RANDOM FOREST

Complete Reference | Ensemble Learning | Bagging + Feature Randomness

Bootstrap | Decision Trees | OOB Error | Feature Importance | Hyperparameters

What is Random Forest?

An ENSEMBLE learning method that builds MULTIPLE decision trees and combines their predictions.

Classification: majority vote of all trees. Regression: average of all tree predictions.

Invented by Leo Breiman (2001). Combines BAGGING (Bootstrap Aggregating) + RANDOM FEATURE SELECTION.

Core idea: "Many weak learners together make a strong learner" -- Wisdom of Crowds for ML.

The 3 Pillars of Random Forest

1. BOOTSTRAP Sampling

Draw N samples WITH REPLACEMENT from training data. Each tree gets a different dataset.

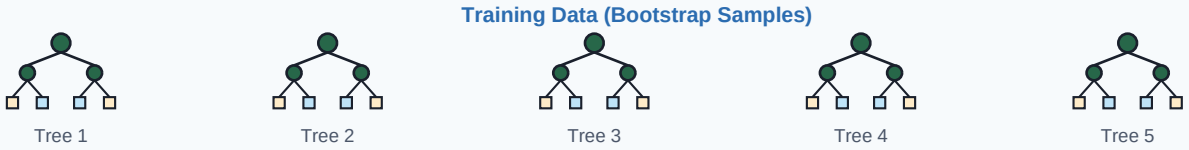
2. RANDOM Feature Selection

At each split, consider only m random features out of total M (not all). m = sqrt(M) typical.

3. AGGREGATION

Combine all tree outputs: MAJORITY VOTE (classification) or MEAN (regression).

How Random Forest Works (Visual Overview)



Why Does Random Forest Work?

VARIANCE REDUCTION: Averaging multiple high-variance trees reduces overall variance (Bias-Variance tradeoff).

DECORRELATION: Random feature selection ensures trees are different (not all split on the same strong feature).

ROBUSTNESS: Less prone to overfitting than a single decision tree. Works well with default hyperparameters.

Mathematically: Var(avg of B trees) = rho * sigma^2 + (1-rho)/B * sigma^2, where rho = avg correlation.

Mnemonic: "BRA" = Bootstrap + Random features + Aggregate

B = Bootstrap data, R = Random subset of features at each split, A = Aggregate predictions.

BOOTSTRAP SAMPLING: The Foundation

Sampling WITH replacement creates diverse training sets

What is Bootstrap Sampling?

Given a dataset of N samples, draw N samples WITH REPLACEMENT to create a new dataset.
Some samples appear multiple times, some don't appear at all. Each bootstrap set is ~63.2% unique.
The ~36.8% left out are called OOB (Out-Of-Bag) samples -- used for free validation!

Worked Example: Bootstrap on 6-Sample Dataset

Original Dataset (N=6):

ID	Age	Income	Buys
S1	25	40K	Yes
S2	30	60K	No
S3	35	80K	Yes
S4	40	50K	No
S5	45	70K	Yes
S6	50	90K	No

Bootstrap Sample 1 (draw 6 with replacement):

ID	Age	Income	Buys
S1	25	40K	Yes
S3	35	80K	Yes
S3	35	80K	Yes
S5	45	70K	Yes
S2	30	60K	No
S1	25	40K	Yes

S1 appears 2x, S3 appears 2x (duplicates!)

S4 and S6 are MISSING (OOB samples!)

OOB = {S4, S6} -> used to estimate error

Key Statistics:

P(sample NOT picked in 1 draw) = (1 - 1/N)
P(sample NOT in bootstrap) = (1 - 1/N)^N
As N -> infinity, this -> 1/e = 0.368 = 36.8%

OOB (Out-Of-Bag) Error: Free Cross-Validation!

For each sample x_i, ~1/3 of the trees did NOT train on it (it was OOB for those trees).
Predict x_i using ONLY the trees where x_i was OOB. Compare with true label -> OOB error.
OOB error is an unbiased estimate of test error -- NO NEED for separate validation set!
This is equivalent to ~3-fold cross-validation but comes for free during training.
OOB error formula: OOB_error = (1/N) * sum(I(y_i != y_hat_OOB_i)) for classification.

With Bootstrap (Bagging)

Each tree sees different data
Reduces variance via averaging
~63.2% unique per tree
OOB gives free validation

Without Bootstrap (Pasting)

Each tree sees same data
Higher variance, lower diversity
All samples used exactly once
Need external cross-validation

Mnemonic: "OOB = Oh, Others are Busy (training)"

The ~37% of data NOT used for a tree can validate that tree. Free lunch in ML!

RANDOM FEATURE SELECTION & TREE BUILDING

How each decision tree is constructed in the forest

Random Feature Selection (Feature Bagging)

At EACH node split, randomly select m features from total M features. Find best split among these m only.
This is different from regular bagging! Regular bagging considers ALL features at each split.
The random subset is RE-DRAWN at every node, not once per tree. This maximizes decorrelation.
Default m values: Classification: m = sqrt(M) | Regression: m = M/3 | Can be tuned.
Why? If one feature is very strong, without this, ALL trees would split on it first -> correlated trees.

Example: Feature Selection at a Node

Total features M = 9: {Age, Income, Education, Gender, City, Job, Score, Hours, Dept}

m = sqrt(9) = 3. At this node, randomly pick 3:

Selected: {Income, Score, City}. Find best split among ONLY these 3. Suppose Income < 50K is best.

At the NEXT node, pick 3 again (could be different!): {Age, Dept, Hours}. Best split: Age < 30.

This random re-selection at every split is what decorrelates the trees!

Building a Single Tree: Step-by-Step

Step 1: Draw bootstrap sample of N rows from training data (with replacement).

Step 2: Start at root node with the entire bootstrap sample.

Step 3: Randomly select m features from M total features.

Step 4: Find the best split among these m features (using Gini / Entropy / MSE).

Step 5: Split the node into two child nodes based on best split.

Step 6: Repeat Steps 3-5 recursively for each child node.

Step 7: Stop when: max_depth reached, min_samples_leaf met, node is pure, or no improvement.

Step 8: Leaf nodes store: class label (classification) or mean value (regression).

Splitting Criteria: How to Choose the Best Split

Gini Impurity (Default for classification in sklearn):

Gini(t) = 1 - sum(p_k^2) for all classes k. Range: [0, 1-1/K]. 0 = pure node.

Information Gain / Entropy:

Entropy(t) = -sum(p_k * log2(p_k)). IG = Entropy(parent) - weighted_avg(Entropy(children)).

MSE (For regression):

MSE(t) = (1/N) * sum((y_i - y_mean)^2). Choose split that minimizes total child MSE.

Key Insight: Trees Are Grown DEEP (Usually Unpruned!)

Individual trees overfit (high variance, low bias). That's OK! Averaging many overfitting trees REDUCES variance.

COMPLETE ALGORITHM & PSEUDOCODE

Training + Prediction in detail

RANDOM-FOREST-TRAIN

```
Input:
 D (dataset), B (num trees),
       m (features per split)

for
 i = 1 to B:
  // Step 1: Bootstrap
  D_i = BootstrapSample(D, N)

  // Step 2: Build tree
  T_i = GrowTree(D_i, m)

def
 GrowTree
(D, m):
  if stopping_condition(D):
    return LeafNode(predict(D))
  // Pick m random features
  F = RandomSubset(features, m)
  // Find best split in F
  (f*, t*) = BestSplit(D, F)
  D_L, D_R = Split(D, f*, t*)
  left = GrowTree(D_L, m)
  right = GrowTree(D_R, m)
```

RANDOM-FOREST-PREDICT

```
Input: Forest, x_new

// Get prediction from each tree
for i = 1 to B:
  y_i = TreePredict(T_i, x_new)

// Classification: Majority Vote
if task == classification:
  y_final = mode(y_1, ..., y_B)
  confidence = count(mode) / B

// Regression: Average
if task == regression:
  y_final = mean(y_1, ..., y_B)
  uncertainty = std(y_1, ..., y_B)

return y_final

// TreePredict: walk down tree
def TreePredict(T, x):
  if T is leaf: return T.value
  if x[T.feature] <= T.threshold:
    return TreePredict(T.left, x)
  else:
    return TreePredict(T.right, x)
```

Complexity Analysis

Phase	Time Complexity	Space	Notes
Training	O(B * N * M' * log N)	O(B * N)	M' = m features, parallelizable
Prediction	O(B * log N)	O(1) per sample	Can be parallelized
OOB Error	O(N * B/3 * log N)	O(N)	Computed during training

Key Implementation Details

Trees are INDEPENDENT -- fully parallelizable! Set n_jobs=-1 in sklearn for all CPU cores.

Each tree stores its bootstrap indices -> enables OOB computation without extra data splitting.

Default: B=100 trees in sklearn. More trees = better but diminishing returns after ~200-500.

WORKED EXAMPLE: Building a Random Forest

Step-by-step with 3 trees, 6 data points

Original Dataset: "Will Customer Buy?" (N=6, M=3 features)

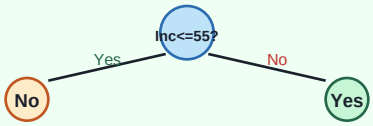
ID	Age	Income(K)	Student?	Buys? (label)
S1	25	40	Yes	No
S2	30	60	No	No
S3	35	80	Yes	Yes
S4	40	50	No	Yes
S5	45	70	Yes	Yes
S6	50	90	No	No

Tree 1: Bootstrap = {S1, S3, S3, S5, S2, S1} | OOB = {S4, S6}

m = sqrt(3) ~ 2 features per split.

Node 1: Random features selected = {Age, Income}

Best split: Income <= 55K? Yes -> {S1,S1,S2} (2 No, 0 Yes = No), Right -> {S3,S3,S5} (3 Yes = Yes)



Tree 2: Bootstrap = {S4, S6, S2, S4, S5, S3} | OOB = {S1}

m = 2. Node 1: features = {Student?, Age}

Best split: Age <= 42? Yes -> {S2,S4,S4,S3} (2Y,2N -> further split), No -> {S6,S5} (1Y,1N -> further split)

After full growth: Tree 2 predicts based on Age and Student status.

OOB prediction for S1: Tree 2 predicts "No" (correct!)

Tree 3: Bootstrap = {S2, S5, S1, S5, S6, S3} | OOB = {S4}

m = 2. Node 1: features = {Income, Student?}. Best split: Student? = Yes.

Yes -> {S5,S5,S1,S3} (3Y,1N -> Yes), No -> {S2,S6} (0Y,2N -> No)

OOB prediction for S4 (not student, income=50K): Tree 3 predicts "No" (actual=Yes, wrong!)

Aggregation: Final Predictions (Majority Vote)

For a NEW sample x = (Age=32, Income=55K, Student=Yes):

Tree 1: Income<=55 -> YES path... predicts "No"

Tree 2: Age<=42 -> YES path... predicts "Yes"

Tree 3: Student?=Yes path... predicts "Yes"

Majority Vote: Yes=2, No=1 --> Final Prediction: "Yes" (confidence = 2/3 = 66.7%)

OOB Error: S4 was misclassified by Tree 3. Overall OOB accuracy depends on all trees' OOB predictions.

FEATURE IMPORTANCE & OOB ERROR

Understanding which features matter and model validation

Feature Importance: 2 Methods

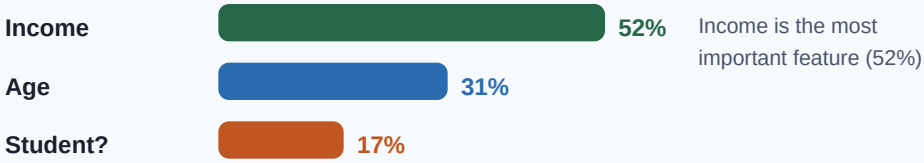
Method 1: Mean Decrease in Impurity (MDI) -- Default in sklearn

For each feature, sum up the total reduction in Gini/Entropy across ALL nodes in ALL trees where that feature was used.
Importance(f) = sum over all trees, all nodes where f splits: (N_t/N) * delta_impurity.
Normalized so all importances sum to 1. Fast (computed during training). Biased toward high-cardinality features.

Method 2: Permutation Importance (More Reliable)

For each feature f: randomly shuffle column f in test/OOB data. Measure drop in accuracy.
Importance(f) = accuracy_original - accuracy_after_permuting_f. If important, accuracy drops a lot.

Example: Feature Importance (from our dataset)



Income is the most important feature (52%)

OOB Error Estimation: Detailed Process

1. For each sample S_i, identify all trees where S_i was OOB (not in that tree's bootstrap).
2. Collect predictions from those OOB trees only. Take majority vote (classification) or mean (regression).
3. Compare OOB prediction with true label. Count misclassifications.
4. OOB error rate = total misclassified / N. This approximates leave-one-out cross-validation error.
5. As B (number of trees) increases, OOB estimate becomes more accurate (more trees vote per sample).

OOB Advantages

Free (no extra computation)
Unbiased test error estimate
No data wasted on validation
Converges as B increases
Good for hyperparameter tuning

When to Use k-Fold CV Instead

Very small datasets (N < 50)
Need precise CI on error
Comparing with non-RF models
Bootstrap ratio is unusual
Stratified splits needed

Bonus: Proximity Matrix (Similarity Between Samples)

For each pair (i,j): count how often i and j end up in the SAME leaf node across all trees.
Proximity(i,j) = (count of same leaf) / B. Used for: clustering, outlier detection, missing value imputation.

HYPERPARAMETERS: Complete Tuning Guide

Every parameter with default, range, and impact

Hyperparameter Reference (sklearn names)

Parameter	Default	Range/Options	Effect	Priority
`n_estimators`	100	50 - 1000+	More = better but slower	HIGH
`max_features`	sqrt(M)	sqrt, log2, int, float	Controls tree decorrelation	HIGH
`max_depth`	None	3 - 30, or None	Limits tree depth	MEDIUM
`min_samples_split`	2	2 - 20	Min samples to split node	MEDIUM
`min_samples_leaf`	1	1 - 10	Min samples in leaf	MEDIUM
`bootstrap`	True	True / False	Enable bootstrap sampling	HIGH
`oob_score`	False	True / False	Compute OOB error	LOW
`n_jobs`	None	-1, 1, 2, ...	Parallel cores (-1 = all)	UTIL
`random_state`	None	any int	Reproducibility seed	UTIL
`class_weight`	None	balanced, dict	Handle imbalanced classes	MEDIUM
`max_samples`	None	float (0,1]	Bootstrap sample size ratio	LOW
`criterion`	gini	gini, entropy, log_loss	Split quality measure	LOW
`min_impurity_decrease`	0.0	0.0 - 0.5	Min impurity decrease to split	LOW
`max_leaf_nodes`	None	int or None	Max number of leaves	LOW

Hyperparameter Tuning Strategy (Recommended Order)

Step 1: Start with defaults. Compute OOB score. This is your baseline.

Step 2: Increase n_estimators (200, 500, 1000). Plot OOB error vs n_estimators. Stop when it plateaus.

Step 3: Tune max_features: try sqrt(M), log2(M), M/3, 0.3, 0.5. Use OOB or CV.

Step 4: Tune max_depth (5, 10, 15, 20, None) and min_samples_leaf (1, 2, 5, 10).

Step 5: For imbalanced data: set class_weight='balanced' or use SMOTE.

Step 6: Use RandomizedSearchCV or Optuna for fine-tuning. OOB score is a fast proxy.

Tuning Tips & Gotchas

More trees NEVER hurts accuracy (only computation time). When in doubt, add more trees.

Lower max_features = more randomness = more decorrelation = often better generalization.

If OOB error and CV error diverge significantly, your bootstrap ratio might be off.

CLASSIFICATION vs REGRESSION + VARIANTS

How RF adapts to different tasks

RF for Classification

Each tree votes for a class label
Final: MAJORITY VOTE
Confidence = vote fraction
Criterion: Gini or Entropy
max_features = sqrt(M) default
Leaf stores: class label
Output: predict() = class
Output: predict_proba() = [p1,..,pk]

RF for Regression

Each tree predicts a real value
Final: AVERAGE (mean)
Uncertainty = std of predictions
Criterion: MSE or MAE
max_features = M/3 default
Leaf stores: mean of samples
Output: predict() = real number
Can also use median for robustness

Random Forest Variants & Related Methods

Bagging (Bootstrap Aggregating) — Breiman 1996
Same as RF but uses ALL features at each split. Less decorrelation.

Extra Trees (Extremely Randomized) — Geurts 2006
Random thresholds too (not just features). Even faster, more random.

Isolation Forest — Liu 2008
RF for anomaly detection. Isolates outliers with fewer splits.

Gradient Boosted Trees (GBT) — Friedman 2001
Trees built SEQUENTIALLY, each correcting errors of previous. Different!

XGBoost / LightGBM / CatBoost — 2014-2017
Optimized GBT implementations. Often outperform RF on tabular data.

Random Forest vs Gradient Boosting (Key Exam Comparison)

Aspect	Random Forest	Gradient Boosting
Trees built	PARALLEL (independent)	SEQUENTIAL (dependent)
Reduces	Variance	Bias (then variance)
Overfitting	Rarely overfits	Can overfit if too many trees
Speed	Faster (parallelizable)	Slower (sequential)
Tuning	Works well with defaults	Needs careful tuning (lr, depth)

When to Use Random Forest

Tabular data (structured)
Need feature importance
Want minimal tuning
Medium-sized datasets
Need robust baseline
Imbalanced classes (with weights)

When NOT to Use RF

Images/text/audio (use deep learning)
Very high-dimensional sparse data
Need interpretable single model
Streaming/online learning
Very small datasets (< 50 samples)
When GBT consistently outperforms

SKLEARN CODE & COMMON PITFALLS

Ready-to-use code templates + mistakes to avoid

Classification Template

```
from sklearn.ensemble import \
    RandomForestClassifier
from sklearn.model_selection import \
    train_test_split

X_train, X_test, y_train, y_test = \
  train_test_split(X, y, test_size=0.2)

rf = RandomForestClassifier(
    n_estimators=200,
    max_features='sqrt',
    oob_score=True,
    n_jobs=-1,
    random_state=42
)
rf.fit(X_train, y_train)

print(f'OOB: {rf.oob_score_:.3f}')
print(f'Test: {rf.score(X_test,y_test):.3f}')
```

Regression Template

```
from sklearn.ensemble import \
    RandomForestRegressor

rf_reg = RandomForestRegressor(
    n_estimators=200,
    max_features=0.33,  # M/3
    max_depth=15,
    min_samples_leaf=3,
    oob_score=True,
    n_jobs=-1,
    random_state=42
)
rf_reg.fit(X_train, y_train)

y_pred = rf_reg.predict(X_test)

from sklearn.metrics import \
    mean_squared_error
mse = mean_squared_error(y_test, y_pred)
```

Feature Importance Extraction

```
import numpy as np
# MDI Importance (built-in)
importances = rf.feature_importances_
indices = np.argsort(importances)[::-1]
for i in range(X.shape[1]):
    print(f"{feature_names[indices[i]]}: {importances[indices[i]]:.4f}")

# Permutation Importance (more reliable)
```

Common Pitfalls & How to Avoid Them

Data leakage in features: Don't include target-derived features. Split BEFORE any preprocessing.

Not scaling? That's OK!: RF doesn't need feature scaling (unlike SVM, kNN). Tree splits are rank-based.

Imbalanced classes: Use class_weight='balanced' or 'balanced_subsample'. Or oversample minority.

Too few trees: Start with 200+. Plot OOB error vs n_estimators to find the plateau.

Ignoring OOB score: Always set oob_score=True. It's free validation! Compare with test score.

High-cardinality categoricals: RF favors features with many unique values (MDI bias). Use permutation importance.

Extrapolation: RF CANNOT extrapolate beyond training range (regression). It predicts means, not trends.

Pro Trick: RF as Feature Selector

Train RF -> get feature importances -> select top-k features -> train a simpler model (logistic regression, SVM) on those features.

MASTER CHEATSHEET & EXAM QUICK REFERENCE

"BRA" = Bootstrap + Random Features + Aggregate

Build B trees, each on bootstrap sample, using m random features per split. Combine: vote (class) or mean (reg).

Why it works: reduces VARIANCE by averaging decorrelated high-variance trees. Bias stays same.

Algorithm in 7 Steps

1. For each tree t = 1 to B:
2. Draw bootstrap sample D_t (N samples with replacement)
3. At each node: randomly select m features from M total
4. Find best split among m features (Gini/Entropy/MSE)
5. Grow tree fully (no pruning, unpruned = high variance, that's OK)
6. Predict: pass new x through ALL B trees
7. Aggregate: majority vote (classification) or mean (regression)

Key Numbers to Remember

~63.2% unique samples per bootstrap
~36.8% OOB (out-of-bag) per tree
m = sqrt(M) for classification
m = M/3 for regression
B >= 200 trees recommended

Strengths

Works out-of-the-box (great defaults)
Handles missing values, outliers
No feature scaling needed
Built-in feature importance
Parallelizable, fast training

Quick Comparison Table

Method	Trees	Built	Reduces	Tuning	Speed
Bagging	Parallel	All features	Variance	Easy	Fast
Random Forest	Parallel	Random features	Variance+	Easy	Fast
Boosting (GBT)	Sequential	All features	Bias first	Hard	Slower
Extra Trees	Parallel	Random thresholds	Variance++	Easiest	Fastest

All Mnemonics for Random Forest

"BRA" = Bootstrap + Random features + Aggregate (the 3 pillars)
"OOB = Oh, Others are Busy" (~37% left out per tree = free validation)
"Vote for Class, Average for Value" (classification vs regression)
"More Trees, More Peace" (adding trees never hurts accuracy, only computation)

Exam Tips & Key Formulas

1. Bootstrap: P(sample not in tree) = (1 - 1/N)^N -> 1/e = 36.8%. Unique fraction = 63.2%.
2. Variance reduction: Var(RF) = rho*sigma^2 + (1-rho)*sigma^2/B. Lower rho (correlation) = lower variance.
3. RF reduces VARIANCE not bias. Individual trees are unbiased but high variance.
4. OOB error ~ test error (unbiased). No separate validation needed if using OOB.
5. Feature importance (MDI): sum of weighted impurity decreases. Biased for high-cardinality.
6. RF cannot extrapolate! Predictions bounded by training data range (key limitation for regression).