Lecture 16

Decision Trees, splitting methods

Decision trees are a type of supervised learning algorithm commonly used for both classification and regression tasks. The basic idea is to recursively partition the data into subsets that maximize the homogeneity of the target variable within each subset.

How the Decision Tree Algorithm Works:
Splitting Criteria: The algorithm begins by selecting the best feature (and corresponding threshold for numeric features) to split the data based on a specific criterion. Common criteria include:

- **Gini Impurity** (used in classification): Measures the impurity or impurity of a dataset; the goal is to minimize this impurity.
- **Entropy/Information Gain** (used in classification): Measures the amount of information gained by splitting the data on a particular feature.
- **Variance Reduction** (used in regression): Measures the reduction in variance (spread of data) after a split.

Splitting Process: The dataset is split into subsets using the selected feature and threshold. The process repeats recursively on each subset, creating branches of the tree. This continues until a stopping criterion is met (e.g., maximum depth, minimum number of samples in a node).

Leaf Nodes: When the recursive splitting process stops, the data in each subset becomes a **leaf node**. For classification, the leaf node predicts the class with the majority of samples. For regression, the leaf node predicts the average value of the target variable in that node.

Prediction: To make a prediction, start at the root of the tree and follow the branches according to the feature values of the input until you reach a leaf node. The value at that leaf node is the prediction.

General Use in Machine Learning:

- *Classification:* Decision trees are often used in scenarios where interpretability is important. They can classify data into predefined classes based on input features.
- *Regression:* Decision trees can also predict continuous values by partitioning the feature space.

Pros of Decision Trees:
Interpretability: Decision trees are easy to interpret and visualize. Each decision path from the root to a leaf can be understood as a series of if-then rules.

No Need for Feature Scaling: Decision trees do not require feature scaling (e.g., standardization or normalization).

Handles Both Numerical and Categorical Data: Decision trees can handle both types of data without requiring much preprocessing.

Non-Linear Relationships: They can model non-linear relationships between features and the target variable.

Versatile: Can be used for both classification and regression tasks.

Cons of Decision Trees:
Overfitting: Decision trees are prone to overfitting, especially when the tree is deep and complex. This happens because the model can capture noise in the training data rather than the underlying data patterns. Pruning techniques and setting constraints like maximum tree depth can help mitigate this issue.

Instability: Small changes in the data can lead to different splits and, therefore, different trees (high variance).

Bias Towards Dominant Classes: If some classes dominate the training data, the decision tree might be biased towards those classes.

Difficulty in Capturing Linear Relationships: Decision trees might not capture simple linear relationships as efficiently as other algorithms like linear regression.

Greedy Algorithm: The algorithm is greedy and does not guarantee an optimal solution. It selects the best feature to split at each node based on local criteria, which might not lead to a globally optimal tree.

Common Use Cases:
- Medical Diagnosis: Predicting the presence of a disease based on symptoms and other patient data.
- Customer Segmentation: Identifying groups of customers with similar characteristics.
- Fraud Detection: Classifying transactions as fraudulent or non-fraudulent based on various features.
- Credit Scoring: Assessing the creditworthiness of an individual.

Variants and Enhancements:
- Ensemble Methods: Decision trees are the base learners in ensemble methods like Random Forests and Gradient Boosting Machines (GBMs), which combine multiple trees to improve accuracy and reduce overfitting.
- Pruning: Techniques like cost-complexity pruning (used in CART) help reduce the size of the tree by removing sections of the tree that provide little power to classify instances.

Decision trees are a fundamental tool in machine learning, valued for their simplicity, interpretability, and flexibility. However, to leverage them effectively, it's important to be aware of their limitations, particularly with overfitting and instability, and to consider ensemble methods to overcome some of these issues.

Our goal here will be to consider customizing the splitting criteria and the stopping criteria. We'll look at ensembles in the next lecture.

We mentioned three common splitting criteria above, but let's look at those more closely along with some other options. Different splitting criteria are chosen based on the nature of the task (classification vs. regression), the characteristics of the data, and the specific goals of the analysis. The choice of criterion can significantly affect the performance and interpretability of the decision tree model.

1. Gini Impurity (Classification)

- Description: Gini impurity measures the likelihood of incorrectly classifying a randomly chosen element if it were randomly labeled according to the distribution of labels in the dataset.
- Formula: $Gini\ Impurity = 1 - \sum_{i=1}^{C} p_i^2$ where p_i is the probability of an element being classified for class i and C is the number of classes.
- Usage: This is the default criterion used in CART (Classification and Regression Trees) for classification tasks.

2. Entropy/Information Gain (Classification)

- Description: Entropy measures the disorder or impurity in the dataset. Information gain is the reduction in entropy after a dataset is split on an attribute.
- Formula: $Entropy = -\sum_{i=1}^{C} p_i \log_2(p_i)$
  Information gain is calculated as:
  $$Information\ Gain = Entropy(parent) - \sum_{children} \frac{N_{child}}{N_{parent}} \times Entropy(child)$$
- Usage: Commonly used in algorithms like ID3 and C4.5.

3. Variance Reduction (Regression)

- Description: Variance reduction is used as a splitting criterion for regression tasks. It measures the reduction in variance (spread of the target values) after a split.
- Formula: $Variance = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2$
- Usage: This is typically used in regression trees.

4. Mean Squared Error (MSE) Reduction (Regression)

- Description: Mean squared error (MSE) is another criterion used in regression trees to evaluate the quality of a split.
- Formula: $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_1 - \hat{y})^2$ The split is chosen to minimize the MSE.
- Usage: Used in many regression trees and algorithms that build regression trees like CART.

5. Chi-Square Statistic (Classification)

- Description: The Chi-Square statistic measures the divergence between the observed and expected frequencies of the target variable. A split is chosen based on the highest Chi-Square value.
- Formula: $\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$ where O_i is the observed frequency, and E_i is the expected frequency.
- Usage: Sometimes used in decision trees that prioritize the significance of splits (like CHAID).

6. Reduction in Impurity (Classification and Regression)

- Description: This is a general criterion that includes Gini impurity, entropy, and variance. The reduction in impurity after a split is measured, and the best split is selected.
- Usage: Commonly used in decision trees and ensemble methods.

7. Twoing Rule (Classification)

- Description: The Twoing rule is designed to maximize the difference between two groups formed by a split, focusing on creating two equally large groups with different class distributions.
- Formula: $Twoing\ Score = \left(\frac{N_L}{N} \times \frac{N_R}{N}\right) \times (\sum |p_{Li} - p_{Ri}|)^2$ where p_{Li} and p_{Ri} are the proportions of class i in the left and right nodes.

- Usage: Often used in multi-class classification problems.

8. Friedman's MSE (Regression)

- Description: This criterion is used in gradient boosting machines and is based on minimizing the MSE across multiple iterations.
- Usage: Used in boosting algorithms like Gradient Boosted Trees (GBTs) or XGBoost for regression.

9. Log-Rank Test (Survival Analysis)

- Description: Used in survival analysis trees, the log-rank test measures the difference between survival times across groups.
- Usage: Used in survival decision trees where the outcome is the time until an event occurs (e.g., survival times).

10. Laplacian Smoothing (Classification)

- Description: A variant of Gini and entropy that applies Laplacian smoothing to handle cases with small sample sizes or skewed class distributions.
- Usage: Rarely used but applicable when handling noisy or imbalanced datasets.

To see how some of these work, we'll start with a random split of the iris dataset and see how the calculations are performed for classification before we use them in the bigger algorithm. To figure out where the split should occur, many such calculations are tested to determine where the optimal split should be (or an optimization algorithm can be applied).

```
# Load the dataset
data(iris)

# Set a seed for reproducibility
set.seed(123)

# Randomly split the dataset into a training set and test set
split_index <- sample(1:nrow(iris), 0.7 * nrow(iris))
train_data <- iris[split_index, ]
test_data <- iris[-split_index, ]

# Take a quick look at the training data
head(train_data)

calculate_gini <- function(class_counts) {
  total_samples <- sum(class_counts)
  gini <- 1 - sum((class_counts / total_samples) ^ 2)
  return(gini)
}

calculate_entropy <- function(class_counts) {
  total_samples <- sum(class_counts)
  probs <- class_counts / total_samples
  entropy <- -sum(probs * log2(probs + 1e-9)) # Adding a small value to avoid log(0)
  return(entropy)
```

```r
}

calculate_information_gain <- function(parent_class_counts, left_class_counts,
right_class_counts) {
  total_samples <- sum(parent_class_counts)
  left_weight <- sum(left_class_counts) / total_samples
  right_weight <- sum(right_class_counts) / total_samples

  parent_entropy <- calculate_entropy(parent_class_counts)
  left_entropy <- calculate_entropy(left_class_counts)
  right_entropy <- calculate_entropy(right_class_counts)

  info_gain <- parent_entropy - (left_weight * left_entropy + right_weight * right_entropy)
  return(info_gain)
}

calculate_chi_squared <- function(observed_counts, expected_counts) {
  chi_squared <- sum((observed_counts - expected_counts) ^ 2 / (expected_counts + 1e-9)) #
Adding a small value to avoid division by 0
  return(chi_squared)
}

calculate_twoing_rule <- function(left_class_counts, right_class_counts) {
  total_left <- sum(left_class_counts)
  total_right <- sum(right_class_counts)

  left_probs <- left_class_counts / total_left
  right_probs <- right_class_counts / total_right

  twoing <- 0.25 * sum(abs(left_probs - right_probs)) ^ 2
  return(twoing)
}

# Choose a feature and make a random split
feature <- "Petal.Length"
split_value <- median(train_data[[feature]])

# Split the data based on the chosen feature and split value
left_split <- train_data[train_data[[feature]] <= split_value, ]
right_split <- train_data[train_data[[feature]] > split_value, ]

# Count the class occurrences in each split
parent_class_counts <- table(train_data$Species)
left_class_counts <- table(left_split$Species)
right_class_counts <- table(right_split$Species)

# Ensure that class counts include all three species, even if some have zero count
```

```
left_class_counts <- left_class_counts[match(levels(train_data$Species),
names(left_class_counts), nomatch = 0)]
right_class_counts <- right_class_counts[match(levels(train_data$Species),
names(right_class_counts), nomatch = 0)]

# Fill NAs with zeros
left_class_counts[is.na(left_class_counts)] <- 0
right_class_counts[is.na(right_class_counts)] <- 0

# Calculate Gini Impurity for the parent and child nodes
gini_left <- calculate_gini(left_class_counts)
gini_right <- calculate_gini(right_class_counts)
gini_parent <- calculate_gini(parent_class_counts)

# Calculate Information Gain
info_gain <- calculate_information_gain(parent_class_counts, left_class_counts,
right_class_counts)

# Calculate Chi-Squared
expected_left <- parent_class_counts * (sum(left_class_counts) / sum(parent_class_counts))
expected_right <- parent_class_counts * (sum(right_class_counts) / sum(parent_class_counts))

chi_squared_left <- calculate_chi_squared(left_class_counts, expected_left)
chi_squared_right <- calculate_chi_squared(right_class_counts, expected_right)
chi_squared_total <- chi_squared_left + chi_squared_right

# Calculate Twoing Rule
twoing <- calculate_twoing_rule(left_class_counts, right_class_counts)

# Print results
cat("Gini Impurity (Parent):", gini_parent, "\n")
cat("Gini Impurity (Left):", gini_left, "\n")
cat("Gini Impurity (Right):", gini_right, "\n")
cat("Information Gain:", info_gain, "\n")
cat("Chi-Squared:", chi_squared_total, "\n")
cat("Twoing Rule:", twoing, "\n")
```

Each of these represents just one calculation. The comparison of splitting locations is where the splitting rules operate and we aren't doing comparisons just yet. Before we do that, let's look at the regression splitting criteria.

```
# Load the dataset
data(mtcars)

# Set a seed for reproducibility
set.seed(123)

# Randomly split the dataset into a training set and test set
```

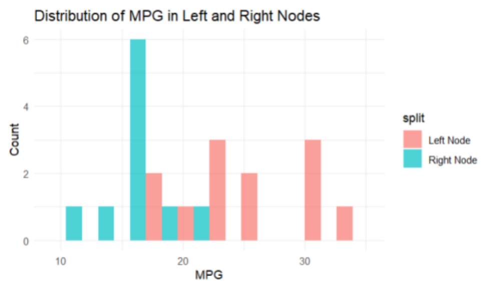
```r
split_index <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))
train_data <- mtcars[split_index, ]
test_data <- mtcars[-split_index, ]

# Take a quick look at the training data
head(train_data)

calculate_variance_reduction <- function(parent_values, left_values, right_values) {
  parent_variance <- var(parent_values)

  left_weight <- length(left_values) / length(parent_values)
  right_weight <- length(right_values) / length(parent_values)

  left_variance <- var(left_values)
  right_variance <- var(right_values)

  variance_reduction <- parent_variance - (left_weight * left_variance + right_weight *
right_variance)
  return(variance_reduction)
}

calculate_mse_reduction <- function(parent_values, left_values, right_values) {
  mse_parent <- mean((parent_values - mean(parent_values)) ^ 2)

  left_weight <- length(left_values) / length(parent_values)
  right_weight <- length(right_values) / length(parent_values)

  mse_left <- mean((left_values - mean(left_values)) ^ 2)
  mse_right <- mean((right_values - mean(right_values)) ^ 2)

  mse_reduction <- mse_parent - (left_weight * mse_left + right_weight * mse_right)
  return(mse_reduction)
}

# Choose a feature and make a random split
feature <- "hp"
split_value <- median(train_data[[feature]])

# Split the data based on the chosen feature and split value
left_split <- train_data[train_data[[feature]] <= split_value, ]
right_split <- train_data[train_data[[feature]] > split_value, ]

# Extract the target variable for parent and child nodes
parent_values <- train_data$mpg
left_values <- left_split$mpg
right_values <- right_split$mpg

# Calculate Variance Reduction
```

```
variance_reduction <- calculate_variance_reduction(parent_values, left_values, right_values)

# Calculate MSE Reduction
mse_reduction <- calculate_mse_reduction(parent_values, left_values, right_values)

# Print results
cat("Variance Reduction:", variance_reduction, "\n")
cat("MSE Reduction:", mse_reduction, "\n")
```

In the case of the Gini Impurity index, lower values indicate better splits, but in the others, higher values indicate better splits.

We can create a plot to visualize the results of the split.

```
# Load ggplot2 for visualization
library(ggplot2)

# Create a data frame to plot
plot_data <- data.frame(
  mpg = train_data$mpg,
  split = ifelse(train_data[[feature]] <= split_value, "Left Node", "Right Node")
)

# Plot histograms for each split
ggplot(plot_data, aes(x = mpg, fill = split)) +
  geom_histogram(position = "dodge", bins = 10, alpha = 0.7) +
  labs(title = "Distribution of MPG in Left and Right Nodes",
     x = "MPG", y = "Count") +
  theme_minimal()
```



Before we move on to the full decision tree algorithm, lets look at the stopping criteria options.

In decision tree algorithms, stopping criteria are conditions that prevent the tree from growing indefinitely and help avoid overfitting. Common stopping criteria include:

1. *Maximum Depth*: The tree is only allowed to grow to a certain depth.
2. *Minimum Number of Samples per Node*: A node will only be split if it contains a minimum number of samples.
3. *Minimum Reduction in Impurity*: A node will only be split if the reduction in impurity (e.g., variance reduction, MSE reduction) is above a certain threshold.
4. *Minimum Leaf Node Size*: The leaf nodes should have at least a minimum number of samples.

5. *Early Stopping Based on Validation Error*: If the validation error starts to increase, stop further splits.

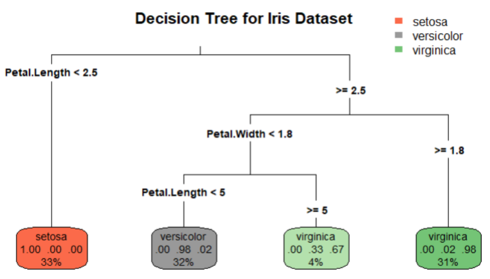
```
# Load the dataset
data(mtcars)

# Stopping criterion: Maximum Depth
max_depth_criterion <- function(current_depth, max_depth) {
  return(current_depth >= max_depth)
}

# Example Usage:
current_depth <- 3
max_depth <- 5
cat("Should we stop? ", max_depth_criterion(current_depth, max_depth), "\n")

# Stopping criterion: Minimum Number of Samples per Node
min_samples_criterion <- function(node_samples, min_samples) {
  return(length(node_samples) < min_samples)
}

# Example Usage:
node_samples <- train_data$mpg
min_samples <- 5
cat("Should we stop? ", min_samples_criterion(node_samples, min_samples), "\n")

# Stopping criterion: Minimum Reduction in Impurity
min_impurity_criterion <- function(impurity_reduction, min_impurity_reduction) {
  return(impurity_reduction < min_impurity_reduction)
}

# Example Usage:
impurity_reduction <- 0.02
min_impurity_reduction <- 0.05
cat("Should we stop? ", min_impurity_criterion(impurity_reduction, min_impurity_reduction), "\n")

# Stopping criterion: Minimum Leaf Node Size
min_leaf_size_criterion <- function(left_node, right_node, min_leaf_size) {
  return(length(left_node) < min_leaf_size || length(right_node) < min_leaf_size)
}

# Example Usage:
left_node <- train_data$mpg[train_data$hp <= 100]
right_node <- train_data$mpg[train_data$hp > 100]
min_leaf_size <- 3
cat("Should we stop? ", min_leaf_size_criterion(left_node, right_node, min_leaf_size), "\n")

# Stopping criterion: Early Stopping Based on Validation Error
```

```
early_stopping_criterion <- function(validation_errors) {
  if (length(validation_errors) < 2) {
    return(FALSE)
  }
  return(validation_errors[length(validation_errors)] > validation_errors[length(validation_errors) -
1])
}

# Example Usage:
validation_errors <- c(0.3, 0.28, 0.27, 0.29)  # Example validation errors from previous splits
cat("Should we stop? ", early_stopping_criterion(validation_errors), "\n")
```

Now that we have all the pieces that can be customized, let's look at the full decision tree algorithm to see how all the pieces fit together.

```
#full classification algorithm
# Load the iris dataset
data(iris)

# Calculate Gini Impurity
gini_impurity <- function(labels) {
  probs <- table(labels) / length(labels)
  return(1 - sum(probs^2))
}

# Calculate the mode (most common class)
get_mode <- function(labels) {
  return(names(sort(table(labels), decreasing = TRUE))[1])
}

# Function to split the dataset
split_dataset <- function(dataset, feature_index, threshold) {
  left <- dataset[dataset[, feature_index] <= threshold, ]
  right <- dataset[dataset[, feature_index] > threshold, ]
  return(list(left = left, right = right))
}

# Stopping criteria: max depth and minimum samples
max_depth_criterion <- function(current_depth, max_depth) {
  return(current_depth >= max_depth)
}

min_samples_criterion <- function(data, min_samples) {
  return(nrow(data) < min_samples)
}

# Recursive function to build the tree
build_tree <- function(data, current_depth = 0, max_depth = 5, min_samples = 5) {
  labels <- data[, ncol(data)]

  # Stopping criteria
  if (max_depth_criterion(current_depth, max_depth) || min_samples_criterion(data,
min_samples) || length(unique(labels)) == 1) {
    return(list(leaf = TRUE, prediction = get_mode(labels)))
  }

  best_gain <- -Inf
  best_split <- NULL

  # Find the best split
  for (feature_index in 1:(ncol(data) - 1)) {
    unique_values <- unique(data[, feature_index])

    for (threshold in unique_values) {
      splits <- split_dataset(data, feature_index, threshold)
      left_labels <- splits$left[, ncol(splits$left)]
      right_labels <- splits$right[, ncol(splits$right)]

      # Skip if either split is empty
      if (length(left_labels) == 0 || length(right_labels) == 0) {
        next
      }

      # Compute the Gini Gain
      gain <- gini_impurity(labels) -
        (length(left_labels) / length(labels)) * gini_impurity(left_labels) -
        (length(right_labels) / length(labels)) * gini_impurity(right_labels)

      if (!is.na(gain) && gain > best_gain) {
        best_gain <- gain
        best_split <- list(feature_index = feature_index, threshold = threshold, left = splits$left, right
= splits$right)
      }
    }
  }

  # If no good split found, return a leaf
  if (is.null(best_split)) {
    return(list(leaf = TRUE, prediction = get_mode(labels)))
  }

  # Recurse for the left and right branches
  left_branch <- build_tree(best_split$left, current_depth + 1, max_depth, min_samples)
  right_branch <- build_tree(best_split$right, current_depth + 1, max_depth, min_samples)
  return(list(leaf = FALSE, feature_index = best_split$feature_index, threshold =
best_split$threshold, left = left_branch, right = right_branch))
}

# Predict using the tree
predict_tree <- function(tree, sample) {
  if (tree$leaf) {
    return(tree$prediction)
  }

  if (sample[tree$feature_index] <= tree$threshold) {
    return(predict_tree(tree$left, sample))
  } else {
    return(predict_tree(tree$right, sample))
  }
}

# Train the tree
tree <- build_tree(iris, max_depth = 4, min_samples = 5)

# Predict on the entire dataset
predictions <- sapply(1:nrow(iris), function(i) predict_tree(tree, iris[i, -ncol(iris)]))

# Confusion Matrix
table(Predicted = predictions, Actual = iris$Species)
```

Visualizing a manually coded tree is more complicated than trees built from packages, so to visualize the tree itself, we'll re-implement a similar tree using a package and then apply its built-in visualization tools to get an idea of what we have here.

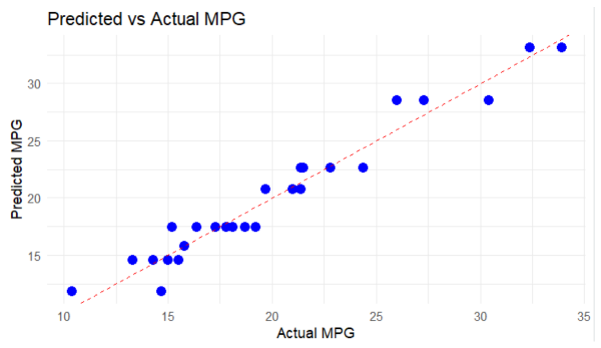


Now, let's consider the full regression algorithm.

```
#regression tree
# Load the mtcars dataset
data(mtcars)

# Step 1: Define a function to calculate the Mean Squared Error (MSE)
mse <- function(y) {
  mean((y - mean(y))^2)
```

```
}

# Step 2: Define a function to calculate the variance reduction (gain)
variance_reduction <- function(y, left_indices, right_indices) {
  total_var = mse(y)
  left_var = mse(y[left_indices])
  right_var = mse(y[right_indices])
  n = length(y)
  left_weight = length(left_indices) / n
  right_weight = length(right_indices) / n
  reduction = total_var - (left_weight * left_var + right_weight * right_var)
  return(reduction)
}

# Step 3: Define a function to find the best split
find_best_split <- function(X, y) {
  best_gain = -Inf
  best_split = NULL

  n_features = ncol(X)

  for (feature in 1:n_features) {
    feature_values = X[, feature]
    possible_splits = unique(feature_values)

    for (split_value in possible_splits) {
      left_indices = which(feature_values <= split_value)
      right_indices = which(feature_values > split_value)

      if (length(left_indices) == 0 || length(right_indices) == 0) {
        next  # Skip if any split has no data points
      }

      gain = variance_reduction(y, left_indices, right_indices)

      # Ensure gain is not NA or NaN
      if (is.na(gain) || is.nan(gain)) {
        next  # Skip invalid gain values
      }

      if (gain > best_gain) {
        best_gain = gain
        best_split = list(feature = feature, value = split_value, left_indices = left_indices,
right_indices = right_indices)
      }
    }
  }
  return(best_split)
}

# Step 4: Define a function to build the regression tree
build_tree <- function(X, y, min_samples_split = 10, max_depth = 5, current_depth = 1) {
  if (nrow(X) < min_samples_split || current_depth > max_depth) {
    return(list(prediction = mean(y)))  # Return a leaf node with the mean value
  }

  split = find_best_split(X, y)

  if (is.null(split)) {
    return(list(prediction = mean(y)))  # Return a leaf node if no valid split is found
  }

  left_tree = build_tree(X[split$left_indices, , drop = FALSE], y[split$left_indices],
min_samples_split, max_depth, current_depth + 1)
  right_tree = build_tree(X[split$right_indices, , drop = FALSE], y[split$right_indices],
min_samples_split, max_depth, current_depth + 1)

  return(list(
    feature = split$feature,
    value = split$value,
    left = left_tree,
    right = right_tree
  ))
}

# Step 5: Define a function to make predictions with the regression tree
predict_tree <- function(tree, X) {
  if (!is.null(tree$prediction)) {
    return(tree$prediction)
  }

  feature_value = X[tree$feature]

  if (feature_value <= tree$value) {
    return(predict_tree(tree$left, X))
  } else {
    return(predict_tree(tree$right, X))
  }
}

# Step 6: Example usage with the mtcars dataset
set.seed(42)
X <- as.matrix(mtcars[, c("wt", "qsec", "am")])
y <- mtcars$mpg
```

```r
# Build the tree
regression_tree <- build_tree(X, y, min_samples_split = 5, max_depth = 4)

# Make predictions
predictions <- apply(X, 1, function(row) predict_tree(regression_tree, row))

# Print the predictions
print(predictions)

# Function to print the decision tree structure
print_tree <- function(tree, indent = "") {
  if (!is.null(tree$prediction)) {
    cat(indent, "Prediction:", round(tree$prediction, 2), "\n")
  } else {
    cat(indent, "Feature", tree$feature, "<=", tree$value, "\n")
    cat(indent, "Left:\n")
    print_tree(tree$left, indent = paste0(indent, "  "))
    cat(indent, "Right:\n")
    print_tree(tree$right, indent = paste0(indent, "  "))
  }
}

# Print the tree
print_tree(regression_tree)

# Visualize the predictions vs actual values
library(ggplot2)

# Create a data frame for the comparison
results <- data.frame(
  Actual = y,
  Predicted = predictions
)

# Plot the results
ggplot(results, aes(x = Actual, y = Predicted)) +
  geom_point(color = "blue", size = 3) +
  geom_abline(intercept = 0, slope = 1, color = "red", linetype = "dashed") +
  labs(title = "Predicted vs Actual MPG",
       x = "Actual MPG",
       y = "Predicted MPG") +
  theme_minimal()
```



This code includes both a visualization of the predictions versus actual, and a printed (text) representation of the decision tree which prints to the console.

Resources:

1. https://scientistcafe.com/ids/splitting-criteria
2. https://www.analyticsvidhya.com/blog/2020/06/4-ways-split-decision-tree/
3. https://www.geeksforgeeks.org/how-to-determine-the-best-split-in-decision-tree/
4. https://www.machinelearningnuggets.com/splitting-criteria-in-decision-trees/