CART Regression Model

Group Name : Print ("Hello World")

Group Members : Arpit Dang, Devraj Raghuvanshi, Matthew Dall'Asen, Varun Satheesh

GitHub Link: https://github.com/adang66/Print-Hello-World-

Date: December 15th, 2024

1. Overview

The objective behind the Classification and Regression Tree (CART) algorithm is to iteratively select features within the feature space, and splitting it into a binary decsion, based on a determined split point, to predict the target variable. This process is repeated for each resulting subset until a stopping criterion is met, such as the depth of the tree, number of epochs and/or inability to further improve homogeneity. This report will leverage the regression tree of CART, where the objective is to predict the value of a new data point by taking the average of the target variable values in the leaf node corresponding to the data point.

Decision trees are widely used for machine learning problems due to their ability to handle non-linearity, making them well-suited for modeling complex relationships between features and the target variable. Additionally, decision trees inherently perform feature selection, focusing splits on relevant features and reducing the risk of overfitting to irrelevant variables. Finally, their interpretability provides a clear and easy-to-understand decision paths that explain their predictions.

However, decisions trees are prone to overfitting, especially when the tree grows too deep, capturing noise in the data rather than meaningful patterns. Another limitation is their instability, as small changes in the training data can result in significantly different tree structures due to their greedy splitting approach. Moreover, decision trees can exhibit bias in their splitting criteria, often favoring features with more levels or categories, which might skew the results.

2. Representation

In CART (Classification and Regression Trees) for regression tasks, we define the input space as $\mathscr{X} = \mathbb{R}^d$, where each data point $\mathbf{x_i} = [x_{i1}, x_{i2}, \ldots, x_{id}]$ consists of d features. Each feature corresponds to a dimension in this d-dimensional space. The output space is defined as $\mathscr{Y} = \mathbb{R}$, indicating that the model predicts continuous values, which is suitable for regression.

The regression tree itself is a binary decision tree. At each internal node (N), the tree partitions the input space based on a single feature, j, selected from among the d features. This feature may vary at each node, depending on which split minimizes the prediction error best. To create a split, the algorithm selects a threshold value t for the chosen feature j, effectively dividing the data into two groups based on whether the j-th feature value of each point is less than or equal to t or greater than t. For any data point $\mathbf{x}_i$, the tree assigns a direction based on the following condition:

$$\text{Direction}(\mathbf{x}_i, N) = \begin{cases} \text{left child node}, & x_{ij} \leq t \\ \text{right child node}, & x_{ij} > t \end{cases}$$

This recursive partitioning of the input space continues until the data reaches a leaf node. Each leaf node, denoted L_k, represents a specific region in the feature space and corresponds to a unique subset of the training data that falls within that region. For any data point that reaches L_k, the prediction $\hat{y}_k$ is typically calculated as the mean of the target values y for all training observations that lie within that leaf. Specifically, if n_k represents the number of observations in L_k, then:

$$\hat{y}_k = \frac{1}{n_k} \sum_{m \in L_k} y_m$$

Thus, each region defined by a leaf node provides a constant prediction that reflects the average target value of the training points within that region.

3. Loss Function

In contrast to the classification version of CART, which typically uses Gini impurity or entropy to measure the quality of splits, CART regression uses Mean of Squared Errors (MSE) as its loss function. MSE measures the total squared difference between the actual target values and the predicted values within each node and averages them out. Minimizing MSE allows the model to identify splits that reduce the overall variance in target values within each region, leading to more accurate predictions.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where:

- n is the total number of observations
- y_i is the target value of the i^{th} sample

- $\hat{y}_i$ is the predicted value for the i^{th} sample given the learned model weights

In CART regression, MSE is used at each node to evaluate and select the optimal feature and threshold for splitting. By choosing splits that minimize MSE, the model creates more homogenous regions with respect to the target variable, enhancing the accuracy of predictions in each region.

4. Decision Tree Algorithm

At each internal node, CART regression evaluates potential splits across different features and threshold values to find the optimal partition that minimizes MSE. The process for determining the best split is as follows:

3.1 Splitting the Data into Groups

For a given feature j and threshold t, the data is divided into two groups:

- **Group 1**: $D_1 = \{(\mathbf{x}_i, y_i) \mid x_{ij} \leq t\}$, containing all data points where the j-th feature value is less than or equal to t.
- **Group 2**: $D_2 = \{(\mathbf{x}_i, y_i) \mid x_{ij} > t\}$, containing all data points where the j-th feature value is greater than t.

This division creates two subsets of data, each representing a distinct region in the feature space based on the chosen split.

3.2 Calculating MSE for Each Group

The **Mean of Squared Errors (MSE)** for each subset D_1 and D_2 is calculated to assess the quality of the split. This calculation reflects the variance of the target values within each group, which should ideally be minimized. The MSE for each group is given by:

$$\text{MSE}(D_1) = \frac{1}{n} \sum_{(\mathbf{x}_i, y_i) \in D_1} (y_i - \hat{y}_{D_1})^2$$

$$\text{MSE}(D_2) = \frac{1}{n} \sum_{(\mathbf{x}_i, y_i) \in D_2} (y_i - \hat{y}_{D_2})^2$$

where:

- $|D_1|$ and $|D_2|$ denote the number of observations in D_1 and D_2, respectively,
- $\hat{y}_{D_1}$ and $\hat{y}_{D_2}$ represent the mean target values of observations in D_1 and D_2, respectively.

By calculating the MSE for each group, the algorithm quantifies how closely the target values are clustered around their mean, indicating the “purity” of each split.

3.3 Calculating the Gain for the Split

The total gain for a split based on feature j and threshold t is derived from the decrease in impurity between the parent node and the resulting groups:

$$\text{Gain} = \text{MSE}_{\text{parent}} - (\text{MSE}_{\text{left child}} + \text{MSE}_{\text{right child}})$$

This overall gain reflects the quality of the split: a higher gain indicates that the split has effectively reduced the variance in the target values within each resulting group, making it more likely to be selected as an optimal partition.

3.4 Selecting the Best Split

The algorithm evaluates all possible splits across features and thresholds and selects the split that maximizes the gain. Mathematically, the best split is determined as:

$$\text{Best Split} = \arg\max_{t,j} \text{Gain}(t, j)$$

This recursive process of selecting optimal splits continues, partitioning each node to maximize gain, until a stopping criterion is met. This approach ensures that each node split contributes to a more accurate representation of the target values within the tree’s regions.

3.5. Stopping Criteria in CART Regression

To avoid overfitting and control the complexity of the tree, CART regression employs specific stopping criteria. These criteria determine when the recursive partitioning should halt, ensuring the tree is both interpretable and generalizes well on unseen data. Common stopping criteria include:

- Dataset is Empty: If the dataset is empty, the mean target value computed over the entire dataset is predicted.
- Maximum Tree Depth: A predefined maximum depth limits the growth of the tree, preventing it from becoming overly complex. Once the maximum depth is reached, no further splits are performed, and nodes at this depth become leaf nodes.
- Minimum Samples per Leaf: The tree stops splitting a node if the number of samples within that node is below a specified threshold. This criterion ensures that each leaf node contains a sufficient number of observations to produce stable predictions, reducing the likelihood of high variance and overfitting.

- Homogeneous Target Values: If all the target values in a node are identical, no further splits are made. This condition ensures that the tree does not split unnecessarily when the target values are already perfectly predicted, making the node a leaf.

5. Pseudocode for Decision Tree

4.1. Input

- **D**: Dataset containing N samples, where each sample $\mathbf{x_i} = [y_i, \mathbf{f_i}]$ includes the target value y_i and features $\mathbf{f_i}$.
- **max_depth**: Maximum allowed depth of the tree.
- **min_samples_split**: Minimum number of samples required to split a node.

4.2. Algorithm

- Initialization
  - Compute the **mean target value** across the dataset D.
  - Initialize the **root node** R with this mean value.
- Recursive Splitting
  - Call the function `SplitRecursively(Node, Data)` starting from the root.
  - Function: `SplitRecursively(Node, Data)`
  - Terminal Condition
    - If one of the following holds:
      - $\text{depth}(Node) \geq \text{max\\_depth}$
      - $\text{number of samples} < \text{min\\_samples\\_split}$
      - All target values y_i in the current node are identical.
    - Set the node as a **leaf** and assign it the **mean target value** of the current data.
  - Feature Selection
    - For each feature index j:
      - Sort the data by feature j.
      - For each valid split point i, calculate the **split threshold**: $\text{threshold} = \frac{\mathbf{f_i}[j] + \mathbf{f_{i-1}}[j]}{2}$
      - Compute the **gain** (decrease in impurity) from this split using: $\text{Gain} = \text{MSE}_{\text{before}} - \text{MSE}_{\text{after}}$
    - Select the feature $j*$ and threshold with the **maximum gain**.
  - Data Partitioning

- Partition the data into:
  $D_{\text{left}} = \{\mathbf{x_i} \mid \mathbf{f_i}[j*] \leq \text{threshold}\}, \quad D_{\text{right}} = \{\mathbf{x_i} \mid \mathbf{f_i}[j*] > \text{threshold}\}$
- Create left and right child nodes.

- Recursion

  - Call `SplitRecursively` on the left and right child nodes with their respective datasets.

4.3. Output

- Trained Decision Tree model for regression.

6. Model

In [9]:
```python
import numpy as np
import copy
import math

class Node:
    def __init__(self, left=None, right=None, depth=0, index_split_on=0,
                 isleaf=False, pred=0.0, threshold=0):
        """
        Initialize a Node for the CART tree.

        Parameters:
            left (Node): The left child node.
            right (Node): The right child node.
            depth (int): The depth of the node in the tree.
            index_split_on (int): The feature index the node splits on.
            isleaf (bool): Whether the node is a leaf.
            pred (float): The predicted value for regression.
            threshold (float): The threshold value for splitting.
        """

        self.left = left
        self.right = right
        self.depth = depth
        self.index_split_on = index_split_on
        self.threshold = threshold
        self.isleaf = isleaf
        self.pred = pred  # Stores mean value for regression
        self.info = {}


    def _set_info(self, gain, num_samples):
        """
        Helper function to set additional information about the node.

        Parameters:
            gain (float): The gain achieved at this node.
            num_samples (int): The number of samples at this node.
        """

        self.info['gain'] = gain
        self.info['num_samples'] = num_samples


class CART:

    def __init__(self, data, min_samples_split=20, max_depth=5):
        """
        Initialize the CART regression tree.

        Parameters:
            data (list of lists): The dataset where the first column
            is the target value.
            min_samples_split (int): Minimum number of samples required
            to split a node.
            max_depth (int): Maximum depth of the tree.
        """

        y = [row[0] for row in data]
        self.y_mean_dataset = np.mean(y)

        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = Node(pred = self.y_mean_dataset)

        indices = list(range(1, len(data[0])))

        self._split_recurs(self.root, data, indices)


    def predict(self, features):
        """
        Predict the target value for a given set of features.

        Parameters:
            features (list): The feature values for prediction.

        Returns:
            float: Predicted target value.
        """

        return self._predict_recurs(self.root, features)


    def calculate_mae(self, data):
        """
        Calculate Mean Absolute Error (MAE) on the given dataset.

        Parameters:
            data (list of lists): The dataset where the first column is
            the target value.

        Returns:
            tuple: MAE and a list of predictions.
        """

        absolute_error = 0.0
        predictions = []
        for row in data:
            prediction = self.predict(row)
            absolute_error += abs(prediction - row[0])
            # row[0] is the true target value
            predictions.append(prediction)
        mae = absolute_error / len(data)
        return mae, predictions


    def loss(self, data):
        """
        Calculate Mean Squared Error (MSE) on the given dataset.

        Parameters:
            data (list of lists): The dataset where the first column
            is the target value.

        Returns:
            tuple: MSE and a list of predictions.
        """

        squared_error = 0.0
        predictions = []
        for row in data:
            prediction = self.predict(row)
            squared_error += (prediction - row[0]) ** 2  # row[0] is the
true target value
            predictions.append(prediction)
        return squared_error / len(data), predictions


    def _predict_recurs(self, node, row):
        """
        Recursive helper function to predict the target value
        for a row of data.

        Parameters:
            node (Node): The current node being traversed.
            row (list): The feature values of the row.

        Returns:
            float: Predicted target value.
        """

        if node.isleaf or node.index_split_on == 0:
            return node.pred
        if row[node.index_split_on] <= node.threshold:
            return self._predict_recurs(node.left, row)
        else:
            return self._predict_recurs(node.right, row)
```

```python
    def _is_terminal(self, node, data, indices):
        """
        Check if a node should be a terminal node.

        Parameters:
            node (Node): The current node being evaluated.
            data (list of lists): The data at this node.
            indices (list): List of feature indices.

        Returns:
            tuple: (bool indicating if terminal,
            predicted value if terminal).
        """

        y = [row[0] for row in data]

        # If the dataset is empty
        if len(data) == 0:
            return True, self.y_mean_dataset  # Default value

        # If all target values are the same
        if len(set(y)) == 1:
            return True, y[0]  # All values are identical

        # If maximum depth is reached
        if node.depth >= self.max_depth:
            return True, sum(y) / len(y)  # Mean value

        # If the number of samples is less than min_samples_split
        if len(data) < self.min_samples_split:
            return True, sum(y) / len(y)  # Mean value

        return False, sum(y) / len(y)  # Continue splitting


    def _split_recurs(self, node, data, indices):
        """
        Recursive function to split the data and create child nodes.

        Parameters:
            node (Node): The current node being processed.
            data (list of lists): The data at this node.
            indices (list): List of feature indices to consider for
            splitting.
        """

        # Check if the current node is terminal
        node.isleaf, node.pred = self._is_terminal(node, data, indices)

        if not node.isleaf:
            max_gain = -float('inf')
            best_index = -1

            # Iterate through each feature
            for index in indices:
                # Sort data based on the current feature
                sorted_data = sorted(data, key=lambda x: x[index])
                for i in range(1, len(sorted_data)):
                    if sorted_data[i][index] != sorted_data[i-1][index]:
                        threshold = (sorted_data[i][index] +
                                     sorted_data[i-1][index]) / 2
                        gain = self._calc_gain(data, index, threshold)
                        if gain > max_gain:
                            max_gain = gain
                            best_index = index
                            best_threshold = threshold

            if best_index == -1:
                node.isleaf = True
                node.pred = sum(row[0] for row in data) / len(data)
                return

            # Store the best index and threshold
            node.index_split_on = best_index
            node.threshold = best_threshold
            node._set_info(max_gain, len(data))

            # Split the data
            left_data = [row for row in data
                         if row[best_index] <= best_threshold]
            right_data = [row for row in data
                          if row[best_index] > best_threshold]

            # Create child nodes
            node.left = Node(depth=node.depth + 1)
            node.right = Node(depth=node.depth + 1)

            # Recur on the left and right nodes
            self._split_recurs(node.left, left_data, indices)
            self._split_recurs(node.right, right_data, indices)


    def _calc_gain(self, data, split_index, threshold):
        """
        Calculate the variance reduction (gain) for a given split.

        Parameters:
            data (list of lists): The data to evaluate the split.
            split_index (int): The index of the feature to split on.
            threshold (float): The threshold value for splitting.

        Returns:
            float: The variance reduction achieved by the split.
        """

        y = [row[0] for row in data]
        xi = [row[split_index] for row in data]

        if len(y) == 0:
            return 0
        # Variance before the split
        variance_before = self._variance(y)

        # Split data by feature value
        left_y = [y[i] for i in range(len(y)) if xi[i] <= threshold]
        right_y = [y[i] for i in range(len(y)) if xi[i] > threshold]

        # Variance after the split
        left_variance = self._variance(left_y) if len(left_y) > 0 else 0
        right_variance = self._variance(right_y) if len(right_y) > 0 else 0

        variance_after = (len(left_y) / len(y)) * left_variance +
(len(right_y) / len(y)) * right_variance   # WEIGHTED VARIANCE REDUCTION
        # variance_after = left_variance + right_variance

        gain = variance_before - variance_after
        return gain


    def _variance(self, values):
        """
        Compute the variance of a list of values.

        Parameters:
            values (list): Numerical values for which variance
            is to be calculated.

        Returns:
            float: Variance of the input values.
        """
        mean_value = sum(values) / len(values)
        return sum((x - mean_value) ** 2 for x in values) / len(values)
```

7. Unit Tests

Test 1

In [12]:
```
import pytest
# Sets random seed for testing purposes
np.random.seed(0)

# Create test data
data1 = [
    [2.5, 0, 1, 0],
    [1.5, 1, 0, 1],
    [3.0, 0, 1, 1],
    [2.0, 1, 0, 0],
    [3.5, 0, 1, 1],
]

data2 = [
    [1.0, 0, 0, 0],
    [2.0, 1, 1, 0],
    [3.0, 1, 0, 1],
    [4.0, 0, 1, 1],
    [5.0, 1, 1, 1],
    [6.0, 0, 0, 0],
]

# Test Models
test_model1 = CART(data1, min_samples_split=2, max_depth=3)
test_model2 = CART(data2, min_samples_split=2, max_depth=3)

# Test model initialization
assert isinstance(test_model1, CART)
assert isinstance(test_model2, CART)
assert test_model1.min_samples_split == 2
assert test_model1.max_depth == 3
assert test_model2.min_samples_split == 2
assert test_model2.max_depth == 3

# Test root node properties
assert isinstance(test_model1.root, Node)
assert isinstance(test_model2.root, Node)
assert test_model1.root.depth == 0
assert test_model2.root.depth == 0

# Test prediction functionality
pred1 = test_model1.predict([0, 1, 0, 1])
pred2 = test_model2.predict([0, 1, 1, 1])

assert isinstance(pred1, (int, float))
assert isinstance(pred2, (int, float))

# Test Mean Absolute Error (MAE) calculation
mae1, predictions1 = test_model1.calculate_mae(data1)
mae2, predictions2 = test_model2.calculate_mae(data2)

assert isinstance(mae1, float)
assert isinstance(mae2, float)
assert len(predictions1) == len(data1)
assert len(predictions2) == len(data2)

# Test Mean Squared Error (MSE) calculation
mse1, predictions1 = test_model1.loss(data1)
mse2, predictions2 = test_model2.loss(data2)


assert isinstance(mse1, float)
assert isinstance(mse2, float)
assert len(predictions1) == len(data1)
assert len(predictions2) == len(data2)

# Function to recursively check node structure
def check_node(node):
    if node.isleaf:
        assert node.left is None and node.right is None
    else:
        assert isinstance(node.left, Node)
        assert isinstance(node.right, Node)
        assert node.left.depth == node.depth + 1
        assert node.right.depth == node.depth + 1
        check_node(node.left)
        check_node(node.right)

# Test tree structure
check_node(test_model1.root)
check_node(test_model2.root)

# Function to recursively check split information
def check_split_info(node):
    if not node.isleaf:
        assert 'gain' in node.info
        assert 'num_samples' in node.info
        assert isinstance(node.info['gain'], float)
        assert isinstance(node.info['num_samples'], int)
        check_split_info(node.left)
        check_split_info(node.right)

# Test split information
check_split_info(test_model1.root)
check_split_info(test_model2.root)

# Test variance calculation
values = [1, 2, 3, 4, 5]
variance = test_model1._variance(values)
assert isinstance(variance, float)
assert np.isclose(variance, 2.0)

# Test gain calculation
data = [[1, 0], [2, 1], [3, 0], [4, 1], [5, 0]]
gain = test_model1._calc_gain(data, 1, 0.5)
assert isinstance(gain, float)
assert gain >= 0

print("All tests pass successfully!")
```

```
All tests pass successfully!
```

Test 2

In [ ]:
```
# Define the training data and validation data
data = [
    [6, 10.5, 3],  # Target is in the first column
    [4, 8.3, 2],
    [7, 12.4, 5],
    [5, 9.1, 3],
    [3, 7.2, 1]
]

validation_data = [
    [3, 2.5, 6],  # Expected: approximately 3
    [3, 4.0, 3],  # Expected: approximately 3
    [3, 5.5, 8],  # Expected: approximately 3
    [3, 6.0, 4]   # Expected: approximately 3
]
```

```python
# Helper function to check approximate equality
def approx(actual, expected, tolerance=0.1):
    """Check if the actual value is within the expected
    value +/- tolerance."""
    return abs(actual - expected) <= tolerance

# Initialize your CART model
cart_model = CART(data, min_samples_split=2, max_depth=3)

# Calculate loss and predictions
loss, predictions = cart_model.loss(validation_data)

# Display results
print("Predictions on validation data:", predictions)
print("MSE Loss on validation data:", loss)

# Expected predictions for validation data
expected_predictions = [3.0, 3.0, 3.0, 3.0]

# Validate predictions
assert len(predictions) == len(validation_data), "Prediction length 
mismatch."
# "Prediction length mismatch."

# Validate predictions against expected values
for i, (pred, expected) in enumerate(zip(predictions, 
expected_predictions)):
    assert approx(pred, expected), f"Prediction {pred} is not approx 
{expected} for validation data index {i}."

# Manually calculate expected MSE
true_targets = [row[0] for row in validation_data]
expected_mse = np.mean([(pred - true) ** 2 for pred, 
                        true in zip(expected_predictions, true_targets)])

# Validate MSE
assert approx(loss, expected_mse), f"MSE {loss} is not approx 
{expected_mse}."

print("All assertions passed successfully!")
```

```
Predictions on validation data: [3, 3, 3, 3]
MSE Loss on validation data: 0.0
All assertions passed successfully!
```

Test 3

In [ ]:
```python
# Define the test data and validation data
data = [
    [4, 7.2, 1],  # Target is in the first column
    [3, 6.8, 2],
    [2, 5.5, 3],
    [1, 3.3, 4],
    [5, 8.1, 5]
]
```

```
validation_data = [
    [3, 6.0, 3],  # Expected: approximately 2
    [1, 4.5, 1],  # Expected: approximately 2
    [5, 8.0, 5],  # Expected: approximately 5
    [2, 5.5, 2]   # Expected: approximately 2
]

# Helper function to check approximate equality
def approx(actual, expected, tolerance=0.1):
    """Check if the actual value is within the
    expected value +/- tolerance."""
    return abs(actual - expected) <= tolerance

# Initialize your CART model
cart_model = CART(data, min_samples_split=2, max_depth=3)

# Calculate loss and predictions
loss, predictions = cart_model.loss(validation_data)

# Display results
print("Predictions on validation data:", predictions)
print("MSE Loss on validation data:", loss)

# Expected predictions for validation data
expected_predictions = [2.0, 2.0, 5.0, 2.0]

# Validate predictions
assert len(predictions) == len(validation_data), "Prediction length
mismatch."
assert approx(predictions[0], 2.0), f"Prediction {predictions[0]} is not
approx 2.0."
assert approx(predictions[1], 2.0), f"Prediction {predictions[1]} is not
approx 2.0."
assert approx(predictions[2], 5.0), f"Prediction {predictions[2]} is not
approx 5.0."
assert approx(predictions[3], 2.0), f"Prediction {predictions[3]} is not
approx 2.0."


# Manually calculate expected MSE
expected_mse = np.mean([(pred - true[0]) ** 2 for pred,
                        true in zip(expected_predictions,
validation_data)])

# Validate MSE
assert approx(loss, expected_mse), f"MSE {loss} is not approx
{expected_mse}."

print("All assertions passed successfully!")
```

```
Predictions on validation data: [2, 2, 5, 2]
MSE Loss on validation data: 0.5
All assertions passed successfully!
```

Test 4 (Edge)

In [10]:
```
import numpy as np
import warnings
warnings.filterwarnings('ignore', category=RuntimeWarning)


# Empty dataset test
empty_data = []
try:
    cart = CART(empty_data)
except IndexError as e:
    print("Empty dataset test passed:", str(e))

# Single data point test
single_point_data = [[5.0, 1.2, 3.4]]
cart = CART(single_point_data, min_samples_split=2, max_depth=3)
prediction = cart.predict([1.2, 3.4])
assert np.isclose(prediction, 5.0), "Single data point test failed"

# Identical features test
identical_features = [[3.0, 2.0, 2.0], [3.0, 2.0, 2.0], [3.0, 2.0, 2.0]]
cart = CART(identical_features, min_samples_split=2, max_depth=3)
prediction = cart.predict([2.0, 2.0])
assert np.isclose(prediction, 3.0), "Identical features test failed"

# No variance in target test
no_variance_target = [[1.0, 3.4, 5.6], [1.0, 2.1, 4.3], [1.0, 3.0, 3.3]]
cart = CART(no_variance_target, min_samples_split=2, max_depth=3)
prediction = cart.predict([3.4, 5.6])
assert np.isclose(prediction, 1.0), "No variance in target test failed"

# Min samples split not met test
data = [[5.0, 1.0], [10.0, 2.0]]
cart = CART(data, min_samples_split=5, max_depth=3)
assert cart.root.isleaf, "Min samples split not met test failed"

# High variance split test
data = [[1.0, 2.0], [100.0, 3.0], [1000.0, 4.0]]
cart = CART(data, min_samples_split=2, max_depth=3)
assert cart.root.threshold is not None, "High variance split test failed"

# Predict with unseen features test
data = [[5.0, 1.0], [10.0, 2.0], [15.0, 3.0]]
cart = CART(data, min_samples_split=1, max_depth=3)
prediction = cart.predict([100.0, 4.0])
assert prediction is not None, "Predict with unseen features test failed"

print("All tests passed!")
```

```
Empty dataset test passed: list index out of range
All tests passed!
```

8. Main

California Housing Dataset

In [ ]:
```
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

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

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Format the data for the custom CART implementation (y comes first)
train_data = np.hstack((y_train.reshape(-1, 1), X_train))
test_data = np.hstack((y_test.reshape(-1, 1), X_test))

# Train sklearn's DecisionTreeRegressor
sklearn_model = DecisionTreeRegressor(max_depth=5, min_samples_split=20)
sklearn_model.fit(X_train, y_train)
sklearn_predictions = sklearn_model.predict(X_test)
sklearn_mse = mean_squared_error(y_test, sklearn_predictions)

# Train your CART implementation
custom_model = CART(data=train_data, max_depth=5, min_samples_split=20)
custom_mse, custom_predictions = custom_model.loss(test_data)

# Compare results
print(f"Sklearn MSE: {sklearn_mse:.4f}")
print(f"Custom CART MSE: {custom_mse:.4f}")
```

```
Sklearn MSE: 0.5245
Custom CART MSE: 0.5245
```

Diabetes Dataset

In [ ]:
```
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

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

# Split the dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Format the data for the custom CART implementation (y comes first)
train_data = np.hstack((y_train.reshape(-1, 1), X_train))
test_data = np.hstack((y_test.reshape(-1, 1), X_test))

# Train sklearn's DecisionTreeRegressor
sklearn_model = DecisionTreeRegressor(max_depth=5, min_samples_split=20,
random_state=42)
sklearn_model.fit(X_train, y_train)
sklearn_predictions = sklearn_model.predict(X_test)
sklearn_mse = mean_squared_error(y_test, sklearn_predictions)

# Train your CART implementation
custom_model = CART(data=train_data, max_depth=5, min_samples_split=20)
custom_mse, custom_predictions = custom_model.loss(test_data)

# Compare results
print(f"Sklearn MSE: {sklearn_mse:.4f}")
print(f"Custom CART MSE: {custom_mse:.4f}")
```

```
Sklearn MSE: 3358.6384
Custom CART MSE: 3378.0334
```

Previous Work: A Data Mining Approach to Predict Forest Fires using Meteorological Data

In their paper "A Data Mining Approach to Predict Forest Fires using Meteorological Data," Cortez et al. (2007) explored the use of machine learning algorithms to predict the area burned in forest fires using meteorological data. They tested several ML techniques, including CART. The study demonstrated that meteorological features such as temperature, wind, relative humidity, and rainfall significantly contribute to predictive performance. The dataset used was the Forest Fires dataset (available on UCI ML Repository), consisting of 517 samples and 12 features.

Evaluation Metric: Mean Absolute Error (MAE)

We replicate the methodology outlined in the paper using the following steps:

- Preprocess the data (e.g., one-hot encoding and standardization).
- Use Decision Tree with CART implementation.
- Perform 30 rounds of 10-fold cross-validation.
- Evaluate performance using MAE.

```
In [ ]: !pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.7-py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/di
st-packages (from ucimlrepo) (2.2.2)
Requirement already satisfied: certifi>=2020.12.5 in /usr/local/lib/python3.
10/dist-packages (from ucimlrepo) (2024.8.30)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/di
st-packages (from pandas>=1.0.0->ucimlrepo) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/pyth
on3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dis
t-packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/d
ist-packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-pa
ckages (from python-dateutil>=2.8.2->pandas>=1.0.0->ucimlrepo) (1.16.0)
Downloading ucimlrepo-0.0.7-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.7
```

In [ ]:
```python
from ucimlrepo import fetch_ucirepo

# fetch dataset
forest_fires = fetch_ucirepo(id=162)

# data (as pandas dataframes)
X = forest_fires.data.features
y = forest_fires.data.targets
```

In [ ]:
```python
X
```

Out[ ]:

	X	Y	month	day	FFMC	DMC	DC	ISI	temp	RH	wind	rain
0	7	5	mar	fri	86.2	26.2	94.3	5.1	8.2	51	6.7	0.0
1	7	4	oct	tue	90.6	35.4	669.1	6.7	18.0	33	0.9	0.0
2	7	4	oct	sat	90.6	43.7	686.9	6.7	14.6	33	1.3	0.0
3	8	6	mar	fri	91.7	33.3	77.5	9.0	8.3	97	4.0	0.2
4	8	6	mar	sun	89.3	51.3	102.2	9.6	11.4	99	1.8	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...
512	4	3	aug	sun	81.6	56.7	665.6	1.9	27.8	32	2.7	0.0
513	2	4	aug	sun	81.6	56.7	665.6	1.9	21.9	71	5.8	0.0
514	7	4	aug	sun	81.6	56.7	665.6	1.9	21.2	70	6.7	0.0
515	1	4	aug	sat	94.4	146.0	614.7	11.3	25.6	42	4.0	0.0
516	6	3	nov	tue	79.5	3.0	106.7	1.1	11.8	31	4.5	0.0

517 rows × 12 columns

In [ ]:
```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.preprocessing import StandardScaler
```

Testing Agasint Sklearn's Implementation

Sklearn

In [ ]:
```
X_encoded = pd.get_dummies(X, columns=["month", "day"])
y_transformed = np.log1p(y)  # ln(x+1) transformation

mae_avg = []

for i in range(30):

  mae_values = []

  kf = KFold(n_splits=10, shuffle=True, random_state=i*42)

  for i, (train_index, test_index) in enumerate(kf.split(X_encoded)):
      X_train, X_test = X_encoded.iloc[train_index],
X_encoded.iloc[test_index]
      y_train, y_test = y_transformed.iloc[train_index],
y_transformed.iloc[test_index]

      scaler = StandardScaler()
      X_train = scaler.fit_transform(X_train)
      X_test = scaler.transform(X_test)

      sklearn_model = DecisionTreeRegressor(max_depth=5,
min_samples_split=42)
      sklearn_model.fit(X_train, y_train)

      sklearn_predictions = sklearn_model.predict(X_test)

      y_pred = np.expm1(sklearn_predictions)
      y_true = np.expm1(y_test)

      sklearn_mae = mean_absolute_error(y_true, y_pred)
      mae_values.append(sklearn_mae)

  average_mae = np.mean(mae_values)
  mae_avg.append(average_mae)

print(f"MAE: {np.mean(mae_avg)} +/- {np.std(mae_avg)}")
```

```
MAE: 13.609762745243872 +/- 0.37620042307863827
```

Custom Cart Model

```
In [ ]: from sklearn.model_selection import KFold
        from sklearn.tree import DecisionTreeRegressor
        from sklearn.metrics import mean_absolute_error, mean_squared_error

        X_encoded = pd.get_dummies(X, columns=["month", "day"])
        y_transformed = np.log1p(y)  # ln(x+1) transformation

        mae_avg = []

        for i in range(30):

          mae_values = []

          kf = KFold(n_splits=10, shuffle=True, random_state=i*42)

          for i, (train_index, test_index) in enumerate(kf.split(X_encoded)):
              X_train, X_test = X_encoded.iloc[train_index],
        X_encoded.iloc[test_index]
              y_train, y_test = y_transformed.iloc[train_index],
        y_transformed.iloc[test_index]

              X_train = np.array(X_train)
              X_test = np.array(X_test)
              y_train = np.array(y_train)
              y_test = np.array(y_test)

              scaler = StandardScaler()
              X_train = scaler.fit_transform(X_train)
              X_test = scaler.transform(X_test)

              # Format the data for the custom CART implementation (y comes first)
              train_data = np.hstack((y_train.reshape(-1, 1), X_train))
              test_data = np.hstack((y_test.reshape(-1, 1), X_test))

              # Train the CART implementation
              custom_model = CART(data=train_data, max_depth=5,
        min_samples_split=42)
              custom_mae, custom_predictions =
        custom_model.calculate_mae(test_data)

              y_pred = np.expm1(custom_predictions)
              y_true = np.expm1(y_test)

              custom_mae = mean_absolute_error(y_true, y_pred)
              mae_values.append(custom_mae)

          average_mae = np.mean(mae_values)
          mae_avg.append(average_mae)

        print(f"MAE: {np.mean(mae_avg)} +/- {np.std(mae_avg)}")
```

```
MAE: 13.609079338058589 +/- 0.3774537599572436
```

9. GitHub Link

https://github.com/adang66/Print-Hello-World-

10. References

- Cortez, P. and Morais, A.D.J.R., 2007. A data mining approach to predict forest fires using meteorological data.
- Breiman, L., 2017. Classification and regression trees. Routledge.
- Scikit-learn: Tree Algorithms (CART)