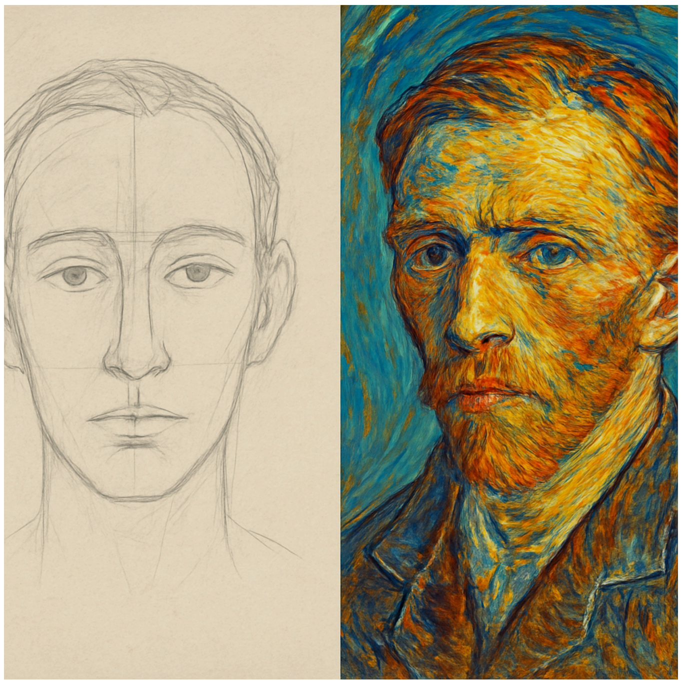
XGBoost

A deep dive into the optimization and regularization powering extreme gradient boosting

A Gradient Boosting model is like a novice artist sketching a portrait in layers. They begin with rough outlines, then gradually add shading, depth, and detail with each pass, learning and refining along the way. Now imagine that same portrait in the hands of a master, like Van Gogh, with precision tools, practiced technique, and an uncompromising eye for nuance. That's XGBoost: gradient boosting executed with expert-level control, speed, and mathematical finesse.



While traditional Gradient Boosting improves predictions incrementally by correcting previous errors, XGBoost takes the process further. It introduces methods like regularization to reduce overfitting, parallelizes tree construction for speed, and handles missing data intelligently. This post builds on our earlier discussion of gradient boosting, so if any terminology or process is unfamiliar, please refer to the previous post for foundational concepts. In this post, we'll apply XGBoost to the Gas Turbine Emissions dataset, where the goal is to predict nitric oxide (NO_x) levels from real-world operational data. This task requires a level of accuracy and efficiency that necessitates an XGBoost model.

What is XGBoost?

XGBoost, short for **Extreme Gradient Boosting**, is a scalable and efficient implementation of gradient boosting that has become a mainstay in real-world machine learning applications. According to the academic paper it was first cited in, XGBoost is an "optimized distributed gradient boosting library designed to be highly efficient, flexible and portable" [1]. In simpler terms, it's a versatile tool that builds decision trees smartly and resource-efficiently. XGBoost has helpful features built in, like handling of missing data, automatic regularization to prevent overfitting, and support for parallel processing and early stopping.

Tianqi Chen introduced the model as part of his PhD research at the University of Washington. Chen developed XGBoost to address some of the computational inefficiencies and limitations of traditional gradient boosting methods, resulting in his 2016 paper 'XGBoost: A Scalable Tree Boosting System'.

Beyond academia and competitions, XGBoost is widely used in the real world by growth teams and data scientists, especially for personalization, segmentation, and recommendation systems. Growth hackers often rely on XGBoost to power models that predict churn, personalize user journeys, or recommend products. Thanks to its high performance and ease of interpretability with large-scale or high-dimensional datasets, XGBoost tends to be the first serious model deployed in production before scaling to deeper architectures.

In this post, we'll apply the XGBoost Regressor to predict nitrogen oxide emissions from a gas turbine. For context, these emissions are key environmental indicators recorded hourly at a power plant and are influenced by factors such as temperature, pressure, and turbine performance.

[1] *Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System*, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Structure

Structurally, XGBoost builds upon the same core principles as Gradient Boosting Machines (GBMs): sequentially adding decision trees that minimize a loss function. However, it introduces several key innovations that make it faster, more accurate, and better suited for real-world applications.

Unlike traditional GBMs, which rely mainly on shrinkage (learning rate) to control overfitting, XGBoost incorporates additional **regularization** techniques directly into its **objective function.** Before diving into how XGBoost trains its models, let's unpack some of these concepts.

1. Regularization is a technique used to prevent overfitting when a model captures noise in the training data rather than meaningful patterns. XGBoost includes both **L1** and **L2** regularization
    a) L1 Regularization (Lasso): Adds a penalty proportional to the absolute value of the model's coefficients. It can drive some coefficients to zero, effectively performing feature selection.
    Also known as Lasso, which stands for Least Absolute Shrinkage and Selection Operator. Use case: When you believe only a few features are truly predictive—e.g., identifying the key drivers of customer churn.

    b) L2 Regularization (Ridge): Adds a penalty proportional to the square of the coefficients. It shrinks all weights toward zero but rarely eliminates them entirely.
    Use case: When most features are useful, L2 regularization keeps all features in the model while reducing the chance of overfitting by keeping coefficients small—e.g., in housing price prediction, where many variables contribute.

2. Objective function: At the heart of XGBoost is its objective function, which the model seeks to minimize. It combines:
    a) Loss function: Measures how far off predictions are from actual values.
    b) Regularization term: Penalizes overly complex trees to prevent overfitting.

This design enables XGBoost to construct trees that not only minimize errors but also remain as simple and generalizable as possible.

3. System Optimization: XGBoost is engineered for performance at scale, with several optimizations baked in:
    a) **Out-of-core computation**: Processes data in batches from disk rather than memory, allowing it to handle datasets too large to fit in RAM.
    b) **Cache awareness**: Organizes memory access patterns to efficiently use the CPU cache, speeding up training.
    c) **Parallel processing**: Splits the tree-building process across multiple CPU cores, significantly reducing training time.

4. Sparsity Handling: Missing values? No problem. XGBoost handles them natively.

5. Pruning and Approximate Tree Learning: Instead of stopping tree growth at a fixed depth, XGBoost uses a loss-guided pruning strategy.

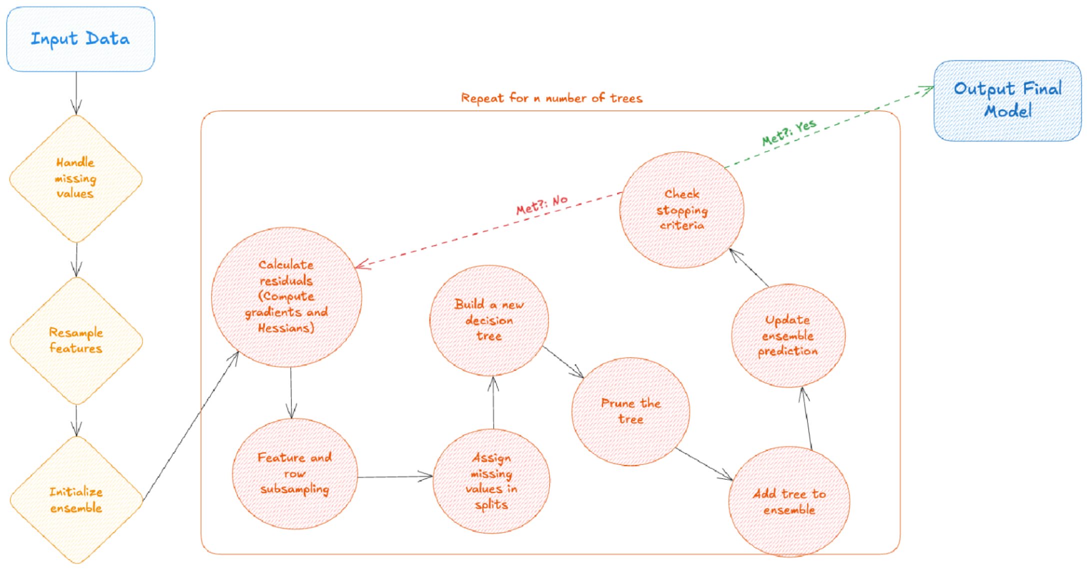
6. Other Innovations include:
    a) Feature Subsampling: Randomly samples features for each tree to introduce additional regularization and reduce overfitting.
    b) Second-Order Optimization: XGBoost uses both gradients and Hessians (second derivatives of the loss function) for more precise updates—a technique known as Newton boosting.

Putting it all together, the XGBoost training flow proceeds roughly as follows:

1. Input Data: Load and prepare the raw dataset.
2. Sparsity Handling: At each split, XGBoost determines the best way to route missing values.
3. Feature Subsampling: Select a random subset of features to use for this iteration.
4. Gradient & Hessian Calculation: Compute first- and second-order derivatives of the loss. (We'll discuss this further in the coming section)

5. Tree Construction: Build a new decision tree using an optimized, approximate algorithm that minimizes the regularized objective.
6. Pruning: Remove branches that don't contribute meaningfully to reducing the loss.
7. Add to Ensemble: The new tree is added to the model.
8. System Optimization: Throughout, XGBoost uses efficient memory and parallel processing.
9. Update Predictions: Predictions are updated using the new tree, scaled by the learning rate.
10. Repeat: Steps 3–9 are repeated until a stopping criterion is met.

Here's a simplified visual of the XGBoost structure:



Input Data
Handle missing values
Resample features
Initialize ensemble
Repeat for n number of trees
Calculate residuals (Compute gradients and Hessians)
Feature and row subsampling
Assign missing values in splits
Build a new decision tree
Prune the tree
Add tree to ensemble
Update ensemble prediction
Check stopping criteria
Met?: No
Met?: Yes
Output Final Model

Figure 1

Can We Build It? Yes, We Can!

With Python, initializing and training an XGBoost model is remarkably straightforward – it takes just two lines of code:

```
model = XGBClassifier(parameter_1,…,parameter_n)
```
or
```
model = XGBRegressor(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 an XBGoost model using default settings, the library xgboost offers several parameters to help control the tree's complexity and prevent overfitting. Here are the parameters I use in my model:

a) n_estimators: Sets the number of trees in the forest.
b) max_depth: Limits the maximum height of the tree.
c) learning_rate: Controls the speed of adjustments in the gradient descent.
d) subsample: The fraction of the training data randomly sampled for building each tree. Values <1.0 introduce randomness, which helps reduce overfitting and improves generalization. For instance, the argument 0.8 would mean 80% of the data is used per tree.
e) colsample_bytree: The fraction of features randomly selected to construct each tree to inject randomness and reduce correlation between trees.

These parameters afforded to us by Python's xgboost library make it pretty easy to control the XGBoost 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 XGBoost 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 NOx emissions of a gas turbine using a range of input features, including ambient conditions, compressor efficiency, and turbine performance. Since emissions are continuous values, we apply an XGBoost regression model to tackle this prediction task.

At its core, XGBoost builds an additive model by sequentially training decision trees to minimize a regularized objective function. Each new tree is added to correct the errors of the current ensemble, steadily improving predictions.

A distinguishing strength of XGBoost is its ability to handle missing values natively. Rather than requiring imputation, XGBoost evaluates sending missing values down both left and right branches at each split and learns the optimal default direction for them during training. This makes it robust to real-world, messy datasets without extra preprocessing.

Traditional gradient boosting relies solely on the gradient – the first derivative of the loss function – to guide updates. XGBoost enhances this by also computing the Hessian – the second derivative – at every step:

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

First derivative: $-2(y_i - \widehat{y_i})$

Ignoring constants that scale the gradient to get: $(y_i - \widehat{y_i})$

Hessian/ Second derivative: $-\frac{\partial(y_i - \widehat{y_i})}{\partial\widehat{y_i}} = 1$

The gradient shows the slope of the loss function, while the Hessian measures how rapidly that slope itself changes. Leveraging both allows XGBoost to use a **Taylor approximation of the loss**, making more informed and stable decisions during training. This is especially important when optimizing the tree splits and assigning values to leaf nodes.

- -

Deep Dive - Taylor Series

The Taylor series is about taking polynomials and non-polynomial functions and finding polynomials that approximate them near a specific input value. Since it is easier to take derivatives of polynomial functions, this process will come in handy for calculating the loss between the current and previous predictions using the gradient and Hessian.

While a Taylor series is an infinite sum, in practice, XGBoost uses a second-order (quadratic) Taylor approximation to estimate how much the loss will change as each new tree adjusts predictions. For the MSE loss:

$$L(y, \hat{y} + f) \approx L(y, \hat{y}) + g \cdot f + \frac{h}{2} \cdot f^2$$

Where g is the gradient at $\widehat{y}$, h is the Hessian at $\widehat{y}$, and f is the output of the new tree (correction). For mean squared error, substituting values:

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

$$h = 2$$

So: $$L(y, \widehat{y} + f) \approx (y_i - \widehat{y_i})^2 - 2(y_i - \widehat{y_i})f + f^2$$

This quadratic is what XGBoost actually minimizes at each boosting step, allowing efficient, robust optimization.

- -

At every boosting iteration, XGBoost first applies feature subsampling, randomly selecting a subset of features. This regularization step reduces overfitting, adds diversity to the ensemble, and speeds up computation. The model then computes the gradient and Hessian for each training instance using the current predictions, providing both a measure of how wrong the model is (the gradient) and an indication of its confidence in that error (the Hessian).

Next, the model constructs a new decision tree using a fast, approximate algorithm. Rather than exhaustively evaluating all possible split points, XGBoost bins feature values into discrete histograms to quickly estimate the best splits. The choice of each split is based on how much it improves the objective function, which, if you recall, consists of the loss function and a regularization penalty for model complexity. This penalty encompasses both L1 and L2 terms. While L1 promotes sparsity and can be utilized for feature selection, the L2 term (applied to leaf weights) prevents the model from over-relying on any single split or leaf, thereby enhancing its generalization capabilities.

- -

Deep Dive - Regularization

Having introduced L1 and L2 regularization as methods to control complexity and prevent overfitting, let's examine how XGBoost incorporates these penalties into its training.

XGBoost incorporates regularization directly into its objective function, which balances minimizing prediction error with controlling model complexity:

$$Objective = Loss + \gamma T + \alpha \sum_{j} \left| w_{j} \right| + \frac{1}{2} \lambda$$

$Loss = L_{MSE} = (y_{i} - \widehat{y_{i}})^{2}$

$\gamma T =$ A penalty for adding more leaves T to the tree

w_{j} = leaf weights

α = L1 regularization

λ = L2 regularization

This formulation arises from the goal of reducing training error while preventing overfitting by discouraging overly complex trees. XGBoost optimizes this using a second-order Taylor expansion of the loss around current predictions, employing gradient and Hessian to approximate the objective as a quadratic function. This makes training more efficient and stable.

Here’s how to apply regularization to leaf weights:

- $\alpha = \beta_{1} \sum_{j=1}^{T} \left| w_{j} \right|$
- $\lambda = \beta_{2} \sum_{j=1}^{T} \left| w_{j} \right|$

Where the betas are the regularization coefficients that you manually set.

By tuning alpha and lambda, you control the trade-off between flexibility and overfitting, tailoring XGBoost's performance to your data.

- -

Once a full tree is grown, XGBoost prunes the tree. Instead of stopping growth early with a maximum depth parameter, the algorithm allows trees to grow deep and then cuts off branches that fail to produce a meaningful gain in the objective function. This ensures that only valuable splits are retained, leading to simpler and more robust models. The completed tree's outputs are scaled by the learning rate and added to the current ensemble.

Throughout the process, XGBoost leverages system-level optimizations: parallel computation for faster split finding, out-of-core training for large datasets, and cache-optimized data structures. The full training loop—feature subsampling, gradient/Hessian computation, tree building, pruning, and ensemble updating—repeats until the stopping criterion is met.

This integration of mathematical rigor and engineering efficiency makes XGBoost one of the most accurate, scalable algorithms available for structured data.

Computational Mathematics

Let's break down XGBoost by hand, building its core components one function at a time. Unlike earlier posts where I mirrored scikit-learn's implementations, this exercise focuses on replicating the bare essentials of XGBoost. Keep in mind that some of its more advanced features require complex engineering, so they're beyond the scope of this manual example. With our dataset preprocessed and ready, let's dive into the fundamental computations that power XGBoost's fabled performance.

At the heart of XGBoost is boosting with both gradients and Hessians of our loss function. For regression with MSE, both are pleasantly simple; the gradient is just $(y_i - \widehat{y_i})$, and the Hessian is always 1. Here is a helper function that reflects these expressions:

```
def compute_grad_hess(y_true, y_pred):
      grad = y_pred - y_true
      hess = np.ones_like(y_true)
      return grad, hess
```

Unlike classic decision trees that directly reduce the sum of squared errors, XGBoost chooses splits by maximizing a gain metric. This gain incorporates both first- and second-order information (via a Taylor series approximation) and applies regularization to prevent overfitting. The formula for gain is:

$$gain = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right) - \gamma$$

This translates into code as:

```
def calc_split_gain(Gl, Hl, Gr, Hr, reg_lambda, gamma):
      left = Gl ** 2 / (Hl + reg_lambda)
      right = Gr ** 2 / (Hr + reg_lambda)
      parent = (Gl + Gr) ** 2 / (Hl + Hr + reg_lambda)
      gain = 0.5 * (left + right - parent) - gamma
      return gain
```

The function to find the best split iterates over all possible thresholds on a feature. It skips any splits where the resulting child nodes are too small, controlled by the parameter min_child_weight. For each candidate split, it sums gradients and Hessians for the left and right nodes and selects the threshold with the highest gain:

```
def best_split(X_col, grad, hess, reg_lambda, gamma, min_child_weight):
      …
      for thr in thresholds:
            left_mask = X_sorted <= thr
            right_mask = ~left_mask
                  …
            if Hl < min_child_weight or Hr < min_child_weight:
                  continue
            gain = calc_split_gain(Gl, Hl, Gr, Hr, reg_lambda, gamma)
            if gain > best_gain:
                 best_gain, best_thresh = gain, thr
          return best_thresh, best_gain
```

To decide predictions for leaf nodes, XGBoost calculates an optimal constant value that minimizes the regularized loss using:

$$w = -\frac{H+\lambda}{G}$$

Here, G and H are the sums of gradients and Hessians within that leaf, balancing improved fit with controlled model complexity.

The recursive tree-building function either creates a leaf node if a stopping criterion is met or finds the best split threshold to partition the data, then repeats the process recursively for the child nodes:

```
def fit_tree_xgb(X, grad, hess, depth, reg_lambda, gamma,
min_child_weight):
      if depth == 0 or np.sum(hess) < 2 * min_child_weight:
            G, H = grad.sum(), hess.sum()
            value = -G / (H + reg_lambda)
            return {'leaf': True, 'value': value}
      n_samples, n_features = X.shape
      best_feat, best_thr, best_gain = None, None, -np.inf
      for f in range(n_features):
            thr, gain = best_split(X[:, f], grad, hess, reg_lambda, gamma,
      min_child_weight)
            if thr is not None and gain > best_gain:
                  best_feat, best_thr, best_gain = f, thr, gain
      if best_feat is None:
            G, H = grad.sum(), hess.sum()
            value = -G / (H + reg_lambda)
            return {'leaf': True, 'value': value}
      left_mask = X[:, best_feat] <= best_thr
      right_mask = ~left_mask
      left_tree = fit_tree_xgb(X[left_mask], grad[left_mask],
hess[left_mask], depth - 1,reg_lambda, gamma, min_child_weight)
       right_tree = fit_tree_xgb(X[right_mask], grad[right_mask],
hess[right_mask], depth - 1, reg_lambda, gamma, min_child_weight)
          return {'leaf': False, 'feature': best_feat,
      'threshold': best_thr,'left': left_tree, 'right': right_tree}
```

`` ``` ``

To generate predictions, samples are traversed through the tree by comparing their feature values against thresholds until reaching leaves, whose weights provide the prediction values:

```
def predict_tree_xgb(tree, X):
    preds = np.zeros(X.shape[0])
    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']
    return preds
```

Finally, the full XGBoost model builds an ensemble of these trees. It starts with the mean target value, then iteratively fits trees to predict the residual errors using calculated gradients and Hessians:

```
def build_xgboost(X, y, n_estimators=50, learning_rate=0.1, max_depth=3,
reg_lambda=1.0, gamma=0.0, min_child_weight=1):
    X, y = np.asarray(X), np.asarray(y)
    y_pred = np.full(len(y), y.mean())
    trees = []
    for _ in range(n_estimators):
        grad, hess = compute_grad_hess(y, y_pred)
         tree = fit_tree_xgb(X, grad, hess, max_depth, reg_lambda, gamma,
min_child_weight)
        update = predict_tree_xgb(tree, X)
        y_pred += learning_rate * update
        trees.append(tree)
```

```
    return {'init_value': y.mean(), 'trees': trees, 'learning_rate': learning_rate}
```

While this manual code faithfully captures the mathematical essence of XGBoost, it omits several crucial features present in mature libraries such as xgboost and scikit-learn. This code file offers a learning scaffold for understanding the “why” and “how” of gradient boosting at a mathematical level, but it’s not suited for large-scale production or many real-world datasets without significant further engineering.

Predictions

Now that we’ve built our XGBoost model, let’s create a function to predict outputs—such as nitric oxide (NO_x) levels—for new data samples from the Gas Turbine Emissions dataset!

The prediction function begins by initializing an array of predictions, where every sample’s initial prediction is set to the baseline value stored in the model. This baseline is usually the mean target value calculated from the training set. Then, the function goes through each tree in the trained ensemble one by one. For each tree, it calculates the predicted adjustments for all samples by traversing the tree down to the leaves. These predicted values are scaled by the model’s learning rate to control the impact of each tree, and added cumulatively to the current predictions. After processing all trees, the function returns the final combined predictions, which effectively sum up all the incremental corrections each tree contributes to the baseline, giving the model’s overall output on the new data.

```
def predict_xgboost(X, model):
    X = np.asarray(X)
    y_pred = np.full(X.shape[0], model['init_value'])
    for tree in model['trees']:
        y_pred += model['learning_rate'] * predict_tree_xgb(tree, X)
    return y_pred
```

Running this code completes the prediction stage for our manual XGBoost model, enabling it to make accurate predictions of nitric oxide (NO_x) levels based on the complex patterns learned during training.

Evaluation

With predictions in hand from our models, it's time to evaluate how well they performed on the task of predicting nitric oxide (NO_x) levels in the Gas Turbine Emissions dataset. Since this is a regression problem, we use metrics suited for continuous targets: Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the R-squared score. Together, these metrics give a comprehensive view of average error size, the impact of large deviations, and the amount of variance in the data explained by the models.

Here are the evaluation results for both our manual XGBoost implementation and the official XGBoost library model:

	MAE	**MSE**	**RMSE**	**R-squared**
Manual XGBoost model	4.19	34.27	5.85	0.7414
XGBoost Library model	2.43	12.89	3.59	0.9027

Here's an interpretation of these numbers:

- The MAE of 4.19 for our manual model means on average, predictions deviate from the actual NO_x levels by about 4.19 units. The library model's lower MAE of 2.43 indicates more precise average predictions.
- The MSE further emphasizes error magnitude by squaring deviations. Our manual model's higher MSE (34.27) shows it struggles more with large errors compared to the library's 12.89.

- The RMSE, the square root of MSE, suggests that typical prediction errors for the manual model are about 5.85 units, versus 3.59 units for the library implementation.

- Finally, the R-squared score reflects predictive power: the manual model explains roughly 74% of the variance in NO_x levels, while the official XGBoost explains about 90%, indicating a substantially better fit.

In summary, while both models perform reasonably well across the meaningful range of NO_x levels, the library implementation's superior scores highlight the advantage of its advanced optimization and regularization features, resulting in a noticeably tighter and more reliable fit.

Interpretation

Our evaluation reveals important insights into the performance of both the manual and library XGBoost models on predicting nitric oxide (NO_x) levels in the Gas Turbine Emissions dataset.

The manual XGBoost implementation achieves a Mean Absolute Error (MAE) of 4.19, indicating that on average its predictions are within about 4 units of the true NO_x values. Given the average NO_x level of approximately 65.29, this reflects reasonable predictive accuracy. The Root Mean Squared Error (RMSE) of 5.85 suggests most errors are modest, with some larger deviations captured by the Mean Squared Error (MSE) of 34.27. The R-squared score of 0.7414 shows the model explains roughly 74% of the variance in NO_x levels, demonstrating a solid but improvable fit considering the data's natural variability.

By comparison, the XGBoost library model performs noticeably better, with an MAE of 2.43, RMSE of 3.59, and an R-squared of 0.9027. These metrics mean the library implementation reduces errors by roughly half and explains over 90% of the variance, highlighting its superior fit and predictive power. This substantial difference arises because the library incorporates numerous advanced features absent in the manual model, including efficient internal binning and histogram algorithms for handling large or complex data, row and feature subsampling for better generalization, early stopping and cross-validation to prevent overfitting, multithreaded parallel processing for faster training, and comprehensive hyperparameter optimization and support for custom loss functions and callbacks. These optimizations not only boost accuracy but also make training more efficient and adaptable to diverse scenarios.

While our manual model provides a valuable educational foundation showcasing the core mathematical principles of XGBoost, its simpler design limits its performance in practical applications. The comparison illustrates how these advanced features play a crucial role in enhancing model robustness and speed.

Future improvements could involve incorporating some of these advanced techniques into the manual model, such as smart histogram binning or subsampling, to close the performance gap. Additionally, thorough hyperparameter tuning and feature engineering are key to maximizing predictive power, especially in real-world, noisy datasets.

Conclusion

XGBoost enhances the gradient boosting framework by incorporating both first- and second-order derivatives of the loss function, enabling more precise and efficient tree building. Unlike Random Forests, which aggregate independently grown trees, XGBoost sequentially fits trees to the residual errors, carefully optimizing splits with gain calculations that balance accuracy and regularization.

In this post, we progressed from foundational concepts to practical implementation, showing how XGBoost uses gradients and Hessians to guide tree construction, how leaf weights are calculated optimally, and how the ensemble iteratively improves predictions. We trained a manual XGBoost model on real-world data, evaluated its performance against the official XGBoost library, and explored the reasons behind their differences in accuracy and speed.

Our evaluation highlighted that while the manual model captures key principles and provides solid predictions, the full-featured XGBoost library model significantly outperforms it by leveraging advanced techniques such as efficient histogram-based split finding, subsampling, early stopping, parallel processing, and hyperparameter tuning. These enhancements not only improve accuracy but also make training faster and more robust in diverse scenarios.

Ultimately, XGBoost is about transforming gradient and Hessian information into powerful incremental corrections, refining predictions step-by-step. Understanding its underlying mechanics equips you to build, interpret, and customize models with confidence.

In this post, we have learned:

- What XGBoost models are
- The structure of an XGBoost model
- How to build an XGBoost regressor using the xgboost library in Python
- How to manually build a skeletal XGBoost regressor in Python
- How to predict values using the XGBoost library and manual models in Python
- How to evaluate a model using MAE, MSE, RMSE, and R-squared score
- How to interpret model evaluations

That's a wrap on XGBoost models! Stay tuned to learn what's inside the black box of the next machine learning algorithm.