Decision Tree from Scratch: A Comprehensive Guide

🌐 Follow Me on Social Media:

- LinkedIn Connect
- GitHub Follow
- Kaggle Follow
- colab View Notebooks
- Read Blog Post

Decision trees are one of the most popular and interpretable algorithms in machine learning, commonly used for both classification and regression tasks. They work by recursively splitting the dataset based on feature thresholds, creating a tree-like structure where each internal node represents a decision based on a feature, and each leaf node corresponds to an output label or value.

The main advantages of decision trees are their simplicity, ease of visualization, and ability to handle both numerical and categorical data. However, when implemented from scratch, careful handling of split criteria, stopping conditions, and data preprocessing is required to ensure the model performs optimally.

This document provides an in-depth exploration of implementing a decision tree from scratch in Python. It covers key concepts such as splitting data, calculating information gain using metrics like Gini Impurity and Entropy, and building the tree structure recursively. Additionally, the implementation accounts for various parameters like maximum tree depth and minimum samples required for a split to prevent overfitting.

By understanding and building a decision tree from the ground up, we gain valuable insights into the mechanics of tree-based algorithms and lay a strong foundation for extending these concepts to advanced methods like Random Forests and Gradient Boosted Trees.

In [1]:
```
import numpy as np
import pandas as pd
```

1. Gini Impurity Formula

$$Gini(y) = 1 - \sum_{i=1}^{k} p_i^2$$

- **Explanation**:
  - (p_i) is the proportion of class (i) in the dataset.
  - (k) is the total number of classes.
  - `np.unique(y, return_counts=True)` calculates the unique classes and their counts.
  - `probabilities = counts / len(y)` computes (p_i) for each class.
  - `1 - np.sum(probabilities**2)` computes $(Gini(y))$.

In [2]:
```
def gini_impurity(y):
    classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    return 1 - np.sum(probabilities**2)
```

2. Entropy Formula

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

- **Explanation**:
  - (p_i) is the proportion of class (i) in the dataset.
  - Entropy measures the "impurity" or "uncertainty" in the dataset.
  - $(\log_2(p_i))$ is calculated using `np.log2(probabilities + 1e-9)`.
  - Adding $(1e - 9)$ prevents numerical errors when $(p_i = 0)$.

In [3]:
```
def entropy(y):
    classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    return -np.sum(probabilities * np.log2(probabilities + 1e-9))  # Add small valu
```

In [4]:
```
def split_data(X, y, feature_index, threshold):
    left_indices = X[:, feature_index] <= threshold
    right_indices = X[:, feature_index] > threshold
    return X[left_indices], X[right_indices], y[left_indices], y[right_indices]
```

3. Information Gain Formula

$$Gain = Impurity_{parent} - \left(\frac{n_{left}}{n} \cdot Impurity_{left} + \frac{n_{right}}{n} \cdot Impurity_{right}\right)$$

- **Explanation**:
  - $(Impurity_{parent})$: Gini or Entropy of the parent node.
  - $(Impurity_{left})$: Gini or Entropy of the left child node.
  - $(Impurity_{right})$: Gini or Entropy of the right child node.
  - (n_{left}, n_{right}): Number of samples in the left and right child nodes.
  - (n): Total number of samples in the parent node.
  - The parent impurity is calculated as `impurity_function(y)`.
  - Weighted child impurity is calculated using the proportions $(\frac{n_{left}}{n})$ and $(\frac{n_{right}}{n})$.

In [5]:
```python
def information_gain(y, y_left, y_right, impurity_function=gini_impurity):
    parent_impurity = impurity_function(y)
    n = len(y)
    n_left, n_right = len(y_left), len(y_right)

    # Weighted impurity of children
    child_impurity = (n_left / n) * impurity_function(y_left) + (n_right / n) * imp

    return parent_impurity - child_impurity
```

The `Node` class represents a single node in the decision tree. Each node can either be:

1. **An internal node**: Contains information about a feature index and threshold used for splitting the data, along with pointers to its left and right child nodes.
2. **A leaf node**: Contains a classification value when further splits are no longer possible or desirable.

Here's an explanation of each parameter in the `Node` class:

---

1. `feature_index`

- **Purpose**:
  - Stores the index of the feature used for splitting at this node.
  - Example: If `feature_index = 2`, it means this node splits based on the third feature in the dataset.
- **Type**: Integer or `None`.
- **Usage**:
  - Used only in internal nodes. It is `None` for leaf nodes.

---

2. `threshold`

- **Purpose**:
  - Stores the threshold value for the feature used to split the data at this node.
  - Example: If `threshold = 5.5`, this node splits data into:
    - Left child: Samples where the feature value is ≤ 5.5.
    - Right child: Samples where the feature value is > 5.5.
- **Type**: Float or `None`.
- **Usage**:
  - Used only in internal nodes. It is `None` for leaf nodes.

---

3. `left`

- **Purpose**:
  - A reference to the left child node.
  - Represents the subset of data that satisfies the condition `feature_value <= threshold`.
- **Type**: Instance of `Node` or `None`.
- **Usage**:
  - Points to the left child in the decision tree structure.

---

4. `right`

- **Purpose**:
  - A reference to the right child node.
  - Represents the subset of data that satisfies the condition `feature_value > threshold`.
- **Type**: Instance of `Node` or `None`.
- **Usage**:
  - Points to the right child in the decision tree structure.

---

5. `value`

- **Purpose**:

- Stores the value of the prediction (or class label) at a **leaf node**.
- For classification tasks:
  - It's the most common label in the data at this node.
- For regression tasks:
  - It's the mean or another aggregation metric of the target values at this node.

- **Type**: Depends on the task:
  - For classification: Integer (class label).
  - For regression: Float (predicted value).
  - It is `None` for internal nodes.

Node Behavior

- **Internal Nodes**:
  - Contain `feature_index`, `threshold`, `left`, and `right`.
  - Example:
    ```
    Node(feature_index=1, threshold=2.5, left=left_node,
    right=right_node)
    ```
- **Leaf Nodes**:
  - Contain `value` and no references to children.
  - Example:
    ```
    Node(value=0)
    ```

Example Usage

Internal Node Example:

An internal node that splits based on the feature at index 2 with a threshold of 3.5:

```
node = Node(feature_index=2, threshold=3.5, left=left_child,
right=right_child)
```

Leaf Node Example:

A leaf node that predicts class `1`:

```
leaf = Node(value=1)
```

Integration with `DecisionTree`

- When building the tree (`_build_tree` method):
  - Internal nodes are created with `feature_index`, `threshold`, and pointers to child nodes.

- Leaf nodes are created with `value` when the stopping criteria are met.

In [6]:
```
class Node:
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, v
        self.feature_index = feature_index  # Index of feature to split
        self.threshold = threshold          # Threshold for splitting
        self.left = left                    # Left child
        self.right = right                  # Right child
        self.value = value                  # Leaf node value (for classification)
```

1. `__init__(self, max_depth=5, min_samples_split=2)`

- **Purpose**: Initializes the decision tree with two hyperparameters:
  - `max_depth` : The maximum depth the tree is allowed to grow.
  - `min_samples_split` : The minimum number of samples required to split a node.
- **Attributes**:
  - `self.root` : The root node of the decision tree, initialized as `None` .

---

2. `_build_tree(self, X, y, depth)`

- **Purpose**: Recursively builds the decision tree.
- **Steps**:
  1. **Check Stopping Criteria**:
     - Stops growing the tree if:
       - Maximum depth is reached.
       - Number of samples is less than `min_samples_split` .
       - All samples belong to the same class.
  2. **Find the Best Split**:
     - Loops over all features and possible thresholds.
     - Uses `information_gain` to evaluate each split.
     - Tracks the best feature and threshold.
  3. **No Valid Split**:
     - If no split improves the gain, create a **leaf node** with the most common label in `y` .
  4. **Split Data and Recur**:
     - Splits data into `X_left` and `X_right` .

    - Recursively calls `_build_tree` to build the left and right children.
- **Returns**: A `Node` object (either a leaf node or an internal node).

---

3. `_most_common_label(self, y)`

- **Purpose**: Finds the most common class in the given labels `y`.
- **Steps**:
    - Uses `np.unique` to count occurrences of each class.
    - Returns the class with the highest count using `np.argmax`.
- **Returns**: The majority class in `y`.

---

4. `fit(self, X, y)`

- **Purpose**: Fits (or trains) the decision tree on the training data.
- **Steps**:
    - Calls `_build_tree` with the training data `X`, labels `y`, and an initial depth of `0`.
    - Stores the resulting tree in `self.root`.

---

5. `_predict(self, x, node)`

- **Purpose**: Predicts the class for a single sample `x` by traversing the tree.
- **Steps**:
    - If the current `node` is a **leaf node**, return its value.
    - If `x[node.feature_index] <= node.threshold`, recurse into the left child.
    - Otherwise, recurse into the right child.
- **Returns**: The predicted class for the input sample `x`.

---

6. `predict(self, X)`

- **Purpose**: Predicts the class for all samples in the dataset `X`.
- **Steps**:
    - Loops through each sample in `X` and calls `_predict` for it.
    - Collects predictions into a NumPy array.
- **Returns**: A NumPy array of predictions for all samples.

---

7. `_count_nodes(self, node, counts)`

- **Purpose**: Recursively counts the number of nodes in the tree, including:
    - Root node
    - Internal nodes
    - Leaf nodes
- **Steps**:
    - If the node is a **leaf node** (its value is not `None`), increment the `leaves` count.
    - Otherwise, increment the `internal_nodes` count.
    - Recurse into the left and right children.

---

8. `count_nodes(self)`

- **Purpose**: Provides a summary of the number of different types of nodes in the tree.
- **Steps**:
    - Initializes a dictionary `counts` with:
        - `root`: 1 (always one root).
        - `internal_nodes`: 0.
        - `leaves`: 0.
    - Calls `_count_nodes` starting from the root node.
- **Returns**: A dictionary containing the counts of root, internal, and leaf nodes.

---

9. `_print_tree(self, node, depth=0)`

- **Purpose**: Recursively prints the structure of the decision tree.
- **Steps**:
    - For **leaf nodes**, print "Leaf Node: Class = ..." with indentation proportional to depth.
    - For **internal nodes**, print "Internal Node: Feature[...] <= ..." with the feature and threshold.
    - Recurse into the left and right children, increasing the depth.

---

10. `print_tree(self)`

- **Purpose**: Prints the entire tree structure starting from the root.
- **Steps**:

- Calls `_print_tree` with the root node and an initial depth of `0`.

Class Summary

This `DecisionTree` class:

1. **Builds a Tree**:
   - Using recursive splitting based on the best information gain.
2. **Predicts Classes**:
   - Traverses the tree to make predictions for given inputs.
3. **Analyzes the Tree**:
   - Counts the types of nodes.
   - Prints the tree structure.

In [31]:
```
class DecisionTree:
    def __init__(self, max_depth=5, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None

    def _build_tree(self, X, y, depth):
        n_samples, n_features = X.shape
        unique_classes = np.unique(y)

        # Stop criteria
        if depth >= self.max_depth or n_samples < self.min_samples_split or len(uni
            leaf_value = self._most_common_label(y)
            return Node(value=leaf_value)

        # Find the best split
        best_gain = -1
        best_feature, best_threshold = None, None

        for feature_index in range(n_features):
            thresholds = np.unique(X[:, feature_index])
            for threshold in thresholds:
                X_left, X_right, y_left, y_right = split_data(X, y, feature_index,
                if len(y_left) > 0 and len(y_right) > 0:
                    gain = information_gain(y, y_left, y_right)
                    if gain > best_gain:
                        best_gain, best_feature, best_threshold = gain, feature_ind

        # If no split improves the gain, create a leaf node
        if best_gain == -1:
            leaf_value = self._most_common_label(y)
            return Node(value=leaf_value)

        # Split the data
        X_left, X_right, y_left, y_right = split_data(X, y, best_feature, best_thre
```

```
        # Recursively build children
        left_child = self._build_tree(X_left, y_left, depth + 1)
        right_child = self._build_tree(X_right, y_right, depth + 1)

        return Node(feature_index=best_feature, threshold=best_threshold, left=left

    def _most_common_label(self, y):
        classes, counts = np.unique(y, return_counts=True)
        return classes[np.argmax(counts)]

    def fit(self, X, y):
        # Convert X and y to NumPy arrays if they are DataFrames or Series
        X = np.array(X)
        y = np.array(y)
        self.root = self._build_tree(X, y, 0)


    def _predict(self, x, node):
        if node.value is not None:
            return node.value

        if x[node.feature_index] <= node.threshold:
            return self._predict(x, node.left)
        else:
            return self._predict(x, node.right)

    def predict(self, X):
        return np.array([self._predict(x, self.root) for x in X])

    def _count_nodes(self, node, counts):
        if node is None:
            return

        if node.value is not None:  # Leaf node
            counts["leaves"] += 1
        else:  # Internal or root node
            counts["internal_nodes"] += 1

        # Recursively count for left and right children
        self._count_nodes(node.left, counts)
        self._count_nodes(node.right, counts)

    def count_nodes(self):
        counts = {"root": 1, "internal_nodes": 0, "leaves": 0}
        self._count_nodes(self.root, counts)
        return counts

    def _print_tree(self, node, depth=0):
        if node is None:
            return

        if node.value is not None:  # Leaf node
            print(f"{'|   ' * depth}Leaf Node: Class = {node.value}")
        else:
            print(f"{'|   ' * depth}Internal Node: Feature[{node.feature_index}] <=
```

```
        # Traverse left and right children
        self._print_tree(node.left, depth + 1)
        self._print_tree(node.right, depth + 1)

    def print_tree(self):
        print("Decision Tree Structure:")
        self._print_tree(self.root)
```

In [24]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
# data = load_iris()
# X, y = data.data, data.target

df = pd.read_csv('heart.csv')
df
```

Out[24]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	52	1	0	125	212	0	1	168	0	1.0	2	2	3
1	53	1	0	140	203	1	0	155	1	3.1	0	0	3
2	70	1	0	145	174	0	1	125	1	2.6	0	0	3
3	61	1	0	148	203	0	1	161	0	0.0	2	1	3
4	62	0	0	138	294	1	1	106	0	1.9	1	3	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1020	59	1	1	140	221	0	1	164	1	0.0	2	0	2
1021	60	1	0	125	258	0	0	141	1	2.8	1	1	3
1022	47	1	0	110	275	0	0	118	1	1.0	1	1	2
1023	50	0	0	110	254	0	0	159	0	0.0	2	0	2
1024	54	1	0	120	188	0	1	113	0	1.4	1	1	3

1025 rows × 14 columns

In [28]:
```
X = df.iloc[:, 0:-1]
y = df.iloc[:, -1]
```

In [29]:
```
# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [35]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
for column in X_train.columns:
```

```
    if X_train[column].dtype == 'object':  # Encode only non-numerical columns
        X_train[column] = le.fit_transform(X_train[column])
        X_test[column] = le.transform(X_test[column])

# Convert to NumPy arrays for compatibility
X_train = np.array(X_train)
X_test = np.array(X_test)
```

The following code shows the performance of the Build from Scratch tree algorithm

In [47]:

```
# Train decision tree
tree = DecisionTree(max_depth=10)
tree.fit(X_train, y_train)

# Make predictions
y_pred = tree.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Scratch Tree : {accuracy}")
```

```
Accuracy of Scratch Tree : 0.9853658536585366
```

The following code shows the performance of the Sklearn tree algorithm

In [48]:

```
from sklearn.tree import DecisionTreeClassifier

tree1 = DecisionTreeClassifier(max_depth=10)

tree1.fit(X_train, y_train)

predictions1 = tree1.predict(X_test)

accuracy1 = accuracy_score(y_test, predictions1)
print(f"Accuracy of sklearn Tree: {accuracy1}")
```

```
Accuracy of sklearn Tree: 0.9853658536585366
```

In [22]:

```
# Count the parameters of the tree
counts = tree.count_nodes()
print(f"Root Node: 1")
print(f"Internal Nodes: {counts['internal_nodes']}")
print(f"Leaf Nodes: {counts['leaves']}")
```

```
Root Node: 1
Internal Nodes: 2
Leaf Nodes: 3
```

In [23]:

```
# Print the decision tree
tree.print_tree()
```

```
Decision Tree Structure:
Internal Node: Feature[2] <= 1.9
|   Leaf Node: Class = 0
|   Internal Node: Feature[2] <= 4.7
|   |   Leaf Node: Class = 1
|   |   Leaf Node: Class = 2
```