**Machine Learning Algorithms Part2**

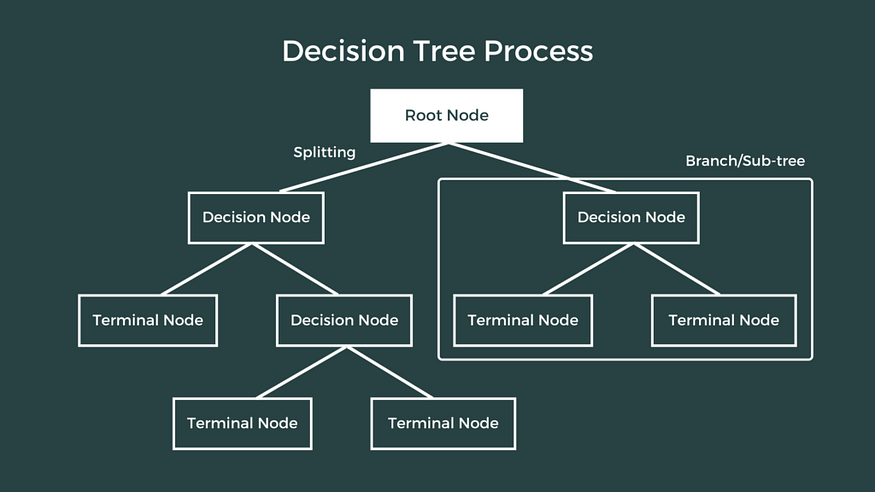
* + **1.1.1.3 Decision Tree Regressor**

<https://www.geeksforgeeks.org/python-decision-tree-regression-using-sklearn/>

<https://www.youtube.com/watch?v=b_4nzmNK3tY&list=PLKnIA16_RmvYGY_n9PP8zN-0LG9MoZRjU&index=5>

Decision tree hyperparameters

Hyper parameter tuning using grid search CV

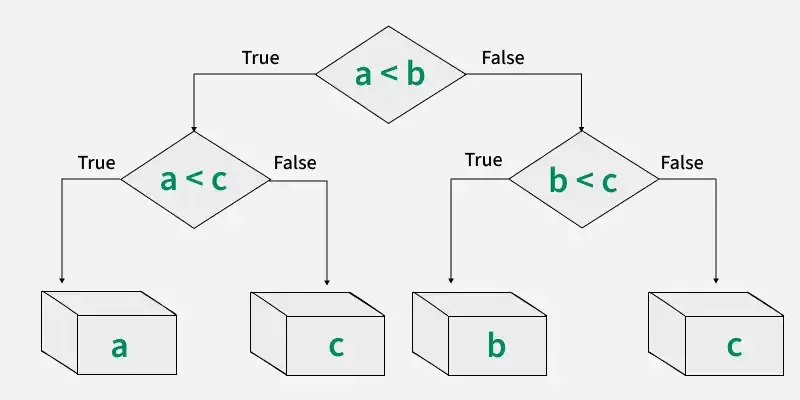


**What is it?**

A **Decision Tree Regressor** splits the dataset into smaller subsets based on feature values and learns simple decision rules at each node to predict continuous values. It is non-linear and non-parametric.

Decision Tree Regression is a method used to predict continuous values like prices or scores by using a tree-like structure. It works by splitting the data into smaller parts based on simple rules taken from the input features. These splits help reduce errors in prediction. At the end of each branch, called a leaf node the model gives a prediction usually the average value of that group. In the tree:

* **Decision Nodes** (shown as diamonds) ask yes/no questions about the data, like “Is age greater than 50?”
* **Leaf Nodes** (shown as rectangles) give the final predicted number based on the data that reached that point.



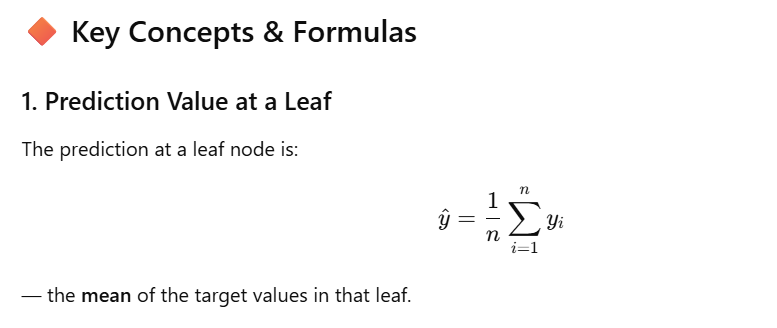
**Workflow of Decision Tree Regression**

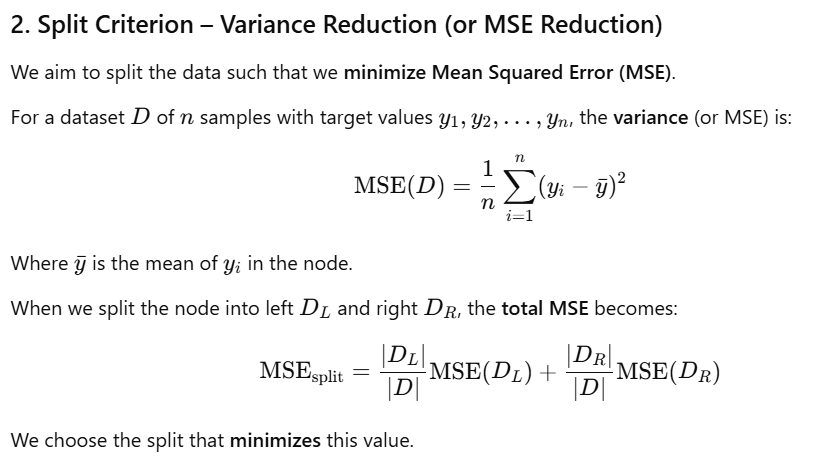
Branches connect nodes and represent the outcome of a decision. For example if the answer to a condition is "Yes," you follow one branch; if "No," you follow another. In below example it shows a decision tree that evaluates the smallest of three numbers:

**What is Decision Tree Regression?**

Unlike classification, which predicts categories (e.g., "yes"/"no"), **regression trees** predict **continuous numerical values** (e.g., price, age, temperature).

The tree is built by **splitting the dataset based on input features** in such a way that **the variance (error) of the target values is minimized** in each branch.





**Introduction**

Decision Tree Regressors are a powerful, interpretable, and non-linear method used widely for regression tasks in machine learning. Unlike linear regression, decision trees partition the feature space in a hierarchical, rule-based way that enables them to capture complex, non-linear relationships. In this post, we’ll delve deeply into the intuition, structure, and mechanics of decision trees for regression, exploring how they work, why they’re useful, and what makes them a versatile choice for various prediction tasks.

**1. Core Concept of Decision Tree Regressor**

A Decision Tree Regressor is a type of supervised learning algorithm that uses a flowchart-like structure to predict a continuous target variable based on decision rules inferred from the data features. The model continuously splits data into subsets, based on the features that result in the lowest prediction error, forming a tree-like structure where:

* **Each internal node** represents a decision rule on a feature.
* **Each branch** represents the outcome of a decision.
* **Each leaf node** provides the predicted value (usually the average of values in that subset of data).

This structure allows the model to capture non-linear relationships in the data by focusing on minimizing errors through the splits.

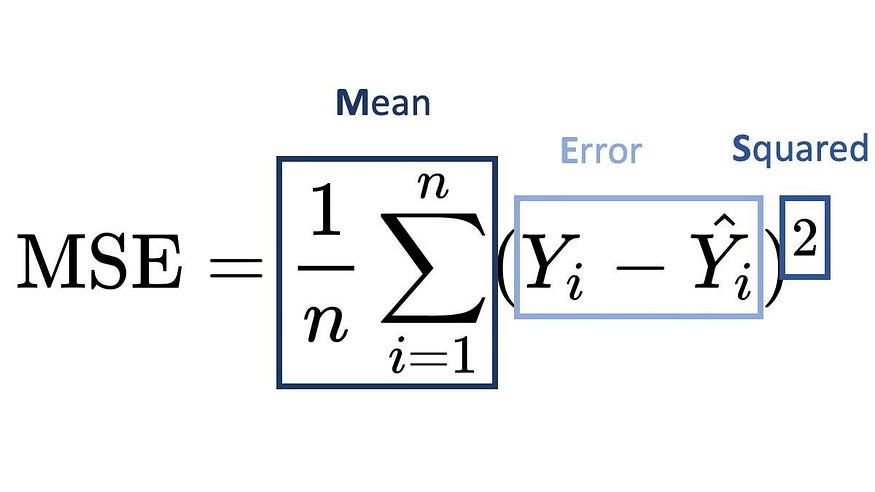
**2. How Splitting Happens: The Mechanics of Decision Tree Regression**

The most critical step in decision tree regression is determining where to split the data at each node. Here’s how it works:

**a. Mean Squared Error (MSE) as the Splitting Criterion**

To evaluate potential splits, Decision Tree Regressors typically use **Mean Squared Error (MSE)** as the criterion for minimizing error. When considering a split at a node, the model aims to reduce the MSE of the target variable, calculated as:

Zoom image will be displayed



where:

* N is the number of data points in the node,
* yi​ is the target value for observation i,
* yˉ​ is the mean of the target values in that node.

Each possible split is evaluated by calculating the MSE for the left and right child nodes after the split. The best split minimizes the weighted average of the MSE of these child nodes:

MSE\_split = (N\_L / N) \* MSE\_L + (N\_R / N) \* MSE\_R

where NL and NR are the sizes of the left and right child nodes, respectively.

**b. Recursive Binary Splitting**

Once the best split is determined, the data is divided into two subsets (left and right child nodes), and the process is repeated for each subset. This **recursive binary splitting** continues until a stopping criterion is met, such as:

* Maximum tree depth.
* Minimum number of samples in a node.
* Minimum reduction in error by further splitting.

**3. Building the Tree: An Example with Simple Data**

To illustrate the mechanics of a Decision Tree Regressor, let’s consider a small dataset with a single feature XXX and target variable yyy:

X = 1, 2, 3, 4, 5

Y = 2, 3, 4, 5, 6

The tree-building process would involve:

1. **Evaluating Splits**: Consider potential split points within X (e.g., between values 1.5, 2.5, etc.).
2. **Calculating MSE**: For each split point, compute MSE for the resulting subsets.
3. **Choosing the Split**: Select the split with the lowest weighted MSE.

For instance, if splitting at X=2.5 gives the lowest MSE, it becomes the first split, creating left and right child nodes. This process continues recursively.

**4. Key Parameters That Control Decision Tree Behavior**

To prevent overfitting or underfitting, Decision Tree Regressors have key parameters that control the depth and structure of the tree:

* **Max Depth**: The maximum depth of the tree. Limits growth to avoid overfitting.
* **Min Samples Split**: The minimum number of samples required to consider a split at a node.
* **Min Samples Leaf**: The minimum number of samples required to be in a leaf node.

These parameters ensure the model generalizes better by controlling how deeply the data can be divided.

Certainly! Here’s an in-depth blog-style exploration of Decision Tree Regressor, breaking down each aspect of the model, its intuition, and application.

**5. Advantages and Drawbacks**

**Advantages:**

* **Interpretability**: Each decision path in the tree represents a clear set of rules.
* **Non-linearity**: Decision trees can capture complex relationships without requiring feature transformation.
* **Feature Importance**: Trees inherently provide information on feature importance.

**Drawbacks:**

* **Overfitting**: Deep trees can capture noise, leading to poor generalization.
* **Instability**: Small changes in data can lead to entirely different trees.
* **Limited Smoothness**: Predictions may not be as smooth as linear models.

**6. Pruning: Avoiding Overfitting**

Pruning is a technique used to remove sections of the tree that provide little predictive power, simplifying the model to prevent overfitting. **Cost-complexity pruning** (or post-pruning) is commonly applied, where the tree is grown fully, then branches are pruned back based on a threshold for error reduction.

This allows the model to keep only the most significant splits, ensuring a balance between complexity and generalization.

**7. Applications and Use Cases**

Decision Tree Regressors are suited for:

* **Predicting housing prices**: Accounting for location, size, and other non-linear factors.
* **Estimating demand**: Factoring in complex seasonality and external conditions.
* **Stock market analysis**: Capturing patterns in historical data that impact prices.

They’re also useful as base models in ensemble methods like **Random Forests** and **Gradient Boosting**, which aggregate multiple trees to reduce variance and improve accuracy.

**8. A Walkthrough Example in Python**

Using Python’s DecisionTreeRegressor from Scikit-learn, we can build a Decision Tree Regressor on the Boston Housing dataset:

from sklearn.tree import DecisionTreeRegressor  
from sklearn.datasets import load\_boston  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import mean\_squared\_error

# Load dataset  
data = load\_boston()  
X, y = data.data, data.target# Split data  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)# Instantiate and train model  
tree\_regressor = DecisionTreeRegressor(max\_depth=4)  
tree\_regressor.fit(X\_train, y\_train)# Predictions and evaluation  
y\_pred = tree\_regressor.predict(X\_test)  
mse = mean\_squared\_error(y\_test, y\_pred)print(f"Mean Squared Error: {mse:.2f}")

In this example, we limited the depth to prevent overfitting, achieving a reasonable trade-off between accuracy and complexity.

Certainly! Adding a section on dtreeviz provides a way to visualize and interpret the decision tree structure and splitting process, which enhances our understanding of how decision trees operate.

**9. Visualizing Decision Tree Splits with**dtreeviz

One of the most effective ways to understand a Decision Tree Regressor is to visualize its structure. **dtreeviz** is a Python library that provides an intuitive, visual representation of decision trees, showing detailed splits, decision rules, and even error distributions at each node. This can be particularly helpful for interpreting complex trees and gaining insights into how the model divides data at each step.

**a. Installation and Setup**

To get started with dtreeviz, install it using pip:

pip install dtreeviz

Once installed, we can visualize our Decision Tree Regressor model on a sample dataset, such as the famous Iris dataset.

**b. Visualizing a Decision Tree Regressor on the Iris Dataset**

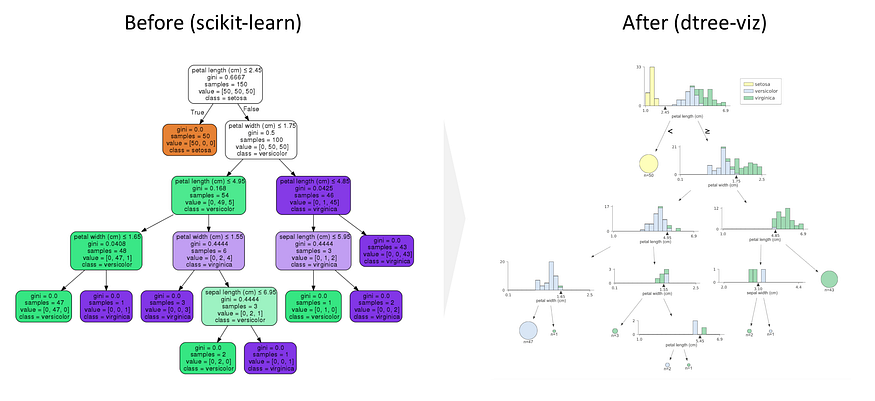
Let’s create a decision tree, fit it on the Iris dataset, and visualize it using dtreeviz. Although the Iris dataset is more commonly used for classification, this example will demonstrate how to use dtreeviz with a Decision Tree Regressor.

from sklearn.datasets import load\_iris  
from sklearn.tree import DecisionTreeRegressor  
from dtreeviz.trees import dtreeviz  
import matplotlib.pyplot as plt

# Load the Iris dataset  
data = load\_iris()  
X = data.data[:, 2:] # Use petal length and width  
y = data.data[:, 0] # Predict sepal length as a continuous variable# Fit the Decision Tree Regressor  
regressor = DecisionTreeRegressor(max\_depth=3)  
regressor.fit(X, y)# Visualize the tree with dtreeviz  
viz = dtreeviz(regressor, X, y,  
 feature\_names=data.feature\_names[2:],  
 target\_name="sepal length",  
 title="Decision Tree Regressor on Iris Data")viz.view() # This will open an interactive window with tree visualization

**c. Understanding the Visualization**

Zoom image will be displayed



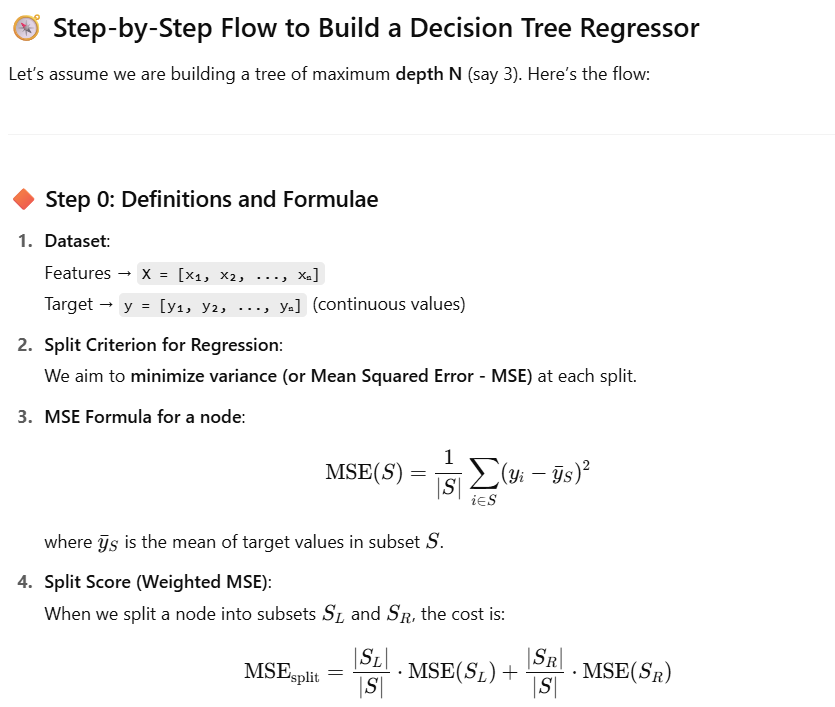
With dtreeviz, we get a detailed graphical representation of the decision tree, highlighting:

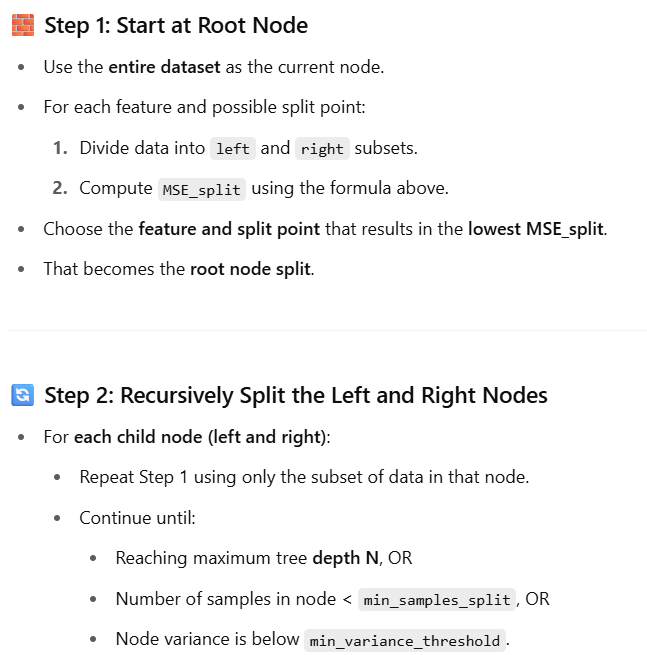
* **Split Conditions**: Each node displays the feature and threshold used to make the split, allowing us to see which features are most influential in dividing the data at different tree depths.
* **Error Reduction**: dtreeviz shows histograms of target values in each node, providing insight into the distribution and error at each step.
* **Color Coding**: The visualizer uses color coding to make the regression tree’s prediction boundaries and node distributions clearer. This is particularly helpful for understanding how the tree structure adapts to data patterns in non-linear regression tasks.

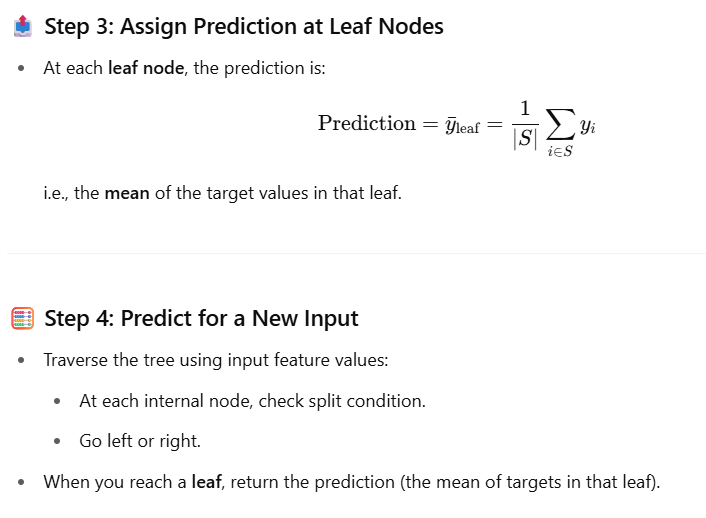
By examining these splits visually, we can see exactly how the tree makes decisions and reduces the error, step by step, in a clear and intuitive way. Visual tools like dtreeviz make interpreting decision trees much easier, turning a complex series of splits into an accessible, meaningful visualization.

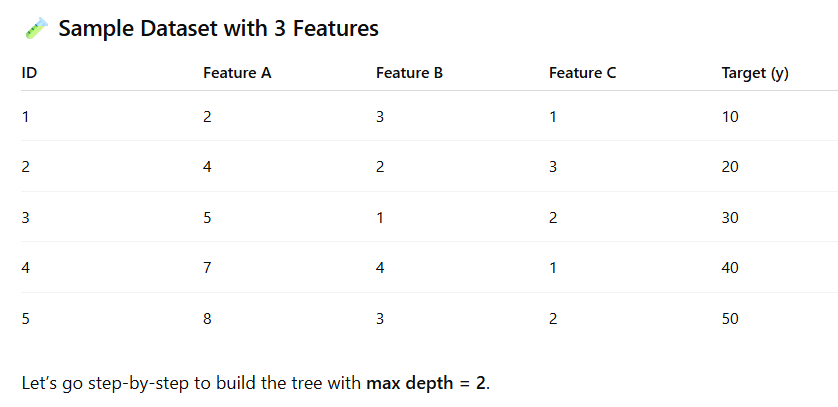
**Conclusion**

Decision Tree Regressors are versatile and intuitive models that allow us to capture complex, non-linear relationships in data through a simple, rule-based approach. By recursively splitting the data into smaller and more homogeneous subsets, decision trees create a structured pathway for making predictions. While their interpretability and non-linear capabilities make them highly valuable, especially for small to medium datasets, they can be prone to overfitting without proper tuning or pruning. However, this limitation is often mitigated when they serve as building blocks in ensemble methods like Random Forests or Gradient Boosting, which boost their predictive power and generalization.









**🪜 GENERIC TREE BUILDING STEPS**

**Step 1: For current dataset**

* If max depth reached or samples too small → make a **leaf node with mean target value**
* Else:
  1. Loop through each **feature**
  2. For each feature, find **all possible split points** between adjacent sorted values
  3. For each split, compute **Total MSE**
  4. Choose split with **minimum Total MSE**
  5. Split data into left and right subsets
  6. Recur on each subset

**Step-by-Step Algorithm Overview**

At **each node**, perform:

1. For **each feature**:
   * Sort the data points based on that feature.
   * Compute candidate split points as the **average of adjacent feature values**.
   * For each split point:
     + Split the data into left/right subsets.
     + Calculate **MSE** for the left and right splits.
     + Calculate **Weighted MSE**.
2. Select the **feature + split value** that gives the **lowest Weighted MSE**.
3. Repeat the process recursively for left and right subsets (until max depth or minimum samples reached).

**STEP 1: Find Best Split for Root Node (Depth 1)**

We'll compute all candidate splits for each feature and pick the best.

**🔹 Feature 1 (X1)**

Sorted X1 values: [2.0, 3.0, 4.0, 5.0, 6.0, 7.0]

Candidate split points (midpoints):

* **(2.0 + 3.0)/2 = 2.5**
* **(3.0 + 4.0)/2 = 3.5**
* **(4.0 + 5.0)/2 = 4.5**
* **(5.0 + 6.0)/2 = 5.5**
* **(6.0 + 7.0)/2 = 6.5**

Let’s calculate for **split = 4.5**

**➤ Left Split (X1 ≤ 4.5):**

Rows: ID 1, 2, 3  
Target Y: [10, 12, 13]  
Mean = (10+12+13)/3 = 11.67  
MSE = [(10-11.67)² + (12-11.67)² + (13-11.67)²] / 3  
= [2.78 + 0.11 + 1.78] / 3 ≈ **1.56**

**➤ Right Split (X1 > 4.5):**

Rows: ID 4, 5, 6  
Target Y: [20, 22, 25]  
Mean = (20+22+25)/3 = 22.33  
MSE = [(20-22.33)² + (22-22.33)² + (25-22.33)²] / 3  
= [5.44 + 0.11 + 7.11] / 3 ≈ **4.22**

**➤ Weighted MSE:**

= (3/6) \* 1.56 + (3/6) \* 4.22 = **2.89**

We'll store this. Try next feature.

**🔹 Feature 2 (X2)**

Unique sorted: [1.0, 2.0, 3.0]

Midpoints: 1.5, 2.5

Try split at **2.5**

**➤ Left: X2 ≤ 2.5 → ID 1,2,3,5**

Y: [10,12,13,22] → Mean: 14.25  
MSE ≈ [(10-14.25)² + (12-14.25)² + (13-14.25)² + (22-14.25)²] / 4  
≈ [18.06 + 5.06 + 1.56 + 60.06] / 4 ≈ **21.69**

**➤ Right: ID 4,6 → Y: [20,25] → Mean: 22.5**

MSE = [(20-22.5)² + (25-22.5)²]/2 = [6.25 + 6.25]/2 = **6.25**

Weighted MSE = (4/6)\*21.69 + (2/6)\*6.25 = **16.54**

Not better than Feature 1 (2.89)

**🔹 Feature 3 (X3)**

Sorted: [5.0, 6.0, 7.0, 7.5, 8.0, 9.0]

Midpoints: 5.5, 6.5, 7.25, 7.75, 8.5

Try split at **7.25**

**➤ Left: X3 ≤ 7.25 → ID 1,2,3**

Y = [10,12,13] → same as earlier → MSE = **1.56**

**➤ Right: X3 > 7.25 → ID 4,5,6**

Y = [20,22,25] → same → MSE = **4.22**

Weighted MSE = **2.89**

Equal to best so far.

**✅ Best Split (Step 1):**

Choose **Feature 1 @ 4.5** (or Feature 3 @ 7.25), both give Weighted MSE = **2.89**  
We’ll choose **Feature 1 (X1) ≤ 4.5** as root split.

**✅ STEP 2: Recursively split Left and Right branches (Depth 2)**

**🔹 Left Node (X1 ≤ 4.5): [ID 1, 2, 3], Y = [10,12,13]**

Try splitting again.

**Try Feature 2 @ 1.5**

Left: X2 ≤ 1.5 → ID 1,3 → Y = [10,13] → Mean = 11.5  
MSE = [(10-11.5)² + (13-11.5)²]/2 = [2.25 + 2.25]/2 = **2.25**

Right: ID 2 → Y = [12] → MSE = 0

Weighted MSE = (2/3)\*2.25 + (1/3)\*0 = **1.5**

Try Feature 3 @ 5.5

Left: X3 ≤ 5.5 → ID 1 → Y = [10] → MSE = 0  
Right: ID 2,3 → Y = [12,13] → Mean = 12.5 → MSE = [(12-12.5)² + (13-12.5)²]/2 = 0.25

Weighted MSE = (1/3)\*0 + (2/3)\*0.25 = **0.17** → ✅ Best

Split this node at **X3 ≤ 5.5**

**✅ STEP 3: Right Node (X1 > 4.5) → [ID 4,5,6], Y = [20,22,25]**

Try Feature 3 @ 7.75

Left: [20,22] → Mean = 21 → MSE = 1  
Right: [25] → MSE = 0  
Weighted MSE = (2/3)\*1 + (1/3)\*0 = **0.67**

This is best split → ✅ Use **Feature 3 ≤ 7.75**

**🌳 Final Tree (Depth = 3)**

text

CopyEdit

Root: X1 ≤ 4.5

├── Yes:

│ └── X3 ≤ 5.5

│ ├── Yes: Predict 10

│ └── No: Predict mean(12,13)=12.5

└── No:

└── X3 ≤ 7.75

├── Yes: Predict mean(20,22)=21

└── No: Predict 25