Lecture 19: Regression Trees and Pruning

CS109A: Introduction to Data Science

Harvard University

■ **Course:** CS109A: Introduction to Data Science

■ **Lecture:** Lecture 19

■ **Instructors:** Pavlos Protopapas, Kevin Rader, Chris Gumb

■ **Topics:** Regression Trees, MSE as Splitting Criterion, Categorical Variables, Pruning, Cost Complexity

Lecture Overview

This lecture extends decision trees from classification to **regression** and introduces **pruning** as a powerful technique for controlling overfitting.

Key Topics:

1. **Regression Trees:** Using decision trees to predict continuous outcomes (instead of class labels)
2. **Categorical Variables:** Handling non-numeric features in decision trees
3. **Pruning:** A post-hoc approach to reducing tree complexity and preventing overfitting

Key Insight: The core ideas from classification trees carry over directly—we just swap impurity measures (Gini, Entropy) for **MSE**, and swap majority voting for **averaging**.

Contents

1 Regression Trees: From Classification to Prediction

1.1 Recap: Classification Trees

In classification trees, we learned:

- **Goal:** Predict a categorical outcome (e.g., “lemon” or “orange”)
- **Splitting criterion:** Minimize impurity (Gini index or Entropy)
- **Prediction:** Majority vote—predict the most common class in the leaf node

1.2 What Changes for Regression?

For **regression trees**, we want to predict a **continuous** outcome (e.g., house price, temperature, stock return).

Definition:

Regression Tree A **regression tree** is a decision tree where:

- The target variable y is **continuous** (not categorical)
- Each leaf node predicts the **mean** of the training samples in that region
- Splits are chosen to minimize **MSE** (Mean Squared Error) instead of impurity

Table 1: *Classification vs Regression Trees*

Aspect	**Classification Tree**	**Regression Tree**
Target variable	Categorical (classes)	Continuous (numbers)
Splitting criterion	Gini impurity or Entropy	MSE (Mean Squared Error)
Prediction at leaf	Majority class	Mean of samples
Example prediction	“Survived” or “Did not survive”	“$450,000” or “25.3 degrees”

1.3 The Splitting Criterion: Minimizing MSE

In classification, we wanted each region to be “pure”—containing mostly one class.

In regression, we want each region to have **low variance**—containing samples with similar y values.

Definition:

MSE as Splitting Criterion For a region R with n samples, the MSE is:

$$\text{MSE}(R) = \frac{1}{n}\sum_{i\in R}(y_i - \bar{y}_R)^2 \tag{1}$$

Where $\bar{y}_R$ is the mean of all y values in region R.
This is simply the **variance** of y in that region!

Key Information

Why MSE Makes Sense:

If all samples in a region have similar y values:

- The mean $\bar{y}_R$ represents them well
- MSE is low (samples are close to the mean)
- Predicting $\bar{y}_R$ for new samples will be accurate

If samples have very different y values:

- The mean doesn't represent any sample well
- MSE is high (samples are far from the mean)
- Predicting $\bar{y}_R$ will have large errors

1.4 Finding the Best Split

Just like classification trees, we search for the best split by:

1. For each feature p
2. For each possible threshold t
3. Calculate the **weighted average MSE** of the two resulting regions:

$$\text{Split MSE} = \frac{N_1}{N} \cdot \text{MSE}(R_1) + \frac{N_2}{N} \cdot \text{MSE}(R_2) \tag{2}$$

4. Choose (p^*, t^*) that minimizes this weighted MSE

Example:

Simple Regression Tree Split Consider 1D data with X values from 0 to 10 and continuous Y values.

Try split at $X = 6.5$:

- Region 1 ($X \leq 6.5$): Contains samples with $\bar{y}_{R_1} = -0.008$
- Region 2 ($X > 6.5$): Contains samples with $\bar{y}_{R_2} = 0.697$

Calculate MSE for each region and their weighted average.

The algorithm tries **every possible split point** (every unique X value) and picks the one with lowest weighted MSE.

Warning

Computational Cost:

For each predictor with n unique values, we try n possible splits. With p predictors, that's potentially $n \times p$ calculations at each node.

For 1000 data points and 5 predictors: 5000 split evaluations per node!

1.5 Making Predictions

Once the tree is built:

1. A new data point traverses the tree (same as classification)

2. It lands in some leaf node
3. The prediction is the **mean** of training samples in that leaf

Example:

Regression Tree Prediction A trained tree has leaf node L_3 containing training samples with y values: $\{2.1, 2.3, 2.5, 2.0, 2.4\}$
The stored prediction for L_3 is: $\bar{y}_{L_3} = \frac{2.1+2.3+2.5+2.0+2.4}{5} = 2.26$
Any new sample that reaches L_3 gets prediction $\hat{y} = 2.26$

1.6 Visualizing Regression Trees

In 1D (one predictor):

- The predicted function is a **step function**
- Each step corresponds to a leaf node's mean
- More splits = more steps = more complex function

In 2D (two predictors):

- Feature space is partitioned into rectangles
- Each rectangle has a constant predicted value (the mean)
- It's like a "terraced landscape" with flat regions at different heights

1.7 Stopping Conditions

The same stopping conditions from classification trees apply:

- `max_depth`: Maximum tree depth
- `min_samples_leaf`: Minimum samples required in a leaf
- `max_leaf_nodes`: Maximum number of leaf nodes
- **MSE Gain Threshold**: Stop if MSE reduction from split is below threshold

Definition:

Accuracy Gain (MSE Reduction) The "accuracy gain" or MSE reduction from a split is:

$$\text{Gain} = \text{MSE}(\text{parent}) - \left[\frac{N_1}{N} \cdot \text{MSE}(R_1) + \frac{N_2}{N} \cdot \text{MSE}(R_2)\right] \tag{3}$$

If Gain < threshold, don't split (the improvement isn't worth the added complexity).

1.8 Cross-Validation for Regression Trees

How do we choose the right stopping condition values?

Cross-validation! But with a different metric:

- Classification: Use accuracy, F1-score, or AUC
- Regression: Use MSE (or R^2) on the validation set

Key Information

MSE vs R^2 for Model Selection:

The instructor prefers using MSE over R^2 for cross-validation because:

- MSE is consistent across validation folds
- R^2 can vary because the mean $\bar{y}$ changes with different validation sets
- This introduces extra randomness in R^2 comparisons

2 Handling Categorical Variables

2.1 The Problem

Decision trees make splits based on comparisons: "Is $X > t$?"

For **numerical** features, this makes sense: "Is height > 178 cm?"

For **categorical** features, it doesn't: "Is color > Red?" has no meaning!

2.2 Bad Approach: Ordinal Encoding

You might think: "Just assign numbers! Yellow=0, Red=1, Purple=2"

Warning

Why Ordinal Encoding Fails:
This creates an **artificial ordering** that doesn't exist in the data.
With Yellow=0, Red=1, Purple=2:

- Split at "Color ≥ 1" separates {Yellow} from {Red, Purple}

With Yellow=2, Red=0, Purple=1:

- Split at "Color ≥ 1" separates {Red} from {Yellow, Purple}

The tree structure depends on an arbitrary encoding choice!

2.3 Good Approach: One-Hot Encoding

Definition:

One-Hot Encoding (OHE) Convert each categorical variable into multiple binary (0/1) columns—one for each category.
Original: `Color` $\in$ {Yellow, Red, Purple}
After OHE:

- `Color_Yellow`: 1 if Yellow, 0 otherwise
- `Color_Red`: 1 if Red, 0 otherwise
- `Color_Purple`: 1 if Purple, 0 otherwise

Now splits make sense: "Is Color_Red ≥ 1?" means "Is the color Red?"

Example:

One-Hot Encoding Example **Before:**

Sepal Width	Color
3.0	Yellow
3.5	Red
3.7	Purple

After One-Hot Encoding:

Sepal Width	Color_Yellow	Color_Red	Color_Purple
3.0	1	0	0
3.5	0	1	0
3.7	0	0	1

Warning

Sklearn Warning:
Scikit-learn's `DecisionTreeClassifier` and `DecisionTreeRegressor` do **NOT** automatically handle categorical variables.
You must apply one-hot encoding **before** fitting the model:

```python
import pandas as pd

# Method 1: Pandas get_dummies
X_encoded = pd.get_dummies(X, columns=['Color'])

# Method 2: sklearn OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)
```

Note: Libraries like XGBoost, LightGBM, and CatBoost can handle categoricals natively.

Key Information

Dropping One Column:
With 3 categories, you only need 2 binary columns (the third is determined).
If Color_Yellow=0 and Color_Red=0, then it must be Purple.
However, for trees this redundancy usually doesn't matter much.

3 Pruning: A Better Way to Prevent Overfitting

3.1 The Problem with Stopping Conditions

We've seen stopping conditions like `max_depth`. But there's a problem:

- **Too restrictive:** Stop too early → underfitting
- **Too lenient:** Stop too late → overfitting
- **Hard to know in advance:** The optimal stopping point depends on the data

3.2 The Pruning Philosophy

Definition:

Pruning **Pruning** is a post-hoc technique where we:

1. First, grow the tree **fully** (or very deep)
2. Then, **remove** branches that don't improve performance

It's like letting the tree grow wild, then trimming it back carefully.

Key Information

Analogy: Pruning Real Trees

When pruning a real tree:

- Let it grow
- Cut back "dry, dead, or diseased" branches
- Result: A healthier, better-shaped tree

When pruning a decision tree:

- Let it grow (overfit)
- Cut back "useless" branches (that don't help on validation data)
- Result: A simpler, better-generalizing model

3.3 Cost Complexity Pruning (CCP)

The standard approach to pruning is **Cost Complexity Pruning**, which adds a penalty for tree size.

Definition:

Cost Complexity The **cost complexity** of a tree T is:

$$C_\alpha(T) = \text{Error}(T) + \alpha \cdot |T| \tag{4}$$

Where:

- $\text{Error}(T)$ = Classification error (or MSE for regression) on training data
- $|T|$ = Number of **leaf nodes** (terminal nodes)
- α = Complexity parameter (hyperparameter, $\alpha \geq 0$)

This is **regularization** applied to trees!

- Like Ridge regression adds $\lambda \sum \beta_j^2$ to penalize large coefficients
- CCP adds $\alpha \cdot |T|$ to penalize large trees

3.4 The Role of Alpha (α)

The complexity parameter α controls the trade-off:

- $\alpha = 0$**:** No penalty for tree size $\rightarrow$ favor large trees (full tree)
- $\alpha \rightarrow \infty$**:** Huge penalty $\rightarrow$ favor tiny trees (just the root)
- $0 < \alpha < \infty$**:** Balance between error and simplicity

Example:

Cost Complexity Comparison Let $\alpha = 0.2$

Full Tree T:

- Error(T) = 0.32
- $|T| = 8$ leaves
- $C_\alpha(T) = 0.32 + 0.2 \times 8 = \mathbf{1.92}$

Pruned Tree T':

- Error(T') = 0.33 (slightly worse training error)
- $|T'| = 7$ leaves
- $C_\alpha(T') = 0.33 + 0.2 \times 7 = \mathbf{1.73}$

Even though T' has higher training error, it has **lower cost complexity**!

The regularization favors T' because the added complexity of T isn't worth the small error reduction.

3.5 The Pruning Algorithm

Step 1: Generate Candidate Trees

Start with the full tree T_0. Iteratively remove the "weakest link"—the subtree whose removal **increases cost complexity the least** (or decreases it most).

This generates a sequence: $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_L$ (root only)

Step 2: Find Optimal Tree for Given α

For a fixed α, calculate $C_\alpha(T)$ for each candidate tree. Select the one with minimum cost complexity.

Step 3: Find Optimal α via Cross-Validation

Try different α values and use cross-validation to find which gives best validation performance.

Key Information

Nested Cross-Validation:

This is a two-level process:

1. **Inner loop:** For fixed α, find best pruned tree

2. **Outer loop:** Try different α values, select best via CV

This is computationally expensive but gives more robust model selection.

3.6 Pruning in Practice

```python
from sklearn.tree import DecisionTreeRegressor

# Train a full tree
tree = DecisionTreeRegressor(random_state=42)
tree.fit(X_train, y_train)

# Get the cost complexity pruning path
path = tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path.ccp_alphas  # Array of alpha values

# Train trees for each alpha value
trees = []
for alpha in ccp_alphas:
    t = DecisionTreeRegressor(ccp_alpha=alpha, random_state=42)
    t.fit(X_train, y_train)
    trees.append(t)

# Evaluate on validation set to choose best alpha
val_scores = [t.score(X_val, y_val) for t in trees]
best_alpha = ccp_alphas[np.argmax(val_scores)]

# Final model
final_tree = DecisionTreeRegressor(ccp_alpha=best_alpha)
final_tree.fit(X_train, y_train)
```

4 Important Details and Edge Cases

4.1 Can the Same Feature Be Used Multiple Times?

Yes! A feature can appear in multiple splits throughout the tree.

Example:

Multiple Splits on Same Feature Consider predicting income based on work experience:

- Root: "Work experience $>$ 2 years?"
- Left child: "Work experience $>$ 0.5 years?"
- Right child: "Work experience $>$ 5 years?"

The same feature (work experience) is split at different thresholds at different levels. This allows the tree to capture non-linear relationships!

4.2 What Metrics for Cross-Validation?

For Classification:

- Accuracy (for balanced data)
- F1-score (for imbalanced data)
- AUC-ROC (for general performance)
- Custom utility function (if you care more about certain errors)

For Regression:

- MSE (preferred—consistent across folds)
- R^2 (works but less stable)
- MAE (if you want robustness to outliers)

4.3 Decision Boundary Shape

Important:

Linear vs Non-Linear Boundaries **max_depth=1:** Only one split $\rightarrow$ **Linear** decision boundary (single line/hyperplane)

max_depth$\geq$2: Multiple splits $\rightarrow$ **Non-linear** boundaries (staircase pattern)

The deeper the tree, the more complex (more "wiggly") the boundary.

4.4 Total MSE of a Regression Tree

The overall MSE of a tree with multiple leaves is the **weighted average** of leaf MSEs:

$$\text{Total MSE} = \sum_{l=1}^{L} \frac{N_l}{N} \cdot \text{MSE}(R_l) \tag{5}$$

Example:

Computing Total MSE A tree has 4 leaf nodes:

- R_1: 90 samples, MSE = 0.2
- R_2: 5 samples, MSE = 1.2
- R_3: 3 samples, MSE = 1.5
- R_4: 2 samples, MSE = 1.8

Total samples: $N = 90 + 5 + 3 + 2 = 100$

Total MSE = $\frac{90}{100}(0.2) + \frac{5}{100}(1.2) + \frac{3}{100}(1.5) + \frac{2}{100}(1.8)$

$= 0.18 + 0.06 + 0.045 + 0.036 = \mathbf{0.321}$

Note: The total is dominated by R_1 because it contains 90% of samples!

5 Summary and Key Takeaways

Key Summary

Regression Trees:

- Same structure as classification trees, different criterion
- Splitting criterion: Minimize **weighted average MSE**
- Prediction: **Mean** of training samples in leaf node
- Stopping conditions: Same as classification (max_depth, min_samples_leaf, etc.)

Categorical Variables:

- Cannot use ordinal encoding (creates artificial order)
- Use **One-Hot Encoding** to create binary columns
- Sklearn requires manual OHE before fitting

Pruning:

- Alternative to pre-defined stopping conditions
- "Grow full, then cut back"
- Cost complexity: $C_\alpha(T) = \text{Error}(T) + \alpha|T|$
- α controls trade-off between error and simplicity
- Find optimal α via cross-validation

6 Learning Checklist

☐ Can you explain the difference between classification and regression trees?

☐ Do you understand why MSE is used as the splitting criterion for regression?

☐ Can you explain what the prediction at a leaf node represents in regression trees?

☐ Do you know why ordinal encoding is problematic for categorical variables?

☐ Can you apply one-hot encoding to categorical features?

☐ Do you understand the cost complexity formula: $C_\alpha(T) = \text{Error}(T) + \alpha|T|$?

☐ Can you explain how α affects the complexity of the pruned tree?

☐ Do you know the two-level cross-validation process for finding optimal α?

☐ Can you compute the total MSE of a tree from its leaf node MSEs?

☐ Do you understand why `max_depth=1` gives a linear decision boundary?

7 Looking Ahead

Next lectures will cover **ensemble methods**:

- **Bagging:** Building many trees on bootstrap samples
- **Random Forests:** Bagging + random feature selection
- **Boosting:** Building trees sequentially to correct errors

- **Gradient Boosting:** XGBoost, LightGBM—state-of-the-art methods

These methods combine multiple trees to create more powerful and robust models!