Tree-Based Methods: CART, Bagging, Random Forests, Boosting

M2 Statistics - Statistical Learning

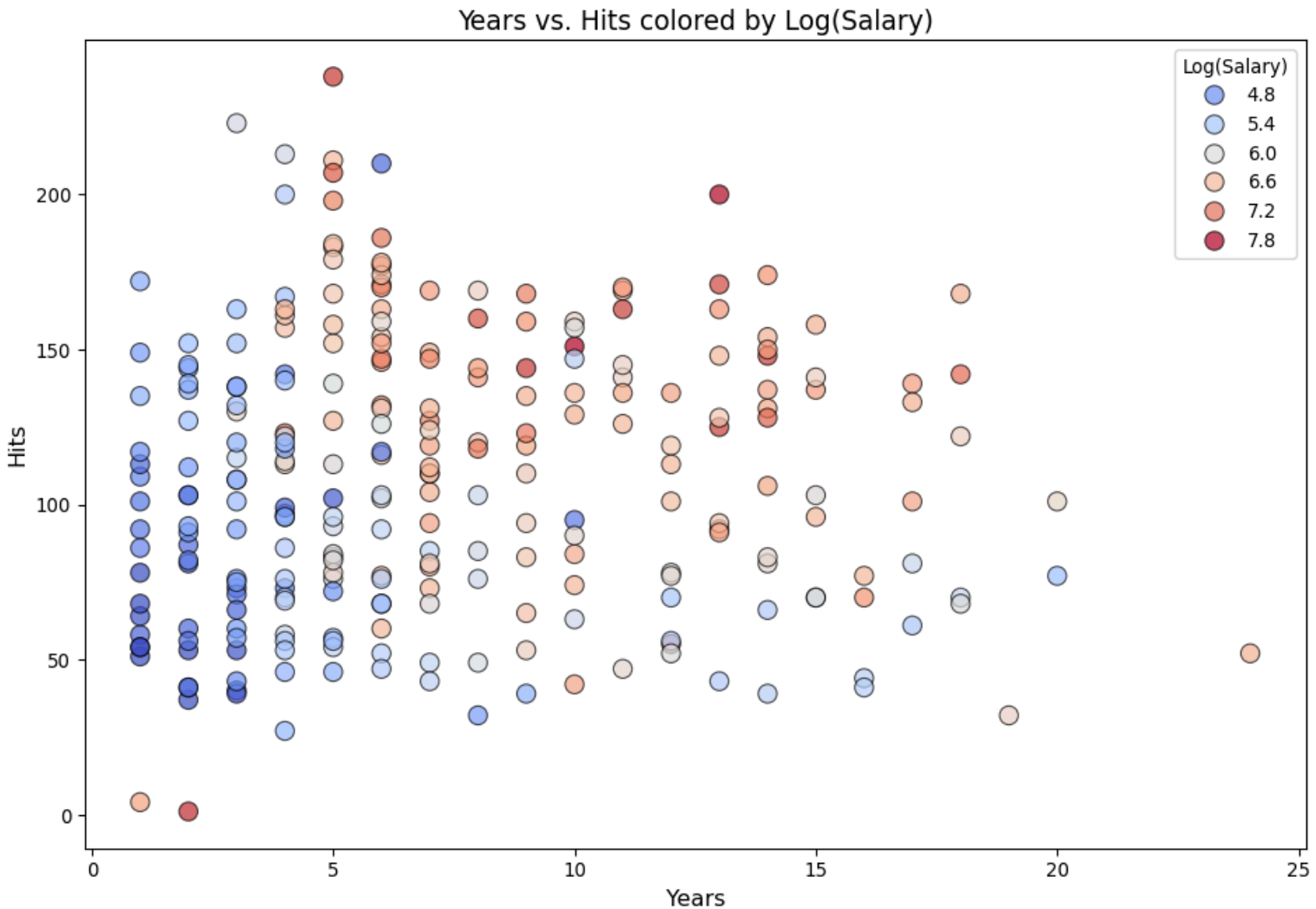
Pierre Pudlo

Aix-Marseille Université / Faculté des Sciences

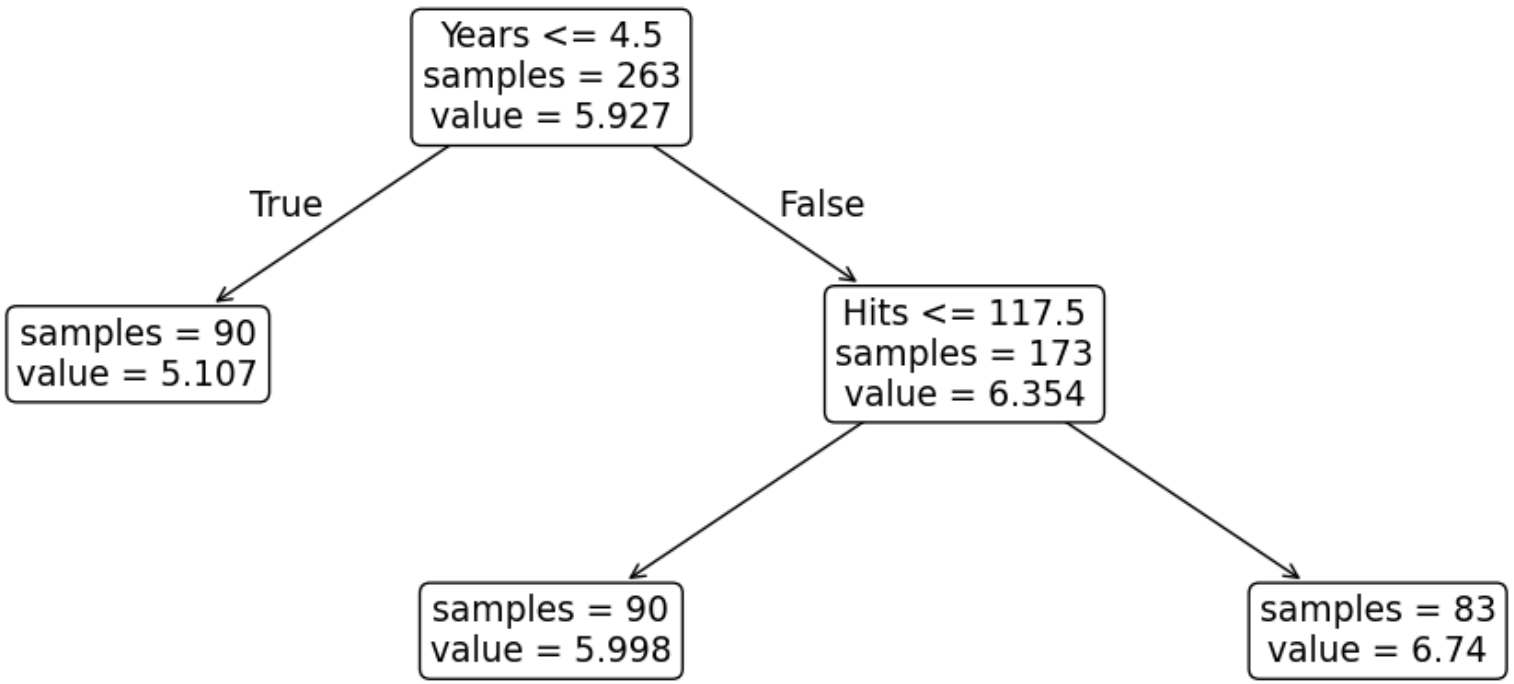
Introduction to Tree-Based Methods

Example: Baseball Salaries

- **Years**: number of years in the major leagues
- **Hits**: number of hits in previous year

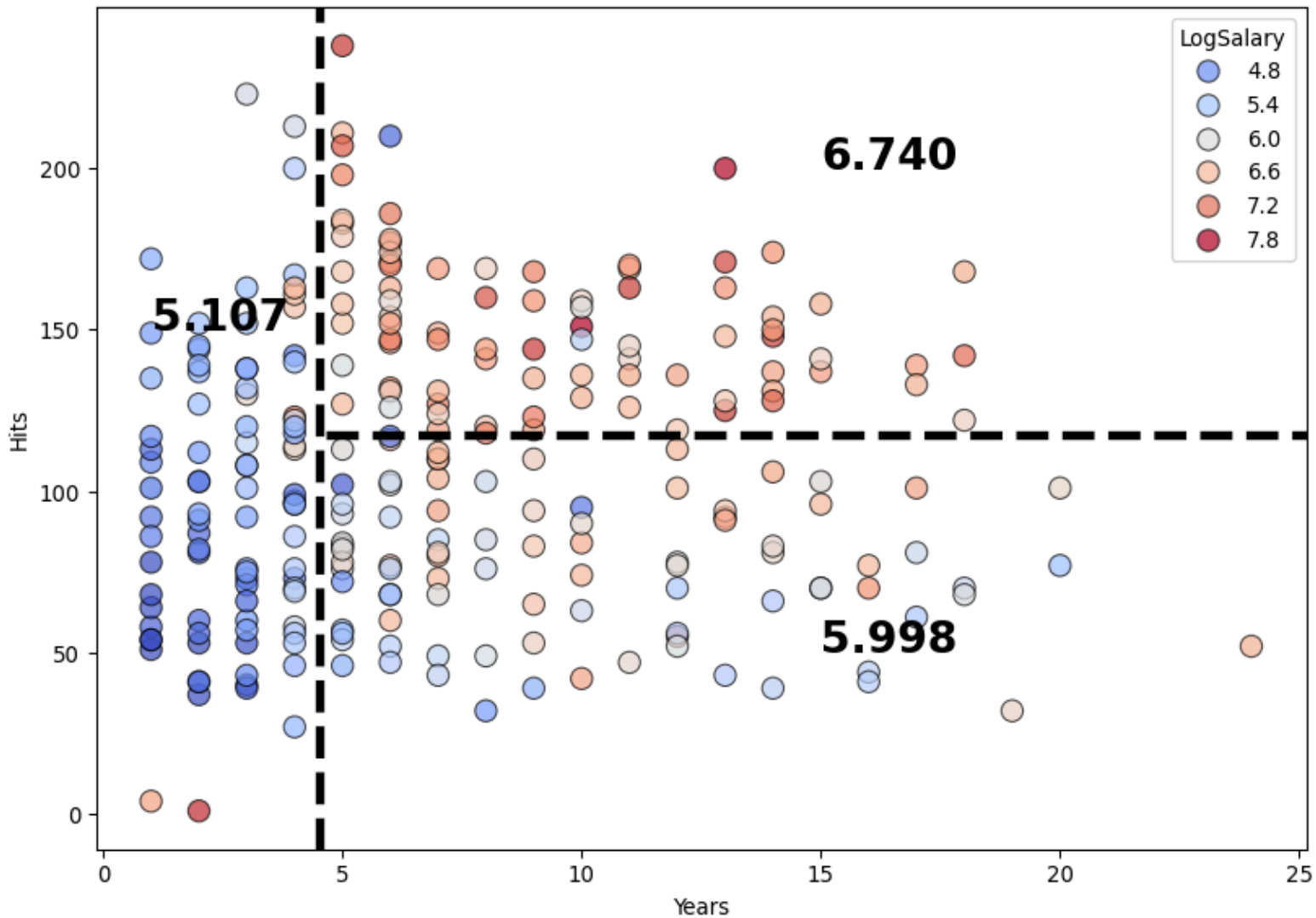


A regression tree with 3 terminal nodes



value = predicted value in the node

Interpretation of the Tree



A piecewise constant predictor of log(Salary), taking only three values

Overview

Tree-based methods are powerful techniques for both regression and classification.

Key idea: Stratify or segment the feature space into simple regions.

- Decision rules can be summarized in a tree $\rightarrow$ **decision trees**
- Simple and useful for interpretation
- Individual trees: not competitive with best supervised learning approaches
- **Solution:** Combine many trees!
  - **Bagging** (Bootstrap Aggregation) | **Random Forests**
  - **Boosting** | **AdaBoost**

Advantages and Disadvantages

Advantages:

- Simple and easy to explain (more than linear models); Can be displayed graphically
- Handle qualitative features naturally; No need for dummy variables, etc.

Disadvantages:

- Lower predictive accuracy (for single trees)
- High variance
- Not robust to small changes in data

Solution: Aggregate many decision trees $\rightarrow$ substantially improved predictions!

1 CART: Classification and Regression Trees

1.1 The Basics of Decision Trees

Regression Trees:

1. Divide the feature space into J distinct non-overlapping regions: $R_1, R_2, \ldots, R_J$
2. For every observation that falls into region R_j, predict the **mean** of response values in R_j

Goal: Find boxes $R_1, \ldots, R_J$ that minimize the Residual Sum of Squares:

$$\text{RSS} = \sum_{j=1}^{J} \sum_{i:x_i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\hat{y}_{R_j}$ is the mean response in region R_j.

Tree Construction Strategy

Problem: Impossible to consider every possible partition of the feature space into J boxes.

Solution: Use a **greedy**, **top-down** recursive binary splitting algorithm.

- **Top-down:** Starts at the root (all observations) and successively splits
- **Greedy:** At each step, makes the **best** split at that particular step (no look-ahead)
- **Binary:** Each split divides a region into exactly two sub-regions

Recursive Binary Splitting Algorithm

At each step, for every feature X_j and cutpoint s:

1. Define two half-planes:

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

2. Find j and s that minimize:

$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2$$

   where $\hat{c}_m = \mathbf{mean}(y_i \mid x_i \in R_m(j, s))$

3. Repeat the process on each resulting region

4. Stop when regions contain fewer than some minimum number of observations (e.g., 5)

1.2 Example: Baseball Salary Data

Predicting **log(Salary)** based on:

- **Years**: number of years in the major leagues
- **Hits**: number of hits in previous year

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree
from ISLP import load_data

# Load Hitters data
Hitters = load_data('Hitters')
Hitters = Hitters.dropna()

# Create log salary
Hitters['LogSalary'] = np.log(Hitters['Salary'])

# Select features
X = Hitters[['Years', 'Hits']]
y = Hitters['LogSalary']
```

Visualizing the Tree

```python
# Fit a simple tree with 3 terminal nodes
tree = DecisionTreeRegressor(max_leaf_nodes=3, random_state=42)
tree.fit(X, y)

# Plot the tree
fig, ax = plt.subplots(figsize=(14, 8))
plot_tree(tree, feature_names=['Years', 'Hits'],
          filled=True, rounded=True, fontsize=12, ax=ax)
plt.tight_layout()
plt.show()
```

Result: Three regions:

- R_1: Years < 4.5
- R_2: Years ≥ 4.5, Hits < 117.5
- R_3: Years ≥ 4.5, Hits ≥ 117.5

Interpretation

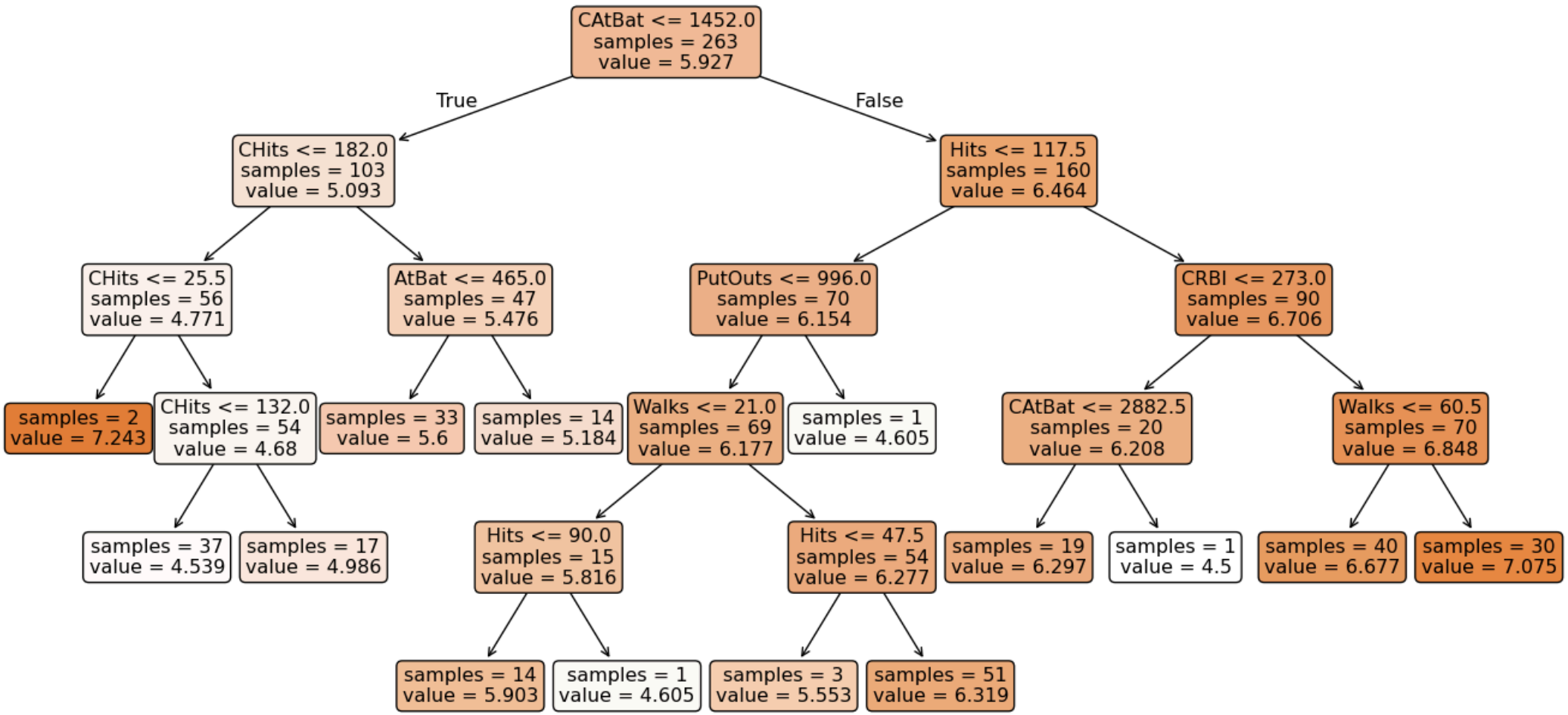
Key findings:

- **Years** is the most important factor
- Less experienced players (Years < 4.5) earn less on average
- Among players with Years $\geq$ 4.5, **Hits** matters:
  - Low Hits (< 117.5) $\rightarrow$ medium salary
  - High Hits ($\geq$ 117.5) $\rightarrow$ high salary

Simple to interpret and explain!

This captures an interaction effect: the effect of Hits depends on Years.

With more features



Where to stop?

The Problem of Overfitting

Phase 1: Build a large tree $\mathcal{T}_0$

- Continue splitting until each leaf has few observations
- Results in a very deep tree
- **Problem:** Overfitting!

Trade-off:

- Tree too deep $\longrightarrow$ overfitting (low bias, high variance)
- Tree too shallow $\longrightarrow$ underfitting (high bias, low variance)

Solution: Tree pruning via cost-complexity pruning

1.3 Cost-Complexity Pruning

For a subtree $T \subseteq \mathcal{T}_0$, define the **cost-complexity criterion**:

$$C_\alpha(T) = \sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T|$$

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

- $|T|$ = number of terminal nodes (leaves)
- N_m = number of observations in leaf m
- $Q_m(T) = \frac{1}{N_m} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2$ (within-leaf variance)
- $\alpha \geq 0$ = complexity parameter (tuning parameter)

Interpretation of the Penalty

The cost-complexity criterion balances:

1. **Fit to the data:** $\sum_{m=1}^{|T|} N_m Q_m(T)$ (smaller = better fit)
2. **Tree complexity:** $\alpha|T|$ (smaller tree = simpler model)

Effect of α:

- $\alpha = 0$: No penalty $\rightarrow$ full tree $\mathcal{T}_0$ (overfitting)
- α small: Large tree allowed (some complexity)
- α large: Penalty on complexity $\rightarrow$ small tree (underfitting)

Goal: Find optimal α via cross-validation

Pruning Algorithm

Step 1: Build the full tree $\mathcal{T}_0$ by recursive binary splitting

Step 2: Obtain a sequence of best subtrees as a function of α:

- Start from $\mathcal{T}_0$
- Iteratively collapse the internal node that produces the smallest increase in $\sum_m N_m Q_m(T)$
- Continue until only the root remains

Result: A sequence of nested subtrees $\mathcal{T}_0 \supset \mathcal{T}_1 \supset \mathcal{T}_2 \supset \cdots$ indexed by α

Each $\mathcal{T}_\alpha$ minimizes $C_\alpha(T)$ for its corresponding α value.

Choosing α by K-fold Cross-Validation

1. For $k = 1, \ldots, K$:
   - Use all data except Fold k to build $\mathcal{T}_0^{-k}$
   - For each α, obtain the pruned $\mathcal{T}_\alpha^{-k}$; Evaluate its validation err. on Fold k: $\mathrm{MSE}_k(\alpha)$
2. Compute average cross-validation error and standard error:

$$\mathrm{CV}(\alpha) = \frac{1}{K}\sum_{k=1}^{K}\mathrm{MSE}_k(\alpha), \qquad \mathrm{SE}(\alpha) = \sqrt{\frac{1}{K-1}\sum_{k=1}^{K}(\mathrm{MSE}_k(\alpha) - \mathrm{CV}(\alpha))^2}$$

3. Set $\alpha_{\min}$ as the value that minimizes $\mathrm{CV}(\alpha)$
4. Set α_{1SE} as the **largest** α such that: $\mathrm{CV}(\alpha) \leq \mathrm{CV}(\alpha_{\min}) + \mathrm{SE}(\alpha_{\min})$

Rationale of the 1-SE rule: Select the simplest model within 1 SE of the minimum $\rightarrow$ more interpretable and often better test performance

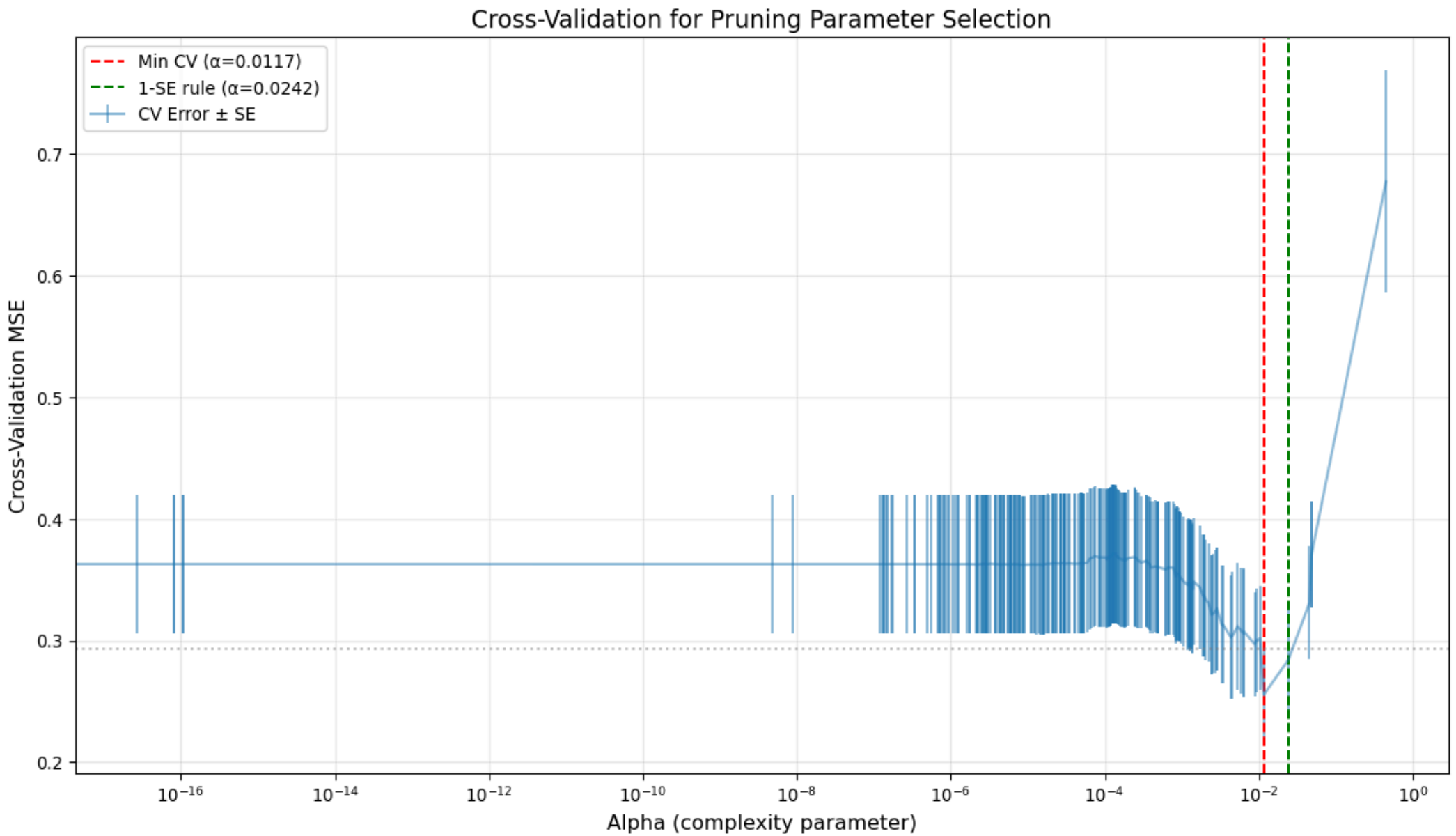
1.4 Example of CV-Based Pruning: Baseball Salary Data

```
1  from sklearn.model_selection import cross_val_score
2
3  X_full = Hitters.drop(columns=['Salary', 'LogSalary', 'League', 'Division', 'NewLeague'
4  Y_full = Hitters['LogSalary']
5
6  # Test different complexity parameters (ccp_alpha)
7  # First, get the path of alphas
8  tree_full = DecisionTreeRegressor(random_state=42)
9  path = tree_full.cost_complexity_pruning_path(X_full, Y_full)
10 alphas = path.ccp_alphas
```

Cross-Validation (continued)

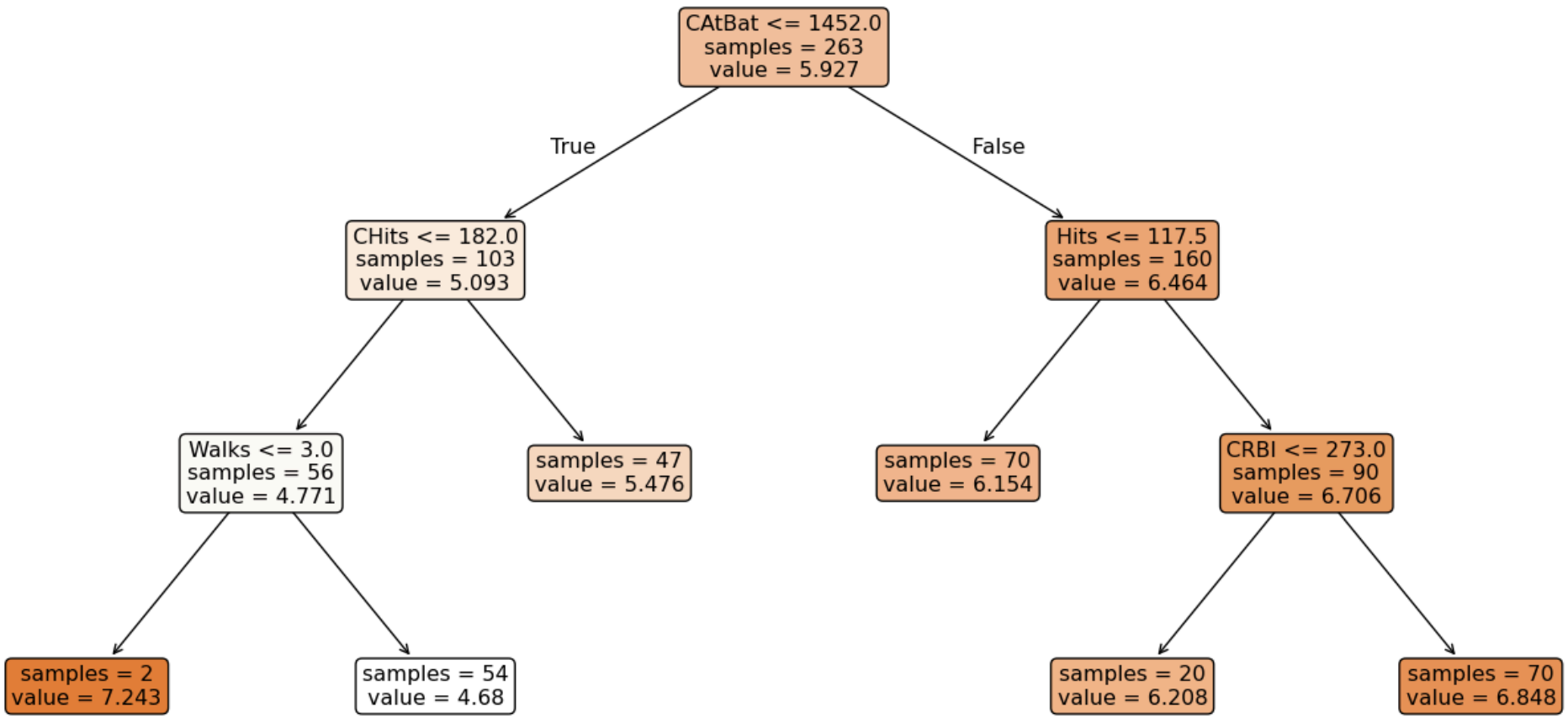
```
# Perform cross-validation for each alpha
cv_scores = []
cv_stds = []

for alpha in alphas:
    tree = DecisionTreeRegressor(ccp_alpha=alpha, random_state=42)
    scores = cross_val_score(tree, X_full, Y_full, cv=10,
                             scoring='neg_mean_squared_error')
    cv_scores.append(-scores.mean())  # Convert to MSE
    cv_stds.append(scores.std())

cv_scores = np.array(cv_scores)
cv_stds = np.array(cv_stds)
```

Plotting CV Results with 1-SE Rule



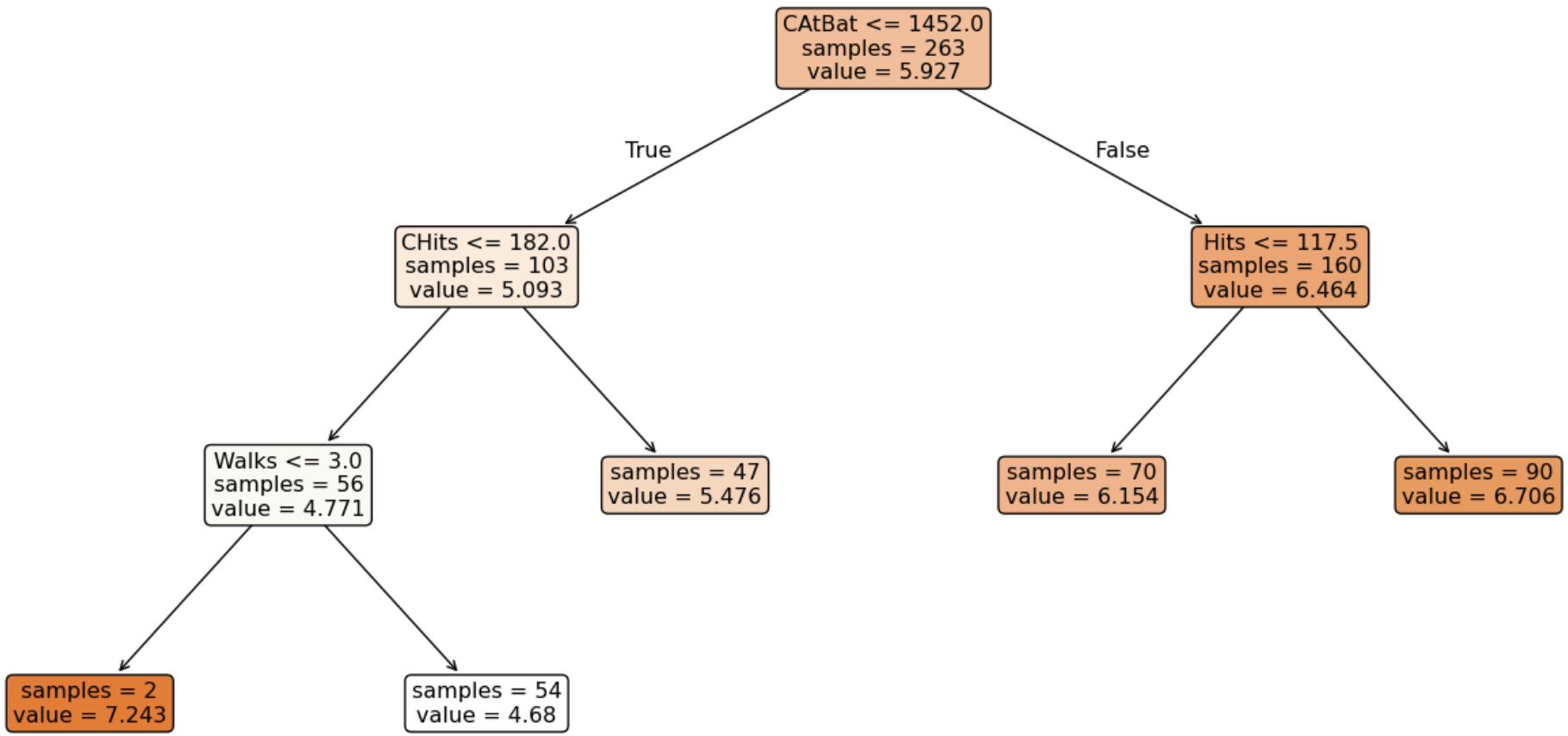
Final Tree with $\alpha_{\min}$

```
Number of leaves: 6
Tree depth: 3
Training R²: 0.7800
```



Final Tree with α_{1SE}

```
Number of leaves: 5
Tree depth: 3
Training R²: 0.7492
```



1.5 Classification Trees

Similar to regression trees, but:

- Predict the **most common class** in each leaf
- Use different splitting criteria

For leaf m representing region R_m with N_m observations:

$$\widehat{p}_{mk} = \frac{1}{N_m} \sum_{i:x_i \in R_m} \mathbb{1}(y_i = k)$$

This is the proportion of class k observations in node m.

Classification rule: Assign observation to class $\widehat{k}(m) = \arg\max_k \widehat{p}_{mk}$

Node Impurity Measures

Three common measures of node **impurity** Q_m:

1. **Classification error rate:** $E_m = 1 - \max_k \widehat{p}_{mk}$

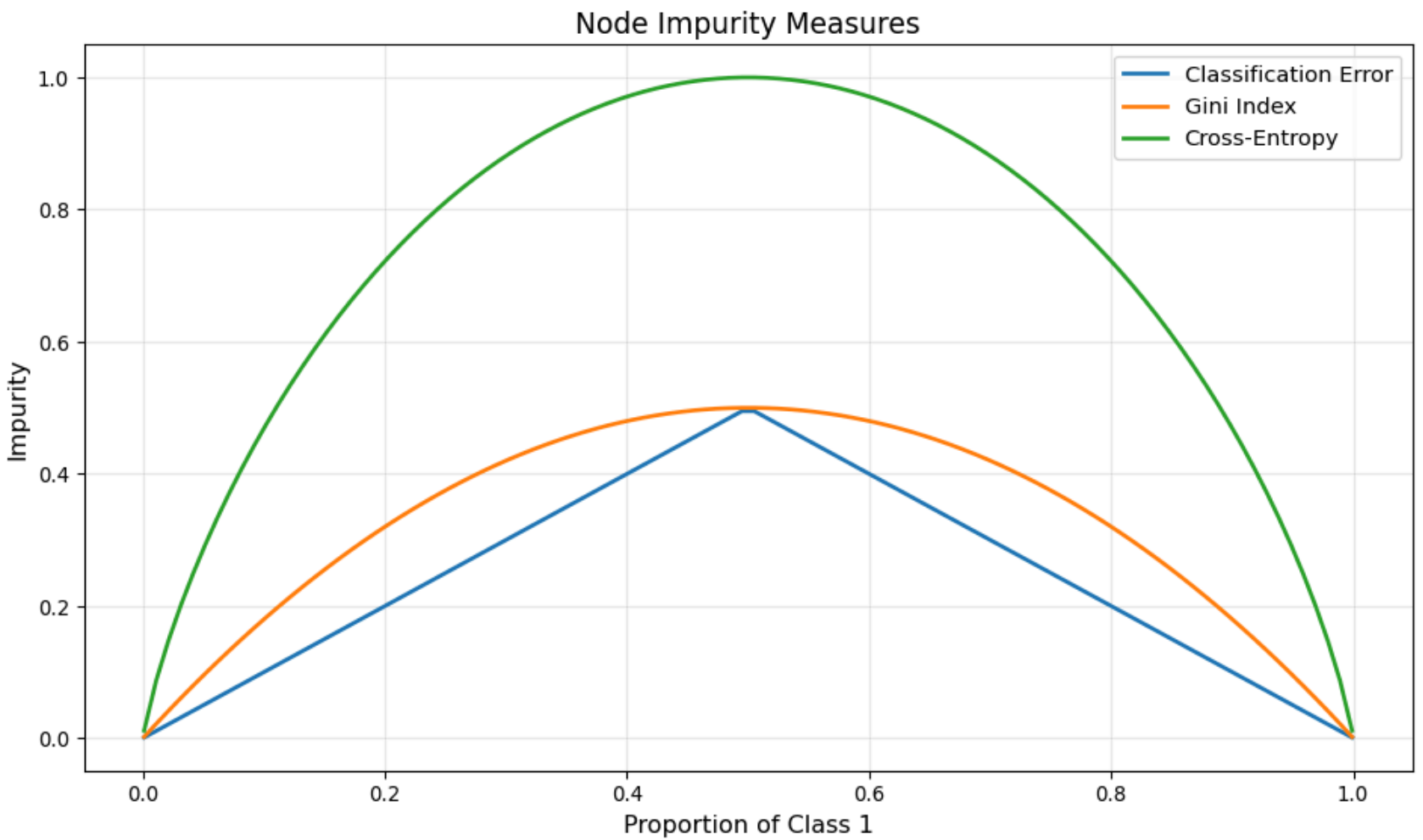
2. **Gini index:** $G_m = \sum_{k=1}^{K} \widehat{p}_{mk}(1 - \widehat{p}_{mk}) = 1 - \sum_{k=1}^{K} \widehat{p}_{mk}^2$

3. **Cross-entropy:** $D_m = -\sum_{k=1}^{K} \widehat{p}_{mk} \log(\widehat{p}_{mk})$

Usage: Gini index or cross-entropy preferred for growing trees (more sensitive to node purity). Any can be used for pruning.

Comparing Impurity Measures

For a two-class problem with $\widehat{p}$ = proportion of class 1:



Observation: Gini and entropy are more sensitive near 0.5 (maximum impurity)

1.6 Python Implementation Summary

```python
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor

# Regression tree
reg_tree = DecisionTreeRegressor(
    criterion='squared_error',  # splitting criterion
    max_depth=None,             # maximum depth
    min_samples_split=2,        # min samples to split
    min_samples_leaf=1,         # min samples in leaf
    ccp_alpha=0.0               # pruning parameter
)

# Classification tree
clf_tree = DecisionTreeClassifier(
    criterion='gini',           # or 'entropy', 'log_loss'
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    ccp_alpha=0.0               # pruning parameter
)
```

2 Bootstrap and Bagging

2.1 What is Bootstrap?

Bootstrap is a resampling method for estimating the sampling distribution of a statistic.

Key idea: Sample with replacement from the original data to create new datasets.

Bootstrap sample:

- Given dataset with n observations: $(x_1, y_1), \ldots, (x_n, y_n)$
- A bootstrap sample $\mathcal{D}^*$ is obtained by randomly sampling n observations **with replacement** from the original data
- Some observations appear multiple times, others not at all

In Supervised Learning

- The training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$ is a table with
  - n rows (observations) and
  - $p + 1$ columns (features + response)
- An observation (x_i, y_i) is a row in the training dataset
- A bootstrap sample $\mathcal{D}^*$ is thus
  - a new training dataset of size as the original
  - rows are drawn with replacement from the original training data

Bootstrap Probability

What proportion of original observations appear in a bootstrap sample?

Probability that a specific observation is **NOT** selected in one draw: $1 - \frac{1}{n}$

Probability that it is **NOT** selected in n draws:

$$\left(1 - \frac{1}{n}\right)^n \xrightarrow{n \to \infty} e^{-1} \approx 0.368$$

Therefore: Each bootstrap sample contains approximately **63.2%** unique observations from the original sample.

The remaining **36.8%** are called **out-of-bag (OOB)** observations.

2.2 Bagging: Bootstrap Aggregating

Bagging = **B**ootstrap **Agg**regat**ing** (Breiman, 1996)

Motivation: Reduce variance of predictions

- Recall: For n i.i.d. observations $Z_1, \ldots, Z_n$ each with variance σ^2:

$$\mathrm{Var}(\bar{Z}) = \frac{\sigma^2}{n}$$

Averaging reduces variance!

Idea: Average predictions from models fitted on different datasets

Problem: We only have one training dataset!

Solution: Create multiple datasets via bootstrap

Bagging Algorithm

Input: Training data $\mathcal{D}$, number of bootstrap samples B

1. **For** $b = 1, \ldots, B$:
   - Generate bootstrap sample $\mathcal{D}^{*b}$ from $\mathcal{D}$
   - Fit model $\widehat{f}^{*b}$ on $\mathcal{D}^{*b}$
2. **Bagged prediction:**
   - **Regression:** $\widehat{f}_{\text{bag}}(x) = \dfrac{1}{B} \displaystyle\sum_{b=1}^{B} \widehat{f}^{*b}(x)$
   - **Classification:** Use **majority vote** $\widehat{f}_{\text{bag}}(x) = \text{majority}\{\widehat{f}^{*1}(x), \ldots, \widehat{f}^{*B}(x)\}$

Why Does Bagging Work?

Variance reduction through averaging:

Suppose we have B independent predictors $\widehat{f}^{*1}, \ldots, \widehat{f}^{*B}$, each with variance σ^2:

$$\mathrm{Var}\left(\frac{1}{B}\sum_{b=1}^{B}\widehat{f}^{*b}\right) = \frac{\sigma^2}{B}$$

In practice: Bootstrap samples are not independent (they overlap)

If pairwise correlation between predictors is ρ: $\mathrm{Var}(\widehat{f}_{\mathrm{bag}}) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$

As $B \to \infty$: $\mathrm{Var}(\widehat{f}_{\mathrm{bag}}) \to \rho\sigma^2$, with $\rho \in [0, 1)$

Lesson: High correlation between trees limits variance reduction!

2.3 Out-of-Bag (OOB) Error Estimation

Key observation: Each bootstrap sample excludes ~37% of observations

Out-of-Bag samples for tree b: Observations NOT in $\mathcal{D}^{*b}$

OOB prediction for observation i:

1. Find all trees where observation i was OOB: $\mathcal{B}_i = \{b : (x_i, y_i) \notin \mathcal{D}^{*b}\}$
2. Compute OOB prediction: $\hat{y}_i^{\text{OOB}} = \frac{1}{|\mathcal{B}_i|} \sum_{b \in \mathcal{B}_i} \hat{f}^{*b}(x_i)$

OOB error estimate:

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

Advantage: Valid error estimate without a separate test set! Similar to leave-one-out cross-validation.

Bagging Trees in Python

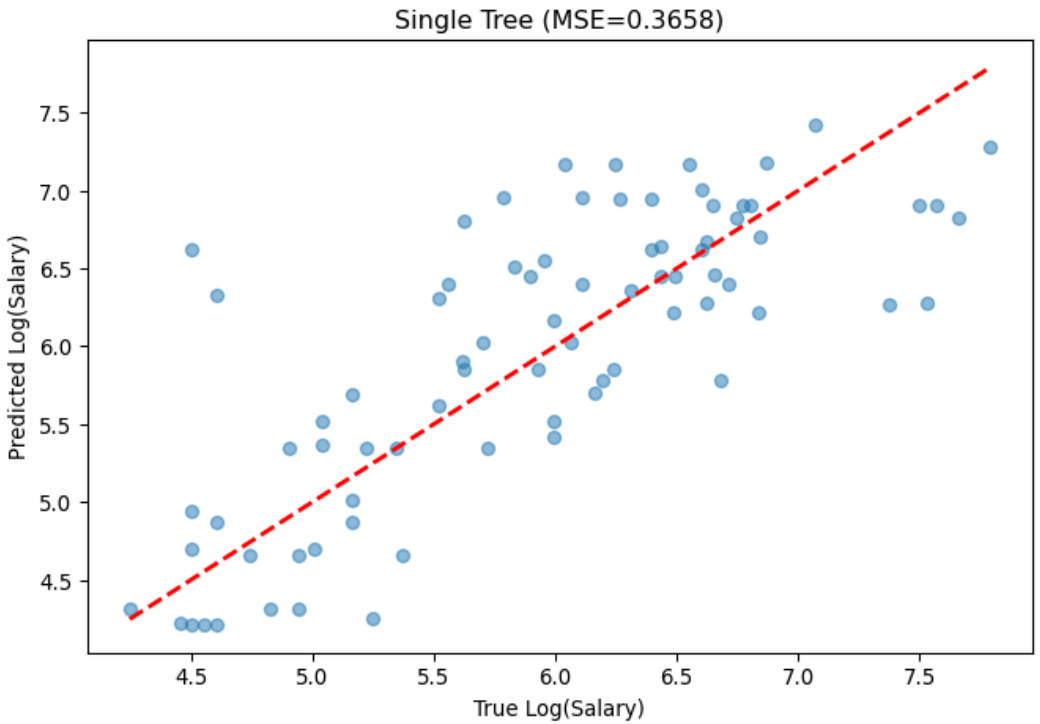
```python
from sklearn.ensemble import BaggingRegressor
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_full, Y_full, test_size=0.3, rand

bag_tree = BaggingRegressor(
    estimator=DecisionTreeRegressor(),
    n_estimators=100,         # number of trees (B)
    max_samples=1.0,          # size of bootstrap sample (100%)
    max_features=1.0,         # use all features
    bootstrap=True,           # sample with replacement
    oob_score=True,           # compute OOB error
    random_state=42,
    n_jobs=2                  # parallel processing
)

bag_tree.fit(X_train, y_train)

print(f"OOB Score (R²): {bag_tree.oob_score_:.4f}")
print(f"Test Score (R²): {bag_tree.score(X_test, y_test):.4f}")
print(f"Train Score (R²): {bag_tree.score(X_train, y_train):.4f}")
```

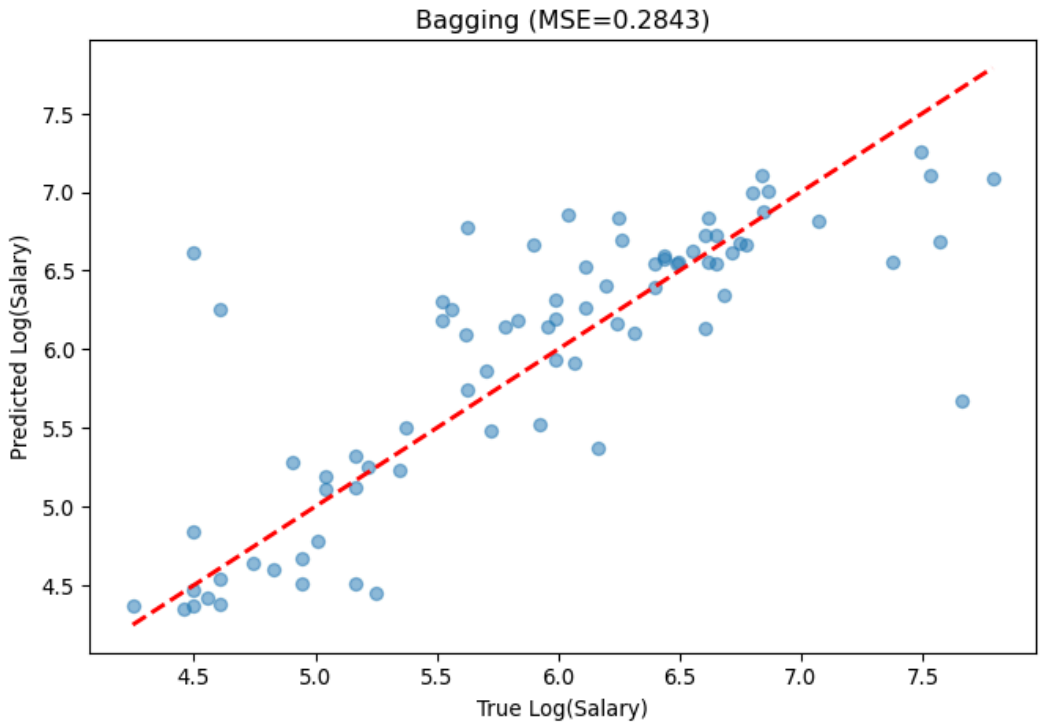
```
OOB Score (R²): 0.7724
Test Score (R²): 0.6352
Train Score (R²): 0.9668
```

Comparing Single Tree vs. Bagging

```python
single_tree = DecisionTreeRegressor(random_state=42)
single_tree.fit(X_train, y_train)

y_pred_single = single_tree.predict(X_test)
y_pred_bag = bag_tree.predict(X_test)

from sklearn.metrics import mean_squared_error
mse_single = mean_squared_error(y_test, y_pred_single)
mse_bag = mean_squared_error(y_test, y_pred_bag)

print(f"Single Tree Test MSE: {mse_single:.4f}")
print(f"Bagging Test MSE: {mse_bag:.4f}")
print(f"Improvement: {(1 - mse_bag/mse_single)*100:.1f}%")
```

```
Single Tree Test MSE: 0.3658
Bagging Test MSE: 0.2843
Improvement: 22.3%
```


Single Tree (MSE=0.3658)
Predicted Log(Salary)
True Log(Salary)
4.5
5.0
5.5
6.0
6.5
7.0
7.5


Bagging (MSE=0.2843)
Predicted Log(Salary)
True Log(Salary)
4.5
5.0
5.5
6.0
6.5
7.0
7.5

3 Random Forests

3.1 The Idea Behind Random Forests

Problem with bagging: Trees are often **correlated**

- Eg., if one feature is very strong, most trees will split on it first
- Trees look similar $\rightarrow$ limited variance reduction
- Recall: $\mathrm{Var}(\hat{f}_{\mathrm{bag}}) \rightarrow \rho\sigma^2$ as $B \rightarrow \infty$

Random Forests solution (Breiman, 2001):

- At each split, consider only a **random subset** of m features (out of p total)
- Forces diversity among trees: Decorrelated trees $\rightarrow$ smaller ρ

> **Warning**
>
> At **each node**, pick **new random set** of m features, independently of other nodes.

- The number of features to consider at each split is a **key hyperparameter**: m (or `max_features` in sklearn)
- **Typical choices:**
  - Classification: $m \approx \sqrt{p}$
  - Regression: $m \approx p/3$
  - Sometimes: $m = \lfloor \log_2(p) \rfloor + 1$
- Can tune m using OOB error

3.2 Random Forest Algorithm

Input:

- Training data $\mathcal{D}$
- Number of trees B
- Number of features per split m

Prediction:

- Regression: $\widehat{f}_{\mathrm{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B}\widehat{\mathcal{T}}_b(x)$
- Classification: majority vote across trees

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

1. Generate bootstrap sample $\mathcal{D}^{*b}$
2. Grow tree $\mathcal{T}_b$ on $\mathcal{D}^{*b}$:
   - **At each split:**
     - Randomly select m features
     - Find best split using only these m features
     - Split the node
   - Continue until minimum node size reached
   - **Do NOT prune the tree**

Why Random Forests Work

Decorrelation effect:

- Bagging: $m = p$ at each split (all features considered)
- Random Forest: $m < p$ at each split (random subset)

When strong predictor exists:

- Bagging: Most trees use it at top $\rightarrow$ trees are correlated
- Random Forest: Strong predictor excluded in $\frac{p-m}{p}$ of splits $\rightarrow$ trees are decorrelated

Example: $p = 100$, one very strong predictor

- Bagging ($m = 100$): All trees split on strong predictor first
- RF ($m = 10$): Only 10% of trees split on strong predictor first $\rightarrow$ more diverse trees

Result: Lower correlation $\rho \rightarrow$ lower variance!

Hyperparameter Tuning: The Role of m

Key hyperparameter: Number of features to consider at each split (m or `max_features`)

Special cases:

- $m = p$: Equivalent to bagging (all features considered)
- $m = 1$: Maximum decorrelation (very random trees)
- $m = \sqrt{p}$: Good default for classification
- $m = p/3$: Good default for regression

Other parameters:

- Number of trees B: Usually 100-500 (more is better but diminishing returns)
- Minimum node size: Default usually works well
- No need to prune: aggregation prevents overfitting

Strategy: Use OOB error to select m (no separate validation set needed!)

3.3 Example: California Housing Data

Target: Median house value in California districts

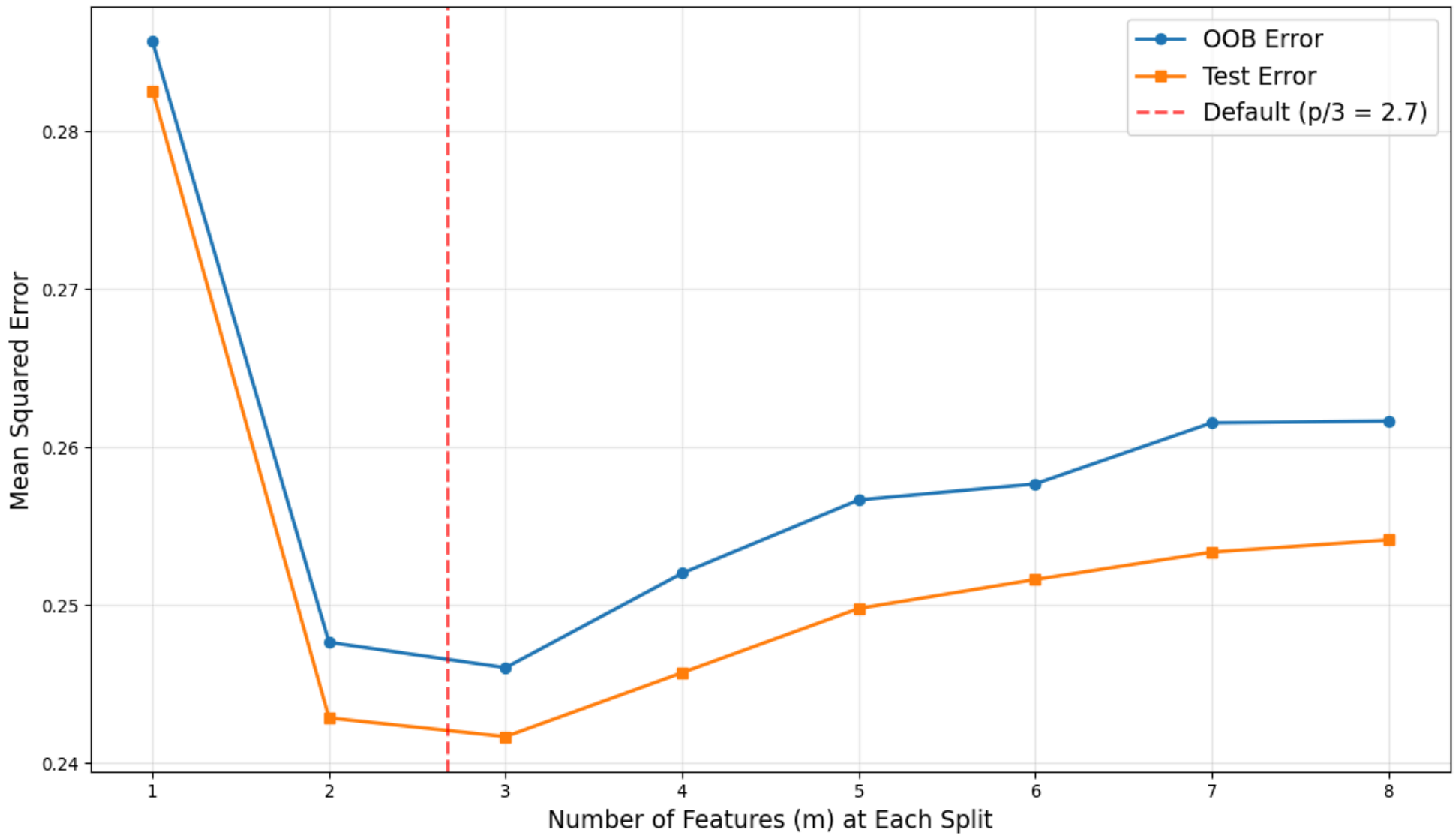
Features: 8 numerical features (e.g., median income, house age, latitude, longitude)

Tuning m on California Housing Data

```
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor

housing = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(housing.data, housing.target, test_

p = X_train.shape[1]  # number of features
m_values = range(1, p+1); oob_errors = []; test_errors = []
for m in m_values:
    rf = RandomForestRegressor(n_estimators=200, max_features=m,
        oob_score=True, random_state=42, n_jobs=2)
    rf.fit(X_train, y_train)

    oob_pred = rf.oob_prediction_
    oob_mse = np.mean((y_train - oob_pred)**2)
    oob_errors.append(oob_mse)

    y_pred = rf.predict(X_test)
    test_mse = np.mean((y_test - y_pred)**2)
    test_errors.append(test_mse)
```

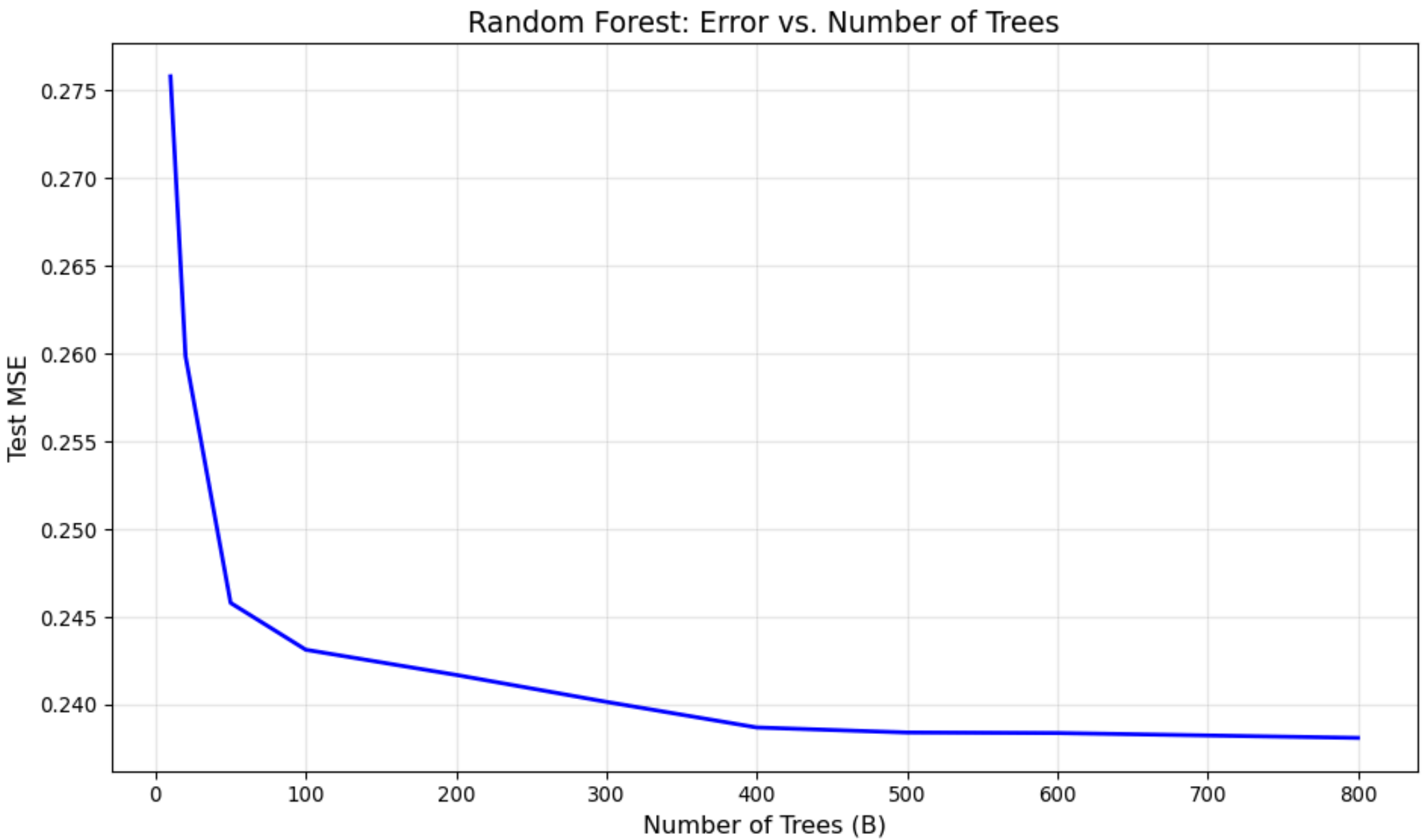
Tuning Plot for m

```
Best m (OOB): 3
Best m (Test): 3
```



Error vs. Number of Trees

```
# Evaluate how error decreases with number of trees
n_trees_range = [10, 20, 50, 100, 200, 300, 400, 500, 600, 800]
errors_by_ntrees = []

for n_trees in n_trees_range:
    rf = RandomForestRegressor(n_estimators=n_trees, max_features=3, oob_score=False,
        random_state=42, n_jobs=2)
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
    test_mse = np.mean((y_test - y_pred)**2)
    errors_by_ntrees.append(test_mse)
```


Random Forest: Error vs. Number of Trees
Test MSE
0.275
0.270
0.265
0.260
0.255
0.250
0.245
0.240
0
100
200
300
400
500
600
700
800
Number of Trees (B)

3.4 Variable Importance

How to measure feature importance in Random Forests?

For Regression:

- For each tree and each feature X_j:
  - Record total decrease in RSS from splits on feature X_j
  - $\text{Importance}_j^{(b)} = \sum_{t:\text{split on } X_j} \text{RSS decrease at node } t$
- Average across all trees: $\text{Importance}_j = \frac{1}{B} \sum_{b=1}^{B} \text{Importance}_j^{(b)}$

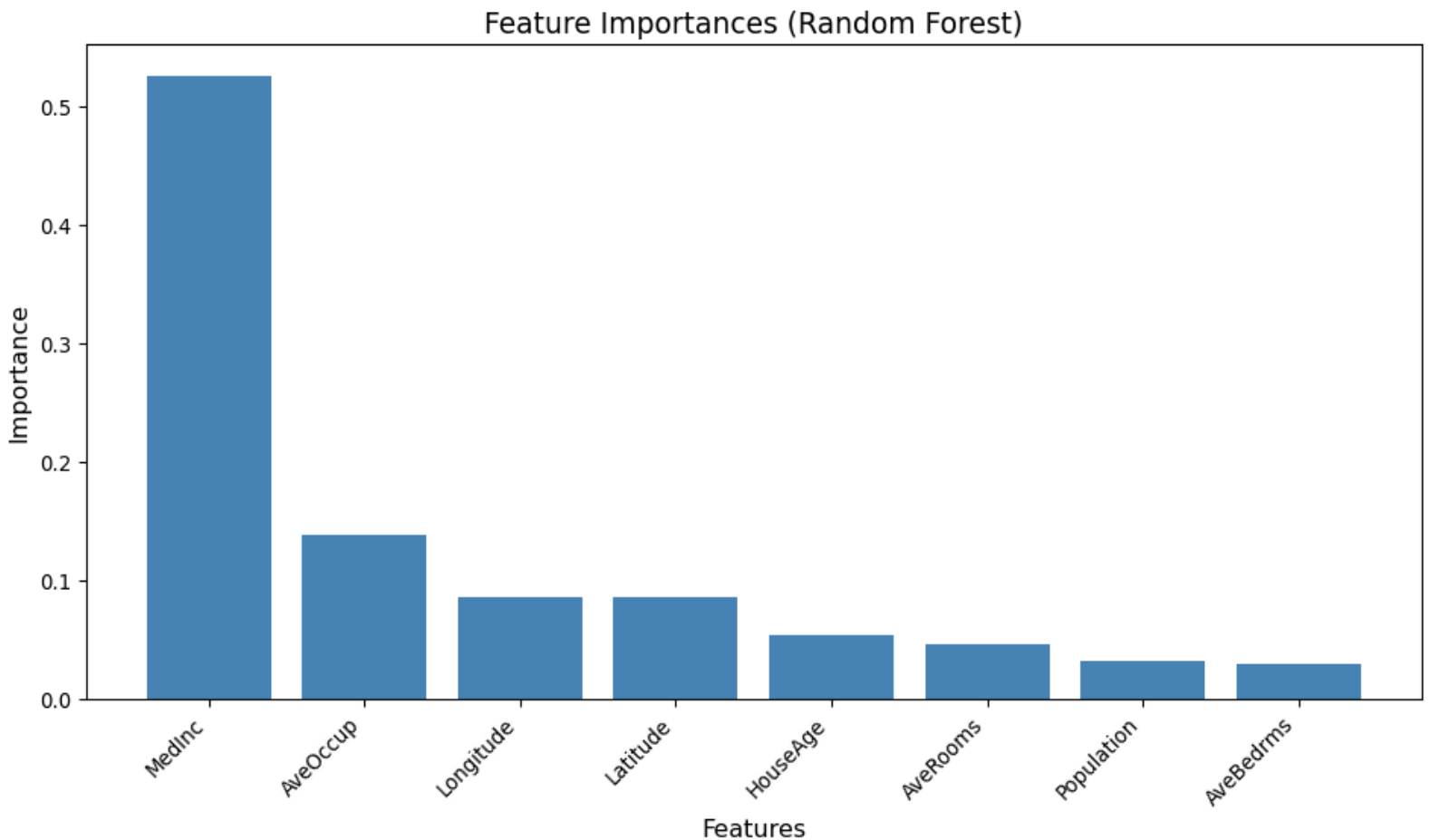
For Classification:

- Record total decrease in Gini index from splits on each feature
- Average across all trees

In scikit-learn: `feature_importances_` attribute

Variable Importance Example

```
Top 5 most important features:
1. MedInc: 0.5254
2. AveOccup: 0.1389
3. Longitude: 0.0867
4. Latitude: 0.0865
5. HouseAge: 0.0547
```



3.5 Random Forests in Practice

Advantages:

- Excellent predictive performance out-of-the-box
- Handles high-dimensional data well
- Provides feature importance estimates
- OOB error provides built-in validation
- Robust to overfitting
- Works well with default parameters
- Can handle missing values (with appropriate methods)
- Can capture complex non-linear relationships

Disadvantages:

- Less interpretable than single tree
- Can be slow on very large datasets
- Requires more memory than single tree
- Not great for extrapolation (predictions beyond training range)

When to use:

- Default choice for many ML tasks
- When predictive accuracy is paramount
- When you have enough data and computational resources

4 Boosting

4.1 The Boosting Idea

Bagging/Random Forests: Build trees **independently** on bootstrap samples, then average

Boosting: Build trees **sequentially**, each improving on the previous ones

Key principles:

1. **Learn slowly:** Each tree corrects errors of the current ensemble
2. **Weak learners:** Use shallow trees (low variance, high bias)
3. **Sequential:** Each new tree focuses on mistakes of previous trees
4. **Additive:** New trees are added to the ensemble

Result: Strong learner from combination of many weak learners!

Key insight: Boosting is a form of **gradient descent** in function space

Boosting Algorithm for Regression

Input:

- Training data $\{(x_i, y_i)\}_{i=1}^n$
- Number of trees B
- Learning rate λ (shrinkage)
- Tree depth d

Output: Boosted model

$$\widehat{f}(x) = \bar{y} + \lambda \sum_{b=1}^{B} \widehat{f}^b(x)$$

Initialize:

$\widehat{f}(x) = \bar{y}$ and residuals $r_i = y_i - \bar{y}$ for all i

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

1. Fit a tree $\widehat{f}^b$:
   - with d splits
   - to predict residuals r_i using features x_i
2. Update residuals:

$$r_i \leftarrow r_i - \lambda \widehat{f}^b(x_i)$$

Why Does Boosting Work?

Intuition:

- Start with simple model (high bias, low variance): $\widehat{f}(x) = \bar{y}$
- Each iteration:
  - Identify where current model performs poorly (residuals)
  - Fit new tree to reduce these residuals
  - Add correction slowly with learning rate λ

Unlike bagging:

- Doesn't reduce variance by averaging independent predictions
- Reduces **bias** by sequentially improving fit
- **Risk of overfitting** if too many iterations

Boosting as Gradient Descent

Boosting can be viewed as gradient descent with fixed step size in function space

Consider minimizing the loss function:

$$L(\widehat{f}) = \sum_{i=1}^{n} \ell(y_i, \widehat{f}(x_i))$$

For squared loss $\ell(y, \widehat{f}(x)) = \frac{1}{2}(y - \widehat{f}(x))^2$:

- **Gradient** at iteration b: $\dfrac{\partial L}{\partial \widehat{f}(x_i)} = -(y_i - \widehat{f}(x_i)) = -r_i$
- **Gradient descent update:** $\widehat{f}(x) \leftarrow \widehat{f}(x) - \lambda \times (-r_i) = \widehat{f}(x) + \lambda r_i$
- **Boosting update:** Fit tree $\widehat{f}^b$ to residuals r_i, then $\widehat{f}(x) \leftarrow \widehat{f}(x) + \lambda \widehat{f}^b(x)$

Interpretation: Each tree approximates the negative gradient, and λ is the step size!

What About Classification?

Generalization: This extends to any differentiable loss (classification, ranking, etc.) and leads to other **Gradient Boosting Machines**

Example: For classification ($y \in \{0, 1\}$), with the cross-entropy loss

$$\ell(y, \widehat{f}(x)) = -[y \log \sigma(\widehat{f}(x)) + (1 - y) \log(1 - \sigma(\widehat{f}(x)))]$$

where $\sigma(z) = 1 \Big/ (1 + e^{-z})$ is the logistic function, the residuals are:

$$r_i = -\frac{\partial L}{\partial \widehat{f}(x_i)} = y_i - \sigma(\widehat{f}(x_i))$$

Unlike regression, the second derivative in not constant:

$$\frac{\partial^2 L}{\partial \widehat{f}(x_i)^2} = \sigma(\widehat{f}(x_i))\Big(1 - \sigma(\widehat{f}(x_i))\Big)$$

The leaf values are then updated as:

$$\text{leaf value} = \frac{\sum_{i \in \text{leaf}} \Big(y_i - \sigma(\widehat{f}(x_i))\Big)}{\sum_{i \in \text{leaf}} \sigma(\widehat{f}(x_i))\Big(1 - \sigma(\widehat{f}(x_i))\Big)}$$

4.2 Boosting Hyperparameters

Three key tuning parameters:

1. **Number of trees B:**
   - Unlike bagging/RF, boosting **can overfit** if B too large
   - Use cross-validation to select
   - Early stopping: monitor validation error
   - Typical: 100-5000 trees
2. **Learning rate λ (shrinkage):**
   - Controls how fast boosting learns
   - Smaller $\lambda \rightarrow$ slower learning $\rightarrow$ needs larger B
   - Typical: 0.01 or 0.001 (sometimes 0.1)
   - Trade-off: small λ + large B vs. large λ + small B

3. **Tree depth d (interaction depth):**
   - Controls complexity of each tree
   - $d = 1$: "stumps" (no interactions, additive model)
   - $d = 2$: allows 2-way interactions
   - Typical: $d = 1$ to $d = 6$

Other parameters to consider:

4. **Subsampling fraction:**
   - Use only a fraction of training data for each tree
   - Typical: 0.5-0.8
   - Reduces variance, speeds up computation
5. **Column sampling:**
   - Random subset of features for each tree (like Random Forest)
   - Further decorrelates trees

Hyperparameter interaction

- Small λ (learning rate) requires large B (number of trees)
- Larger d (tree depth) requires smaller λ to prevent overfitting
- Always use cross-validation for final selection

Warning

Boosting is sensitive to hyperparameters! **Careful tuning** is essential.

Since the number of hyperparameters is large, consider using automated hyperparameter optimization (e.g., grid search, random search, **Bayesian optimization**).

4.3 XGBoost: Extreme Gradient Boosting

XGBoost (Chen & Guestrin, 2016) is a highly optimized implementation of gradient boosting

Dominates ML competitions (Kaggle, etc.) apart from deep learning

Key improvements over basic boosting:

- **Regularization:** L1 and L2 penalties on leaf weights $\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$
- **Second-order gradient:** Uses Hessian for better approximation
- **Efficient handling of sparse data**
- **Parallel processing:** Fast tree construction
- **Built-in cross-validation**
- **Early stopping:** Automatically stops when no improvement
- **Missing value handling:** Learns best direction for missing values

4.4 XGBoost in Python

```python
import xgboost as xgb
from sklearn.metrics import mean_squared_error

# Create DMatrix (XGBoost data structure)
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set parameters
params = {
    'objective': 'reg:squarederror',  # for regression
    'max_depth': 4,                    # tree depth
    'learning_rate': 0.1,              # eta (λ)
    'subsample': 0.8,                  # row sampling
    'colsample_bytree': 0.8,           # column sampling
    'seed': 42
}
```

XGBoost Training with Cross-Validation

```python
# Cross-validation to find optimal number of rounds
cv_results = xgb.cv(
    params,
    dtrain,
    num_boost_round=1000,
    nfold=5,
    metrics='rmse',
    early_stopping_rounds=10,
    seed=42,
    verbose_eval=50
)

print(f"Best iteration: {len(cv_results)}")
print(f"Best CV RMSE: {cv_results['test-rmse-mean'].min():.4f}")
```

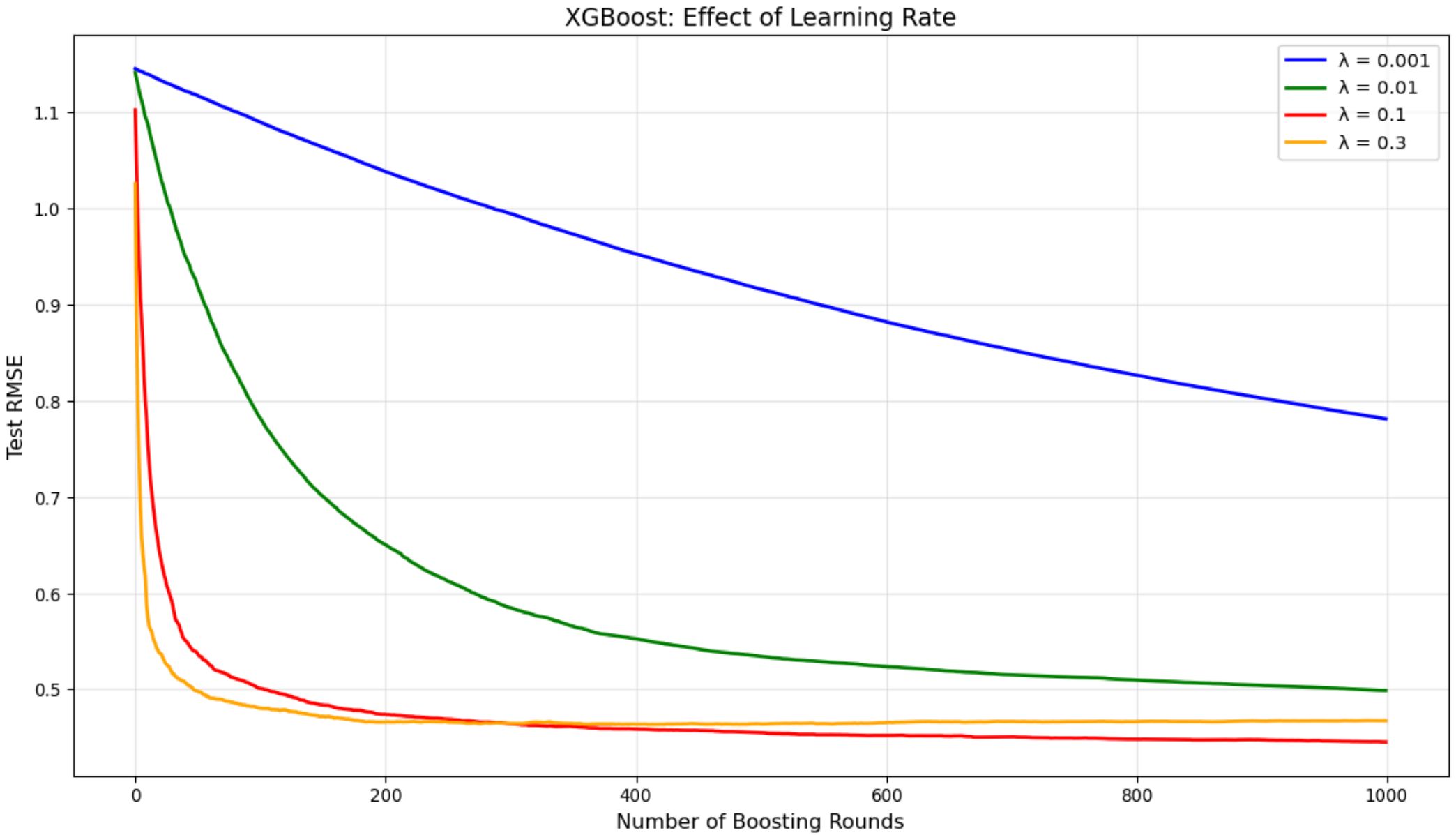
```
[0] train-rmse:1.11363+0.00398  test-rmse:1.11493+0.01582
[50]    train-rmse:0.50842+0.00232  test-rmse:0.53778+0.01226
[100]   train-rmse:0.46059+0.00359  test-rmse:0.50839+0.00967
[150]   train-rmse:0.43174+0.00356  test-rmse:0.49542+0.01040
[200]   train-rmse:0.41072+0.00355  test-rmse:0.48813+0.00956
[250]   train-rmse:0.39312+0.00277  test-rmse:0.48243+0.01003
[300]   train-rmse:0.37838+0.00235  test-rmse:0.47831+0.01077
[350]   train-rmse:0.36597+0.00253  test-rmse:0.47666+0.01069
[400]   train-rmse:0.35441+0.00229  test-rmse:0.47428+0.01041
[450]   train-rmse:0.34363+0.00224  test-rmse:0.47208+0.01015
[500]   train-rmse:0.33391+0.00220  test-rmse:0.47052+0.01025
[539]   train-rmse:0.32689+0.00192  test-rmse:0.47006+0.01013
Best iteration: 530
Best CV RMSE: 0.4700
```

Final Model Training and Evaluation

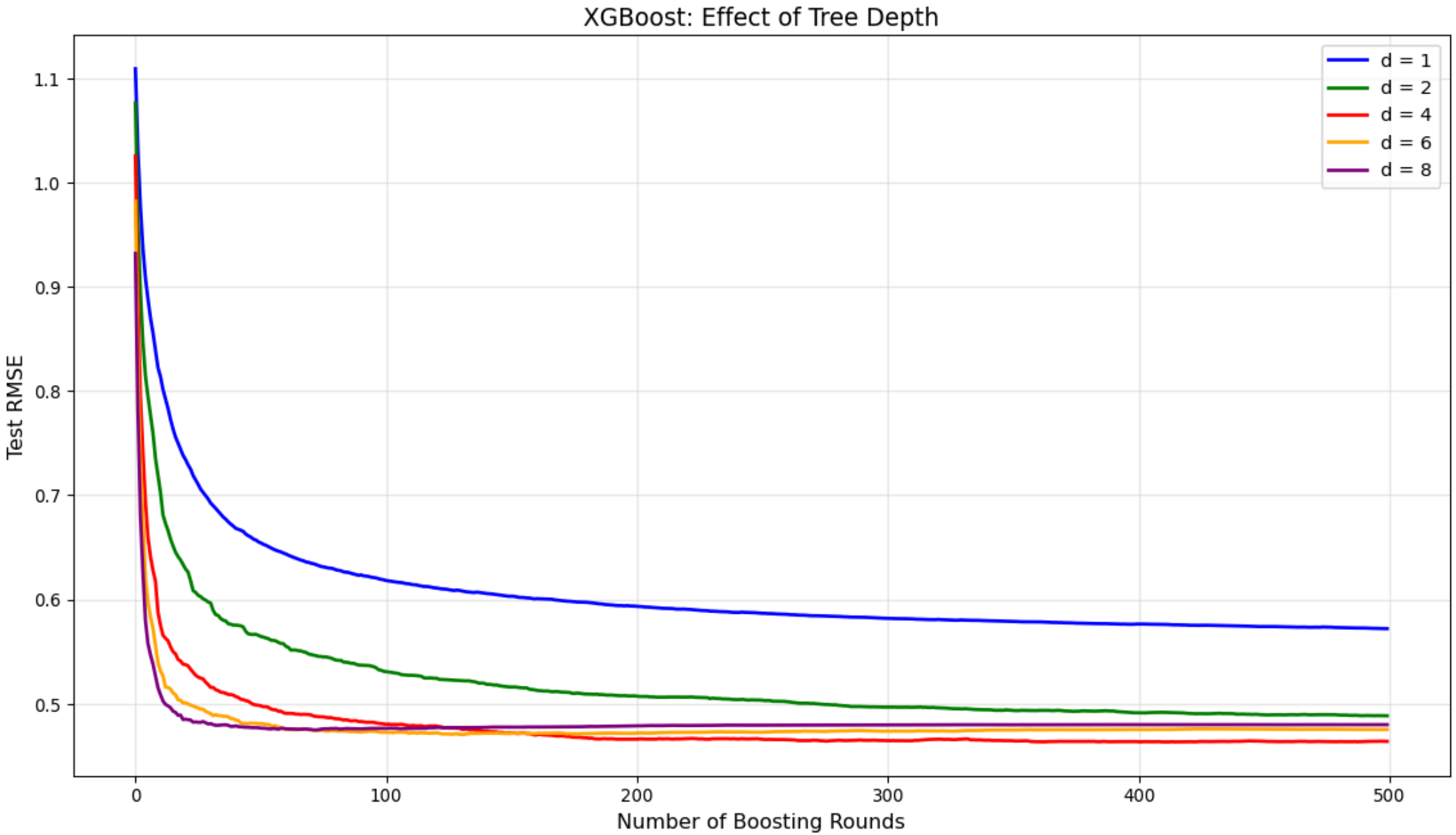
```python
# Train final model
num_boost_round = len(cv_results)
model = xgb.train(
    params,
    dtrain,
    num_boost_round=num_boost_round,
    evals=[(dtrain, 'train'), (dtest, 'test')],
    verbose_eval=50
)

# Predictions
y_pred = model.predict(dtest)
test_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"\nTest RMSE: {test_rmse:.4f}")
```

```
[0] train-rmse:1.11371  test-rmse:1.10233
[50]    train-rmse:0.51224  test-rmse:0.53654
[100]   train-rmse:0.46399  test-rmse:0.50077
[150]   train-rmse:0.43639  test-rmse:0.48359
[200]   train-rmse:0.41678  test-rmse:0.47398
[250]   train-rmse:0.40209  test-rmse:0.46882
[300]   train-rmse:0.38882  test-rmse:0.46412
[350]   train-rmse:0.37683  test-rmse:0.46126
[400]   train-rmse:0.36617  test-rmse:0.45868
[450]   train-rmse:0.35626  test-rmse:0.45707
[500]   train-rmse:0.34770  test-rmse:0.45488
[529]   train-rmse:0.34228  test-rmse:0.45318

Test RMSE: 0.4532
```

Tuning Learning Rate and Number of Trees

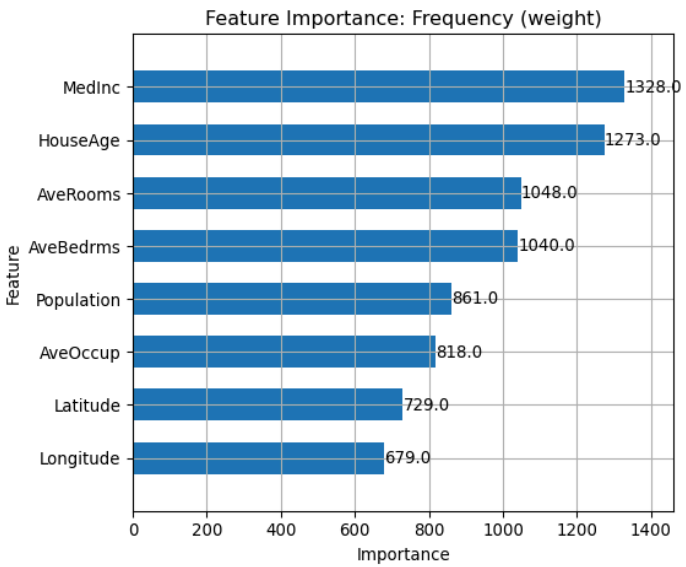


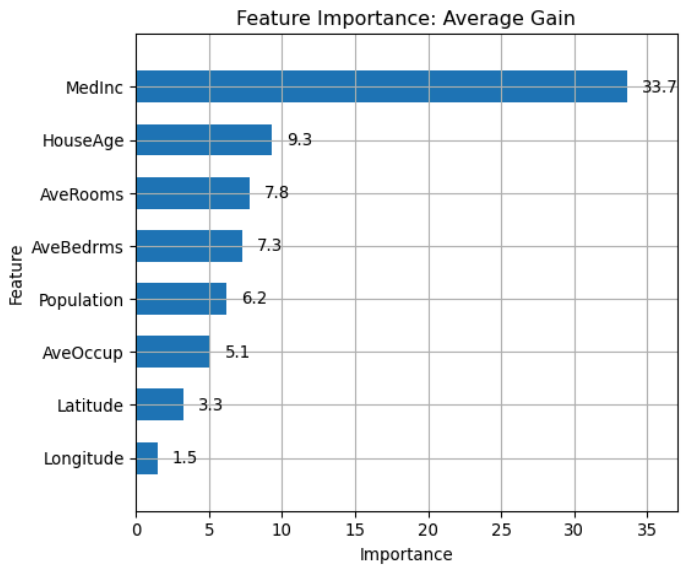
Tuning Tree Depth

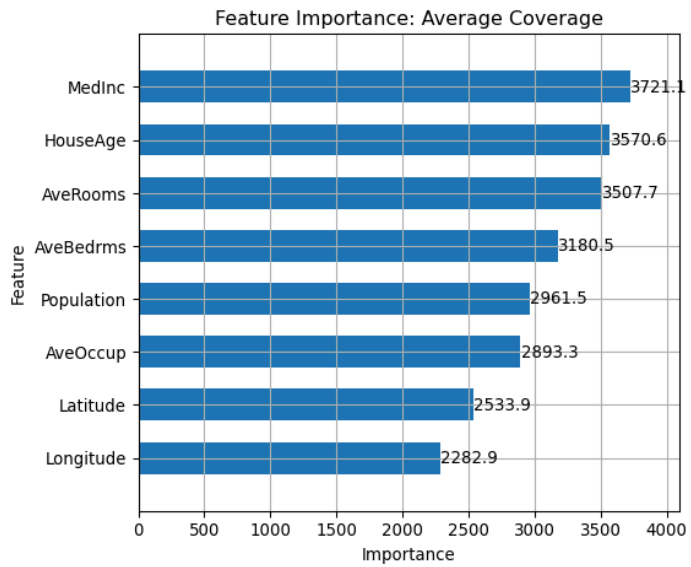


Feature Importance in XGBoost

```
Top 5 features by gain:
1. MedInc: 33.67
2. AveOccup: 9.29
3. Longitude: 7.76
4. Latitude: 7.30
5. AveRooms: 6.20
```







Comparing Methods on California Housing

Method	Test RMSE	Training Time (s)
Single Tree	0.725354	0.118851
Bagging	0.506838	7.531850
Random Forest	0.507132	7.448034
Untuned GradientBoosting	0.536995	2.614154
Untuned XGBoost	0.460242	0.224922

5 Summary and Comparison

5.1 Summary of Tree-Based Methods

Method	Key Idea	Variance	Bias	Interpretability	Speed
Single Tree	Recursive splitting	High	Low-Med	★★★★★	★★★★★
Bagging	Average independent trees	Reduced	Low-Med	★★	★★★
Random Forest	Average decorrelated trees	Lower	Low-Med	★★	★★★
Boosting	Sequential weak learners	Low	Reduced	★★	★

5.2 When to Use Each Method

Single Decision Tree:

- Need interpretability
- Quick baseline model
- Small datasets
- Domain requires explainability

Bagging:

- Reduce variance of high-variance models
- Stable predictions needed
- Computational resources available

Random Forest:

- **Default choice for many problems**
- High-dimensional data
- When accuracy is paramount
- Need feature importance
- Don’t need interpretability

Gradient Boosting / XGBoost:

- Maximum predictive performance
- Structured/tabular data
- ML competitions
- Willing to tune hyperparameters carefully
- Have computational resources

5.3 Practical Recommendations

Hyperparameter Tuning Priority:

1. **Random Forest:** Usually works well with defaults
   - Main tuning: `max_features` ($\sqrt{p}$ or $p/3$)
   - Less critical: `n_estimators` (100-500), `max_depth`, `min_samples_split`
2. **XGBoost:** More tuning needed for optimal performance (Early stopping recommended)
   - Critical: `learning_rate`, `max_depth`, `n_estimators`
   - Important: `subsample`, `colsample_bytree`

General advice:

- Start with Random Forest (good defaults, fast to run)
- Try XGBoost for better performance (requires more tuning)
- Use cross-validation for all methods

5.4 Computational Considerations

Training time (fastest to slowest):

1. Single tree (seconds)
2. Random Forest (minutes, parallelizable)
3. Bagging (minutes, parallelizable)
4. Boosting (minutes to hours, sequential)
5. XGBoost (optimized, but may be slower than RF because of tuning)

Prediction time:

- All ensemble methods slower than single tree
- Roughly proportional to number of trees

Memory:

- Ensemble methods require storing multiple trees
- Random Forest: typically requires most memory
- Boosting: can be more memory-efficient (smaller trees)

Scalability:

- Random Forest: Scales well (easily parallelized)
- XGBoost: Optimized for large datasets, good scalability

5.5 Key Takeaways

1. **Single trees are interpretable but unstable**
    - High variance, prone to overfitting
    - Use pruning (with 1-SE rule) to control complexity
    - Good for understanding relationships
2. **Ensemble methods combine multiple trees**
    - Trade interpretability for accuracy
    - Different strategies: parallel (bagging/RF) vs. sequential (boosting)
3. **Random Forests are robust and reliable**
    - Usually first choice for practitioners
    - Good performance with minimal tuning
    - Works well out-of-the-box

4. **Boosting can achieve highest accuracy**
    - Requires more careful tuning
    - XGBoost is state-of-the-art for tabular data
    - Risk of overfitting—use early stopping
5. **Always validate properly**
    - OOB error for bagging/RF
    - Cross-validation for parameter selection
    - Early stopping for boosting
    - Always check on held-out test set

5.6 References and Further Reading

Books:

- Hastie, Tibshirani, Friedman (2009). *The Elements of Statistical Learning*
- James, Witten, Hastie, Tibshirani (2021). *An Introduction to Statistical Learning* (2nd ed.)

Key Papers:

- Breiman (1996). “Bagging Predictors.” *Machine Learning*
- Breiman (2001). “Random Forests.” *Machine Learning*
- Friedman (2001). “Greedy Function Approximation: A Gradient Boosting Machine”
- Chen & Guestrin (2016). “XGBoost: A Scalable Tree Boosting System.” *KDD*

Python Libraries:

- `scikit-learn`: `sklearn.tree`, `sklearn.ensemble`
- `xgboost`: Extreme Gradient Boosting

5.7 Thank You!

Questions?

Key Resources:

- ISLR book: https://www.statlearning.com/
- scikit-learn: https://scikit-learn.org/
- XGBoost: https://xgboost.readthedocs.io/

Contact: pierre.pudlo@univ-amu.fr