Improving predictive performance of GBoost and XGBoost Algorithms on Time Series Data Forecasting by working on learning rate

Aasneh Prasad 210101001 * **Pranav Jain 210101078** * **Sahil Danayak 210101092** * **Sanjana Kolisetty 210101093** *

Abstract

Regression trees have been proved to be an apt choice for time series forecasting. In this project, we implemented two successful boosting algorithms i.e., Gradient Boosting and XGBoost. Boosting involves sequentially training multiple base models, where each subsequent model corrects the errors made by the previous ones. This approach allows it to adapt it's focus to the misclassified instances, leading to better performance. Thus, we trained models on the chosen dataset using both the boosting algorithms.

1. Introduction

The selection of a time series dataset prompted an investigation into the suitability of regression (or decision) trees for its analysis. Regression trees offer several advantages within this domain, including their capacity to discern and model non-linear relationships between variables inherent in time series data. Furthermore, their adaptability to accommodate the presence of seasonality and trends within time series datasets renders them particularly pertinent. Additionally, the robustness of regression trees to outliers further enhances their utility in such analytical pursuits.

Ensemble methods, such as Random Forests or Gradient Boosting with regression trees as base learners, augment forecasting accuracy by aggregating predictions from multiple trees. These methodologies harness the inherent diversity among the constituent trees to produce more robust and accurate predictions.

Both bagging and boosting represent prominent ensemble learning techniques within the machine learning domain. Bagging involves the independent training of multiple base models on different subsets of the training data, with Random Forests being a notable example. In contrast, boosting entails the sequential training of multiple base models, wherein each subsequent model seeks to rectify the errors made by its predecessors. This iterative learning process, characteristic of boosting, enables the algorithm to focus on correcting misclassifications, thus potentially leading to superior predictive performance. Prominent examples of boosting algorithms include AdaBoost, Gradient Boosting Machines (GBM), XGBoost, LightGBM, and CatBoost.

Amongst the plethora of tree boosting algorithms, XGBoost stands out for its versatility and efficacy across a multitude of applications. Renowned for its scalability, expedited runtime, and numerous algorithmic optimizations, XGBoost represents a formidable choice for time series analysis. Serving as an enhanced iteration of the Gradient Boosting Algorithm, a comparative analysis between XGBoost and GBM elucidates their respective performance characteristics and capabilities within this context.

In addition to leveraging boosting algorithms, the optimization of learning rates during successive boosting rounds was undertaken to minimize residuals. Utilizing optimization techniques such as Simulated Annealing and Adam facilitated the refinement of model parameters and further enhanced predictive accuracy. Furthermore, the implementation of K-Fold Cross Validation during boosting rounds served to augment the computational process, ensuring robustness and generalizability of results

2. Methods

2.1. Exploratory Data Analysis

Exploratory Data Analysis (EDA) plays a pivotal role in comprehending and extracting insights from any dataset. Time series data commonly demonstrate trends and seasonal fluctuations, necessitating their identification to grasp essential dataset characteristics. Anomalies or outliers within a dataset may stem from data collection errors, system glitches, or infrequent occurrences. Time series data can also reveal intricate inter-dependencies and correlations among different variables or within the same variable across distinct time periods.

Trigonometric encoding, also known as Fourier encoding, emerges as a valuable technique when handling periodic or seasonal patterns recurring at fixed intervals, such as daily, weekly, or yearly cycles. This method entails recognizing the cyclical features inherent in the time series data and substituting the original values of these features with their corresponding sine and cosine transformations. By doing so, the cyclical nature of the data is effectively encoded into two continuous variables, capturing both the amplitude and phase of the cyclical pattern. The dataset has a Signal-to-Noise Ratio (SNR) of 4.96 which suggests minimal noise

interference. We employed FFT denoising as a preprocessing step to further refine the dataset and analyzed its impact on model performance (Library implementation) by comparing the RMSE of predictions. FFT denoising involved transforming the energy consumption data from the time domain to the frequency domain. A threshold was set to identify and eliminate frequencies deemed to be noise while preserving the fundamental signal components. The inverse FFT was then used to reconstruct the denoised time-series data.

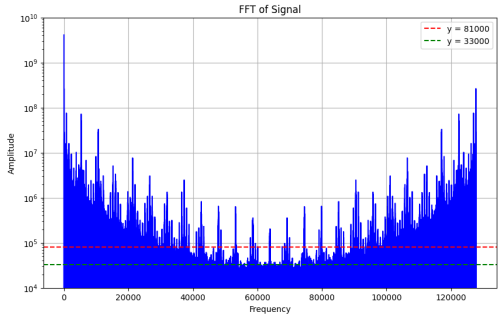


Figure 1. FFT Frequency Split

2.2. Gradient Boosting Algorithm

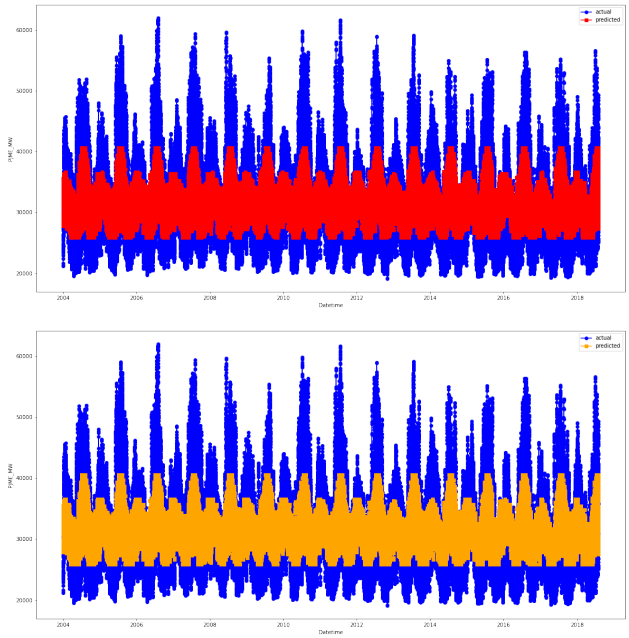


Figure 2. Actual vs Predicted values of Fixed Learning Rate & Library model of GBoost

Gradient Boosting is a powerful machine learning algorithm which sequentially combining weak learners, often decision trees, to create an ensemble model that predicts the target variable. The Gradient Boosting algorithm follows a stage-wise approach to build an ensemble of weak learners. At each stage, a new weak learner, typically a decision tree, is trained to predict the negative gradient of the loss function with respect to the current ensemble's predictions. The algorithm optimizes this objective function by iteratively adding weak learners to correct the errors made by the previous model.

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) \tag{1}$$

$\mathcal{L}(\phi)$ is the objective function where $\hat{y}_i$ is the predicted value, y_i is the actual target value, and l is a differentiable loss function. By iteratively updating predictions based on the negative gradient of the loss function, the algorithm converges towards the optimal solution.

$$\hat{y}^{(t)} = \hat{y}^{(t-1)} - \lambda \nabla_{\hat{y}^{(t-1)}} \mathcal{L}(\hat{y}^{(t-1)}) \tag{2}$$

where $\hat{y}^{(t)}$ is the updated predictions at iteration t, λ is the learning rate, and $\nabla_{\hat{y}^{(t-1)}} \mathcal{L}(\hat{y}^{(t-1)})$ is the negative gradient of the loss function with respect to the predictions at iteration $t-1$.

Friedman's Gradient Boosting Machine (GBM) enhances the standard Gradient Boosting algorithm. It trains sequential regression trees as weak learners to predict gradients rather than updating predictions directly. Let $\hat{y}^{(t-1)}$ represent the predictions from the ensemble at iteration $(t-1)$, and let $g_i^{(t-1)}$ represent the negative gradient of the loss function with respect to the i-th instance's prediction at iteration $(t-1)$. The sequential regression tree is trained to predict these gradients:

$$\hat{g}_i^{(t)} = f_t(\mathbf{x}_i) \tag{3}$$

where $\mathbf{x}_i$ represents the features of the i-th instance and $f_t(\cdot)$ represents the regression tree at iteration t. The step length, denoted by η, is determined through a line search, which iteratively adjusts the step length until an optimal value is found:

$$\eta = \arg\min_{\eta} \sum_i l(y_i, \hat{y}_i^{(t-1)} - \eta \hat{g}_i^{(t)}) \tag{4}$$

where l is the loss function, y_i is the true target value, and $\hat{y}_i^{(t-1)}$ is the prediction from the ensemble at iteration $(t-1)$. By rearranging the loss function in terms of a tree's structure and considering the gradients and number of samples in each leaf node, the algorithm evaluates the quality of different tree structures by:

$$\mathcal{L}_{\text{tree}} = \sum_{j=1}^{T} \left[\left(\frac{\left(\sum_{i \in I_L} g_i\right)^2}{n_L} + \frac{\left(\sum_{i \in I_R} g_i\right)^2}{n_R} \right) - \frac{\left(\sum_{i \in I} g_i\right)^2}{n} \right] \tag{5}$$

where I_L and I_R represent instance sets after splits, g_i are the gradients of the loss function, n_L and n_R are the size of I_L and I_R, respectively and n is the sum of n_L and n_R.

2.3. Extreme Gradient Boosting Algorithm

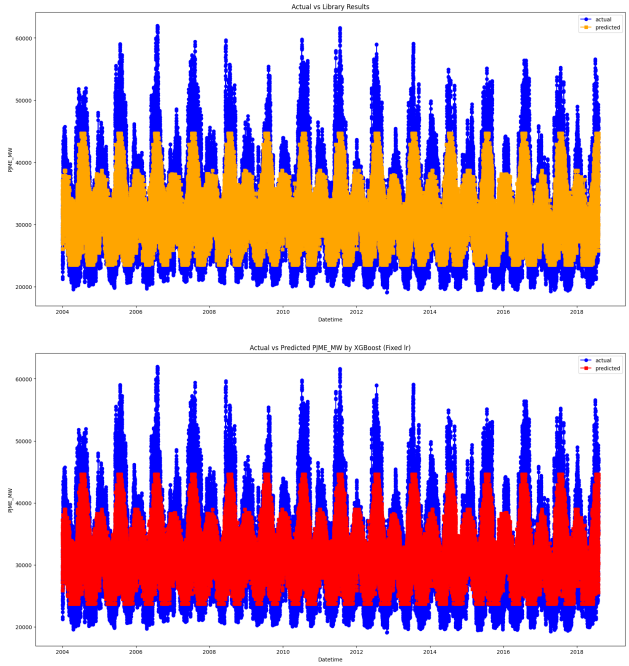


Figure 3. Actual vs Predicted values of Library & Fixed learning rate mode of XGBoost

XGBoost is based on the gradient boosting framework, which sequentially combines weak learners (typically decision trees) to create a strong predictive model. In gradient boosting, each subsequent model is trained to correct the errors made by the previous models, thereby minimizing the overall prediction error. To learn the set of functions used in the model, we minimize the following $regularized$ objective.

$$\mathcal{L}(\phi) = \sum_i l\left(\hat{y}_i, y_i\right) + \sum_k \Omega\left(f_k\right) \tag{6}$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Here $\mathcal{L}$ is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target y_i. The second term $\Omega(f)$ penalizes the complexity of the model ($regularization$ and $pruning$). Formally, let $\hat{y}_i{}^{(t)}$ be the prediction of the i-th instance at the t-th iteration, we will need to add f_t to minimize the following objective.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i{}^{(t-1)} + f_t\left(\mathbf{x}_i\right)\right) + \Omega\left(f_t\right) \tag{7}$$

XGBoost is built makes use of $Newton's$ Method. So instead of just computing the gradient and following it, it uses the second order derivative ($Hessian$) to gather more information to make a better approximation about the direction of the maximum decrease (in the loss function).

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n}\left[l\left(y_i, \hat{y}^{(t-1)}\right) + g_i f_t\left(\mathbf{x}_i\right) + \frac{1}{2} h_i f_t^2\left(\mathbf{x}_i\right)\right] + \Omega\left(f_t\right) \tag{8}$$

where

$$g_i = \partial_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}^{(t-1)}\right) \text{ and } h_i = \partial^2_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}^{(t-1)}\right)$$

We can remove the constant terms to obtain the following simplified objective at step t.

$$\mathcal{L}^t = \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{9}$$

Define $I_j = \{i \mid q\left(\mathbf{x}_i\right) = j\}$ as the instance set of leaf j. We can rewrite Eq(3) by expanding Ω as follows

$$\mathcal{L}^{(t)} = \sum_{j=1}^{T}\left[\left(\sum_{i \in I_j} g_i\right) w_j + \frac{1}{2}\left(\sum_{i \in I_j} h_i + \lambda\right) w_j^2\right] + \gamma T \tag{10}$$

We can compute the optimal weight w_j^* of leaf j by

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \tag{11}$$

and calculate the corresponding optimal value by

$$\mathcal{L}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i \in I_j} g_i\right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T. \tag{12}$$

A greedy algorithm that starts from a single leaf and iteratively adds branches to the tree is used instead. Assume that I_L and I_R are the instance sets of left and right nodes after the split. Letting $I = I_L \cup I_R$, then the loss reduction after the split is given by

$$\mathcal{L}_{\text{split}} = \sum_{j=1}^{T}\left[\frac{1}{2}\left(\frac{\left(\sum_{i \in I_L} g_i\right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left(\sum_{i \in I_R} g_i\right)^2}{\sum_{i \in I_R} h_i + \lambda}\right) - \frac{\left(\sum_{i \in I} g_i\right)^2}{\sum_{i \in I} h_i + \lambda}\right] - \gamma T \tag{13}$$

Algorithm 1 Exact Greedy Algorithm for Split Finding

Input: I, Instance set of correct node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0, H_L \leftarrow 0$
  for j *in sorted* $(I,$ *by* $x_{jk})$ **do**
    $G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$
    $score \leftarrow \max\left(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right)$
  end
end
Output: Split with max score

2.4. Simulated Annealing

Simulated annealing involves adaption of the learning rate in a way that mimics the annealing process in metallurgy. Initially, the learning rate is set high, allowing for rapid exploration of the solution space. As training progresses, the learning rate is gradually decreased, akin to cooling metal slowly to minimize energy and reach a stable state. This gradual reduction in learning rate helps the model to converge more effectively towards an optimal solution while avoiding getting trapped in local minima. It allows the model to navigate the solution space more efficiently and improve convergence towards the global optimum.

Algorithm 2 Simulated Annealing Optimizer

$T \leftarrow T_{\max}$
$x \leftarrow$ generate the initial candidate solution
$E \leftarrow E(x)$ compute the energy of the initial solution
while $(T > T_{\min})$ *and* $(E > E_{th})$ **do**
  $x_{\text{new}} \leftarrow$ generate a new candidate solution
  $E_{\text{new}} \leftarrow$ compute the energy of the new candidate x_{new}
  $\Delta E \leftarrow E_{\text{new}} - E$
  r $\leftarrow$ generate a random value in the range $[0, 1)$
  if $\Delta E < 0$ *or* $r < \exp(-\Delta E/T)$ **then**
    $x \leftarrow x_{\text{new}}$
    $E \leftarrow E_{\text{new}}$
  end
  $T \leftarrow \frac{T}{\alpha}$ cool the temperature
end
return x

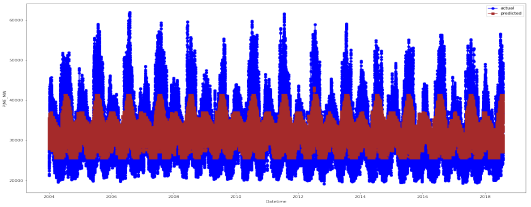


Figure 4. Actual vs Predicted values Simulated Annealing model of Gradient Boost

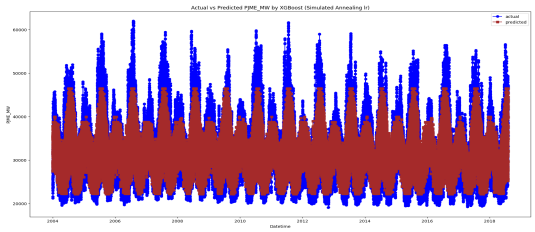


Figure 5. Actual vs Predicted values Simulated Annealing model of Extreme Gradient Boost

2.5. Adam Optimizer

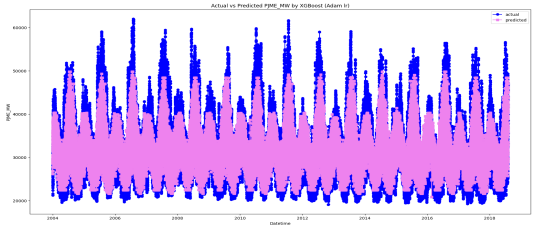


Figure 6. Actual vs Predicted values of Adam Learning Rate model of Extreme Gradient Boost

The Adam optimizer is an adaptive learning rate optimization algorithm used for training deep learning models. It combines ideas from both Momentum and RMSprop algorithms. Adam initializes two moving average variables for each parameter: m and v, which stores the first and second moment of the gradients. At each iteration during training, Adam computes the gradients of the loss function. Adam updates the moving averages of the gradients m and v with exponential decay. To prevent the moving averages from being biased towards zero during the early iterations, Adam performs bias correction by dividing m and v by $1 - \beta_1^t$ and $1 - \beta_2^t$ respectively wehre t is the iteration step and β_1, β_2 are the decay rates. Finally, Adam updates the parameters of the model.

Algorithm 3 Adam Optimizer

Input: α (Step Size), β_1, β_2, $f(\theta)$, θ_0
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
while θ_t *not converged* **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $m_t \leftarrow \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$
  $v_t \leftarrow \beta_2 * v_{t-1} + (1 - \beta_1) * {g_t}^2$
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \alpha * \hat{m}_t / \sqrt{\hat{v}_t} + \epsilon$
end
return θ_t

Table 1. Comparison of RMSE scores for entire dataset

Learning Rate	GBoost	XGBoost
Fixed	4335.07	3864.85
Simulated Annealing	4237.27	3747.61
Adam	4126.99	3745.79
Library	4349.03	3861.53

2.6. Approximate Algorithm & Weighted Quantile Sketch

The exact greedy algorithm efficiently explores all potential splitting points but struggles with memory constraints in large datasets. To address this, an approximate algorithm is proposed, initially suggesting candidate splitting points based on feature percentiles. These features are then discretized into buckets defined by these points, and optimal splits are identified based on aggregated statistics.

Two algorithm variants exist: global and local. The global variant generates all candidate splits during tree construction, while the local variant re-proposes after each split. The global method requires fewer steps but more candidate points, lacking refinement after splits. In contrast, the local proposal refines candidates post-split, potentially suiting deeper trees better.

Percentiles of features are used for even distribution of candidates. Formally, let multi-set $D_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \ldots, (x_{nk}, h_n)\}$ represent the k-th feature values and second-order gradient statistics of each training instance. We can define a rank function $r_k : \mathbb{R} \to [0, +\infty)$ as

$$r_k(z) = \frac{1}{\sum_{(x,h)\in D_k} h} \sum_{(x,h)\in D_k, x<z} h, \quad (8) \tag{14}$$

Here ε is an approximation factor. Intuitively, this means that there are roughly $1/\varepsilon$ candidate points.

$$|r_k(s_{kj}) - r_k(s_{kj+1})| < \varepsilon, \quad s_{k1} = \min_i x_{ki}, \quad s_{kt} = \max_i x_{ki}. \tag{15}$$

Each data point is weighted by h_i, reflecting its importance in the loss function. Here each data point is weighted by h_i. To see why h_i represents the weight, we can rewrite Eq (3) as

$$\sum_{i=1}^{n} \frac{1}{2} h_i (f_t(x_i) - g_i/h_i)^2 + \Omega(f_t) + \text{constant}, \tag{16}$$

which depicts the fact that each prediction has been weighted by h_i. We try to split the dataset into bins with equal weights.

Algorithm 4 Approximate Algorithm for Split Finding

for $k = 1$ *to* m **do**
  Propose $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ by percentiles on feature k. Proposal can be done per tree (global), or per split (local).
end
for $k = 1$ *to* m **do**
  $G_{kv} \leftarrow \sum_{j\in\{j|s_{kv}\geq x_{kj}>s_{kv}\}} g_j$
  $H_{kv} \leftarrow \sum_{j\in\{j|s_{kv}\geq x_{kj}>s_{kv}\}} h_j$
end
Follow same step as in previous section to find max score only among the bins created by the proposed splits.

3. Experiments Conducted & Results

3.1. Data Analysis

The Hourly Energy Consumption Dataset, sourced from Kaggle, offers a comprehensive view of energy consumption patterns over a period of 10 years. The dataset typically includes several variables, with the primary ones being:
1. **Timestamp**: Date and time information indicating when the energy consumption measurement was recorded.
2. **Energy Consumption**: The amount of energy consumed during the specified hour, measured in megawatts (MW).

Outlier analysis is a method used to identify data points that significantly deviate from the norm. These outliers may result from errors in data collection or represent rare events. In our dataset, outliers are likely to be removed as they are unlikely to represent genuine phenomena and could distort analysis results.

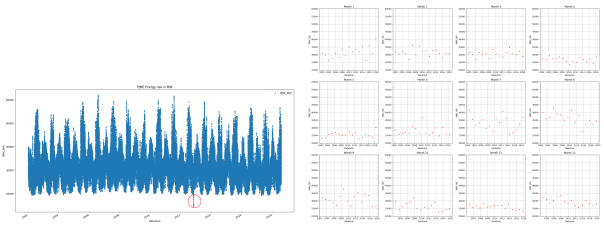
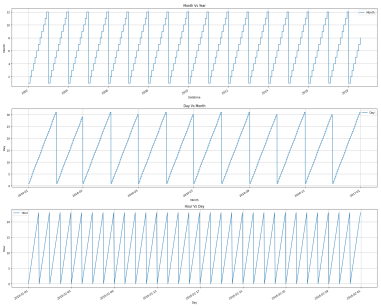


Figure 7. Outlier Analysis & Lag Feature Extraction

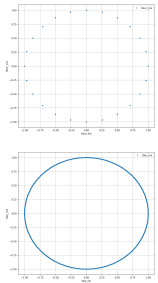
A lag k for a time series variable X_t refers to the value of X at time $t - k$ and is denoted by X_{t-k}. By incorporating lag features as predictors, the model can harness the temporal patterns and dependencies inherent in the data, enhancing its predictive capabilities. In our dataset, values observed on a given day closely resemble those recorded on the same day in previous or forthcoming years. This temporal consistency underscores the significance of leveraging lag features to capture and utilize historical patterns for accurate predictions.

The concept underlying trigonometric encoding involves the

transformation of cyclical features, such as the hour of the day or day of the week, onto periodic functions, commonly sine and cosine functions. In the context of energy consumption analysis, where patterns exhibit cyclic behavior on a yearly basis, employing trigonometric encoding proves advantageous. Specifically, energy consumption values on the same day of the year demonstrate similarity, indicative of cyclical trends. Additionally, energy consumption values observed on days with minimal temporal separation, as well as those recorded during proximate hours of the day, exhibit resemblances. Trigonometric encoding facilitates the capture of such temporal relationships, thereby enhancing the model's ability to effectively interpret and utilize cyclical patterns for accurate predictions.



((a)) Cyclicity of hours, days, months



((b)) Trigonometric encoding

Figure 8. Handling Cyclic Features

Post-denoising, a marginal improvement in model accuracy was observed. RMSE was reduced by only 0.43% for GBoost and 0.36% for XGBoost when trained on the FFT-denoised dataset compared to the original dataset using Library Implementation. While FFT denoising can slightly enhance model performance, the already low noise level (SNR 5) limits the extent of potential improvements.

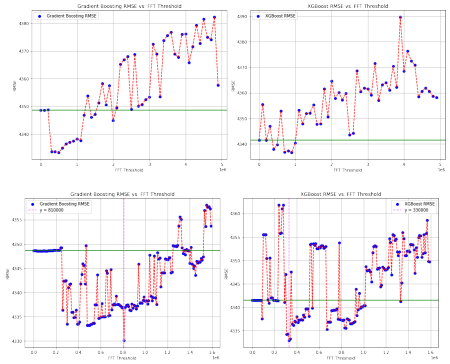


Figure 9. RMSE Plot for various thresholds in FFT in GBoost & XGBoost

As threshold increases parts of the signal that contain valuable information for predicting the target variable are also filtered. This loss of information can lead to underfitting, hence the RMSE increases. Different thresholds can impact various aspects of the signal in non-linear and complex ways — some thresholds might accidentally retain useful frequencies along with noise, and others might remove too much useful information. Hence, we observe a variance in RMSE.

3.2. Greedy Split

The *exact greedy algorithm* is a split finding algorithm that enumerates over all the possible splits on all the features. Most existing single machine tree boosting implementations, such as xgboost, R's gbm, as well as the single machine version of XGBoost support the exact greedy algorithm. It is computationally demanding to enumerate all the possible splits for continuous features.

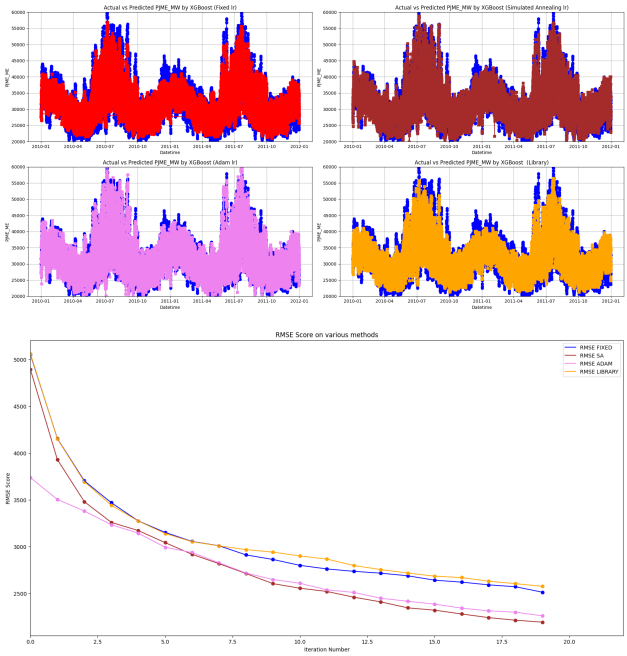


Figure 10. Actual vs Predicted values and RMSE Scores of Greedy Split with different Learning Rate on Extreme Gradient Boost

Table 2. Comparison of RMSE scores using greedy split

Learning Rate	GBoost	XGBoost
Fixed	1935.32	2511.87
Simulated Annealing	1935.32	2191.98
Adam	–	2259.10
Library	2970.53	2574.98

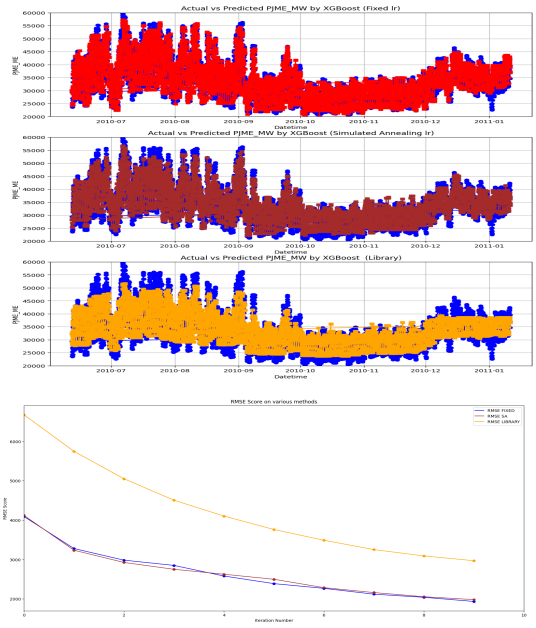


Figure 11. Actual vs Predicted values and RMSE values of Greedy Split with Different Learning Rate on Gradient Boost

3.3. Approximate Split

The *Approximate split* finding algorithm becomes imperative due to the heavy computational nature of *exact greedy algorithm*. We sort the data points based on the k-th feature and calculate the rank of each data point, using *hessians*. We create bins (roughly $1/\varepsilon$), based on these ranks, and calculate the best score in these bins, exclusively.

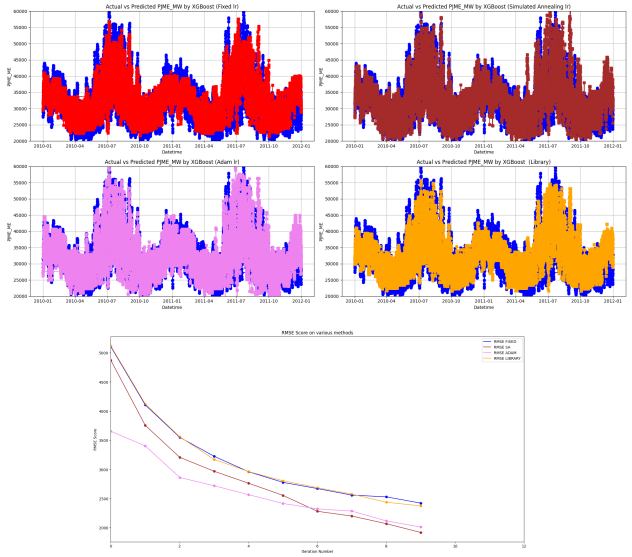


Figure 12. Actual vs Predicted values and RMSE Scores of Approximate Split with different Learning Rate on Extreme Gradient Boost

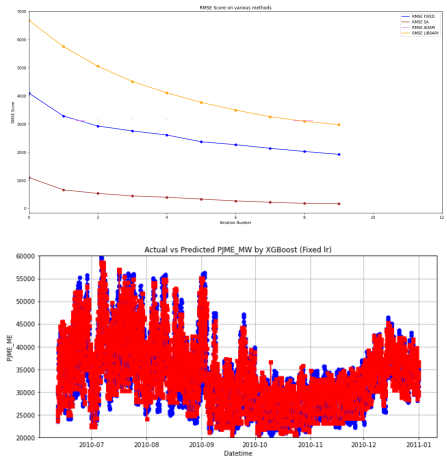


Figure 13. Actual vs Predicted values and RMSE of Approximate split of Fixed Learning Rate of Gradient Boost

Table 3. Comparison of RMSE scores using Approximate split

Learning Rate	GBoost	XGBoost
Fixed	1916.55	2423.132
Simulated Annealing	1620.35	1919.711
Adam	2051.07	2012.235
Library	2970.53	2375.621

3.4. Hyperparameter Tuning & K-Fold Cross Validation

Hyperparameter tuning is a crucial aspect of machine learning model development, where the best set of hyperparameters for the model to achieve optimal performance on a given dataset are searched. Hyperparameters are parameters that are set before the learning process begins, and they control aspects of the learning algorithm's behavior.

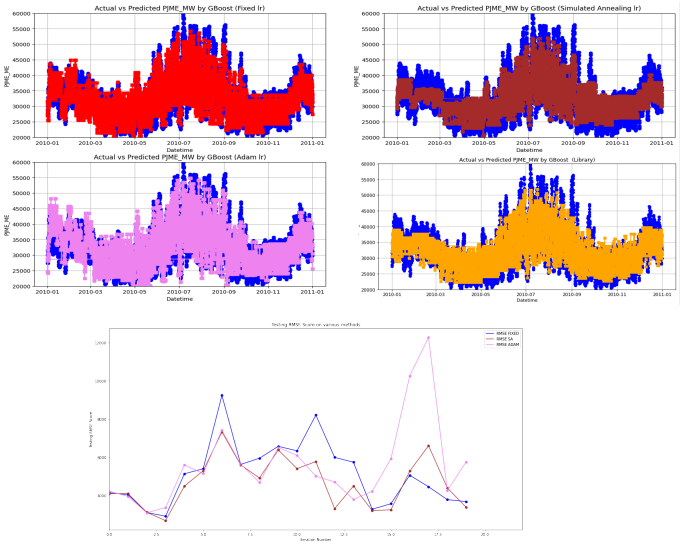


Figure 14. Actual vs Predicted values and RMSE of K-Fold Cross Validation of Gradient Boost

Grid search systematically explores a predefined grid of hyperparameter values, evaluating each combination by training and validating the model. It selects the combination that produces the best performance metric on the validation data. It can be computationally intensive due to its exhaustive search.

Table 4. Training RMSE scores using K-Fold Cross Validation

Learning Rate	GBoost	XGBoost
Fixed	3317.93	2149.41
Simulated Annealing	3159.61	1922.48
Adam	3278.81	2021.78
Library	–	2642.47

Table 5. Testing RMSE scores using K-Fold Cross Validation

Learning Rate	GBoost	XGBoost
Fixed	3674.27	3861.39
Simulated Annealing	3381.40	5662.67
Adam	5747.19	4462.85
Library	–	2841.97

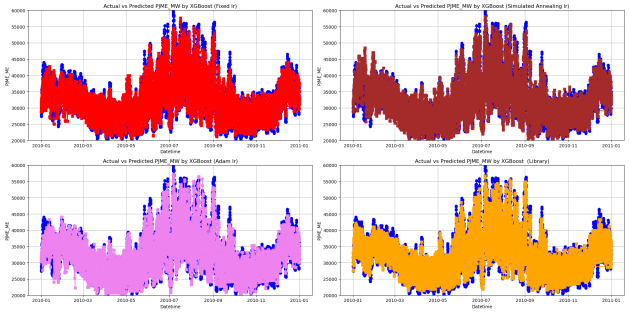


Figure 15. Actual vs Predicted values of K-Fold cross validation with different Learning Rate on Extreme Gradient Boost

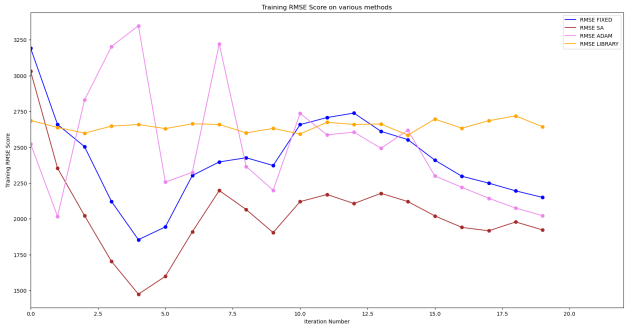


Figure 16. Training RMSE of K-Fold cross validation with different Learning Rate on Extreme Gradient Boost

Thus, we use Grid search on a reduced dataset with 13622 samples from 2010-06-12 to 2012-01-01. The parameters that we have tuned are `depth`, `min_leaf`, `learning_rate`, `boosting_rounds`, `min_child_weight`. Grid search returns the optimal parameter values and these parameters are used to train a model on a reduced dataset with 8754 samples from 2010-01 to 2011-01 using K-Fold cross validation during training.

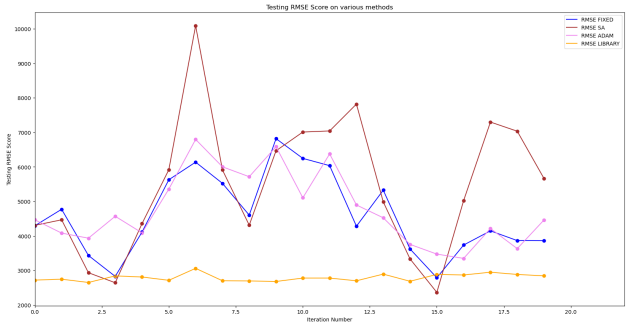


Figure 17. Testing RMSE of K-Fold cross validation with different Learning Rate on Extreme Gradient Boost

4. Comparative Analysis

4.1. Efficient Data Management

- scikit-learn efficiently handles sparse data formats through the use of csc_matrix and csr_matrix from the scipy.sparse package. This approach is beneficial when dealing with datasets that contain a lot of zeros, as it optimizes memory usage and computation time.
- The library implementation uses 32-bit float while out implementation uses 64-bit float.

4.2. Tree Construction Techniques

- Custom Implementation:
  - **Exhaustive Greedy Search**: Our method implements an exact greedy algorithm for finding the best split by scanning every possible split point. While thorough, this can be computationally expensive and time-consuming as it evaluates every possible split for every feature at each node.
  - **Recursive Node Creation**: This straightforward method is easy to understand and implement but can lead to inefficiencies, particularly with deep recursion, which might hit system recursion limits.
- scikit-learn approach: scikit-learn enhances the efficiency of decision tree construction by employing parallelism in finding the best splits within trees. scikit-learn GBoost implementation, isn't about parallelizing

the construction of sequential trees but optimizing the operations within the tree-building process, like evaluating node splits across different features.

4.3. Parallelism Techniques in scikit-learn

- **Lower-level Parallelism with OpenMP:** OpenMP is utilized to parallelize computationally intensive tasks written in Cython or C. This type of multi-threading aims to use as many threads as there are logical cores available, optimizing the computational efficiency of lower-level operations.
- **Parallel Routines in NumPy and SciPy:** scikit-learn relies on NumPy and SciPy, which themselves use multi-threaded routines from numerical libraries like MKL, OpenBLAS, or BLIS. These routines automatically parallelize linear algebra operations, significantly boosting performance.
- **Higher-level Parallelism with joblib:**
  - Task Parallelism: scikit-learn uses joblib for task-level parallelism, particularly effective in scenarios like cross-validation and grid search where independent jobs can be executed in parallel. This approach is managed through the n_jobs parameter, which dictates the number of threads or processes spawned.
  - Memory Management: The loky backend, generally used by joblib in scikit-learn, employs multi-processing while efficiently managing memory through mechanisms like memmap when dealing with large datasets. This prevents the duplication of memory across processes and is critical for handling substantial data volumes without excessive memory consumption.

The novel attempt of working on the learning rate of gradient boosted trees lead to better performance than the library implementation. The fixed learning rate method, closely represented the library implementation. Both simulated annealing and Adam adaptively adjust the learning rate during training based on the dynamics of the optimization process. Simulated annealing introduces stochasticity into the optimization process, which enables it to explore the solution space more effectively. This stochasticity helps prevent the optimizer from getting stuck in local minima, allowing it to find better solutions. Adam's adaptive learning rates and momentum mechanisms allow it to efficiently navigate complex optimization landscapes. By adjusting the learning rates based on the past gradients and their magnitudes, Adam can overcome challenges such as vanishing or exploding gradients, leading to faster convergence and better performance. This leads to better performance as compared to library's inbuilt fixed learning rate method.

5. Conclusions and Further Improvements

In conclusion, this paper presented the implementation of GBoost and XGboost from scratch, incorporating fixed and modified learning rates through simulated annealing and Adam Optimizer. Utilizing techniques such as FFT, K-Fold Cross Validation, and Hyper-parameter tuning, we aimed to enhance performance. Despite achieving superior results compared to library implementations, our methods exhibited slower computational speeds.

Future enhancements could focus on employing function wrappers, using multi-threading architecture to utilize max available cores. approximations, and optimizations in node split computations to mitigate training time inefficiencies.

6. References

- XGBoost: Introduction to XGBoost Algorithm in Machine Learning, Analytics Vidhya
- XGBoost: A Scalable Tree Boosting System
- XGBoost Documentation
- Hourly Energy Consumption Dataset, Kaggle
- Handling cyclical features, such as hours in a day, for machine learning pipelines with Python example, Selfidian Academy
- A Comparative Analysis of XGBoost: Candice BentéjacAnna Csörgő, Gonzalo Martínez-Muñoz
- A Proof of Local Convergence for the Adam Optimizer: Sebastian Bock, Martin Weiß
- Simulated Annealing Research Paper: Rafael E. Banchs
- Understanding gradient boosting as gradient descent.
- Deconstructing time series using Fourier Transform
- Denoising data with Fast Fourier Transform
- Parallelism in scikit-learn library