Overview

A decision tree classifier recursively splits the dataset based on feature values to create a tree structure where leaf nodes represent class predictions. This implementation uses Gini impurity to measure node purity and finds optimal splits by evaluating information gain for each feature. The code assumes numerical features and binary classification for simplicity, but can be extended.

Gini Impurity Calculation

Gini impurity quantifies the probability of misclassifying a randomly selected element in a node, with lower values indicating purer nodes. It is calculated as $ Gini = 1 - \sum (p_i)^2 $, where $ p_i $ is the proportion of class $ i $ in the node. For a split, information gain is the reduction in Gini impurity: $ Gain = Gini_{parent} - \sum (w_j \cdot Gini_j) $, where $ w_j $ is the weight of child node $ j $.

To compute Gini for a set of labels, count class occurrences and apply the formula. This metric is preferred over entropy for its computational efficiency in decision trees.

```
import numpy as np

def gini_impurity(y):
    """
    Calculate Gini impurity for a set of labels.
    y: array of labels
    """
    if len(y) == 0:
        return 0
    classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    gini = 1 - np.sum(probabilities ** 2)
    return gini
```

This function handles empty nodes by returning 0 and uses NumPy for efficient computation. The loop-free approach ensures scalability for larger datasets.

Information Gain for Splits

To find the best split, evaluate possible thresholds for each feature and compute the resulting Gini gain. For continuous features, thresholds are midpoints between sorted unique values to avoid redundant splits. The best split maximizes information gain, partitioning data into left (≤ threshold) and right (> threshold) subsets.

```
def information_gain(y, y_left, y_right):
    """
    Calculate information gain from a split.
    y: parent labels
    y_left, y_right: labels in child nodes
    """
    parent_gini = gini_impurity(y)
    n = len(y)
    n_left = len(y_left)
    n_right = len(y_right)
    if n_left == 0 or n_right == 0:
        return 0
    weighted_gini = (n_left / n) * gini_impurity(y_left) + (n_right / n) * gini_impurity(
    return parent_gini - weighted_gini

def best_split(X, y):
    """
    Find the best feature and threshold for splitting.
    X: features (n_samples, n_features)
    y: labels
    Returns: best_feature_index, best_threshold, best_gain
    """
    n_samples, n_features = X.shape
    best_gain = -1
    best_feature = None
    best_threshold = None

    for feature in range(n_features):
        thresholds = np.unique(X[:, feature])
        if len(thresholds) < 2:
            continue
        for i in range(1, len(thresholds)):
            threshold = (thresholds[i-1] + thresholds[i]) / 2
            left_mask = X[:, feature] <= threshold
            right_mask = ~left_mask
            y_left = y[left_mask]
            y_right = y[right_mask]
            gain = information_gain(y, y_left, y_right)
            if gain > best_gain:
                best_gain = gain
                best_feature = feature
                best_threshold = threshold

    return best_feature, best_threshold, best_gain
```

The `best_split` function iterates over features and potential thresholds, using masks for efficient partitioning without copying data. It skips features with insufficient unique values to avoid invalid splits.

Node Structure

Each node in the tree stores a split decision or a leaf prediction. Internal nodes hold the feature index, threshold, and child nodes; leaves store the majority class label.

```
class DecisionNode:
    """
    Represents a node in the decision tree.
    """
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, value=N
        self.feature_index = feature_index  # Index of feature to split on
        self.threshold = threshold  # Threshold for split
        self.left = left  # Left child node
        self.right = right  # Right child node
        self.value = value  # Class label for leaf node
```

This class uses None for leaf indicators, allowing easy traversal. The value is set to the most common class in the subset during construction.

Recursive Tree Building

The tree is built recursively: at each node, find the best split if possible; otherwise, make it a leaf. Stopping criteria include pure nodes (all same class), minimum samples per leaf, or maximum depth to prevent overfitting.

```
def build_tree(X, y, max_depth=10, min_samples_split=2, depth=0):
    """
    Recursively build the decision tree.
    X: features
    y: labels
    Returns: DecisionNode
    """
    n_samples = len(y)
    if n_samples < min_samples_split or depth >= max_depth or len(np.unique(y)) == 1:
        # Leaf node: majority class
        leaf_value = np.bincount(y).argmax()
        return DecisionNode(value=leaf_value)

    feature, threshold, gain = best_split(X, y)
    if gain == 0:
        # No gain: make leaf
        leaf_value = np.bincount(y).argmax()
        return DecisionNode(value=leaf_value)

    # Split data
    left_mask = X[:, feature] <= threshold
    right_mask = ~left_mask
    X_left, y_left = X[left_mask], y[left_mask]
    X_right, y_right = X[right_mask], y[right_mask]

    # Recurse
    left_child = build_tree(X_left, y_left, max_depth, min_samples_split, depth + 1)
    right_child = build_tree(X_right, y_right, max_depth, min_samples_split, depth + 1)
    return DecisionNode(feature_index=feature, threshold=threshold, left=left_child, righ
```

Recursion starts at depth 0 and increments per level, ensuring controlled growth. Masks enable in-place splitting for memory efficiency. If no split improves purity, it defaults to a leaf.

Prediction Function

To predict, traverse from root to leaf by comparing feature values against thresholds, selecting left or right child accordingly. For a dataset, apply this to each sample.

```
def predict(node, x):
    """
    Predict class for a single sample.
    node: root DecisionNode
    x: feature vector
    """
    if node.value is not None:
        return node.value
    if x[node.feature_index] <= node.threshold:
        return predict(node.left, x)
    else:
        return predict(node.right, x)

def predict_batch(root, X):
    """
    Predict for multiple samples.
    """
    return np.array([predict(root, x) for x in X])
```

Traversal is O(depth) per prediction, efficient for shallow trees. The batch version vectorizes over samples using list comprehension for clarity.

Complete DecisionTree Class

Wrap the components into a class for easy usage, similar to scikit-learn's API.

```
class DecisionTreeClassifier:
    """
    Decision Tree Classifier from scratch.
    """
    def __init__(self, max_depth=10, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None

    def fit(self, X, y):
        self.root = build_tree(X, y, self.max_depth, self.min_samples_split)

    def predict(self, X):
        if self.root is None:
            raise ValueError("Tree not fitted yet.")
        return predict_batch(self.root, X)
```

This class initializes parameters, fits by building the tree, and predicts using the batch function. Labels should be integers for np.bincount.

Example Usage

Load a sample dataset like Iris, train, and evaluate. Assume y is encoded as 0,1,2.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load data
iris = load_iris()
X, y = iris.data, iris.target  # Binary: keep only two classes for simplicity
X, y = X[y < 2], y[y < 2]  # Setosa vs Versicolor

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train
clf = DecisionTreeClassifier(max_depth=3, min_samples_split=5)
clf.fit(X_train, y_train)

# Predict
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

This example uses scikit-learn for data and evaluation but implements the core tree without its algorithms. Adjust max_depth to balance bias-variance; higher values risk overfitting on small datasets.

⁂

Modifying Leaf Nodes for Probabilities

Decision trees calculate probabilities by storing the class distribution in each leaf node rather than just the majority class. When a sample reaches a leaf, the probability for class $ k $ is computed as the ratio of training samples of that class in the leaf to the total samples in that leaf. This requires updating the node structure to store class counts.[21] [22]

```
import numpy as np

class DecisionNode:
    """
    Enhanced node that stores class distribution for probabilities.
    """
    def __init__(self, feature_index=None, threshold=None, left=None, right=None,
                 value=None, class_counts=None):
        self.feature_index = feature_index  # Feature to split on
        self.threshold = threshold  # Split threshold
        self.left = left  # Left child
        self.right = right  # Right child
        self.value = value  # Predicted class (majority)
        self.class_counts = class_counts  # Distribution of classes in this node
```

The `class_counts` attribute stores an array where each index represents a class and the value is the count of training samples. For example, `[^2_21][^2_22][^2_23]` means 10 samples of class 0, 5 of class 1, and 3 of class 2 in this leaf.[23] [22]

Updated Tree Building with Class Distributions

Modify the `build_tree` function to track class distributions at each leaf node. This requires counting occurrences of each class and storing them in the node structure.

```
def build_tree_with_proba(X, y, n_classes, max_depth=10, min_samples_split=2, depth=0):
    """
    Build tree that tracks class distributions for probability predictions.
    X: features
    y: labels
    n_classes: total number of classes in dataset
    """
    n_samples = len(y)

    # Calculate class counts for this node
    class_counts = np.bincount(y, minlength=n_classes)

    # Stopping criteria: make leaf node
    if n_samples < min_samples_split or depth >= max_depth or len(np.unique(y)) == 1:
        leaf_value = np.argmax(class_counts)  # Majority class
        return DecisionNode(value=leaf_value, class_counts=class_counts)

    # Find best split
    feature, threshold, gain = best_split(X, y)
    if gain == 0:
        leaf_value = np.argmax(class_counts)
        return DecisionNode(value=leaf_value, class_counts=class_counts)

    # Partition data
    left_mask = X[:, feature] <= threshold
    right_mask = ~left_mask
    X_left, y_left = X[left_mask], y[left_mask]
    X_right, y_right = X[right_mask], y[right_mask]

    # Recursively build children
    left_child = build_tree_with_proba(X_left, y_left, n_classes, max_depth, min_samples_
    right_child = build_tree_with_proba(X_right, y_right, n_classes, max_depth, min_sampl

    return DecisionNode(feature_index=feature, threshold=threshold,
                       left=left_child, right=right_child, class_counts=class_counts)
```

The `n_classes` parameter ensures `np.bincount` creates arrays of consistent length even if some classes are absent in a subset. Using `minlength` prevents dimension mismatches during prediction. Internal nodes also store counts for reference, though only leaf counts matter for probabilities.[21]

Probability Prediction Function

To predict probabilities, traverse to the leaf node and normalize its class counts to sum to 1. This gives the estimated probability distribution over all classes.[22]

```
def predict_proba_single(node, x, n_classes):
    """
    Predict class probabilities for a single sample.
    node: root DecisionNode
    x: feature vector
    n_classes: number of classes
    Returns: array of probabilities for each class
    """
    # Traverse to leaf
    if node.value is not None:  # Leaf node
        # Convert counts to probabilities
        total_samples = np.sum(node.class_counts)
        if total_samples == 0:
            return np.ones(n_classes) / n_classes  # Uniform if empty
        probabilities = node.class_counts / total_samples
        return probabilities

    # Navigate based on threshold
    if x[node.feature_index] <= node.threshold:
        return predict_proba_single(node.left, x, n_classes)
    else:
        return predict_proba_single(node.right, x, n_classes)

def predict_proba_batch(root, X, n_classes):
    """
    Predict probabilities for multiple samples.
    Returns: array of shape (n_samples, n_classes)
    """
    return np.array([predict_proba_single(root, x, n_classes) for x in X])
```

The probability for class $ k $ is $ P(k) = \frac{count_k}{\sum counts} $. Each row sums to 1.0, representing a valid probability distribution. The uniform fallback handles edge cases of empty nodes gracefully.[22] [21]

Complete Classifier with Probability Support

Integrate probability prediction into the main classifier class, adding a `predict_proba` method alongside `predict`.

```
class DecisionTreeClassifier:
    """
    Decision Tree with probability prediction support.
    """
    def __init__(self, max_depth=10, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None
        self.n_classes = None

    def fit(self, X, y):
        """
        Train the decision tree.
        X: features (n_samples, n_features)
        y: labels (n_samples,) - must be integers 0 to n_classes-1
        """
        self.n_classes = len(np.unique(y))
        self.root = build_tree_with_proba(X, y, self.n_classes,
                                          self.max_depth, self.min_samples_split)
        return self

    def predict(self, X):
        """
        Predict class labels.
        Returns: array of shape (n_samples,)
        """
        if self.root is None:
            raise ValueError("Model not fitted yet.")
        return np.array([self._predict_single(self.root, x) for x in X])

    def _predict_single(self, node, x):
        """Helper to predict single sample."""
        if node.value is not None:
            return node.value
        if x[node.feature_index] <= node.threshold:
            return self._predict_single(node.left, x)
        else:
            return self._predict_single(node.right, x)

    def predict_proba(self, X):
        """
        Predict class probabilities.
        Returns: array of shape (n_samples, n_classes)
        Each row sums to 1.0 and represents P(class | sample)
        """
        if self.root is None:
            raise ValueError("Model not fitted yet.")
        return predict_proba_batch(self.root, X, self.n_classes)
```

The `fit` method now automatically detects the number of classes from the training labels. The `predict_proba` method returns a 2D array where each row corresponds to a sample and each column to a class probability.[23] [21] [22]

Example with Probability Predictions

Demonstrate using the Iris dataset with all three classes to show probability outputs.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, log_loss

# Load full Iris dataset (3 classes)
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train classifier
clf = DecisionTreeClassifier(max_depth=4, min_samples_split=5)
clf.fit(X_train, y_train)

# Predict classes
y_pred = clf.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.3f}")

# Predict probabilities
y_proba = clf.predict_proba(X_test)
print(f"\nProbability predictions for first 5 samples:")
print(y_proba[:5])
print(f"\nSum of probabilities per sample: {y_proba[:5].sum(axis=1)}")

# Log loss (lower is better, measures probability quality)
loss = log_loss(y_test, y_proba)
print(f"Log Loss: {loss:.3f}")
```

The output shows probabilities for each of the three Iris classes (Setosa, Versicolor, Virginica). Rows sum to 1.0, confirming valid probability distributions. Log loss evaluates how well the predicted probabilities match actual labels—lower values indicate better calibrated probabilities. [21] [22]

Key Differences from Hard Predictions

Hard predictions (`predict`) return only the class with maximum probability, losing information about uncertainty. Probability predictions preserve this: a sample with probabilities `[0.51, 0.49]` is much less certain than `[0.95, 0.05]`, even though both predict class 0. This is crucial for applications requiring confidence estimates, such as medical diagnosis or fraud detection where knowing prediction certainty matters as much as the prediction itself. [23] [22]

⁂

Overview

A decision tree regressor predicts continuous values by recursively partitioning the feature space to minimize variance within each region. Unlike classification trees that use Gini impurity, regression trees use **Mean Squared Error (MSE)** or variance reduction as the splitting criterion. Leaf nodes predict the mean of training samples that reach them.[41] [42] [43]

Variance and MSE Calculation

For regression, node impurity is measured by variance or MSE, which quantifies how spread out the target values are. The formula is $ MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \bar{y})^2 $, where $ \bar{y} $ is the mean of values in the node. A split is good if it reduces the weighted MSE of child nodes compared to the parent.[44] [42] [41]

```
import numpy as np

def mse(y):
    """
    Calculate Mean Squared Error (variance) for a set of values.
    y: array of target values
    """
    if len(y) == 0:
        return 0
    mean_y = np.mean(y)
    return np.mean((y - mean_y) ** 2)
```

This function computes the average squared distance from the mean. Lower MSE indicates more homogeneous values, making the node a better predictor. The computation is vectorized for efficiency with large datasets.[42] [41]

Variance Reduction for Splits

Variance reduction measures how much a split decreases MSE. It's calculated as $ Reduction = MSE_{parent} - (w_{left} \cdot MSE_{left} + w_{right} \cdot MSE_{right}) $, where $ w $ represents the weight (proportion) of samples in each child.[41] [44] [42]

```
def variance_reduction(y, y_left, y_right):
    """
    Calculate variance reduction from a split.
    y: parent target values
    y_left, y_right: target values in child nodes
    """
    parent_mse = mse(y)
    n = len(y)
    n_left = len(y_left)
    n_right = len(y_right)

    if n_left == 0 or n_right == 0:
        return 0

    weighted_mse = (n_left / n) * mse(y_left) + (n_right / n) * mse(y_right)
    return parent_mse - weighted_mse

def best_split_regressor(X, y):
    """
    Find the best feature and threshold for splitting (regression).
    X: features (n_samples, n_features)
    y: continuous target values
    Returns: best_feature_index, best_threshold, best_reduction
    """
    n_samples, n_features = X.shape
    best_reduction = -1
    best_feature = None
    best_threshold = None

    for feature in range(n_features):
        thresholds = np.unique(X[:, feature])
        if len(thresholds) < 2:
            continue

        for i in range(1, len(thresholds)):
            threshold = (thresholds[i-1] + thresholds[i]) / 2
            left_mask = X[:, feature] <= threshold
            right_mask = ~left_mask

            y_left = y[left_mask]
            y_right = y[right_mask]

            reduction = variance_reduction(y, y_left, y_right)

            if reduction > best_reduction:
                best_reduction = reduction
                best_feature = feature
                best_threshold = threshold

    return best_feature, best_threshold, best_reduction
```

The function evaluates all possible splits and selects the one maximizing variance reduction. Thresholds are computed as midpoints between consecutive unique values to ensure efficient splitting. This is equivalent to minimizing the sum of squared residuals in child nodes.[45] [44] [42]

Node Structure for Regression

Regression tree nodes store the mean prediction value for leaf nodes instead of class labels. The structure also tracks variance for confidence estimation.[43] [41]

```
class RegressionNode:
    """
    Represents a node in the regression tree.
    """
    def __init__(self, feature_index=None, threshold=None, left=None, right=None,
                 value=None, variance=None):
        self.feature_index = feature_index  # Feature to split on
        self.threshold = threshold  # Split threshold
        self.left = left  # Left child node
        self.right = right  # Right child node
        self.value = value  # Predicted value (mean for leaf)
        self.variance = variance  # Variance of values in this node
```

The `value` stores the mean of target values for samples reaching this node. The `variance` attribute helps quantify prediction uncertainty, useful for confidence intervals.[42] [43] [41]

Recursive Tree Building

Building follows the same recursive pattern as classification, but stops when further splits don't reduce variance significantly.[44] [45]

```
def build_regression_tree(X, y, max_depth=10, min_samples_split=2, min_variance_reductio
    """
    Recursively build the regression tree.
    X: features
    y: continuous target values
    min_variance_reduction: minimum reduction required to split
    """
    n_samples = len(y)
    node_variance = mse(y)

    # Stopping criteria: make leaf node
    if (n_samples < min_samples_split or
        depth >= max_depth or
        node_variance < 1e-7):  # Already homogeneous
        leaf_value = np.mean(y)
        return RegressionNode(value=leaf_value, variance=node_variance)

    # Find best split
    feature, threshold, reduction = best_split_regressor(X, y)

    if reduction < min_variance_reduction:
        # Not enough improvement
        leaf_value = np.mean(y)
        return RegressionNode(value=leaf_value, variance=node_variance)

    # Partition data
    left_mask = X[:, feature] <= threshold
    right_mask = ~left_mask
    X_left, y_left = X[left_mask], y[left_mask]
    X_right, y_right = X[right_mask], y[right_mask]

    # Recursively build children
    left_child = build_regression_tree(X_left, y_left, max_depth, min_samples_split,
                                       min_variance_reduction, depth + 1)
    right_child = build_regression_tree(X_right, y_right, max_depth, min_samples_split,
                                        min_variance_reduction, depth + 1)

    return RegressionNode(feature_index=feature, threshold=threshold,
                          left=left_child, right=right_child, variance=node_variance)
```

The `min_variance_reduction` parameter prevents splits that provide minimal improvement, acting as early stopping. When variance is near zero, all values are identical, making further splits unnecessary. Leaf predictions are computed as the mean of all training samples in that region. [43] [41] [42]

Prediction Functions

Predictions traverse the tree to find the appropriate leaf and return its stored mean value. [45] [43]

```
def predict_single(node, x):
    """
    Predict continuous value for a single sample.
    node: root RegressionNode
    x: feature vector
    """
    if node.value is not None:  # Leaf node
        return node.value

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

def predict_batch(root, X):
    """
    Predict for multiple samples.
    """
    return np.array([predict_single(root, x) for x in X])
```

Each prediction is the mean of training targets that landed in the same leaf during tree construction. This creates a piecewise constant function over the feature space. [41] [45] [43]

Prediction Variance (Confidence Estimation)

Unlike classifiers, regressors can return prediction variance to indicate uncertainty. Higher variance in a leaf suggests less reliable predictions. [42]

```
def predict_with_variance(node, x):
    """
    Predict value and variance for a single sample.
    Returns: (predicted_value, variance)
    """
    if node.value is not None:  # Leaf node
        return node.value, node.variance

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

def predict_batch_with_variance(root, X):
    """
    Predict values and variances for multiple samples.
    Returns: (predictions, variances)
    """
    results = [predict_with_variance(root, x) for x in X]
    predictions = np.array([r[^3_0] for r in results])
    variances = np.array([r[^3_1] for r in results])
    return predictions, variances
```

The variance indicates the spread of training data in each leaf. Leaves with lower variance produce more confident predictions, while high variance suggests the model is uncertain.[41] [42]

Complete DecisionTreeRegressor Class

Wrap everything into a class compatible with scikit-learn's interface.[46] [45]

```
class DecisionTreeRegressor:
    """
    Decision Tree Regressor from scratch.
    """
    def __init__(self, max_depth=10, min_samples_split=2, min_variance_reduction=0.0):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_variance_reduction = min_variance_reduction
        self.root = None

    def fit(self, X, y):
        """
        Train the regression tree.
        X: features (n_samples, n_features)
        y: continuous target values (n_samples,)
        """
        self.root = build_regression_tree(X, y, self.max_depth,
                                          self.min_samples_split,
                                          self.min_variance_reduction)
        return self

    def predict(self, X):
        """
        Predict continuous values.
        Returns: array of shape (n_samples,)
        """
        if self.root is None:
            raise ValueError("Model not fitted yet.")
        return predict_batch(self.root, X)

    def predict_with_variance(self, X):
        """
        Predict with uncertainty estimates.
        Returns: (predictions, variances) as two arrays
        """
        if self.root is None:
            raise ValueError("Model not fitted yet.")
        return predict_batch_with_variance(self.root, X)
```

The class follows standard conventions with `fit` and `predict` methods. The additional `predict_with_variance` method provides uncertainty quantification not available in classification trees.[46] [45] [42]

Example Usage

Demonstrate with a synthetic regression dataset and compare to ground truth.[47] [45]

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Generate synthetic regression data
X, y = make_regression(n_samples=200, n_features=1, noise=10, random_state=42)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train regressor
regressor = DecisionTreeRegressor(max_depth=5, min_samples_split=5, min_variance_reductic
regressor.fit(X_train, y_train)

# Predict
y_pred = regressor.predict(X_test)

# Evaluate
mse_score = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse_score:.2f}")
print(f"R² Score: {r2:.3f}")

# Predict with variance
y_pred_var, variances = regressor.predict_with_variance(X_test)
print(f"\nPredictions with variance for first 5 samples:")
for i in range(5):
    print(f"  Predicted: {y_pred_var[i]:.2f}, Variance: {variances[i]:.2f}")
```

The R^2 score measures how well the model explains variance in the target, with 1.0 being perfect. MSE quantifies average squared prediction error—lower is better. Variance estimates help identify which predictions are more reliable based on training data distribution.[47] [42]

Key Differences from Classification

Regression trees predict continuous values using mean instead of mode. They optimize MSE rather than Gini impurity. Unlike classifiers that output class probabilities, regressors can provide variance estimates to quantify prediction uncertainty. The splitting logic remains similar, but the evaluation metric and leaf predictions fundamentally differ.[44] [45] [43] [42] [41]

⁂

1. https://www.datacamp.com/tutorial/decision-tree-classification-python

2. https://www.kaggle.com/code/fareselmenshawii/decision-tree-from-scratch
3. https://towardsdatascience.com/decision-trees-explained-entropy-information-gain-gini-index-ccp-pruning-4d78070db36c/
4. https://bradleyboehmke.github.io/HOML/DT.html
5. https://www.dataquest.io/blog/tutorial-decision-trees-in-machine-learning/
6. https://www.machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/
7. https://quantdare.com/decision-trees-gini-vs-entropy/
8. https://en.wikipedia.org/wiki/Recursive_partitioning
9. https://www.w3schools.com/python/python_ml_decision_tree.asp
10. https://www.youtube.com/watch?v=NxEHSAfFlK8
11. https://www.geeksforgeeks.org/machine-learning/gini-impurity-and-entropy-in-decision-tree-ml/
12. https://www.youtube.com/watch?v=sgQAhG5Q7iY
13. https://community.deeplearning.ai/t/decision-tree-recursive-splitting/566468
14. https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/
15. https://www.geeksforgeeks.org/machine-learning/decision-tree-implementation-python/
16. https://codesignal.com/learn/courses/classification-algorithms-and-metrics/lessons/building-a-decision-tree-from-scratch-in-python
17. https://github.com/fakemonk1/decision-tree-implementation-from-scratch
18. https://datasciencedojo.com/blog/gini-index-and-entropy/
19. https://vidushiratra.neocities.org
20. https://towardsdatascience.com/decision-tree-classifier-explained-a-visual-guide-with-code-examples-for-beginners-7c863f06a71e/
21. https://scikit-learn.org/stable/modules/tree.html
22. https://mmuratarat.github.io/2019-08-21/decision-trees
23. https://stackoverflow.com/questions/74544624/how-does-sklearn-tree-decisiontreeclassifier-function-predict-proba-work-inter
24. https://www.geeksforgeeks.org/machine-learning/python-decision-tree-regression-using-sklearn/
25. https://www.reddit.com/r/learnmachinelearning/comments/g9stqj/how_do_i_interpret_the_output_of_sklearns_predict/
26. https://www.geeksforgeeks.org/machine-learning/decision-tree-introduction-example/
27. https://www.geeksforgeeks.org/machine-learning/building-and-implementing-decision-tree-classifiers-with-scikit-learn-a-comprehensive-guide/
28. https://www.geeksforgeeks.org/machine-learning/understanding-the-predictproba-function-in-scikit-learns-svc/
29. https://www.xoriant.com/blog/decision-trees-for-classification-a-machine-learning-algorithm
30. https://www.simplilearn.com/tutorials/scikit-learn-tutorial/sklearn-decision-trees
31. https://www.reddit.com/r/learnmachinelearning/comments/mlymjf/how_to_interpret_predict_proba_in_classification/
32. https://www.geeksforgeeks.org/machine-learning/decision-tree/
33. https://www.activestate.com/resources/quick-reads/how-to-make-predictions-with-scikit-learn/

34. https://github.com/scikit-learn/scikit-learn/issues/1460
35. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
36. https://stackoverflow.com/questions/30814231/using-the-predict-proba-function-of-randomforestclassifier-in-the-safe-and-rig
37. https://qiita.com/miya8/items/f3f5756e8f2fa3d9d53f
38. https://elvanco.com/blog/how-to-implement-a-decision-tree-classifier-in
39. https://github.com/scikit-learn/scikit-learn/issues/16479
40. https://en.wikipedia.org/wiki/Decision_tree
41. https://nerchukoacademy.graphy.com/blog/making-predictions-with-decision-trees-regression-trees
42. https://machinelearning-basics.com/decision-trees-splitting-criteria-for-classification-and-regression/
43. https://christophm.github.io/interpretable-ml-book/tree.html
44. https://www.youtube.com/watch?v=g9c66TUylZ4
45. https://towardsdatascience.com/the-only-guide-you-need-to-understand-regression-trees-4964992a07a8/
46. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html
47. https://www.geeksforgeeks.org/machine-learning/python-decision-tree-regression-using-sklearn/
48. https://data36.com/regression-tree-python-scikit-learn/
49. https://infoaryan.com/blog/implementing-decision-tree-regression-from-scratch-using-python/
50. https://en.wikipedia.org/wiki/Mean_squared_error
51. https://uc-r.github.io/regression_trees
52. https://www.datacamp.com/tutorial/decision-tree-classification-python
53. https://github.com/jyotipmahes/Implementation-of-ML-algos-in-Python/blob/master/Decision Tree Regression .ipynb
54. https://en.wikipedia.org/wiki/Bias–variance_tradeoff
55. https://www.youtube.com/watch?v=sgQAhG5Q7iY
56. http://www2.stat.duke.edu/~rcs46/lectures_2017/08-trees/08-tree-advanced.pdf
57. https://www2.stat.duke.edu/~rcs46/lectures_2017/08-trees/08-tree-regression.pdf
58. https://scikit-learn.org/stable/modules/tree.html
59. https://www.youtube.com/watch?v=P2ZB8c5Ha1Q
60. https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/