**Exploring Time Series Prediction of Energy Consumption Using XGBoost and Cross Validation**

[[Ehsan Nabatchian](https://medium.com/@ethannabatchian?source=post_page-----5d299655bec6--------------------------------)](https://medium.com/@ethannabatchian?source=post_page-----5d299655bec6--------------------------------)

[Ehsan Nabatchian](https://medium.com/@ethannabatchian?source=post_page-----5d299655bec6--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F857d6c04fd81&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40ethannabatchian%2Fexploring-time-series-prediction-of-energy-consumption-using-xgboost-and-cross-validation-5d299655bec6&user=Ehsan+Nabatchian&userId=857d6c04fd81&source=post_page-857d6c04fd81----5d299655bec6---------------------post_header-----------)

14 min read

·

Jan 11

168

1

In this project, we leverage a comprehensive dataset of hourly energy consumption spanning over a decade, obtained from Kaggle. Our objective is to employ the powerful XGBoost model and implement cross-validation techniques to make accurate time series predictions of future energy consumption.

**Data Analysis Setup**

To begin our analysis, we import essential Python libraries for data manipulation and visualization. This includes Pandas for data handling, NumPy for numerical operations, Seaborn for enhanced visualizations, and Matplotlib for customizable plotting. Additionally, we incorporate the XGBoost model for predictive analytics and the mean squared error metric for evaluating model performance.

import pandas as pd  
import numpy as np   
import seaborn as sns  
color\_pal = sns.color\_palette()  
import matplotlib.pyplot as plt  
plt.style.use('fivethirtyeight')  
  
import xgboost as xgb  
from sklearn.metrics import mean\_squared\_error

**Data Loading and Indexing**

The next step involves loading our dataset from a CSV file. We set the index to be the ‘Datetime’ column, a crucial step for consistency in time series predictions. The following code snippet illustrates these actions:

df = pd.read\_csv('PJME\_hourly.csv')  
  
### Set Index as Datetime  
df = df.set\_index('Datetime')  
  
Change Datetime Datatype to Datetime  
#df.index = pd.to\_datetime(df.index)

**Exploring Time Series Patterns**

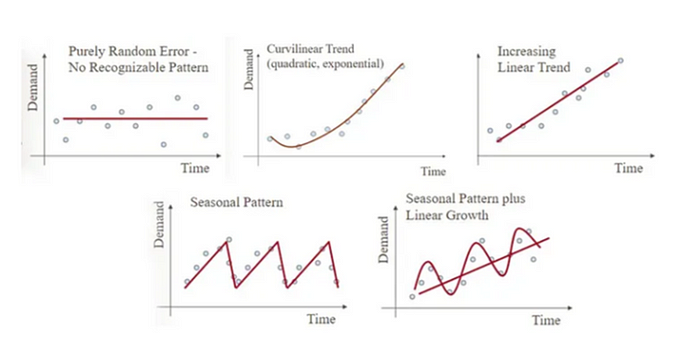
Before diving into the modelling phase, it is imperative to examine the various patterns inherent in time series data. Time series models often exhibit distinct patterns that can significantly impact predictive accuracy. By gaining insights into these patterns, we can better tailor our modelling approach for effective energy consumption predictions.

Stay tuned for further updates as we delve into creating features, defining lag variables, and implementing the XGBoost model with cross-validation to enhance the accuracy of our time series predictions.

**Understanding Time Series Components and Data Visualization**

**Understanding Time Series Components**

In many instances, time series data exhibits a combination of various patterns. It is crucial to comprehend the components of a time series thoroughly. In the upcoming sections, we will delve into a detailed exploration of the components, focusing on their intricacies and implications for accurate time series predictions.



**Seasonal Patterns in Time Series**

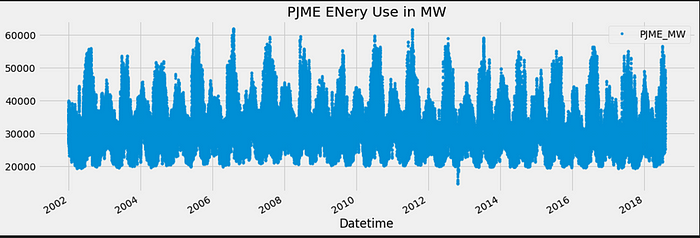
Our analysis later reveals that the patterns present in our model fall under the category of seasonal patterns. This implies that our data undergoes changes in trends depending on the time of the year, a factor we will carefully consider in our predictive modelling.

**Data Visualization**

Moving forward, we turn our attention to visualizing the data for in-depth analysis. A simple graph is plotted to observe trends over time and identify any outliers within the dataset. The code snippet below generates the graph:

df.plot(style = '.',   
 figsize = (15, 5),   
 color = color\_pal[0],   
 title = 'PJME ENery Use in MW');

This visualization serves as a foundational step in understanding the underlying patterns and characteristics of our time series data, paving the way for subsequent modelling and analysis.



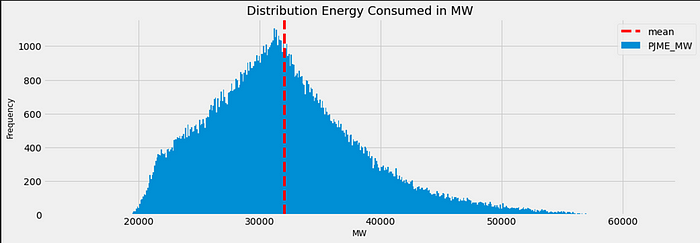
**Outlier Analysis and Data Distribution Exploration**

Upon examining the graph illustrating energy consumption over time, a notable area between 2012–2013 reveals extremely low values. These anomalies may signify outliers in our data, potentially resulting from unusual events such as meltdowns, blackouts, or sensor malfunctions. Addressing outliers is crucial, as our model may learn from these data points, potentially compromising the accuracy of predictions.

**Histogram Analysis**

Taking a step further, we aim to classify values under 20,000 MW as outliers through a histogram analysis. To create the histogram, we calculate the mean energy consumption (`mean\_MW`) and plot the distribution using the following code:

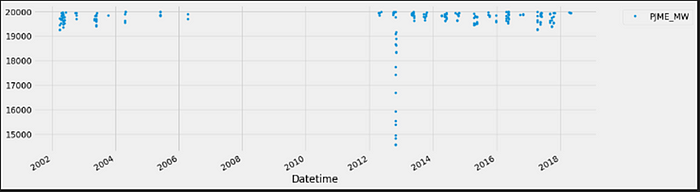
mean\_MW = df['PJME\_MW'].mean()  
mean\_MW  
  
df['PJME\_MW'].plot(figsize = (15,5), kind = 'hist', bins = 500)  
  
plt.title('Distribution Energy Consumed in MW', fontsize=18)  
  
plt.xlabel('MW', fontsize=12)  
  
plt.ylabel('Frequency', fontsize=12)  
  
plt.axvline(x=mean\_MW, color='r', linestyle = '--', label = 'mean')  
  
plt.legend(bbox\_to\_anchor=(1.04,1));



**Histogram Observation**

The histogram reveals that the majority of values fall within the range of 20,000 to 55,000 MW. Consequently, values below 20,000 MW are identified as outliers. To pinpoint when these low values occurred, a ‘query’ method is utilized:

df.query('PJME\_MW < 20000').plot(figsize = (15,5), style = '.')  
plt.legend(bbox\_to\_anchor=(1.04,1));

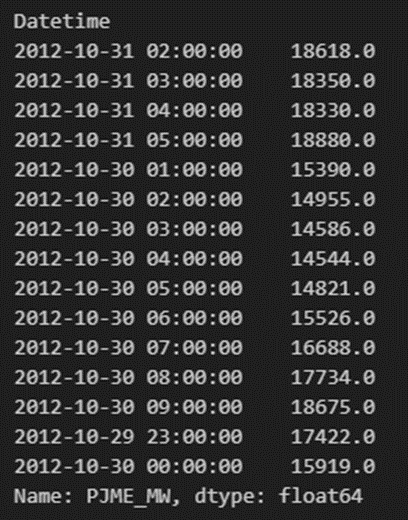


**In-Depth Outlier Analysis and Data Cleaning**

**Identification of Persistent Outliers**

A distinct period stands out in the graph, marked by a consistent straight line where energy consumption values plummet to as low as 15,000 MW. To delve deeper, we utilize the ‘query()’ method to pinpoint the specific dates and times when these outliers occurred. The code snippet is as follows:

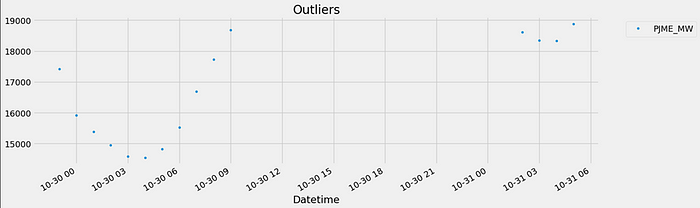
df.query('PJME\_MW < 19000')['PJME\_MW']



**Analysis of Outlier Occurrence**

Based on the table output, the outliers occurred during the night of October 29, 2012, extending until 9 am the next day. Furthermore, additional outliers emerged in the early morning hours of October 31. To visualize and gain a more detailed understanding of these occurrences, we use the following code:

df.query('PJME\_MW < 19000').plot(figsize = (15,5), style = '.')  
plt.legend(bbox\_to\_anchor=(1.04,1));



The resulting graph illustrates the date of occurrence (October 30) and the corresponding hour of the day when these outliers transpired.

**Data Cleaning**

In the final stage of outlier analysis, we create a copy of the current DataFrame using the ‘copy()’ method. We then update our DataFrame (`df`) by removing values less than 19,000 MW. The code for this data cleaning step is as follows:

df = df[df['PJME\_MW'] > 19000].copy()

This meticulous outlier analysis and data cleaning process ensure the integrity of our dataset, paving the way for more accurate and reliable time series predictions.

**Time Series Cross-Validation for Robust Model Evaluation**

**Introduction to Time Series Split**

Time series split is a specialized cross-validator designed for time series data, ensuring the chronological order of the data is maintained during the splitting process. This approach is crucial for realistic evaluation of model performance in time series analysis. By splitting the dataset into training and testing sets while respecting the temporal order, time series split provides a more accurate assessment of a model’s predictive capabilities.

**Time Series Split Configuration**

To implement time series split, the following code is utilized:

tss = TimeSeriesSplit(n\_splits = 5, test\_size = 24\*365\*1, gap = 24)  
df = df.sort\_index()

**Explanation:**

- `n\_splits`: Specifies the number of splits or folds to create, set to 5 in this case.

- `test\_size`: Defines the size of the testing set for each split, representing 1 year’s worth of data and 24 hours in a day.

- `gap`: Sets the gap between the training and testing sets, represented by 24 hours.

**Sorting the Index**

The data frame’s index is sorted in ascending order using the `sort\_index()` method, a common practice in time series split analysis to ensure the data is arranged chronologically.

**For Loop to Generate Cross-Validation Splits**

To generate cross-validation splits, a for loop iterates over the data frame to create training and validation indices. The following code snippet exemplifies this process:

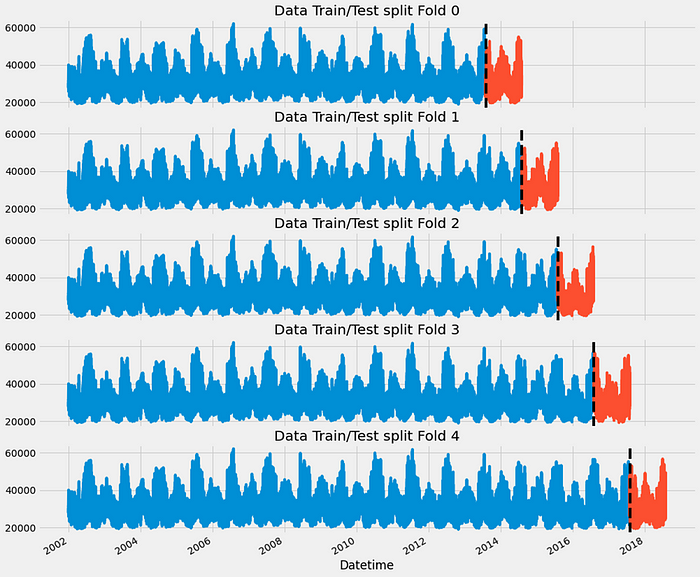
for train\_index, val\_index in tss.split(df):  
 break

This loop generates training and validation indices, which will be looped through five times to produce five different cross-validation splits spanning various years.

**Visualization of Cross-Validation Splits**

The created cross-validation splits are visualized using a subplot graph with the following code:

fig, axs = plt.subplots(5, 1,figsize = (15,15), sharex=True)  
  
fold = 0  
for train\_index, val\_index in tss.split(df):  
 train =df.iloc[train\_index]  
 test = df.iloc[val\_index]  
 train['PJME\_MW'].plot(ax = axs[fold],  
 label = 'Training Set',  
 title = f'Data Train/Test split Fold {fold}')  
 test['PJME\_MW'].plot(ax = axs[fold], label = 'Test Set')  
   
 axs[fold].axvline(test.index.min(), color = 'black', ls = '--')  
 fold += 1



This graph visually represents the training and testing sets for each cross-validation split, aiding in the understanding of how the data is partitioned over time. Stay tuned for further exploration of feature engineering and model training in our time series analysis journey.

The graph above visualizes the outcome of time series cross-validation, illustrating the testing of five different years independently. This approach is employed to ensure robust model evaluation and is particularly beneficial when dealing with large datasets. By testing on distinct yearly segments, we gain a comprehensive understanding of the model’s performance across various temporal contexts.

The decision to test independently over five different years is a strategic choice, maximizing the utility of a large dataset. As we proceed with feature engineering and model training, this approach ensures a comprehensive evaluation and enhances the model’s adaptability to different temporal patterns. Stay tuned for further developments in our time series analysis journey.

**Feature Engineering for Enhanced Time Series Analysis**

To enhance our time series analysis and capture patterns in energy consumption, we employ a feature creation function that adds several time-related features to our dataframe. These features, derived from the time series index, include Hour, Day\_of\_Week, Quarter, Month, Year, Day\_of\_Year, Day\_of\_Month, and Week\_of\_Year. This enrichment facilitates a deeper understanding of the temporal aspects of our data, aiding our model in recognizing patterns and seasonality.

**Feature Creation Function**

The feature creation function, named `create\_features`, is implemented as follows:

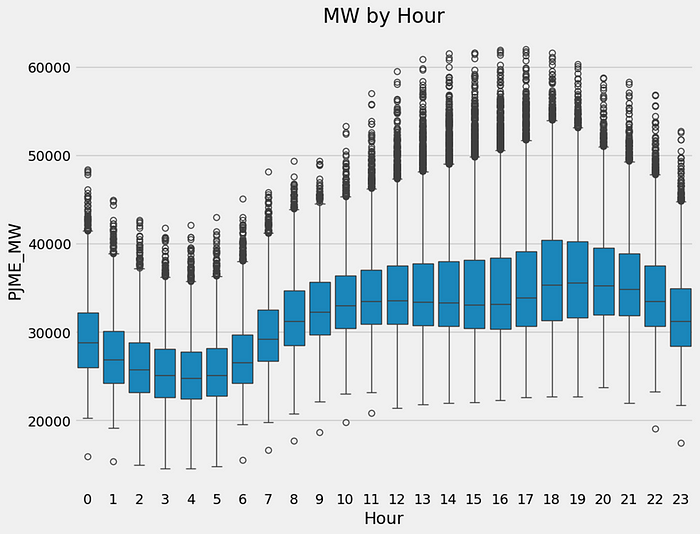
def create\_features(df):  
 """  
 Create time series features based on time series index.   
 """  
 df.copy()  
 df['Hour'] = df.index.hour  
 df['Day\_of\_Week'] = df.index.dayofweek

**Visualizing Feature Relationships**

To confirm the patterns and seasonality captured by our features, we utilize box plots to visualize the relationship between certain features and the target variable (energy consumption). Two specific visualizations are highlighted below.

1. ***Box Plot for Hourly Consumption:***

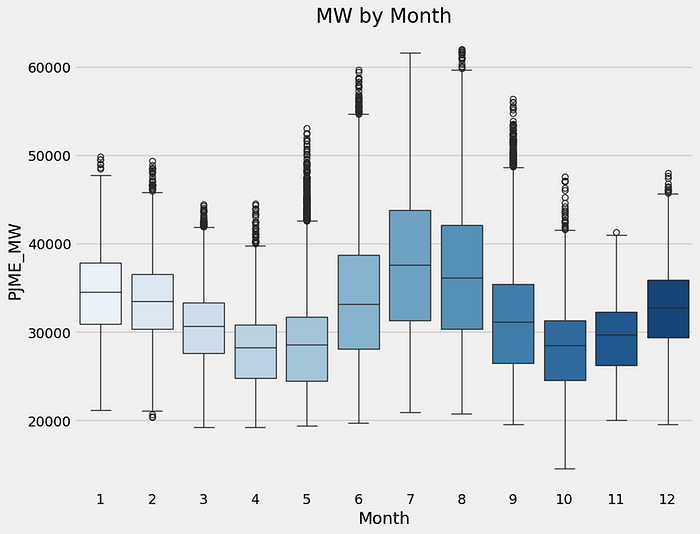
fig, ax= plt.subplots(figsize=(10, 8))  
sns.boxplot(data=df, x='Hour', y='PJME\_MW')  
ax.set\_title('MW by Hour');



The box plot illustrates a dip in energy consumption during the early morning hours, with a notable rise from 6 am onwards.

***2. Box Plot for Monthly Consumption***

fig, ax= plt.subplots(figsize=(10, 8))  
sns.boxplot(data=df, x='Month', y='PJME\_MW', palette='Blues')  
ax.set\_title('MW by Month');



This graph indicates a peak in energy consumption during the summer season, particularly in July, suggesting a correlation with increased air conditioning usage.

**Conclusion**

The feature engineering process enhances our understanding of temporal patterns within the dataset, paving the way for improved time series analysis. These visualizations affirm the existence of seasonal patterns in energy consumption, providing valuable insights for subsequent modelling and predictive analysis. Stay tuned as we delve deeper into the application of time series models on our enriched dataset.

**Lag Features for Temporal Context in Time Series Analysis**

**Introduction to Lag Features**

Lag features are instrumental in providing the model with historical context, allowing it to consider past data when making predictions. In our case, the target variable is ‘PJME\_MW,’ representing energy consumption. By creating lag features, we instruct the model to incorporate the energy consumption values from a specified number of days in the past as new features.

**Target Mapping**

To facilitate the creation of lag features, we create a dictionary (`target\_map`) that maps the ‘PJME\_MW’ values to their corresponding dates in the dataframe:

target\_map = df['PJME\_MW'].to\_dict()

**Creating Lag Features**

The `add\_lags` function is implemented to add lag features to the dataframe. In this function, three lag features (‘lag1’, ‘lag2’, and ‘lag3’) are created by referencing the target variable from 364, 728, and 1092 days ago, respectively:

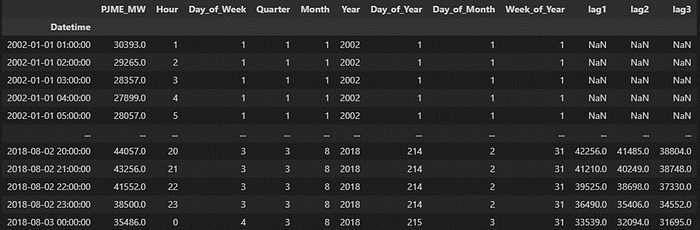
def add\_lags(df):   
 df['lag1'] = (df.index - pd.Timedelta('364 days')).map(target\_map)  
 df['lag2'] = (df.index - pd.Timedelta('728 days')).map(target\_map)  
 df['lag3'] = (df.index - pd.Timedelta('1092 days')).map(target\_map)  
 return df

***Explanation:***

- The use of 364 days instead of 365 ensures a whole number when dividing by 7, aligning with weekly patterns in the data.

- The `.map(target\_map)` retrieves the actual target values based on the mapped dates.

**Dataset Transformation**



The resulting dataframe now includes lag features, enhancing the model’s ability to capture temporal dependencies. It is important to note that the first few rows of the dataset contain NaN values for lag features since historical data for the previous years is not available. However, towards the end of the dataset, the lag features are populated with meaningful values.

**Conclusion**

The incorporation of lag features enriches the dataset, providing the model with historical information to better understand and predict energy consumption patterns. As we progress, these lag features will contribute to the accuracy and effectiveness of our time series model. Stay tuned for further developments in our time series analysis journey.

**Time Series Cross-Validation and XGBoost Regression**

**Introduction**

At this stage of our time series analysis, we employ a time series cross-validation approach using the ‘TimeSeriesSplit’ method from scikit-learn. The goal is to fit an XGBoost regressor (‘XGBRegressor’) to predict the target variable (‘PJME\_MW’). This process ensures robust evaluation of our model’s performance across multiple time periods.

**Time Series Split Configuration**

We configure the time series split (`tss`) to create 5 folds with a test size representing one year of data and a gap of 24 hours between the training and testing sets:

tss = TimeSeriesSplit(n\_splits = 5, test\_size = 24\*365\*1, gap = 24)  
df = df.sort\_index()  
  
fold = 0  
preds = []  
scores = []  
  
for train\_index, val\_index in tss.split(df):  
 train =df.iloc[train\_index]  
 test = df.iloc[val\_index]  
  
 ###run the train and test sets through create fetures function 5 different times  
 train = create\_features(train)  
 test = create\_features(test)  
  
 ### Define the features including the lags  
 FEATURES = ['Hour', 'Day\_of\_Week', 'Quarter', 'Month', 'Year','Day\_of\_Year','lag1','lag2','lag3']  
 TARGET = 'PJME\_MW'  
   
   
 X\_train = train[FEATURES]  
 y\_train = train[TARGET]  
  
 X\_test = test[FEATURES]  
 y\_test = test[TARGET]  
   
 reg = xgb.XGBRegressor(base\_score = 0.5, booster = 'gbtree',  
 n\_estimators = 1000,   
 early\_Stopping\_rounds=50,  
 objective = 'reg:linear',  
 max\_depth = 3,  
 learning\_rate = 0.01)  
  
 reg.fit(X\_train, y\_train,  
 eval\_set = [(X\_train, y\_train), (X\_test, y\_test)],  
 verbose = 100)  
   
 y\_pred = reg.predict(X\_test)  
 preds.append(y\_pred)  
 # give score to the model  
 score = np.sqrt(mean\_squared\_error(y\_test, y\_pred))  
 #save the scores into list  
 scores.append(score)

**Model Training and Evaluation**

The subsequent loop iterates through the folds generated by `tss.split(df)`. For each fold:

- It separates the data into training (`train`) and testing (`test`) sets.

- Applies the `create\_features` function to both training and testing sets for feature enrichment.

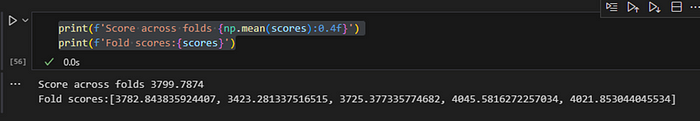
- Defines features (`FEATURES`) and the target variable (`TARGET`).

- Initializes an XGBoost regressor (`reg`) with specified hyperparameters.

- Fits the model on the training set and evaluates on the testing set using the root mean squared error.

- Appends the predictions (`y\_pred`) and the root mean squared error (`score`) to the lists `preds` and `scores`, respectively.

The score of our 5 cross validations are printed out using the following code and the overall average of our score is 3799.7874.



**Conclusion**

The implementation of time series cross-validation with XGBoost regression enhances the robustness of our predictive model. The iterative evaluation across different time periods provides a comprehensive understanding of the model’s performance. As we progress, further analysis of the results and potential model refinement will be explored. Stay tuned for more insights and developments in our time series analysis journey.

**Predicting the Future with XGBoost: Time Series Analysis**

**Introduction**

As we advance in our time series analysis journey, the next step involves predicting the future energy consumption using the XGBoost algorithm. Initially, we refine and train our model on the entire dataset, leveraging all available historical data. Subsequently, we extend our dataframe into the future, create features, and employ our trained model to make predictions on an hourly basis.

**Training the Model on the Entire Dataset**

We modify our previous code to train the XGBoost regressor (`reg`) on the entire dataset without splitting into training and testing sets:

## Retian on all data  
df = create\_features(df)  
  
### Define the features including the lags  
FEATURES = ['Hour', 'Day\_of\_Week', 'Quarter', 'Month', 'Year','Day\_of\_Year',  
 'Day\_of\_Month','lag1','lag2','lag3']  
TARGET = 'PJME\_MW'  
   
X\_all = df[FEATURES]  
y\_all = df[TARGET]  
   
reg = xgb.XGBRegressor(base\_score = 0.5,   
 booster = 'gbtree',  
 n\_estimators = 1000,   
 objective = 'reg:linear',  
 max\_depth = 3,  
 learning\_rate = 0.01)  
  
reg.fit(X\_all, y\_all,   
 eval\_set = [(X\_all, y\_all)],  
 verbose = 100)

**Extending the Dataframe into the Future**

To predict the future, we create a new dataframe (`future\_df`) with future dates and introduce a column ‘isFuture’ to identify future predictions. We then concatenate this with the existing dataframe and add features and lag values:

future = pd.date\_range('2018-08-03','2019-08-01', freq = '1h')  
future\_df = pd.DataFrame(index = future)  
  
future\_df['isFuture'] = True  
df['isFuture'] = False  
  
df\_and\_future = pd.concat([df, future\_df])  
  
# add the features  
df\_and\_future = create\_features(df\_and\_future)  
df\_and\_future = add\_lags(df\_and\_future)

**Predicting the Future with XGBoost**

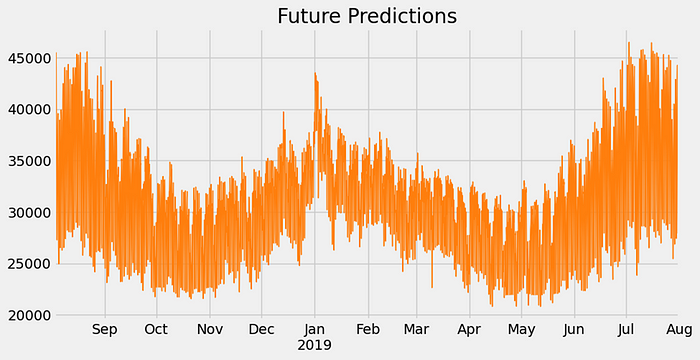
We create a new column ‘pred’ in the future dataset and use the trained XGBoost regressor to predict future energy consumption:

future\_with\_features = df\_and\_future.query('isFuture').copy()  
  
future\_with\_features['pred'] = reg.predict(future\_with\_features[FEATURES])

**Visualization of Future Predictions**

Finally, we plot a graph illustrating the predicted future values from August 2018 to August 2019 on an hourly basis:

future\_with\_features['pred'].plot(figsize = (10,5),  
 color = color\_pal[1],  
 ms=1,  
 lw=1,  
 title='Future Predictions');



**Conclusion**

This graph serves as a visual representation of our XGBoost model’s predictions for future energy consumption. While XGBoost is a powerful algorithm, there are various other methods for predicting future values, and mastery of these techniques opens doors to more sophisticated time series analyses. Stay tuned for further explorations and insights into the dynamic realm of time series forecasting.

**Overview**

XGBoost is a powerful and popular machine learning algorithm, especially for regression and classification tasks. However, like any other algorithm, it has its limitations and considerations when it comes to time series forecasting. Here are some of the limitations of using XGBoost for time series forecasting:

1. Lack of Temporal Understanding:

- XGBoost is a tree-based model that does not inherently capture the sequential nature of time series data. It treats each data point as independent, which may not be suitable for tasks where the order and time dependencies of data are essential.

2. Stationarity Assumption:

- XGBoost assumes that the statistical properties of the data do not change over time, which might not hold for many time series data. Time series data often exhibit trends, seasonality, and other temporal patterns that XGBoost may not handle well without proper preprocessing.

3. Seasonality Handling:

- Dealing with seasonality (e.g., daily, weekly, or yearly patterns) in time series data can be challenging for XGBoost. While some preprocessing techniques can be applied, incorporating seasonality effectively may require domain-specific knowledge.

4. Handling Time Lags:

- XGBoost doesn’t inherently account for time lags, which are often crucial in time series forecasting. Lagged variables or other time-related features must be manually engineered and included in the model.

5. Hyperparameter Tuning:

- XGBoost requires careful hyperparameter tuning, which can be time-consuming. In time series forecasting, it’s essential to find the right balance between model complexity, regularization, and other hyperparameters.

6. Handling Missing Values:

- Time series data often have missing values due to gaps or irregularities in data collection. Handling missing values appropriately can be challenging, and it requires additional preprocessing steps.

7. Forecast Horizon:

- XGBoost’s forecasting capabilities may be limited when predicting over long time horizons. Forecasts become less accurate as the time horizon increases, and short-term predictions are generally more reliable.

8. Large Datasets:

- For very large time series datasets, training an XGBoost model can be computationally expensive and may require significant computational resources.

9. Model Interpretability:

- While XGBoost is a powerful model, its black-box nature may make it challenging to interpret and explain forecasts, which can be important in some applications.

To address these limitations, practitioners often combine XGBoost with other time series forecasting techniques, such as autoregressive models (ARIMA), exponential smoothing (ETS), or Long Short-Term Memory (LSTM) networks, which are better suited for capturing temporal dependencies and handling seasonality. Additionally, preprocessing, feature engineering, and careful consideration of the problem domain play a crucial role in successful time series forecasting using XGBoost.

I have saved the model in a ‘.json’ file so that I don’t have to train the data over and over again next time I come across it. The file is uploaded in my GitHub repository and you are welcomed to access it [here](https://github.com/Ethann93/Ultra-Marathon.github.io).

[Data Science](https://medium.com/tag/data-science?source=post_page-----5d299655bec6---------------data_science-----------------)

[Xgboost](https://medium.com/tag/xgboost?source=post_page-----5d299655bec6---------------xgboost-----------------)

[Crossvalidation](https://medium.com/tag/crossvalidation?source=post_page-----5d299655bec6---------------crossvalidation-----------------)

[Machine Learning](https://medium.com/tag/machine-learning?source=post_page-----5d299655bec6---------------machine_learning-----------------)

[Visualization](https://medium.com/tag/visualization?source=post_page-----5d299655bec6---------------visualization-----------------)