Random Forests

A single decision tree is like a solo detective solving a case: sometimes spot-on, but often prone to overthinking or bias. Now imagine an entire team of detectives, each investigating different clues in parallel and reaching a consensus. That's a Random Forest - an ensemble of trees working together to produce stronger, more reliable predictions. Whether you're classifying emails or predicting household incomes, two brains (or more) are better than one.

In the previous post, we explored the inner workings of a decision tree. Since their introduction in the 1960s, several models have been developed that utilize multiple decision trees to predict or classify data. In this post, we'll study one such model: the random forest.

What is a Random Forest?

A random forest builds multiple decision trees using random subsets of data and features, then combines their predictions - by majority vote for classification or averaging for regression - to improve accuracy and reduce overfitting. Developed by Leo Breiman and Adele Cutler, the model is more robust to noisy data and often computationally faster than many other decision-tree-based ensemble methods.

The dataset used in this post is sourced from the UCI Machine Learning Repository and contains 32,561 observations and 15 variables in a .data file format. The data has been cleaned and feature-engineered to prepare it for machine learning. The outcome variable, Income, has been transformed into a binary label indicating whether an individual's annual income is above or below $50,000. Given this binary structure, we'll be using a classification random forest model.

Structure

Before proceeding, it's important to understand the key terminology associated with random forests. Grasping this terminology will help you better follow and explain how the model works:

- **Bootstrapping** is a resampling method that creates new datasets by drawing samples with replacement from the original dataset. Each new dataset is the same size as the original, but some rows are repeated while others may be missing.
- **Ensemble methods**, also known as ensemble learning, combine multiple individual models to produce more accurate and stable predictions than any single model alone.
- **Bagging**, short for bootstrap aggregating, is an ensemble technique that builds multiple models in parallel, each trained on a different bootstrapped sample of the data. Each model sees a slightly different view of the data, and their predictions are aggregated.

To bring it all together, random forests are an ensemble method that combines multiple decision trees using bagging, where each tree is trained on a different bootstrapped sample of the dataset.

A random forest is built using many decision trees. Each tree on its own may be a weak predictor, but when combined, they form a strong and reliable model. The final prediction is made by averaging (for regression) or majority vote (for classification) across all the trees in the forest.

To build each tree, the algorithm first generates a new training dataset by drawing a bootstrap sample. In addition to this, random forests introduce another layer of randomness: when splitting each node, only a random subset of features is considered. This prevents all trees from learning the same patterns, leading to decorrelated trees and a more powerful ensemble. Each tree’s predictions are then aggregated through bagging. For classification tasks, the trees vote on each instance, and the majority class is selected. For regression, the trees’ outputs are averaged to produce the final prediction. To better understand how these elements come together, let’s look at a visualization of a random forest in action:

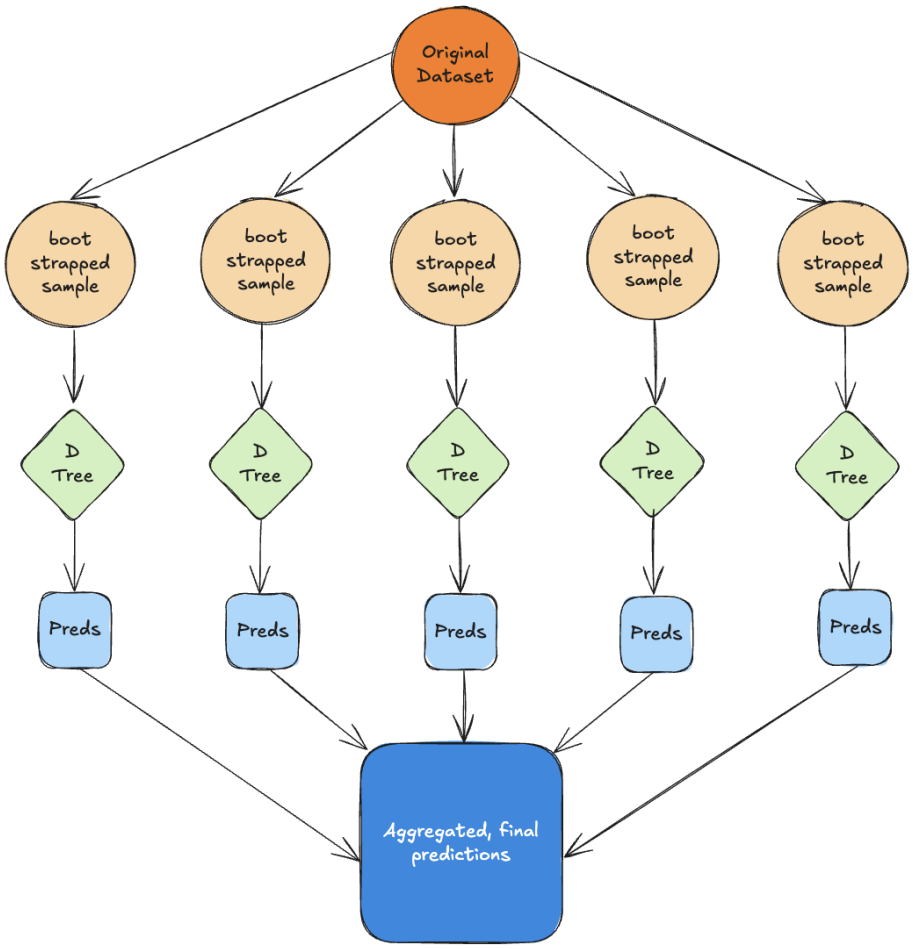

Figure 1

The strength of a Random Forest lies in two key ideas: low correlation between trees and reducing variance. Since random feature selection ensures that each tree learns something slightly different, averaging multiple trees enables the model to generalize patterns more effectively. While each tree may overfit its own bootstrapped data, averaging many such trees leads to a stable and accurate prediction.

Can We Build It? Yes, We Can!

With Python, initializing and training a random forest model is remarkably straightforward – it takes just two lines of code:

```
model = RandomForestClassifier(parameter_1,…,parameter_n)
```
or
```
model = RandomForestRegressor(parameter_1,…,parameter_n)
model.fit(X, y)
```

The first line sets up the model, and the second line uses the fit() function to learn the relationship between all the independent variables and the price. While you can create a random forest using default settings, such as RandomForestClassifier(), scikit-learn offers several parameters to help control the tree's complexity and prevent overfitting. Here are a few random forest–specific parameters that don't appear in a standard decision tree model:

1. n_estimators: Sets the number of trees in the forest.
2. bootstrap: Determines whether bootstrap samples are used when building trees. (boolean)
3. max_features: Controls the number of features considered when searching for the optimal split.

These parameters afforded to us by Python's Scikit-learn library make it pretty easy to control the forest's input, size, and structure, but what's going on inside the 'black box'? In machine learning, when we say something is a 'black box', we mean that the internal process is hidden from us. Though scikit-learn makes it easy to use this function, we still want to understand what's going on inside.

Math in the Black Box

The goal of our random forest is to predict an individual's income class based on predictors such as education, marital status, occupation, and more. Since our dataset's target variable is a binary income label, we're using a classification random forest model. This model doesn't introduce new mathematics beyond what we've already seen in decision trees, but it does involve more complex logic, given that it's an ensemble method. Let's revisit the main steps:

1. The dataset is used to create bootstrapped samples.
2. Each tree is trained on a unique bootstrapped sample. At each node split, only a random subset of features is considered.

3. Each tree makes its own prediction.
4. The random forest aggregates predictions from all trees: in classifiers, it tallies the votes and selects the majority class.

Computational Mathematics

Let's manually build a random forest. The dataset has already been cleaned and feature-engineered. If you're an aspiring data scientist, I encourage you to walk through the entire pipeline, from raw .data file to a well-structured DataFrame suitable for modeling.

First, we need to build a function that draws bootstrap samples. Below, I've used NumPy's `random.choice()` to select indices from the dataset with replacement randomly. We also store the target labels for these rows so that the forest can learn from the features and their associated outcomes:

```
def bootstrap_sample(X, y):
# size of the sample is the number of rows in the original dataset
    n_samples = X.shape[0]
    indices = np.random.choice(n_samples, size=n_samples, replace=True)
    return X[indices], y[indices]
```

Next, we need to build a decision tree to train on each sample. The construction of the decision tree is identical to the one from my previous post. If your goal is not to implement the forest manually, I recommend using `DecisionTreeClassifier(param_1, ..., param_n)`. Otherwise, refer to the Computational Mathematics section of my Decision Trees post to learn how to construct a tree from scratch.

In a nutshell, decision trees recursively select the best feature to split the data, continuing this process until a stopping criterion is met. The "best" feature is the one that yields the lowest weighted average Gini impurity across potential splits.

Once the decision tree model is built, we incorporate ensemble logic. The `bootstrap_sample()` function provides the data for `build_decision_tree()`, and the

resulting decision tree is stored as a nested Python dictionary called `tree`. Each tree is then added to a list, as shown below:

```
def build_random_forest(X, y, n_trees=10, max_depth=None,
max_features='sqrt'):
    forest = []
    for _ in range(n_trees):
        X_sample, y_sample = bootstrap_sample(X, y)
        tree = build_decision_tree(X_sample, y_sample,
max_depth=max_depth, max_features=max_features)
        forest.append(tree)
    return forest
```

Let's break down the parameters of this function:

- `X` and `y` represent the dataset's features and labels.
- `n_trees` sets the number of trees in the forest, equivalent to `n_estimators` in sklearn.
- `max_depth` limits how deep each decision tree can grow. While deep trees can lead to overfitting, random forests mitigate this risk due to bootstrapping and feature randomness. Allowing deeper trees can reduce bias, which is why I've opted not to set a maximum depth here.
- `max_features='sqrt'` means that at each split, only a random subset of features is considered. Specifically, it uses the square root of the total number of features. In our Adult dataset, we have 14 features, and the root of 14 ≈ 3.74. The model floors this to 3 features at each split. This technique is standard in random forest construction and helps diversify the trees.

Here's how this setup translates into an actual scikit-learn model:

```
rf = RandomForestClassifier(n_estimators=10, bootstrap = True,
max_features = 'sqrt', random_state=42)
rf.fit(X_train, y_train)
```

Predictions

Now that we've built a manual random forest, how do we use it to predict whether an individual earns more or less than $50K?

We use the `predict_forest()` function to generate predictions from each decision tree in the ensemble. For every observation, we collect predictions from all trees and select the most common class using majority voting:

```
def predict_forest(trees, X):
    # Get predictions from each tree
    tree_preds = np.array([[predict_tree(tree, x) for tree in trees] for x in X])

    # Majority vote across rows
    y_pred = []
    for row in tree_preds:
        majority_vote = Counter(row).most_common(1)[0][0]
        y_pred.append(majority_vote)
    return np.array(y_pred)
```

The `tree_preds` matrix will have 25,436 rows (one per observation in the training dataset) and 10 columns (one per tree). For each row, we take the most frequently predicted label, either 0 or 1, and store it in our final prediction list, `y_pred`.

To call the function, pass in the forest and the feature matrix to predict on:

```
manual_rf = build_random_forest(X_train, y_train, n_trees=10, max_depth=None, max_features='sqrt')
y_pred_manual = predict_forest(manual_rf, X_test)
```

Once you run this code, the random forest model will have made its predictions!

Evaluation

The model has made its predictions; now let's evaluate how well it performed against actual values. We use key evaluation metrics to measure its effectiveness: accuracy, precision, recall, and F1-score (for a refresher, refer to my Introduction to Machine Learning post).

Before diving in, it's worth noting that while the evaluation metrics for the manual and scikit-learn models are similar, they are not identical. This is the first post where my manually built and sklearn-based models don't produce the same results, a discrepancy likely caused by the two layers of randomness inherent in random forests: bootstrapped data and random feature selection. However, since both models achieve the same accuracy, we'll proceed by interpreting the results of the manual model.

Here is the confusion matrix for the manual random forest model predicting whether an individual's annual income is above or below $50,000 (stored in `y_pred_manual`):

```
[[4492  343]
 [ 596  928]]
```

This matrix tells us that out of 6,359 test datapoints:

- 4,492 people earning below $50,000 were correctly classified (true negatives)
- 928 people earning above $50,000 were correctly identified (true positives)
- 343 individuals were incorrectly predicted to earn more than $50,000 (false positives)
- 596 individuals who earn more than $50,000 were mistakenly predicted to earn below $50,000 (false negatives)

While the model demonstrates a great ability to distinguish between individuals with high and low incomes, there is a slight imbalance in its predictions. The high number of false negatives (596) suggests that the model often fails to identify individuals with annual incomes greater than $50,000. Let's break down the key metrics:

1. Accuracy: 85%

The model correctly classified 85% of all individuals. This is a strong result overall, especially for a binary classification problem with imbalanced classes. (0: 4835, 1: 1524)

2. Precision:
    - ≤ $50K (class 0): 88%
    - > $50K (class 1): 73%

Precision for class 0 is relatively better than class 1. Precision for high-income earners is 73%, meaning when the model predicts someone earns more than $50K, it's correct about three-quarters of the time. This is quite good, though not perfect, and indicates some over-prediction of high earners.

3. Recall:
    - ≤ $50K (class 0): 93%
    - > $50K (class 1): 61%

The model is highly effective at identifying individuals who earn $50K or less, with a recall of 93%, meaning it correctly captures the vast majority of low-to-mid income earners. In contrast, the recall for high-income individuals is 61%, so the model misses about 4 in 10 actual high earners: a meaningful gap if catching high-income individuals is the goal.

4. F-1 score: 85%
    - ≤ $50K (class 0): 0.91
    - > $50K (class 1): 0.66

The F1-score for individuals earning less than $50K is 0.91, reflecting the model's strong and balanced performance in precision and recall for the majority class. However, the F1-score for high-income individuals is 0.66, indicating the model still struggles to identify and predict the minority class consistently.

Interpretation

From evaluating our metrics, it's clear that the model is highly effective at identifying individuals who earn $50,000 or less, as shown by the high recall (93%) and strong F1-score (0.91) for class 0. In contrast, it struggles more with detecting high-income individuals, with a noticeably lower recall (61%) and F1-score (0.66) for class 1. In practical terms, this means the model is **more conservative** in predicting high earners and tends to err on the side of predicting

lower income. This behavior could stem from class imbalance, as the majority of individuals in the dataset fall into the below $50,000 category, or from features that are less informative in distinguishing high earners from the rest.

In real-world applications such as income prediction, not all classification errors are equally costly, a concept known as **asymmetric error cost**. For instance, falsely predicting someone earns more than $50K (a false positive) might lead to inappropriate targeting in financial marketing or risk assessment, while failing to identify a true high earner (a false negative) could result in missed business opportunities, such as credit offers, recruitment outreach, or personalized services. Traditional metrics like accuracy treat all errors the same, but in practice, the consequences of overestimating vs. underestimating income can diverge sharply, especially when these predictions are used to drive decisions with financial or strategic impact.

Overall, while the model provides strong baseline performance, especially for the dominant class, there is meaningful room for improvement in capturing the minority class. Future enhancements might include adjusting the classification threshold to favor recall for high earners, incorporating class weighting, or using different sampling strategies to mitigate imbalance. These steps could help balance the model's predictive strengths and align it more closely with the real-world goals of the income prediction task.

Conclusion

Random forests offer a powerful, flexible approach to classification, particularly when individual decision trees fall short. By combining the outputs of multiple trees trained on random subsets of data and features, the ensemble reduces variance, increases robustness, and achieves stronger performance than any one tree on its own.

In this post, we moved beyond using built-in libraries and built a random forest manually, step by step. We explored the logic behind the model, coded key functions from scratch, and used the resulting ensemble to predict whether individuals earn more or less than $50,000 per year.

The evaluation revealed that while the model excels at predicting low- to mid-income earners, it struggles somewhat with identifying high earners, a common challenge in imbalanced datasets. Still, with an overall accuracy of 85% and strong metrics for the majority class, the model presents a solid baseline. With further tuning, particularly to enhance performance for the minority class, it has the potential to be even more effective in real-world applications.

In short, random forests prove that many weak learners can come together to form a strong one - a principle at the heart of modern ensemble learning. By building it yourself, you gain not only insight into the math and mechanics but also unlock the full creative power to tune, customize, and improve your models.

In this post, we've learned:

- What random forests are
- The structure of a random forest
- How to build a random forest using the scikit-learn library in Python
- How to build a random forest manually in Python
- How to predict values using scikit and manually in Python
- How to evaluate the model using a confusion matrix, accuracy, precision, recall, and F1 metrics with Scikit-learn
- How to interpret model evaluations

That's a wrap on random forests! Next, we'll study another ensemble learning model, Gradient Boosting Machines (GBMs). Stay tuned!