Forecasting Hourly Demand For EV Charging

Abhilash

Table of Content:

A. Data Cleaning and Data Preprocessing

The information we got has some important parts like when something was connected ('connectionTime'), when it was disconnected ('disconnectTime'), when charging was done ('doneChargingTime'), and how much energy was used ('kWhDelivered'). We decided that the start time would be when it was connected and the end time would be when it was disconnected or when charging was done, whichever came first. If we didn't have one of those times, we just used the one we did have as the end time. Then, for each ID, we figured out how much power was being used every second between the start and end times. After that, we grouped together the power usage for each hour. So now, our final list has two things: the time each hour represents ('timestamp') and how much power was used during that hour ('power').

B. Feature Engineering

We realized that our chosen model could perform even better with some additional features. One important reason for this is that there are many factors that can cause variations in power consumption in time series data. For example, public holidays or events like the COVID-19 pandemic in 2020, with lockdowns impacting EV power usage. Additionally, people tend to charge their vehicles more on weekdays compared to weekends when they stay at home, which creates cyclic patterns. For instance, after December comes January, leading to a linear progression. To capture these patterns, we incorporated several features into our model.

Firstly, we included a feature called **'is_covid'**, where we set the value to '1' if the date is on or after April 1, indicating the period of lockdown in 2020.

Furthermore, we utilized the **sine** and **cosine** of the month, hour, and day to identify cyclic patterns in the data.

Additionally, we incorporated the **weekday** (coded as 1 for Monday, 2 for Tuesday, and so on) as a feature to discern usage patterns between weekdays and weekends.

C. Data Analysis and Preprocessing:

To enhance the understanding of the dataset, we considered a variety of analyses.

1. **Power consumption vs Time plot:**

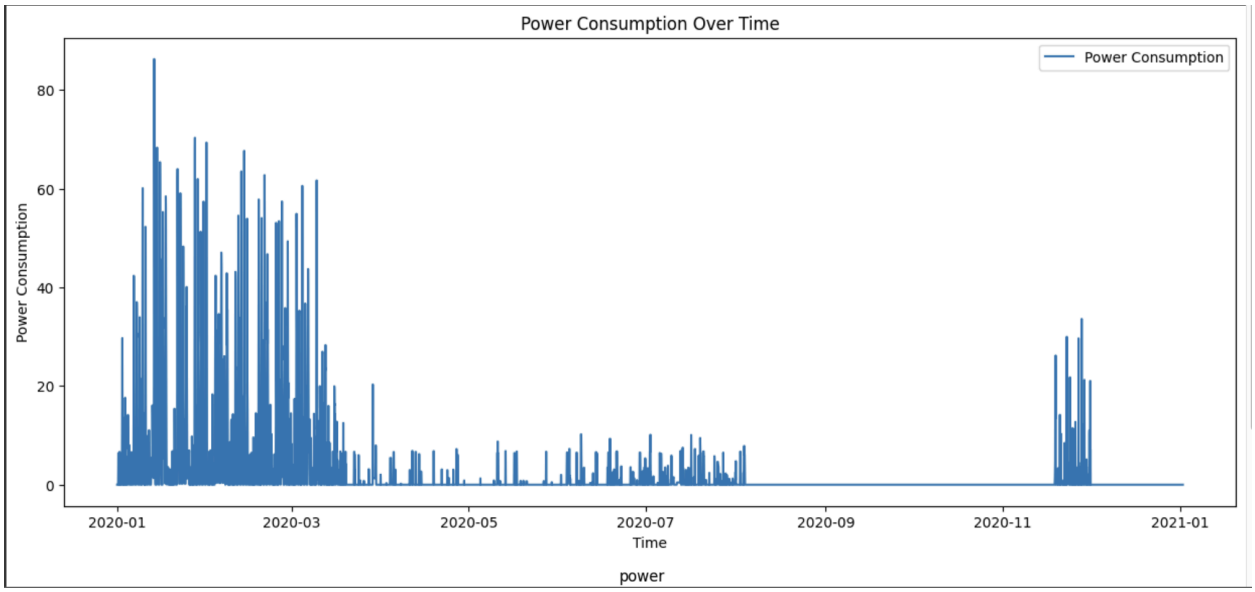

Figure 1: Power consumption vs Time

From the plot shown in Figure 1, it's evident that power consumption was notably high during the initial three months of 2020. However, after March, there's a significant decrease in power usage. This could be attributed to the COVID lockdown implemented around that time. Additionally, there are noticeable gaps in the data, such as from August to October 2020. Furthermore, there are instances where there's no recorded power consumption for certain hours on specific days. This could either be due to missing data or actual absence of consumption.

To handle these missing values, we applied imputation techniques like back fill and averaging. The analysis for these two are provided later. This was necessary because time series data often follows specific patterns over time. Back fill ensures that the chronological order of the data is maintained by replacing missing values with the most recent observed ones. This method is particularly useful when there's expected continuity in the data. On the other hand, averaging helps in preserving the statistical characteristics of the dataset, such as the mean, variance, and distribution. Maintaining these properties is essential for the integrity of the time series data, especially when the

missing values are randomly distributed. Imputing with averages also helps in reducing bias in the analysis.

2. Time Series Decomposition

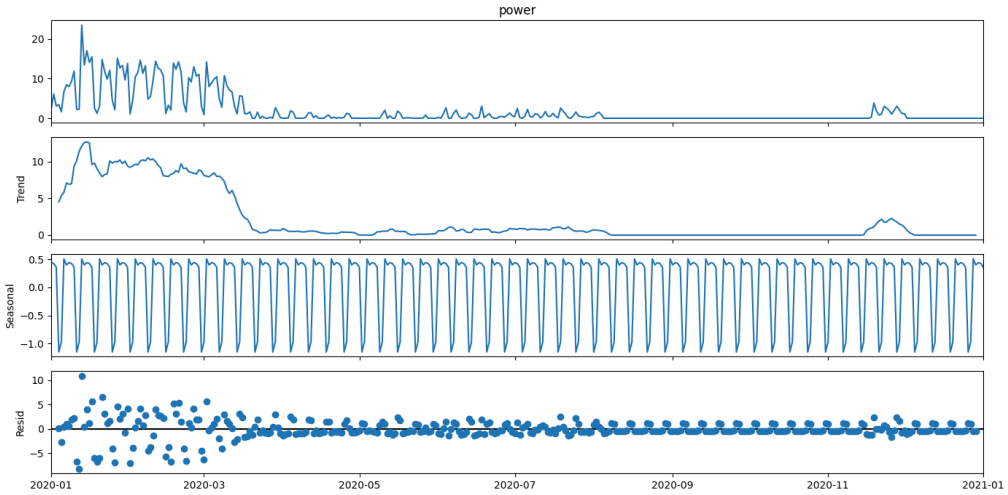

Figure 2: Time series decomposition

The Time Series Decomposition plot breaks down the daily average power consumption into three components:

a. **Trend**: This indicates the long-term movement in power consumption, helping to identify whether the usage is increasing, decreasing, or stable over time.
b. **Seasonality**: This component reveals any regular patterns that repeat over a known period, such as weekly cycles in this case. It's useful for understanding predictable fluctuations in consumption.
c. **Residual**: These are the irregularities that cannot be explained by the trend or seasonality. It includes random fluctuations and potentially anomalous events.

3. Lag Features and Autocorrelation

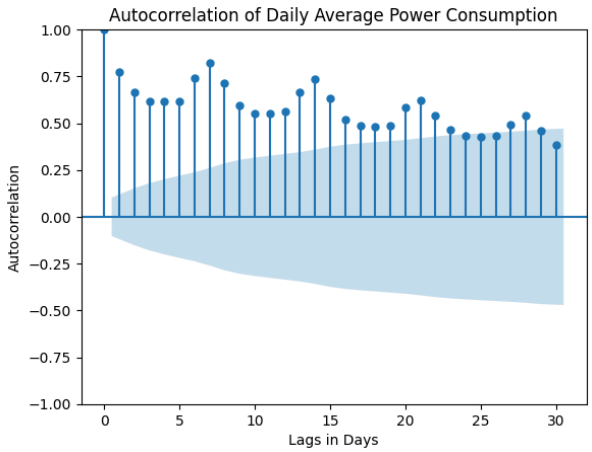

Figure 3: Auto correlation between daily average power consumption

A significant autocorrelation at specific lags suggests that past values of power consumption can be predictive of future values. For instance, if the autocorrelation is strong at lag 7, it indicates a weekly pattern where the power consumption today is similar to what it was a week ago.

4. Rolling Window Statistics

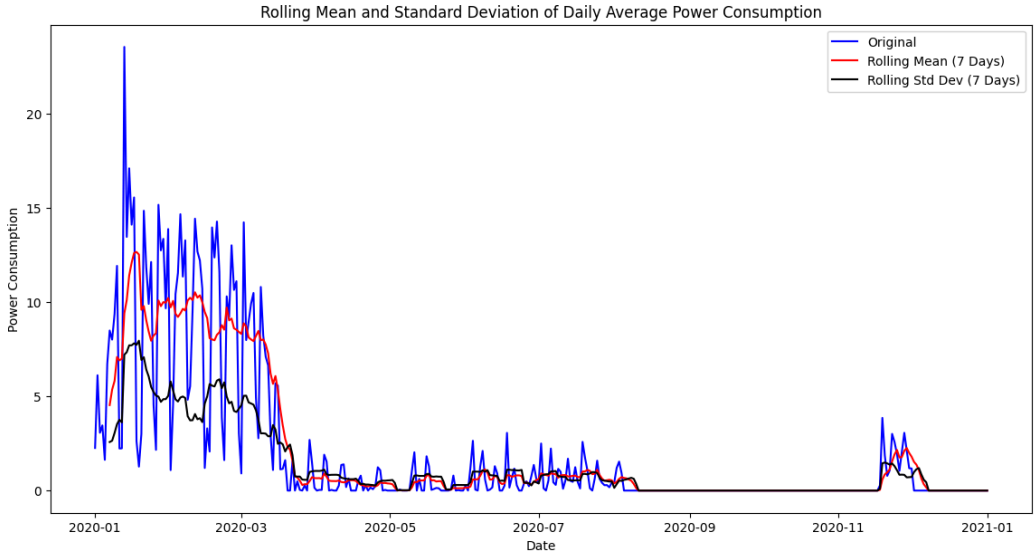

Figure 4: Rolling mean and standard deviation of daily average power consumption

The above plot focuses on a 7-day rolling mean and standard deviation to highlight short-term trends and volatility. This will help us understand how power consumption varies over a week.

Rolling Mean (Red Line): This line smooths out fluctuations to reveal the underlying trend in power consumption over time. It highlights periods of increase or decrease in consumption, offering a clearer view of the overall trend.

Rolling Standard Deviation (Black Line): This measures the variability or volatility in power consumption. Periods of high volatility indicate significant fluctuations in consumption, whereas low volatility suggests more stable consumption patterns.

5. Correlation Heat Map

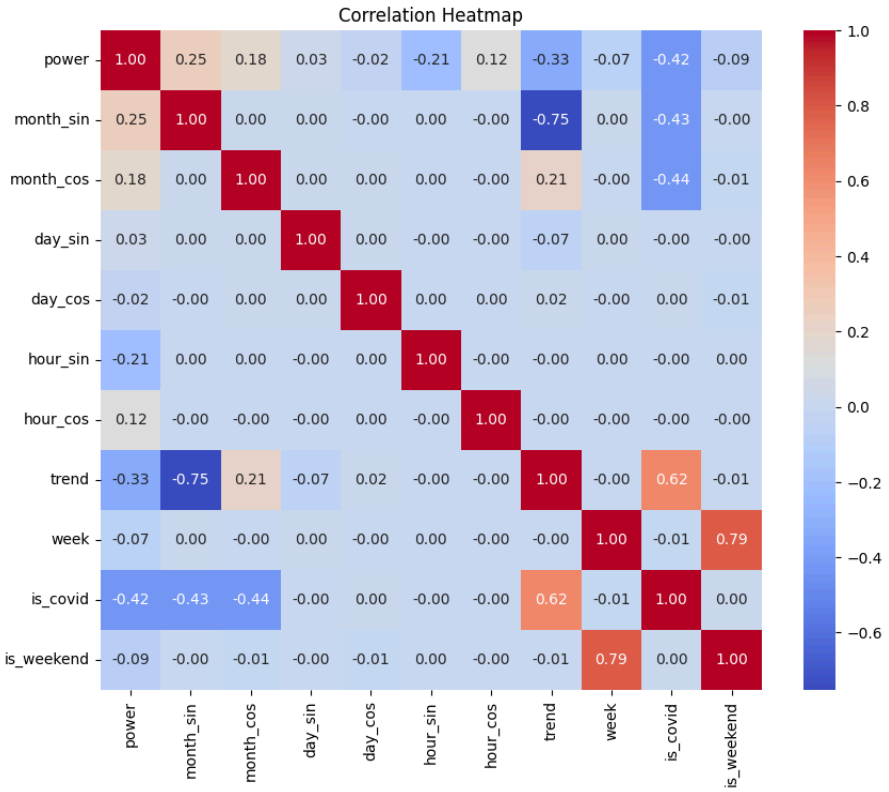

Figure 5: Correlation heat map

The above heat map tells us that. There is kind of high correlation between the **trend** and **is_covid.** Also, between **month_sin** and **trend.** We can also observe a little high correlation between **is_covid** and **power**, which tells us that power consumption is high when there is no covid.

To sum up, we filled in missing data using backfill and averaged values. Now, let's talk about choosing the right model.

D. Model Selection

We tried out various models like **GradientBoostRegressor, ARIMA, SARIMA, LSTM,** and **Prophet**. We'll analyze how well each of these models performed based on the data we have. The reason why we chose these models is because GradientBoost can be able to identify complex patterns in the data(non-linear relationships). However, the other models are specifically designed for time series data that is able to capture patterns such as trends and seasonality. Moreover, SARIMA is useful when there are fluctuations either daily, weekly, or monthly.

Training and Testing Method:
For every model we chose, we took each month's data as test data and trained the model on the remaining data.

Baseline Model:

For our simple model, we only used **three** features: 'month', 'day', and 'hour'. Additionally, we filled in any missing values using a backfill strategy. We checked how well this basic model performed with **GradientBoostRegressor**, **ARIMA**, **SARIMA**, **LSTM**, and **Prophet**. The GradientBoost model had an error of about 9 for the first month, while ARIMA, SARIMA, and Prophet had errors of 9, 12, 9, and 8 respectively for the first month.

We also tested the models with the same three features but by removing the backfill strategy. The results were reasonably the same as with the backfill strategy.

Since our baseline models had only a few features, we expected the errors to be higher. Also, it couldn't spot patterns that weren't already in the data. That's why we added new features, as we described about them in the feature engineering part. Now, let's discuss models with more features, as we mentioned earlier.

Model with Sophisticated Features:

We used the same features and models as before, but this time, we applied Hyper Parameter Tuning. This technique significantly improved the GradientBoostRegressor's performance, reducing the Mean Absolute Error (MAE) by about 3.5. However, the performance of the other models remained almost the same, with only a slight increase or decrease of about 0.5 compared to the baseline model. Hyper Parameter Tuning along with Feature Engineering proved to be very effective for the GradientBoostRegressor, making it perform much better, but it didn't make a big difference for the other models.

E. Diagnostics

We used GridSearchCV that inherently helps in underfitting and overfitting. Basically, GridSearchCV uses cross-validation to assess the performance of different hyperparameter combinations.

Evaluation:

1. **Cross-Validation Process**: Cross-validation involves dividing the training data into several smaller parts, called "**folds**."(we used 5 folds) For every set of settings we're testing, we train the model on all but one of these folds and then test it on the fold we didn't use for training. We repeat this process, each time using a different fold for testing,

until every fold has been used for testing exactly once. Then, we average the results from all these rounds to see how well the model does overall.

2. **Detecting Underfitting**: Underfitting happens when the model performs poorly both during training and testing, no matter what settings we use. This usually means the model is too simple. With **cross-validation**, we try increasing the model's complexity (like using **more trees** in a forest model or letting it make deeper decisions) to see if it improves its performance on the tests.
3. **Detecting Overfitting**: Overfitting is noticed when the model does really well on the training data but not so well on the testing data. This means the model is too complicated, catching random noise as if it were true patterns. **Cross-validation** is used to find the best balance where the model does well in both training and testing. Adjusting settings such as making the model simpler or using techniques to prevent it from learning the noise too well can help fix overfitting.

Regularization:

In the context of our GradientBoostRegressor model, regularization is achieved by tuning several key hyperparameters that control the model's complexity and its ability to learn from the training data without overfitting. These include:

1. **regressor__n_estimators** (Number of Boosting Stages)
    a. **Purpose**: Determines how many trees to build. More trees can improve accuracy but too many might cause overfitting.
    b. **Regularization Role**: Balances complexity and overfitting risk; finding the right number is key to avoiding overfitting.

2. **regressor__learning_rate** (Step Size Shrinkage)
    a. **Purpose**: Controls how quickly the model learns, with smaller steps requiring more trees but potentially leading to a more accurate model.
    b. **Regularization Role**: A lower rate slows learning, helping prevent overfitting by making each tree's impact on the final model smaller.
3. **regressor__max_depth** (Maximum Depth of Trees)
    a. **Purpose**: Limits how deep each tree can grow, affecting the model's ability to learn detailed patterns.
    b. **Regularization** Role: Constraints tree complexity to prevent fitting to noise, aiding in generalization.
4. **regressor__min_samples_split** (Minimum Samples to Split)
    a. **Purpose**: Sets the minimum number of samples required to split a node, influencing tree depth and complexity.
    b. **Regularization Role**: Higher thresholds help avoid overly complex models by reducing over-specific splitting.
5. **regressor__subsample** (Subsample)
    a. **Purpose**: Determines the fraction of data used for training each tree, introducing randomness.

b. **Regularization Role**: Reduces variance and helps avoid overfitting by ensuring trees don't just memorize the data.

F. Performance Evaluation

We chose the GradientBoostRegressor as our best model because it worked really well. It uses the special features we made, along with some adjustments (called hyperparameters) and a backfill strategy. Now, we're going to compare how well this model does in terms of mistakes (MAE) with the other models and strategies we talked about.

	GradientBoost	ARIMA	SARIMA	Prophet	LSTM
January	5.64	9.93	12.67	8.36	8.51
February	5.22	10.31	12.47	8.62	8.65
March	3.64	8.31	12.72	4.87	8.04

The table above compares the Mean Absolute Error (MAE) for different models. We see that GradientBoost performs better than other time-series models. Each month serves as a test month for evaluation.

	GradientBoost	ARIMA	SARIMA	Prophet	LSTM
January	9.24	11.9	12.00	13.19	12.61
February	8.19	11.80	12.10	11.37	8.66
March	8.87	9.11	11.04	9.99	8.21

The tables above display the Mean Absolute Error (MAE) for the **baseline** features we discussed previously. The earlier table/model performs well, achieved through the implementation of backfilling strategy, feature engineering, and Hyperparameter Tuning.