Gradient Boosting Machine

A single decision tree is like a lone detective solving a mystery: swift to act but sometimes overconfident in early leads. Now imagine a team of detectives working sequentially – each reviewing the case, learning from previous mistakes, and sharpening the investigation. That's a **Gradient Boosting Machine**, a model that builds trees one after another, where each tree corrects the missteps of the last, gradually closing in on the truth.

In the previous post, we explored how Random Forests combine many independent trees grown in parallel to mitigate bias and variance. This time, we'll examine a different approach: **boosting**. Boosting is an ensemble learning technique that combines a series of weak learners into a strong learner, minimizing training errors by focusing increasingly on the hardest-to-predict data. Specifically, we'll explore the Gradient Boosting Regressor, which trains trees in an ordered sequence to produce highly accurate predictions.

What is Gradient Boosting?

Gradient Boosting builds an ensemble of weak learners, typically shallow decision trees, in a sequential manner. Each new model is trained to reduce the loss function of the ensemble by fitting the negative gradient (also known as pseudo-residuals) of the loss with respect to current predictions. This approach iteratively corrects errors made by previous models using gradient descent.

How does this differ from boosting in general? Traditional boosting methods like AdaBoost adjust the weights of misclassified samples to focus learning. Gradient boosting instead directly models the gradient of a loss function, fitting new learners to the residual errors, which are the differences between true values and predictions of previous models.

Gradient Boosting Machines (GBMs) belong to this family of algorithms, building the ensemble stage-wise and continually refining predictions by focusing on residual errors. Both regression

and classification tasks benefit from Gradient Boosted Decision Trees; popular implementations include XGBoost, LightGBM, and CatBoost. Today, we'll focus on the Gradient Boosting Regressor applied to predicting Customer Lifetime Value (CLV). For context, CLV is the total revenue a customer generates over their engagement with a business.

Structure

A Gradient Boosting Machine assembles its predictive power from a sequence of decision trees, but unlike Random Forests, these trees are grown sequentially, not independently or in parallel. Each tree is designed specifically to address the weaknesses of the current ensemble, honing in on data points where errors remain high and enhancing prediction accuracy iteratively.

At the core of this process is gradient descent, an optimization technique that guides the model to minimize a chosen loss function. For regression problems, this is often the Mean Squared Error (MSE); for classification, cross-entropy loss or similar functions are used. The model begins with an initial prediction and then computes the negative gradients of the loss function for the current predictions. These negative gradients indicate the direction and magnitude of the steepest loss reduction.

At each iteration, a new decision tree is trained to predict these negative gradients (also called pseudo-residuals). For example, if the true CLV is $2,000 but the model currently predicts $1,500, the residual is +$500. The next tree learns to predict this residual using the same input features, nudging the ensemble's prediction closer to the true value. After each step, the residuals are recalculated based on the updated predictions, focusing training on the remaining errors.

The **learning rate** scales each tree's contribution, preventing overfitting by ensuring gradual, cautious updates. Instead of correcting all errors at once, the model takes many small, informed steps toward a better solution.

The final prediction sums the initial estimate and all scaled tree outputs, each making targeted corrections. Through this stepwise refinement, gradient boosting transforms many simple models into a highly accurate predictor. Here is what this process looks like:

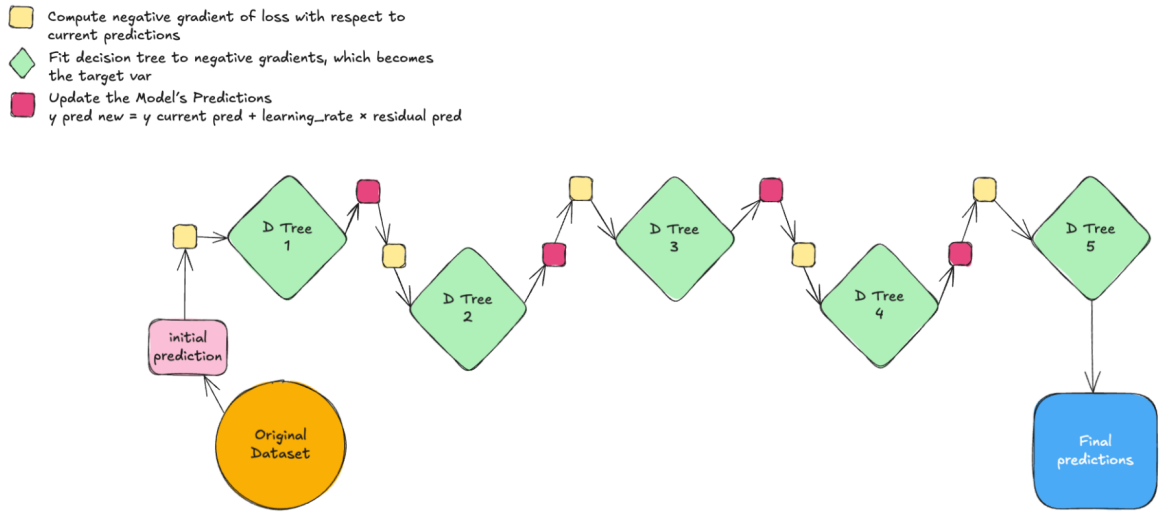

Figure 1

Can We Build It? Yes, We Can!

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

```
model = GradientBoostingClassifier(parameter_1,…,parameter_n)
```

or

```
model = GradientBoostingRegressor(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 gradient boosting model using default settings, such as GradientBoostingRegressor(), sklearn offers several parameters to help control the tree's complexity and prevent overfitting. Here are a few common gradient boosting model–specific parameters:

1. n_estimators: Sets the number of trees in the forest.
2. learning_rate: Controls the speed of adjustments in the gradient descent.
3. max_depth: Limits the maximum height of the tree.

These parameters afforded to us by Python's Scikit-learn library make it pretty easy to control the gradient boosting model'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 sklearn makes it easy to use this function, we still want to understand what's going on inside.

Math in the Black Box

Our goal is to accurately predict the Customer Lifetime Value (CLV) of an insurance company's clients using a variety of predictors, including coverage type, income, and the number of policies. Since CLV is a continuous variable, we employ a regression gradient boosting model for this task.

How Regression Decision Trees Work

A regression decision tree estimates continuous outcomes by repeatedly splitting the dataset into smaller, more homogeneous groups based on the target values.

1. Splitting: At each node, the algorithm selects the feature and split point that leads to the greatest reduction in the sum of squared errors (SSE) within the resulting subsets.
2. Leaf Assignment: The Process continues recursively until a pre-set stopping criterion is met.
3. Prediction: The predicted value for any leaf is the average of the target values of the training samples in that group.

Let me illustrate this process:

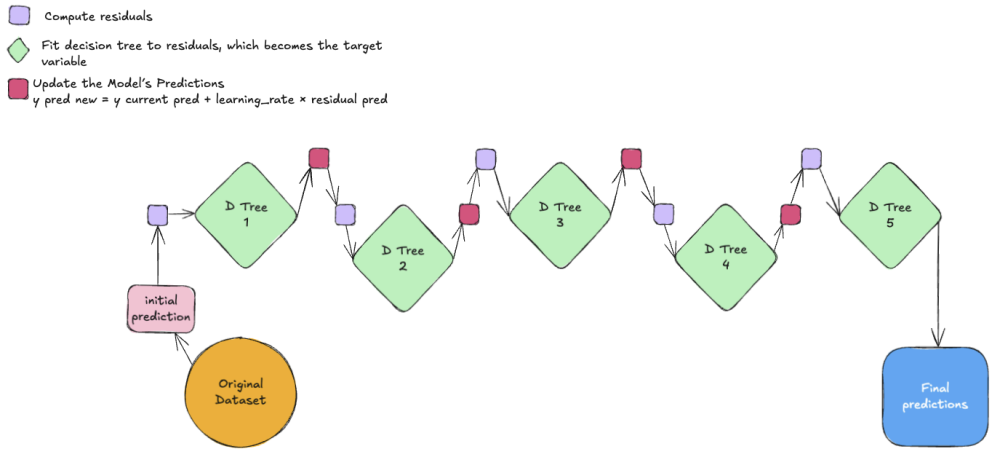

Figure 2

Gradient Boosting Process

Gradient boosting works by building an ensemble of decision trees one at a time, where each new tree helps to correct the errors made by the combined predictions of the trees before it. The process begins with a baseline prediction for every data point: sklearn's function generally uses the mean of the training dataset's target variable. This provides a simple baseline estimate, serving as a starting point for the boosting process. This initial, feature-agnostic prediction establishes a starting point for the model.

Next, the algorithm evaluates the gradient of the loss function with respect to the current model's predictions. The loss function of our GradientBoostingRegressor is the Mean Squared Error loss function:

$$L_{MSE} = (y_i - \widehat{y_i})^2$$

To find the negative gradient, we must calculate the negative partial derivative of our loss function for the prediction (y hat):

$$-\frac{\partial L_{MSE}}{\partial \widehat{y_i}} = -\frac{\partial (y_i - \widehat{y_i})^2}{\partial \widehat{y_i}}$$

$$= -2(y_i - \widehat{y_i})$$

To focus on the gradient, let's omit the constant factor -2, which scales (adjusts the step size of) the gradient but doesn't affect the direction at all. We are left with:

$$(y_i - \widehat{y_i})$$

Which is the expression for the residual. This tells us that in sklearn's GradientBoostingRegressor function, residuals and the negative gradient are actually the same. However, keep in mind that with other loss functions, the negative gradient may not be identical to the raw residuals as seen above.

Each tree is trained to predict this negative gradient. Before each new tree is built, the algorithm recalculates how much the current model's predictions need to change to most efficiently reduce the overall error, according to the chosen loss function. Once trained, the predictions of this tree are scaled by the learning rate parameter and added to the model's current predictions. This process is repeated iteratively: compute gradients → fit a new tree to them → scale its output → update the predictions, until a stopping criterion is met.

While studying this model, I would often refer to my notes on logistic regression to compare gradient descent and gradient ascent. I realized that gradient descent minimizes a loss function by moving in the direction of the negative gradient. In contrast, gradient ascent maximizes a function (like the log-likelihood in logistic regression) by moving in the direction of the positive gradient. Mathematically, one subtracts the gradient, the other adds it. With gradient descent, the goal is to find the minima (lowest point) of a loss function to reduce errors; with gradient ascent, the goal is to find the maxima (highest point) of a log-likelihood function to identify the most probable coefficients.

Computational Mathematics

With this context in mind, let's manually build a gradient boosting regressor. The dataset has already been cleaned and feature-engineered to feed the model. Be sure to check these sections out in the code file.

To manually build a gradient boosting regressor step-by-step, we start by defining helper functions the tree will need. First up is the prediction at a leaf node, which is simply the average target value of the data points assigned to that leaf. This function also serves to compute the initial prediction for the very first tree, namely, the mean of the target variable across the entire training dataset.

```
def prediction(y):
      return np.mean(y)
```

We also need a function that computes the sum of squared errors, which is the sum of all residuals squared. It can be written as:

```
def SSE(y):
      return ((y - y.mean()) ** 2).sum()
```

Next, we build a function that computes the negative gradient of the loss function, which we've learnt is essentially the residual:

```
def compute_neg_gradient(y_true, y_pred):
      return y_true - y_pred
```

Now, let's build a function, `best_sse_split`, that executes a regression decision tree's feature selection logic. Decision trees for regression operate much like classification trees in that they evaluate potential splits at the midpoints between consecutive values of each feature, but instead of choosing splits that best separate classes, they select the ones that minimize the SSE.

To identify possible split thresholds, we'll sort and extract all unique values of each feature, and calculate midpoints like this:

```
sorted_idx = np.argsort(X_columns)
X_columns_sorted = X_columns[sorted_idx]
y_column_sorted = y[sorted_idx]
unique_vals = np.unique(X_columns_sorted)
thresholds = (unique_vals[:-1] + unique_vals[1:]) / 2
```

For each threshold, the feature is split into a left side, where the feature value is less than or equal to the threshold, and a right side where the feature value is greater:

```
left_mask = X_sorted <= threshold
right_mask = ~left_mask
y_left = y_sorted[left_mask]
y_right = y_sorted[right_mask]
```

Now, we need to measure the "goodness" of each split using the SSE deviations from the mean of the left and right child nodes. This is calculated by:

```
sse_left = SSE(y_left)
sse_right = SSE(y_right)
weighted_sse = sse_left + sse_right
if weighted_sse < best_sse:
      best_sse = weighted_sse
     best_threshold = threshold
```

The algorithm checks all possible thresholds and retains the split that leads to the lowest weighted sum of squared errors. This means it finds the split where the predictions of the left and right child nodes are as close as possible to the actual values within those nodes, resulting in more accurate predictions when the tree is used to predict unseen data. If no threshold improves the SSE, the function returns None to indicate that no further splitting is possible.

The helper functions have been constructed; now it's time to implement the recursive core of the regression decision tree in `build_decision_tree`. The function first checks if any stopping criterion is met:

- The maximum allowed depth of the tree is reached (max_depth == 0).
- There are not enough samples to split (len(X) < min_samples_split).
- All target values in the node are identical (np.std(y) == 0).

If any of these are true, the node becomes a leaf. Its prediction is the mean of its current target values:

```
if (max_depth == 0 or len(X)< min_samples_split or np.std(y) == 0):
        return {'leaf': True, 'value': prediction(y)}
```

The function then evaluates each feature in the data by calling best_sse_split on each feature to look for the best threshold, based on minimizing the SSE. It keeps track of the feature and threshold combination with the lowest SSE.

```
n_samples, n_features = X.shape
best_feature, best_threshold, best_sse = None, None, float('inf')
for feature in range(n_features):
      threshold, sse = best_sse_split(X[:, feature], y)
      if threshold is not None and sse < best_sse:
            best_feature = feature
            best_threshold = threshold
            best_sse = sse
```

If no split improves SSE (either all values are the same, or all samples fall on one side for every possible threshold), the function again returns a leaf node with the mean target as prediction.

```
if best_feature is None:
      return {'leaf': True, 'value': prediction(y)}
```

Using the best feature and threshold found, the data is divided into left and right branches. The function then recursively builds left and right subtrees, reducing the maximum allowed depth by 1 at each level:

```
left_mask = X[:, best_feature] <= best_threshold
right_mask = ~left_mask
left_subtree = build_decision_tree(X[left_mask], y[left_mask],
max_depth-1, min_samples_split)
right_subtree = build_decision_tree(X[right_mask], y[right_mask],
max_depth-1, min_samples_split)
```

The `build_decision_tree` function returns a tree node as a dictionary containing whether the node is a leaf (boolean), feature and threshold split, left and right subtrees. By recursively calling this function for each left and right branch, we are assembling the entire tree as a nested dictionary structure. Here is an example output:

```
'feature': 19,
'threshold': 0.5,
'left': {'leaf': True, 'value': -5053.183893302541},
'right': {'leaf': True, 'value': -4941.1732656999775}
```

We've built a function that implements a regression decision tree, but now we need the tree to make predictions. The goal is to enable our gradient boosting model to generate predictions by leveraging the outputs of its individual decision trees. The following function, `predict_tree`, handles this task for a manually constructed regression decision tree represented as a nested dictionary. Given a set of input samples, it returns the predicted values for each instance.

For each sample, the function traverses the tree from the root to a leaf node, determining the prediction along the way. At each node, it checks whether it has reached a leaf; if not, it examines the feature and threshold that dictate the data split at that point. The function then compares the current sample's feature value to the threshold to decide which branch to follow

next. This "walking" down the tree continues until a leaf node is reached, where the prediction corresponds to the average target value stored there.

```
for i, x in enumerate(X):
      node = tree
      while not node['leaf']:
            if x[node['feature']] <= node['threshold']:
                node = node['left']
            else:
                node = node['right']
        preds[i] = node['value']
```

When all the samples have been processed, the function returns the `preds` array, which holds the tree's prediction for each input.

Finally, let's get to building the master function of this model: `build_gradient_boosting`. First, the model makes an initial prediction:

```
y_pred = np.full(y.shape, prediction(y))
```

It then enters a loop that runs for the number of specified boosting rounds (`n_estimators`). In each iteration, it calculates the residuals using the `compute_neg_gradient` function. Next, it fits a regression tree to these residuals using the provided tree-building function, and appends that tree to the list of trees. After fitting the tree, it uses the tree to predict updates for all samples and adjusts the current predictions by adding these updates, scaled by the learning rate.

```
residual = compute_neg_gradient(y, y_pred)
tree = build_decision_tree(X, residual, max_depth=max_depth,
min_samples_split=min_samples_split)
trees.append(tree)
```

```
update = predict_tree(tree, X)
y_pred += learning_rate * update
```

This process repeats, incrementally refining the predictions by adding corrections from each newly trained tree. Finally, the function returns the list of all fitted trees, the initial mean prediction, and the learning rate.

Predictions

Now that we've built our gradient boosting model, let's create a function to predict the Customer Lifetime Value (CLV) for new insurance customers!

The prediction function starts by initializing an array where every sample's prediction is set to the initial baseline—usually the mean target value from the training set. It then sequentially processes each tree in the trained ensemble. For each tree, it predicts the residual values for all samples, scales those predictions by the learning rate, and adds the result to the running total of predictions. After iterating through all trees, the function outputs the final combined predictions for every input sample, effectively aggregating the incremental corrections each tree contributes.

```
def predict_gradient_boosting(X, trees, init_val, learning_rate):
    y_pred = np.full(X.shape[0], init_val)
    for tree in trees:
        y_pred += learning_rate * predict_tree(tree, X)
    return y_pred
```

Once you run this code, the gradient boosting regressor will have made its predictions.

Evaluation

With predictions in hand, it's time to evaluate how well our Gradient Boosting Regressor performed against the actual CLV values. Since this is a regression problem, we rely on metrics designed for continuous targets: Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the R-squared score. Together, these metrics help us understand the average prediction error, the impact of large deviations, and how well the model explains variance in the data.

Let's take a look at the results of the Gradient Boosting Regressor:

1. Mean Absolute Error (MAE): 1506.13
2. Mean Squared Error (MSE): 17,267,994.99
3. Root Mean Squared Error (RMSE): 4155.48
4. R-squared Score: 0.6649

To put these numbers in perspective, consider the distribution of the target variable:

- Maximum CLV: 83,325.38
- Average CLV: 8,004.94
- Minimum CLV: 1,898.01

These results tell us several things:

1. The MAE of 1506.13 means that, on average, the model's predictions are off by about $1,500 from the true CLV. This is a fairly modest error given the average CLV is around $8,000.

2. The MSE of 17,267,994.99 represents the average squared difference between predicted and actual CLV values. While the number itself is large — since it's in squared dollars — it's not meant to be interpreted directly in dollar terms. Rather, it signals that larger errors are penalized more heavily, making this metric particularly sensitive to outliers. The high value suggests that, although most predictions are reasonably accurate, a few cases with large errors are inflating the overall error score.

3. The RMSE of 4155.48 is a bit higher due to squaring large errors, suggesting that while most predictions are fairly close, a few outliers may be far off. RMSE is especially sensitive to these large deviations.

4. The R-squared score of 0.6649 tells us that the model explains about 66.5% of the variance in customer lifetime value. While not perfect, this indicates a solid fit for a regression model on real-world business data, where noise and unobservable factors are common.

In summary, the model captures the main trends in the data with reasonable accuracy. Like all models, there's room for improvement, perhaps by engineering better features or tuning hyperparameters, but this baseline Gradient Boosting Regressor offers a robust foundation for CLV prediction.

Interpretation

From evaluating our regression metrics, it's clear that the Gradient Boosting Regressor provides a robust baseline for predicting Customer Lifetime Value. An MAE of $1,506 shows fairly precise predictions relative to the average customer value, while the RMSE of $4,155 highlights that some outliers still induce larger errors. The high MSE reiterates the presence of these few large deviations.

The R^2 value of 0.6649 means the model explains about two-thirds of the variation in CLV, which is a fabulous result given the complexity and uncertainty inherent in customer behavior data. However, about one-third of the variance remains unexplained, pointing to opportunities for model refinement.

Practically, this model supports broad business strategies such as customer segmentation or tiered marketing programs. However, for more sensitive applications, like personalized pricing or credit limits, further tuning, feature engineering, or adopting robust loss functions, like Huber loss, may be necessary to manage outliers and improve precision.

Feature importance analysis further reveals drivers of CLV: higher education (notably doctorate holders) aligns with increased customer value; policy type and specialized insurance products influence variations in value; retired employment status suggests distinct spending patterns; suburban residence signals geographic effects; and vehicle class correlates with differing value levels. Together, these insights can guide targeted marketing and risk strategies. Together, the following insights help inform more targeted, data-driven business strategies:

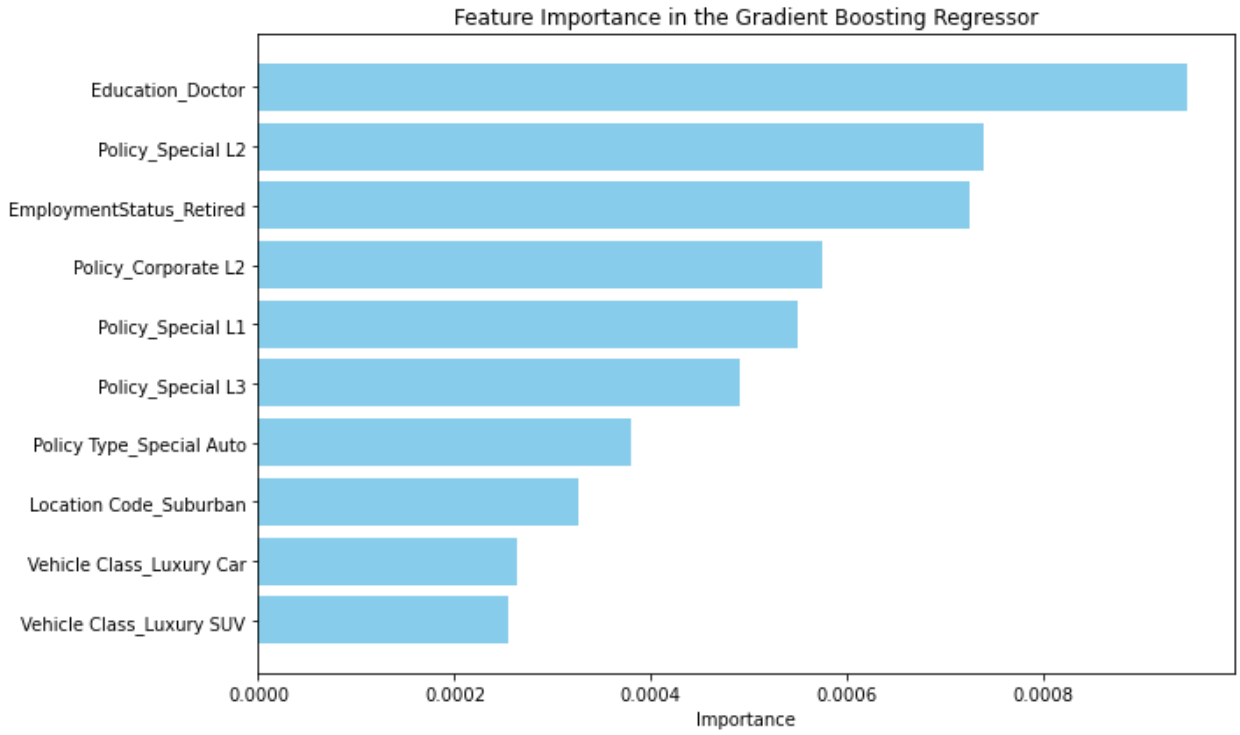

Figure 3

Conclusion

Gradient Boosting refines regression predictions by building a sequence of models that each learn from the mistakes of their predecessors. Unlike Random Forests, which average independently grown trees, Gradient Boosting trains new trees to correct residual errors sequentially, guided by gradient descent to minimize loss.

In this post, we moved from intuition to implementation, exploring how gradient boosting uses decision trees to fit residuals, leverages gradients for iterative updates, and incrementally

reduces prediction error. We demonstrated training a Gradient Boosting Regressor on real CLV data, evaluated its performance, and interpreted its outputs.

Our evaluation shows the model explains 66% of CLV variance and maintains a manageable prediction error, handling most cases well despite some outliers. Gradient Boosting proves to be a powerful, flexible tool for practical business forecasting.

Ultimately, gradient boosting is about learning from errors; transforming residuals into improved predictions, one tree at a time. By understanding its core mechanics, you gain the insight needed to fine-tune, extend, and deploy it effectively.

In this post, we've learned:

- What gradient boosting machines are
- The structure of a gradient boosting machine
- How to build a gradient boosting regressor using the scikit-learn library in Python
- How to build a gradient boosting regressor in Python
- How to predict values using scikit and manually in Python
- How to evaluate the model using MAE, MSE, RMSE, and R-squared score
- How to interpret model evaluations
- As a bonus: Methods to optimize tree number and visualize feature influence

That's a wrap on gradient boosting machines! Next, we'll dive into XGBoost, an optimized, scalable variation of gradient boosting that takes the technique to the next level. Stay tuned!