Building a Random Forest Regressor From Scratch

Tanishq Kumar

September 13, 2024

Abstract

This paper showcases my process of building a Random Forest Regressor from scratch (not just using a library). The purpose of this paper is to demonstrate my understanding of decision trees, random forests, and the mathematical foundations behind them. The structure of this paper will be an introduction into the paper, the mathematics that underlies Random Forest Regression, and finally a detailed explanation of the structure and functionality of my code.

1 Introduction

A Random Forest Regression model combines multiple traditional decision trees to create a large single model. Each tree in the forest builds from a different subset of the data and makes its independent prediction. The final prediction for input is based on the average of all the individual trees' predictions, which lowers variance, and adds randomness to the process, preventing overfitting.

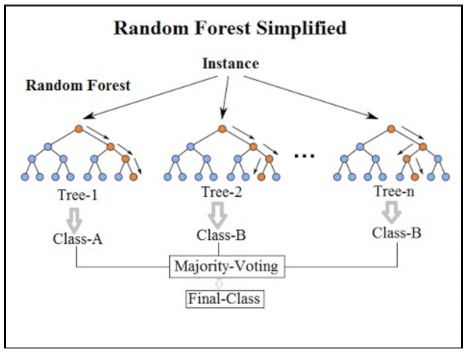


Figure 1: Random Forest Visualised

2 Mathematical Foundations

2.1 Mean Squared Error (MSE)

In regression tasks, the goal is to minimise the prediction error. In our decision tree implementation, we use the Mean Squared Error (MSE) to measure this. The formula for MSE is given by:

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

Where:

y_i is the actual value of the target.

$\hat{y}$ is the predicted value (usually the mean of the target values in the region).

N is the number of samples.

Our goal is to minimise MSE at each node of the tree, leading to a better overall prediction.

2.2 Recursive Splitting in Decision Trees

A decision tree recursively splits the dataset based on feature values that minimise the MSE. At each node, the best split is selected by examining all possible splits of the features. For each split, the weighted MSE of the left and right child nodes is calculated:

$$\text{Weighted MSE} = \frac{N_L}{N} \cdot \text{MSE}_L + \frac{N_R}{N} \cdot \text{MSE}_R$$

Where:

N_L and N_R are the number of samples in the left and right child nodes, respectively.

MSE_L and MSE_R are the Mean Squared Errors for the left and right child nodes.

N is the total number of samples in the current node.

3 Implementation Structure

3.1 Starting with Data

To get started a dataset is needed. The dataset contains various *features* (independent variables) and *target values* (dependent variables). The task is to use past data to predict the house price for a new set of features.

3.2 Splitting the Data

To build a robust model, the data is divided into two sets: a *training set* and a *test set*. The training set is used to train the model to predict the target value based on the features, while the test set is used later to evaluate the performance of the model on unseen data.

3.3 Defining and Refining Splits

Once the training data is prepared, decision trees are built. A *decision tree* is a model that recursively splits the data into smaller groups based on different feature values. At each split, the tree chooses the best feature to split on by minimising the *Mean Squared Error (MSE)*. At each node, the MSE is calculated for potential splits, and the data continues splitting until it reaches a defined stopping criterion.

3.4 Averaging MSE for Predictions

As the tree grows, each branch represents a subset of data with increasingly accurate predictions. At the leaves (end nodes) of the tree, predictions are made based on the *mean* of the target values for that branch. The tree chooses the split with the lowest MSE, leading to more refined and accurate predictions. In a *Random Forest*, this process is repeated across multiple decision trees. Each tree is trained on a random bootstrapped subset of the data (with replacement to increase randomness, as well as allows for the existence of an Out-of-Bag dataset which can be used for further testing), and the final prediction is the average of all the individual tree predictions. This approach helps reduce variance and improves the model's ability to generalise.

3.5 Making Predictions on the Training Data

During training, each decision tree learns how to follow a chain of splits based on the training data. When making a prediction on new data, the tree follows the same series of splits until it reaches a leaf node, where the prediction is made. For example, for the dataset I am using in my project, the tree might follow a series of questions like:

Is the house older than 20 years?

Does it have more than 5 rooms?

Is it located in California? Once it reaches the leaf node, it outputs the average target value for similar data points in that branch.

3.6 Testing the Model

After training, the model is evaluated on the *test set*. During testing, each decision tree uses the same series of splits to navigate the trees and generates a prediction based on the leaves. The model's performance is measured by comparing the predicted values to the actual values in the test set using *Mean Squared Error (MSE)*.

3.7 Visualising the Results

Finally, the results are visualised by plotting both the actual values and the predicted values, allowing a visual comparison to see how closely the model's predictions match the real outcomes.

4 Going through the Code

4.1 Decision Tree: `decision_tree.py`

The decision tree is the core component of the Random Forest. In this file, we define a class `DecisionTreeRegressor` that recursively builds the tree by finding the best split based on the MSE.

```
class Node:
```

This class represents a node in the decision tree. It is mainly used to initialise all nodes. Nodes can be:

internal: contain features/thresholds.

leaf nodes: contain predicted values.

```
class DecisionTreeRegressor:
```

This class is responsible for constructing a decision tree model that recursively splits the data based on feature values to minimise prediction error (typically using Mean Squared Error), enabling accurate predictions for regression tasks. It has various functions within it, in order to conduct the regression:

```
    def __init__(self, min_samples_split=2, max_depth=100):
```

This function initialises various parameters:

`min_sample_split`: this specifies the minimum number of samples required to split a node.

`max_depth`: specifies the maximum depth a tree can be.

`root`: sets the root node of the tree.

```
    def fit(self, X, y):
```

This function starts the tree building process by calling `grow_tree` method on the training data X, and targets Y. It also:

assigns the root node to `self.node`:

hands the training data to grow the tree.

```
    def _grow_tree(self, X, y, depth=0):
```

This is a very critical function of the decision tree. It recursively builds the tree by finding the best feature and threshold to build on at each node. It does this by:

Setting the base condition for recursion: if `num_samples >= min_samples_split` and `depth < max_depth`, then stop split.

Find the best split: done by calling the `best_split` function.

It does this recursively until the stopping condition is met.

Finally it returns the internal nodes and returns the leaf nodes.

```
    def predict(self, X):
```

This function makes predictions for the input data X, by traversing the tree from root node to leaf node via the `traverse_tree` method.

```
    def traverse_tree(self, x, node):
```

This function recursively travels the tree to make predictions for a single sample, X. It simply checks:

If the node if a leaf node - at which case it knows it has reached the end of that path, and then it returns the prediction stored at that node.

Otherwise, it checks whether the input feature value is less than or equal to the node's threshold and moves left or right accordingly.

4.2 Random Forest: `random_forest.py`

In the `random_forest.py` file, we implement the `RandomForestRegressor` class. This aggregates predictions from multiple decision trees trained on different subsets of the data.

```
    def __init__(self, n_trees=100, min_samples_split=2 ..):
```

This function initialises the random forest regressor with specified hyper-parameters.

Stores the number of trees `n_trees`, the minimum number of samples required to split a node `min_samples_split`, and the maximum depth of the trees `max_trees`,

Initialises an empty list `self.trees` to hold the individual decision trees

```
    def fit(self, X, y):
```

Trains the random forest by fitting each decision tree to a bootstrap sample of the training data.

Resets `self.trees` to an empty list to store new trees.

Iterates `n_trees` times with a progress bar using tqdm:

Calls the `build_tree` method to create and train a decision tree on a bootstrap sample.

Appends the trained tree to `self.trees`

```python
    def _build_tree(self, X, y):
```

Builds and trains a single decision tree using a bootstrap sample of the data.

Generates bootstrap indices by randomly selecting samples with replacement from the dataset.

Creates the bootstrap samples `X_sample` and `y_sample` using these indices.

Initialises a `DecisionTreeRegressor` with the same `min_samples_split` and `max_depth`

Fits the decision tree to the bootstrap sample and returns the trained tree.

```python
    def predict(self, X):
```

Predicts target values for new input data by averaging predictions from all decision trees in the forest.

Uses a list comprehension to collect predictions from each decision tree in `self.trees` for the input X.

Converts the list of predictions into a `NumPy` array `tree_preds`.

Computes the mean of the predictions across all trees using `np.mean(tree_preds, axis=0)` to produce the final prediction.

4.3 Utility Functions: `utils.py`

The `utils.py` file contains essential functions for decision tree regression, specifically for calculating mean squared error and determining the best feature and threshold for splitting data to minimise prediction error.

```python
    def calculate_mse(y):
```

Calculates the mean squared error (MSE) of a given set of target values y.

Checks if the input array y is empty; if it is, returns 0 to avoid division by sero or undefined operations.

Computes the MSE by calculating the mean of the squared differences between each element in y and the mean of y.

```python
    def best_split(X, y):
```

Calculates the mean squared error (MSE) of a given set of target values y.

Iterates over each feature in X, sorting the data based on the current feature to consider potential split points at midpoints between consecutive unique values.

For each potential split, divides the data into left and right subsets, calculates the weighted MSE for these subsets, and updates the best split parameters if the current split yields a lower MSE than previous splits.

4.4 Main Script (`main.py`)

The main script serves as the entry point for the project by loading the dataset, training the custom and Scikit-Learn random forest models, evaluating their performance, and visualising the results.

4.4.1 Structure and Explanation

Import Statements Imports necessary modules and classes required for data handling, model training, evaluation, and visualisation.

Imports the custom `RandomForestRegressor` from `random_forest.py` and Scikit-Learn's version for comparison.

Imports datasets, model selection tools, evaluation metrics, and plotting libraries to manage the workflow.

Loading the Dataset Loads the California housing dataset, which will be used for training and evaluating the random forest models.

- Calls `fetch_california_housing()` to retrieve the dataset containing features and target values.
- Separates the data into input features `X` and target variable `y`.

4.4.2 Integration of Other Files

- `random_forest.py`: The custom `RandomForestRegressor` class defined here is imported and used to create and train a random forest model on the dataset.
- `decision_tree.py`: Contains the `DecisionTreeRegressor` class, which is utilised internally by the `RandomForestRegressor` to build individual decision trees.
- `utils.py`: Provides utility functions like `calculate_mse` and `best_split` that are essential for training the decision trees.

How the Files Work Together

- The main script initialises and trains the custom random forest model using the classes and functions from the other modules.
- The `RandomForestRegressor` relies on `DecisionTreeRegressor`, which in turn uses utility functions from `utils.py` to build and train the decision trees.
- This modular approach allows for a clear separation of concerns and easier maintenance.

5 Results

In this section, we compare our custom `RandomForestRegressor` with Scikit-Learn's `RandomForestRegressor` using the California Housing dataset. The Mean Squared Error (MSE) was used as the evaluation metric to measure the models' predictive accuracy on the test set. You can see all the predicted values vs the actual values in Figure 2.

5.1 Model Performance

- **Custom Random Forest Mean Squared Error:** 0.3112
- **Scikit-Learn Random Forest Mean Squared Error:** 0.2616

5.2 Interpretation of Results

The results suggest that the Scikit-Learn model has a better predictive performance compared to the custom model on the same dataset. This can be seen by the tighter prediction in image (b) - the Scikit-Learn model, vs the slighly looser spread in image (a) - my Custom Model. Here are some reasons that I believe could be the reason for this:

- The number of trees: due to personal compute restrictions, my custom model was made with 5 trees, whereas the Scikit-Learn model was made with 100.
- The Scikit-Learn library is a highly optimised and mature library, with various extra enhancements that are not implemented in my custom model.
- This modular approach allows for a clear separation of concerns and easier maintenance.

5.3 Visualisation of Results

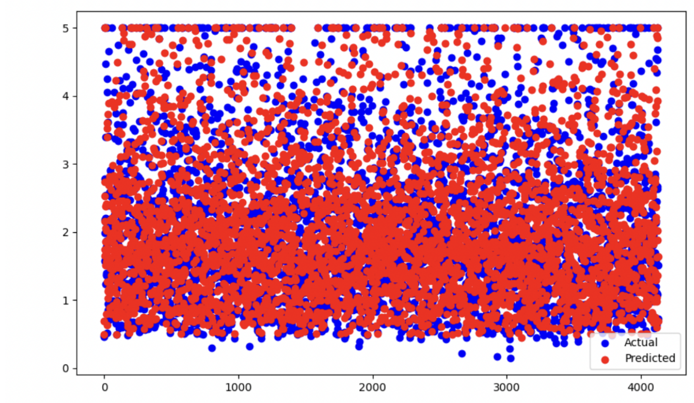
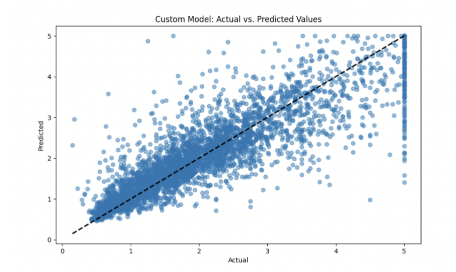
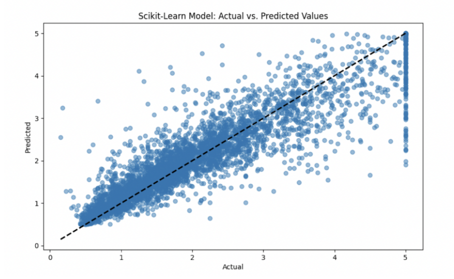


Figure 2: Plot of Predicted vs Actual Prices Using Custom Model



(a) Custom Model's Prices vs Actual Prices



(b) Scikit-Learn Model's Prices vs Actual Prices