Lec 6: Tree-Based Methods and Bagging

Xin Xing

Tree-Based Methods: Overview

Key Idea:

- Partition the feature space into rectangles
- Fit a simple model (constant) in each region
- Use recursive binary splitting

Advantages:

- Conceptually simple and interpretable
- Handle both regression and classification
- Automatic variable selection
- No need for transformations or dummy variables
- Can capture non-linear relationships and interactions

Popular Methods:

- CART (Classification and Regression Trees)

Recursive Binary Partitioning

The Process:

1. Start with all data in one region
2. Split into two regions along one variable at one split point
3. Continue splitting each region recursively
4. Stop when some criterion is met (e.g., minimum node size)

Example Partition:

- First split: $X_1 \leq t_1$
- Left region split: $X_2 \leq t_2$
- Right region split: $X_1 \leq t_3$
- Further split: $X_2 \leq t_4$
- Results in regions $R_1, R_2, \ldots, R_M$

Prediction Model:

$$\hat{f}(X) = \sum_{m=1}^{M} c_m I(X \in R_m)$$

Toy Example: Setup

Simple Regression Problem:

Consider a dataset with 8 observations, 2 predictors (X_1, X_2), and response Y:

i	X_1	X_2	Y
1	1	3	5
2	2	4	7
3	3	2	6
4	4	1	4
5	5	5	12
6	6	6	15
7	7	3	10
8	8	4	14

Goal: Build a regression tree to predict Y using X_1 and X_2

Criterion: Minimize sum of squared errors (SSE)

Toy Example: First Split - Consider X_1

Try splitting on X_1 at different values:

Split at $X_1 \leq 4$:

- Left region (R_1): observations 1,2,3,4 with $Y = \{5, 7, 6, 4\}$
- $\bar{y}_1 = (5 + 7 + 6 + 4)/4 = 5.5$
- $\text{SSE}_1 = (5 - 5.5)^2 + (7 - 5.5)^2 + (6 - 5.5)^2 + (4 - 5.5)^2 = 5.0$
- Right region (R_2): observations 5,6,7,8 with $Y = \{12, 15, 10, 14\}$
- $\bar{y}_2 = (12 + 15 + 10 + 14)/4 = 12.75$
- $\text{SSE}_2 =$
  $(12 - 12.75)^2 + (15 - 12.75)^2 + (10 - 12.75)^2 + (14 - 12.75)^2 = 14.75$

Total SSE for this split: $5.0 + 14.75 = 19.75$

Toy Example: Try Other Splits on X_1

Split at $X_1 \leq 5$:

- Left (R_1): $Y = \{5, 7, 6, 4, 12\}$, $\bar{y}_1 = 6.8$, $\text{SSE}_1 = 34.8$
- Right (R_2): $Y = \{15, 10, 14\}$, $\bar{y}_2 = 13.0$, $\text{SSE}_2 = 14.0$
- **Total SSE:** $34.8 + 14.0 = 48.8$

Split at $X_1 \leq 3$:

- Left (R_1): $Y = \{5, 7, 6\}$, $\bar{y}_1 = 6.0$, $\text{SSE}_1 = 2.0$
- Right (R_2): $Y = \{4, 12, 15, 10, 14\}$, $\bar{y}_2 = 11.0$, $\text{SSE}_2 = 74.0$
- **Total SSE:** $2.0 + 74.0 = 76.0$

Best split on X_1: $X_1 \leq 4$ with $\text{SSE} = 19.75$

Toy Example: Consider X_2

Now try splitting on X_2:

Split at $X_2 \leq 3$:

- Left (R_1): obs 1,4,7 with $Y = \{5, 4, 10\}$, $\bar{y}_1 = 6.33$, $\text{SSE}_1 = 18.67$
- Right (R_2): obs 2,3,5,6,8 with $Y = \{7, 6, 12, 15, 14\}$, $\bar{y}_2 = 10.8$, $\text{SSE}_2 = 60.8$
- **Total SSE:** $18.67 + 60.8 = 79.47$

Split at $X_2 \leq 4$:

- Left (R_1): obs 1,2,4,7,8 with $Y = \{5, 7, 4, 10, 14\}$, $\bar{y}_1 = 8.0$, $\text{SSE}_1 = 62.0$
- Right (R_2): obs 3,5,6 with $Y = \{6, 12, 15\}$, $\bar{y}_2 = 11.0$, $\text{SSE}_2 = 42.0$
- **Total SSE:** $62.0 + 42.0 = 104.0$

Best split overall: $X_1 \leq 4$ with $\text{SSE} = 19.75$

Toy Example: First Split Decision

Summary of all candidate splits:

Split	Total SSE
$X_1 \leq 3$	76.0
$X_1 \leq 4$	**19.75 $\leftarrow$ BEST**
$X_1 \leq 5$	48.8
$X_2 \leq 3$	79.47
$X_2 \leq 4$	104.0

First split: $X_1 \leq 4$

Resulting regions:

- R_1: obs 1,2,3,4 $\rightarrow$ predict $\hat{y} = 5.5$
- R_2: obs 5,6,7,8 $\rightarrow$ predict $\hat{y} = 12.75$

Next step: Recursively split R_1 and R_2 if beneficial

Toy Example: Recursive Splitting

Should we split R_1 further?

R_1 has obs 1,2,3,4 with $Y = \{5, 7, 6, 4\}$, current SSE $= 5.0$

Try splitting R_1 at $X_2 \leq 2$:

- Left: obs 4 with $Y = \{4\}$, $\bar{y} = 4$, SSE $= 0$
- Right: obs 1,2,3 with $Y = \{5, 7, 6\}$, $\bar{y} = 6$, SSE $= 2.0$
- Total SSE for R_1: $0 + 2.0 = 2.0$ (better than 5.0!)

Should we split R_2 further?

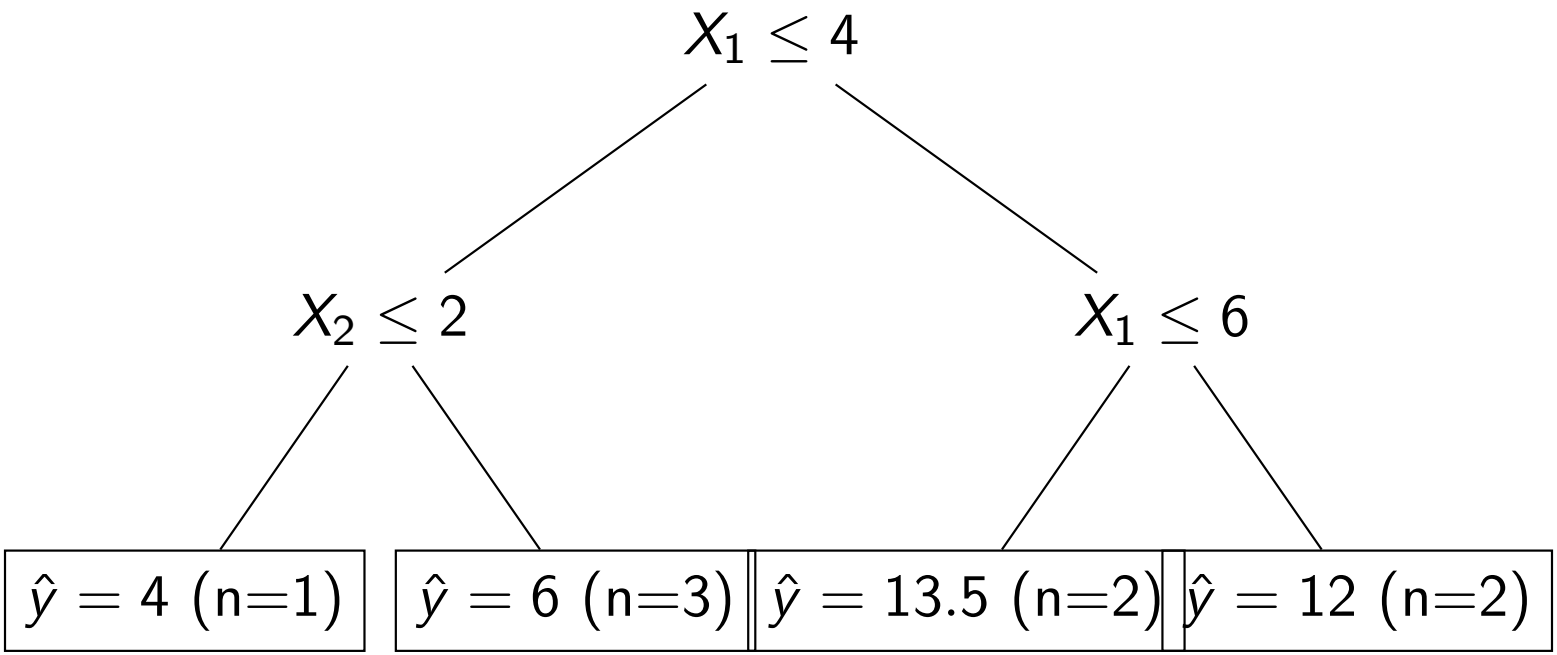
R_2 has obs 5,6,7,8 with $Y = \{12, 15, 10, 14\}$, current SSE $= 14.75$

Try splitting R_2 at $X_1 \leq 6$:

- Left: obs 5,6 with $Y = \{12, 15\}$, $\bar{y} = 13.5$, SSE $= 4.5$
- Right: obs 7,8 with $Y = \{10, 14\}$, $\bar{y} = 12$, SSE $= 8.0$
- Total SSE for R_2: $4.5 + 8.0 = 12.5$ (better than 14.75!)

Toy Example: Final Tree Structure

Tree after recursive splitting:

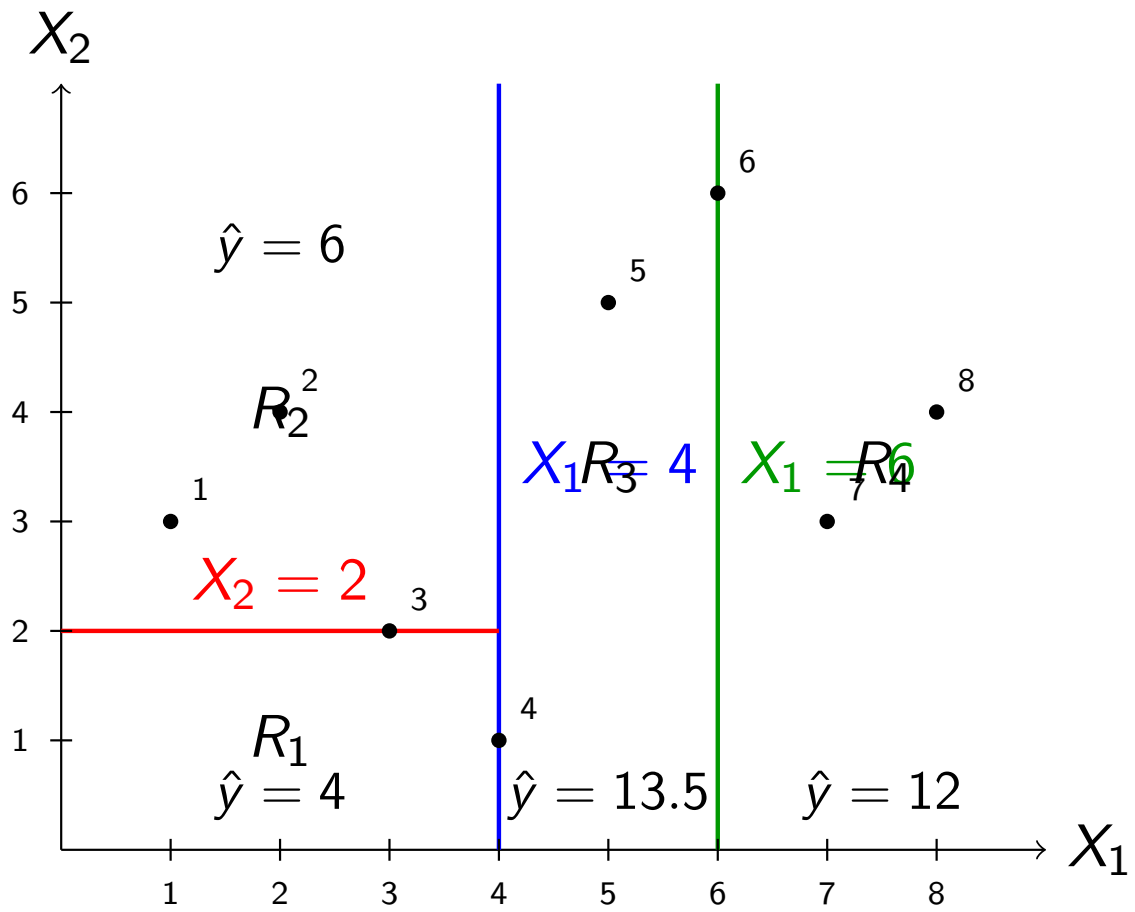


Four terminal regions:

1. $X_1 \leq 4$ and $X_2 \leq 2$: $\hat{y} = 4$
2. $X_1 \leq 4$ and $X_2 > 2$: $\hat{y} = 6$
3. $X_1 > 4$ and $X_1 \leq 6$: $\hat{y} = 13.5$
4. $X_1 > 4$ and $X_1 > 6$: $\hat{y} = 12$

Toy Example: Visual Representation

Partition of feature space:



Note: Recursive binary partitioning creates axis-aligned rectangles

Toy Example: Prediction Function

Mathematical representation:

$$\hat{f}(x_1, x_2) = \begin{cases} 4 & \text{if } x_1 \leq 4 \text{ and } x_2 \leq 2 \\ 6 & \text{if } x_1 \leq 4 \text{ and } x_2 > 2 \\ 13.5 & \text{if } x_1 > 4 \text{ and } x_1 \leq 6 \\ 12 & \text{if } x_1 > 6 \end{cases}$$

Or equivalently:

$$\hat{f}(X) = \sum_{m=1}^{4} c_m I(X \in R_m)$$

where $c_1 = 4, c_2 = 6, c_3 = 13.5, c_4 = 12$

Example prediction:

- For new point $(x_1 = 3, x_2 = 5)$: $x_1 \leq 4$ and $x_2 > 2 \rightarrow \hat{y} = 6$
- For new point $(x_1 = 7, x_2 = 2)$: $x_1 > 6 \rightarrow \hat{y} = 12$

Toy Example: Key Insights

What we learned:

1. **Greedy algorithm:** At each step, choose split that minimizes SSE
   - Don't look ahead to future splits
   - Locally optimal, not globally optimal
2. **Exhaustive search at each node:**
   - Try all variables
   - Try all possible split points
   - Choose best combination
3. **Recursive partitioning:**
   - Apply same procedure to each resulting region
   - Continue until stopping criterion met
4. **Simple predictions:**
   - Each region predicts constant (mean for regression)
   - Piecewise constant function

Why Greedy Can Miss Better Solutions

Example of greedy limitation:

Suppose the "true" best tree has this structure:

- First split: $X_2 \leq 3.5$ (SSE $= 25$)
- Then good splits follow (final SSE $= 10$)

But our greedy algorithm chose:

- First split: $X_1 \leq 4$ (SSE $= 19.75$) $\leftarrow$ better initially
- Then further splits (final SSE $= 14.5$)

The greedy choice was locally optimal but globally suboptimal!

Why we accept this:

- Finding globally optimal tree is NP-complete
- Greedy algorithm is computationally feasible
- Works well in practice
- Can be improved with ensemble methods

Regression Trees: Setup

Data:

- N observations: (x_i, y_i) for $i = 1, 2, \ldots, N$
- $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$ (p inputs)
- y_i is continuous response

Model with M regions:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m)$$

Optimal constants (minimize sum of squares):

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m)$$

Challenge:

- Finding the best binary partition is computationally infeasible
- Solution: Use a greedy algorithm

Growing Regression Trees: Greedy Algorithm

At each step, find best split:

- Consider splitting variable j and split point s
- Define half-planes:

$$R_1(j,s) = \{X|X_j \leq s\}, \quad R_2(j,s) = \{X|X_j > s\}$$

Optimization problem:

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right]$$

Solution:

- For fixed j, s: $\hat{c}_1 = \text{ave}(y_i|x_i \in R_1(j,s))$, $\hat{c}_2 = \text{ave}(y_i|x_i \in R_2(j,s))$
- Scan through all variables and split points
- Choose (j,s) that minimizes criterion
- Repeat recursively for each resulting region

Tree Size and Pruning

Problem: How large should the tree be?

- Too large: overfitting
- Too small: miss important structure

Strategy:

1. Grow a large tree T_0 (stop at minimum node size, e.g., 5)
2. Prune using cost-complexity pruning

Why not stop early?

- A seemingly worthless split might lead to very good splits below it
- Better to grow large and prune back

Cost-Complexity Pruning

Definitions:

- $|T|$ = number of terminal nodes
- N_m = number of observations in node m
- $\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i$
- $Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$

Cost-complexity criterion:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|$$

Interpretation:

- First term: model fit (training error)
- Second term: complexity penalty
- $\alpha \geq 0$: tuning parameter (larger $\alpha \rightarrow$ smaller tree)

Finding Optimal Subtree

Goal: For each α, find subtree T_α that minimizes $C_\alpha(T)$

Weakest Link Pruning:

1. Start with full tree T_0
2. Collapse internal node that produces smallest per-node increase in $\sum_m N_m Q_m(T)$
3. Continue until single-node tree
4. This sequence contains T_α for all α

Selecting α:

- Use 5-fold or 10-fold cross-validation
- Choose $\hat{\alpha}$ to minimize cross-validated error
- Final tree: $T_{\hat{\alpha}}$

Classification Trees

Setup:

- Response: $Y \in \{1, 2, \ldots, K\}$ (K classes)
- In node m: $\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$
- Prediction: majority class $k(m) = \arg\max_k \hat{p}_{mk}$

Key Difference from Regression:

- Need different node impurity measures
- Cannot use squared error

Node Impurity Measures

Three common measures $Q_m(T)$:

1. Misclassification Error:

$$\frac{1}{N_m} \sum_{i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}$$

2. Gini Index:

$$\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

3.Entropy (Deviance):

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

Comparing Impurity Measures

Similarities:

- All three measures are similar in shape
- Maximized when classes are equally distributed
- Minimized (zero) when node is pure

Key Differences:

- Gini and cross-entropy are differentiable
- Gini and cross-entropy more sensitive to probability changes
- Misclassification rate less sensitive

Example: Node with (400, 400) observations

- Split 1: (300, 100) and (100, 300)
- Split 2: (200, 400) and (200, 0)
- Both have misclassification rate $= 0.25$
- Split 2 creates pure node $\rightarrow$ preferable
- Gini and cross-entropy distinguish these splits

Recommendations for Classification Trees

Growing the tree:

- Use Gini index or cross-entropy
- These are more sensitive to node purity
- Better for identifying good splits

Cost-complexity pruning:

- Can use any of the three measures
- Typically use misclassification rate
- More directly related to classification accuracy

Interpretations of Gini Index

Interpretation 1: Classification Error

- Instead of using majority rule, classify to class k with probability $\hat{p}_{mk}$
- Expected training error: $\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'}$ (Gini index)

Interpretation 2: Variance

- Code each observation as 1 for class k, 0 otherwise
- Variance of this 0-1 response: $\hat{p}_{mk}(1 - \hat{p}_{mk})$
- Sum over classes $k \rightarrow$ Gini index

Both interpretations:

- Provide intuition for why Gini measures node impurity
- Low Gini $\rightarrow$ node is pure (dominated by one class)
- High Gini $\rightarrow$ node is mixed (classes roughly balanced)

Summary: Tree-Based Methods

Key Steps:

1. Grow tree using greedy recursive binary splitting
2. Choose splits to minimize impurity criterion
3. Prune using cost-complexity pruning
4. Select final tree size via cross-validation

Advantages:

- Easy to interpret and visualize
- Handles mixed variable types naturally
- Captures non-linearities and interactions

Disadvantages:

- High variance (unstable)
- Not competitive with best supervised methods
- Can be improved by ensemble methods (bagging, boosting, random forests)

R Code Example: Regression Tree

Load Libraries and Data:

```
library(rpart)
library(rpart.plot)

# Example: Boston housing data
library(MASS)
data(Boston)

# Split into train/test
set.seed(123)
train_idx <- sample(1:nrow(Boston), 0.7*nrow(Boston))
train <- Boston[train_idx, ]
test <- Boston[-train_idx, ]
```

R Code Example: Fitting Regression Tree

Fit Full Tree:

```
# Grow a large tree
tree_fit <- rpart(medv ~ .,
                  data = train,
                  method = "anova",  # for regression
                  control = rpart.control(
                    minsplit = 10,    # min obs for split
                    cp = 0.001))      # complexity param

# Print tree structure
print(tree_fit)
```

Key Parameters:

- `method = "anova"`: regression (use "class" for classification)
- `minsplit`: minimum observations to attempt split
- `cp`: complexity parameter (like α)

R Code Example: Pruning Regression Tree

Cross-Validation for Optimal cp:

```
# Print CV results
printcp(tree_fit)

# Plot CV error vs tree size
plotcp(tree_fit)

# Get optimal cp (minimum xerror)
opt_cp <- tree_fit$cptable[
  which.min(tree_fit$cptable[,"xerror"]), "CP"]

# Prune tree
pruned_tree <- prune(tree_fit, cp = opt_cp)
```

Note: `xerror` is the cross-validated error, `rel error` is training error

R Code Example: Visualizing Regression Tree

Plot the Tree:

```
# Simple plot
plot(pruned_tree)
text(pruned_tree, cex = 0.8)

# Better plot using rpart.plot
rpart.plot(pruned_tree,
           type = 4,              # label style
           extra = 101,           # show n and mean
           under = TRUE,          # put extra info under
           fallen.leaves = TRUE)
```

Make Predictions:

```
# Predictions on test set
pred <- predict(pruned_tree, newdata = test)

# Calculate RMSE
```

R Code Example: Classification Tree Setup

Load Data (Iris Example):

```
# Use iris dataset
data(iris)

# Split into train/test
set.seed(456)
train_idx <- sample(1:nrow(iris), 0.7*nrow(iris))
train <- iris[train_idx, ]
test <- iris[-train_idx, ]
```

R Code Example: Fitting Classification Tree

Fit Classification Tree:

```
# Grow classification tree
class_tree <- rpart(Species ~ .,
                    data = train,
                    method = "class",  # classification
                    parms = list(split = "gini"),
                    control = rpart.control(
                      minsplit = 5,
                      cp = 0.01))

# Can also use "information" for entropy
# parms = list(split = "information")
```

Split Criteria:

- `split = "gini"`: Gini index (default)
- `split = "information"`: Cross-entropy

R Code Example: Classification Tree Pruning

Prune Using Cross-Validation:

```
# View CV results
printcp(class_tree)

# Get optimal cp
opt_cp <- class_tree$cptable[
  which.min(class_tree$cptable[,"xerror"]), "CP"]

# Prune tree
pruned_class <- prune(class_tree, cp = opt_cp)

# Summary of pruned tree
summary(pruned_class)
```

R Code Example: Classification Predictions

Make Predictions:

```
# Class predictions
pred_class <- predict(pruned_class,
                      newdata = test,
                      type = "class")

# Probability predictions
pred_prob <- predict(pruned_class,
                     newdata = test,
                     type = "prob")

# Confusion matrix
table(Predicted = pred_class, Actual = test$Species)

# Accuracy
accuracy <- mean(pred_class == test$Species)
print(paste("Accuracy:", round(accuracy, 3)))
```

R Code Example: Visualizing Classification Tree

Plot Classification Tree:

```
# Enhanced plot
rpart.plot(pruned_class,
           type = 4,
           extra = 104,     # show prob and counts
           box.palette = "auto",
           shadow.col = "gray",
           nn = TRUE)       # display node numbers

# Variable importance
var_imp <- pruned_class$variable.importance
print(var_imp)

# Visualize importance
barplot(var_imp,
        main = "Variable Importance",
        las = 2,  # rotate labels
```

Complete Workflow Example

Putting It All Together:

```
# 1. Load and prepare data
library(rpart); library(rpart.plot)
data(iris)
set.seed(123)
idx <- sample(1:nrow(iris), 0.7*nrow(iris))

# 2. Fit tree
tree <- rpart(Species ~ ., data = iris[idx,],
              method = "class")

# 3. Cross-validate and prune
opt_cp <- tree$cptable[
  which.min(tree$cptable[,"xerror"]), "CP"]
pruned <- prune(tree, cp = opt_cp)

# 4. Evaluate
```

Alternative: Using tree Package

Another Popular Package:

```
library(tree)

# Fit tree
tree_fit <- tree(Species ~ ., data = train)

# Cross-validation
cv_tree <- cv.tree(tree_fit,
                   FUN = prune.misclass)

# Find optimal size
opt_size <- cv_tree$size[
  which.min(cv_tree$dev)]

# Prune
pruned <- prune.misclass(tree_fit,
                         best = opt_size)
```

Key R Packages for Trees

Package	**Description**
`rpart`	Most popular CART implementation
`rpart.plot`	Enhanced visualization for rpart
`tree`	Alternative tree package
`party`	Conditional inference trees
`C50`	C5.0 algorithm (successor to C4.5)
`randomForest`	Random forests
`gbm`	Gradient boosting machines
`xgboost`	Extreme gradient boosting

Recommendation: Start with `rpart` for learning, then explore ensemble methods.

Bootstrap Aggregation (Bagging)

Motivation:

- High-variance models (e.g., decision trees) are unstable
- Small changes in training data lead to very different predictions
- Can we reduce variance without increasing bias?

Key Idea:

- Average predictions from multiple bootstrap samples
- Exploits connection between bootstrap mean and posterior averaging

The Bagging Algorithm

Input: Training data $Z = \{(x_1, y_1), \ldots, (x_N, y_N)\}$

For $b = 1, 2, \ldots, B$:

1. Draw bootstrap sample Z^{*b} by sampling N observations from Z with replacement
2. Fit model to Z^{*b}, obtaining prediction $\hat{f}^{*b}(x)$

Output: Bagged estimate

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Note: As $B \to \infty$, this approaches $E_{\hat{P}}[\hat{f}^*(x)]$ where $\hat{P}$ is the empirical distribution.

When Does Bagging Help?

Bagging is effective when:

- The base learner is **nonlinear** or **adaptive**
- The model has **high variance**
- The procedure is **unstable**

Ineffective for linear models:

- Linear models: $\hat{f}_{\text{bag}}(x) \to \hat{f}(x)$ as $B \to \infty$
- Example: B-spline smoother (linear in data for fixed inputs)
- Bagging just reproduces the original smooth

Highly effective for:

- Decision trees
- Neural networks
- Other high-variance methods

Why Bagging Reduces Variance

Under squared-error loss, let:

- $f_{\text{ag}}(x) = E_P[\hat{f}^*(x)]$ be the ideal aggregate estimator
- Bootstrap samples drawn from true population P

Then:

$$\begin{aligned} E_P[Y - \hat{f}^*(x)]^2 &= E_P[Y - f_{\text{ag}}(x) + f_{\text{ag}}(x) - \hat{f}^*(x)]^2 \\ &= E_P[Y - f_{\text{ag}}(x)]^2 + E_P[\hat{f}^*(x) - f_{\text{ag}}(x)]^2 \\ &\geq E_P[Y - f_{\text{ag}}(x)]^2 \end{aligned}$$

Interpretation:

- Extra error comes from variance of $\hat{f}^*(x)$ around $f_{\text{ag}}(x)$
- Aggregation eliminates this variance term
- Averaging reduces variance while leaving bias unchanged

Bagging for Classification

Two Approaches:

1. Majority Vote (Hard Classification):

- Each tree predicts class $\hat{G}^{*b}(x) \in \{1, \ldots, K\}$
- $p_k(x) =$ proportion of trees predicting class k
- Final prediction: $\hat{G}_{\text{bag}}(x) = \arg\max_k p_k(x)$

2. Average Probabilities (Soft Classification):

- Each tree estimates class probabilities $\hat{f}^{*b}(x)$
- Average these probability estimates:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

- Final prediction: $\hat{G}_{\text{bag}}(x) = \arg\max_k \hat{f}_{\text{bag},k}(x)$

Recommendation: Averaging probabilities generally performs better, especially for small B.

Pitfall: Voting Proportions $\neq$ Probabilities

Simple Example:

- True probability: $P(Y = 1|x) = 0.75$
- Each bagged classifier accurately predicts class 1
- Voting proportion: $p_1(x) = 1.0$ (all trees vote for class 1)
- This is **not** a good estimate of 0.75!

Better Approach:

- For trees: use class proportions in terminal nodes
- Average these probability estimates across trees
- Produces calibrated probability estimates
- Lower variance, especially for small B

Bagging and 0-1 Loss

Important: The variance reduction argument doesn't hold for classification under 0-1 loss!

Why? Bias and variance are not additive for 0-1 loss.

Consequences:

- Bagging a **good** classifier can make it better
- Bagging a **bad** classifier can make it worse

Counterexample:

- Suppose $Y = 1$ for all x
- Classifier $\hat{G}(x)$ predicts:
  - $Y = 1$ with probability 0.4
  - $Y = 0$ with probability 0.6
- Misclassification rate of $\hat{G}(x)$: 0.6
- Misclassification rate of bagged classifier: 1.0 (worse!)

Wisdom of Crowds Perspective

For two-class problem, suppose:

- Bayes optimal decision: $G(x) = 1$
- Each weak learner has error rate $e < 0.5$
- Learners are independent
- Consensus vote: $S_1(x) = \sum_{b=1}^{B} I(G^{*b}(x) = 1)$

Result:

- $S_1(x) \sim \text{Binomial}(B, 1 - e)$
- $P(S_1 > B/2) \to 1$ as $B \to \infty$
- Majority vote becomes perfect as B increases!

Caveat:

- Requires independence (bootstrap trees are correlated!)
- Related to "Wisdom of Crowds" concept
- Random forests improve on this by reducing correlation

Example: Simulated Data with Trees

Setup:

- Sample size: $N = 30$
- Features: $p = 5$ (standard Gaussian, pairwise correlation 0.95)
- Response: $P(Y = 1|x_1 \leq 0.5) = 0.2$, $P(Y = 1|x_1 > 0.5) = 0.8$
- Bayes error rate: 0.2
- Test set: 2000 observations

Method:

- Fit unpruned classification trees
- Use $B = 200$ bootstrap samples

Observations:

- Trees differ in splitting features and cutpoints
- High variance due to correlated predictors
- Bagging smooths out this variance

Results: Test Error Comparison

Key Findings:

- Original tree: high test error (unstable)
- Bagged tree: substantially lower test error
- Averaging probabilities outperforms consensus voting
- Error decreases as B increases, stabilizes around $B = 50$

Why Bagging Helps Here:

- Trees have high variance (correlated predictors)
- Each bootstrap tree uses different features
- Averaging smooths out the instability
- Dramatic variance reduction $\Rightarrow$ improved prediction

Summary: Bagging

Main Ideas:

- Bootstrap aggregation averages predictions over bootstrap samples
- Reduces variance while maintaining bias (for squared error)
- Most effective for high-variance, unstable models

Best Practices:

- Use with unstable base learners (trees, neural nets)
- Average probability estimates, not just votes
- Typical choice: $B = 50$ to 200

Extensions:

- Random forests: reduce correlation between trees
- Out-of-bag error estimation
- Variable importance measures

R Code Example: Bagging for Regression

Using ipred Package:

```
library(ipred)

# Bagging for regression
bag_model <- bagging(medv ~ .,
                     data = train,
                     nbagg = 100,     # 100 bootstrap samples
                     coob = TRUE)     # compute OOB error

# Make predictions
pred_bag <- predict(bag_model, newdata = test)

# Calculate RMSE
rmse_bag <- sqrt(mean((test$medv - pred_bag)^2))
print(paste("Bagging RMSE:", round(rmse_bag, 3)))

# OOB error estimate
```

R Code Example: Bagging for Classification

Classification with Bagging:

```
library(ipred)

# Bagging for classification
bag_class <- bagging(Species ~ .,
                     data = train,
                     nbagg = 100,
                     coob = TRUE)

# Predictions
pred_class <- predict(bag_class, newdata = test)

# Accuracy
accuracy <- mean(pred_class == test$Species)
print(paste("Accuracy:", round(accuracy, 3)))

# Confusion matrix
```

R Code Example: Manual Bagging Implementation

Understanding Bagging by Coding It:

```
# Manual bagging for regression
set.seed(123)
B <- 100  # number of bootstrap samples
predictions <- matrix(0, nrow = nrow(test), ncol = B)

for(b in 1:B) {
  # Bootstrap sample
  boot_idx <- sample(1:nrow(train),
                     nrow(train),
                     replace = TRUE)
  boot_data <- train[boot_idx, ]

  # Fit tree
  tree_b <- rpart(medv ~ ., data = boot_data)

  # Predict
```