Chapter 6

Modeling

Introduction

In the pursuit of developing an effective forecasting model for our time series data, a comprehensive exploration of various algorithms was undertaken. Initially, we experimented with several machine learning techniques, including TimeGPT[8] and transformer-based forecasting models, to evaluate their performance in capturing the underlying patterns of the data. While these models provided valuable insights, we ultimately opted for XGBoost[2] as the primary algorithm for this project, The following sections will detail the rationale behind this selection, the implementation process, and the training and testing methodologies employed with XGBoost.

6.1 Model Selection: XGBoost for time series forecasting

Introduction

XGBoost [2] is a highly efficient and scalable implementation of gradient boosting frameworks, specifically optimized for structured or tabular data. Originally

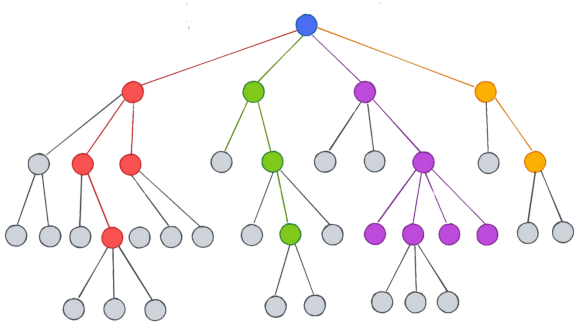

Figure 6.1: An illustration of a decision tree

developed in 2016 by Tianqi Chen[1] and Carlos Guestrin[2] XGBoost has garnered widespread acclaim for its remarkable performance in various machine learning applications and its robustness in tackling diverse predictive modeling challenges. This section delves into the mechanisms and key concepts underlying XGBoost, providing a comprehensive understanding of its operational principles.

6.1.1 Key Concepts and Mechanisms

XGBoost, short for Extreme Gradient Boosting, is an advanced implementation of gradient boosting algorithms, primarily used for classification and regression tasks. At its core, XGBoost utilizes decision trees as base learners. The method builds a strong predictive model by combining multiple weak learners (in this case, shallow decision trees) in an ensemble approach.

Regularized Learning Objective XGBoost employs a regularized learning objective, which is a combination of a primary loss function with a regularization term that penalizes model complexity, to optimize the model.

[1]Creator of XGBoost, Ph.D. from the University of Washington, recognized expert in machine learning systems.

[2]Professor at Stanford University, prominent AI and machine learning researcher, and co-founder of Turi, which was acquired by Apple.

For a given dataset with n examples and m features $D = \{(x_i, y_i)\}$ ($|D| = n$, $x_i \in \mathbb{R}^m$, $y_i \in \mathbb{R}$), a tree ensemble model (shown in Fig. 6.1) uses K additive functions to predict the output:

$$\hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in F, \tag{6.1}$$

where $F = \{f(x) = w_q(x)\}$ ($q : \mathbb{R}^m \to T, w \in \mathbb{R}^T$) is the space of regression trees. Here q represents the structure of each tree that maps an example to the corresponding leaf index. T is the number of leaves in the tree. Each f_k corresponds to an independent tree structure q and leaf weights w. Unlike decision trees, each regression tree contains a continuous score on each leaf;w_i is used to represent the score on the i-th leaf. For a given example,the decision rules in the trees (given by q) is used to classify it into the leaves and calculate the final prediction by summing up the score in the corresponding leaves (given by w).

To learn the set of functions used in the model,the following regularized objective will be minimized:

$$L(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{6.2}$$

where

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

Here 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 Ω penalizes the complexity of the model (i.e., the regression tree functions). The additional regularization term helps to smooth the final learned weights to avoid overfitting[3]. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions. When the regularization parameter is set to zero, the objective falls back to the traditional gradient tree boosting.

Gradient Tree Boosting The tree ensemble model in Eq 6.2 includes functions as parameters and cannot be optimized using traditional optimization methods in

[3]**Overfitting** is when a model learns the training data too well, including noise, leading to poor performance on unseen data.

Euclidean space. Instead, the model is trained in an additive manner.

Let $\hat{y}_i^{(t)}$ be the prediction of the i-th instance at the t-th iteration; adding f_t is needed to minimize the following objective:

$$L^{(t)} = \sum_n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) \tag{6.3}$$

This means f_t that most improves our model according to Eq 6.2 is greedily added . A second-order approximation can be used to quickly optimize the objective in the general setting [6].

$$L^{(t)} \approx \sum_n \left[l(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \tag{6.4}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ are the first and second-order gradient statistics on the loss function.

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

$$\tilde{L}^{(t)} = \sum_n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \tag{6.5}$$

Define $I_j = \{i | q(x_i) = j\}$ as the instance set of leaf j. The equation 6.2 can be rewritten by expanding Ω as follows:

$$\tilde{L}^{(t)} = \sum_n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \gamma T + \frac{1}{2} \lambda \sum_T w_j^2 \tag{6.6}$$

$$= \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \tag{6.7}$$

For a fixed structure $q(x)$, the optimal weight w_j^* of leaf j can be computed by:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{6.8}$$

and calculate the corresponding optimal value by

$$\tilde{L}^{(t)}(q) = -\frac{1}{2} \sum_T \left(\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \right)^2 + \gamma T. \tag{6.9}$$

Eq.6.5 can be used as a scoring function to measure the quality of a tree

structure q. This score is similar to the impurity score for evaluating decision trees, except that it is derived for a wider range of objective functions.

Normally, it is impossible to enumerate all the possible tree structures q. 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 $I = I_L \cup I_R$, the loss reduction after the split is given by:

$$L_{\text{split}} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{6.10}$$

This formula is usually used in practice for evaluating the split candidates.

Shrinkage and feature sub-sampling Besides the regularized objective, two additional techniques are used to further prevent overfitting. The first technique is shrinkage, introduced by Friedman[4] [7].

Shrinkage scales newly added weights by a factor η after each step of tree boosting. Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model.

The second technique is column or feature subsampling. This technique is used in Random Forest [1]. In the context of XGBoost, this technique involves randomly selecting a subset of features during the training of each decision tree. Infact during the construction of each tree in the boosting process, XGBoost randomly selects a specified fraction of the features to be considered for splitting at each node.

6.1.2 XGBoost for Time Series

XGBoost, originally designed for structured data, has been successfully adapted to time series forecasting through careful feature engineering and lag-based transformations. Time series data is inherently sequential, where values at earlier time steps influence future values. However, XGBoost does not directly handle this

[4]**Jerome Friedman** is a prominent statistician known for his work in statistical learning and data mining, particularly as a co-author of "The Elements of Statistical Learning." He developed key methods like the gradient boosting machine.

temporal dependency as time series models like ARIMA or LSTM do. Instead, XGBoost leverages the following techniques to tackle time series problems:

Lag Features To adapt XGBoost for time series forecasting, the first step is to transform the time series data into a supervised learning format. This is typically done by creating *lag features*. Lag features capture past values as predictors for the current or future values. For example, to predict the value at time step t, we can use the values at $t-1$, $t-2$, ..., $t-n$ as input features:
Features at time t : $[y(t-1), y(t-2), ..., y(t-n)]$

This transforms the univariate time series into a multivariate regression problem where each observation contains past values as features.

Windowing and Rolling Statistics Another common approach is to use windowing techniques, where you aggregate past values over a defined window to capture trends and patterns. For instance, rolling statistics such as the rolling mean, rolling standard deviation, or maximum/minimum values within a certain window (e.g., the last 7, 30, or 60 time steps) can be added as features:

$$\text{Rolling Mean}(y, \text{window size} = 7) = \frac{1}{7} \sum_{i=t-6}^{t} y(i)$$

These features help the model understand short-term trends and seasonal effects that are crucial in time series forecasting.

Time-Based Features In addition to lagged values, time series data often contains important cyclical patterns, such as daily, weekly, or seasonal trends. To capture these periodicities, additional features representing the time dimension are added, such as:

- **Hour of the day, Day of the week, Month of the year**: Useful for capturing diurnal or seasonal variations.
- **Holiday indicators**: Captures the impact of special events or holidays on the time series.

These time-based features allow XGBoost to account for recurrent patterns in the data, which are common in time series.

Handling Exogenous Variables (Multivariate Time Series) In multivariate time series forecasting, where exogenous variables (other time series or external factors) influence the target series, XGBoost can easily incorporate these additional inputs as features. For example, when forecasting international travel demand, additional variables like the occurrence of the COVID-19 pandemic can be included as features to improve prediction accuracy.

Advantages of XGBoost in Time Series

- **Efficiency**: XGBoost is optimized for performance, handling large datasets quickly with parallelized tree construction.
- **Interpretability**: XGBoost provides feature importance metrics, which allow for understanding the influence of specific lag features or time-based features.
- **Flexibility**: The model can easily incorporate additional covariates and deal with missing values or non-stationary data through feature engineering.
  Additionally, XGBoost can handle both **global** and **local** perspectives across a set of time series by leveraging its ability to include various features and model interactions between them.
  In fact , When looking at a set of time series globally, XGBoost treats all time series in the dataset as a part of a single, unified modeling framework. This means that the model learns patterns that are shared across all the time series, such as general trends, seasonality, and common impacts of exogenous variables.
  Locally, XGBoost can focus on individual time series' unique characteristics by including series-specific features, such as the historical data of each time series, and by making use of features that encode the specific behavior of each series
- **Regularization**: Built-in regularization helps reduce overfitting, which can be a significant issue in time series data with strong autocorrelation.

Conclusion

XGBoost has thus proven to be a powerful tool for time series forecasting, especially when combined with appropriate feature engineering techniques. While it doesn't natively model sequential dependencies like traditional time series models, its flexibility and performance make it a popular choice for forecasting problems. In the next section we will delve into the implementation of XGBoost in my project.

6.2 Hyperparameter Optimization

To optimize the performance of the forecasting model, effective hyperparameter tuning is essential. Hyperparameters, unlike parameters learned during the training process, need to be set prior to model training and have a significant impact on the model's performance. In this project, we utilized Optuna, an open-source hyperparameter optimization framework, to systematically search for the best hyperparameter configurations.

6.2.1 Mathematical background of Optuna's Tree-Structured Parzen Estimator

The Tree-Structured Parzen Estimator (TPE)[18] forms the core of Optuna's hyperparameter optimization framework. It models the objective function using probability distributions, allowing for an efficient search of the hyperparameter space by balancing exploration and exploitation.

The optimization process aims to minimize the objective function $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ represents the hyperparameter configuration space, and the observed performance is modeled as $y = f(x) + \epsilon$, with ϵ representing noise.

TPE splits the observed hyperparameter configurations D into two groups:

- $D(l)$: The better-performing group, consisting of the top-quantile γ.
- $D(g)$: The worse-performing group, comprising the remaining configurations.

The goal is to maximize the likelihood of selecting configurations from the better-performing group (by picking the configuration with the best acquisition function value (green star) in the samples shown in the figure 6.2). TPE achieves

this through *Kernel Density Estimation*, where two probability density functions are built for the better and worse groups:

$$p(x|D(l)) = w_0^{(l)} p_0(x) + \sum_{n=1}^{N(l)} w_n k(x, x_n|b^{(l)})$$

$$p(x|D(g)) = w_0^{(g)} p_0(x) + \sum_{n=N(l)+1}^{N} w_n k(x, x_n|b^{(g)})$$

where $k(x, x_n|b)$ is the kernel function (e.g., Gaussian functions), and $b^{(l)}, b^{(g)}$ represent the bandwidths[5] used for the KDEs. These bandwidths control how tightly the probability distribution is fitted around the observed configurations.

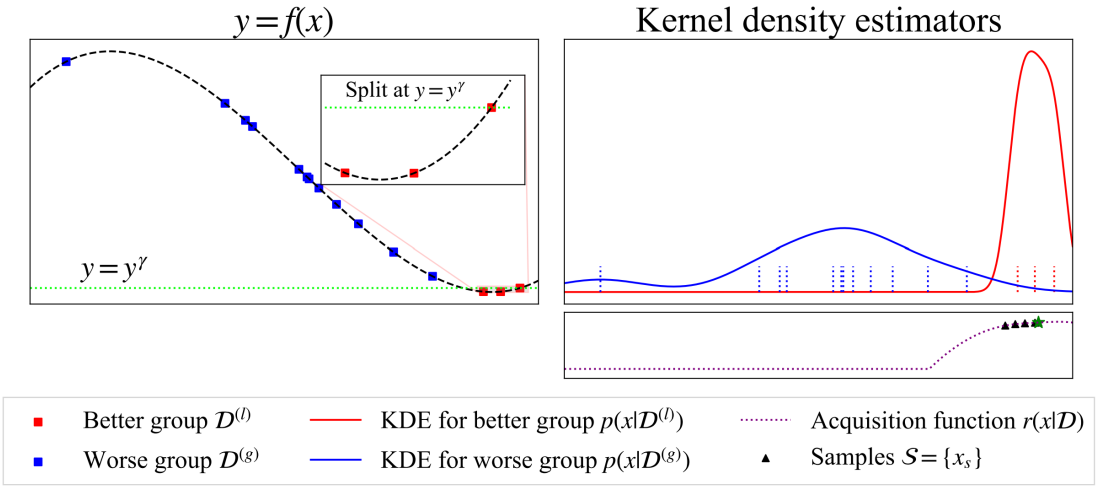

Figure 6.2: The conceptual visualization of TPE[18].

The *density ratio* r (called acquisition function in TPE) of the two PDFs is used to rank hyperparameter configurations, prioritizing those that are more likely to belong to the better-performing group.

$$r(x|D) = \frac{p(x|D(l))}{p(x|D(g))}$$

[5]The bandwidth is a value that controls the spread of the kernel function.

6.2.2 TPE Algorithm

The TPE algorithm optimizes the acquisition function iteratively, following these steps:

Algorithm 1 Tree-structured Parzen estimator (TPE)

N_{init} (The number of initial configurations, `n_startup_trials` in Optuna), N_s (The number of candidates to consider in the optimization of the acquisition function, `n_ei_candidates` in Optuna), Γ (A function to compute the top quantile γ, `gamma` in Optuna), W (A function to compute weights $\{w_n\}_{n=0}^{N+1}$, `weights` in Optuna), k (A kernel function), B (A function to compute a bandwidth b for k).

$\mathcal{D} \leftarrow \emptyset$
for $n = 1, 2, \ldots, N_{\text{init}}$ **do** ▷ Initialization
  Randomly pick $\boldsymbol{x}_n$
  $y_n \coloneqq f(\boldsymbol{x}_n) + \epsilon_n$ ▷ Evaluate the (expensive) objective function
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}_n, y_n)\}$
while Budget is left **do**
  Compute $\gamma \leftarrow \Gamma(N)$ with $N \coloneqq |\mathcal{D}|$ ▷ Section 3.1 (Splitting algorithm)
  Split $\mathcal{D}$ into $\mathcal{D}^{(l)}$ and $\mathcal{D}^{(g)}$
  Compute $\{w_n\}_{n=0}^{N+1} \leftarrow W(\mathcal{D})$ ▷ See Section 3.2 (Weighting algorithm)
  Compute $b^{(l)} \leftarrow B(\mathcal{D}^{(l)}), b^{(g)} \leftarrow B(\mathcal{D}^{(g)})$ ▷ Section 3.3.4 (Bandwidth selection)
  Build $p(\boldsymbol{x}|\mathcal{D}^{(l)}), p(\boldsymbol{x}|\mathcal{D}^{(g)})$ based on Eq. (5) ▷ Use $\{w_n\}_{n=0}^{N+1}$ and $b^{(l)}, b^{(g)}$
  Sample $\mathcal{S} \coloneqq \{\boldsymbol{x}_s\}_{s=1}^{N_s} \sim p(\boldsymbol{x}|\mathcal{D}^{(l)})$
  Pick $\boldsymbol{x}_{N+1} \coloneqq \boldsymbol{x}^\star \in \operatorname{argmax}_{\boldsymbol{x} \in \mathcal{S}} r(\boldsymbol{x}|\mathcal{D})$ ▷ The evaluations by the acquisition function
  $y_{N+1} \coloneqq f(\boldsymbol{x}_{N+1}) + \epsilon_{N+1}$ ▷ Evaluate the (expensive) objective function
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\boldsymbol{x}_{N+1}, y_{N+1}\}$

Figure 6.3: Tree-structured Parzen estimator algorithm[18]

Through this iterative refinement of the search space, TPE effectively balances exploration of new hyperparameter configurations and exploitation of known good configurations.

The use of Optuna for hyperparameter optimization had a significant effect on the model's results. By automating the tuning process, Optuna allowed for a more efficient exploration of the hyperparameter space. Key hyperparameters optimized included learning rate, maximum depth of trees, subsample ratio, and column sample by tree.

6.3 Model Implementation

6.3.1 Libraries and tools used

For the implementation of the model, three primary libraries were utilized: Nixtla MLForecast, XGBoost, and Optuna. Each of these libraries plays a crucial role in data preprocessing, model training, and hyperparameter optimization, respectively. Below is an overview of each library, including their history, and how they were applied in this project.

Nixtla MLForecast

Nixtla's MLForecast [12] is a relatively new yet rapidly growing open-source library dedicated to time series forecasting. Developed by Nixtla[9], a company focused on democratizing advanced forecasting tools, its core goal is to make it easier for users to apply machine learning models to time series data, reducing the complexity involved in model training and deployment.

Figure 6.4: Nixtla's logo

In this project, MLForecast was used to prepare the time series data by *Generating Lag Features* , *Rolling Aggregates* and *Handling Exogenous Variables.*

XGBoost

XGBoost[4] developed by the Distributed Machine Learning Community or DMLC[3] is an open-source software library designed for optimized gradient boosting algorithms. In this project, XGBoost was used as the core forecasting model.

Figure 6.5: DMLC XGBoost's logo

Optuna

Optuna[13] is an open-source hyperparameter optimization framework designed to automate the tuning process for machine learning models. It offers a straightforward and efficient approach to optimizing complex models, featuring several key functionalities that we will discuss later in the hyperparameter section, along with an explanation of its core concepts.

Figure 6.6: Optuna's logo

In this project, Optuna was used to optimize the hyperparameters of the XGBoost model.

6.3.2 Training and Hyperparameter Tuning Process

The training process for each time series involved iterating over 196 relationships, with hyperparameter tuning incorporated using Optuna. The workflow can be summarized in the following steps:

1. **Data Preparation:** For each time series, the relevant data was extracted, and the necessary features were generated using the Nixtla MLForecast library. This included creating lag features and incorporating exogenous variables.

2. **Model Initialization:** An instance of the XGBoost model was initialized for each time series.

3. **Hyperparameter Tuning with Optuna:** For each time series, an Optuna optimization study was conducted within the training loop. The number of optimization trials was customized based on the priority of the time series relationship:

    - **P0 priority:** 200 trials
    - **P1 priority:** 100 trials
    - **P2 priority:** 80 trials

The objective function for Optuna was defined using the Mean Absolute Percentage Error (MAPE), and the hyperparameter tuning was performed on the test set due to the lack of sufficient data for a dedicated validation set.

4. **Training Loop:** After completing the Optuna study and selecting the optimal hyperparameters, each model was trained on its corresponding time series data using the best configuration found during the tuning process.

5. **Evaluation:** Once the training was complete, each model's performance was evaluated using MAPE to assess its accuracy and overall performance.

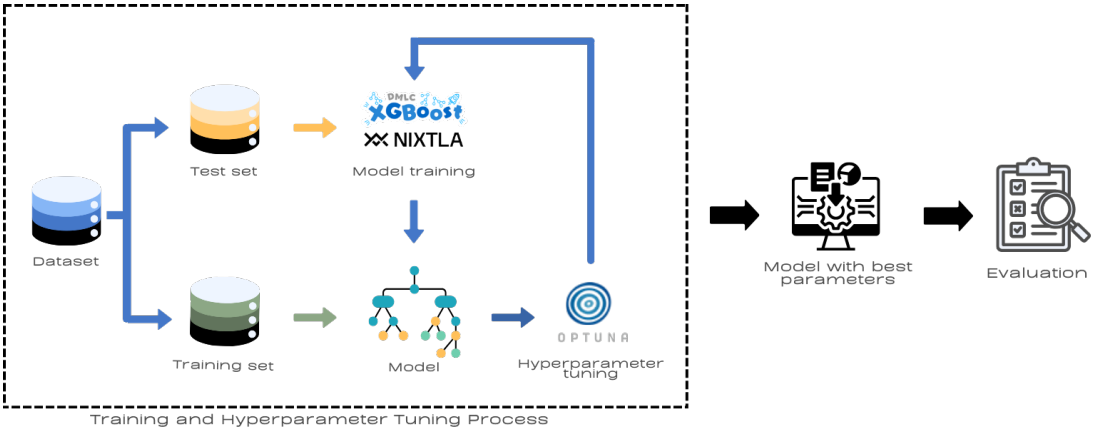

Figure 6.7: Training process flowchart

Conclusion

In conclusion, the modeling process involved a comprehensive evaluation of multiple machine learning algorithms for time series forecasting, ultimately identifying XGBoost as the optimal choice due to its flexibility, efficiency, and strong performance on sequential data. By combining advanced techniques and customized feature engineering from Nixtla's MLForecast with the optimized hyperparameters derived through Optuna, XGBoost proved to be a highly effective solution for the time series forecasting challenges in this project. In the following sections, we will delve into the performance details of the XGBoost model.