Week 6: Tree Ensembles for Portfolio Optimization

Big Data & Statistical Learning for Finance

MSc Banking and Finance – FinTech Course

A.U.E.B.

October 14, 2025

Outline

Why Should You Care About Trees?

The Problem with Linear Models

Linear models assume relationships are **straight lines**:

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

Real Life is Complicated!

- Non-linear interactions
- Regime changes (bull vs. bear)
- Threshold effects

Trees Excel At:

- Capturing non-linearities
- Feature interactions
- Robust to outliers
- No scaling needed!

What Makes Finance Special?

Financial Markets Are Complex

- **Regime Changes**: Markets behave differently when volatility is high vs. low
- **Interactions**: Good returns + low risk = great, but good returns + high risk = scary
- **Non-linear Relationships**: What works in bull markets fails in bear markets

Trees naturally learn these complex rules!

How Decision Trees Work

Like Playing 20 Questions!

A decision tree recursively splits data based on feature thresholds.

Example: Predicting Bitcoin Returns

- Is volatility $> 3\%$?
  - Yes: Risky! $\rightarrow$ Predict -2%
  - No: Safe! $\rightarrow$ Predict $+3\%$

Splitting Criterion (MSE):

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

Choose split that minimizes weighted MSE:

$$\text{MSE}_{\text{total}} = \frac{|D_L|}{n}\text{MSE}_L + \frac{|D_R|}{n}\text{MSE}_R$$

Single Tree: Pros and Cons

Advantages

- Captures non-linearities
- No feature scaling needed
- Handles mixed data types
- Interpretable (visualizable)

Disadvantages

- **High variance**
- Overfitting prone
- Unstable predictions
- Sensitive to data changes

Solution: Use Ensembles!

Bias-Variance Decomposition

Expected Prediction Error

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{\text{Bias}[\hat{f}(x)]^2}_{\text{Underfitting}} + \underbrace{\text{Var}[\hat{f}(x)]}_{\text{Overfitting}} + \underbrace{\sigma_\epsilon^2}_{\text{Irreducible}}$$

Bias (Systematic Error):

- Model too simple
- Missing patterns
- Always wrong in same way

Variance (Random Error):

- Model too complex
- Overfits training data
- Unpredictable on new data

The Goldilocks Principle

Model Type	**Bias**	**Variance**
Simple (Linear)	High	Low
Single Tree	Low	High
Ensemble of Trees	Low	Low

Key Insight

Ensembles = Many trees working together = Best of both worlds!

Wisdom of the Crowd

Think of it Like This...

Scenario: Estimate how many jellybeans are in a jar.

- Bad Strategy: Ask one person (might be way off)
- Good Strategy: Ask 100 people and average their guesses

Result: The average is usually very close to the true number!

Random Forest = Wisdom of the Crowd for Trees

1. Train many trees on different random data samples
2. At each split, only consider a random subset of features
3. Average all predictions

Random Forest Algorithm

Algorithm 1 Random Forest

```
1: Input: Dataset $D$, number of trees $B$
2: Parameter: Features per split $m < p$
3: for $b = 1$ to $B$ do
4:     Draw bootstrap sample $D_b$ from $D$
5:     Train tree $T_b$ on $D_b$ where:
6:         At each split, randomly select $m$ features
7:         Choose best split among the $m$ features
8: end for
9: Output: $\hat{y}_{\text{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$
```

Why Random Forest Works Better

Variance Reduction Through Averaging

For B independent predictions with variance σ^2:

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

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

The Magic of Randomness

- **Bootstrap sampling**: Each tree sees different data
- **Feature sampling**: Trees make different mistakes
- **Averaging**: Mistakes cancel out!

Mathematical fact: Averaging 100 independent predictions reduces error by $\approx$ 10-fold!

Key Hyperparameters

Parameter	Description	Typical Values
`n_estimators`	Number of trees	50–500
`max_depth`	Tree depth	10–20 or None
`min_samples_split`	Min samples to split	2–20
`max_features`	Features per split	'sqrt', 'log2'
`bootstrap`	Use bootstrap sampling	True

Free Validation Trick: Out-of-Bag (OOB) Score

About 1/3 of data isn't used by each tree $\implies$ use for validation!

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

Learning from Mistakes

Random Forest

- Trees trained **in parallel**
- Independent of each other
- Average predictions

Gradient Boosting

- Trees trained **sequentially**
- Each fixes previous mistakes
- Additive model

Basketball Free Throw Analogy

1. **Attempt 1:** Miss – ball hits front of rim
2. **Lesson:** Shoot with more power
3. **Attempt 2:** Miss – now it's too far (back of rim)
4. **Lesson:** Reduce power slightly
5. **Attempt 3:** Adjust... closer!

Gradient Boosting Process

Step-by-Step

1. **Start simple**: Begin with average prediction
2. **Find mistakes**: Calculate residuals (errors)

$$r_i = y_i - \hat{y}_i$$

3. **Train tree on mistakes**: Build small tree to predict errors
4. **Update model**: Add correction

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

   where η is the learning rate
5. **Repeat**: Go back to step 2

Mathematical Justification

Gradient Descent in Function Space

Minimize loss $L(y, F(x))$ by iteratively moving in the negative gradient direction:

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

For squared loss: $L(y, \hat{y}) = (y - \hat{y})^2$

$$-\frac{\partial L}{\partial \hat{y}} = y - \hat{y} = \text{residual}$$

Key Insight

Training a tree on residuals $\approx$ Following negative gradient!

Important Settings

Parameter	Description	Typical Values
`n_estimators`	Boosting rounds	100–1000
`learning_rate`	Shrinkage η	0.01–0.1
`max_depth`	Tree depth	3–7 (shallow!)
`subsample`	Row sampling	0.5–1.0

Tuning Tips

- Lower `learning_rate` + more `n_estimators` $\implies$ Better (but slower)
- Typical: `learning_rate=0.01` with `n_estimators=1000`
- Keep trees shallow – each tree only fixes small part of problem
- `subsample < 1.0` adds randomness (like bagging)

What Makes XGBoost Special?

eXtreme Gradient Boosting

Enhanced gradient boosting with three superpowers:

1. **Built-in regularization** – Prevents overfitting automatically
2. **Optimized for speed** – Uses computer tricks to run fast
3. **Smart missing data handling** – Doesn't break with missing values

Why XGBoost Dominates Competitions

XGBoost is the "Swiss Army knife" of machine learning:

- Random Forest's column sampling (randomness)
- Gradient Boosting's sequential learning
- Ridge/Lasso regularization
- Plus computational tricks (GPU support, parallel processing)

XGBoost Objective Function

Regularized Objective

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

where $\Omega(f)$ is the regularization penalty:

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

- T = number of leaves
- w_j = leaf weights
- γ = complexity penalty (like L0 regularization)
- λ = L2 regularization (like Ridge!)

Effect: Forces trees to stay simple and learn only important patterns.

XGBoost in Python

```python
import xgboost as xgb

xgb_model = xgb.XGBRegressor(
    n_estimators=100,          # Boosting rounds
    learning_rate=0.1,         # Shrinkage
    max_depth=6,               # Tree depth
    min_child_weight=1,        # Min sum of weights in leaf
    subsample=0.8,             # Row sampling
    colsample_bytree=0.8,      # Column sampling
    reg_alpha=0.1,             # L1 regularization
    reg_lambda=1.0,            # L2 regularization
    gamma=0,                   # Min loss reduction
    random_state=42
)

xgb_model.fit(X_train, y_train)
predictions = xgb_model.predict(X_test)
```

Understanding What Your Model Learned

Why This Matters

Just because your model works doesn't mean you should trust it blindly!

You need to understand:

- Which features actually matter?
- Is the model using real patterns or random noise?
- Can you explain decisions to your boss/client?

Three Methods:

1. **Impurity-based**: Count splits (fast but biased)
2. **Permutation importance**: Shuffle feature and measure drop (reliable)
3. **SHAP values**: Fair attribution based on game theory (best!)

Permutation Importance

The Cookie Baking Analogy

Want to know if sugar is important?

1. Bake cookies normally $\rightarrow$ Taste: $90/100$
2. Randomly replace sugar amounts $\rightarrow$ Taste: $20/100$
3. **Conclusion:** Sugar is VERY important (70 point drop!)

Algorithm

For each feature j:

1. Measure model performance on original data
2. Randomly shuffle feature j values
3. Measure new performance
4. Importance $=$ Performance drop

Bigger drop $=$ More important feature!

SHAP: Shapley Additive exPlanations

What is SHAP?

Based on **Shapley values** from game theory – fairly divides prediction among features.

Decompose prediction:

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

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

Example: Predicting Bitcoin Return

- Base prediction: $+1.5\%$ (average)
- Volatility is high today: -0.8% (pushes down)
- Momentum is positive: $+1.2\%$ (pushes up)
- **Final prediction:** $1.5 - 0.8 + 1.2 = +1.9\%$

SHAP Properties

Fairness Axioms

SHAP values satisfy:

1. **Efficiency**: $\sum_{j=1}^{p} \phi_j = f(x) - \mathbb{E}[f(X)]$
2. **Symmetry**: Equal contributions $\implies$ equal SHAP values
3. **Dummy**: Irrelevant feature $\implies$ SHAP value $= 0$
4. **Additivity**: For ensemble $f = f_1 + f_2$: $\phi_j^f = \phi_j^{f_1} + \phi_j^{f_2}$

TreeSHAP Algorithm

Efficient computation for tree ensembles:

- Exact Shapley values (not approximations!)
- Polynomial time: $O(TLD^2)$ instead of exponential $O(2^p)$
- Works for Random Forest, Gradient Boosting, XGBoost

Why SHAP is Powerful for Finance

Explainability is Crucial in Finance

- **Regulators**: Want to know why you made a trading decision
- **Clients**: Need to understand recommendations
- **Risk Management**: Must identify what drives predictions
- **Debugging**: Find if model learned something wrong

Example Statement

"The model recommended buying Bitcoin because momentum is strong (+1.2%), despite higher volatility (-0.8%)."

The Journey So Far

$$\begin{tikzcd}
\textbf{Week 4} \rightarrow \text{ARIMA/GARCH forecasts} \arrow[d] \\
\textbf{Week 5} \rightarrow \text{Feature engineering + Ridge/Lasso} \arrow[d] \\
\textbf{Week 6} \rightarrow \text{Tree ensembles (non-linear patterns)} \arrow[d] \\
\textbf{Portfolio Optimization} \rightarrow \text{Optimal weight allocation}
\end{tikzcd}$$

Why Trees Are Perfect for Portfolios

- **Capture regime changes**: Bull vs. bear markets
- **Risk-return balance**: High return + low vol = great investment
- **Feature interactions**: Volatility × Momentum effects
- **Threshold effects**: Different behavior at vol > 30%

Using Tree Predictions for Portfolio Weights

Optimization Process

1. Train tree ensemble to predict asset returns and extract asset-specific predictions
2. Optimize weights using Sharpe ratios:

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

3. Apply constraints:
   - $\sum_i w_i = 1$ (invest all money) and $w_i \geq 0$ (no short selling)

Example Allocation

- Bitcoin: $+2\%$ return, 3% volatility $\rightarrow$ 35% weight
- Ethereum: $+1.5\%$ return, 2% volatility $\rightarrow$ 40% weight
- Dogecoin: $+3\%$ return, 5% volatility $\rightarrow$ 25% weight

What Are Hyperparameters?

The Baking Analogy

Think of training a model like baking a cake:

- **Ingredients**: Your data (can't change)
- **Recipe**: Your algorithm (Random Forest, XGBoost)
- **Oven settings**: Your hyperparameters (temperature, time)

Just like 350°F for 30 min might be perfect, while 500°F for 10 min burns everything, **the right hyperparameters make a huge difference!**

Grid Search vs Random Search

Grid Search

Strategy: Test all combinations

- `n_estimators`: [50, 100, 200]
- `max_depth`: [5, 10, 15]
- `learning_rate`: [0.01, 0.1]

Total: $3 \times 3 \times 2 = 18$ models

Pro: Guaranteed to find best

Con: Slow for many parameters

Random Search

Strategy: Random sampling

- Try 50–100 random combinations
- More efficient!

Surprising Result: Often works as well as grid search but much faster!

Why? Usually only 2–3 hyperparameters really matter.

Practical Tuning Tips

Smart Tuning Strategy

Round 1 – Coarse Search:

- Try wide range: depths [3, 10, 20], trees [50, 200, 500]
- Find general area: "Ah, depth around 10 seems best"

Round 2 – Fine Search:

- Narrow down: depths [7, 10, 13], trees [150, 200, 250]
- Find exact sweet spot

Key Rules

1. Always use cross-validation
2. Start with default values, then adjust
3. Don't spend hours tuning – diminishing returns!
4. More trees = almost always better (until you run out of time)

Don’t Trust Training Scores!

Common Mistake

- Train model on January–October data
- Test on **same** January–October data
- Get 95% accuracy!
- **Problem:** Model memorized the answers!

This is like giving students the exam questions beforehand – of course they do well!

Solution: Cross-Validation

1. Split data into 5 equal parts (folds)
2. Train on 4 folds, test on 1 fold
3. Repeat 5 times (each fold used as test once)
4. Average all 5 test scores $\rightarrow$ True performance!

Comparing Random Forest vs XGBoost

Example: 5-Fold Cross-Validation Results

	Random Forest	**XGBoost**
Fold 1	82%	87%
Fold 2	85%	88%
Fold 3	83%	86%
Fold 4	84%	87%
Fold 5	81%	87%
Average	83% $\pm$ 1.5%	87% $\pm$ 0.7%

Winner: XGBoost (higher score AND more consistent!)

Statistical Testing: Use t-test to verify difference is real (p-value < 0.05)

What You Must Remember

1. **Decision trees** capture non-linear patterns 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. **Use SHAP** for explainability – don't treat trees as black boxes
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

Comparison Summary

Property	**Random Forest**	**Gradient Boosting**	**XGBoost**
Training	Parallel	Sequential	Sequential
Main Goal	$\downarrow$ Variance	$\downarrow$ Bias & Variance	Both + Speed
Tree Depth	Deep	Shallow (3–7)	Flexible
Learning Rate	N/A	Important	Important
Regularization	No	Limited	Yes (L1 + L2)
Speed	Fast	Moderate	Very Fast
Interpretability	Good	Good	Good
Competition Performance	Good	Better	**Best**

Some Tips

Start Simple, Then Improve

1. Train single decision tree – understand behavior
2. Add Random Forest – should be much better!
3. Try Gradient Boosting and XGBoost – find best model
4. Analyze with SHAP – understand what model learned

Warning Signs to Watch For

- Perfect training score (100%) $\rightarrow$ Overfitting!
- Big train/test gap (95% vs 70%) $\rightarrow$ Reduce complexity
- Weird feature importance $\rightarrow$ Check with SHAP
- Unstable predictions $\rightarrow$ Use more trees, cross-validation

Quick Debugging Checklist

- ☐ Used cross-validation (not just train/test split)?
- ☐ Checked for overfitting (train vs test gap)?
- ☐ Tuned hyperparameters (tried different settings)?
- ☐ Analyzed feature importance (makes sense)?
- ☐ Created SHAP plots (understand decisions)?
- ☐ Compared multiple models (which is best)?
- ☐ Tested on completely unseen data (real performance)?

If you can check all these boxes, you're doing great!

Example 1: Single Tree vs Random Forest

```python
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score

# 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!)
```

Key Observation: Random Forest dramatically reduces variance while maintaining accuracy!

Example 2: XGBoost Regularization Impact

```
# 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
# Gap of 0.25 indicates overfitting!

# With regularization: Generalizes better
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
# Gap of only 0.03 - much better generalization!
```

Lesson: Always use regularization with XGBoost!

Integration with Previous Weeks

Week	Topic	Connection to Week 6
1	Data Collection	Raw price data $\longrightarrow$ Features for trees
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 $\longrightarrow$ Trees (next week!)

The Big Picture

1. Collect & clean data (Weeks 1-2)
2. Create features from time series (Weeks 3-4)
3. Try linear models first (Week 5)
4. Capture non-linearities with trees (Week 6)
5. Reduce dimensions if needed (Week 7)
6. Build optimal portfolio!

Recommended Books

1. **Hastie, T., Tibshirani, R., & Friedman, J. (2009).**
   The Elements of Statistical Learning (2nd ed.). Springer.
   Chapters 10, 15, 16.
   The mathematical foundation – more advanced

2. **James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023).**
   An Introduction to Statistical Learning with Applications in Python (2nd ed.). Springer.
   Chapter 8.
   More accessible with Python examples

Free Resources

Both books are freely available online from the authors' websites!

Key Research Papers

1. **Breiman, L. (2001).** Random forests. *Machine Learning*, 45(1), 5–32.
   Original Random Forest paper
2. **Friedman, J. H. (2001).** Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5), 1189–1232.
   Original Gradient Boosting paper
3. **Chen, T., & Guestrin, C. (2016).** XGBoost: A scalable tree boosting system. *KDD 2016*, 785–794.
   XGBoost paper – most cited!
4. **Lundberg, S. M., & Lee, S.-I. (2017).** A unified approach to interpreting model predictions. *NeurIPS 2017*, 4765–4774.
   SHAP values paper

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.

   Comprehensive comparison of ML methods for asset pricing, including tree ensembles. Shows XGBoost performs best for predicting stock returns.

2. **Rasekhschaffe, K. C., & Jones, R. C. (2019).**
   Machine learning for stock selection.
   Financial Analysts Journal, 75(3), 70–88.

   Practical guide to using tree ensembles for portfolio construction and stock selection strategies.

Statistical Testing for Model Comparison

The Question

Is XGBoost **really** better than Random Forest, or did we just get lucky?

Hypothesis Test

- H_0: No difference between models
- H_1: XGBoost performs better

Use **paired t-test** on cross-validation scores, if p-value < 0.05, the difference is statistically significant!

$$t = \frac{\bar{d}}{s_d/\sqrt{k}}$$

- $\bar{d}$ = mean difference in CV scores
- s_d = standard deviation of differences
- k = number of CV folds

Statistical Testing in Python

```python
from scipy import stats
from sklearn.model_selection import cross_val_score

# Get CV scores for both models
rf_scores = cross_val_score(rf_model, X, y, cv=5)
xgb_scores = cross_val_score(xgb_model, X, y, cv=5)

# Paired t-test
t_stat, p_value = stats.ttest_rel(xgb_scores, rf_scores)

print(f"RF scores: {rf_scores.mean():.3f} +/- {rf_scores.std():.3f}")
print(f"XGB scores: {xgb_scores.mean():.3f} +/- {xgb_scores.std():.3f}")
print(f"t-statistic: {t_stat:.3f}")
print(f"p-value: {p_value:.4f}")

if p_value < 0.05:
    print("Difference is statistically significant!")
else:
    print("No significant difference detected.")
```

Next Week: Dimensionality Reduction

Coming Next Week

The Problem: What if you have 100+ features? Trees can get confused!

The Solution:

- **PCA**: Compress 100 features into 10 “super-features”
- **Clustering**: Group similar assets together
- **Factor Models**: Find hidden patterns in markets

Thank you