[3]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

[4]:
```python
class Node:
    """
    Represents a node in the decision tree.

    Attributes:
    -----------
    feature : int, optional
        Index of the feature used for splitting
    threshold : float, optional
        Threshold value for feature splitting
    left : Node, optional
        Left child node
    right : Node, optional
        Right child node
    value : int, optional
        Predicted class for leaf nodes
    info_gain : float, optional
        Information gain at this node
    """
    def __init__(self, feature=None, threshold=None, left=None, right=None,
 info_gain=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        self.value = value
```

[5]:
```python
class DecisionTree:
    """
    Custom Decision Tree Classifier implementation.

    Parameters:
    -----------
    max_depth : int, default=5
        Maximum depth of the tree
    min_samples_split : int, default=2
        Minimum number of samples required to split an internal node

    Methods:
    --------
    entropy(y): Calculate entropy of a dataset
    information_gain(parent, left_child, right_child): Calculate information
 ↪gain
    best_split(X, y): Find the best feature and threshold for splitting
    build_tree(X, y, depth): Recursively build the decision tree
    predict_single(x, node): Predict class for a single sample
    predict(X): Predict classes for multiple samples
    fit(X, y): Train the decision tree
    score(X, y): Calculate accuracy of the model
    """
    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 entropy(self, y):
        """
        Calculate the entropy of a dataset.

        Parameters:
        -----------
        y : array-like
            Target variable

        Returns:
        --------
        float
            Entropy value
        """
        # Prevent log(0) by adding small epsilon
        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        return -np.sum(probabilities * np.log2(probabilities + 1e-10))
```

```python
    def information_gain(self, parent, left_child, right_child):
        """
        Calculate information gain for a split.

        Parameters:
        -----------
        parent : array-like
            Original dataset
        left_child : array-like
            Left split of the dataset
        right_child : array-like
            Right split of the dataset

        Returns:
        --------
        float
            Information gain of the split
        """
        parent_entropy = self.entropy(parent)
        left_entropy = self.entropy(left_child)
        right_entropy = self.entropy(right_child)

        n = len(parent)
        left_weight = len(left_child) / n
        right_weight = len(right_child) / n

        # Calculate weighted entropy reduction
        gain = parent_entropy - (left_weight * left_entropy + right_weight *
right_entropy)
        return gain

    def best_split(self, X, y):
        """
        Find the best feature and threshold for splitting.

        Parameters:
        -----------
        X : array-like
            Feature matrix
        y : array-like
            Target variable

        Returns:
        --------
        tuple
            Best feature index and threshold value
        """
```

```
        best_gain = -1
        best_feature = None
        best_threshold = None

        for feature in range(X.shape[1]):
            # Use unique values as potential thresholds
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                # Split the data
                left_mask = X[:, feature] <= threshold
                right_mask = ~left_mask

                # Skip if split creates too small subsets
                if len(y[left_mask]) < self.min_samples_split or len(y[right_mask]) < self.min_samples_split:
                    continue

                # Calculate information gain
                gain = self.information_gain(y, y[left_mask], y[right_mask])

                # Update best split if gain is improved
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold

        return best_feature, best_threshold

    def build_tree(self, X, y, depth=0):
        """
        Recursively build the decision tree.

        Parameters:
        -----------
        X : array-like
            Feature matrix
        y : array-like
            Target variable
        depth : int, optional
            Current depth of the tree

        Returns:
        --------
        Node
            Root node of the (sub)tree
        """
        n_samples, n_features = X.shape
```

```
        unique_classes = np.unique(y)

        # Stopping criteria
        if (depth >= self.max_depth or
            len(unique_classes) == 1 or
            n_samples < self.min_samples_split):
            # Return most common class as leaf node
            leaf_value = np.bincount(y).argmax()
            return Node(value=leaf_value)

        # Find best split
        best_feature, best_threshold = self.best_split(X, y)

        # If no good split is found, create a leaf node
        if best_feature is None:
            leaf_value = np.bincount(y).argmax()
            return Node(value=leaf_value)

        # Create split masks
        left_mask = X[:, best_feature] <= best_threshold
        right_mask = ~left_mask

        # Recursive tree building
        left = self.build_tree(X[left_mask], y[left_mask], depth+1)
        right = self.build_tree(X[right_mask], y[right_mask], depth+1)

        return Node(feature=best_feature,
                    threshold=best_threshold,
                    left=left,
                    right=right)

    def predict_single(self, x, node):
        """
        Predict class for a single sample by traversing the tree.

        Parameters:
        -----------
        x : array-like
            Single sample
        node : Node
            Current node in tree traversal

        Returns:
        --------
        int
            Predicted class
        """
```

```python
        # Leaf node reached
        if node.value is not None:
            return node.value

        # Recursive tree traversal
        if x[node.feature] <= node.threshold:
            return self.predict_single(x, node.left)
        return self.predict_single(x, node.right)

    def predict(self, X):
        """
        Predict classes for multiple samples.

        Parameters:
        -----------
        X : array-like
            Feature matrix

        Returns:
        --------
        ndarray
            Predicted classes
        """
        return np.array([self.predict_single(x, self.root) for x in X])

    def fit(self, X, y):
        """
        Train the decision tree.

        Parameters:
        -----------
        X : array-like
            Feature matrix
        y : array-like
            Target variable
        """
        self.root = self.build_tree(X, y)

    def score(self, X, y):
        """
        Calculate model accuracy.

        Parameters:
        -----------
        X : array-like
            Feature matrix
        y : array-like
```

```
            True labels

        Returns:
        --------
        float
            Accuracy score
        """
        predictions = self.predict(X)
        return np.mean(predictions == y)
```

[6]:
```
def plot_decision_tree(node, depth=0, prefix="root"):
    """
    Print decision tree structure for visualization.

    Parameters:
    -----------
    node : Node
        Current node to visualize
    depth : int, optional
        Current depth in the tree
    prefix : str, optional
        Prefix for current node (Left/Right/Root)
    """
    if node.value is not None:
        print("  " * depth + f"{prefix} Leaf: Class {node.value}")
        return

    print("  " * depth + f"{prefix} Split: Feature {node.feature}, Threshold
 ↪{node.threshold:.4f}")
    plot_decision_tree(node.left, depth+1, "Left")
    plot_decision_tree(node.right, depth+1, "Right")
```

[7]:
```
# Import preprocessing tools
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

[8]:
```
# Load and preprocess data
diabetes = load_diabetes()
X, y = diabetes.data, (diabetes.target > diabetes.target.mean()).astype(int)
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 ↪random_state=42)
```

[9]:
```
# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)
```

[10]:
```
# Train the model
dt = DecisionTree(max_depth=5, min_samples_split=10)
dt.fit(X_train_scaled, y_train)
```

[11]:
```
# Evaluate
train_accuracy = dt.score(X_train_scaled, y_train)
test_accuracy = dt.score(X_test_scaled, y_test)

print(f"Train Accuracy: {train_accuracy:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
```

```
Train Accuracy: 0.8130
Test Accuracy: 0.7303
```

[12]:
```
# Visualize tree structure
print("\nDecision Tree Structure:")
plot_decision_tree(dt.root)
```

```
Decision Tree Structure:
root Split: Feature 2, Threshold 0.2199
  Left Split: Feature 8, Threshold -
    0.0198 Left Split: Feature 6,
    Threshold 0.4295
      Left Split: Feature 2, Threshold -
      0.3509
        Left Split: Feature 9, Threshold -
        0.7473
          Left Leaf: Class 0
          Right Leaf: Class 0
        Right Leaf: Class 0
      Right Split: Feature 3, Threshold -
      0.2838
        Left Leaf: Class 0
        Right Split: Feature 0, Threshold
        0.6348
          Left Leaf: Class 0
          Right Leaf: Class 0
 Right Split: Feature 5, Threshold -0.5900
      Left Leaf: Class 1
      Right Split: Feature 6, Threshold
      0.6646
        Left Split: Feature 4, Threshold -
        0.1366
          Left Leaf: Class 0
```

```
        Right Leaf: Class 1
      Right Leaf: Class 0
  Right Split: Feature 3, Threshold 0.2853
 Left Split: Feature 9, Threshold -0.1472
     Left Leaf: Class 0
     Right Split: Feature 8, Threshold
     0.9405
      Left Split: Feature 2, Threshold
      0.6080
       Left Leaf: Class 1
       Right Leaf: Class 0
      Right Leaf: Class 1
Right Split: Feature 8, Threshold -0.4047
     Left Leaf: Class 1
 Right Split: Feature 2, Threshold 0.5167
      Left Leaf: Class 1
      Right Leaf: Class 1
```

[ ]: