■ **Course:** CS109A: Introduction to Data Science

■ **Lecture:** Lecture 22: Random Forests

■ **Instructors:** Pavlos Protopapas, Kevin Rader, Chris Gumb

■ **Objective:** Understand how Random Forests decorrelate trees, learn variable importance methods, and handle class imbalance and missing data

Contents

1 The Problem with Bagging: Correlated Trees

Key Summary

Random Forest is an enhancement of Bagging that adds **feature randomization** at each split to reduce correlation between trees. This lecture covers:

1. Why bagged trees are often correlated and why that's a problem
2. How Random Forests decorrelate trees
3. Variable importance methods (MDI and Permutation)
4. Handling class imbalance
5. Missing data with surrogate splits

1.1 Quick Review: Bagging

Recall from last lecture:

- **Bagging** = Bootstrap + Aggregating
- Create B bootstrap samples from your training data
- Train a decision tree on each sample
- Aggregate predictions (majority vote or average)
- Use **OOB error** for validation

The key insight was that averaging multiple predictions reduces variance by a factor related to $1/\sqrt{n}$.

1.2 The Correlation Problem

Here's the issue we discovered: **Bagged trees often look very similar**.

Example: Why Trees Are Correlated

Imagine you have a dataset with 10 features, and one of them—`Glucose`—is an extremely strong predictor of diabetes.

When you build Tree 1 on Bootstrap Sample 1:

- You check all 10 features for the best split
- `Glucose` wins (it always does—it's the strongest!)
- Tree 1's root node splits on `Glucose`

When you build Tree 2 on Bootstrap Sample 2:

- You check all 10 features for the best split
- `Glucose` wins again!
- Tree 2's root node also splits on `Glucose`

This pattern continues for all B trees. They all start the same way!

Important: Why Correlation Matters

The variance reduction formula assumes predictions are **independent**:

$$\text{Var}(\bar{X}) = \frac{\sigma^2}{n} \quad \text{(if independent)}$$

But if predictions are **correlated**, the variance reduction is weaker:

$$\text{Var}(\bar{X}) = \frac{\sigma^2}{n} + \rho\sigma^2 \cdot \frac{n-1}{n} \approx \rho\sigma^2 \quad (\text{as } n \to \infty)$$

where ρ is the correlation. High correlation means less variance reduction—exactly what we don't want!

Example: The Doctor Analogy Revisited

Remember our analogy of getting multiple medical opinions? If all doctors were trained at the same hospital with the same textbooks, they'd give you the same advice: "Rest, drink water, come back in 3 days."

That's not very useful! You want doctors with **diverse training** and **different perspectives**. The same applies to trees—we want them to be diverse, not copies of each other.

2 The Random Forest Solution: Feature Randomization

Definition: Random Forest

A **Random Forest** is a modification of Bagging where, at **each split in each tree**, we only consider a **random subset of features** for splitting.
This forces trees to make different choices, reducing correlation.

2.1 How Random Forests Work

The algorithm is beautifully simple—just one change from bagging:

1. Create B bootstrap samples (same as bagging)
2. For each tree, at **each split**:
   (a) Randomly select m features from the total p features (where $m < p$)
   (b) Find the best split among **only these m features**
   (c) Make the split
3. Repeat until trees are fully grown (or reach stopping criteria)
4. Aggregate predictions (same as bagging)

Important: The Key Difference

Bagging: At each split, consider **all p features** → strong features always win
Random Forest: At each split, consider **only m features** → strong features sometimes excluded
This "controlled randomness" forces diversity!

Example: Random Forest in Action

Dataset: Heart disease prediction with 5 features:
Age, Sex, Max Heart Rate, Cholesterol, Chest Pain
Setting: $m = 3$ (consider 3 random features at each split)
Building Tree 1:

- *Root split*: Randomly select 3 features → {Age, Sex, Cholesterol}
- Best split among these 3: **Sex**
- *Next split (left child)*: Randomly select 3 features → {Age, Max HR, Chest Pain}
- Best split among these 3: **Max Heart Rate**
- Continue...

Building Tree 2:

- *Root split*: Randomly select 3 features → {Cholesterol, Chest Pain, Max HR}
- Best split among these 3: **Chest Pain**
- Different root! Trees will be less correlated.

Notice: Tree 1 and Tree 2 start with different splits because they considered different feature subsets!

2.2 Choosing m: The Number of Features to Consider

The parameter m (often called `max_features` in sklearn) controls how many features we randomly select at each split.

Definition: Rules of Thumb for m

For Classification:

$$m \approx \sqrt{p}$$

where p is the total number of features.

For Regression:

$$m \approx \frac{p}{3}$$

These are starting points—tune with cross-validation or OOB error!

Example: Choosing m in Practice

If you have 100 features:

- **Classification**: Start with $m = \sqrt{100} = 10$
- **Regression**: Start with $m = 100/3 \approx 33$

If you have 9 features (like the Diabetes dataset):

- **Classification**: Start with $m = \sqrt{9} = 3$
- **Regression**: Start with $m = 9/3 = 3$

2.3 Extreme Cases: What Happens with Different m?

Understanding the extremes helps build intuition:

- $m = p$ **(all features)**: This is just regular Bagging! Trees are highly correlated.
- $m = 1$ **(one feature)**: Completely random trees. Each split is determined by chance. Trees are uncorrelated but individually very weak.
- $m = \sqrt{p}$ **or** $m = p/3$: The sweet spot. Trees are diverse but still make reasonably good splits.

3 Random Forest Hyperparameters

Random Forests have more hyperparameters than single trees. Let's organize them:

3.1 Overview of Hyperparameters

Table 1: *Random Forest Hyperparameters*

Parameter	Controls	Notes
`n_estimators`	Number of trees (B)	More is better, never overfits
`max_features`	Features per split (m)	Key decorrelation parameter
`max_depth`	Tree depth	Controls tree complexity
`min_samples_split`	Min samples to split	Stopping condition
`min_samples_leaf`	Min samples per leaf	Stopping condition
`criterion`	Gini or Entropy	Usually doesn't matter much

3.2 Practical Advice for Tuning

Key Information

Expert Tips from the Instructors:

1. **Start with defaults**: sklearn's defaults are usually reasonable
2. **Focus on the important ones**:
   - `n_estimators`: Start with 100-500. More is always safe (no overfitting)
   - `max_features`: Start with $\sqrt{p}$ for classification, $p/3$ for regression
   - `max_depth`: Try None (fully grown) first, then limit if needed
3. **Don't overthink criterion**: Gini and Entropy give similar results. Just pick Gini.
4. **Use OOB instead of CV**: Random Forests give you OOB error for free—use it!

3.3 The Hyperparameter Explosion Problem

With multiple hyperparameters, the search space explodes:

- 5 choices for `max_features` $\times$
- 5 choices for `max_depth` $\times$
- 5 choices for `n_estimators` $\times$
- 2 choices for `criterion`
- $= 250$ combinations!

With 5-fold CV, that's 1,250 models to train. This is why:

1. We use rules of thumb to narrow the search
2. We use OOB error instead of CV (no additional training needed)
3. We use random search instead of grid search

4 Variable Importance

When we moved from single decision trees to ensembles, we lost **interpretability**. You can no longer say "if Glucose $>$ 120 and Age $>$ 50, then Diabetic" because you have hundreds of different trees!

Variable Importance is how we get some interpretability back.

4.1 Why Variable Importance Matters

Example: Motivating Example

You build a Random Forest to predict customer churn with 50 features. The model achieves 95% accuracy. Your boss asks:
"Great accuracy, but **why** are customers leaving? What should we focus on?"
Variable importance answers this: "The top 3 factors are: Contract Type, Monthly Charges, and Tenure. Customers with month-to-month contracts and high monthly charges who've been with us less than a year are most likely to leave."
Now you have actionable insights!

4.2 Method 1: Mean Decrease in Impurity (MDI)

Definition: Mean Decrease in Impurity (MDI)

MDI measures how much each feature contributes to reducing impurity (Gini or MSE) across all trees in the forest.
For each feature:

1. Find all nodes in all trees where that feature is used for splitting
2. Sum up the impurity decrease at each of those nodes
3. Average across all trees

Features that cause larger decreases in impurity are more important.

4.2.1 MDI Calculation Step by Step

1. **For each node n that uses feature j:**

$$\Delta I_n = \frac{n_{samples}}{N} \times \left[I_{before} - \sum_{c \in children} \frac{n_c}{n_{samples}} I_c \right]$$

   where I is impurity (Gini), $n_{samples}$ is samples at node, N is total samples

2. **Sum for feature j in tree t:**

$$F_j^{(t)} = \sum_{\text{nodes using } j} \Delta I_n$$

3. **Normalize within each tree:**

$$\hat{F}_j^{(t)} = \frac{F_j^{(t)}}{\sum_k F_k^{(t)}}$$

4. **Average across all trees:**

$$\text{MDI}_j = \frac{1}{B}\sum_{t=1}^{B} \hat{F}_j^{(t)}$$

4.2.2 MDI Pros and Cons

+ **Fast**: Computed during training, no extra computation
+ **Built into sklearn**: Just access `model.feature_importances_`
− **Biased toward high-cardinality features**: Features with many unique values (continuous or categorical with many categories) get artificially inflated importance
− **Computed on training data**: Can be misleading if model overfits

4.3 Method 2: Permutation Importance

Definition: Permutation Importance

Permutation importance measures how much model performance **decreases** when a feature's values are randomly shuffled.
If shuffling a feature destroys model accuracy, that feature was important!

4.3.1 Permutation Importance Algorithm

1. **Baseline**: Compute baseline OOB accuracy $s_{baseline}$
2. **For each feature** j:
   (a) Randomly shuffle (permute) the values of feature j
   (b) Compute OOB accuracy with shuffled feature: $s_j^{(k)}$
   (c) Repeat K times and average: $s_j = \frac{1}{K}\sum_k s_j^{(k)}$
3. **Calculate importance**:

$$\text{Importance}_j = s_{baseline} - s_j$$

Example: Permutation Importance Intuition

Dataset: Predicting fitness level from height, weight, and age
Baseline OOB accuracy: 88%
Shuffle height:
- Original: $[170, 165, 180, 175, 160]$
- Shuffled: $[175, 180, 165, 160, 170]$
- New accuracy: 87%
- Importance = 88% - 87% = 1% (height matters a little)

Shuffle weight:
- New accuracy: 72%
- Importance = 88% - 72% = 16% (weight matters A LOT!)

Conclusion: Weight is much more important than height for predicting fitness.

4.3.2 Permutation Importance Pros and Cons

+ **Not biased**: Works equally well for all feature types
+ **Intuitive**: Directly measures impact on model performance
+ **Uses validation data**: Reflects true generalization, not training fit
− **Slower**: Requires multiple predictions per feature
− **Correlated features**: Can underestimate importance when features are correlated

4.4 MDI vs. Permutation: Which to Use?

Important: Recommendation

Use Permutation Importance when possible.
MDI is convenient (it's built into sklearn and fast), but it's biased toward high-cardinality features. Permutation importance gives more reliable results.
In practice:

- Use MDI for quick exploration
- Use Permutation Importance for final analysis and reporting

4.5 Comparing Bagging vs. Random Forest Importance

An interesting observation: variable importance looks different in Bagging vs. Random Forest!

- **Bagging**: A few features dominate (because trees are correlated)
- **Random Forest**: Importance is spread across more features (trees are diverse)

This smoother distribution in Random Forest reflects the fact that trees are considering different features, leading to a more complete picture of which features matter.

5 When Random Forest Doesn't Work Well

Random Forest is powerful, but not perfect. Here's when it struggles:

Warning

Too Many Irrelevant Features
If you have 1000 features but only 10 are actually useful, Random Forest may perform poorly.
Why? At each split, we randomly select m features. If most features are junk, there's a high probability that our random subset contains mostly junk features.
Solution: Use feature selection or PCA to reduce dimensionality first. Then apply Random Forest.

Example: When Bagging Beats Random Forest

You might encounter this surprising result:

Method	OOB Accuracy
Random Forest	78%
Bagging	82%

This often happens when:

- You have many noisy/irrelevant features
- Only a few features are truly predictive
- Random Forest keeps "missing" the good features due to random selection

In this case, remove irrelevant features and try again!

6 Handling Class Imbalance

Class imbalance occurs when one class has far more samples than another. This is extremely common in real-world problems.

Example: Class Imbalance Examples

- **Fraud detection**: 99.9% legitimate transactions, 0.1% fraudulent
- **Disease diagnosis**: 99% healthy, 1% have the disease
- **Spam detection**: 80% legitimate emails, 20% spam
- **Manufacturing defects**: 99.5% good products, 0.5% defective

6.1 Why Accuracy is Misleading

Warning

Accuracy can be useless for imbalanced data!
If 99% of emails are not spam, a model that predicts "not spam" for everything achieves 99% accuracy—but catches zero spam.
Solution: Use better metrics like F1-score, AUC-ROC, or precision/recall.

6.2 Three Approaches to Handle Imbalance

6.2.1 1. Undersampling (Reduce Majority Class)

Definition: Undersampling

Remove samples from the majority class until classes are balanced.

Methods:

- **Random Undersampling**: Randomly remove majority class samples
- **Near Miss**: Remove majority samples that are **far** from the decision boundary (keep the "hard" examples near the boundary)

Pros: Reduces training time
Cons: Throws away potentially useful data

6.2.2 2. Oversampling (Increase Minority Class)

Definition: Oversampling

Create more samples of the minority class until classes are balanced.

Methods:

- **Random Oversampling**: Duplicate minority samples (bootstrap with replacement)
- **SMOTE** (Synthetic Minority Oversampling Technique): Create **synthetic** samples by interpolating

between existing minority samples

Example: SMOTE Intuition

Instead of just copying minority samples:

1. Pick a minority sample x_i
2. Find its k nearest neighbors (also minority class)
3. Randomly pick one neighbor x_j
4. Create a new sample along the line between x_i and x_j:

$$x_{new} = x_i + \lambda \cdot (x_j - x_i) \quad \text{where } \lambda \in [0, 1]$$

This creates new, plausible minority samples that expand the feature space rather than just duplicating existing points.

6.2.3 3. Class Weighting

Definition: Class Weighting

Modify the loss function to penalize errors on the minority class more heavily.

Instead of treating all misclassifications equally, we weight them:

$$w_k = \frac{N}{K \times N_k}$$

where:

- N = total samples
- K = number of classes
- N_k = samples in class k

In sklearn, just set `class_weight='balanced'`:

```python
from sklearn.ensemble import RandomForestClassifier

# Automatically adjusts weights based on class frequencies
rf = RandomForestClassifier(
    n_estimators=100,
    class_weight='balanced',  # This handles imbalance!
    random_state=42
)
```

6.3 Practical Advice for Imbalanced Data

Key Information

When to use which approach:

- **Mild imbalance (60/40)**: Class weighting is often enough
- **Moderate imbalance (90/10)**: Try SMOTE or combination of over/undersampling
- **Severe imbalance (99/1)**: May need domain-specific techniques or anomaly detection

General rule: Always address imbalance! Even 40/60 splits benefit from handling.

7 Missing Data: Surrogate Splits

Decision trees have a unique way of handling missing data called **surrogate splits**.

7.1 The Problem

During prediction, what happens when a test sample has a missing value for the feature used to split a node?

Example: The Missing Data Problem

Your trained tree splits on `Blood Pressure` at the root. A new patient arrives, but their blood pressure wasn't recorded.
How do you route this patient through the tree?

7.2 The Solution: Surrogate Splits

Definition: Surrogate Split

A **surrogate split** is a backup split that produces similar results to the primary split.
During training, for each split on feature X, we find other features that would split the data similarly. These become surrogates.

7.2.1 How Surrogate Splits Work

1. **During Training**:
   (a) Find the best split (e.g., `Arteries Blocked > 2`)
   (b) Record how samples are distributed after this split
   (c) For each other feature, find the split that most closely mimics this distribution
   (d) Rank these alternative splits by similarity
2. **During Prediction**:
   (a) If the primary feature has a missing value, use the best surrogate
   (b) If that's also missing, try the second surrogate
   (c) Continue down the list

Example: Surrogate Split Example

Primary split: `Arteries Blocked`
- FALSE $\rightarrow$ 3 No, 1 Yes (heart disease)
- TRUE $\rightarrow$ 0 No, 2 Yes

Finding surrogates—test other features:
`Chest Congestion`:
- FALSE $\rightarrow$ 3 No, 2 Yes
- TRUE $\rightarrow$ 0 No, 1 Yes

Similarity: Very similar distribution! Only 2 “flips” needed.

`Good Blood Circulation`:

- FALSE $\rightarrow$ 2 No, 3 Yes
- TRUE $\rightarrow$ 1 No, 0 Yes

Similarity: Less similar, 6 “flips” needed.

Result: `Chest Congestion` becomes the primary surrogate for `Arteries Blocked`.

If a patient’s `Arteries Blocked` value is missing, we use their `Chest Congestion` value instead!

7.3 Benefits of Surrogate Splits

- **No imputation needed**: Tree handles missing data automatically
- **Interpretable**: Surrogates show which features are related
- **Works well with correlated features**: Multicollinearity actually helps find good surrogates!

Warning

Surrogate splits work best when features are correlated. If features are independent, good surrogates may not exist, and you’ll need other imputation methods.

8 Random Forests in Python

8.1 Basic Implementation

```python
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.inspection import permutation_importance
import numpy as np

# Load your data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create Random Forest classifier
rf_clf = RandomForestClassifier(
    n_estimators=100,         # Number of trees
    max_features='sqrt',      # Features per split (sqrt for classification)
    max_depth=None,           # Grow trees fully
    min_samples_leaf=1,       # Minimum samples per leaf
    oob_score=True,           # Compute OOB score
    class_weight='balanced', # Handle class imbalance
    n_jobs=-1,                # Use all CPU cores
    random_state=42
)

# Train
rf_clf.fit(X_train, y_train)

# Results
print(f"OOB Score: {rf_clf.oob_score_:.4f}")
print(f"Test Accuracy: {rf_clf.score(X_test, y_test):.4f}")
```

8.2 Getting Variable Importance

```python
# Method 1: MDI (built-in, but biased)
mdi_importance = rf_clf.feature_importances_

# Method 2: Permutation Importance (preferred)
perm_importance = permutation_importance(
    rf_clf, X_test, y_test,
    n_repeats=10,
    random_state=42
)

# Display results
import pandas as pd

importance_df = pd.DataFrame({
    'Feature': feature_names,
    'MDI': mdi_importance,
    'Permutation': perm_importance.importances_mean
}).sort_values('Permutation', ascending=False)

print(importance_df)
```

8.3 Handling Class Imbalance with SMOTE

```python
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline

# Create sampling pipeline
over = SMOTE(sampling_strategy=0.5)  # Oversample minority to 50%
under = RandomUnderSampler(sampling_strategy=0.8)  # Then undersample

# Apply to training data
X_resampled, y_resampled = over.fit_resample(X_train, y_train)
X_resampled, y_resampled = under.fit_resample(X_resampled, y_resampled)

# Train on balanced data
rf_clf.fit(X_resampled, y_resampled)
```

9 Key Takeaways

Key Summary

Random Forest:

- Enhancement of Bagging that adds feature randomization at each split
- At each split, randomly select m features and find best split among them
- Rule of thumb: $m = \sqrt{p}$ for classification, $m = p/3$ for regression
- Reduces tree correlation $\rightarrow$ better variance reduction

Key Hyperparameters:

- `n_estimators`: More is always better (no overfitting risk)
- `max_features`: Controls decorrelation
- Use OOB error for validation

Variable Importance:

- MDI: Fast but biased toward high-cardinality features
- Permutation: Slower but more reliable
- Use permutation importance for final analysis

Class Imbalance:

- Don't use accuracy! Use F1-score or AUC
- Options: Undersampling, Oversampling (SMOTE), Class weighting
- Always address imbalance, even for 40/60 splits

Missing Data:

- Surrogate splits provide automatic handling
- Find backup features that split similarly to primary feature
- Works best with correlated features

Table 2: *Comparison: Bagging vs. Random Forest*

Aspect	**Bagging**	**Random Forest**
Features per split	All p	Random subset m
Tree correlation	High	Low
Variance reduction	Good	Better
Importance distribution	Concentrated	Spread out
Works with many junk features	Better	May struggle

10 Practice Questions

1. **Decorrelation**: Explain why selecting a random subset of features at each split reduces correlation between trees.
2. **Hyperparameter Impact**: What happens to Random Forest if you set `max_features` equal to the total number of features? How does this compare to Bagging?
3. **MDI Bias**: You have two features: "Age" (continuous, 80 unique values) and "Gender" (binary, 2 values). Using MDI, which feature is likely to appear more important even if they have equal true importance? Why?
4. **Class Imbalance**: You're building a fraud detection model where only 0.5% of transactions are fraudulent. What metric should you use instead of accuracy? What technique(s) would you apply to handle the imbalance?
5. **Surrogate Splits**: Explain how surrogate splits help when a test sample has a missing value. When might surrogate splits fail to work well?
6. **Practical Scenario**: You build a Random Forest with 100 features, but only 5 are truly predictive. The OOB accuracy is 70%. You try Bagging and get 82%. Why might Bagging outperform Random Forest here?
7. **Code Question**: Write sklearn code to:
   - Train a Random Forest Regressor with OOB scoring enabled
   - Use $m = p/3$ features per split
   - Calculate and display permutation importance