Decision Trees in Scikit-Learn: A Comprehensive Guide

Understanding the implementation and optimization of decision trees using python's scikit-learn library.

Hernández Pérez Erick Fernando

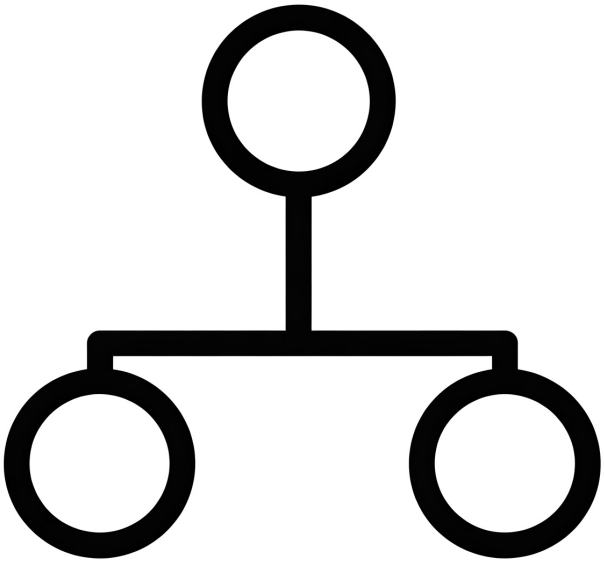

Figure 1

Decision trees are among the most widely used methodologies in machine learning and decision-making. Their hierarchical structure intuitively models classification and regression problems, making it easier to interpret results and identify patterns in data.

A decision tree consists of internal nodes representing data attributes, branches indicating possible values or conditions, and leaf nodes containing the final decision. Constructed from a training dataset, these trees are built using algorithms that recursively partition the feature space, aiming to maximize the purity of the resulting subsets through metrics such as entropy or the Gini index.

Their popularity stems from their interpretability and versatility in handling both numerical and categorical data. Moreover, decision trees form the foundation for more advanced techniques, such as random forests and gradient boosting models, which enhance predictive accuracy and robustness.

With that, it's time to embrace the art of tree-growing.

What is a decision tree?

Han, Kamber and Pei give the next definition:

> A decision tree is a flowchart-like tree structure, where each internal node (non-leaf node) denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (or terminal node) holds a class label. The topmost node in a tree is the root node. (Han, Kamber & Pei, 2012) [2]

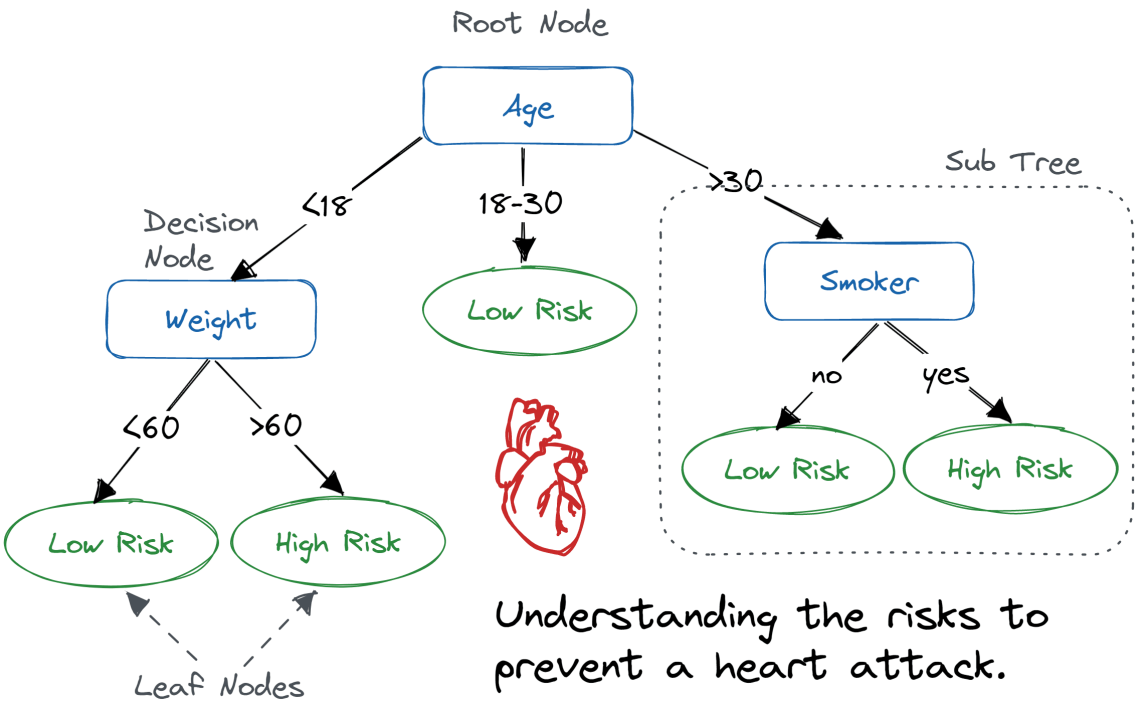

Figure 2: A decision tree. (Ali, 2024) [1]

In this way we can examine each of the characteristics of a tuple X that we want to classify. Going through the tree from top to bottom, taking the correct branches according to the answers to each of the tests, we can reach the corresponding leaf node and therefore its classification label.

Let's analyze the decision tree in Figure 2 and classify a person with the attributes (40 years old, 55 kg, smoker). First, since their age is greater than 30, we follow the right branch. Next, as they are a smoker, we again take the right branch. This leads to the classification of (40 years old, 55 kg, smoker) in the high-risk category for heart attacks.

Now, where would a person with (20 years old, 68 kg, smoker) be classified?

A basic algorithm for inducing a decision tree

The majority of decision tree induction algorithms use a top-down strategy, beginning with a training set that includes tuples and their corresponding class labels. As the tree is constructed, the training set is repeatedly divided into smaller subsets.

Below we present the pseudocode proposed by Han, Kamber and Pei, which has the following inputs:

- Data partition, D, which is a set of training tuples and their associated class labels.
- *attribute_list*, the set of candidate attributes.
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes.

and whose output is the constructed decision tree.

Algorithm 1 Generate_decision_tree

create a node N;
if tuples in D are all of the same class, C **then**
    return N as a leaf node labeled with the class C;
end if
if *attribute_list* is empty **then**
    return N as a leaf node labeled with the majority class in D;
end if
apply **Attribute_selection_method**(D,*attribute_list*) to find the "best" *splitting_criterion*;
label node N with *splitting_criterion*;
if *splitting_attribute* is discrete-valued **and** multiway splits allowed **then**
    splitting_list $\leftarrow$ *attribute_list* $-$ *splitting_attribute*;
end if
for each outcome j of *splitting_criterion* // partition the tuples and grow subtrees for each partition
let D_j be the set of data tuples in D satisfying outcome j; // a partition
if D_j is empty **then**
    attach a leaf labeled with the majority class in D to node N;
elseattach the node returned by **Generate_decision_tree**(D_j, *attribute_list*) to node N;
end if
end for
return N;

And in a more explained way, we would have:

1. **Create a new node,** N. This node will be part of the decision tree.
2. **Check for a pure class:** If all tuples in D belong to the same class C, label N as a leaf node with class C and return it.
3. **Check if there are no more attributes to split on:** If *attribute_list* is empty, label N as a leaf node with the majority class in D and return it.
4. **Determine the best split:** Use the **Attribute_selection_method** to find the optimal splitting criterion.
5. **Label node** N **with the selected splitting criterion.**
6. **Handle multiway splits:** If the chosen splitting attribute is discrete and multiway splits are allowed, update *attribute_list* to exclude the selected attribute.
7. **Partition the data:**
    - For each possible outcome j of the splitting criterion:
        - Create a subset D_j containing tuples that satisfy outcome j.
        - If D_j is empty, attach a leaf node to N labeled with the majority class in D.
        - Otherwise, recursively call **Generate_decision_tree** on D_j, and attach the resulting node to N.
8. **Return node** N**, which now represents a subtree or leaf in the decision tree.**

Attribute Selection Measures

But how do we know which is the best splitting criterion when building a decision tree? To do so, we use attribute selection measures, which help us determine which attribute provides the best partitioning of the data. These metrics assess the purity of the resulting subsets and allow us to choose the split that maximizes the separation between classes, thus improving the accuracy of the model. But let's look at a more formal definition.

> The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure is chosen as the splitting attribute for the given tuples. If the splitting attribute is continuous-valued or if we are restricted to binary trees, then, respectively, either a split point or a splitting subset must also be determined as part of the splitting criterion. The tree node created for partition D is labeled with the splitting criterion,

branches are grown for each out come of the criterion, and the tuples are partitioned accordingly. (Han, Kamber & Pei, 2012) [2]

Let's analyze two of them.

Entropy and Information Gain

Shannon's entropy is a fundamental concept from Claude Shannon's pioneering work in **information theory**. It was introduced to quantify the **"information content"** in a message, measuring the uncertainty or unpredictability of a data source. In simple terms, entropy tells us how much surprise or randomness is in a message: the more unpredictable the content, the higher the entropy. If a message is highly predictable, like a repeated sequence of the same letter, its entropy is low, meaning it carries little new information.

In the construction of decision trees, entropy is used to measure the **uncertainty or impurity within a dataset**. A node N holds a set of tuples that need to be classified, and the goal is to find an attribute that best partitions the data into more organized subsets.

Entropy quantifies the level of disorder in a dataset. A high entropy value indicates that the data is highly mixed, meaning the classes are evenly distributed and the uncertainty is high. Conversely, a low entropy value suggests that the data is more homogeneous, with most tuples belonging to the same class, leading to lower uncertainty.

By selecting an attribute that minimizes entropy, the decision tree ensures that each partition is as pure as possible, effectively grouping similar data together. This approach helps streamline the classification process, reducing the complexity of the tree while still maintaining its effectiveness. Although it does not necessarily guarantee the simplest possible tree, it ensures a structure that efficiently represents the information in the dataset.

The expected information needed to classify a tuple in D (The entropy of D) is given by:

$$H(D) = \text{Info}(D) = -\sum_{i=1}^{m} p_i \cdot \log_2(p_i)$$

where p_i is the nonzero probability that an arbitrary tuple in D belongs to class C_i and is estimated by $\frac{|C_{i,D}|}{|D|}$ (The numerator is the number of tuples in D that belong to class C_i. The denominator is the total number of tuples in the dataset D).

Suppose we split the dataset D based on an attribute A that has v distinct values $\{a_1, a_2, \ldots, a_v\}$, as seen in the training data. If A is a discrete attribute, each value represents a possible outcome when testing for A.

Using A, we divide D into v subsets $\{D_1, D_2, \ldots, D_v\}$, where each subset D_j contains all tuples in D that have the value a_j for attribute A. These subsets correspond to the branches of the decision tree growing from node N.

Ideally, each partition would perfectly separate the data, meaning that all tuples in a subset belong to the same class. However, in practice, partitions often remain impure, containing tuples from multiple classes instead of just one.

How much more information would we still need (after the partitioning) to arrive at an exact classification?

$$\text{Info}_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \cdot \text{Info}(D_j)$$

$\text{Info}_A(D)$ is the expected information required to classify a tuple from D based on the partitioning by A.

Information gain is defined as the difference between the original information requirement and the new requirement:

$$\text{Gain}(A) = \text{Info}(D) - \text{Info}_A(D)$$

$\text{Gain}(A)$ tells us how much would be gained by branching on A. The attribute A with the highest information gain is chosen as the splitting attribute at node N.

A simpler way to understand information gain is through the following formula:

$$\text{Gain}(A) = \text{Info}(\textit{parent}) - [\textit{weighted average}]\text{Info}(\textit{children})$$

Now for an example. Consider the following tuples, which we need to classify by speed: slow or fast.

Grade	Bumpiness	Speed Limit	Speed
Steep	Bumpy	Yes	Slow
Steep	Smooth	Yes	Slow
Flat	Bumpy	No	Fast
Steep	Smooth	No	Fast

Table 1

Let's first see what the entropy of this current state (parent) is. Considering that our class $i = 1$ is slow and our class $i = 2$ is fast.

$$\begin{aligned}\text{Info}(parent) &= -p(slow)\log_2(p(slow)) - p(fast)\log_2(p(fast)) \\ &= -\frac{2}{4}\log_2\left(\frac{2}{4}\right) - \frac{2}{4}\log_2\left(\frac{2}{4}\right) \\ &= -\log_2\left(\frac{2}{4}\right) = 1\end{aligned}$$

Now let's see what happens if we use Grade.

$$\begin{aligned}\text{Info}(steep) &= p(slow)\log_2(p(slow)) - p(fast)\log_2(p(fast)) \\ &= -\frac{2}{3}\log_2\left(\frac{2}{3}\right) - \frac{1}{3}\log_2\left(\frac{1}{3}\right) \\ &= 0.91829\end{aligned}$$

$$\begin{aligned}\text{Info}(flat) &= p(slow)\log_2(p(slow)) - p(fast)\log_2(p(fast)) \\ &= -\frac{0}{1}\log_2\left(\frac{0}{1}\right) - \frac{1}{1}\log_2\left(\frac{1}{1}\right) \\ &\approx -\frac{0}{1}\log_2\left(\frac{0}{1} + 0.00001\right) - \frac{1}{1}\log_2\left(\frac{1}{1} + 0.00001\right) \\ &\approx 0\end{aligned}$$

In the previous calculation, 0.00001 was added to avoid problems with the logarithm calculation.

$$\begin{aligned}\text{Gain}(Grade) &= \text{Info}(parent) - \text{Info}_{Grade}(parent) \\ &= 1 - \left(\frac{3}{4}\text{Info}(steep) + \frac{1}{4}\text{Info}(\text{flat})\right) \\ &= 1 - \left(\frac{3}{4}[0.91829] + \frac{1}{4}[0]\right) \\ &= 1 - 0.6887175 \\ &= 0.3112825\end{aligned}$$

Calculating bumpiness and speed limit is left as an exercise. Once they have been calculated, select the attribute that maximizes the information gain.

Gini index

The **Gini Index**, also known as **Gini Impurity**, is a metric used in decision tree algorithms to measure the impurity or diversity of a dataset. It evaluates how often a randomly chosen element from the dataset would be incorrectly classified if it were randomly labeled according to the class distribution.

Mathematically, for a dataset D with c different classes, the Gini Index is defined as:

$$Gini(D) = 1 - \sum_{i=1}^{c} p_i^2$$

where p_i represents the proportion of instances belonging to class C_i in D. The Gini Index ranges from 0 to 1:

- 0 indicates a **pure node**, meaning all instances belong to the same class.
- Higher values indicate greater impurity, meaning the dataset is more diverse in terms of class distribution.

In decision trees, the attribute that results in the lowest Gini Index after a split is chosen, as it creates the most homogeneous subsets. This helps in constructing efficient trees for classification tasks.

It considers a **binary split** for each attribute. Suppose we have an attribute A with v possible values, like $\{a_1, a_2, \ldots, a_v\}$. To find the best way to split A, we look at all possible subsets of these values. Each subset represents a test to check if an attribute value falls within it. If A has v values, there are 2^v possible subsets. However, we ignore two cases:

- When the subset contains all values of A (because it doesn't split the data).
- When the subset is empty (because it also doesn't divide the data).

So, the number of possible binary splits is $2^v - 2$.

When considering a binary split, we compute a weighted sum of the impurity of each resulting partition. For example, if a binary split on A partitions D into D_1 and D_2, the Gini index of D given that partitioning is:

$$\text{Gini}_A(D) = \frac{|D_1|}{|D|}\text{Gini}(D_1) + \frac{|D_2|}{|D|}\text{Gini}(D_2)$$

The reduction in impurity that would be incurred by a binary split on a discrete or continuous valued attribute A is:

$$\Delta\text{Gini}(A) = \text{Gini}(D) - \text{Gini}_A(D)$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute.

Let's go back to our example. Let's first calculate the Gini index of the parent and then the Gini index Grade.

$$\begin{aligned}
\text{Gini}(parent) &= 1 - (p(slow)^2 + p(fast)^2) \\
&= 1 - (0.5^2 + 0.5^2) \\
&= 1 - (0.25 + 0.25) \\
&= 0.5
\end{aligned}$$

$$\begin{aligned}
\text{Gini}(steep) &= 1 - (p(slow)^2 + p(fast)^2) \\
&= 1 - (\frac{2}{3}^2 + 1/3^2) \\
&= 1 - (\frac{5}{9}) \\
&= \frac{4}{9}
\end{aligned}$$

$$\begin{aligned}
\text{Gini}(flat) &= 1 - (p(slow)^2 + p(fast)^2) \\
&= 1 - (1^2) \\
&= 0
\end{aligned}$$

$$\begin{aligned}\Delta\text{Gini}(Grade) &= \text{Gini}(parent) - \text{Gini}_{Grade}(parent) \\ &= 0.5 - (\frac{3}{4}\frac{4}{9} + \frac{1}{4}[0]) \\ &= \frac{1}{6}\end{aligned}$$

Using scikit-learn

The `DecisionTreeClassifier` from scikit-learn is a machine learning algorithm used for classification tasks. It builds a decision tree, where each node represents a decision rule based on a feature, and each leaf node represents a class label. The algorithm recursively splits the dataset based on the feature that provides the best separation of the data.

Key Features and Parameters

- **criterion**: Defines the function to measure the quality of a split. The default is `"gini"`, which uses the Gini impurity, but you can also use `"entropy"`, which measures information gain.

  ```
  criterion='gini'
  ```

- **splitter**: Determines the strategy used to split at each node. Options include `"best"` (chooses the best split) and `"random"` (chooses the best random split).

  ```
  splitter='best'
  ```

- **max_depth**: Limits the depth of the tree. This parameter can be used to prevent overfitting by stopping the tree from growing too deep.

  ```
  max_depth=5
  ```

- **min_samples_split**: Specifies the minimum number of samples required to split an internal node. A higher value prevents the tree from growing too deep.

  ```
  min_samples_split=2
  ```

- **min_samples_leaf**: Sets the minimum number of samples required to be at a leaf node. Increasing this number can help to smooth the model and reduce overfitting.

  ```
  min_samples_leaf=1
  ```

- **max_features**: The number of features to consider when looking for the best split. This can help to improve performance by reducing the tree's complexity.

  ```
  max_features="sqrt"
  ```

- **random_state**: Controls the randomness of the tree's construction. Setting a fixed value ensures reproducibility.

  ```
  random_state=42
  ```

- **class_weight**: Specifies weights for each class. It can be set to `"balanced"` to automatically adjust weights inversely proportional to class frequencies in the input data.

  ```
  class_weight='balanced'
  ```

DecisionTreeClassifier

```
# Import the necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn import tree

# Function to generate data with a variable number of features
def generate_data(n_features=5):
    # Generate a random dataset for classification (2 classes)
    X, y = make_classification(
        n_features=n_features, n_redundant=0, n_informative=2, random_state=1,
        ↪  n_clusters_per_class=2
    )

    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    ↪  random_state=42)

    # Create the decision tree classifier
    clf = DecisionTreeClassifier(random_state=42)

    # Fit the classifier to the training data
    clf.fit(X_train, y_train)

    # Predict on the test data
    y_pred = clf.predict(X_test)

    # Visualize the decision tree
    plt.figure(figsize=(16,16))
    tree.plot_tree(clf, filled=True, feature_names=[f"Feature {i+1}" for i in
    ↪  range(n_features)],
                   class_names=["Class 0", "Class 1"], rounded=True)
    plt.title("Decision Tree Visualization")
    plt.show()

    # Visualize the data points and the decision of the tree (only for 2 features)
    if n_features == 2:
        plt.figure(figsize=(16, 16))
        scatter = plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, marker='o',
        ↪  edgecolor='k', s=100)
        plt.legend(handles=scatter.legend_elements()[0], labels=["Class 0", "Class 1"])

        plt.title('Generated Data Points and Classified by Decision Tree')
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.show()

    elif n_features > 2:
        print(f"Visualization of points not supported for more than 2 features. Only the
        ↪  decision tree is shown.")

# Call the function with the desired number of features
```

```
generate_data(n_features=5)
```

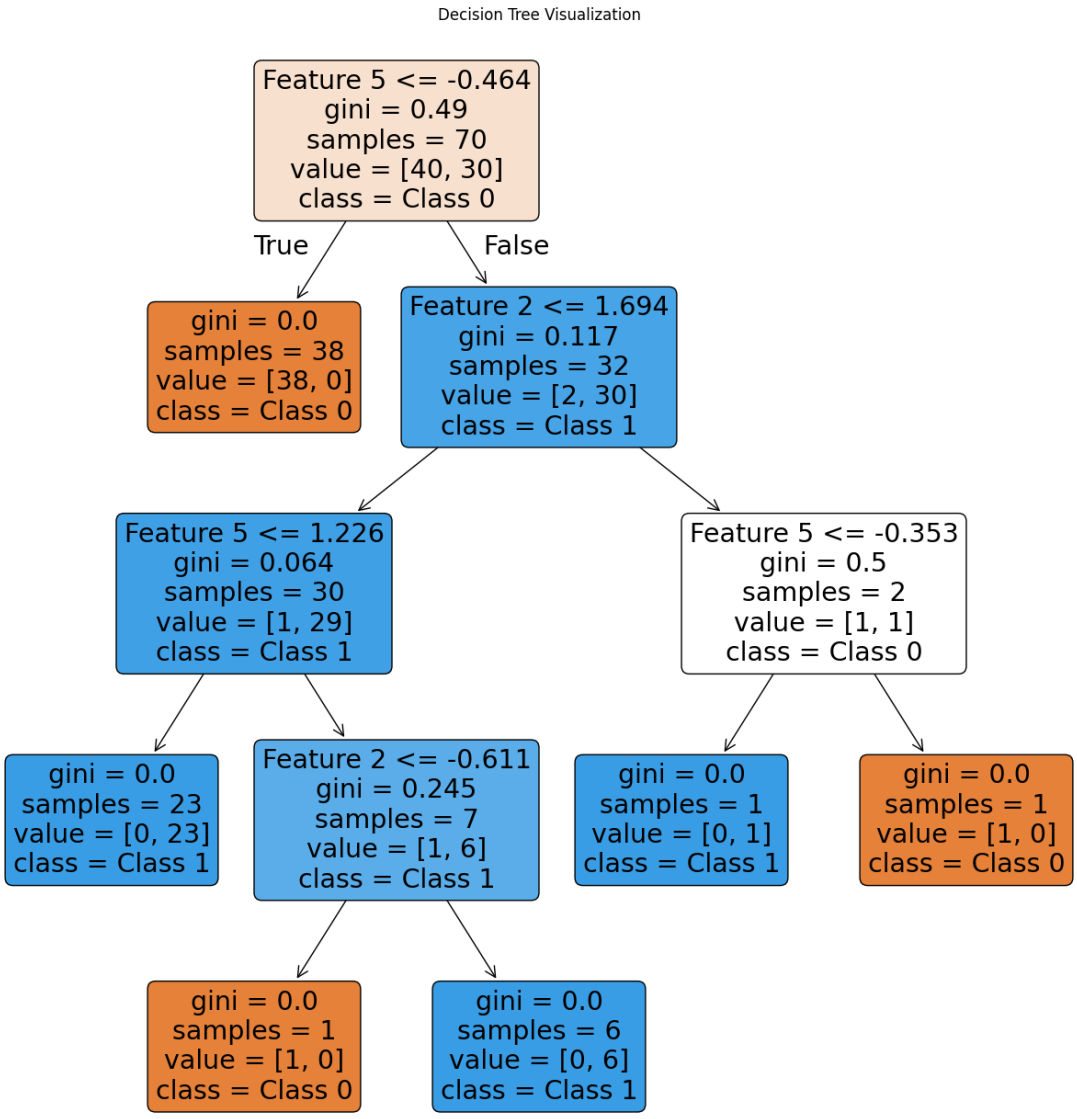

Figure 3: Our decision tree

This Python code demonstrates how to generate a random classification dataset, train a Decision Tree classifier on it, and visualize both the decision tree and the dataset points.

The generate_data function creates a synthetic dataset using make_classification from scikit-learn. The dataset is for a binary classification task with two classes, containing a user-specified number of features (default is 5). It also ensures that two features are informative while others are not redundant. After the dataset is generated, it is split into training and testing sets using train_test_split, allocating 70% for training and 30% for testing.

The code then creates a DecisionTreeClassifier and fits it to the training data using the fit() method. This allows the classifier to learn the decision rules based on the features. The trained model is used to make predictions on the test set, although these predictions are not displayed or used further in the code.

The decision tree itself is visualized with the plot_tree function from scikit-learn, displaying the tree structure with feature names and class labels. This helps in understanding how the tree splits the data based on different feature values.

If the dataset has only two features, the code also visualizes the data points using a scatter plot. The points are color-coded according to their class labels, which helps illustrate how the decision tree classifies them. If the dataset has more than two features, only the decision tree is shown, and a message is printed to inform that visualization of the data points is not supported for higher dimensions.

The function generate_data(n_features=5) can be called with a different number of features, allowing for flexibility. For instance, calling generate_data(n_features=2) will create a dataset with two features and visualize both the decision tree and the data points. This code is a useful tool for understanding how decision trees work, providing insights into both the structure of the tree and the results of the classification in low-dimensional spaces.

References

[1] Ali, A. (2024). *Decision Tree Classification in Python Tutorial*. Data Camp. Available at https://www.datacamp.com/tutorial/decision-tree-classification-python

[2] Han, J., Kamber, M. & Pei, J. (2012). *Data mining. Concepts and Techniques*. Elsevier.