

Decision Trees and Random Forests

Prepared by Dr. Sumaya Alkubaisi

FNT633
Machine Learning in Finance

Lecture 6



Learning Goals

By the end of this lecture, you will be able to:

- Understand how Decision Trees work and how they split data using decision rules.
- Identify the effect of tree depth on model performance (underfitting vs. overfitting).
- Apply cross-validation to select the best max_depth for a Decision Tree.
- Explain why Random Forests are introduced to overcome Decision Tree limitations.
- Build, train, and evaluate Random Forest models using real-world financial data.
- Interpret feature importance and model results in a finance context.

Decision Trees – Overview

- A Decision Tree splits data into branches based on feature values until reaching a decision (leaf node).
- Each internal node: a decision rule on one feature.
- Each leaf node: prediction (mean for regression, class for classification).
- Highly interpretable and visual, but prone to overfitting.

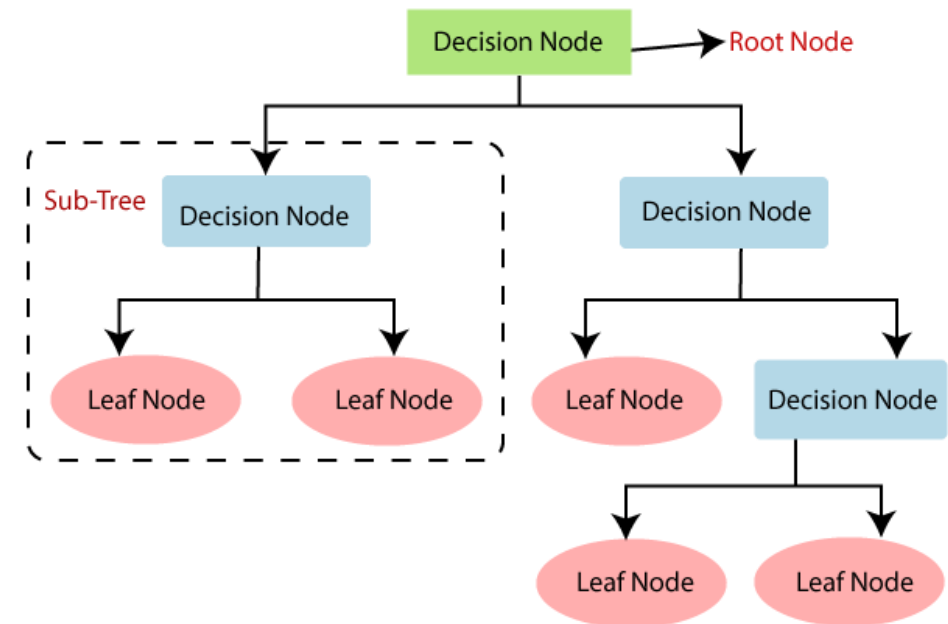
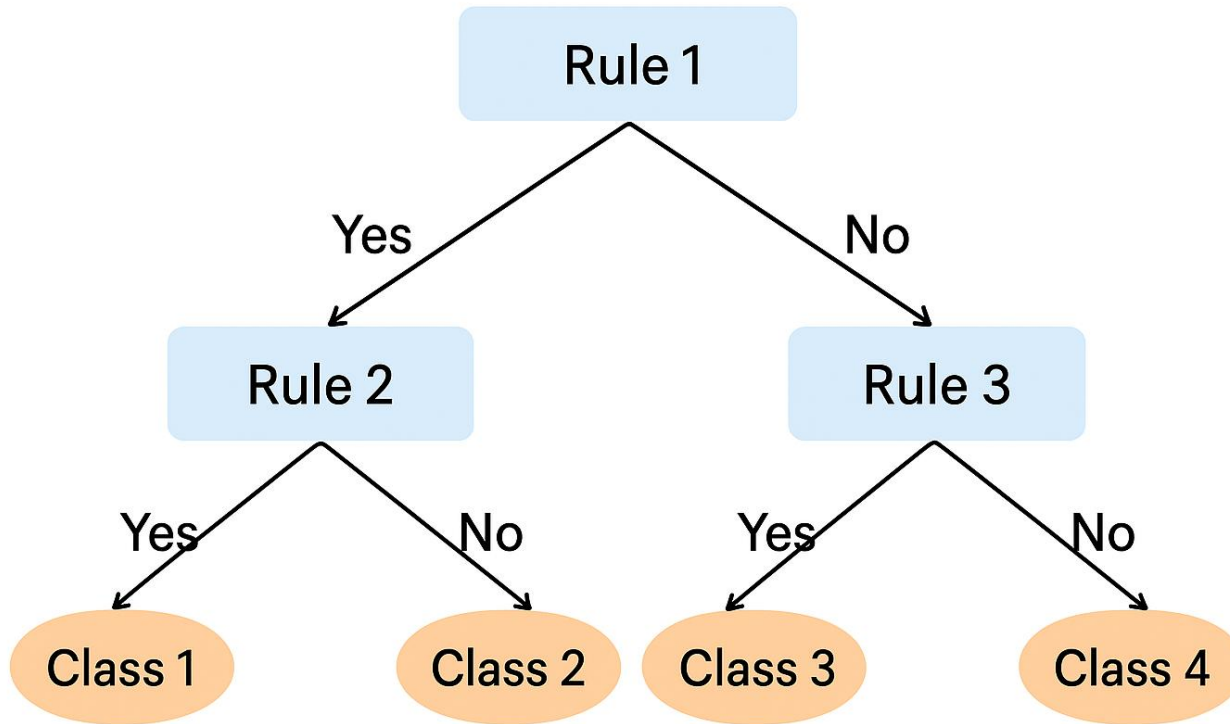


Image source: medium.com

How Decision Trees Work

- A decision tree works like a flowchart that makes decisions step by step. It looks at all the features (columns) in the data and finds the one that best separates the samples into groups (like “yes” vs “no”).
- It measures how pure each split is — using formulas like Gini or Entropy for classification, or MSE (Mean Squared Error) for regression.
- It keeps splitting the data into smaller and smaller groups until a rule (stopping criterion) says to stop — for example, when the tree reaches its maximum depth or the groups are already pure.

DECISION TREE ALGORITHM



Rules:

IF Credit_Score < 600 THEN Risk = High
 ELSE IF Income > 50000 THEN Risk = Low
 ELSE Risk = Medium

Decision Tree Example – Customer Churn Prediction

Goal: Classify customers as Low, Medium, or High Churn Risk

Root Rule: $\text{active_months} \leq 10.5$

Entropy: 1.585 $\rightarrow$ mixed classes ([2,2,2])

If True: $\rightarrow$ High Churn Risk

(Inactive ≤ 10.5 months, entropy = 0.0 $\rightarrow$ pure node)

If False: $\rightarrow$ Next rule (support_tickets ≤ 3)

Second Rule:

True (≤ 3 tickets): $\rightarrow$ Low Churn Risk

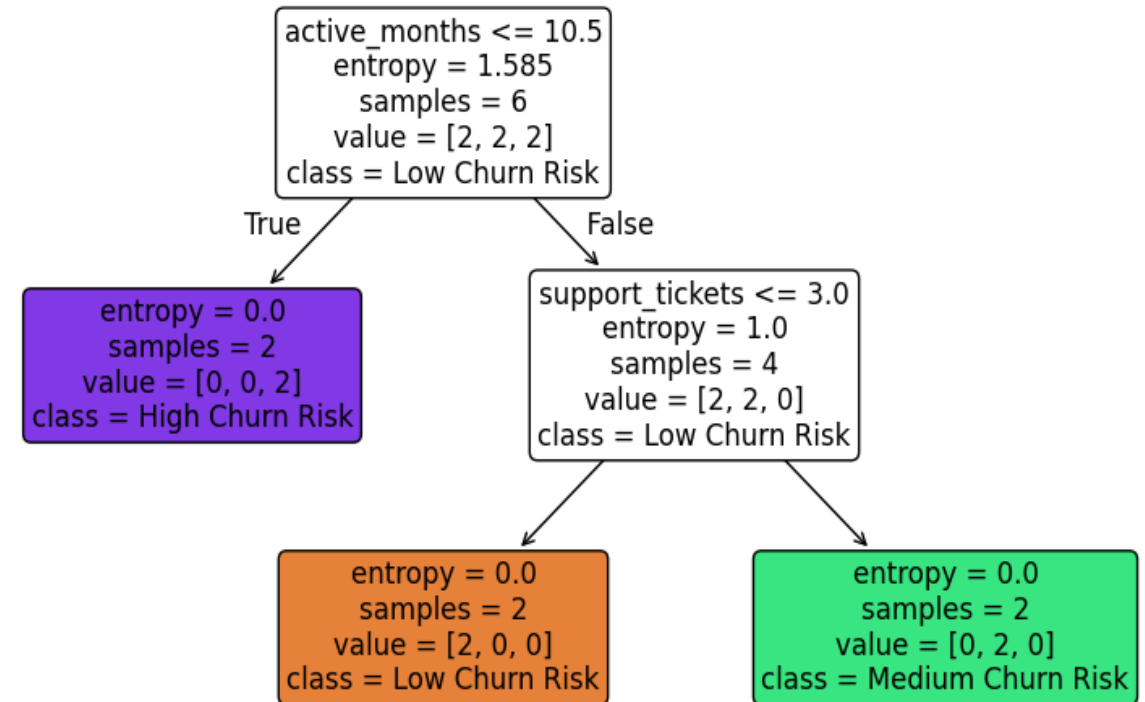
(Satisfied, entropy = 0.0 $\rightarrow$ pure node)

False (> 3 tickets): $\rightarrow$ Medium Churn Risk

(Frequent support, entropy = 0.0 $\rightarrow$ pure node)

Summary:

- Root split reduced entropy from 1.585 $\rightarrow$ 0.0 at all leaves.
- The tree perfectly separates customers into 3 pure risk groups.



Problem: Trees Growing Too Deep

- Without limits, trees can continue splitting until each leaf has one sample → overfitting (the tree memorizes the training data instead of learning general patterns).
- An overfitted tree performs very well on training data but poorly on unseen (test) data because it can't generalize.
- This reduces generalization (how well a model performs on new, unseen data) accuracy on new data.

Problem: Trees Growing Too Deep

Solution:

- To avoid overfitting, we can:
 - Limit tree depth (set a maximum number of levels).
 - Prune the tree — remove branches that add little value to accuracy.
- These methods keep the tree simpler and improve performance on new data.

Controlling Depth with `max_depth` in Python

- `max_depth` limits how deep the tree can grow.
- We can test multiple depth values and find the one that gives the best cross-validation performance.
- `DecisionTreeRegressor()`/ `DecisionTreeClassifier`: builds trees with controlled depth, using the parameter `max_depth`.
- The best `max_depth` minimizes overfitting and underfitting.
- `cross_val_score()` evaluates model performance across folds.

Python Demo: Decision Trees with Different Depths

#Step 1: Build and Visualize a Simple Decision Tree

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
```

1. Create tiny dataset manually

```
data = {
    'Avg_Return': [0.09, 0.05, 0.12, 0.07, 0.10, 0.04, 0.11, 0.06, 0.08, 0.15, 0.02, 0.13, 0.09, 0.05, 0.07],
    'Volatility': [0.10, 0.20, 0.13, 0.08, 0.15, 0.18, 0.11, 0.22, 0.14, 0.17, 0.25, 0.12, 0.09, 0.19, 0.16],
    'Investment_Amount': [20000, 50000, 40000, 60000, 25000, 70000, 30000, 80000, 45000, 90000, 10000,
35000, 55000, 65000, 15000],
    'Sharpe_Ratio': [0.9, 0.25, 0.92, 0.875, 0.67, 0.22, 1.0, 0.27, 0.57, 0.88, 0.08, 1.08, 1.0, 0.26, 0.44],
    'Credit_Score': [700, 620, 680, 750, 710, 600, 730, 590, 680, 720, 550, 690, 710, 640, 700],
    'Age': [30, 45, 40, 28, 35, 50, 33, 60, 38, 42, 25, 36, 39, 48, 31],
    'Experience_Years': [5, 20, 12, 3, 7, 25, 10, 30, 8, 15, 1, 14, 6, 22, 4] }
```

Python Demo: Decision Trees with Different Depths

#Step 1: Build and Visualize a Simple Decision Tree

```
# Risk level based on volatility and return
risk_level = []
for v, r in zip(data['Volatility'], data['Avg_Return']):
    if v < 0.12 and r > 0.07:
        risk_level.append('Low')
    elif 0.12 <= v < 0.18:
        risk_level.append('Medium')
    else:
        risk_level.append('High')

portfolio_df = pd.DataFrame(data)
portfolio_df['Risk_Level'] = risk_level
portfolio_df['Risk_Level_Encoded'] = portfolio_df['Risk_Level'].map({'Low':0, 'Medium':1, 'High':2})
```

Python Demo: Decision Trees with Different Depths

#Step 1: Build and Visualize a Simple Decision Tree

2. Scale features

```
scaler = MinMaxScaler()  
features = ['Avg_Return', 'Volatility', 'Investment_Amount', 'Sharpe_Ratio',  
'Credit_Score', 'Age', 'Experience_Years']  
X_scaled = scaler.fit_transform(portfolio_df[features])  
X = pd.DataFrame(X_scaled, columns=features)  
y = portfolio_df['Risk_Level_Encoded']
```

Python Demo: Decision Trees with Different Depths

#Step 1: Build and Visualize a Simple Decision Tree

3. Train/Test split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

4. Train Decision Tree with fixed depth = 3

```
tree_clf = DecisionTreeClassifier(max_depth=3, random_state=42)
```

```
tree_clf.fit(X_train, y_train)
```

5. Predictions & evaluation

```
y_pred = tree_clf.predict(X_test)
```

```
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

Python Demo: Decision Trees with Different Depths

#Step 1: Build and Visualize a Simple Decision Tree

6. Visualize the tree

```
plt.figure(figsize=(12,6))
```

```
plot_tree(tree_clf, feature_names=X.columns, class_names=['Low','Medium','High'],  
filled=True)
```

```
plt.show()
```

Python Demo: Output

#Step 1: Build and Visualize a Simple Decision Tree

Overall, the model achieved 80% accuracy, but it struggles a bit with Medium-risk portfolios due to the small dataset. The confusion matrix shows where misclassifications happened, mainly between Low and Medium risk.”

```
[[1 0 0] → 1 Low-risk correct, 0 misclassified  
[1 1 0] → 1 Medium misclassified as Low, 1 Medium correct  
[0 0 2]] → 2 High-risk correct, 0 misclassified
```

Classification Report:				
	precision	recall	f1-score	support
0	0.50	1.00	0.67	1
1	1.00	0.50	0.67	2
2	1.00	1.00	1.00	2
accuracy			0.80	5
macro avg	0.83	0.83	0.78	5
weighted avg	0.90	0.80	0.80	5

Python Demo: Output – Investment Risk Decision Tree

#Step 1: Build and Visualize a Simple Decision Tree

Goal: Classify investments as Low, Medium, or High Risk
using: Average Return (Avg_Return), Volatility, and Age.

Root Rule: Avg_Return ≤ 0.423

Samples: 10 $\rightarrow$ Distribution [2,4,4] (Low, Med, High)

Gini = 0.64 $\rightarrow$ Highly mixed, so a split is needed.

Left Branch ($\leq 42.3\%$) – High Risk Path

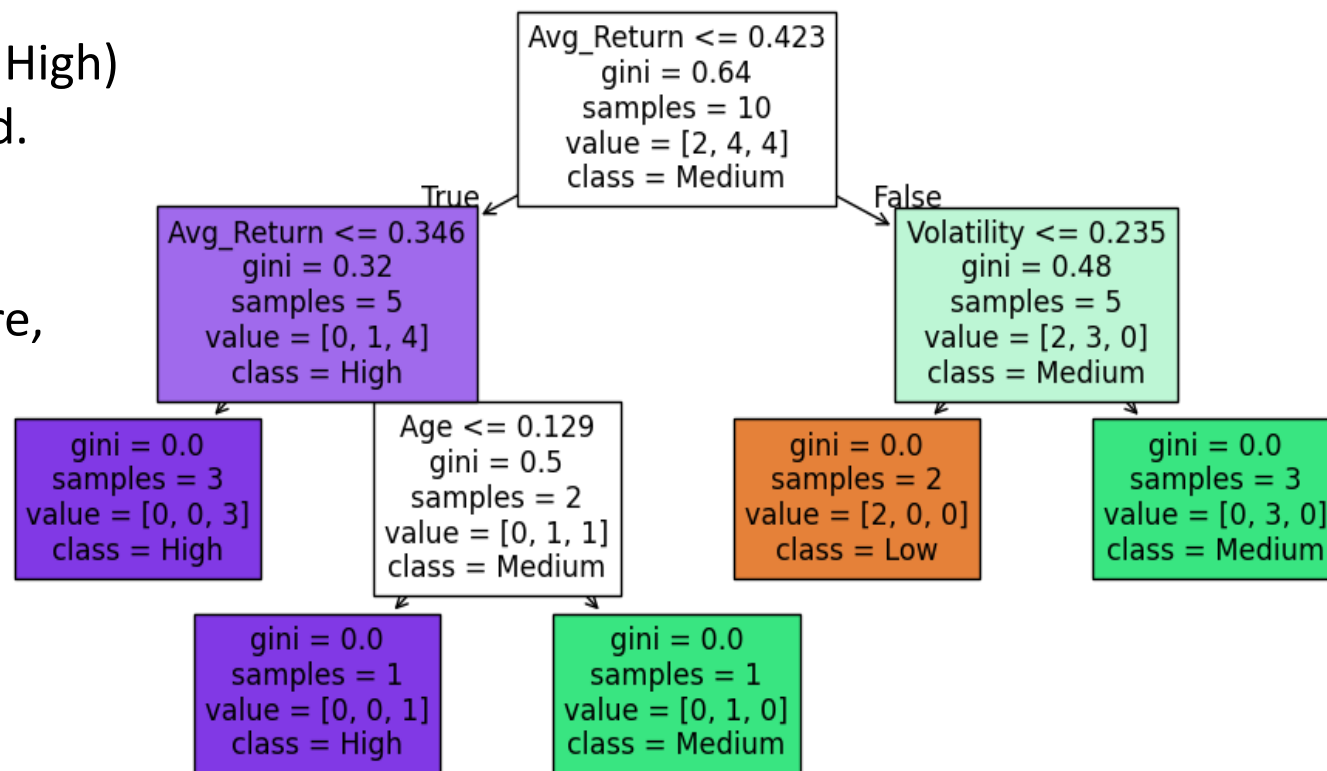
Gini $\downarrow$ to 0.32 $\rightarrow$ 5 samples [0,1,4]

- **If** Avg_Return ≤ 0.346 : $\rightarrow$ High Risk (pure, Gini = 0.0)
- **Else** check Age ≤ 0.129 :
 - **True** $\rightarrow$ High Risk (young investment, pure)
 - **False** $\rightarrow$ Medium Risk (older, pure)

Right Branch ($> 42.3\%$) – Volatility Path

Gini = 0.48 $\rightarrow$ 5 samples [2,3,0]

- **If** Volatility > 0.235 : $\rightarrow$ Low Risk (pure, Gini = 0.0)
- **If** Volatility ≤ 0.235 : $\rightarrow$ Medium Risk (pure, Gini = 0.0)



Python Demo: Decision Trees with Different Depths

Step 2: Use Cross-Validation to Find the Best Depth

```
from sklearn.model_selection import cross_val_score  
import numpy as np
```

```
# Test different max_depth values from 1 to 10  
depth_range = range(1, 11)  
cv_scores = []
```

```
for depth in depth_range:  
    tree = DecisionTreeClassifier(max_depth=depth, random_state=42)  
    scores = cross_val_score(tree, X, y, cv=5, scoring='accuracy') # 5-fold CV  
    cv_scores.append(scores.mean())
```

Python Demo: Decision Trees with Different Depths

Step 2: Use Cross-Validation to Find the Best Depth

```
# Find the depth with the highest CV accuracy
best_depth = depth_range[np.argmax(cv_scores)]
print(f"Best max_depth based on cross-validation: {best_depth}")

# Plot CV accuracy vs depth
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))
plt.plot(depth_range, cv_scores, marker='o')
plt.xlabel('max_depth')
plt.ylabel('CV Accuracy')
plt.title('Cross-Validation Accuracy vs Decision Tree Depth')
plt.grid(True)
plt.show()
```

Python Demo: Output

Step 2: Use Cross-Validation to Find the Best Depth

Goal: Examine how tree depth affects model accuracy.

The best Depth:

$\text{max_depth} = 2 \rightarrow \text{CV Accuracy} \approx 0.60$ (60%)

Best trade-off between bias and variance — captures key patterns efficiently.

Underfitting:

$\text{Depth} = 1 \rightarrow \text{Accuracy} \approx 0.53$

Tree too simple $\rightarrow$ misses important relationships.

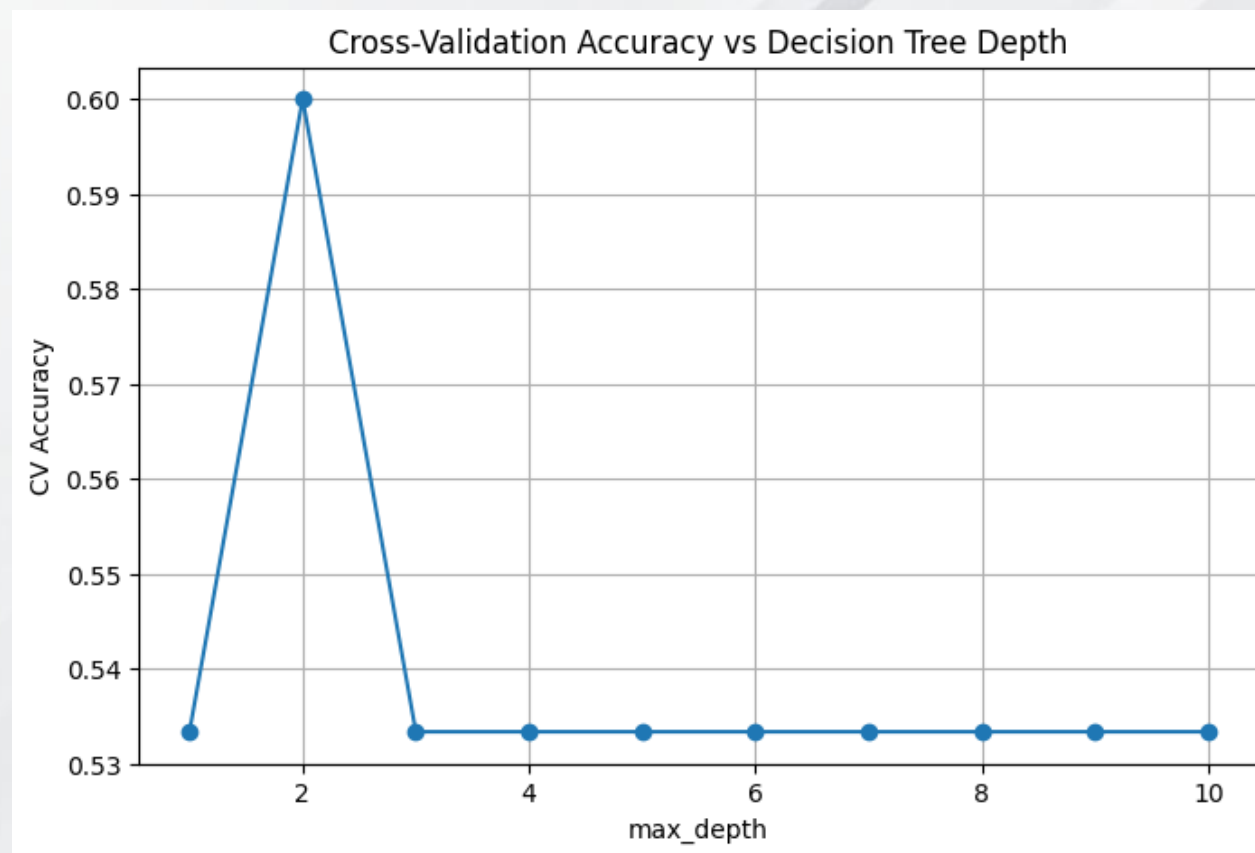
Plateau / Overfitting:

$\text{Depth} \geq 3 \rightarrow \text{Accuracy drops to} \approx 0.53$ and stays flat

The model plateaus after $\text{depth} = 2$, meaning deeper trees add no new value and may overfit the training data.

Conclusion:

Performance peaks at $\text{depth} = 2$ and plateaus beyond it — this is the optimal depth for best generalization.



Python Demo: Decision Trees with Different Depths

Step 3: Retrain the Tree Using the Best Depth and Compare

```
# Retrain the Decision Tree with the best depth
tree_best = DecisionTreeClassifier(max_depth=best_depth, random_state=42)
tree_best.fit(X_train, y_train)

# Predict and evaluate
y_pred_best = tree_best.predict(X_test)
print("Classification Report (Best Depth):\n", classification_report(y_test, y_pred_best))
print("Confusion Matrix (Best Depth):\n", confusion_matrix(y_test, y_pred_best))

# Visualize the optimized tree
plt.figure(figsize=(12,6))
plot_tree(tree_best, feature_names=X.columns, class_names=['Low','Medium','High'],
filled=True)
plt.show()
```

Python Demo: Output

Step 3: Retrain the Tree Using the Best Depth and Compare

- Here, both trees perform about the same, for this tiny dataset, the best depth of 2 did not improve performance compared to the fixed depth tree.
- Decision trees are sensitive to small data; small changes in depth won't change predictions much.
- Cross-validation may not make a difference with such limited data.

```
[[1 0 0]] → 1 Low-risk correct, 0 misclassified  
[[1 1 0]] → 1 Medium-risk misclassified as Low, 1 Medium correct  
[[0 0 2]] → 2 High-risk correct, 0 misclassified
```

Classification Report (Best Depth):				
	precision	recall	f1-score	support
0	0.50	1.00	0.67	1
1	1.00	0.50	0.67	2
2	1.00	1.00	1.00	2
accuracy			0.80	5
macro avg	0.83	0.83	0.78	5
weighted avg	0.90	0.80	0.80	5

Transition to Random Forests

- Decision Trees are simple and interpretable but unstable: small data changes → large model changes — even small data changes can drastically alter the tree's structure and predictions.
- Prone to overfitting, especially with deeper trees or small datasets.
- To reduce variance and improve robustness, we use Random Forests: an ensemble of many trees trained on random subsets.

Random Forest – Concept

- A Random Forest is an ensemble of many Decision Trees (e.g., 100).
- Each tree is trained on a random subset of the data and features (technique known as bagging).

Prediction:

- Classification: Majority vote across trees.
- Regression: Average of all tree outputs.

This randomness ensures that trees are diverse, making the overall model more stable and accurate.

Key parameters:

n_estimators → Number of trees.

max_depth → Tree depth limit.

max_features → Number of features considered per split.

Random Forest – Concept

Limiting Number of Trees ($n_{\text{estimators}}$)

- Too few trees → Unstable and less accurate (not enough averaging).
- Too many trees → minimal gain, higher training/memory cost.
- Accuracy plateaus after a certain number.
- Increases training time, memory use, and prediction time unnecessarily.

Goal: Find the smallest number of trees that gives stable accuracy.

Random Forest – Concept

Limiting Features per Split (`max_features`)

- At each split, Random Forests randomly select a subset of features.
- Why: Ensures trees are less correlated (each tree sees different data parts).
- Reduces overfitting, improving generalization.
- Too many features: Trees become similar → less ensemble benefit.
- Too few features: Weak splits → underfitting possible.

Goal: Balance diversity and predictive strength.

Random Forest Simplified

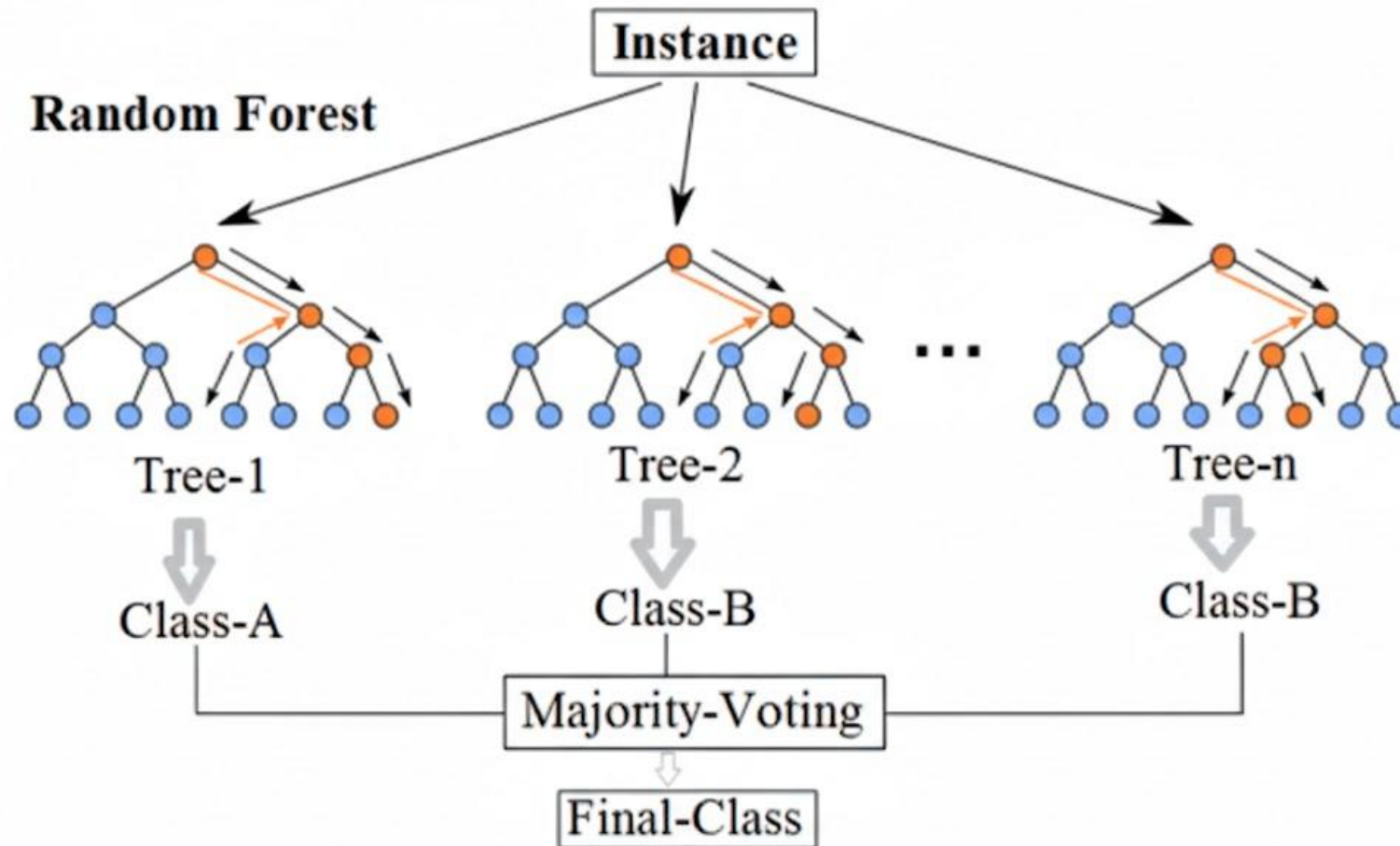


Image source: medium.com

Dealing with Overfitting and Pruning (model tuning)

Random Forests naturally reduce overfitting by averaging results from multiple diverse trees.

Further control can be achieved by:

- Limiting tree depth to avoid excessive complexity.
- Reducing features per split (`max_features`) for more diverse trees.
- Increasing training data size to provide more balanced learning.
- Together, these steps create a model that is both robust and generalizes well to unseen data.

Purpose:

- To estimate how well the Random Forest generalizes to unseen data and to find the best combination of hyperparameters (like `n_estimators`, `max_depth`, and `max_features`).
- Prevents overfitting by testing model stability across data splits.
- Helps choose the optimal number of trees — more trees increase accuracy up to a point where the CV accuracy plateaus.
- Helps determine how many features to consider per split — ensuring the model stays both accurate and diverse.

Python Demo: Random Forest

Step 1: Train Random Forest with max_depth=3

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# we will use the same data set created for Decision Trees in this Lecture6
# to train the Random Forest with max_depth=3

rf_model_fixed = RandomForestClassifier(n_estimators=100, max_depth=3, random_state=42)
rf_model_fixed.fit(X_train, y_train)

# Predictions
rf_pred_fixed = rf_model_fixed.predict(X_test)
```

Python Demo: Random Forest

Step 1: Train Random Forest with max_depth=3

Evaluation

```
print("Random Forest (max_depth=3) Accuracy:", accuracy_score(y_test, rf_pred_fixed))  
print("\nClassification Report:\n", classification_report(y_test, rf_pred_fixed))  
print("Confusion Matrix:\n", confusion_matrix(y_test, rf_pred_fixed))
```

Feature importance

```
feature_importances = rf_model_fixed.feature_importances_  
features = X.columns  
importance_df = pd.DataFrame({'Feature': features, 'Importance':  
feature_importances}).sort_values(by='Importance', ascending=False)  
  
plt.figure(figsize=(10,6))  
sns.barplot(x='Importance', y='Feature', data=importance_df, palette='viridis')  
plt.title('Random Forest Feature Importance (max_depth=3)')  
plt.xlabel('Importance')  
plt.ylabel('Feature')  
plt.show()
```

Python Demo: Random Forest (Output)

Step 1: Train Random Forest with max_depth=3

“Our Random Forest with max depth 3 achieved 60% accuracy on the test set.

Looking at the confusion matrix:

- Low-risk (0): misclassified as Medium-risk.
- Medium-risk (1): one correctly classified, one misclassified as Low-risk.
- High-risk (2): both correctly classified.

This shows the model predicts High-risk well, struggles with Low and Medium-risk portfolios, likely due to the small dataset.”

- Random Forests usually outperform single trees on large datasets.
- On tiny datasets, a single tree can actually look better.

```
Random Forest (max_depth=3) Accuracy: 0.6

Classification Report:
              precision    recall  f1-score   support

      0               0.00      0.00      0.00         1
      1               0.50      0.50      0.50         2
      2               1.00      1.00      1.00         2

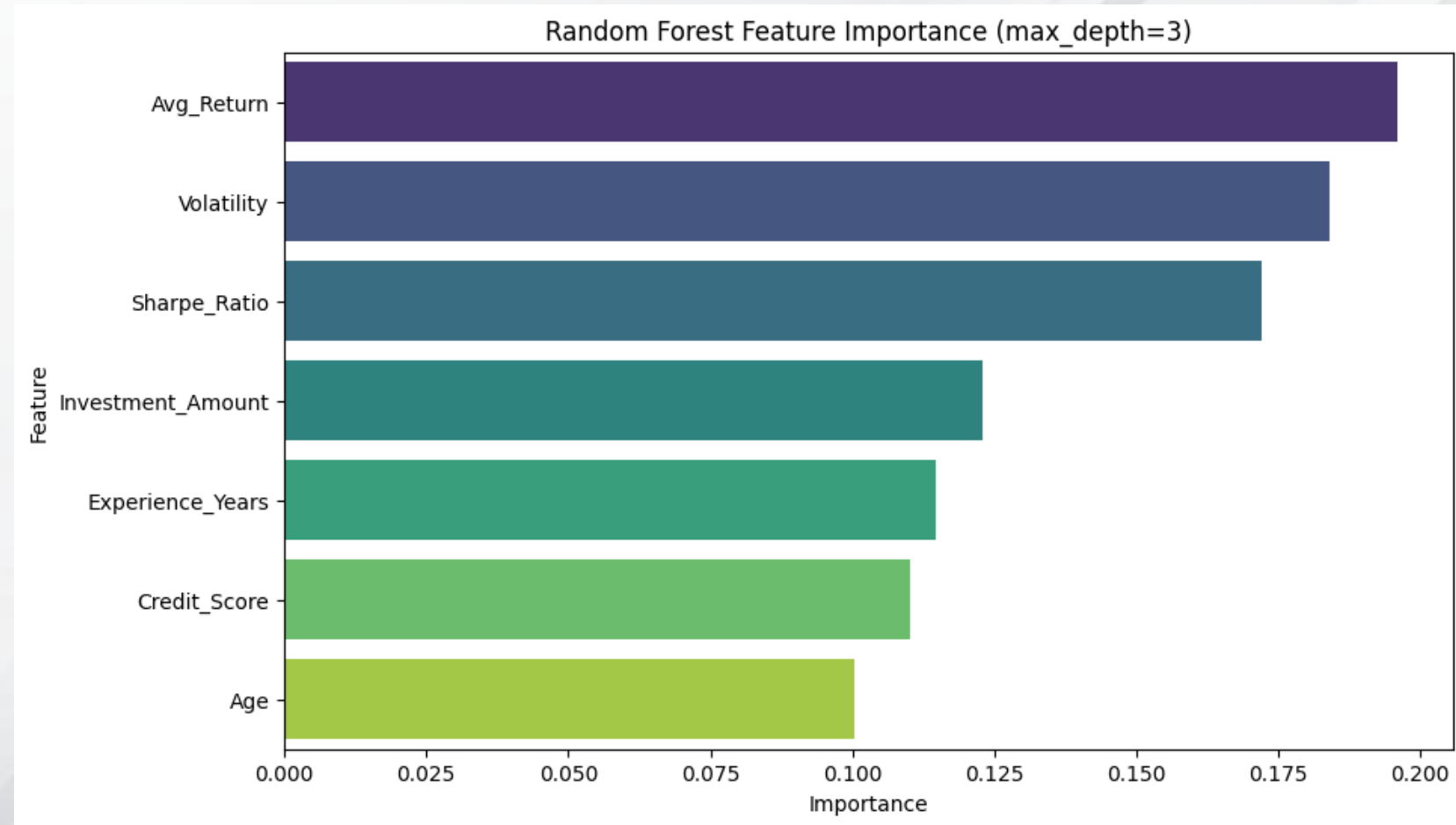
   accuracy               0.60         5
  macro avg               0.50      0.50      0.50         5
 weighted avg               0.60      0.60      0.60         5

Confusion Matrix:
[[0 1 0]
 [1 1 0]
 [0 0 2]]
```

Python Demo: Random Forest (Output)

Step 1: Train Random Forest with max_depth=3

- The Random Forest (max_depth=3) relies most on Avg_Return, followed by Volatility and Sharpe_Ratio, showing financial metrics drive predictions.
- Investment_Amount, Experience_Years, and Credit_Score have moderate influence, while Age contributes least.
- The feature importance chart helps interpret which factors the model considers most relevant for classifying portfolio risk.



Python Demo: Random Forest

Step 2: Cross-Validation to Find Best max_depth

```
depth_range = range(1, 11) # testing depths from 1 to 10  
cv_scores = []
```

```
for depth in depth_range:  
    rf = RandomForestClassifier(n_estimators=100, max_depth=depth, random_state=42)  
    scores = cross_val_score(rf, X, y, cv=5, scoring='accuracy')  
    cv_scores.append(scores.mean())
```

```
# Find best depth  
best_depth = depth_range[cv_scores.index(max(cv_scores))]  
print(f"Best max_depth from cross-validation: {best_depth}")
```

Python Demo: Random Forest

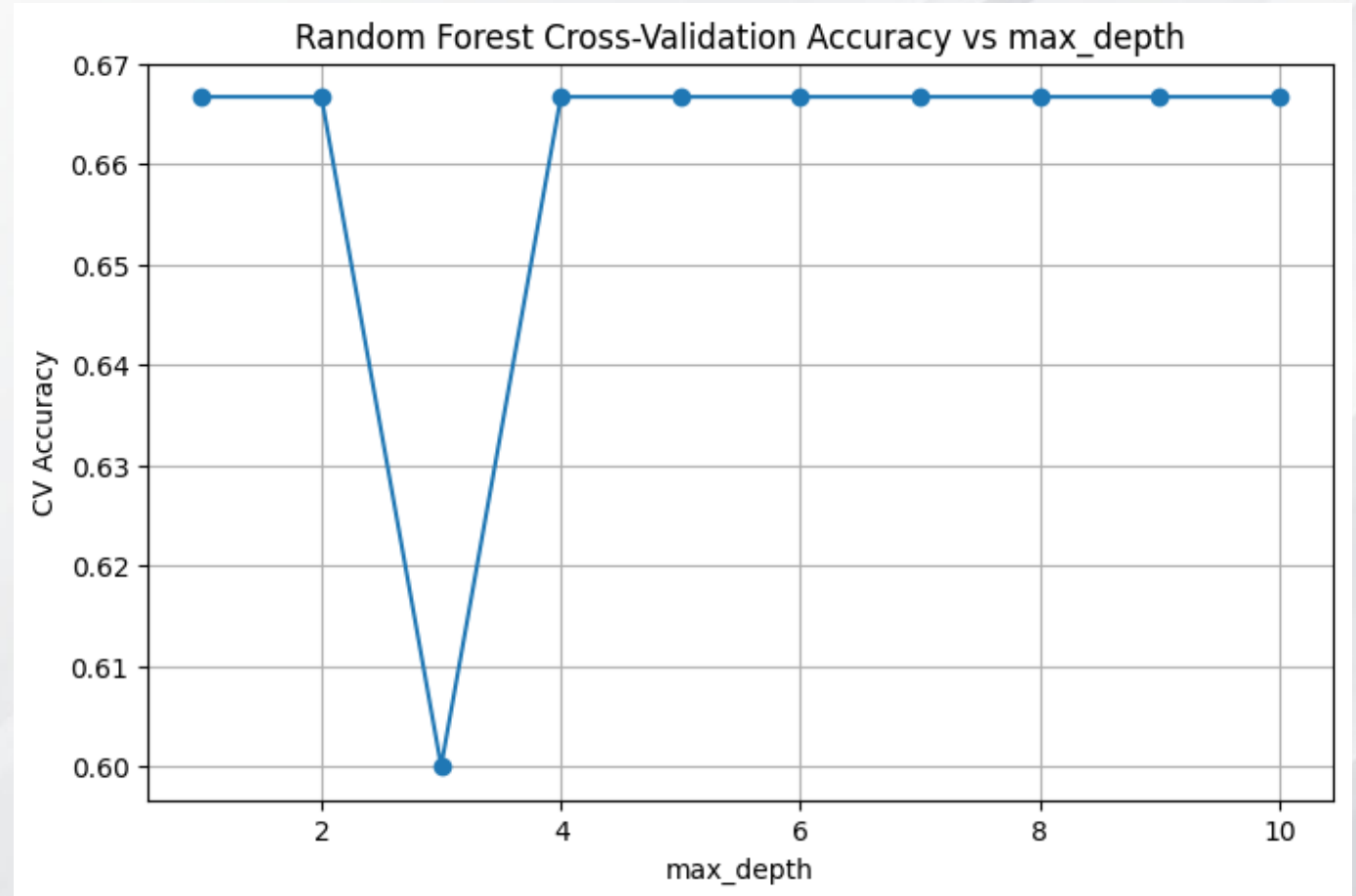
Step 2: Cross-Validation to Find Best max_depth

```
# Plot CV accuracy vs depth
plt.figure(figsize=(8,5))
plt.plot(depth_range, cv_scores, marker='o')
plt.xlabel('max_depth')
plt.ylabel('CV Accuracy')
plt.title('Random Forest Cross-Validation Accuracy vs max_depth')
plt.grid(True)
plt.show()
```

Python Demo: Random Forest (Output)

Step 2: Cross-Validation to Find Best max_depth

- The best max_depth for this Random Forest is 1, 2 or 4 and above, as these give the highest and stable cross-validation accuracy of about 66.7%.
- A depth of 3 performs noticeably worse.
- This shows that some tree depths work better than others for this dataset.



Python Demo: Random Forest

Step 3: Retrain Random Forest with Best Depth

```
rf_model_best = RandomForestClassifier(n_estimators=100, max_depth=best_depth, random_state=42)
rf_model_best.fit(X_train, y_train)
rf_pred_best = rf_model_best.predict(X_test)

print("Random Forest (Best Depth) Accuracy:", accuracy_score(y_test, rf_pred_best))
print("\nClassification Report:\n", classification_report(y_test, rf_pred_best))
print("Confusion Matrix:\n", confusion_matrix(y_test, rf_pred_best))
```

Python Demo: Random Forest (Output)

Step 3: Retrain Random Forest with Best Depth

- The Random Forest with the best depth achieved 80% accuracy, correctly classifying all Medium and High-risk samples, but still missed the single Low-risk sample.
- Compared to the Random Forest with max_depth=3 (60% accuracy), the best-depth model performs noticeably better, especially for Medium-risk portfolios.

```
Random Forest (Best Depth) Accuracy: 0.8

Classification Report:
              precision    recall  f1-score   support

     0       0.00         0.00         0.00         1
     1       0.67         1.00         0.80         2
     2       1.00         1.00         1.00         2

 accuracy          0.80         0.80         0.80         5
 macro avg         0.56         0.67         0.60         5
 weighted avg      0.67         0.80         0.72         5

Confusion Matrix:
[[0 1 0]
 [0 2 0]
 [0 0 2]]
```

Recap

- Decision Trees make predictions by splitting data into smaller, more uniform groups using decision rules.
- A tree that is too deep can overfit the data, while a shallow tree may underfit it.
- Cross-validation helps choose the optimal max_depth that balances bias and variance.
- Random Forests combine many Decision Trees to improve accuracy and reduce overfitting through bagging (training on random subsets of data and features).
- Both models are powerful tools for financial prediction tasks such as portfolio risk, credit scoring, and loan default analysis.

References

- Klaas, J. (2021). Machine Learning for Finance: Principles and Practice for Financial Insiders. Wiley.
- López de Prado, M. (2018). Advances in Financial Machine Learning. Wiley.

