Random Forest from Scratch

Bhanu Prasanna Koppolu

Table of contents

> **Note**
>
> Full implementation available at [GitHub - ML from Scratch]

Random Forest

The random forest just builds on top of the decision tree. We build multiple decision trees and do a **voting** to see which class is highest.

How Trees are Built

The thing is that we have a dataset, the tree is built using the dataset. Even slight changes to the dataset can lead to a completely different tree, which is what we're gonna do.

We sample data from our training data **with replacement** to get multiple training datasets, which we then use to build the decision trees. The part where we sample data with replacement is called **bootstrapping**.

Bootstrap sampling creates diversity, and averaging reduces variance.

Random Feature Selection

Why perform random feature selection at each split?

It helps prevent strong predictors from dominating all trees. Without it, trees would be too similar (highly correlated), reducing ensemble benefit.

The Problem with Decision Trees

Decision trees have a chance of **overfitting**: Low bias, high variance.

Random Forest **averages and reduces variance** without increasing the bias much.

Pros

- Robust to overfitting
- Handles non-linear relationships
- Feature importance is built-in
- Works with missing data
- Minimal hyperparameter tuning

Cons

- Less interpretable than single tree
- Slower prediction (must query B trees)
- Larger memory footprint
- Can struggle with extrapolation (the model would not perfectly comprehend a new case in future outside of the current data)

Implementation

Random Forest Classifier

```
class RandomForestClassification:

    def __init__(self, n_trees=50, max_depth=50, max_features=None,
                 min_sample_split=5, impurity="gini"):
        self.n_trees = n_trees          # Number of trees in forest
        self.max_depth = max_depth      # Max depth per tree
        self.max_features = max_features  # Features to consider at split (default: sqrt)
        self.min_sample_split = min_sample_split
        self.impurity = impurity        # "gini" or "entropy"

        # Initialize all decision trees
        self.trees = [DecisionTreeClassification(...) for _ in range(n_trees)]

    def _get_random_subsets(self, X, y, n_subsets, replacement=True):
        """Bootstrap sampling: create n_subsets with replacement."""
        pass

    def fit(self, X_train, y_train):
        """Train each tree on bootstrap sample with random feature subset."""
        if self.max_features is None:
            self.max_features = int(np.sqrt(n_features))

        subsets = self._get_random_subsets(X_train, y_train, n_subsets=self.n_trees)

        for i in range(self.n_trees):
            X_subset, y_subset = subsets[i]
            # Feature bagging: randomly select features
            idx = np.random.choice(range(n_features), size=self.max_features, replace=True)
```

```python
            self.trees[i].feature_indices = idx
            self.trees[i].fit(X_subset[:, idx], y_subset)

    def predict(self, X):
        """Majority voting across all trees."""
        y_preds = np.empty((X.shape[0], len(self.trees)))
        for i, tree in enumerate(self.trees):
            y_preds[:, i] = tree.predict(X[:, tree.feature_indices])
        # Return most common prediction for each sample
        return np.array([np.bincount(row.astype(int)).argmax() for row in y_preds])
```

Random Forest Regressor

For regression, instead of voting, we take the **mean** of all tree predictions:

```python
class RandomForestRegressor:

    def __init__(self, n_trees=50, max_depth=50, max_features=None,
                 min_samples_split=5, impurity="variance"):
        # Same structure as classifier but uses DecisionTreeRegressor
        self.trees = [DecisionTreeRegressor(...) for _ in range(n_trees)]

    def fit(self, X_train, y_train):
        """Same as classifier - bootstrap + feature bagging."""
        pass

    def predict(self, X):
        """Average predictions across all trees."""
        y_preds = np.empty((X.shape[0], len(self.trees)))
        for i, tree in enumerate(self.trees):
            y_preds[:, i] = tree.predict(X[:, tree.feature_indices])
        # Return mean prediction for each sample
        return np.array([np.mean(row) for row in y_preds])
```