Machine Learning in Finance (RSM338)

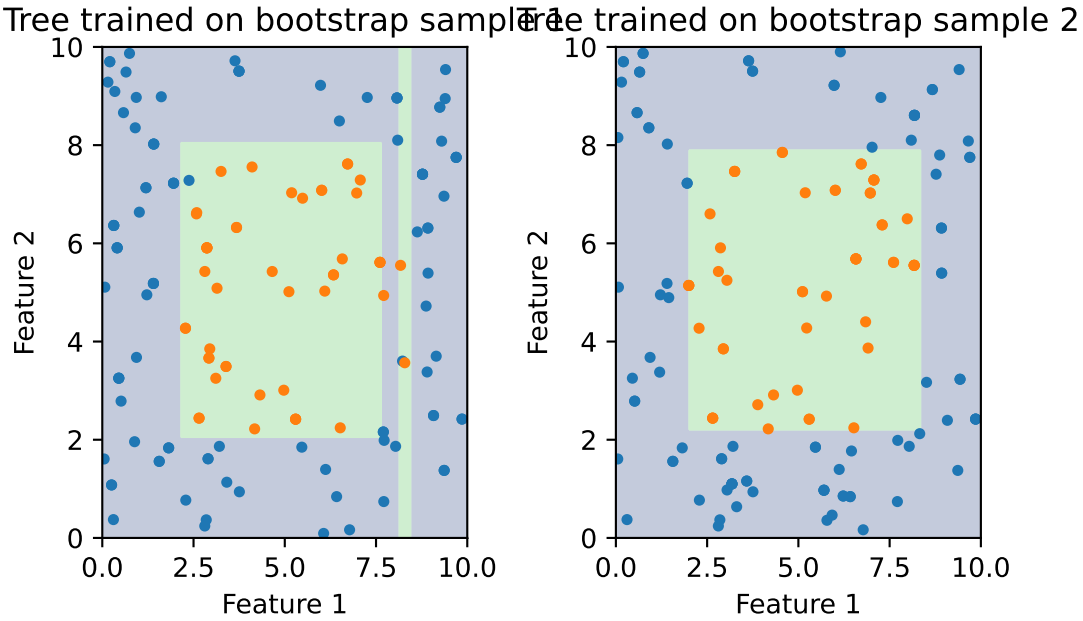
Week 9: Ensemble Methods

Table of contents

1 Introduction

Last week we saw that decision trees are interpretable and flexible, but they have a serious weakness: high variance. Small changes in the training data can produce completely different trees. A tree trained on one bootstrap sample of loan applications might split first on credit score, while a tree trained on a slightly different sample splits first on debt-to-income ratio. The two trees produce different decision boundaries, different predictions, and different conclusions about which features matter. This instability is the price we pay for the tree's flexibility — it can fit almost any pattern in the data, including patterns that are really just noise.

This week we fix this problem by combining many trees into an **ensemble** — a team of models that's stronger than any individual member. The idea is simple: if one tree is noisy, average many trees and the noise cancels out. We'll cover two broad families of ensemble methods. **Random Forests** take the parallel approach: train many trees independently, each on a different random subset of the data and features, then average their predictions. **Boosting** takes the sequential approach: build trees one at a time, with each new tree focusing on the mistakes made by the previous ones. Both strategies produce dramatic improvements over single trees, and ensemble methods built on these ideas are among the most widely used models in finance and machine learning more broadly.



The figure above illustrates the problem. Both trees are trained on the same underlying data, but with different bootstrap samples (random draws with replacement). The resulting decision boundaries look quite different. This is what high variance looks like — the model's output is overly sensitive to which particular training observations happen to be included.

2 The Ensemble Idea

2.1 Wisdom of Crowds

In 1906, Francis Galton observed that when a crowd of 800 people guessed the weight of an ox at a county fair, the average of their guesses was nearly perfect — closer to the true weight than almost any individual guess. The same principle applies to statistical models. If individual models make errors that are somewhat independent, averaging their predictions cancels out the errors. One decision tree might overfit to noise in one part of the data, while another tree overfits to different noise. When we combine them, the individual

mistakes wash out and what remains is the true signal. This is the core idea behind ensemble methods: a committee of models outperforms any single member.

2.2 Why Averaging Reduces Variance

The mathematics behind this are straightforward. Suppose we have B models, each with variance σ^2. If the models are independent, the variance of their average is:

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

With 100 independent models, we cut the variance by a factor of 100. More models means less variance. Each individual model might be noisy, but the noise averages out when we combine many of them. The signal — which is the same across all models — survives the averaging.

But there's a catch. The σ^2/B formula assumes the models are independent. In practice, models trained on the same data are correlated — they tend to make similar mistakes. When models have pairwise correlation ρ (the Greek letter "rho," measuring how similar the models' predictions are), the variance of their average becomes:

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

The first term $\rho\sigma^2$ doesn't shrink as we add more models — it's the irreducible cost of correlation. Only the second term decreases with B. To get the most benefit from ensembling, we need to make the models as different from each other as possible while keeping each one reasonably accurate.

2.3 Three Strategies for Diversity

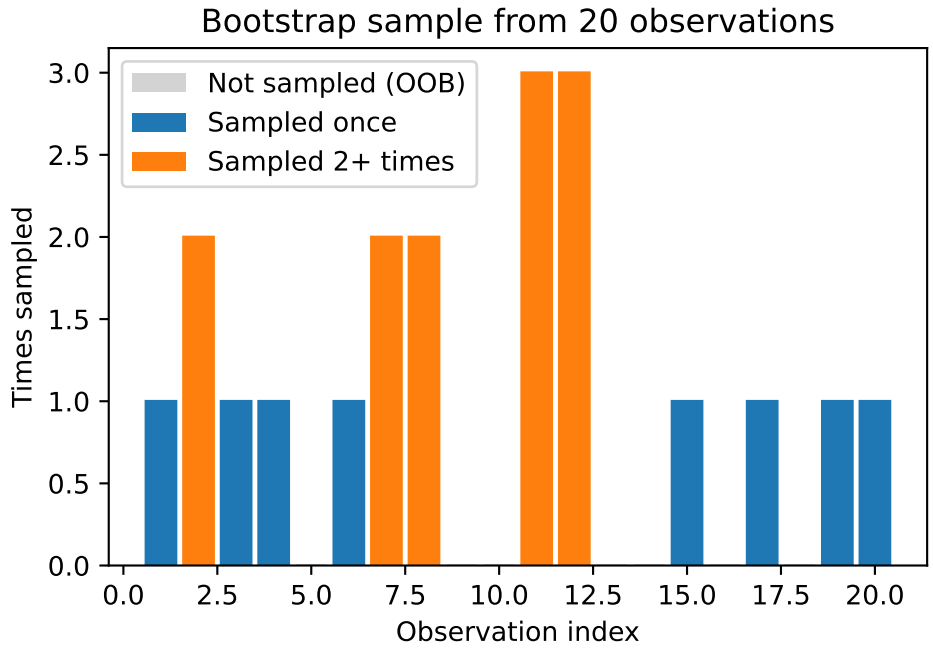
How can we train models on the same dataset but make them disagree? There are three main strategies:

Strategy	How it works	Method
Different data	Train each model on a random subset of observations	Bagging
Different features	At each split, only consider a random subset of features	Random Forests
Sequential correction	Each new model focuses on the mistakes of previous ones	Boosting

Bagging and Random Forests take the parallel approach: train many models independently, then combine. Boosting takes the sequential approach: build models one at a time, each improving on the last.

2.4 Bootstrap Sampling

Bootstrap sampling means drawing n observations from the training data with replacement. Some observations appear multiple times, others are left out entirely.



Output

```
Unique observations in this bootstrap sample: 13 out of 20 (65%)
```

On average, each bootstrap sample contains about 63.2% of the unique observations. The remaining ~36.8% are called **out-of-bag (OOB)** observations — we'll use them for free validation later.

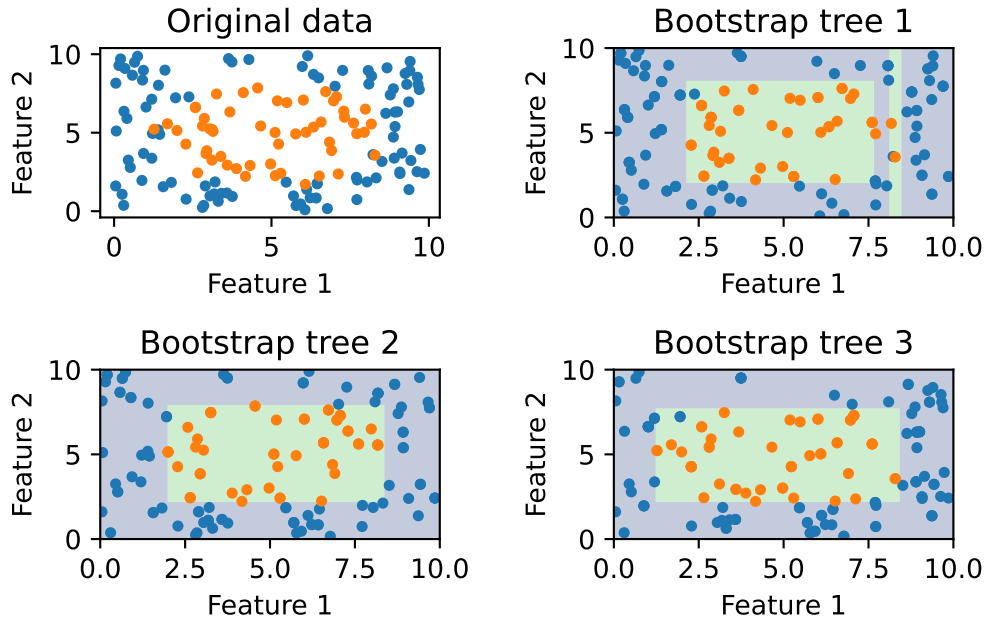
2.5 Bagging: Bootstrap Aggregation

Bagging (Breiman, 1996) combines bootstrap sampling with model averaging. The procedure is:

1. Draw B bootstrap samples from the training data
2. Train one decision tree on each bootstrap sample
3. To predict a new observation: have all B trees vote (classification) or average their predictions (regression)

$$\hat{y}(x) = \text{majority vote of } \{\hat{f}_1(x), \hat{f}_2(x), \ldots, \hat{f}_B(x)\}$$

Each tree sees a slightly different version of the data, so they learn slightly different patterns. The ensemble vote smooths out the individual quirks.



Each bootstrap tree learns a slightly different boundary. When we average all of them, the individual irregularities cancel out and we get a smoother, more stable prediction.

Bagging reduces variance by averaging over many trees, but there's a limitation. If one feature is much stronger than the rest, every tree will split on it first. For example, if FICO score is by far the best predictor of loan default, every bagged tree will put FICO at the root. The trees end up highly correlated despite seeing different bootstrap samples. The correlation ρ stays high, and the variance reduction is limited. We need a way to de-correlate the trees further. That's exactly what Random Forests do.

3 Random Forests

3.1 Bagging Plus Feature Subsampling

Random Forests (Breiman, 2001) add one twist to bagging: at each split in each tree, only a random subset of features is considered as candidates. The algorithm is:

1. Draw a bootstrap sample
2. Grow a tree, but at each split:
   - Randomly select m_{try} features out of the p available
   - Find the best split among only those m_{try} features
3. Repeat for B trees
4. Predict by majority vote (classification) or average (regression)

This forces each tree to explore different parts of the feature space, making the trees less correlated with each other.

3.2 Choosing m_{try}

The parameter m_{try} (sometimes written `max_features`) controls how many features each split is allowed to see. Common defaults are:

Task	Default m_{try}	With $p = 16$ features
Classification	$\sqrt{p}$	4 features per split

Task	Default m_{try}	With $p = 16$ features
Regression	$p/3$	~5 features per split

Smaller m_{try} means the trees are more different from each other (lower ρ), but each individual tree is weaker. Larger m_{try} means individual trees are stronger, but they're more correlated. When $m_{\text{try}} = p$, Random Forest reduces to ordinary bagging — every feature is available at every split.

3.3 Why Feature Subsampling Helps

Imagine you have 10 features, but FICO score is the strongest predictor. Without feature subsampling, every tree splits on FICO first, producing high correlation between trees and limited variance reduction. With feature subsampling ($m_{\text{try}} = 3$), some trees don't see FICO at the first split and must use income, DTI, or other features instead. These trees discover useful patterns that FICO-dominated trees miss. Each individual tree may be slightly less accurate, but the ensemble is better because the trees are less correlated. The whole is greater than the sum of its parts.

3.4 Out-of-Bag Error

Each bootstrap sample leaves out about 36.8% of the training data. For any given observation, we can collect predictions from all the trees that didn't include it in their bootstrap sample. This gives us a free validation estimate called the **out-of-bag (OOB) error**:

1. For each training observation i, identify all trees where i was out-of-bag
2. Average (or vote) the predictions from those trees only
3. Compare to the true label
4. Average across all observations

The OOB error is approximately equivalent to leave-one-out cross-validation, but comes for free — no need for a separate validation set or a cross-validation loop.

3.5 Feature Importance

Random Forests provide two measures of which features matter most.

Impurity-based importance (the default in scikit-learn) works by summing up the total reduction in Gini impurity across all splits that use a given feature, across all trees. Features that produce larger drops in impurity are ranked as more important.

Permutation importance works differently: randomly shuffle one feature's values and measure how much the model's accuracy drops. If shuffling a feature hurts accuracy a lot, that feature was important. If shuffling barely matters, the feature was dispensable.

Permutation importance is generally more reliable, but impurity-based importance is faster and often gives similar rankings.

3.6 Random Forests in Python

Python

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification

# Generate synthetic data
X_synth, y_synth = make_classification(
    n_samples=500, n_features=10, n_informative=5,
    random_state=42
```

```
)

# Train a Random Forest
rf = RandomForestClassifier(
    n_estimators=100,      # number of trees
    max_features='sqrt',   # features per split = sqrt(p)
    oob_score=True,        # compute out-of-bag accuracy
    random_state=42
)
rf.fit(X_synth, y_synth)

print(f"OOB accuracy: {rf.oob_score_:.3f}")
print(f"Training accuracy: {rf.score(X_synth, y_synth):.3f}")
```

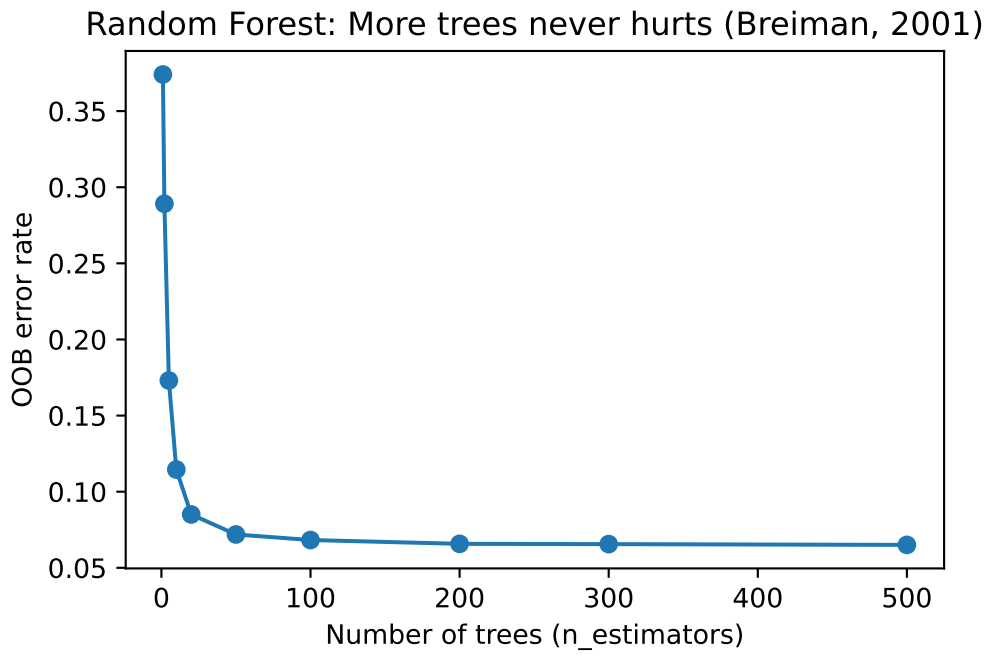
Output

```
OOB accuracy: 0.906
Training accuracy: 1.000
```

The gap between training accuracy and OOB accuracy gives us a sense of how much the model is overfitting. OOB accuracy is a realistic estimate of out-of-sample performance.

3.7 Effect of Number of Trees

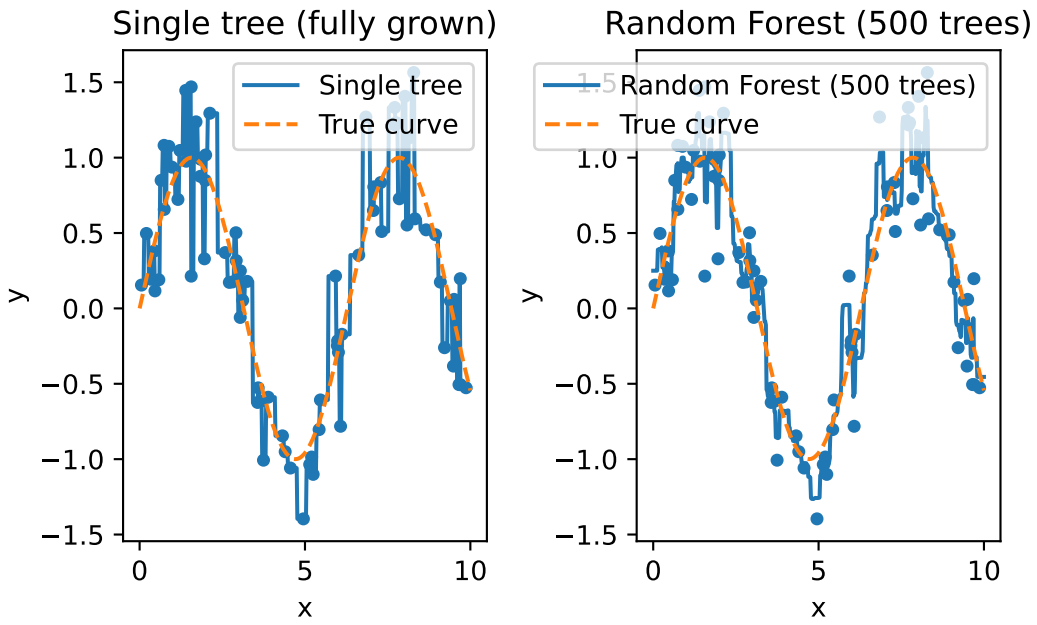
A natural question is how many trees we need. The plot below shows OOB error as a function of the number of trees in the forest.



OOB error decreases as we add more trees and eventually levels off. Unlike many models, Random Forests cannot overfit by adding more trees — more trees only means more stable averaging. The main cost is computation time.

3.8 Averaging Smooths Out Overfitting

To see the variance-reduction effect more clearly, consider a regression example. A single fully-grown tree memorizes every training point, producing a jagged step function. When we average many such trees together, the individual quirks cancel out and the forest recovers a smooth curve close to the true pattern.



The left panel shows a single tree that jumps up and down to hit each observation. The right panel shows the average of 500 such trees — the result is a smooth curve that closely tracks the true sine function. This is variance reduction in action. Each tree overfits differently, but their average does not.

3.9 Hyperparameters

The main hyperparameters for Random Forests are:

Parameter	What it controls	Typical range
`n_estimators`	Number of trees in the forest	100–1000 (more is better, but slower)
`max_features`	Features considered per split	`'sqrt'` (classification), `p/3` (regression)
`max_depth`	Maximum depth of each tree	`None` (fully grown) or 10–30
`min_samples_leaf`	Minimum observations in a leaf	1–10

In practice, the default values work well for most problems. The main tuning decision is `n_estimators` — use as many trees as your computational budget allows.

4 Boosting

4.1 Learning from Mistakes

Random Forests reduce variance by averaging many independent models. **Boosting** takes a completely different approach: build models sequentially, where each new model focuses on the mistakes made by the previous ones.

Think of it like studying for an exam. After your first practice test, you don't re-study everything equally — you focus on the questions you got wrong. After the second attempt, you focus on the remaining mistakes. Each round of studying targets your weakest areas. Boosting does the same thing with models: each new tree pays extra attention to the observations that previous trees got wrong.

4.2 Weak Learners

Boosting works with **weak learners** — models that are only slightly better than random guessing. For classification, "slightly better than random" means accuracy above 50%. The typical weak learner is a **decision stump**: a tree with just one split (depth = 1). A stump can only ask one question ("Is FICO > 700?"), so it's a very limited model on its own.

Schapire (1990) proved a remarkable theoretical result: any combination of weak learners can be boosted into a strong learner with arbitrarily high accuracy. You don't need good individual models — you just need lots of slightly-better-than-random models that each fix a different part of the problem.

4.3 AdaBoost

AdaBoost (Freund & Schapire, 1997) was the first practical boosting algorithm. It works as follows:

- **Start:** Give every training observation equal weight $w_i = 1/n$. Train a decision stump on the data.
- **Evaluate:** The stump gets some observations wrong. Compute its weighted error rate ε — how much total weight falls on misclassified points.
- **Vote weight:** Give this stump a vote weight α based on its accuracy. Better stumps get louder votes.
- **Re-weight:** Increase the weights on misclassified points so the next stump will focus on them. Decrease weights on points we already get right.
- **Repeat** for M rounds. Each round adds one stump that targets the mistakes left over from previous rounds.
- **Final prediction:** Take a weighted vote across all M stumps.

The result is that many weak models — each barely better than a coin flip — combine into a strong classifier.

More formally, each observation starts with weight $w_i = 1/n$. In each round $m = 1, 2, \ldots, M$:

1. Fit a weak learner $h_m(x)$ on the weighted training data
2. Compute the weighted error — the total weight on misclassified points:

$$\varepsilon_m = \frac{\sum_{i:\, h_m(x_i) \neq y_i} w_i}{\sum_{i=1}^{n} w_i}$$

3. Compute this stump's vote weight (Greek letter "alpha"):

$$\alpha_m = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_m}{\varepsilon_m}\right)$$

4. Update weights — multiply each misclassified observation's weight by e^{α_m} and each correct observation's weight by $e^{-\alpha_m}$, then renormalize so weights sum to 1

The final prediction is:

$$\hat{y}(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m \, h_m(x)\right)$$

AdaBoost is formulated for binary classification where the class labels are coded as $y_i \in \{-1, +1\}$. Each stump $h_m(x)$ also outputs -1 or $+1$, so the weighted sum $\sum \alpha_m h_m(x)$ is positive when the $+1$ votes outweigh the -1 votes and negative otherwise. The sign function simply reads off which class won.

4.4 Bagging vs. Boosting

It's worth pausing to compare the two ensemble strategies:

	Bagging / Random Forest	Boosting
Training	Parallel (trees are independent)	Sequential (each tree depends on previous)
Tree type	Full-sized, deep trees	Shallow trees (weak learners)
Primary benefit	Reduces **variance**	Reduces **bias** (and variance)
Overfitting risk	Low (more trees never hurts)	Higher (can overfit if too many rounds)
Sensitivity to noise	Robust	More sensitive
Computation	Easy to parallelize	Harder to parallelize

Random Forests start with strong individual models and reduce their variance by averaging. Boosting starts with weak models and gradually builds them into a strong model by correcting errors.

4.5 From AdaBoost to Gradient Boosting

AdaBoost works by re-weighting misclassified observations. **Gradient Boosting** (Friedman, 2001) generalized this idea: instead of re-weighting, fit each new tree to the **residuals** (errors) of the current model. The intuition is the same — focus on what the model gets wrong — but gradient boosting works with any loss function, not just classification error. This makes it more flexible and more powerful.

The name comes from a connection to gradient descent. Recall from Week 5 that gradient descent updates parameters by moving in the direction that reduces the loss:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

where η (Greek letter "eta") is the learning rate and ∇L is the gradient of the loss function — the direction of steepest ascent. Gradient boosting applies this same idea, but instead of updating numerical parameters, we update our prediction by adding trees one at a time. Let $F_m(x)$ be the ensemble's prediction after m rounds — a running total of every tree's contribution so far. And let $h_m(x)$ be the single new tree fit in round m — the next correction. The update rule is:

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$$

Take yesterday's prediction, and nudge it by a shrunken version of today's new tree. Each h_m is chosen to point in the direction that reduces the loss.

5 Gradient Boosting and XGBoost

5.1 The Gradient Boosting Algorithm

The full algorithm is as follows. Given training data, a loss function L, number of rounds M, and learning rate η:

1. Start with a constant prediction: $F_0(x) = \arg\min_c \sum_{i=1}^{n} L(y_i, c)$
2. For $m = 1, 2, \ldots, M$:
    a. Compute **pseudo-residuals** for each observation — how far off the current model is:

$$r_{im} = -\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}$$

b. Fit a small tree h_m to the pseudo-residuals $\{(x_i, r_{im})\}$

c. Update the model:

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$$

For squared-error loss, the pseudo-residuals are simply $r_{im} = y_i - F_{m-1}(x_i)$ — the ordinary residuals. Each new tree literally predicts what the current model is getting wrong. When the pseudo-residuals hit zero, the negative gradient of the loss is zero — the same condition as being at a minimum in ordinary calculus. There is no direction to move that would reduce the loss further, and the algorithm has converged.

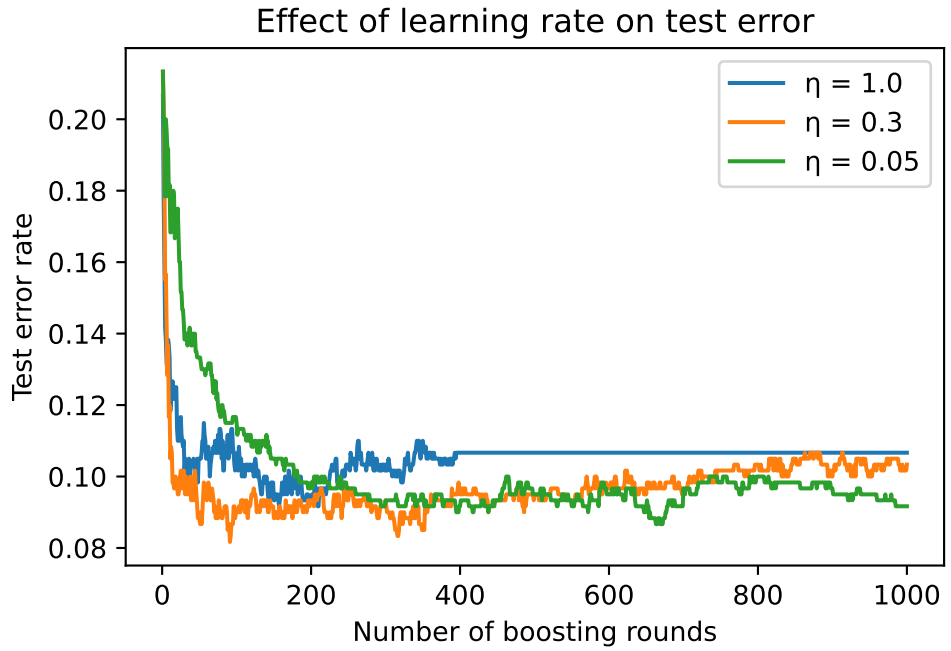
5.2 Why Shallow Trees?

In Random Forests, each tree is grown deep — the individual trees need to be strong because we're just averaging them. In gradient boosting, the strategy is the opposite: each tree should be **shallow** (typically depth 3–6). Each tree is making a small correction to the current model, not trying to solve the whole problem. Deep trees can overfit the residuals, especially in later rounds when the residuals are small. And small trees are weak learners — exactly what boosting theory requires.

This is a common source of confusion. Random Forests want each tree to be as good as possible. Gradient boosting wants each tree to be a small, cautious correction.

5.3 The Learning Rate

The learning rate η controls how much each tree contributes. Smaller η means each tree makes a smaller correction.



A large learning rate ($\eta = 1.0$) converges fast but may overshoot and overfit. A small learning rate ($\eta = 0.05$) requires more rounds but often reaches a lower error. The general rule: small η + many trees > large η + few trees.

5.4 Gradient Boosting in Python

Python

```python
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Generate data
X_gb, y_gb = make_classification(
    n_samples=500, n_features=10, n_informative=5,
    random_state=42
)
X_gb_train, X_gb_test, y_gb_train, y_gb_test = train_test_split(
    X_gb, y_gb, test_size=0.3, random_state=42
)

# Train Gradient Boosting
gb = GradientBoostingClassifier(
    n_estimators=200,      # number of boosting rounds
    learning_rate=0.1,     # step size
    max_depth=3,           # shallow trees
    random_state=42
)
gb.fit(X_gb_train, y_gb_train)

print(f"Training accuracy: {gb.score(X_gb_train, y_gb_train):.3f}")
print(f"Test accuracy: {gb.score(X_gb_test, y_gb_test):.3f}")
```

Output

```
Training accuracy: 1.000
Test accuracy: 0.920
```

5.5 XGBoost: Extreme Gradient Boosting

XGBoost (Chen & Guestrin, 2016) is an optimized implementation of gradient boosting that adds several improvements: regularization that penalizes complex trees to prevent overfitting, efficient parallelized tree construction, automatic handling of missing values (it learns the best direction to send missing values at each split), and column subsampling borrowed from Random Forests (using random feature subsets to further reduce correlation between trees).

XGBoost became famous through machine learning competitions — it won or placed highly in the majority of structured/tabular data competitions on Kaggle for years. It remains one of the strongest off-the-shelf models for tabular financial data.

5.6 The XGBoost Objective Function

XGBoost minimizes a regularized objective at each boosting round:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i,\ \hat{y}_i + h_m(x_i)) + \Omega(h_m)$$

The first term is the usual loss (how well the new tree corrects errors). The second term Ω (Greek capital letter "omega") is a regularization penalty on the tree's complexity:

$$\Omega(h_m) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

where T is the number of leaves in the tree, w_j is the prediction value in leaf j, γ (gamma) is a penalty for adding more leaves (encouraging simpler trees), and λ (lambda) is a penalty on leaf weights (preventing extreme predictions). The regularization is what distinguishes XGBoost from plain gradient boosting. It explicitly trades off prediction accuracy against model complexity.

5.7 XGBoost in Python

Python

```python
from xgboost import XGBClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Same data as before
X_xgb, y_xgb = make_classification(
    n_samples=500, n_features=10, n_informative=5,
    random_state=42
)
X_xgb_train, X_xgb_test, y_xgb_train, y_xgb_test = train_test_split(
    X_xgb, y_xgb, test_size=0.3, random_state=42
)

# Train XGBoost
xgb = XGBClassifier(
    n_estimators=200,
    learning_rate=0.1,
    max_depth=3,
    reg_lambda=1.0,        # L2 regularization
    random_state=42
)
xgb.fit(X_xgb_train, y_xgb_train)

print(f"Training accuracy: {xgb.score(X_xgb_train, y_xgb_train):.3f}")
print(f"Test accuracy: {xgb.score(X_xgb_test, y_xgb_test):.3f}")
```

Output

```
Training accuracy: 1.000
Test accuracy: 0.920
```

5.8 XGBoost Hyperparameters

Parameter	What it controls	Typical range
`n_estimators`	Number of boosting rounds	100–1000
`learning_rate`	Step size per round (η)	0.01–0.3
`max_depth`	Depth of each tree	3–8
`reg_lambda`	L2 regularization on leaf weights (λ)	0–10
`subsample`	Fraction of observations per tree	0.5–1.0
`colsample_bytree`	Fraction of features per tree	0.5–1.0

The `subsample` and `colsample_bytree` parameters borrow ideas from bagging and Random Forests — randomly omitting observations and features adds noise that helps prevent overfitting.

5.9 When to Use Which?

Random Forests are a safe default that's hard to get wrong. They're robust to hyperparameter choices (defaults usually work well), easy to parallelize across cores, and less prone to overfitting. Gradient Boosting and XGBoost have a higher ceiling with careful tuning and often win competitions and benchmarks on tabular data, but they require more hyperparameter tuning (learning rate, depth, regularization) and can overfit if not regularized properly.

In practice, start with a Random Forest to get a solid baseline. If you need better performance and have time to tune, try XGBoost.

6 Summary

Ensemble methods fix the high-variance problem of individual decision trees by combining many trees together. The two main strategies are parallel (Random Forests) and sequential (Boosting).

Random Forests use bagging plus random feature subsampling at each split. They reduce variance by de-correlating trees. More trees never hurts, the default hyperparameters are robust, and we get free validation via OOB error.

Gradient Boosting and XGBoost take the sequential approach, where each tree corrects the residuals of the previous model. They reduce bias (and variance through regularization). The recipe is shallow trees, a small learning rate, and many rounds. XGBoost adds explicit regularization and computational efficiency.

7 References

- Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2), 123–140.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.
- Chen, T. & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of KDD*, 785–794.
- Freund, Y. & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1), 119–139.
- Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5), 1189–1232.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd ed.). Springer.