Lec 7: Boosting and Random Forest

Xin Xing

Boosting Methods: Introduction

Key Idea:

- Combine outputs of many "weak" classifiers to produce a powerful "committee"
- Sequential learning: each classifier focuses on mistakes of previous ones
- One of the most powerful learning ideas in machine learning

Weak Classifier:

- Error rate only slightly better than random guessing
- Example: "stumps" (two-node decision trees)

Boosting vs. Bagging:

- Bagging: parallel, independent training on bootstrap samples
- Boosting: sequential, adaptive training on weighted samples
- Boosting is fundamentally different from committee-based approaches

AdaBoost (Adaptive Boosting)

Setup:

- Two-class problem: $Y \in \{-1, 1\}$
- Weak classifiers: $G_m(x) \in \{-1, 1\}$, $m = 1, 2, \ldots, M$

Final Classifier:

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

- α_m: weights computed by boosting algorithm
- Give higher influence to more accurate classifiers

Training Process:

- Apply weights $w_1, w_2, \ldots, w_N$ to training observations
- Initially: $w_i = 1/N$ (equal weights)
- Adaptively modify weights at each iteration
- Increase weights for misclassified observations

The AdaBoost.M1 Algorithm

1. **Initialize:** $w_i = 1/N$ for $i = 1, 2, \ldots, N$

2. **For** $m = 1$ to M:
   (a) Fit classifier $G_m(x)$ using weights w_i
   (b) Compute weighted error:

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i \, \mathbf{1}[y_i \neq G_m(x_i)]}{\sum_{i=1}^{N} w_i}$$

   (c) Compute classifier weight:

$$\alpha_m = \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$

   (d) Update observation weights:

$$w_i \leftarrow w_i \cdot \exp\left[\alpha_m \cdot \mathbf{1}[y_i \neq G_m(x_i)]\right]$$

3. **Output:** $G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$

How AdaBoost Works

Weight Update Mechanism:

- Misclassified observations: weights multiplied by $\exp(\alpha_m)$
- Correctly classified: weights stay same (relative decrease)
- α_m large when err_m small $\Rightarrow$ good classifier gets more weight

Sequential Focus:

- As iterations proceed, difficult observations get increasing influence
- Each classifier forced to concentrate on previous mistakes
- Later classifiers focus on "hard" examples

Classifier Weighting:

$$\alpha_m = \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$

- $\text{err}_m = 0.5$ (random): $\alpha_m = 0$ (no contribution)
- $\text{err}_m < 0.5$: $\alpha_m > 0$ (positive weight)
- $\text{err}_m \to 0$: $\alpha_m \to \infty$ (very high weight)

Example: Boosting Stumps

Simulation Setup:

- Features: $X_1, \ldots, X_{10} \sim \mathcal{N}(0, 1)$ independent
- Target: $Y = 1$ if $\sum_{j=1}^{10} X_j^2 > \chi^2_{10}(0.5)$, else $Y = -1$
- Training: 2000 cases; Test: 10,000 cases
- Weak classifier: stump (two-node tree)

Results:

- Single stump: 45.8% test error (barely better than random 50%)
- After 400 boosting iterations: 5.8% test error
- Improvement: factor of $\sim$8 reduction in error
- Outperforms single large tree (24.7% error)

Quote: "Best off-the-shelf classifier in the world" (Breiman, 1996)

Training vs. Test Error

Observation from experiments:

- Training misclassification error drops to zero (around iteration 250)
- Exponential loss continues to decrease
- Test error keeps improving after training error reaches zero

Interpretation:

- AdaBoost is NOT optimising training misclassification rate
- Exponential loss more sensitive to changes in class probabilities
- Continues to improve confidence of predictions
- Focus on getting class probabilities right, not just classification

Key Insight:

- Even after perfect training classification, algorithm improves
- Increases *margins* (confidence) of predictions
- This helps generalisation to test data

Summary: Boosting Methods

Key Concepts:

- Sequential combination of weak classifiers
- Adaptive reweighting focuses on difficult examples
- Equivalent to forward stagewise additive modelling
- Minimises exponential loss function

AdaBoost Algorithm:

- Computationally efficient
- Works well with decision trees as base learners
- Often called "best off-the-shelf classifier"
- Effective for data mining applications

Theoretical Properties:

- Estimates log-odds of class probabilities
- Continues improving even after zero training error
- Increases prediction margins (confidence)

Gradient Boosting

Forward Stagewise Modelling · GBM · Shrinkage · XGBoost

Gradient Boosting: Motivation

Limitation of AdaBoost:

- AdaBoost minimises exponential loss — binary classification only
- Sensitive to outliers (exponential loss grows unboundedly)
- No built-in regularisation

Gradient Boosting (Friedman, 2001):

- Generalises AdaBoost to *any differentiable loss function*
- Regression, classification, ranking, survival, ...
- Reinterprets boosting as **gradient descent in function space**

Key idea:

- Instead of reweighting samples (AdaBoost), fit each new base learner to the **negative gradient** of the loss
- Negative gradient = direction of steepest descent
- Works for any smooth loss $L(y, f(x))$

Forward Stagewise Additive Modelling

General additive model:

$$f(x) = \sum_{m=1}^{M} \beta_m \, b(x; \, \gamma_m)$$

where $b(x; \, \gamma)$ are base learners (e.g. regression trees).

Forward stagewise approach — fit one term at a time, never revise earlier terms:

$$(\beta_m, \gamma_m) = \arg\min_{\beta, \gamma} \sum_{i=1}^{N} L\big(y_i, \; f_{m-1}(x_i) + \beta \, b(x_i; \gamma)\big)$$

Connection to AdaBoost:

- AdaBoost = forward stagewise with *exponential loss* $L(y, f) = e^{-yf}$
- Gradient Boosting = forward stagewise with *any* loss
- Greedy stage-by-stage optimisation; tractable even for complex loss

Steepest Descent in Function Space

Minimising $L(f) = \sum_i L(y_i, f(x_i))$ over functions f:

At step m, the *steepest-descent direction* is the negative gradient:

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f = f_{m-1}}$$

These r_{im} are called **pseudo-residuals**.

Why fit a tree to pseudo-residuals?

- We cannot move in the gradient direction at arbitrary points
- A regression tree *generalises* the gradient to new data
- The tree approximates the negative gradient surface
- This makes gradient boosting work on test data, not just training

Special case — squared error loss:

$$L(y, f) = \tfrac{1}{2}(y - f)^2 \implies r_{im} = y_i - f_{m-1}(x_i) \quad \text{(ordinary residuals)}$$

Generic Gradient Boosting Algorithm

1. Initialise:

$$f_0(x) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma) \quad \text{(best constant)}$$

2. For $m = 1$ to M:

(a) Compute pseudo-residuals:

$$r_{im} = -\left[\frac{\partial\, L(y_i, f(x_i))}{\partial\, f(x_i)}\right]_{f = f_{m-1}}, \quad i = 1, \ldots, N$$

(b) Fit a regression tree to $\{(x_i, r_{im})\}$ with terminal regions R_{jm}

(c) Compute optimal leaf values:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L\big(y_i,\ f_{m-1}(x_i) + \gamma\big)$$

(d) Update the model:

$$f_m(x) = f_{m-1}(x) + \nu \sum_{j} \gamma_{jm}\, \mathbf{1}[x \in R_{jm}]$$

Loss Functions in Gradient Boosting

Task	**Loss** $L(y, f)$		**Pseudo-residual** r_i
Regression	$\frac{1}{2}(y-f)^2$		$y_i - f(x_i)$
Regression (robust)	$\lvert y-f\rvert$		$\text{sign}(y_i - f(x_i))$
Regression (Huber)	$\begin{cases} \frac{1}{2}(y-f)^2 \\ \delta(\lvert y-f\rvert - \frac{\delta}{2}) \end{cases}$	$\lvert y-f\rvert \le \delta$ otherwise	(see below)
Classification	$\log(1+e^{-2yf})$		$\dfrac{2y_i}{1+e^{2y_i f(x_i)}}$

Huber pseudo-residual:

$$r_{im} = \begin{cases} y_i - f_{m-1}(x_i) & \text{if } |y_i - f_{m-1}(x_i)| \le \delta_m \\ \delta_m \, \text{sign}(y_i - f_{m-1}(x_i)) & \text{otherwise} \end{cases}$$

where δ_m is the α-quantile of $|y_i - f_{m-1}(x_i)|$.

Key takeaway: Choose the loss that matches your task and noise level. Squared error is simple; Huber/absolute loss is robust to outliers.

Shrinkage and Regularisation

Learning rate (shrinkage) ν:

$$f_m(x) = f_{m-1}(x) + \nu \sum_j \gamma_{jm} \mathbf{1}[x \in R_{jm}], \quad 0 < \nu \leq 1$$

- Small $\nu \Rightarrow$ slower learning $\Rightarrow$ more trees M needed
- Small ν + large M almost always gives best test performance
- Typical values: $\nu \in \{0.01, 0.05, 0.1\}$

Stochastic Gradient Boosting (Friedman, 2002):

- At each iteration, draw a random subsample (e.g. $\eta = 0.5$) of training data *without replacement*
- Fit the tree to this subsample only
- Reduces correlation between successive trees
- Improves generalisation and speeds up computation

Tree depth d:

- $d = 1$ (stumps): purely additive model, no interactions
- $d = 2$: allows pairwise interactions
- Typical: $d \in \{4, 5, 6\}$ for most real problems

Gradient Boosting vs. AdaBoost

Aspect	**AdaBoost**	**Gradient Boosting**
Loss function	Exponential only	Any differentiable loss
Mechanism	Reweight samples	Fit to negative gradient
Base learners	Binary classifiers	Regression trees
Regularisation	None built-in	ν, subsampling, depth
Outlier robustness	Sensitive	Robust (Huber / deviance)
Tasks	Binary classification	Regression, classification, ...
Flexibility	Limited	Very high

Key insight:

- AdaBoost is a *special case* of gradient boosting
- Gradient boosting is a general optimisation framework
- Modern implementations (XGBoost, LightGBM, CatBoost) add additional regularisation and second-order information

Tuning Gradient Boosting

Key hyperparameters:

Parameter	Meaning	Typical range
M	Number of trees	100–5000
ν	Learning rate (shrinkage)	0.001–0.1
d	Tree depth	3–8
η	Row subsample fraction	0.5–0.8
ζ	Column subsample fraction	0.5–0.9

Practical strategy:

- Fix $\nu = 0.1$, tune M via cross-validation (or early stopping)
- Then decrease ν and increase M proportionally for final model
- Enable subsampling ($\eta \approx 0.5$) for speed and generalisation
- Use `gbm.perf()` or `xgb.cv()` to find optimal M

Note: Unlike bagging/RF, increasing M with fixed ν *can* overfit — always

XGBoost

Regularised Gradient Boosting · Second-Order Optimisation

Chen & Guestrin, KDD 2016

XGBoost: Motivation and Overview

What is XGBoost?

- E**X**treme **G**radient **Boost**ing (Chen & Guestrin, 2016)
- Regularised extension of gradient boosting machines
- Dominant algorithm in ML competitions (Kaggle, KDD Cup, . . .)

Key innovations over vanilla GBM:

- **Explicit regularisation** in the objective ($\ell_1 + \ell_2$ on leaf weights)
- **Second-order** Taylor approximation of the loss (Newton boosting)
- **Exact and approximate** split-finding algorithms
- **Sparsity-aware** split finding (handles missing values natively)
- **Cache-aware** block structure for out-of-core computation
- **Column subsampling** per tree and per split level

Why it matters:

- Faster training, lower memory footprint
- Better generalisation through regularisation
- Scales to billions of examples on distributed clusters

XGBoost: Regularised Objective

At step m, XGBoost adds tree $f_m \in \mathcal{F}$ (CART) to minimise:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{N} L\big(y_i,\ \hat{y}_i^{(m-1)} + f_m(x_i)\big) \ + \ \Omega(f_m)$$

Regularisation term:

$$\Omega(f) = \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + \alpha \sum_{j=1}^{T} |w_j|$$

- T = number of leaves in the tree
- w_j = predicted score (weight) of leaf j
- $\gamma \geq 0$: penalty per leaf (controls tree complexity)
- $\lambda \geq 0$: ℓ_2 regularisation on leaf weights
- $\alpha \geq 0$: ℓ_1 regularisation on leaf weights (promotes sparsity)

Effect:

- $\gamma > 0$ prevents adding leaves that do not improve the objective by at least γ

Second-Order Taylor Approximation

Expand the loss to second order around $\hat{y}^{(m-1)}$:

$$L\big(y_i,\, \hat{y}^{(m-1)} + f_m(x_i)\big) \;\approx\; L\big(y_i,\, \hat{y}^{(m-1)}\big) + g_i f_m(x_i) + \tfrac{1}{2} h_i f_m(x_i)^2$$

where

$$g_i = \frac{\partial L(y_i,\, \hat{y}^{(m-1)})}{\partial \hat{y}^{(m-1)}} \quad \text{(first-order gradient)}$$

$$h_i = \frac{\partial^2 L(y_i,\, \hat{y}^{(m-1)})}{\partial \big(\hat{y}^{(m-1)}\big)^2} \quad \text{(second-order Hessian)}$$

Dropping constants, the per-step objective becomes:

$$\tilde{\mathcal{L}}^{(m)} = \sum_{i=1}^{N} \big[g_i f_m(x_i) + \tfrac{1}{2} h_i f_m(x_i)^2\big] + \Omega(f_m)$$

Why second order?

- GBM uses only g_i (gradient); XGBoost also uses h_i (curvature)
- Hessian acts as a per-sample learning rate — faster, more stable convergence

Optimal Leaf Weights and Tree Score

Define $I_j = \{i : x_i \in \text{leaf } j\}$ as the instance set of leaf j.
Rewrite the objective over leaves:

$$\tilde{\mathcal{L}}^{(m)} = \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

Optimal leaf weight (closed form, setting derivative to zero):

$$\boxed{w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}}$$

Optimal tree score (minimum objective value for a fixed structure):

$$\boxed{\tilde{\mathcal{L}}^{*(m)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T}$$

This score measures the *quality* of a tree structure. Lower $\tilde{\mathcal{L}}^* \Rightarrow$ better tree.

XGBoost Split Gain

How to find the best split?

For a candidate split that divides leaf I into I_L and I_R, define:

$$G_I = \sum_{i \in I} g_i, \quad H_I = \sum_{i \in I} h_i$$

Gain from the split:

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma$$

- First three terms: reduction in loss from the split
- $-\gamma$: penalty for adding a new leaf
- Split only if Gain > 0 (i.e. improvement exceeds penalty)
- γ acts as a **minimum gain threshold** for pruning

Comparison with GBM:

- GBM: split on residual SS reduction (squared error) or impurity (classification)
- XGBoost: unified gain formula works for *any* loss via g_i, h_i

XGBoost: Sparsity and Missing Values

Sparsity-aware split finding:

- Real data often has missing values or sparse features (e.g. one-hot encoding)
- XGBoost learns a **default direction** for missing values at each split
- Instances with missing feature value are routed to the direction that reduces the objective more
- Direction learned automatically from data — no imputation needed

Algorithm:

For each feature k and split value s, compute Gain for two cases:

1. Missing values go **left**: add missing instances to I_L
2. Missing values go **right**: add missing instances to I_R

Choose the case with higher Gain; record that direction as the default.

Complexity:

- Only non-missing values are visited in the split enumeration
- Complexity $= O(k \cdot \#\text{non-missing})$ per level
- Handles $> 90\%$ sparsity efficiently (e.g. text/recommendation data)

XGBoost vs. GBM: Key Differences

Aspect	**GBM**	**XGBoost**
Gradient order	First order (g_i)	Second order (g_i, h_i)
Regularisation	None / implicit	ℓ_1, ℓ_2, leaf penalty γ
Split criterion	Impurity / residual SS	Unified gain via g_i, h_i
Missing values	External imputation	Native default-direction learning
Subsampling	Row only	Row, column (per tree & per level)
Split search	Exact greedy	Exact + approximate (quantile sketch)
Parallelism	Sequential within tree	Parallel across features
Pruning	Pre-pruning (max depth)	Post-pruning via γ threshold

Practical summary:

- XGBoost $\approx$ GBM with explicit regularisation + Newton updates
- Typically 2–10$\times$ faster than `gbm` for same accuracy
- Hyperparameter λ reduces need to tune tree depth aggressively

XGBoost Hyperparameters

Parameter	Symbol	Role	Default
`eta`	$\eta\ (=\nu)$	Learning rate / shrinkage	0.3
`max_depth`	d	Max tree depth	6
`n_estimators`	M	Number of trees	100
`gamma`	γ	Min gain to make a split	0
`lambda`	λ	ℓ_2 leaf regularisation	1
`alpha`	α	ℓ_1 leaf regularisation	0
`subsample`	η_r	Row subsampling fraction	1
`colsample_bytree`	η_c	Column subsampling per tree	1
`min_child_weight`	$H_{\min}$	Min $\sum h_i$ in a leaf	1

Typical tuning strategy:

1. Fix `eta` $= 0.1$, find optimal M via `xgb.cv` with early stopping
2. Tune `max_depth` $\in \{3, 4, 5, 6\}$ and `min_child_weight`
3. Tune `subsample`, `colsample_bytree` $\in \{0.6, 0.8, 1.0\}$

R Code Example: AdaBoost Classification

Using `ada` Package:

```
library(ada)

# Fit AdaBoost classifier
ada_model <- ada(Species ~ .,
                 data = train,
                 iter = 100,          # number of iterations
                 type = "discrete") # AdaBoost.M1

# Print model summary
summary(ada_model)

# Predictions
pred_ada <- predict(ada_model, newdata = test)

# Accuracy
accuracy <- mean(pred_ada == test$Species)
```

R Code Example: Gradient Boosting (gbm)

Using gbm Package for Regression:

```
library(gbm)

# Fit gradient boosting model
gbm_model <- gbm(medv ~ .,
                 data = train,
                 distribution = "gaussian",  # squared error
                 n.trees = 1000,     # number of trees M
                 interaction.depth = 4,        # tree depth d
                 shrinkage = 0.01,  # learning rate nu
                 bag.fraction = 0.8,# stochastic subsampling
                 cv.folds = 5)       # 5-fold CV

# Find optimal number of trees
best_iter <- gbm.perf(gbm_model,
                      method = "cv",
                      plot.it = TRUE)
```

R Code Example: GBM Predictions and Importance

Make Predictions:

```
# Predictions using optimal number of trees
pred_gbm <- predict(gbm_model,
                    newdata = test,
                    n.trees = best_iter)

# Calculate RMSE
rmse_gbm <- sqrt(mean((test$medv - pred_gbm)^2))
print(paste("GBM RMSE:", round(rmse_gbm, 3)))
```

Variable Importance:

```
# Relative influence of each variable
importance <- summary(gbm_model, n.trees = best_iter)

# Manual bar plot
barplot(importance$rel.inf,
```

R Code Example: GBM for Classification

Binary Classification:

```
# Convert to binary problem
train$is_setosa <- ifelse(train$Species == "setosa",
                          1, 0)
test$is_setosa <- ifelse(test$Species == "setosa",
                         1, 0)

# Fit GBM
gbm_class <- gbm(is_setosa ~ . - Species,
                 data = train,
                 distribution = "bernoulli",  # log-loss
                 n.trees = 500,
                 interaction.depth = 3,
                 shrinkage = 0.05,
                 cv.folds = 5)

# Optimal trees
```

R Code Example: XGBoost (Modern Boosting)

Using `xgboost` Package:

```
library(xgboost)

# Prepare data (xgboost requires matrix)
train_x <- model.matrix(medv ~ . - 1, data = train)
train_y <- train$medv
test_x <- model.matrix(medv ~ . - 1, data = test)

# Create DMatrix
dtrain <- xgb.DMatrix(data = train_x, label = train_y)
dtest  <- xgb.DMatrix(data = test_x,  label = test$medv)

# Hyperparameters
params <- list(
  objective        = "reg:squarederror",
  eta              = 0.1,    # learning rate nu
  max_depth        = 6       # tree depth d
```

R Code Example: XGBoost Training

Train with Cross-Validation:

```
# Cross-validation to find optimal rounds
cv_model <- xgb.cv(
  params = params,
  data   = dtrain,
  nrounds = 1000,
  nfold   = 5,
  early_stopping_rounds = 50,
  verbose = 0
)

# Get best iteration
best_nrounds <- cv_model$best_iteration

# Train final model
xgb_model <- xgb.train(
  params  = params,
```

R Code Example: XGBoost Evaluation

Predictions and Importance:

```
# Predictions
pred_xgb <- predict(xgb_model, dtest)

# RMSE
rmse_xgb <- sqrt(mean((test$medv - pred_xgb)^2))
print(paste("XGBoost RMSE:", round(rmse_xgb, 3)))

# Variable importance
importance <- xgb.importance(
  model = xgb_model,
  feature_names = colnames(train_x)
)

# Plot top-10 most important variables
xgb.plot.importance(importance, top_n = 10)
```

Comparison: Boosting Packages in R

Package	Notes
`ada`	AdaBoost.M1 implementation
`gbm`	Gradient boosting, Friedman's algorithm
`xgboost`	Fastest, most features, regularisation
`mboost`	Model-based boosting
`adabag`	AdaBoost and bagging

Recommendations:

- Learning: `ada` for AdaBoost, `gbm` for GBM
- Production: `xgboost` (fastest, best performance)
- Research: `mboost` (flexible framework)

Trees, Bagging, and Boosting: Comparison

Aspect	**Trees**	**Bagging**	**Boosting**
Training	Single model	Parallel, independent	Sequential, adaptive
Sampling	Full data	Bootstrap samples	Weighted samples
Combination	N/A	Equal weight average	Weighted vote
Focus	Minimise impurity	Reduce variance	Focus on errors
Interpretability	High	Low	Low
Variance	High	Reduced	Reduced

When to Use:

- Trees: interpretability needed, simple baseline
- Bagging: reduce variance of unstable models
- Boosting: maximum predictive accuracy

Random Forests: Introduction

Motivation:

- Bagging reduces variance by averaging many trees
- But bagged trees are highly correlated
- Correlation limits benefits of averaging

Key Innovation:

- Random forests = substantial modification of bagging
- Build large collection of **de-correlated** trees
- Then average them

Performance:

- Similar to boosting on many problems
- Simpler to train and tune than boosting
- Very popular in practice (minimal tuning required)
- Implemented in many packages (R, Python, etc.)

The Correlation Problem in Bagging

Variance of average of B i.i.d. variables:

$$\text{Var}\left(\frac{1}{B}\sum_{b=1}^{B} X_b\right) = \frac{\sigma^2}{B}$$

With correlation ρ (not independent):

$$\text{Var}\left(\frac{1}{B}\sum_{b=1}^{B} X_b\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

Implications:

- As $B \to \infty$: second term vanishes
- First term $\rho\sigma^2$ remains!
- Correlation between trees limits variance reduction
- Bagged trees have same expectation (i.d.), but are not independent

Random Forest Strategy:

- Reduce correlation ρ without increasing variance σ^2 too much

Random Forest Algorithm

For $b = 1$ to B:

1. Draw bootstrap sample Z^* of size N from training data
2. Grow random-forest tree T_b by recursively repeating:
   For each terminal node:

i. Select m variables at random from p variables

ii. Pick best variable/split-point among the m

iii. Split node into two daughter nodes

Continue until minimum node size $n_{\min}$ reached

Prediction:

- Regression: $\hat{f}_{rf}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$
- Classification: $\hat{C}_{rf}(x) =$ majority vote $\{\hat{C}_b(x)\}_{b=1}^{B}$

Key Differences from Bagging

Bagging:

- At each split: consider all p variables
- Choose best split among all variables
- Trees are correlated (similar dominant splits)

Random Forests:

- At each split: consider only $m < p$ random variables
- Choose best split among these m variables
- Trees are less correlated (different variables get a chance)

Typical values for m:

- Classification: $m = \lfloor \sqrt{p} \rfloor$ (default), min. node size $= 1$
- Regression: $m = \lfloor p/3 \rfloor$ (default), min. node size $= 5$
- Sometimes $m = 1$ works well
- Treat m as a tuning parameter

How Random Forests Reduce Correlation

Intuition:

- If few variables dominate, bagged trees will be similar
- All trees will use same strong predictors at top splits
- Random variable selection forces diversity
- Each tree uses a different subset of predictors

Effect of m:

- Smaller $m \Rightarrow$ lower correlation between trees
- Smaller $m \Rightarrow$ potentially higher individual tree variance
- Trade-off governed by: $\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$
- Reducing ρ helps if σ^2 does not increase too much

Typical correlation values:

- Bootstrapped RF trees: $\rho \approx 0.05$ or lower
- Bootstrapped means: $\rho \approx 0.50$ (no improvement with bagging)

Out-of-Bag (OOB) Error Estimation

Key Feature: Built-in cross-validation

For each observation (x_i, y_i)**:**

- Some bootstrap samples do not include (x_i, y_i)
- Average predictions only from trees where (x_i, y_i) was "out-of-bag"
- This gives an unbiased error estimate

Advantages:

- Almost identical to N-fold cross-validation
- No need for a separate validation set
- Can monitor error during training
- Stop when OOB error stabilises

Practical Use:

- Train continuously, track OOB error
- Typically stabilises around 200 trees
- No need to train thousands of trees

Variable Importance in Random Forests

Method 1: Gini Importance (like boosting)

- At each split, record improvement in split criterion
- Accumulate over all trees for each variable
- Higher score $\Rightarrow$ more important variable

Method 2: Permutation Importance (RF-specific)

- For each tree, pass OOB samples down the tree
- Record prediction accuracy
- Randomly permute values of variable j in OOB samples
- Recompute accuracy
- Decrease in accuracy $=$ importance of variable j
- Average over all trees

Differences:

- Gini: can completely ignore some variables
- Permutation: more uniform importance distribution
- Permutation: measures prediction strength when variable removed

Random Forests and Overfitting

Claim: "Random forests cannot overfit"

True for number of trees B:

- As $B \to \infty$: $\hat{f}_{rf}^{B}(x) \to E_{\Theta|Z}\left[T(x; \Theta(Z))\right]$
- Increasing B does not cause overfitting
- Like bagging: approximates expectation

But the limit itself can overfit!

- Fully grown trees $\Rightarrow$ very rich model
- Can incur unnecessary variance
- Especially problematic in regression
- Less of an issue for classification

Tree depth control:

- Can limit depth of individual trees
- Usually gives modest improvements
- Full-grown trees seldom cost much
- One less tuning parameter

When Random Forests Struggle

Problem: Many irrelevant variables

- Suppose few variables relevant, many noise variables
- With small m: low probability relevant variable selected
- Performance degrades compared to boosting

Example:

- 2 relevant, 100 noise variables (total $p = 102$)
- $m = \sqrt{102} \approx 10$
- Probability relevant variable selected: ≈ 0.19
- Many splits will use only noise variables

Robustness to noise:

- With more relevant variables, quite robust
- 6 relevant, 100 noise: probability $= 0.46$ (works well)
- Classification less sensitive than regression
- Misclassification cost less sensitive to probability estimates

Bias-Variance Trade-off in Random Forests

Variance: (already discussed)

$$\text{Var}\big(\hat{f}_{rf}(x)\big) = \rho(x)\,\sigma^2(x)$$

where $\rho(x)$ = correlation between tree pairs, $\sigma^2(x)$ = variance of single tree.

Bias:

$$\text{Bias}(x) = \mu(x) - E\big[\hat{f}_{rf}(x)\big]$$

- Same as bias of individual sampled trees (like bagging)
- Typically greater than unpruned tree (restrictions imposed)
- Randomisation reduces sample space available
- Improvement is solely from variance reduction

Effect of m:

- As m decreases: correlation ρ decreases (good)
- As m decreases: bias increases (bad)
- Classical bias-variance trade-off
- Optimal m balances these effects

Random Forests vs. Adaptive k-NN

Similarity to k-Nearest Neighbours:

- Each fully-grown tree: prediction is training sample response
- Tree finds "optimal" path using most informative predictors
- Averaging assigns weights to training observations
- Observations close to target get higher weights
- Effectively a weighted k-NN with adaptive kernel

Key Differences:

- k-NN: fixed distance metric, equal weights within k
- Random forests: adaptive distance, data-driven weights
- Random forests: considers variable importance
- Random forests: naturally handles mixed variable types

Decision Boundaries:

- Both produce complex, non-linear boundaries
- Random forests: axis-oriented (from trees)
- Can produce similar classification performance

Random Forests vs. Boosting: Empirical Comparisons

General Patterns:

- Performance often similar on many problems
- Random forests stabilise faster (around 200 trees)
- Boosting can continue improving (1000+ trees)
- Boosting often (slightly) better final performance

When Random Forests Excel:

- High-dimensional with many relevant variables
- When simplicity and speed are priorities
- Less tuning required
- Good "off-the-shelf" performance

When Boosting Excels:

- When maximum accuracy is needed
- Problems with complex interactions
- When willing to tune carefully
- Often better with proper shrinkage

Summary: Random Forests

Core Innovation:

- De-correlate trees via random feature selection
- Reduces ρ in variance formula: $\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$
- Better variance reduction than bagging

Key Features:

- Out-of-bag error estimation (built-in CV)
- Variable importance measures
- Minimal tuning (main parameter: m)
- Cannot overfit in B (number of trees)

Practical Recommendations:

- Default m: $\sqrt{p}$ (classification), $p/3$ (regression)
- Typical B: 200–500 trees
- Monitor OOB error to decide when to stop
- Great for initial modelling; compare with boosting for best performance

R Code Example: Random Forest for Classification

Using `randomForest` Package:

```
library(randomForest)

# Fit random forest
set.seed(123)
rf_model <- randomForest(Species ~ .,
                         data = train,
                         ntree = 500,          # number of trees
                         mtry = 2,             # sqrt(4) = 2
                         importance = TRUE,    # compute importar
                         proximity = TRUE)     # compute proximit

# Print model summary
print(rf_model)

# OOB error estimate
print(paste("OOB Error Rate:",
```

R Code Example: RF Predictions and Evaluation

Make Predictions:

```
# Class predictions
pred_rf <- predict(rf_model, newdata = test)

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

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

# Confusion matrix
conf_mat <- table(Predicted = pred_rf,
                  Actual = test$Species)
print(conf_mat)
```

R Code Example: RF for Regression

Random Forest Regression:

```
library(randomForest)

# Fit random forest
rf_reg <- randomForest(medv ~ .,
                       data = train,
                       ntree = 500,
                       mtry = 4,              # p/3 ~ 13/3
                       importance = TRUE)

# Predictions
pred_rf <- predict(rf_reg, newdata = test)

# RMSE
rmse_rf <- sqrt(mean((test$medv - pred_rf)^2))
print(paste("RF RMSE:", round(rmse_rf, 3)))
```

R Code Example: Variable Importance

Plot Variable Importance:

```
# Variable importance
varImpPlot(rf_model,
           main = "Variable Importance")

# Get importance values
importance(rf_model)

# Alternative: specific plot
# MeanDecreaseAccuracy
imp <- importance(rf_model)[, "MeanDecreaseAccuracy"]
imp_sorted <- sort(imp, decreasing = TRUE)

barplot(imp_sorted,
        las = 2,  # rotate labels
        col = "steelblue",
        main = "Variable Importance (Accuracy)")
```

R Code Example: Tuning mtry

Find Optimal `mtry`:

```
# Try different mtry values
mtry_values <- 1:4
oob_errors <- numeric(length(mtry_values))

for(i in seq_along(mtry_values)) {
  rf_temp <- randomForest(Species ~ .,
                          data = train,
                          ntree = 500,
                          mtry = mtry_values[i])
  oob_errors[i] <- rf_temp$err.rate[500, "OOB"]
}

# Plot OOB error vs mtry
plot(mtry_values, oob_errors, type = "b",
     xlab = "mtry", ylab = "OOB Error",
     main = "Tuning mtry parameter")
```

R Code Example: Using tuneRF

Automatic `mtry` Tuning:

```
# Automatic tuning of mtry
set.seed(123)
tuned <- tuneRF(x = train[, -5],       # predictors
                y = train[, 5],         # response
                mtryStart = 2,          # starting value
                ntreeTry = 500,         # trees per trial
                stepFactor = 1.5,       # inflation factor
                improve = 0.01,         # improvement needed
                trace = TRUE,           # print progress
                plot = TRUE)            # plot results

# Best mtry
best_mtry <- tuned[which.min(tuned[, 2]), 1]
print(paste("Optimal mtry:", best_mtry))
```

R Code Example: Partial Dependence Plots

Visualise Variable Effects:

```
# Partial dependence plot for one variable
partialPlot(rf_model,
            pred.data = train,
            x.var = "Petal.Length",
            which.class = "setosa")

# For regression
partialPlot(rf_reg,
            pred.data = train,
            x.var = "rm",  # avg rooms
            main = "Partial Dependence on Rooms")

# Multiple variables (requires pdp package)
library(pdp)
partial(rf_reg, pred.var = c("rm", "lstat"),
        plot = TRUE, chull = TRUE)
```

R Code Example: OOB Error Evolution

Track OOB Error Over Trees:

```
# Plot OOB error rate vs number of trees
plot(rf_model,
     main = "OOB Error vs Number of Trees")
legend("topright",
       colnames(rf_model$err.rate),
       col = 1:4, lty = 1:4)

# For regression
plot(rf_reg$mse, type = "l",
     xlab = "Number of Trees",
     ylab = "OOB MSE",
     main = "OOB Error Convergence")

# Add line at stabilization point
abline(v = 200, col = "red", lty = 2)
```

R Code Example: Comparing All Methods

Side-by-Side Comparison:

```
library(rpart); library(randomForest)
library(gbm); library(ipred)

# Single tree
tree <- rpart(medv ~ ., data = train)
pred_tree <- predict(tree, test)

# Bagging
bag <- bagging(medv ~ ., data = train, nbagg = 100)
pred_bag <- predict(bag, test)

# Random Forest
rf <- randomForest(medv ~ ., data = train, ntree = 500)
pred_rf <- predict(rf, test)

# GBM
```

R Code Example: Performance Comparison

Calculate RMSEs:

```
# Function to calculate RMSE
rmse <- function(actual, predicted) {
  sqrt(mean((actual - predicted)^2))
}

# Compare all methods
results <- data.frame(
  Method = c("Single Tree", "Bagging",
             "Random Forest", "GBM"),
  RMSE = c(
    rmse(test$medv, pred_tree),
    rmse(test$medv, pred_bag),
    rmse(test$medv, pred_rf),
    rmse(test$medv, pred_gbm)
  )
)
```

Complete Comparison: All Methods

Method	**Trees**	**Bagging**	**Boosting**	**RF**
Training	Single	Parallel	Sequential	Parallel
Sampling	Full	Bootstrap	Weighted	Bootstrap
Features	All	All	All	Random m
Combination	–	Equal avg	Weighted	Equal avg
Correlation	–	High	–	Low
Tuning	Depth	B	B, d, ν	B, m
Speed	Fast	Moderate	Slow	Moderate
Accuracy	Low	Good	Best	Very good
Interpret.	High	Low	Low	Low

Bottom Line:

- Trees: starting point, interpretable
- Bagging: variance reduction for trees
- Random forests: improved bagging (de-correlation)
- Boosting: sequential error correction (often best)