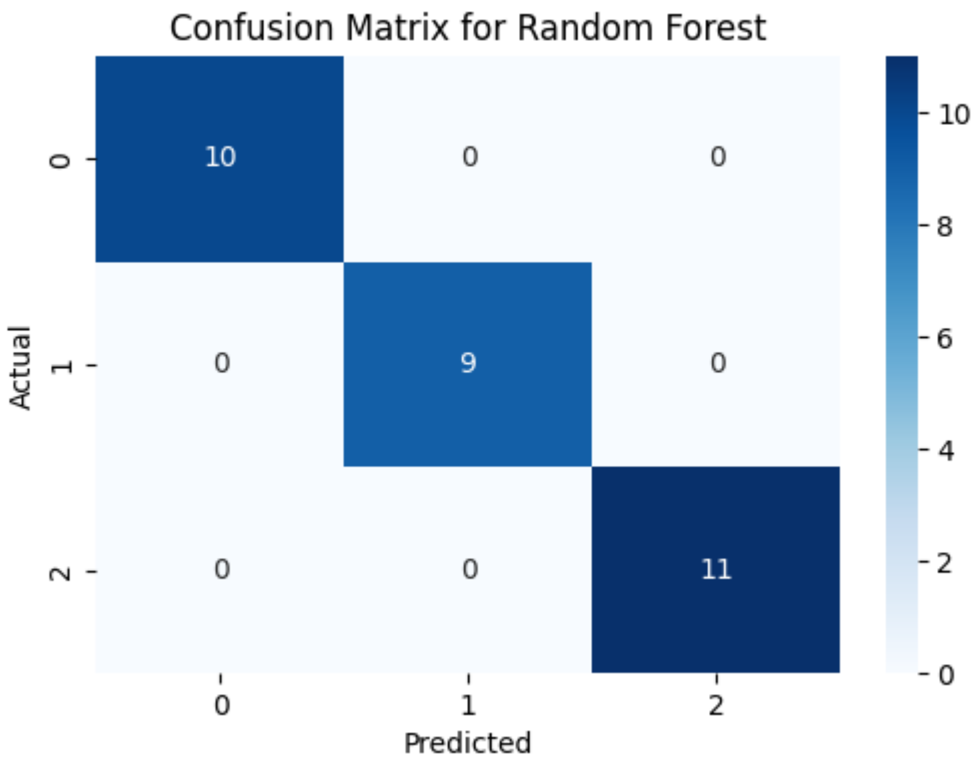
A **confusion matrix** provides a summary of the prediction results by comparing the true labels with the predicted labels. It helps identify misclassifications and assess the performance of the model. To enhance interpretability, we visualize the confusion matrix using a heatmap.

Purpose:
- Display the number of correct and incorrect predictions across different classes.
- Helps in evaluating classification performance (True Positives, False Positives, True Negatives, False Negatives).
- Easily identifies class imbalances and errors.

Visualization Method:
- The `sns.heatmap()` function from Seaborn is used to generate a heatmap of the confusion matrix. The cells of the heatmap are annotated with the count of predictions.

- **Color Palette:** A gradient color scale (such as `Blues` or `viridis`) is used to distinguish between high and low values.

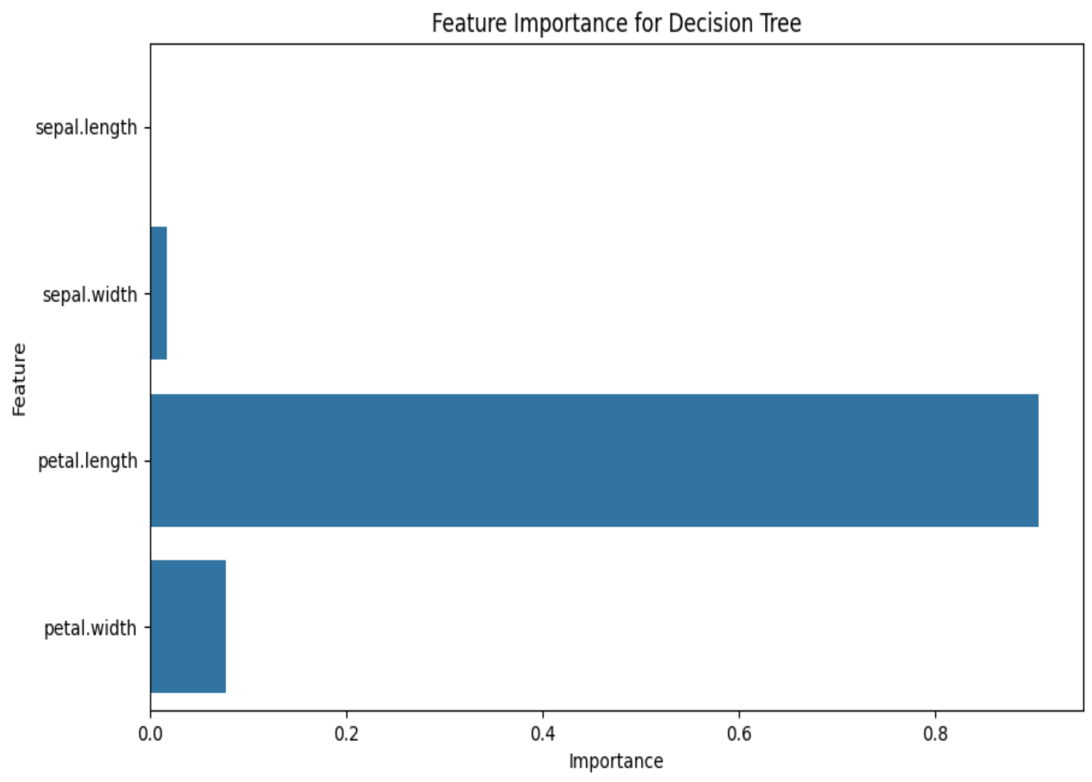


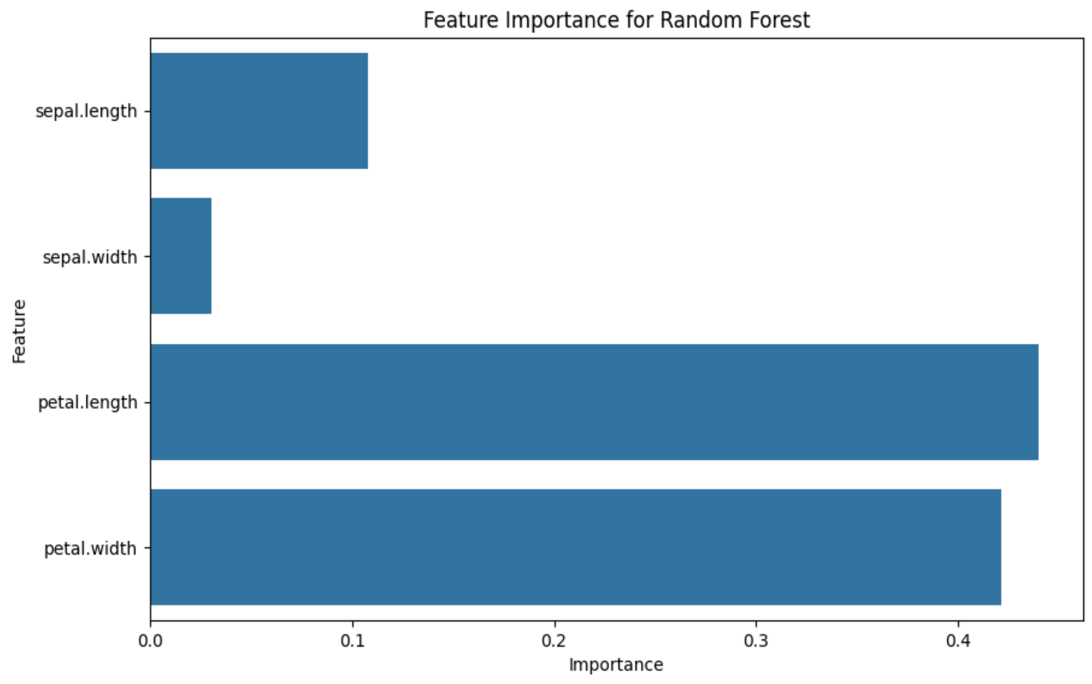


8.2 Feature Importance Bar Plot

Feature importance indicates the relative importance of each feature in predicting the target variable. This visualization helps in understanding which features contribute the most to the model's decision-making process.

- **Purpose:**
  - Highlight the key features that influence model predictions.
  - Provide insight into the importance of each feature for model interpretability and feature selection.
- **Visualization Method:**
  - The feature importances from the **Random Forest** or **Decision Tree** models are extracted using the `feature_importances_` attribute.
  - The importance values are visualized using a **bar plot** (`sns.barplot()`), which allows for easy comparison of the features' contributions.

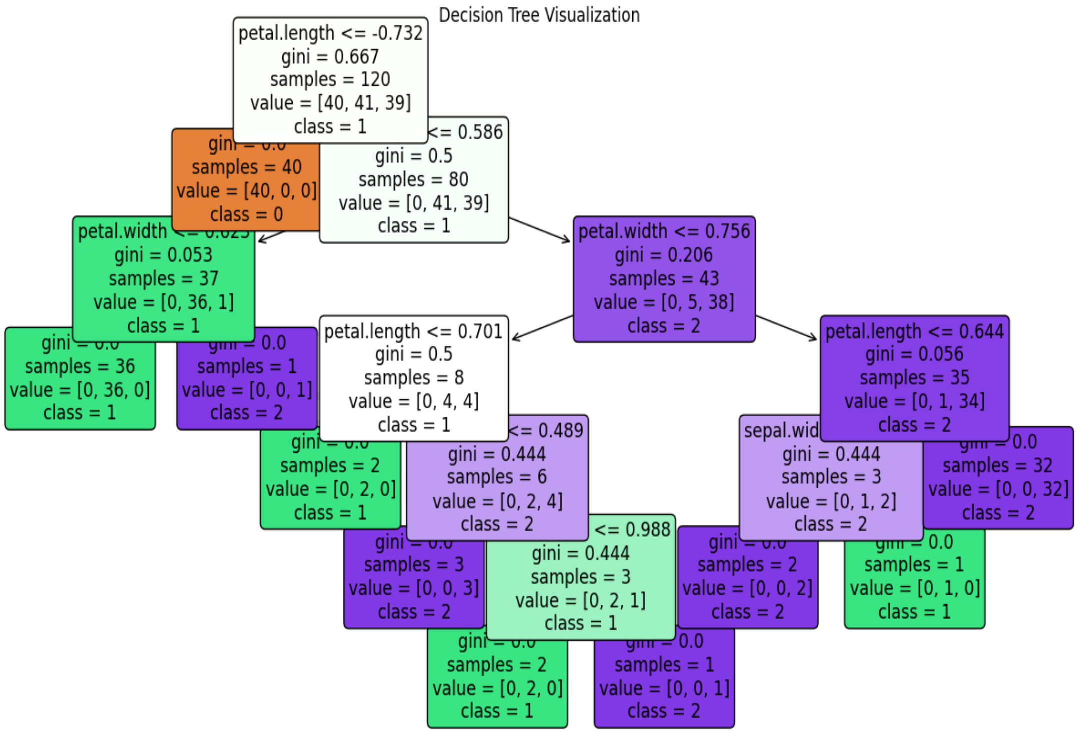




8.3 Decision Tree Visualization

Decision trees are easy to interpret because they represent a series of decision rules that split the data into different classes. The tree can be visualized to help understand the decision-making process of the model.

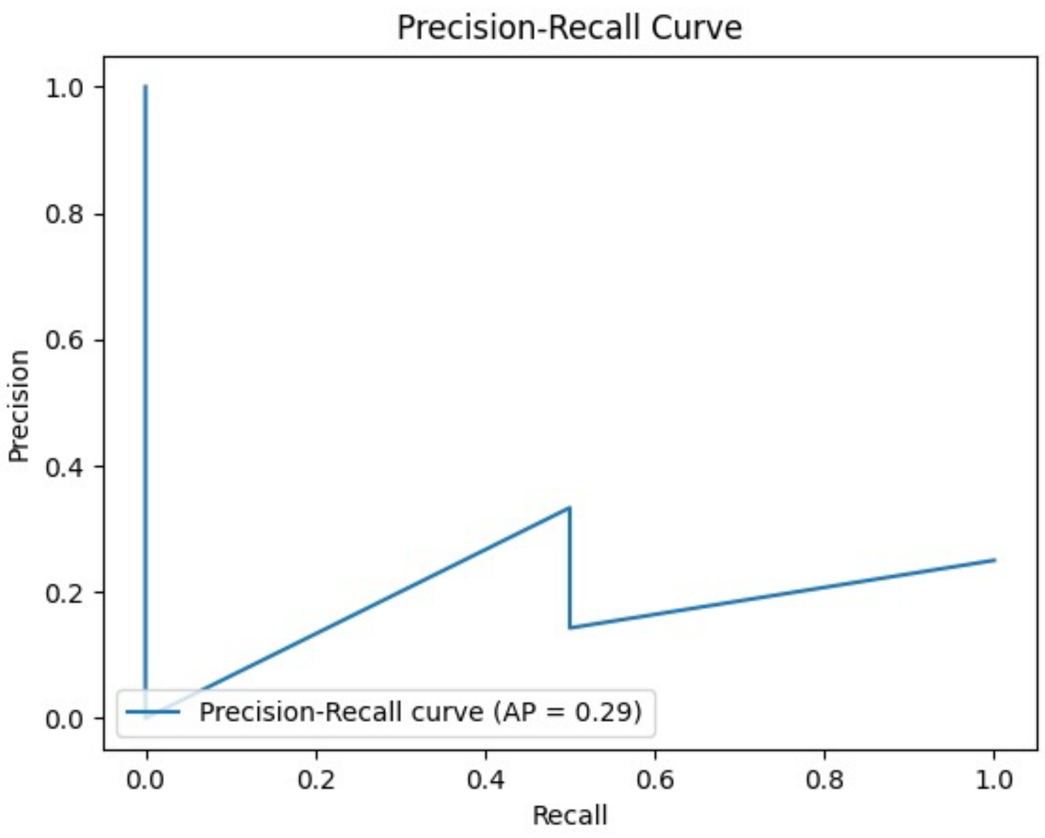
- **Purpose:**
  - Visualize the structure of the decision tree, including the splits, nodes, and leaves.
  - Show how features are used in decision-making and how the tree predicts the target variable based on feature thresholds.
- **Visualization Method:**
  - The `plot_tree()` function from **Scikit-learn** is used to visualize the decision tree. This generates a graphical representation of the tree structure, with nodes containing feature names and decision thresholds.
  - The nodes are color-coded based on the predicted class, and the leaves show the final prediction for each class.



8.4 Precision-Recall Curve

The **precision-recall curve** is particularly useful for evaluating the performance of a classifier on imbalanced datasets. It plots precision (positive predictive value) against recall (sensitivity) at various thresholds.

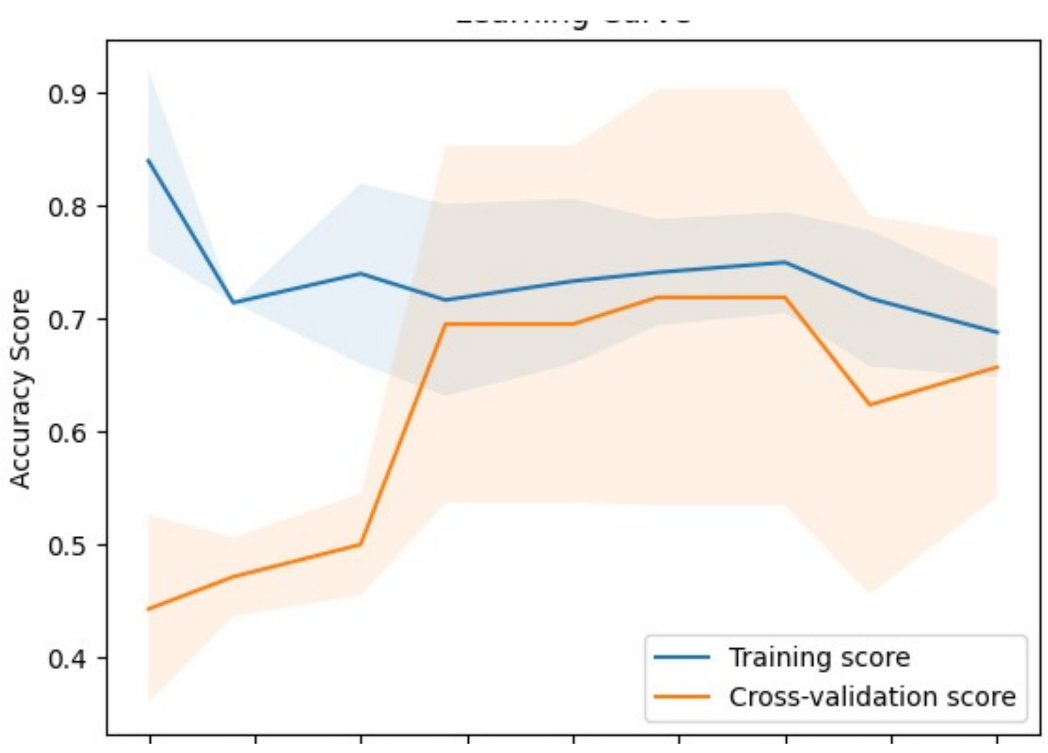
- **Purpose:**
  - Provide a clear view of the trade-off between precision and recall for different threshold values.
  - Especially useful for binary classification problems or imbalanced datasets, where the **accuracy** metric may be misleading.
- **Visualization Method:**
  - The `precision_recall_curve()` function computes precision and recall values, which are then plotted using `PrecisionRecallDisplay()`.
  - The plot visually represents the trade-offs between the two metrics, helping assess the model's ability to correctly identify positive instances.



8.5 Learning Curves

Learning curves provide insight into how the model's performance changes with respect to the number of training samples. They help diagnose issues like overfitting or underfitting, and whether the model benefits from more data.

- **Purpose:**
  - Assess model performance across different amounts of training data.
  - Diagnose overfitting or underfitting by comparing the training and validation scores.
- **Visualization Method:**
  - The `learning_curve()` function from **Scikit-learn** is used to generate training and validation scores across different training set sizes.
  - The learning curve is plotted, showing how the model's performance improves as more data is used for training.
  - The gap between training and validation scores provides insight into potential overfitting or underfitting.



10. Conclusion

In conclusion, the combination of **hyperparameter optimization**, **cross-validation**, and **feature importance analysis has** provided a deeper understanding of the models' behavior, strengths, and weaknesses. The **Random Forest model stands** out as the more robust and reliable classifier for the **Iris dataset**, while the **Decision Tree model** offers a simpler but less generalizable alternative. Future work could focus on further optimizing both models and exploring other ensemble techniques to improve classification performance.