Week 6: Tree Ensembles

for Portfolio Optimization

MSc Banking and Finance – FinTech Course

October 14, 2025

Abstract

This document provides comprehensive coverage of tree-based ensemble methods applied to portfolio optimization and financial forecasting. Building on Week 5's regularized linear models, we introduce non-linear ensemble techniques including Random Forest, Gradient Boosting, and XGBoost. Topics include decision tree mechanics, bagging vs boosting, hyperparameter tuning, feature importance analysis, SHAP interpretability, and practical applications in portfolio management.

Course: Big Data & Statistical Learning
for Finance

Contents

1 Introduction & Motivation

1.1 Why Tree Ensembles?

Linear models (Week 5) assume relationships are linear:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_p x_p \tag{1}$$

Key Point

Reality in finance:

- Non-linear interactions (volatility × momentum)
- Regime changes (bull vs bear markets)
- Threshold effects (volatility > 30% → different behavior)

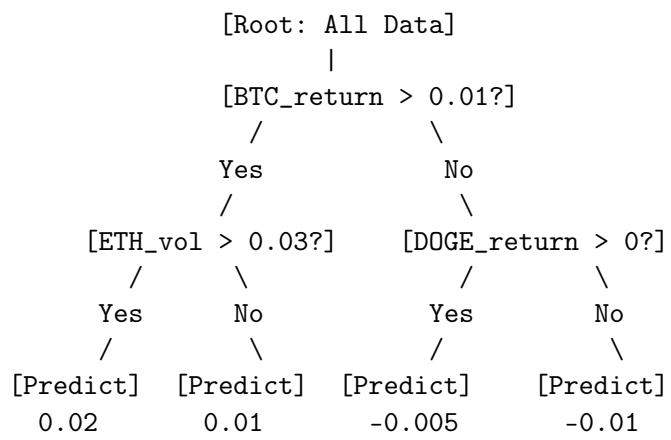
Tree ensembles excel at:

1. Capturing non-linear patterns automatically
2. Handling feature interactions without manual engineering
3. Robust to outliers and mixed-scale features
4. No standardization required (scale-invariant)

2 Decision Trees: Foundation

2.1 How Decision Trees Work

A decision tree recursively splits data based on feature thresholds. The tree structure can be represented as:



2.2 Splitting Criterion

For **regression** (predicting returns), we minimize **Mean Squared Error (MSE)**:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{2}$$

Algorithm 1 Best Split Selection

Input: Dataset $\{(x_i, y_i)\}_{i=1}^n$, features $\{f_1, \ldots, f_p\}$
Initialize best_split $\leftarrow \emptyset$, min_MSE $\leftarrow \infty$
for each feature $j \in \{1, \ldots, p\}$ **do**
  for each threshold t in x_j **do**
    Split data: $D_L = \{i : x_{ij} \leq t\}$, $D_R = \{i : x_{ij} > t\}$
    Compute MSE_L on D_L and MSE_R on D_R
    $\text{MSE}_{\text{total}} = \frac{|D_L|}{n}\text{MSE}_L + \frac{|D_R|}{n}\text{MSE}_R$
    if $\text{MSE}_{\text{total}} <$ min_MSE **then**
      best_split $\leftarrow (j, t)$
      min_MSE $\leftarrow \text{MSE}_{\text{total}}$
    end if
  end for
end for
return best_split

Advantages	Disadvantages
Captures non-linearities	High variance
No feature scaling needed	Overfitting prone
Handles mixed data types	Unstable predictions
Interpretable (visualizable)	Sensitive to data changes

Table 1: Single Decision Tree: Pros and Cons

2.3 Tree Advantages and Disadvantages

Caution

Solution: Use ensemble methods to reduce variance and improve stability!

3 The Bias-Variance Tradeoff

3.1 Decomposition

The expected prediction error can be decomposed as:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma_\epsilon^2 \tag{3}$$

where:

- **Bias**: Error from wrong assumptions (underfitting)
- **Variance**: Error from sensitivity to training data (overfitting)
- σ_ϵ^2: Irreducible error (noise)

Definition 3.1 (Bias). *The bias of an estimator $\hat{f}$ is defined as:*

$$\textit{Bias}[\hat{f}(x)] = \mathbb{E}[\hat{f}(x)] - f(x) \tag{4}$$

where $f(x)$ is the true function.

Definition 3.2 (Variance). *The variance of an estimator $\hat{f}$ is:*

$$\textit{Var}[\hat{f}(x)] = \mathbb{E}[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2] \tag{5}$$

3.2 Single Tree Problem

Intuition

Single Decision Tree:

- Low bias (can fit complex patterns)
- **High variance** (changes drastically with data)

3.3 Ensemble Solution

Two main approaches to reduce error:

1. **Bagging** (Bootstrap Aggregating) → Reduces variance
2. **Boosting** (Sequential Learning) → Reduces bias & variance

4 Random Forest (Bagging)

4.1 Bootstrap Aggregating (Bagging)

Core Idea: Train many trees on different random subsets of data, then average predictions.

Algorithm 2 Random Forest Algorithm

Input: Dataset $D = \{(x_i, y_i)\}_{i=1}^n$, number of trees B
Parameter: Number of features to consider at each split $m < p$
for $b = 1$ to B **do**
    Draw bootstrap sample D_b from D (sample with replacement)
    Train decision tree T_b on D_b with:
        At each split, randomly select m features from p
        Choose best split among the m features
        Grow tree to maximum depth (no pruning)
end for
Output: Ensemble prediction

$$\hat{y}_{\text{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x) \tag{6}$$

4.2 Why It Works: Variance Reduction

Theorem 4.1 (Variance Reduction through Averaging). *If we have B independent and identically distributed random variables $Z_1, \ldots, Z_B$ each with variance σ^2, then the variance of their average is:*

$$Var\left(\frac{1}{B}\sum_{b=1}^{B} Z_b\right) = \frac{\sigma^2}{B} \tag{7}$$

As $B \to \infty$, variance $\to 0$.

Key Point

Random feature selection at each split ensures trees are less correlated, making averaging more effective.

For correlated trees with correlation ρ:

$$\text{Var}\left(\text{average}\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{8}$$

As $B \to \infty$: Var $\to \rho\sigma^2$. Thus, reducing correlation (ρ) is crucial!

4.3 Key Hyperparameters

Parameter	Description	Typical Values
`n_estimators`	Number of trees B	50–500
`max_depth`	Maximum tree depth	5–20 or None
`min_samples_split`	Min samples to split node	2–20
`min_samples_leaf`	Min samples in leaf	1–10
`max_features`	Features per split m	`'sqrt'`, `'log2'`
`bootstrap`	Use bootstrap sampling	`True`

Table 2: Random Forest Hyperparameters

```python
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(
    n_estimators=100,       # Number of trees
    max_depth=10,           # Maximum depth
    min_samples_split=5,    # Min to split
    min_samples_leaf=2,     # Min in leaf
    max_features='sqrt',    # sqrt(p) features
    random_state=42
)

rf.fit(X_train, y_train)
```

Listing 1: Random Forest in Python

4.4 Out-of-Bag (OOB) Score

Free validation! For each tree, approximately 37% of data is **out-of-bag** (not in bootstrap sample).

$$P(\text{observation } i \notin \text{ bootstrap sample}) = \left(1 - \frac{1}{n}\right)^n \approx e^{-1} \approx 0.368 \tag{9}$$

Use OOB samples for validation:

$$\text{OOB Error} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i^{\text{OOB}}\right)^2 \tag{10}$$

where $\hat{y}_i^{\text{OOB}}$ is the average prediction from trees that did not include observation i in training.

5 Gradient Boosting (Sequential Learning)

5.1 Boosting Philosophy

Key Point

Unlike bagging (parallel), boosting trains trees **sequentially**:

- Each new tree corrects mistakes of previous trees
- Focus on "hard" examples (large residuals)
- Build additive model: $f(x) = \sum_{m=1}^{M} \eta h_m(x)$

5.2 Gradient Boosting Algorithm

Algorithm 3 Gradient Boosting for Regression

Input: Dataset $\{(x_i, y_i)\}_{i=1}^n$, learning rate η, # iterations M
Initialize with constant: $F_0(x) = \text{argmin}_\gamma \sum_{i=1}^n L(y_i, \gamma)$
for $m = 1$ to M **do**
    Step 1: Compute pseudo-residuals

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}} \tag{11}$$

    For squared loss: $r_{im} = y_i - F_{m-1}(x_i)$
    Step 2: Fit base learner $h_m(x)$ to residuals $\{(x_i, r_{im})\}_{i=1}^n$
    Step 3: Update model

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x) \tag{12}$$

end for
Output: Final model $F_M(x)$

5.3 Mathematical Justification

Gradient boosting performs **gradient descent in function space**:

$$F_m = F_{m-1} - \eta \nabla_F L(y, F_{m-1}) \tag{13}$$

where the gradient is approximated by fitting a tree to the negative gradient (residuals).

Parameter	Description	Typical Values
`n_estimators`	Boosting stages M	100–1000
`learning_rate`	Shrinkage η	0.01–0.1
`max_depth`	Tree depth	3–7 (shallow!)
`subsample`	Row sampling	0.5–1.0
`min_samples_split`	Min to split	2–20

Table 3: Gradient Boosting Hyperparameters

5.4 Key Hyperparameters

Intuition

Tuning Tips:

- Lower `learning_rate` + more `n_estimators` → Better (but slower)
- Typical: `learning_rate=0.01` with `n_estimators=1000`
- Keep trees shallow (`max_depth=3-7`)
- `subsample<1.0` adds randomness (like bagging)

```python
from sklearn.ensemble import GradientBoostingRegressor

gb = GradientBoostingRegressor(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    min_samples_split=2,
    subsample=0.8,
    random_state=42
)

gb.fit(X_train, y_train)
```

Listing 2: Gradient Boosting in Python

6 XGBoost: State-of-the-Art

6.1 What is XGBoost?

eXtreme Gradient Boosting – Enhanced gradient boosting with:

- Regularized objective function
- Optimized for speed and performance
- Parallel tree construction
- Built-in cross-validation
- Handles missing values automatically

6.2 XGBoost Objective

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{14}$$

where:

- $\ell(y_i, \hat{y}_i)$: Loss function (e.g., MSE)
- $\Omega(f_k)$: Regularization penalty for tree k

Regularization penalty:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{15}$$

where:

- T = number of leaves
- w_j = leaf weights
- γ = complexity penalty (L0-like regularization in linear models penalizes the number of non-zero coefficients. Its goal is to create a sparse model by forcing less important feature coefficients to become exactly zero.)
- λ = L2 regularization (like Ridge!)

Key Point

XGBoost combines:

- Gradient Boosting's sequential learning
- Random Forest's column sampling
- Ridge/Lasso regularization (γ, λ, α)

6.3 Key Hyperparameters

Parameter	Description	Typical Values
`n_estimators`	Boosting rounds	100–1000
`learning_rate`	Shrinkage (eta)	0.01–0.3
`max_depth`	Tree depth	3–10
`min_child_weight`	Min sum of weights in leaf	1–10
`subsample`	Row sampling	0.5–1.0
`colsample_bytree`	Column sampling	0.5–1.0
`reg_alpha`	L1 regularization	0–1
`reg_lambda`	L2 regularization	0–2
`gamma`	Min loss reduction	0–5

Table 4: XGBoost Hyperparameters

```
import xgboost as xgb

xgb_model = xgb.XGBRegressor(
    n_estimators=100,
```

```
    learning_rate=0.1,
    max_depth=6,
    min_child_weight=1,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_alpha=0.1,        # L1 regularization
    reg_lambda=1.0,       # L2 regularization
    random_state=42
)

xgb_model.fit(X_train, y_train)
```

Listing 3: XGBoost in Python

7 Feature Importance & Interpretability

7.1 Built-in Feature Importance

Three common methods:

1. Impurity-based (Gini, MSE):

$$\text{Importance}(x_j) = \sum_{t \in \text{trees}} \sum_{s \in \text{splits on } x_j} \Delta\text{MSE}(s) \tag{16}$$

Average improvement in MSE from splits on feature x_j.

2. Permutation importance:

1. Train model on original data
2. For each feature j:
   - Shuffle feature j values
   - Measure drop in performance
3. Importance = performance drop

3. Drop-column importance:

1. Train model with all features
2. For each feature j:
   - Retrain model without feature j
   - Measure drop in performance
3. Importance = performance drop

Caution

Impurity-based importance can be biased toward high-cardinality features. Permutation importance is more reliable.

8 SHAP Values (Explainable AI)

8.1 What is SHAP?

SHAP = **SH**apley **A**dditive ex**P**lanations

Based on **Shapley values** from cooperative game theory:

- How much does each feature "contribute" to a prediction?
- Fairly distribute prediction among features

8.2 Mathematical Definition

For a prediction $f(x)$, we want to decompose:

$$f(x) = \phi_0 + \sum_{j=1}^{p} \phi_j \tag{17}$$

where:

- $\phi_0 = \mathbb{E}[f(X)]$ (expected prediction)
- ϕ_j = SHAP value for feature j

Definition 8.1 (Shapley Value)**.** *The Shapley value for feature j is:*

$$\phi_j = \sum_{S \subseteq \{1,\ldots,p\} \setminus \{j\}} \frac{|S|!(p - |S| - 1)!}{p!} [f_{S \cup \{j\}}(x_{S \cup \{j\}}) - f_S(x_S)] \tag{18}$$

where S is a subset of features excluding j.

8.3 SHAP Properties

SHAP values satisfy **fairness axioms**:

1. **Efficiency:** $\sum_{j=1}^{p} \phi_j = f(x) - \mathbb{E}[f(X)]$
2. **Symmetry:** If features i and j contribute equally, then $\phi_i = \phi_j$
3. **Dummy:** If feature j doesn't affect prediction, then $\phi_j = 0$
4. **Additivity:** For ensemble $f = f_1 + f_2$: $\phi_j^f = \phi_j^{f_1} + \phi_j^{f_2}$

8.4 TreeSHAP Algorithm

Efficient computation for tree ensembles!

Instead of exponential $O(2^p)$ coalitions, TreeSHAP:

- Traces all paths in tree
- Computes exact Shapley values
- Polynomial time complexity: $O(TLD^2)$
  - T = number of trees
  - L = max leaves per tree
  - D = max depth

```python
import shap

# Create explainer
explainer = shap.TreeExplainer(model)

# Compute SHAP values
shap_values = explainer.shap_values(X)

# Visualize
shap.summary_plot(shap_values, X, feature_names)
```

Listing 4: SHAP Analysis in Python

9 Portfolio Application

9.1 Pipeline: Week 4 → Week 5 → Week 6

Week 4 Output:
ARIMA forecasts (returns)
GARCH forecasts (volatility)
↓
Week 5 Feature Engineering:
Risk-adjusted returns
Momentum indicators
Volatility regimes
Cross-asset features
↓
Week 5 Linear Models:
Ridge: All features
Lasso: Feature selection
↓
Week 6 Tree Ensembles:
Random Forest: Non-linear patterns
Gradient Boosting: Sequential learning
XGBoost: State-of-the-art
↓
Portfolio Optimization:
Optimal-Allocation of weights

9.2 Why Trees for Portfolio Optimization?

1. Capture regime changes: Trees naturally learn decision rules like:

```
if btc_vol > 0.03:
    if btc_return > 0:
        weight_btc = 0.6  # High vol + positive return
    else:
        weight_btc = 0.2  # High vol + negative return
else:
    weight_btc = 0.4      # Low vol
```

2. Feature interactions:

- Linear model: $\hat{y} = w_1 \cdot \text{sharpe} + w_2 \cdot \text{momentum}$

- Tree model: $\hat{y} = f(\text{sharpe} \times \text{momentum})$

Trees capture synergies automatically!

9.3 Using Tree Predictions for Weights

Optimization Process:

1. Train tree ensemble to predict portfolio return
2. Extract asset-specific predictions or features
3. Optimize weights using softmax:

$$w_i = \frac{\exp(\text{Sharpe}_i)}{\sum_j \exp(\text{Sharpe}_j)} \tag{19}$$

4. Constraints:

$$\sum_i w_i = 1 \tag{20}$$

$$w_i \geq 0 \quad \forall i \quad \text{(no short selling)} \tag{21}$$

10 Hyperparameter Tuning Strategies

10.1 Grid Search

Exhaustive search over parameter grid:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15],
    'learning_rate': [0.01, 0.1]
}

grid_search = GridSearchCV(
    model, param_grid, cv=5,
    scoring='neg_mean_squared_error'
)

grid_search.fit(X_train, y_train)
best_model = grid_search.best_estimator_
```

Listing 5: GridSearchCV Example

Complexity: Tests all $\prod_i |P_i|$ combinations where P_i is parameter i's values.

10.2 Random Search

Random sampling from parameter distributions:

```
from sklearn.model_selection import RandomizedSearchCV

param_dist = {
    'n_estimators': [50, 100, 200, 300],
    'max_depth': range(3, 20),
    'learning_rate': [0.001, 0.01, 0.1, 0.5]
}
```

```
random_search = RandomizedSearchCV(
    model, param_dist, n_iter=50, cv=5
)
```

Listing 6: RandomizedSearchCV Example

More efficient for large parameter spaces!

10.3 Practical Tips

1. **Start coarse, then refine:** Wide grid → identify range → narrow grid
2. **Use logarithmic scales:** For learning rate: [0.001, 0.01, 0.1, 1.0]
3. **Monitor overfitting:** Check train vs validation gap
4. **Parallelize:** Use `n_jobs=-1` for all CPU cores

11 Practical Examples

11.1 Example 1: Single Tree vs Random Forest

```
# Single tree: High variance
tree = DecisionTreeRegressor(max_depth=10)
cv_scores = cross_val_score(tree, X, y, cv=5)
print(f"Tree CV R^2: {cv_scores.mean():.3f} +/- {cv_scores.std():.3f}")
# Output: 0.65 +/- 0.15 (high std!)

# Random Forest: Low variance
rf = RandomForestRegressor(n_estimators=100, max_depth=10)
cv_scores = cross_val_score(rf, X, y, cv=5)
print(f"RF CV R^2: {cv_scores.mean():.3f} +/- {cv_scores.std():.3f}")
# Output: 0.82 +/- 0.03 (low std!)
```

Listing 7: Variance Comparison

11.2 Example 2: XGBoost Regularization

```
# No regularization: Overfits
xgb1 = XGBRegressor(reg_alpha=0, reg_lambda=0)
xgb1.fit(X_train, y_train)
print(f"Train R^2: {xgb1.score(X_train, y_train):.3f}")  # 0.95
print(f"Test R^2: {xgb1.score(X_test, y_test):.3f}")    # 0.70

# With regularization: Generalizes
xgb2 = XGBRegressor(reg_alpha=0.1, reg_lambda=1.5)
xgb2.fit(X_train, y_train)
print(f"Train R^2: {xgb2.score(X_train, y_train):.3f}")  # 0.88
print(f"Test R^2: {xgb2.score(X_test, y_test):.3f}")    # 0.85
```

Listing 8: Regularization Impact

12 Key Takeaways

1. **Decision trees** are intuitive but suffer from high variance
2. **Random Forest** (bagging) reduces variance through averaging decorrelated trees

3. **Gradient Boosting** learns sequentially from residual errors
4. **XGBoost** adds regularization and computational optimizations
5. **No feature scaling needed** – trees are scale-invariant
6. **Feature importance** guides understanding, but use **SHAP** for causal interpretation
7. **Hyperparameter tuning** is crucial for optimal performance
8. **Cross-validation** prevents overfitting and ensures generalization
9. **Trees excel at non-linear patterns** that linear models miss
10. **Ensembles $\gg$ single models** for real-world finance

13 Further Reading

13.1 Books

1. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd ed.). Springer. Chapters 10, 15, 16.
2. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023). *An Introduction to Statistical Learning with Applications in Python* (2nd ed.). Springer. Chapter 8.

13.2 Papers

1. Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.
2. Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5), 1189–1232.
3. Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD* (pp. 785–794).
4. Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems* (pp. 4765–4774).

13.3 Applications in Finance

1. Gu, S., Kelly, B., & Xiu, D. (2020). Empirical asset pricing via machine learning. *Review of Financial Studies*, 33(5), 2223–2273.
2. Rasekhschaffe, K. C., & Jones, R. C. (2019). Machine learning for stock selection. *Financial Analysts Journal*, 75(3), 70–88.

14 Connection to Previous Weeks

15 Pro Tips for Students

1. **Start simple:** Train single tree $\rightarrow$ Understand behavior $\rightarrow$ Add ensemble
2. **Monitor overfitting:** Always compare train vs test metrics
3. **Feature engineering still matters:** Good features help even the best models

Week	Topic	Connection to Week 6
1	Data Collection	Raw price data → Features
2	Data Cleaning	Handle missing values in tree inputs
3	Econometrics	Factor models = features for trees
4	Time Series	ARIMA/GARCH forecasts = tree inputs
5	ML Regularization	Ridge/Lasso = linear baseline Trees = non-linear extension
6	**Tree Ensembles**	**Capture interactions & non-linearities**
7	Dimensionality Reduction	PCA features → Trees (next week!)

Table 5: Course Integration: Week 6 Context

4. **Tune hyperparameters:** Default values rarely optimal for finance
5. **Use cross-validation:** Never trust single train/test split
6. **Interpret with SHAP:** Don't treat trees as black boxes
7. **Compare baselines:** Trees should beat Ridge/Lasso significantly
8. **Ensemble diversity:** RF for variance reduction, GBM for bias reduction
9. **Regularize XGBoost:** Always use `reg_alpha` and `reg_lambda` > 0
10. **Domain knowledge:** Combine ML predictions with financial intuition

---

Next Week: Dimensionality Reduction

PCA, Factor Models, Clustering for extracting latent patterns from high-dimensional financial data!
