INTERNSHIP PROJECT REPORT

on

DATA SCIENCE PROJECT

Completed at

COAPPS

Project Title

Forecasting Energy Demand: Harnessing IOT Data for Smart Grid Analytics

Duration
26th March 2024 to 27th May 2024

Role
Junior Data Scientist (Intern)

submitted by

THANUSHRI A (20IT1057)
JEEVITHAA S(20IT1050)

DEPARTMENT OF INFORMATION TECHNOLOGY

PUDUCHERRY TECHNOLOGICAL UNIVERSITY
PUDUCHERRY-605 014.

May 2024

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to the following individuals and organizations for their invaluable support and contribution to the successful completion of this project report on "Forecasting Energy Demand: Harnessing IoT Data for Smart Grid Analytics."

I am grateful to Coapps for providing me with the opportunity to work on this cutting-edge project, allowing me to apply and enhance my data science skills in forecasting energy demand using Internet of Things (IoT) data for smart grid analysis.

I am particularly indebted to Amyjoy Exson sir and Preethi Murali mam for their constant guidance and encouragement throughout the project. Their expertise in the python programming and data science was instrumental in shaping the direction and ensuring the technical soundness of my work on utilizing Python for smart grid analytics.

I am thankful to Puducherry Technological University for providing a strong foundation in Information Technology, which prepared me for this challenging and rewarding internship experience.

I would also like to express my gratitude to Dr. R. Elansezhian, the Training & Placement Officer at Puducherry Technological University, for facilitating the college placement process that led to my internship opportunity at Coapps.

I would also like to thank my colleague for her support and collaboration during this project. Her insights into the practical applications of smart grid technologies were particularly helpful.

Finally, I would like to express my gratitude to my family and friends for their unwavering support and encouragement throughout this project.

THANUSHRI A

JEEVITHAA S

ABSTRACT

This project investigates forecasting energy demand, the process of predicting the future electricity consumption patterns required to meet consumer needs within a specific timeframe (e.g., hourly, daily), in smart grids. We explore the application of various machine learning models for this task. We achieve this by harnessing data collected from Internet-of-Things (IoT) devices, such as smart meters, which provide granular insights into energy usage within the grid. To prepare this data for machine learning models, we employ data preprocessing techniques to clean, transform, and format the data. Additionally, we perform exploratory data analysis (EDA) to understand the characteristics of the data, identify patterns, and guide the selection of appropriate models.

We explore the performance of various machine learning models, including Long Short-Term Memory (LSTM), Autoregressive Integrated Moving Average (ARIMA), Vector Autoregression (VAR), Support Vector Regression (SVR), Holt-Winters Exponential Smoothing, and XGBoost models. These models are used to predict future energy demand based on historical consumption, weather data, and derived features like day of week and time of day. The models' performance is evaluated using a comprehensive set of metrics including Root Mean Squared Error (RMSE), Mean Squared Error (MSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), Explained Variance, R-squared, and Maximum Error. We discuss the strengths and weaknesses of each model, analyse the results, and explore the benefits of accurate energy demand forecasting.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
ABOUT THE COMPANY

Coapps Development Solutions Private Limited, established in 2018, has become a prominent player in Chennai's IT landscape. They position themselves as a one-stop shop for businesses seeking digital transformation. Their core expertise lies in crafting custom web and mobile applications, leveraging cutting-edge technologies like artificial intelligence, machine learning, blockchain, and data analysis.

While the company's history isn't fully documented online, their growth trajectory since 2018 is impressive. News articles suggest a team exceeding 140 professionals and a substantial increase in profitability in 2023, highlighting their recent expansion and success. Though information about the specific founders remains elusive, Coapps' mission can be gleaned from their service offerings and areas of expertise. They strive to deliver innovative and impactful digital solutions that empower businesses across various industries. Their focus lies on leveraging cutting-edge technologies to streamline operations, enhance efficiency, and drive growth for their clients. They achieve this by providing a comprehensive suite of software development, mobile application development, and digital transformation services.

Coapps offers a diverse range of services, including custom web and mobile application development for both Android and iOS platforms. They specialize in cloud solutions like SaaS, PaaS, and IaaS, and seamlessly integrate AI, machine learning, blockchain, and data analysis into their projects. Their expertise extends to various industries, including banking, healthcare, and EdTech. They even provide digital marketing and recruitment services, catering to a wider range of business needs. Notably, they develop specialized applications for Fintech and gaming, and are actively exploring the potential of the Metaverse. Additionally, Coapps offers custom software solutions tailored to address specific client requirements.

In conclusion, Coapps Development Solutions Private Limited appears to be a flourishing company with a strong presence in Chennai's IT sector. Their focus on innovation and comprehensive service offerings makes them a valuable partner for businesses seeking digital transformation.

CHAPTER 2

WORK RESPONSIBILITIES DURING THE INTERNSHIP

Our internship at Coapps Development Solutions Private Limited as a Junior Data Scientist Intern provided a dynamic learning experience. The program was structured to equip interns with the necessary skills before tackling practical projects.

The initial phase focused on building a strong foundation in data science tools. This included:

- **Version Control (Git & GitHub):** We mastered the fundamentals of Git for version control and GitHub for code collaboration. This ensured proper tracking and management of our projects throughout the internship.
- **Python Programming:** We delved into Python programming, grasping syntax, data types, control flow, and functions. Python served as the core language for all subsequent data science tasks and we started by building basic Python codes to reinforce our grasp of syntax and core functionalities.
- **Data Manipulation (NumPy & Pandas):** We explored the functionalities of NumPy for numerical computing and Pandas for data manipulation and analysis to manipulate and analyse real-world datasets. This involved tasks like data cleaning, filtering, and feature creation. These libraries became instrumental in working with large datasets efficiently.
- **Data Visualization (Matplotlib):** We learned to create informative visualizations using Matplotlib, a crucial skill for communicating insights gleaned from data analysis.
- cleaning, filtering, and feature creation.
- **Creating a Streamlit App:** We learned to build a Streamlit application, a web framework for deploying data science models as interactive dashboards. This exercise bridged the gap between data analysis and user interaction.
- **Data Science Project Development:** Through practical exercises, we gained experience in the entire data science workflow, encompassing data acquisition, cleaning, analysis, modelling, and visualization and we developed a project applying our skills.

This experience at Coapps not only equipped us with technical skills but also highlighted the importance of collaboration and working on industry-relevant projects. We are grateful for the opportunity to have contributed to Coapps' efforts in smart grid analytics and energy forecasting.

CHAPTER 3
PROJECT DESCRIPTION

3.1 INTRODUCTION

Energy. It fuels our lives, powers our industries, and underpins the very fabric of modern civilization. From the lights illuminating our homes to the screens we interact with daily; energy is an invisible force driving progress. However, this progress comes with a growing appetite – our demand for energy is relentlessly increasing. Accurate energy demand forecasting hinges on a specific mix of features: historical consumption data (hourly/daily) for understanding trends and baselines, weather data (temperature, humidity, others) to predict weather-driven fluctuations, calendar data (day of week, holidays) to account for cyclical variations, and economic activity indicators to capture demand shifts due to economic growth or downturns. Population data is crucial, as rising populations translate to increased demand. Location data (time zone, geography) and sector-specific information further refine forecasts. Then the energy mix, that is the proportion of different energy sources like coal, solar, hydro etc. can affect overall demand and predictability. The icing on the cake comes from real-time features: smart meter data provides granular insights into individual customer consumption, while sensor data and connected appliance data identify potential supply disruptions and offer insights into specific appliance categories within a sector, informing targeted demand-side management programs. This comprehensive approach provides the necessary groundwork for building robust forecasting models.

While this project aims to develop machine learning models for energy demand forecasting in a smart grid, a critical decision is selecting the appropriate analysis approach considering the use of multiple features. Univariate analysis, focusing on single variables at a time, wouldn't be ideal. Our project benefits from multivariate analysis, which analyses the relationships between multiple variables simultaneously. This is crucial because we're incorporating historical energy consumption data alongside weather data (temperature, humidity) to capture how these factors influence each other.

3.2 SYSTEM REQUIREMENTS

3.2.1 Hardware Requirements

Personal Computer (or Laptop) with the following recommended specifications:

- Processor: Minimum Intel Core i5 or equivalent AMD processor
- RAM: Minimum 8 GB RAM
- Storage: Sufficient storage space to accommodate the chosen dataset and project files – 10MB minimum.

3.2.2 Software Requirements

- **Development Environment:**
    - Anaconda Navigator: https://www.anaconda.com/ (Python distribution with scientific libraries pre-installed)
    - Jupyter Notebook: https://jupyter.org/ (Interactive web-based environment for code execution and data exploration)
    - Visual Studio Code: https://code.visualstudio.com/ (Versatile code editor)
    - Python 3.9.10: https://www.python.org/downloads/ (Open-source, high-level programming language widely used for machine learning and data science).
- **Version Control System:**
    - Git: https://git-scm.com/ (Version control system for tracking code changes)
    - GitHub: https://github.com/Index (Cloud-based platform for version control and code sharing)
- **Report and Presentation Generation:**
    - **Microsoft Word:** https://www.microsoft.com/en-us/microsoft-365/word (creating comprehensive reports documenting the project methodology, results, and insights.)
    - **Microsoft PowerPoint:** https://www.microsoft.com/en-us/microsoft-365/powerpoint (creating visually appealing presentations to effectively communicate project findings to stakeholders.)
- **Data Analysis and Machine Learning Libraries (installed using pip, Python's package manager):**
    - joblib==1.3.2 (Model persistence and sharing)
    - matplotlib==3.8.3 (Data visualization)
    - numpy==1.26.4 (Numerical computing)
    - pandas==2.2.2 (Data manipulation and analysis)

- scikit-learn==1.4.1. post1 (Machine learning algorithms)
- seaborn==0.13.2 (Statistical data visualization)
- statsmodels==0.14.2 (Statistical modelling)
- tensorflow==2.10.1 (TensorFlow machine learning framework, used for LSTM models)
- tabulate==0.9.0 (Printing data in tabular format)
- tqdm==4.66.4 (Progress bar for code execution)
- xgboost==2.0.3 (Gradient boosting framework)
- Streamlit==1.34.0 (Web app development framework for deploying models as interactive dashboards)

3.3 SYSTEM DESIGN

This chapter outlines the proposed system for energy demand forecasting harnessing IoT dataset for smart grid analytics. The system leverages machine learning models trained on historical energy consumption data and weather information to predict future demand patterns. The figure 3.1 shows the system architecture of our project.

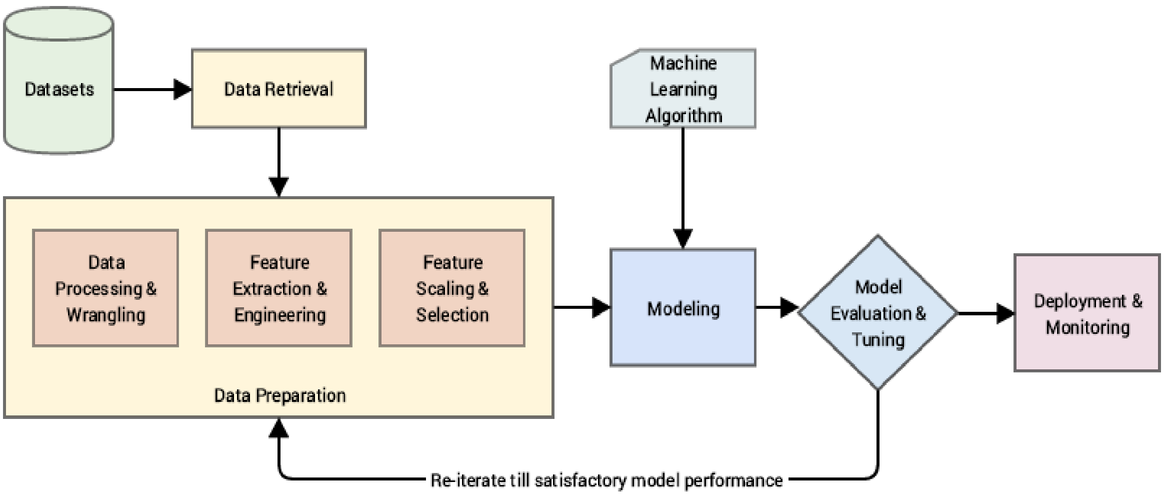


Figure **Error! No text of specified style in document.**.1 SYSTEM ARCHITECTURE

3.3.1 DATA ACQUISITION

The project utilized the "Electricity Load - Logistics - IoT" dataset obtained from Kaggle: https://www.kaggle.com/datasets/nasirayub2/electricityload-logistics-iot/data. This dataset contains a comprehensive collection of electricity-related metrics, environmental conditions, and other influencing factors. Key features relevant to this project include:

- **Time Period:** The dataset spans from January 1, 2015, to January 30, 2024, encompassing a total of 3317 records.
- **Features:** It includes a rich collection of features that capture various aspects of electricity consumption, environmental conditions, and other influencing factors. Here's a breakdown of the key features:
- **Target Variable:**
    - Electricity_Load: Represents the electricity consumption at each timestamp, a crucial metric for understanding energy demand patterns.
- **Environmental Factors:**
    - Temperature and Humidity: Environmental conditions that can significantly influence energy usage, especially for heating or cooling systems.
- **Time-Related Features:**
    - Day_of_Week: Categorizes the records by day of the week, providing insights into weekly consumption patterns.
    - Time_of_Day: Segregates timestamps into Morning, Afternoon, Evening, or Night, offering a nuanced understanding of daily electricity usage trends.
- **Additional Influencing Factors:**
    - Holiday_Indicator: A binary indicator (1 or 0) denoting whether the day is a holiday, as holidays often influence energy demand.
    - Previous_Load: Offers the electricity load in the timestamp immediately preceding the current one, aiding in analyzing load dynamics and trends.
    - Transportation_Data & Operational_Metrics: Information related to transportation and the operation of the electricity system, respectively.
    - IoT_Sensor_Data: Data from Internet of Things (IoT) sensors, indicating the integration of IoT technologies for optimizing energy usage and efficiency.
    - External_Factors: Encompasses external factors impacting electricity demand, such as regulatory, other or economic elements.
- **Demand and Price Prediction:**
    - Day_Ahead_Demand: Predicts electricity demand for the day ahead, providing valuable insight for energy planners and grid operators.
    - Real-Time_LMP & Day_Ahead_LMP: Real-time and predicted Locational Marginal Prices, a key factor in understanding the economic dynamics of electricity pricing.
- **Cost and Load Information:**

- Regulation_Capacity: Represents the capacity reserved for regulation, highlighting the system's flexibility to adapt to demand changes.
- Day_Ahead_ (EC, CC, MLC) & Real-Time_ (EC, CC, MLC): Predicted and actual values for electricity cost, congestion cost, and marginal loss cost.
- System_Load: Provides the overall electricity load on the system at each timestamp.

Table 3.1 Electricity Load Dataset Features

Feature	Range	Unit
Timestamp	-	Datetime
Electricity_Load	500.0 - 999.0	kW (kilowatts)
Temperature	-10.0 - 34.0	Celsius (°C)
Humidity	20.0 - 79.0	Percentage (%)
Day_of_Week	0 (Sunday) to 6 (Saturday)	Numerical
Time_of_Day	Morning, Afternoon, Evening, Night	Categorical
Holiday_Indicator	0 (Not a holiday) or 1 (Holiday)	Numerical
Previous_Load	500.0 - 999.0	kW (kilowatts)
Transportation_Data	10.0 – 49.0	Numerical
Operational_Metrics	100.0 – 499.0	Numerical
IoT_Sensor_Data	0.0 – 1.0	Numerical
External_Factors	Regulatory, Economic and other	Categorical
Day_Ahead_Demand	500.0 - 999.0	kW (kilowatts)
Real-Time_LMP	20.0 – 50.0	$/unit (likely currency)
Regulation_Capacity	50.0 – 99.0	kW (kilowatts)
Day_Ahead_LMP	30.0 – 60.0	$/unit (likely currency)
Day_Ahead_EC	5.0 – 20.0	$/unit (likely currency)
Day_Ahead_CC	2.0 – 9.99	$/unit (likely currency)
Day_Ahead_MLC	1.0 – 5.0	$/unit (likely currency)
Real-Time_EC	10.0 – 25.0	$/unit (likely currency)
Real-Time_CC	1.0 – 7.99	$/unit (likely currency)
Real-Time_MLC	0.5 – 3.0	$/unit (likely currency)
System_Load	500.0 - 999.0	kW (kilowatts)

This dataset offers a wealth of information for building machine learning models to forecast energy demand in smart grids. With its comprehensive collection of features encompassing electricity consumption, environmental conditions, temporal aspects, and various influencing factors, it provides a solid foundation for analysing historical usage patterns and predicting future demand.

3.3.2 DATA CLEANING AND DATA PREPROCESSING

The raw data from the "iot_load_data.csv" file underwent a series of cleaning and preprocessing steps to ensure its quality and suitability for building an energy demand forecasting model. These steps addressed potential issues like missing values, data type inconsistencies, outliers, and categorical data variations.

- **Missing Value Handling:**
    - The initial inspection involved identifying missing values in each column using methods like df.isnull().sum().
    - Based on the quantity and importance of missing entries, a suitable strategy was adopted. This could involve:
        - **Removal:** Eliminating rows with missing values (using df.dropna()) if the number of missing entries was minimal and their impact was negligible.
        - **Imputation:** Employing techniques like mean/median imputation or more advanced methods based on the data's characteristics for more substantial missing value presence or crucial data.
- **Outlier Treatment:**
    - Techniques like boxplots or IQR (Interquartile Range) were employed to identify outliers in the numerical columns. This ensured data quality by addressing extreme values that deviated significantly from the typical data range.
    - Capping, a common outlier handling technique, was implemented. Upper and lower bounds were established based on domain knowledge or statistical methods (e.g., IQR) to replace outlier values with these designated thresholds.
- By meticulously following these data cleaning and preprocessing steps, the quality and usability of the energy demand dataset were significantly enhanced. This prepared

the data for the subsequent modeling stage, ensuring the robustness and accuracy of the energy demand forecasting model.

3.3.3 FEATURE ENGINEERING

Feature engineering aimed to transform and create new features that might improve model performance. So we employed these steps to our dataset.

- **Timestamp Management:**
    - The "Timestamp" column was recognized as an object type, potentially containing string representations of timestamps.
    - To facilitate time-based analysis and feature extraction, the column was converted to a datetime format using pd.to_datetime(df['Timestamp']). Day, month, and year were then extracted as separate features using functionalities like dt.day, dt.month, and dt.year.
- **Categorical Data Preprocessing:**
    - Categorical features were examined for inconsistencies in casing (uppercase/lowercase) and the presence of leading/trailing spaces.
    - To achieve consistency, string manipulation techniques like df['column_name'].str.upper() or .str.lower() were used for casing, while .str.strip() eliminated leading/trailing spaces from the data.
    - Label encoding, a method for converting categorical data to numerical representations suitable for machine learning models, was implemented. This involved using from sklearn.preprocessing import LabelEncoder to create a label encoder and transform the categorical columns.

To enrich the dataset and potentially improve model performance, feature engineering techniques were applied. The "Timestamp" column was converted to datetime format for time-based analysis, extracting day, month, and year. Categorical data underwent preprocessing to ensure consistency in casing and removal of leading/trailing spaces. Label encoding transformed categorical features into numerical representations suitable for machine learning models. With these enhancements, we are prepared to begin the Exploratory Data Analysis (EDA) stage.

3.3.4 EXPLORATORY DATA ANALYSIS

The Exploratory Data Analysis (EDA) stage focused on delving deeper into the characteristics and relationships within the energy demand dataset. The objective was to gain valuable insights that would guide the development of an accurate energy demand forecasting model.

Numerical Data Exploration

- **Histograms:** Histograms were constructed for numerical columns like Electricity_Load, Temperature, and others. These visualizations provided a clear picture of the distribution of values within each feature.
- **Boxplots:** Boxplots were created for key numerical variables. These plots effectively revealed the central tendency (median), spread (interquartile range), and presence of outliers in the data (especially after the capping process). By comparing boxplots of different features, we could gain insights into the relative spread and potential relationships between variables.

Categorical Data Exploration

- **Count Plots:** Count plots were employed to visualize the frequency distribution of each category within categorical features like Day_of_Week, Time_of_Day, and Holiday_Indicator. These plots provided a clear understanding of the composition of these features and any potential imbalances that might need to be addressed during model development.

Relationship Analysis

- **Correlation Matrix:** A correlation matrix was generated to assess the linear relationships between various features in the dataset. Identifying features with high correlations could indicate potential redundancy or their influence on electricity load. This information helped us understand the feature interactions that might be relevant for building the forecasting model.
- **Scatter Plots:** To delve deeper into potential dependencies between features, scatter plots were created. These plots visualized the relationship between Electricity_Load and other features (e.g., Temperature, Time_of_Day). This granular exploration allowed us to identify specific patterns and relationships that might influence electricity demand.

Time Series Analysis

- **Time Series Plot:** The Electricity_Load variable was plotted over time to uncover patterns and trends in energy demand. This visualization provided a high-level overview of how electricity consumption fluctuates throughout the timeframe represented in the dataset.

- **Time Series Decomposition:** Time series decomposition was employed to separate the Electricity_Load data into its trend, seasonality, and residual components. This process helped us understand:
  - **Long-term trends:** By isolating the trend component, we could identify long-term increases or decreases in electricity demand.
  - **Seasonal variations:** Analyzing the seasonality component allowed us to understand how electricity load varies across different seasons (e.g., higher demand during summer months).
  - **Residuals:** Examining the residual component revealed any unexplained fluctuations in electricity load that might be due to factors not captured in the data.
- **Seasonal Analysis:** To gain a more comprehensive understanding of seasonal patterns in electricity load, we conducted the following analyses:
  - **Monthly Average Electricity Load:** We calculated the average electricity load for each month. This analysis helped identify peak demand periods throughout the year.
  - **Electricity Load Distribution by Month:** Visualizing the distribution of electricity load within each month revealed potential variations in consumption patterns even within the same season. This could be due to factors like weather fluctuations or changes in human activity levels.
- **Autocorrelation Analysis:** Autocorrelation analysis was performed to investigate the presence and patterns of dependence within the electricity load time series. This analysis helped us determine if past values of electricity load influence future values. Understanding the autocorrelation structure of the data is crucial for building effective forecasting models.
- **Trend Analysis:** The original electricity load time series was compared with a moving average (e.g., 30-day window) to analyse trends. This visualization helped us identify long-term trends in electricity demand, such as gradual increases or decreases over time.

The EDA process provided invaluable insights into the energy demand dataset. It revealed the distribution of values, potential relationships between features, and the presence of seasonality and trends in electricity load.

3.3.5 MACHINE LEARNING MODEL SELECTION

Time series forecasting involves predicting future values based on a sequence of past observations. In this project, the goal is to forecast future energy demand based on historical consumption data. Several machine learning models are suitable for time series forecasting. This project explores the following models:

- **Long Short-Term Memory (LSTM):** A deep learning model adept at capturing long-term dependencies within time series data like historical energy consumption patterns. It was suitable for learning complex relationships between features, potentially leading to accurate forecasts. LSTMs are a type of recurrent neural network (RNN) specifically designed to address the vanishing gradient problem that hinders traditional RNNs in learning long-term dependencies. LSTMs achieve this through a complex gating mechanism that controls information flow within the network.

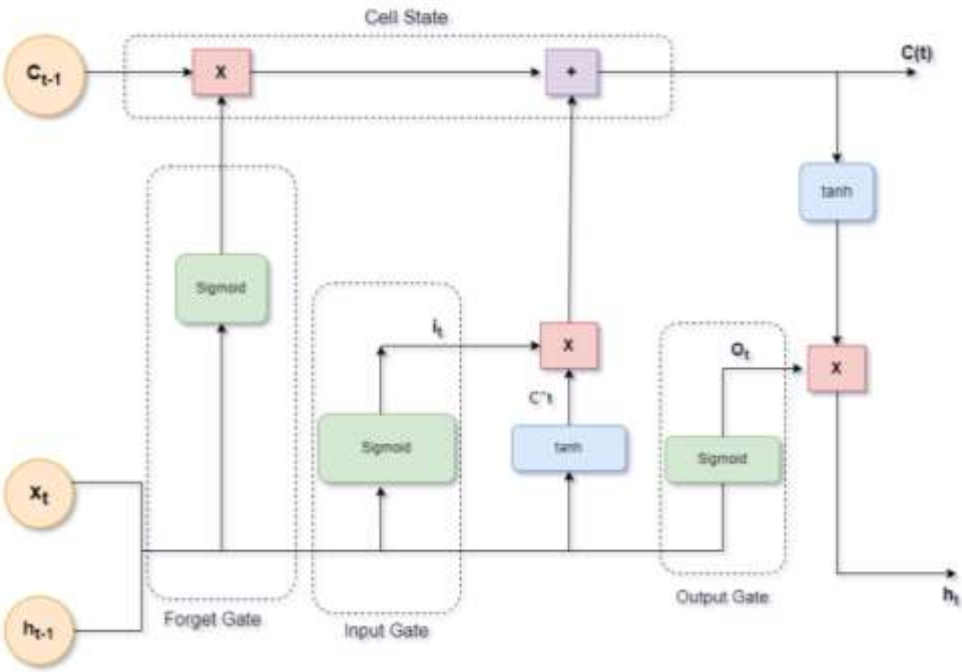


Figure **Error! No text of specified style in document.**.2 LSTM ARCHITECTURE

The cell state: Stores long-term information.

The forget gate: Decides what information to forget from the cell state.

The input gate: Decides what new information to add to the cell state.

The output gate: Decides what information from the cell state to output.

- **ARIMA (Autoregressive Integrated Moving Average):** A widely used statistical model for time series forecasting. It was effective in capturing trends and seasonality present in energy consumption data. ARIMA is a statistical model with three components:

- Autoregressive (AR): Explains the current value based on past values (p).
- Integrated (I): Makes the data stationary through differencing (d times).
- Moving Average (MA): Accounts for randomness by considering past errors (q).

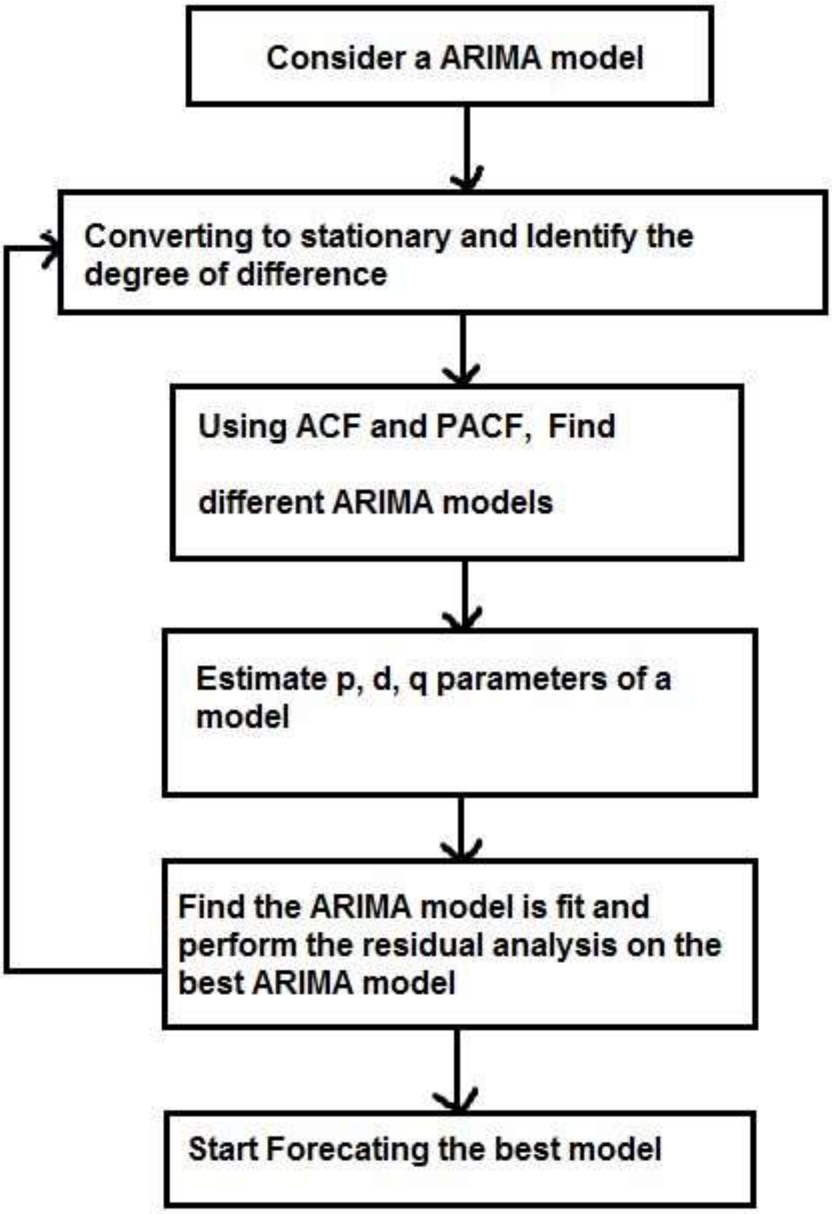


Figure **Error! No text of specified style in document.**.3 ARIMA MODEL ARCHITECTURE

The model takes past values (p) and past errors (q) as input. It uses these to predict the current value by considering both the autoregressive (AR) terms and the moving average (MA) terms.

- **Seasonal ARIMA (SARIMA):** An extension of ARIMA specifically designed to capture seasonal patterns, potentially crucial for capturing recurring fluctuations in energy demand. It considers AR, I, and MA components. Additionally, it includes seasonal AR (P) and seasonal MA (Q) terms with a seasonal period (S).

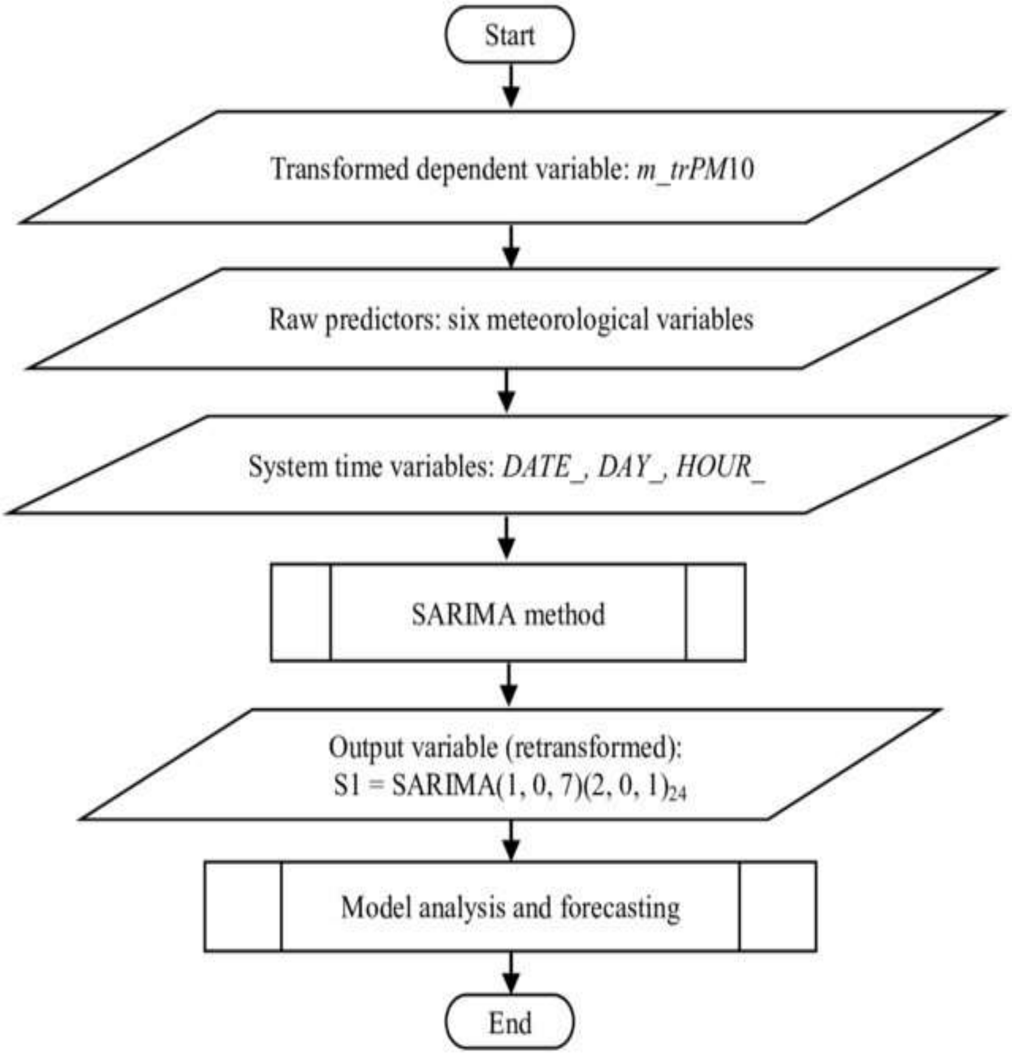


Figure **Error! No text of specified style in document.**.4 SARIMA MODEL ARCHITECTURE

- **Vector Autoregression (VAR):** This model analyses the relationships between multiple time series variables, making it valuable for studying the combined effect of weather data (temperature, humidity) and historical consumption on energy demand forecasting. The basic requirements in order to use VAR are: We need atleast two time series (variables). The time series should influence each other. The vector autoregressive VAR(p) model extends the AR(p) model to k series by creating a system of k equations where each contains p lagged values of all k series. Multivariate time series models allow for lagged values of other time series to affect the target. This effect applies to all series, resulting in complex interactions. In the VAR model, each variable is modelled as a linear combination of past values of itself and the past values of other variables in the system. Since you have multiple time series that influence each other, it is modelled as a system of equations with one equation per variable (time series). VAR(p) models also require stationarity.

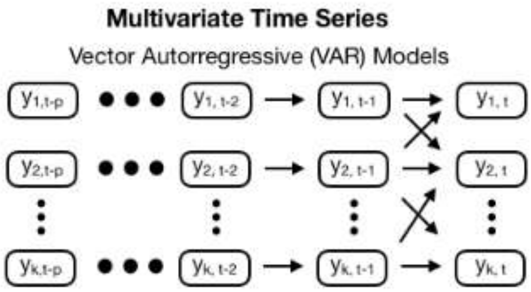


Figure **Error! No text of specified style in document.**.5 VAR MODEL ARCHITECTURE

- **Support Vector Regression (SVR):** A powerful technique for regression problems. It was explored for its ability to learn a hyperplane in the high-dimensional feature space that separates different energy demand values. SVR maps the input data (historical consumption, weather) into a high-dimensional feature space and then finds a hyperplane that separates different energy demand values with the largest margin.

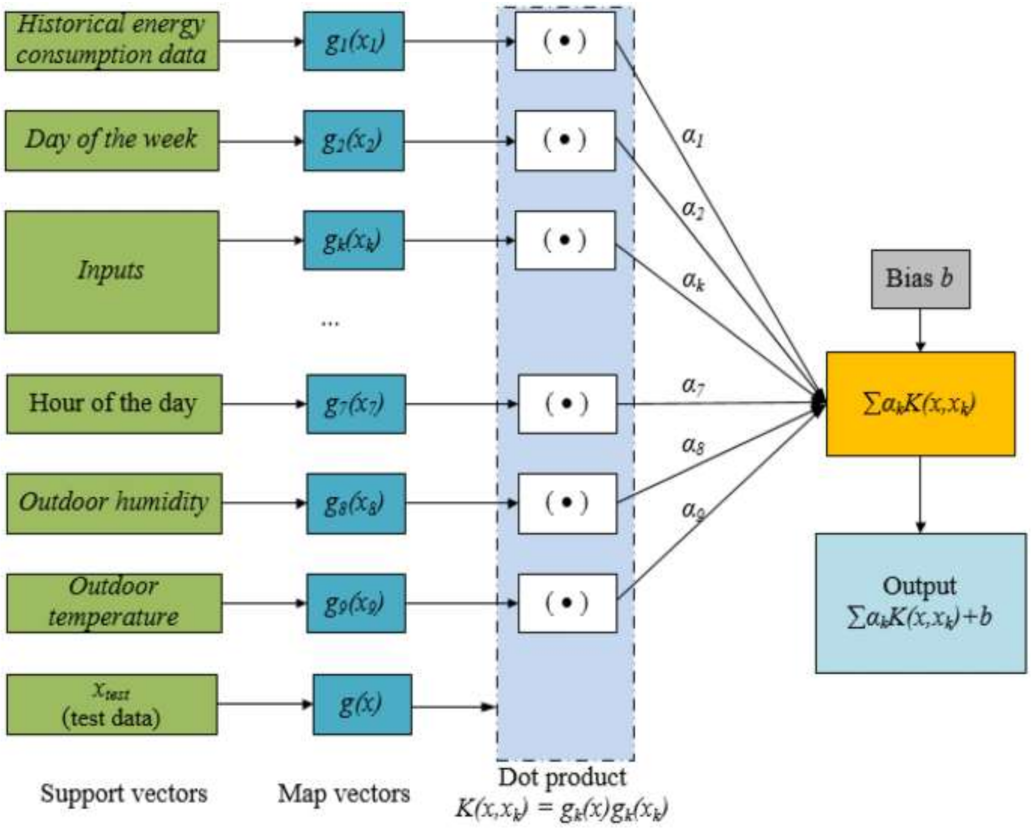


Figure **Error! No text of specified style in document.**.6 SVR MODEL ARCHITECTURE

- **Holt-Winters:** A traditional statistical method for exponential smoothing with trend and seasonality components. It uses weighted averages of past observations to make forecasts. The model considers the level (average value), trend (slope), and seasonality (cyclical patterns) components. It exponentially weights past observations with weights decaying over time, giving more importance to recent data.

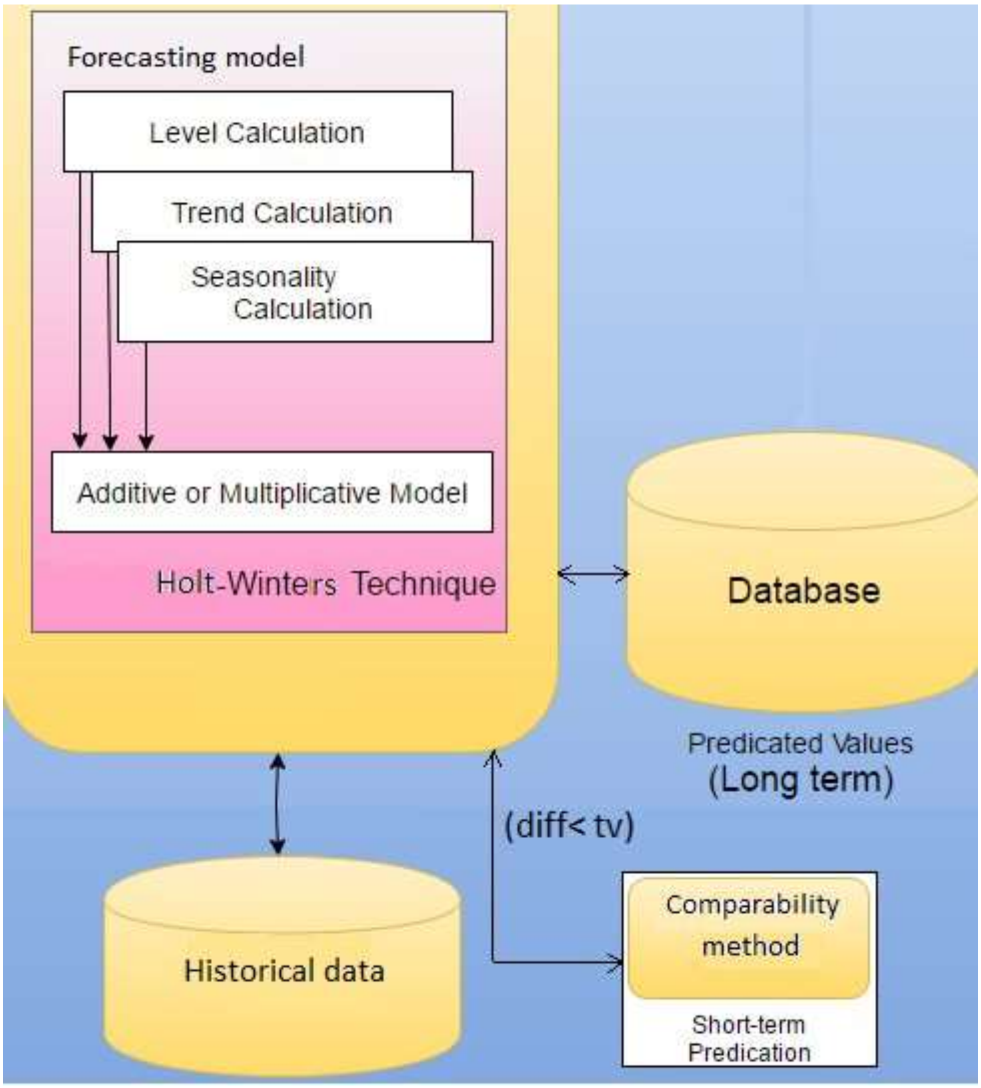


Figure **Error! No text of specified style in document.**.7 HOLT-WINTERS MODEL ARCHITECTURE

- **XGBoost (Extreme Gradient Boosting):** A powerful ensemble learning model known for its ability to handle complex relationships and potentially outperform traditional models. It was investigated for its potential to learn intricate patterns from historical consumption and weather data, leading to superior forecasting accuracy. XGBoost uses a technique called gradient boosting, where each new tree learns to improve upon the errors of the previous trees. This sequential learning leads to a more accurate ensemble model.

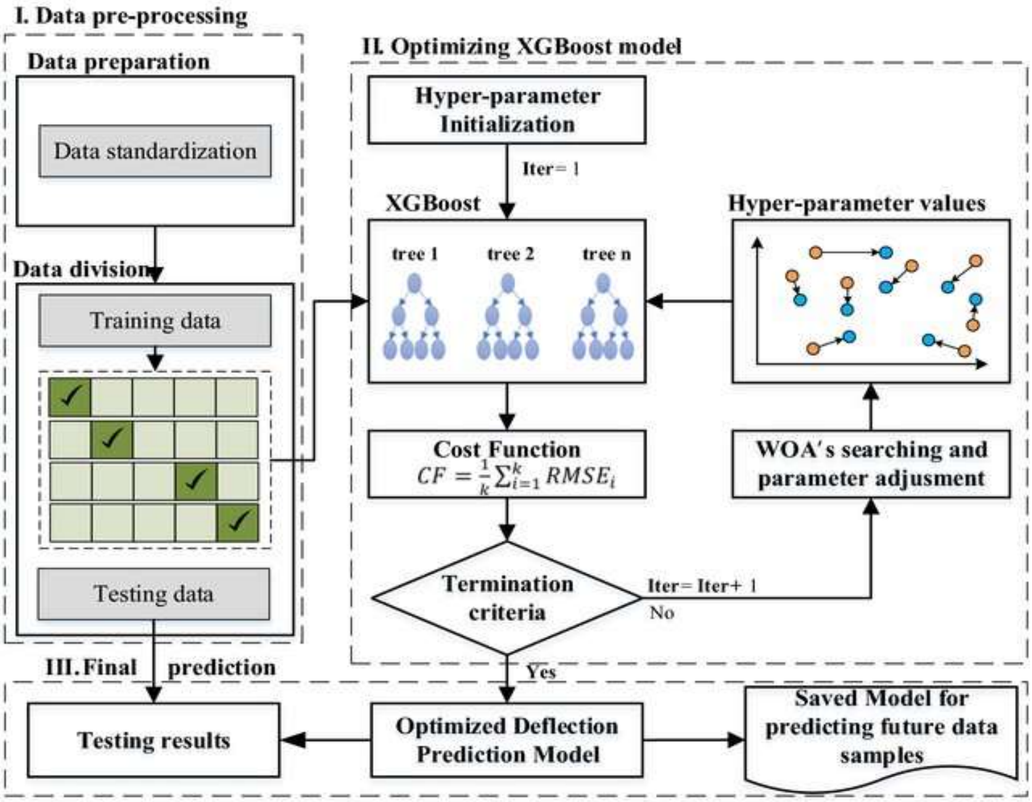


Figure **Error! No text of specified style in document.**.8 XGBOOST MODEL ARCHITECTURE

Table 3.2 Comparison of Machine Learning Algorithms

Feature	LSTM	ARIMA	SARIMA	VAR	SVR	Holt-Winters	XGBoost
Type	Deep Learning	Statistical	Statistical	Statistical	Statistical	Statistical	Ensemble Learning
Data Pre-processing	Often requires more pre-processing	Typically requires stationary data	May require additional seasonal data processing	May require data preparation for multiple variables	May require data pre-processing depending on complexity	Limited pre-processing required	May require data cleaning and feature engineering
Forecast Horizon	Suitable for short-term, medium-term, and long-term forecasting	Primarily for short-term and medium-term forecasting	Primarily for short-term and medium-term forecasting	Suitable for short-term and medium-term forecasting	Suitable for short-term and medium-term forecasting	Primarily for short-term forecasting	Suitable for short-term, medium-term, and long-term forecasting
Error Handling	Can struggle with outliers and data shifts	May require additional checks for stationar-ity	May require additional checks for seasonality	May be sensitive to outliers in multiple variables	May be sensitive to outliers	Limited error handling capabilities	Can handle outliers to some extent

Strengths	Captures complex non-linear relationship, Handles long-term dependency	Easy to interpret, Effective for trends & seasonality	Explicitly models seasonality, Interpret-able	Analyses relationships between multiple variables, Useful for weather data integration	Effective for linear/simple non-linear relationships, Good for moderate data size	Simple to implement, Provides good baseline	Powerful for complex relationships, often outperforms traditional models
Weakness	Requires significant data & computational resources, less interpretable	May struggle with highly complex data/outliers	Can become complex with many seasonal parameters	Computationally expensive with many variables, less interpretable relationships	May not handle highly complex non-linear patterns, Hyperparameter tuning required	May not capture complex patterns/sudden trend changes	Computationally expensive to train, less interpretable ensemble model
Scalability	Scales well with large datasets	Limited scalability with very large datasets	Limited scalability with very large datasets	Can become computationally expensive with many variables	Limited scalability with very large datasets	Scales well with large datasets	Scales well with large datasets
Explainability	Less explainable, Relies on feature importance analysis	Highly interpretable, explains forecast based on past values and errors	Interpretable, Explains influence of seasonal factors	Moderately interpretable, Explains relationships between variables	Limited explainability, Relies on hyperparameter settings	Easy to understand the reasoning behind the forecast	Limited explainability due to ensemble nature
Suitability	Complex energy demand forecasting with long-term patterns	Baseline forecasting, Energy demand with moderate trends & seasonality	Energy demand with recurring seasonal patterns	Energy demand influenced by multiple variables (weather data)	Energy demand with linear/simple non-linear relationships	Baseline comparison, Simple energy demand forecasting	Complex energy demand forecasting with intricate patterns

3.3.6 MODEL TRAINING

For each chosen machine learning model (LSTM, ARIMA, SARIMA, VAR, SVR, Holt-Winters, XGBoost), the following steps were undertaken:

Model training: The training data was used to fit the model parameters. This involved feeding the preprocessed features (including historical consumption, weather data, and engineered features) to the model for it to learn the underlying relationships between features and energy demand. The preprocessed data was divided into training and testing sets using a common split ratio (e.g., 80% training, 20% testing). The training set is used to train the machine learning models, while the testing set is used to evaluate their performance on unseen data. This helps assess how well the models generalize to new data and prevents overfitting.

Hyperparameter tuning: Hyperparameters are key settings that control the learning behavior of the models. Techniques like grid search or randomized search were employed to identify the optimal hyperparameter values for each model. This optimization process aimed to maximize the model's performance on the training data.

3.3.7 MODEL EVALUATION

The trained models will be evaluated on the unseen testing set using the chosen evaluation metrics (RMSE, MSE, MAE, MAPE, Explained Variance, R-squared, Maximum Error). The model with the best overall performance based on these metrics will be considered the most suitable for forecasting energy demand in this specific project.

1. Root Mean Squared Error (RMSE):

- **Formula:** RMSE = sqrt((1/n) * Σ (yi - ŷi)^2)
    - n: Number of data points
    - yi: Actual value
    - ŷi: Predicted value

RMSE measures the average magnitude of the difference between predicted and actual values. Lower RMSE indicates better model performance, as it signifies a smaller average error. Units are the same as the units of the predicted values.

2. Mean Squared Error (MSE):

- **Formula:** MSE = (1/n) * Σ (yi - ŷi)^2

MSE is similar to RMSE, but it squares the errors before averaging. Squaring the errors emphasizes larger errors more heavily compared to RMSE. Units are the square of the units of the predicted values.

3. Mean Absolute Error (MAE):

- **Formula:** MAE = (1/n) * Σ |yi - ŷi|

MAE measures the average absolute difference between predicted and actual values. It's less sensitive to outliers compared to RMSE and MSE. Units are the same as the units of the predicted values.

4. Mean Absolute Percentage Error (MAPE):

- **Formula:** MAPE = (1/n) * Σ |(yi - ŷi) / yi| * 100%

MAPE expresses the error as a percentage of the actual value. It's useful for comparing errors across different datasets or models, especially when dealing with data containing values of varying scales. However, MAPE can be problematic for zero or very small actual values (division by zero).

5. Explained Variance:

- **Formula:** Explained Variance = 1 - (Σ (yi - ŷi)^2) / (Σ (yi - ȳ)^2)
  - ȳ: Mean of the actual values

Explained Variance represents the proportion of variance in the actual data that is explained by the model. It ranges from 0 (no explanation) to 1 (perfect explanation). A higher Explained Variance indicates a better fit between the model and the data.

6. R-squared (Coefficient of Determination):

- **Formula:** R-squared = Explained Variance = 1 - (Σ (yi - ŷi)^2) / (Σ (yi - ȳ)^2)

R-squared is mathematically equivalent to Explained Variance. It's a widely used metric for assessing the goodness of fit of a linear regression model. Higher R-squared values indicate a better fit. However, R-squared doesn't necessarily translate to good forecasting performance, especially for non-linear models.

7. Maximum Error (ME):

- **Formula:** ME = max |yi - ŷi|

Maximum Error simply refers to the largest absolute difference between a predicted value and the corresponding actual value in the dataset. It highlights the worst-case scenario for the model's prediction errors. Units are the same as the units of the predicted values.

CHAPTER 4
IMPLEMENTATION DETAILS

IMPORTING NECESSARY LIBRARIES

In [1]:
```
import pandas as pd
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns
from tabulate import tabulate
import xgboost as xgb
from xgboost import XGBRegressor
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.api import VAR
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.svm import SVR
from sklearn.preprocessing import MinMaxScaler, StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score,
explained_variance_score, max_error

from math import sqrt
from tqdm import tqdm
```

LOADING THE DATASET

In [2]:
```
data = pd.read_csv('iot_load_data.csv')
```

VIEWING THE DATASET

To display the contents of the Dataset by accessing the pandas Dataframe named data.

In [3]:
```
data
```

Out[3]:

	Timestamp	Electricity_Load	Temperature	Humidity	Day_of_Week	Time_of_Day
0	1/1/2015	753	2	69	3	Afternoon
1	1/2/2015	872	14	26	4	Evening
2	1/3/2015	525	19	73	5	Afternoon
3	1/4/2015	568	23	59	6	Morning
4	1/5/2015	636	28	32	0	Afternoon
...	...	...	...	...	...	...
3312	1/26/2024	626	7	78	4	Night
3313	1/27/2024	660	28	76	5	Afternoon
3314	1/28/2024	902	-6	71	6	Morning
3315	1/29/2024	805	8	50	0	Morning
3316	1/30/2024	510	-2	69	1	Night

3317 rows × 23 columns

data.describe() summarizes our data in `data`. It provides statistics like mean, standard deviation for numerical columns, helping us to understand data spread.

In [4]:
```
data.describe()
```

Out[4]:

	Electricity_Load	Temperature	Humidity	Day_of_Week	Holiday_Indicator	Pre
count	3317.000000	3317.000000	3317.000000	3317.000000	3317.000000	3
mean	747.139584	11.826349	49.235454	3.000301	0.048538	
std	146.200936	13.148406	17.240328	2.000528	0.214932	
min	500.000000	-10.000000	20.000000	0.000000	0.000000	
25%	617.000000	0.000000	34.000000	1.000000	0.000000	
50%	745.000000	12.000000	49.000000	3.000000	0.000000	
75%	876.000000	23.000000	64.000000	5.000000	0.000000	
max	999.000000	34.000000	79.000000	6.000000	1.000000	

data.info() in Pandas gives us a quick peek at our data's structure. It shows details like number of

rows, columns, data types, memory usage and missing values.

```
In [5]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3317 entries, 0 to 3316
Data columns (total 23 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Timestamp             3317 non-null   object 
 1   Electricity_Load      3317 non-null   int64  
 2   Temperature           3317 non-null   int64  
 3   Humidity              3317 non-null   int64  
 4   Day_of_Week           3317 non-null   int64  
 5   Time_of_Day           3317 non-null   object 
 6   Holiday_Indicator     3317 non-null   int64  
 7   Previous_Load         3317 non-null   int64  
 8   Transportation_Data   3317 non-null   int64  
 9   Operational_Metrics   3317 non-null   int64  
 10  IoT_Sensor_Data       3317 non-null   float64
 11  External_Factors      3317 non-null   object 
 12  Day_Ahead_Demand      3317 non-null   int64  
 13  Real_Time_LMP         3317 non-null   float64
 14  Regulation_Capacity   3317 non-null   int64  
 15  Day_Ahead_LMP         3317 non-null   float64
 16  Day_Ahead_EC          3317 non-null   float64
 17  Day_Ahead_CC          3317 non-null   float64
 18  Day_Ahead_MLC         3317 non-null   float64
 19  Real_Time_EC          3317 non-null   float64
 20  Real_Time_CC          3317 non-null   float64
 21  Real_Time_MLC         3317 non-null   float64
 22  System_Load           3317 non-null   int64  
dtypes: float64(9), int64(11), object(3)
memory usage: 596.2+ KB
```

DATA PREPROCESSING

To determine the preprocessing and data cleaning steps required for the dataset in `iot_load_data.csv`, We first inspected the data to identify any potential issues such as missing values, incorrect data types, outliers, or inconsistencies.

Step 1: Check for missing values in each column and removing them if any

```
In [6]: data_missing_values = data.isnull().sum()
        print('\Missing Values in Each Column:')
        print(data_missing_values)
```

```
\Missing Values in Each Column:
Timestamp                 0
Electricity_Load          0
Temperature               0
Humidity                  0
Day_of_Week               0
Time_of_Day               0
Holiday_Indicator         0
Previous_Load             0
Transportation_Data       0
Operational_Metrics       0
IoT_Sensor_Data           0
External_Factors          0
Day_Ahead_Demand          0
Real_Time_LMP             0
Regulation_Capacity       0
Day_Ahead_LMP             0
Day_Ahead_EC              0
Day_Ahead_CC              0
Day_Ahead_MLC             0
Real_Time_EC              0
Real_Time_CC              0
Real_Time_MLC             0
System_Load               0
dtype: int64
```

In [7]:

```
data_missing_values = data_missing_values[data_missing_values > 0]

if not data_missing_values.empty:
    for column in data_missing_values.index:
        if data[column].dtype == 'object':
            data[column] = data[column].fillna(data[column].mode()[0])
        else:
            data[column] = data[column].fillna(data[column].median())
    print('Missing values have been handled.')
else:
    print('No missing values to handle.')
```

```
No missing values to handle.
```

Step 2: The 'Timestamp' column is recognized as an object type, which suggests it is stored as a string. This has been converted to a datetime type for our forecasting model and checked we whether it has been updated.

In [8]:

```
data['Timestamp'] = pd.to_datetime(data['Timestamp'])
data['Day'] = data['Timestamp'].dt.day
data['Month'] = data['Timestamp'].dt.month
data['Year'] = data['Timestamp'].dt.year

# Confirm the conversion
data_types = data.dtypes
print('Updated Data Types:')
print(data_types)
```

```
Updated Data Types:
Timestamp                  datetime64[ns]
Electricity_Load                    int64
Temperature                         int64
Humidity                            int64
Day_of_Week                         int64
Time_of_Day                        object
Holiday_Indicator                   int64
Previous_Load                       int64
Transportation_Data                 int64
Operational_Metrics                 int64
IoT_Sensor_Data                   float64
External_Factors                   object
Day_Ahead_Demand                    int64
Real_Time_LMP                     float64
Regulation_Capacity                 int64
Day_Ahead_LMP                     float64
Day_Ahead_EC                      float64
Day_Ahead_CC                      float64
Day_Ahead_MLC                     float64
Real_Time_EC                      float64
Real_Time_CC                      float64
Real_Time_MLC                     float64
System_Load                         int64
Day                                 int32
Month                               int32
Year                                int32
dtype: object
```

Step 3: Checking for outliers in numerical columns to ensure data quality and we applied Capping outliers; it is a technique used in data analysis to address extreme values that fall outside the typical range of the data. It involves setting thresholds and replacing outlier values with those thresholds.

In [9]:

```python
# Check for outliers using the IQR method
def detect_outliers(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
    return outliers

# Apply outlier detection to a few numerical columns
outliers_electricity = detect_outliers(data, 'Electricity_Load')
outliers_temperature = detect_outliers(data, 'Temperature')
outliers_humidity = detect_outliers(data, 'Humidity')

print('Outliers in Electricity Load:')
print(outliers_electricity)
print('\nOutliers in Temperature:')
print(outliers_temperature)
print('\nOutliers in Humidity:')
print(outliers_humidity)
```

```
Outliers in Electricity Load:
Empty DataFrame
Columns: [Timestamp, Electricity_Load, Temperature, Humidity, Day_of_Week, Time_o
f_Day, Holiday_Indicator, Previous_Load, Transportation_Data, Operational_Metric
s, IoT_Sensor_Data, External_Factors, Day_Ahead_Demand, Real_Time_LMP, Regulation
_Capacity, Day_Ahead_LMP, Day_Ahead_EC, Day_Ahead_CC, Day_Ahead_MLC, Real_Time_E
C, Real_Time_CC, Real_Time_MLC, System_Load, Day, Month, Year]
Index: []

[0 rows x 26 columns]

Outliers in Temperature:
Empty DataFrame
Columns: [Timestamp, Electricity_Load, Temperature, Humidity, Day_of_Week, Time_o
f_Day, Holiday_Indicator, Previous_Load, Transportation_Data, Operational_Metric
s, IoT_Sensor_Data, External_Factors, Day_Ahead_Demand, Real_Time_LMP, Regulation
_Capacity, Day_Ahead_LMP, Day_Ahead_EC, Day_Ahead_CC, Day_Ahead_MLC, Real_Time_E
C, Real_Time_CC, Real_Time_MLC, System_Load, Day, Month, Year]
Index: []

[0 rows x 26 columns]

Outliers in Humidity:
Empty DataFrame
Columns: [Timestamp, Electricity_Load, Temperature, Humidity, Day_of_Week, Time_o
f_Day, Holiday_Indicator, Previous_Load, Transportation_Data, Operational_Metric
s, IoT_Sensor_Data, External_Factors, Day_Ahead_Demand, Real_Time_LMP, Regulation
_Capacity, Day_Ahead_LMP, Day_Ahead_EC, Day_Ahead_CC, Day_Ahead_MLC, Real_Time_E
C, Real_Time_CC, Real_Time_MLC, System_Load, Day, Month, Year]
Index: []

[0 rows x 26 columns]
```

In [10]:

```
# Handling outliers by capping them to the upper and lower bounds
def cap_outliers(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    data[column] = data[column].clip(lower=lower_bound, upper=upper_bound)
    return data

data = cap_outliers(data, 'Electricity_Load')
data = cap_outliers(data, 'Temperature')
data = cap_outliers(data, 'Humidity')

print('Outliers have been capped for Electricity Load, Temperature, and
Humidity.’)
```

```
Outliers have been capped for Electricity Load, Temperature, and Humidity.
```

Step 4: Standardizing categorical data by ensuring consistent casing and removing any leading/trailing spaces

In [11]:

```
data['Time_of_Day'] = data['Time_of_Day'].str.strip().str.lower()
data['External_Factors'] = data['External_Factors'].str.strip().str.lower()

unique_time_of_day = data['Time_of_Day'].unique()
unique_external_factors = data['External_Factors'].unique()
```

```
print('Unique Time of Day categories after standardization:')
print(unique_time_of_day)
print('\nUnique External Factors categories after standardization:')
print(unique_external_factors)
```

```
Unique Time of Day categories after standardization:
['afternoon' 'evening' 'morning' 'night']

Unique External Factors categories after standardization:
['regulatory' 'other' 'economic']
```

In [12]:
```
le = LabelEncoder()

data['Time_of_Day_encoded'] = le.fit_transform(data['Time_of_Day'])
data['External_Factors_encoded'] = le.fit_transform(data['External_Factors'])
```

In [13]:
```
data
```

Out[13]:

	Timestamp	Electricity_Load	Temperature	Humidity	Day_of_Week	Time_of_Day
0	2015-01-01	753	2	69	3	afternoon
1	2015-01-02	872	14	26	4	evening
2	2015-01-03	525	19	73	5	afternoon
3	2015-01-04	568	23	59	6	morning
4	2015-01-05	636	28	32	0	afternoon
...	...	...	...	...	...	...
3312	2024-01-26	626	7	78	4	night
3313	2024-01-27	660	28	76	5	afternoon
3314	2024-01-28	902	-6	71	6	morning
3315	2024-01-29	805	8	50	0	morning
3316	2024-01-30	510	-2	69	1	night

3317 rows × 28 columns

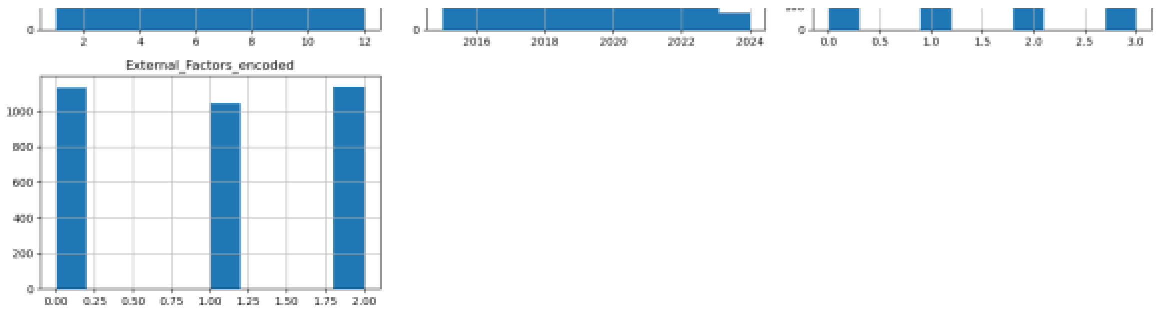
Performing exploratory data analysis (EDA)

Histograms of Numerical Columns: This visualization shows the distribution of values in the numerical columns

In [14]:
```
numerical_cols = [col for col in data.columns if pd.api.types.is_numeric_dtype(d
ata[col])]
fig, axes = plt.subplots(nrows=int((len(numerical_cols) - 1) / 3) + 1, ncols=3,
figsize=(15, 30))
```

```python
for i, col in enumerate(numerical_cols):
    ax = axes.flat[i]
    data[col].hist(ax=ax)
    ax.set_title(col)
    ax.grid(True)

for ax in axes.flat[len(numerical_cols):]:
    ax.axis('off')

# Show the plot
plt.tight_layout()
plt.show()
```


Electricity_Load
Temperature
Humidity
Day_of_Week
Holiday_Indicator
Previous_Load
Transportation_Data
Operational_Metrics
IoT_Sensor_Data
Day_Ahead_Demand
Real_Time_LMP
Regulation_Capacity
Day_Ahead_LMP
Day_Ahead_EC
Day_Ahead_CC
Day_Ahead_MLC
Real_Time_EC
Real_Time_CC
Real_Time_MLC
System_Load
Day
Month
Year
Time_of_Day_encoded

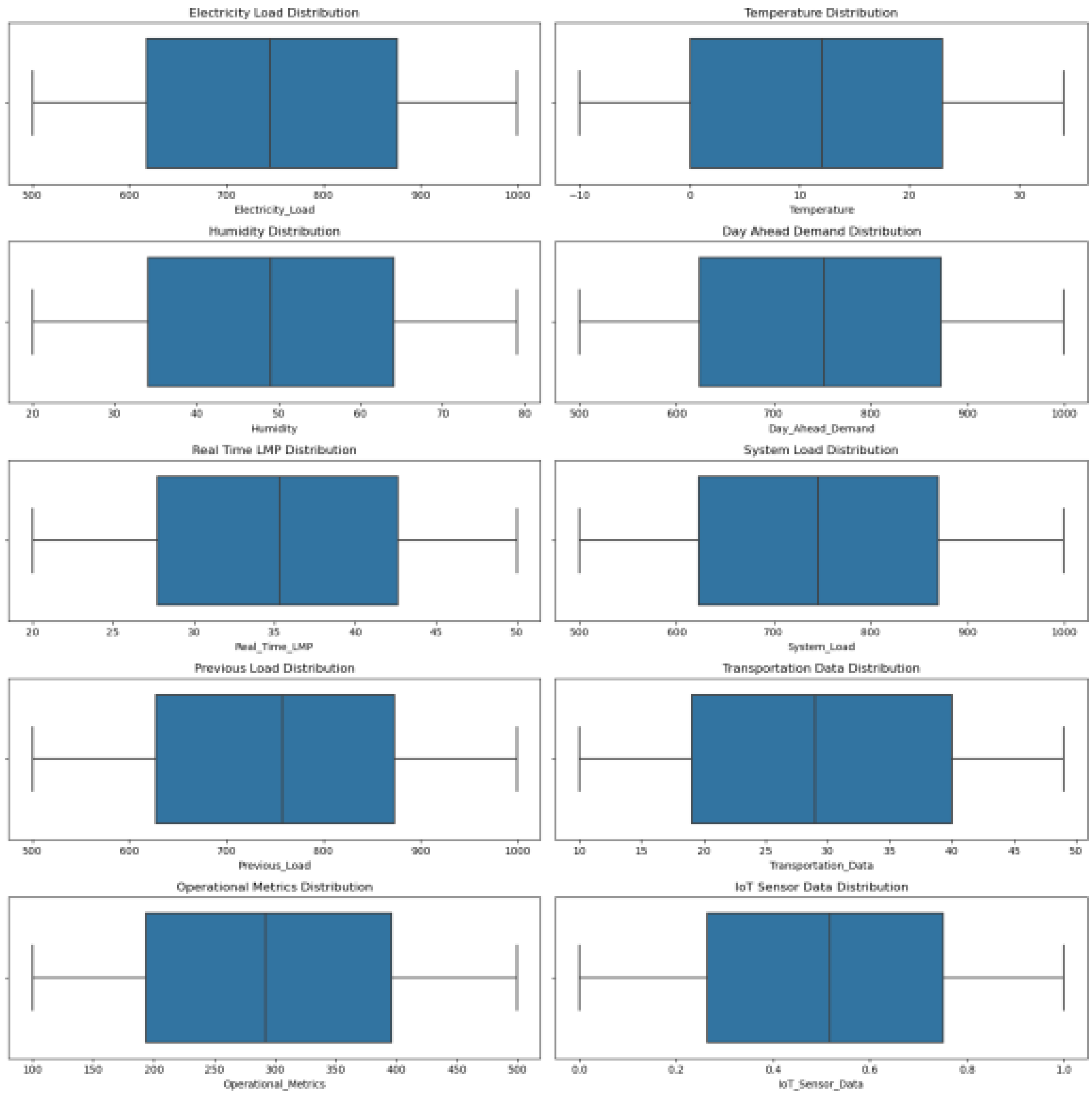


Boxplots of Numerical Columns of key variables: These boxplots provide a view of the central tendency and spread of the numerical data, along with any remaining outliers after capping.

In [15]:

```
plt.figure(facecolor='white')

fig, axes = plt.subplots(5, 2, figsize=(15, 15))

sns.boxplot(ax=axes[0, 0], x=data['Electricity_Load'])
axes[0, 0].set_title('Electricity Load Distribution')

sns.boxplot(ax=axes[0, 1], x=data['Temperature'])
axes[0, 1].set_title('Temperature Distribution')

sns.boxplot(ax=axes[1, 0], x=data['Humidity'])
axes[1, 0].set_title('Humidity Distribution')

sns.boxplot(ax=axes[1, 1], x=data['Day_Ahead_Demand'])
axes[1, 1].set_title('Day Ahead Demand Distribution')

sns.boxplot(ax=axes[2, 0], x=data['Real_Time_LMP'])
axes[2, 0].set_title('Real Time LMP Distribution')

sns.boxplot(ax=axes[2, 1], x=data['System_Load'])
axes[2, 1].set_title('System Load Distribution')

sns.boxplot(ax=axes[3, 0], x=data['Previous_Load'])
axes[3, 0].set_title('Previous Load Distribution')

sns.boxplot(ax=axes[3, 1], x=data['Transportation_Data'])
axes[3, 1].set_title('Transportation Data Distribution')

sns.boxplot(ax=axes[4, 0], x=data['Operational_Metrics'])
axes[4, 0].set_title('Operational Metrics Distribution')

sns.boxplot(ax=axes[4, 1], x=data['IoT_Sensor_Data'])
axes[4, 1].set_title('IoT Sensor Data Distribution')

plt.tight_layout()
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```

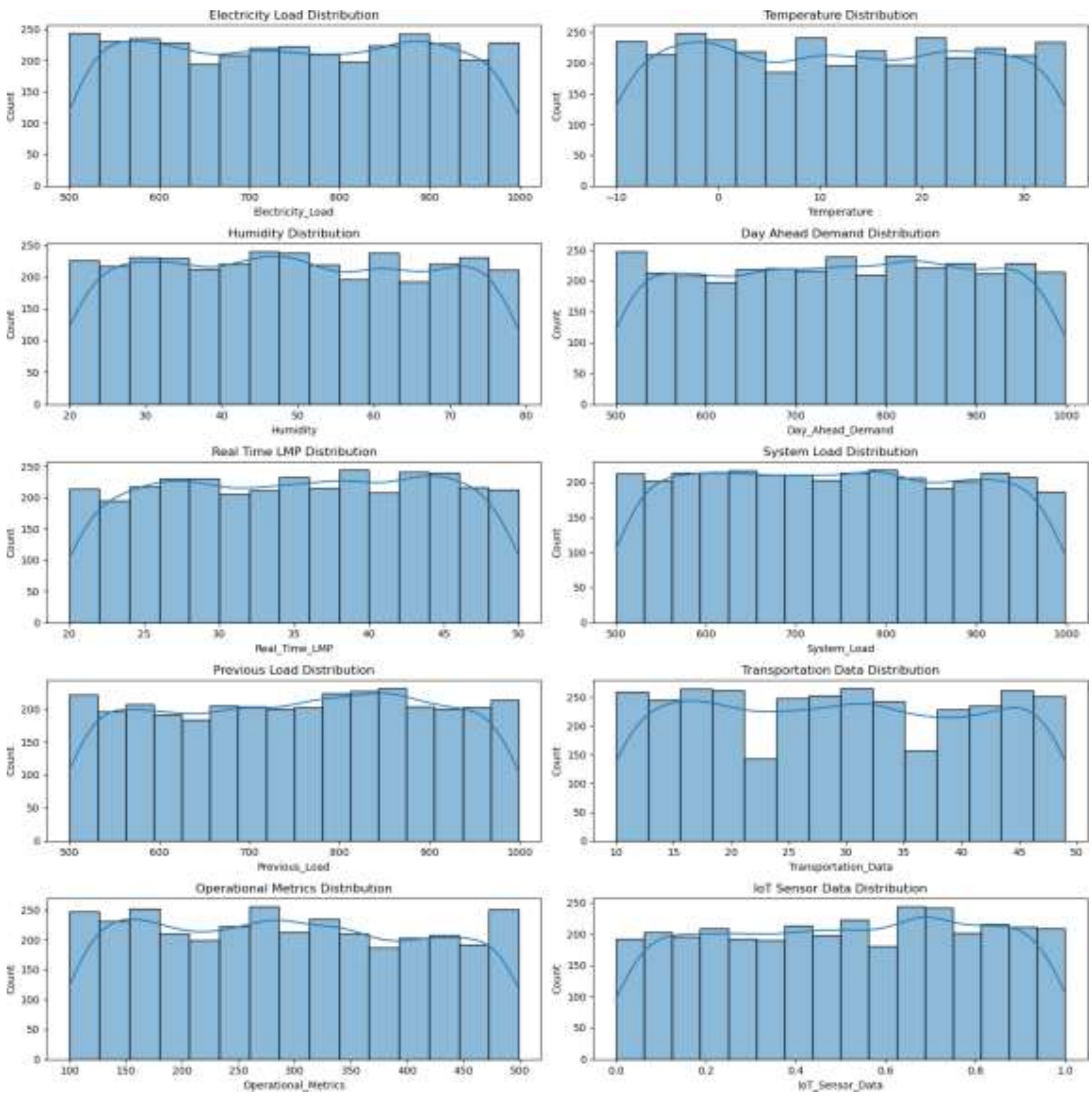


Visualizing the distribution of key variables to understand their spread and central tendency better using Histograms.

In [16]:

```
plt.figure(facecolor='white')

fig, axes = plt.subplots(5, 2, figsize=(15, 15))

sns.histplot(ax=axes[0, 0], x=data['Electricity_Load'], kde=True)
axes[0, 0].set_title('Electricity Load Distribution')

sns.histplot(ax=axes[0, 1], x=data['Temperature'], kde=True)
axes[0, 1].set_title('Temperature Distribution')

sns.histplot(ax=axes[1, 0], x=data['Humidity'], kde=True)
axes[1, 0].set_title('Humidity Distribution')

sns.histplot(ax=axes[1, 1], x=data['Day_Ahead_Demand'], kde=True)
axes[1, 1].set_title('Day Ahead Demand Distribution')

sns.histplot(ax=axes[2, 0], x=data['Real_Time_LMP'], kde=True)
axes[2, 0].set_title('Real Time LMP Distribution')
```

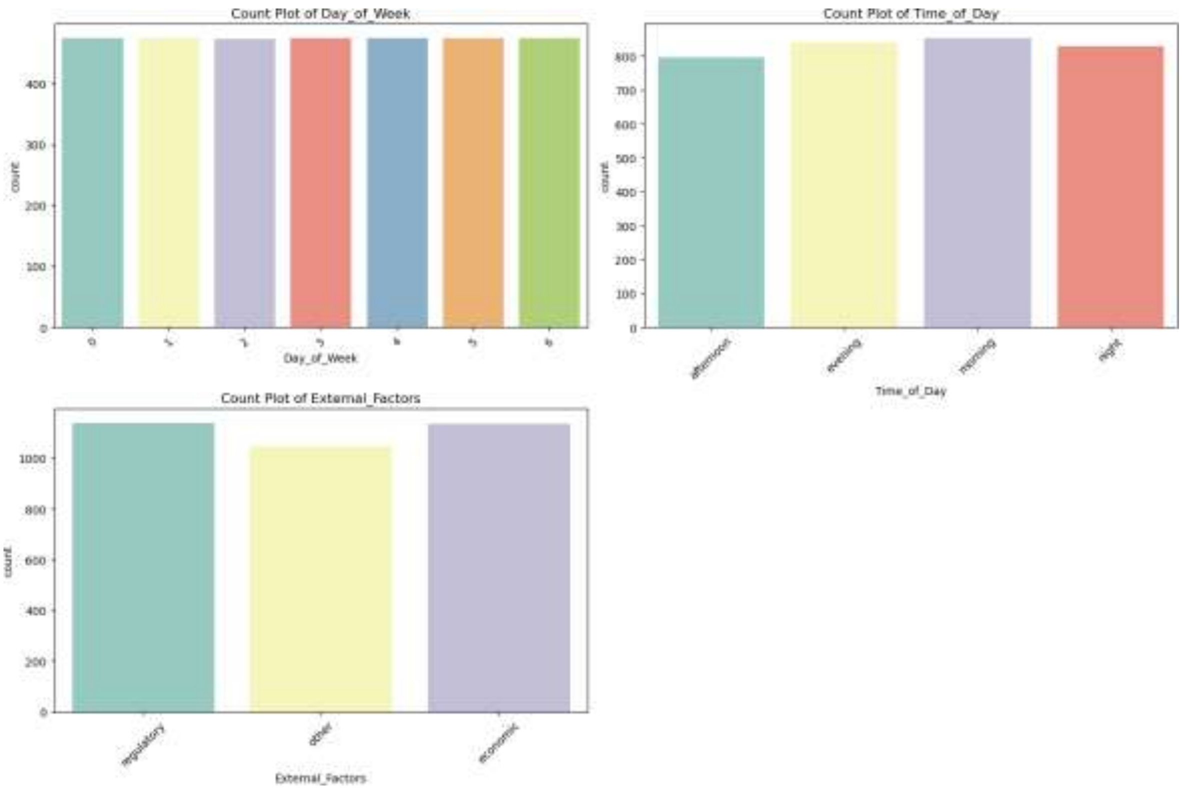
```
sns.histplot(ax=axes[2, 1], x=data['System_Load'], kde=True)
axes[2, 1].set_title('System Load Distribution')

sns.histplot(ax=axes[3, 0], x=data['Previous_Load'], kde=True)
axes[3, 0].set_title('Previous Load Distribution')

sns.histplot(ax=axes[3, 1], x=data['Transportation_Data'], kde=True)
axes[3, 1].set_title('Transportation Data Distribution')

sns.histplot(ax=axes[4, 0], x=data['Operational_Metrics'], kde=True)
axes[4, 0].set_title('Operational Metrics Distribution')

sns.histplot(ax=axes[4, 1], x=data['IoT_Sensor_Data'], kde=True)
axes[4, 1].set_title('IoT Sensor Data Distribution')

plt.tight_layout()
plt.show()
```



Count plots for categorical columns

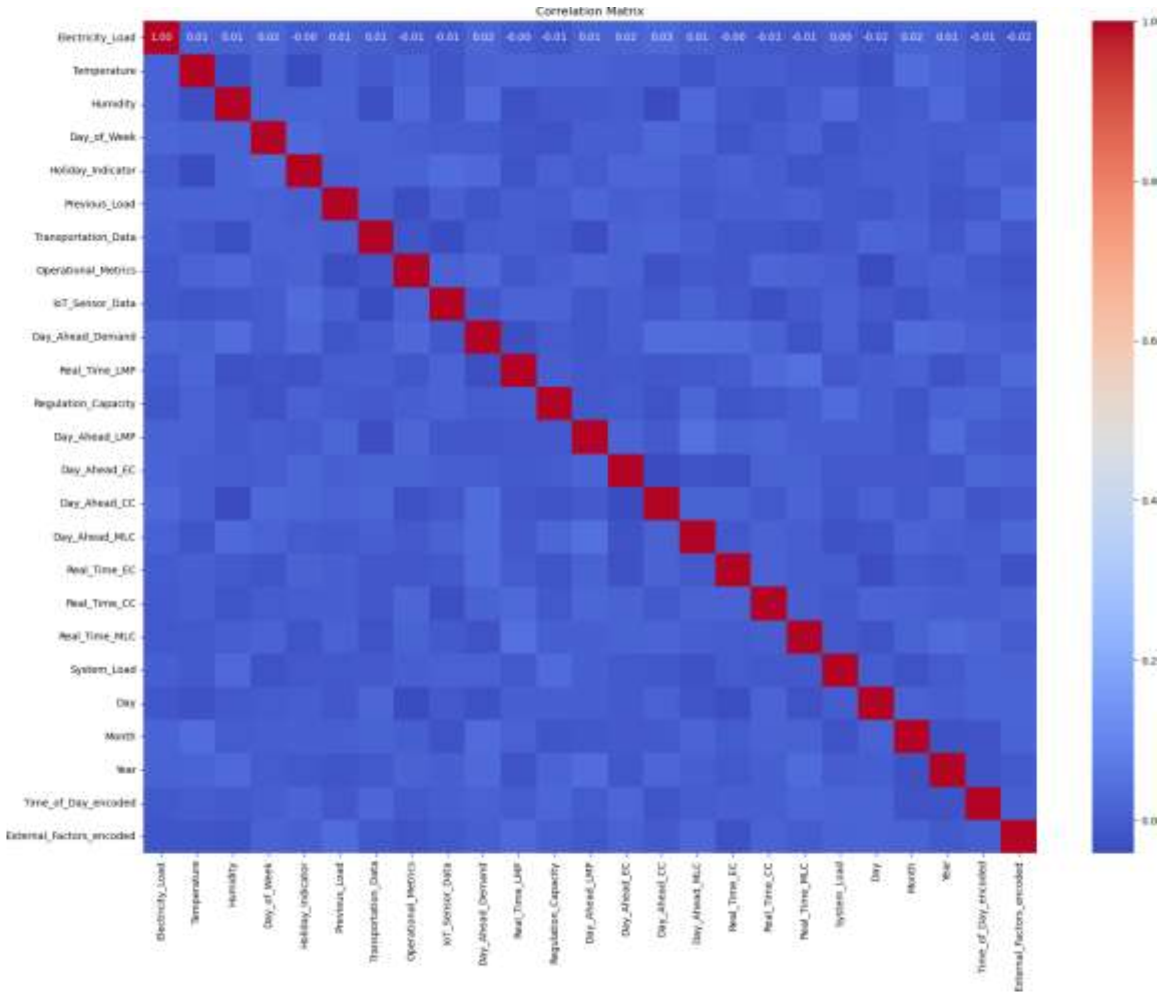
In [17]:
```python
plt.figure(facecolor='white')

categorical_columns = ['Day_of_Week', 'Time_of_Day', 'External_Factors']
plt.figure(figsize=(15, 10), facecolor='white')
for i, column in enumerate(categorical_columns):
    plt.subplot(2, 2, i+1)
    sns.countplot(x=column, data=data, palette='Set3')
    plt.title('Count Plot of ' + column)
    plt.xticks(rotation=45)

plt.tight_layout()
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



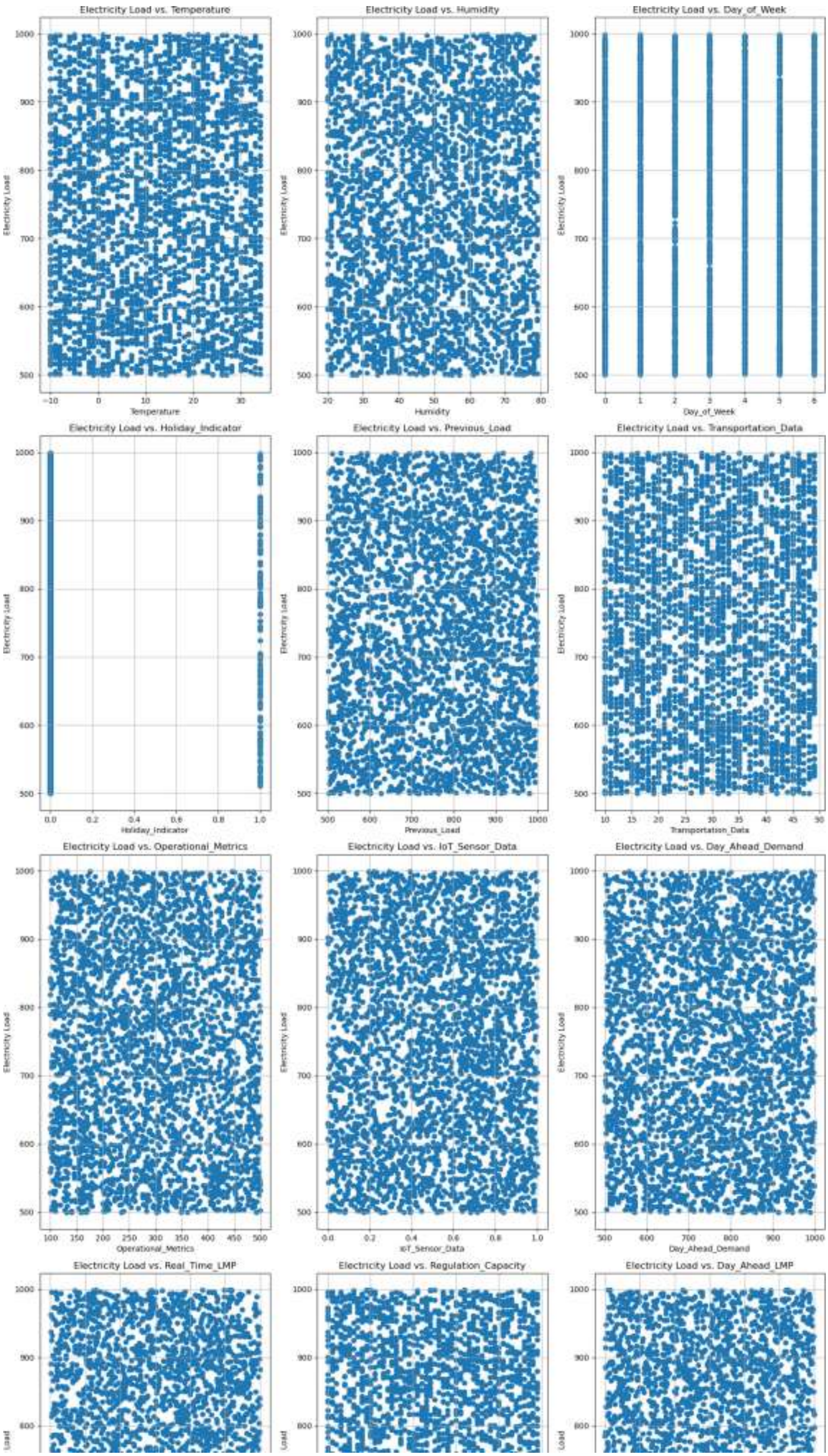
Analyzing the correlations between these variables to understand their relationships

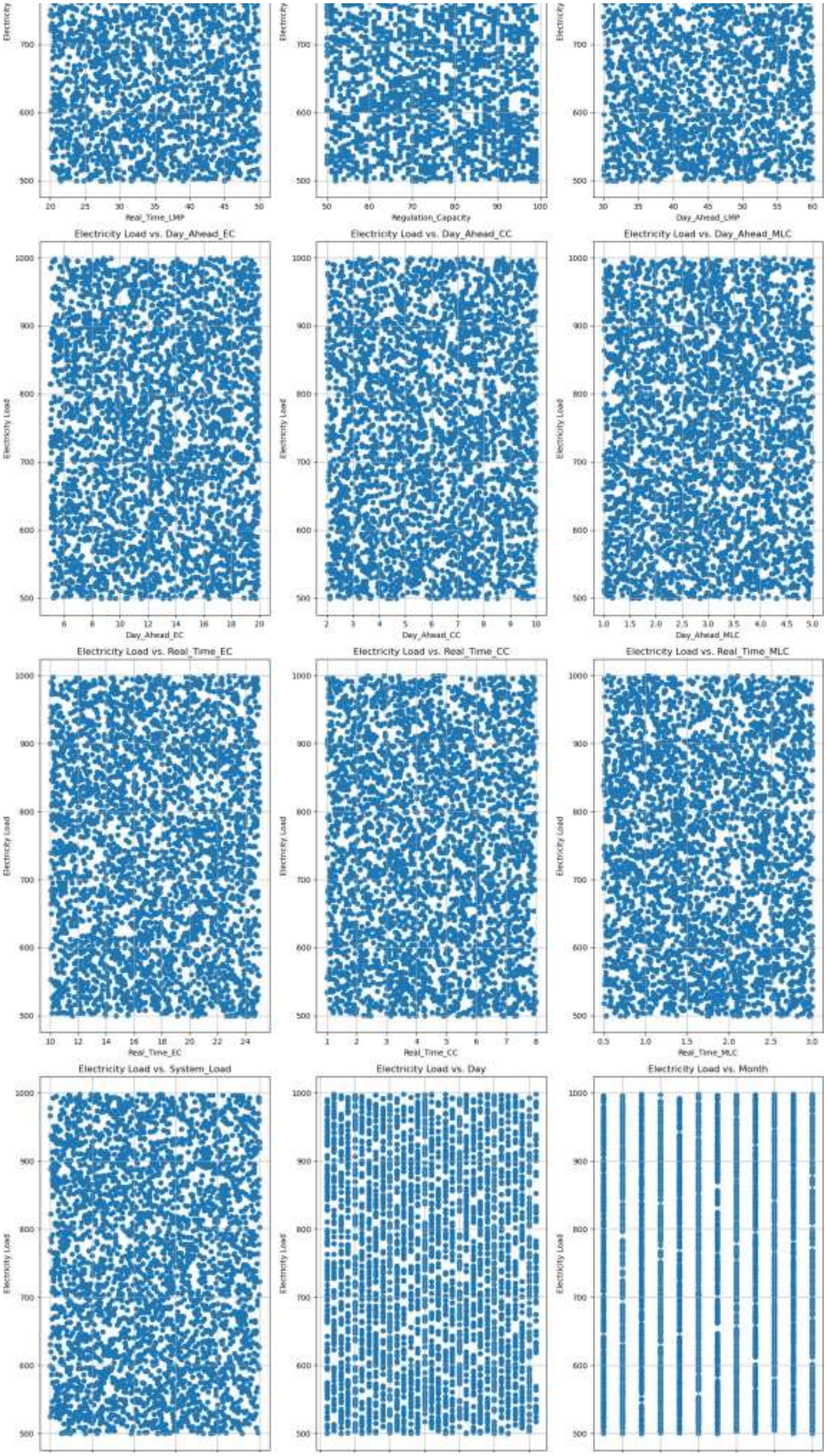
In [18]:
```
numeric_data = data.select_dtypes(include=[np.number])

plt.figure(figsize=(20, 15))
corr_matrix = numeric_data.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```



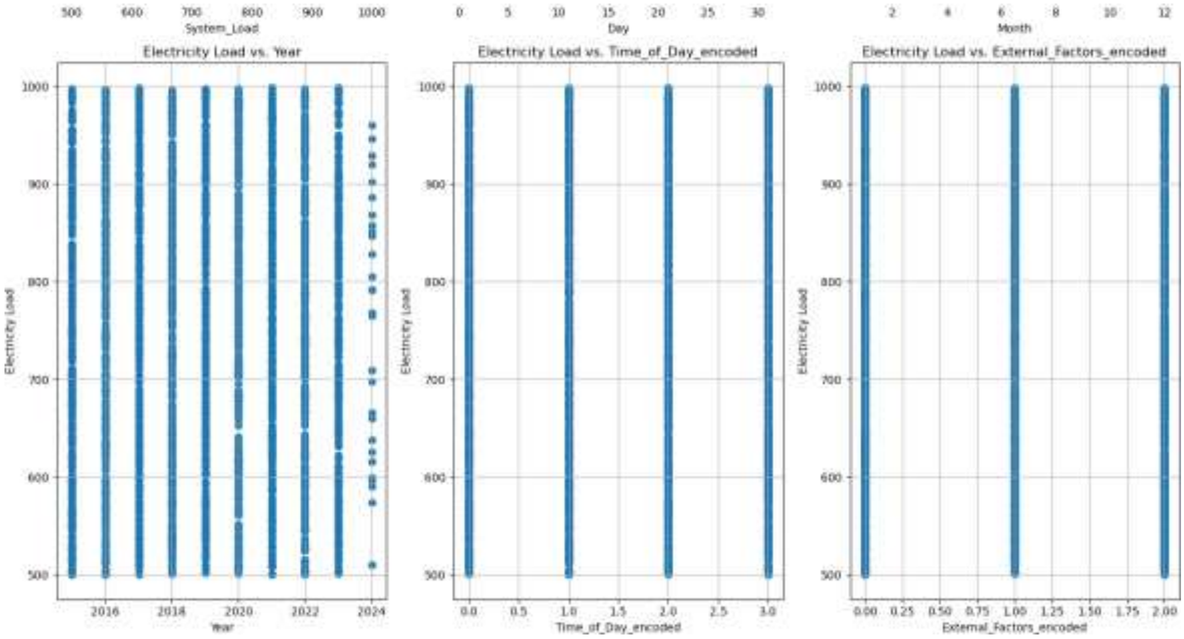
Scatter plot: To analyze the relationship between Electricity Load with other columns in the dataset

In [19]:
```
numerical_cols = [col for col in data.columns if col != 'Electricity_Load' and
pd.api.types.is_numeric_dtype(data[col])]

fig, axes = plt.subplots(nrows=int((len(numerical_cols) - 1) / 3) + 1, ncols=3,
figsize=(15, 60))

for i, col in enumerate(numerical_cols):
    ax = axes.flat[i]
    ax.scatter(data[col], data['Electricity_Load'])
    ax.set_xlabel(col)
    ax.set_ylabel('Electricity  Load')
    ax.set_title('Electricity Load vs. {}'.format(col))
    ax.grid(True)

for ax in axes.flat[len(numerical_cols):]:
    ax.axis('off')

plt.tight_layout()
plt.show()
```


Electricity Load vs. Temperature
Electricity Load vs. Humidity
Electricity Load vs. Day_of_Week
Electricity Load
Temperature
Humidity
Day_of_Week
Electricity Load vs. Holiday_Indicator
Electricity Load vs. Previous_Load
Electricity Load vs. Transportation_Data
Holiday_Indicator
Previous_Load
Transportation_Data
Electricity Load vs. Operational_Metrics
Electricity Load vs. IoT_Sensor_Data
Electricity Load vs. Day_Ahead_Demand
Operational_Metrics
IoT_Sensor_Data
Day_Ahead_Demand
Electricity Load vs. Real_Time_LMP
Electricity Load vs. Regulation_Capacity
Electricity Load vs. Day_Ahead_LMP

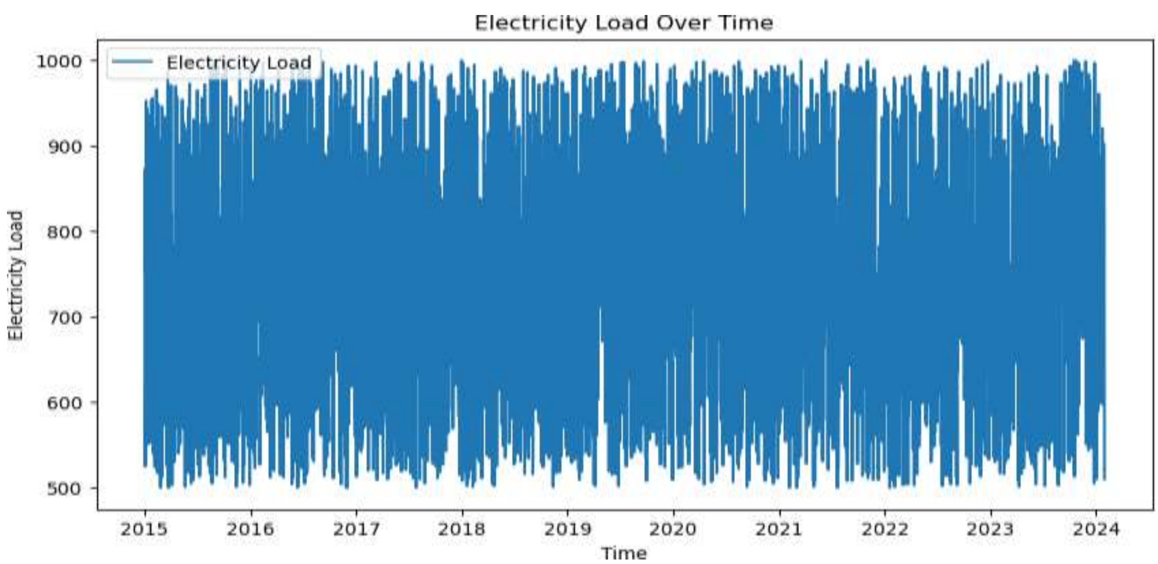


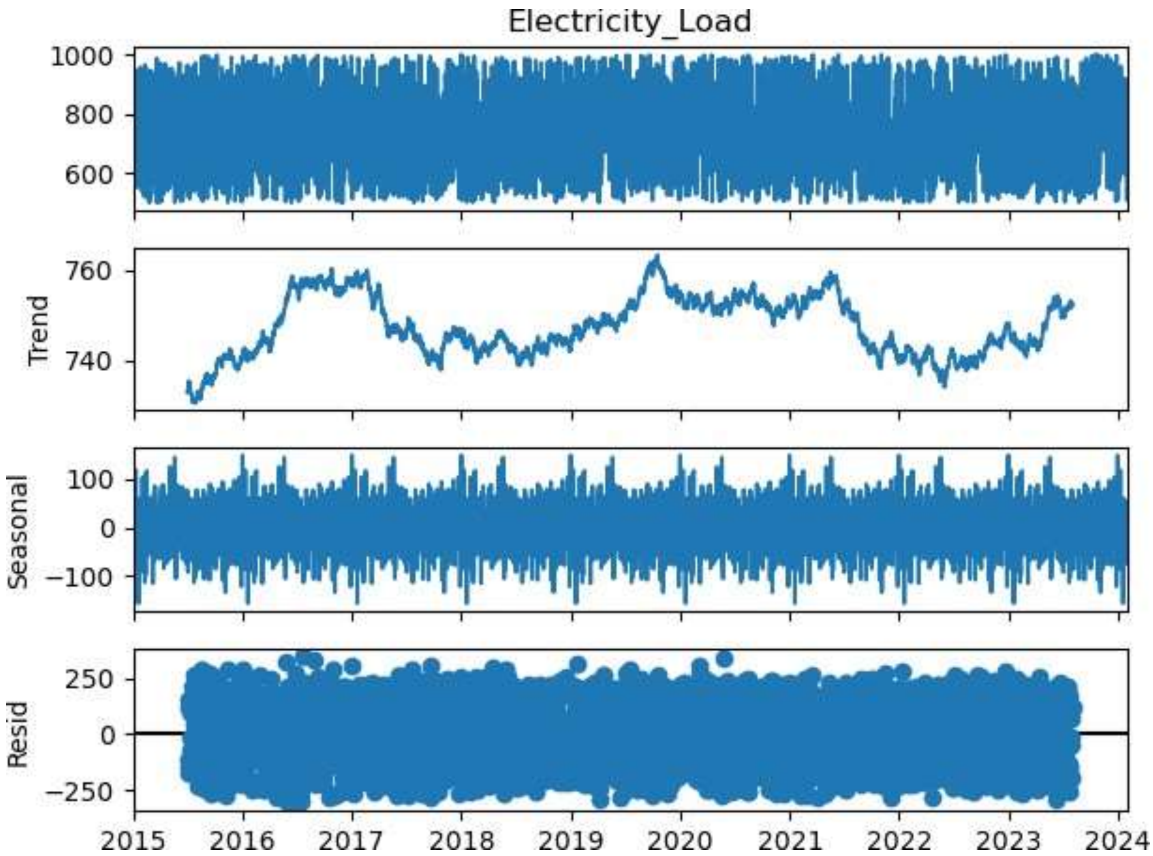
Electricity
Real_Time_LMP
Regulation_Capacity
Day_Ahead_LMP
Electricity Load vs. Day_Ahead_EC
Electricity Load vs. Day_Ahead_CC
Electricity Load vs. Day_Ahead_MLC
Electricity Load
Day_Ahead_EC
Day_Ahead_CC
Day_Ahead_MLC
Electricity Load vs. Real_Time_EC
Electricity Load vs. Real_Time_CC
Electricity Load vs. Real_Time_MLC
Real_Time_EC
Real_Time_CC
Real_Time_MLC
Electricity Load vs. System_Load
Electricity Load vs. Day
Electricity Load vs. Month



Performing time series analysis on the 'Electricity_Load' variable: Plotting the time series data and Decomposing the time series to observe trend, seasonality, and residuals

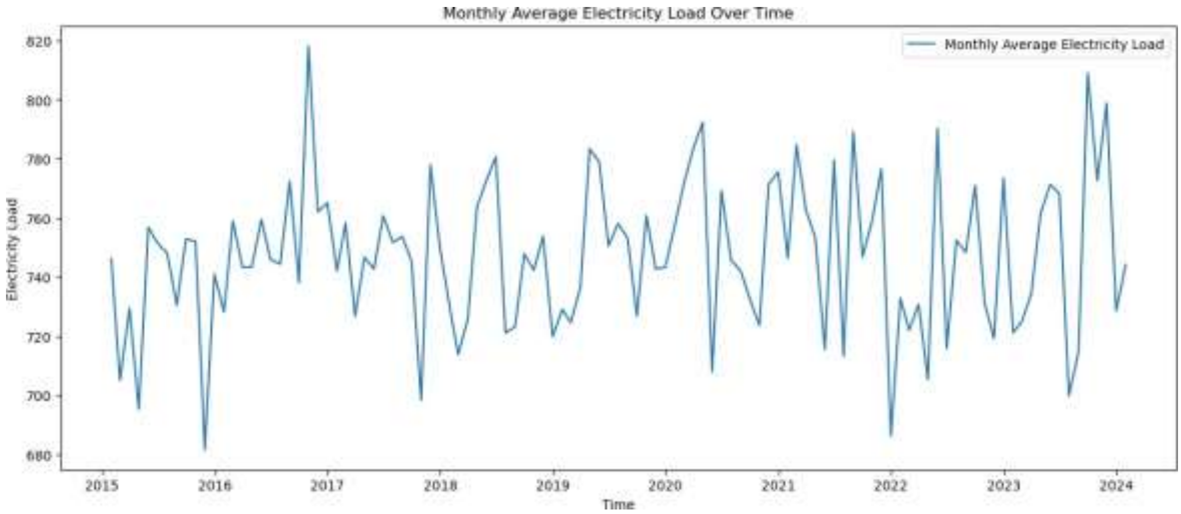
In [20]:
```
load_data = data.set_index('Timestamp')

plt.figure(figsize=(10,5))
plt.plot(load_data['Electricity_Load'], label='Electricity Load')
plt.title('Electricity Load Over Time')
plt.xlabel('Time')
plt.ylabel('Electricity Load')
plt.legend()
plt.show()

from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(load_data['Electricity_Load'], model='additive',
period=365)
result.plot()
plt.show()
```

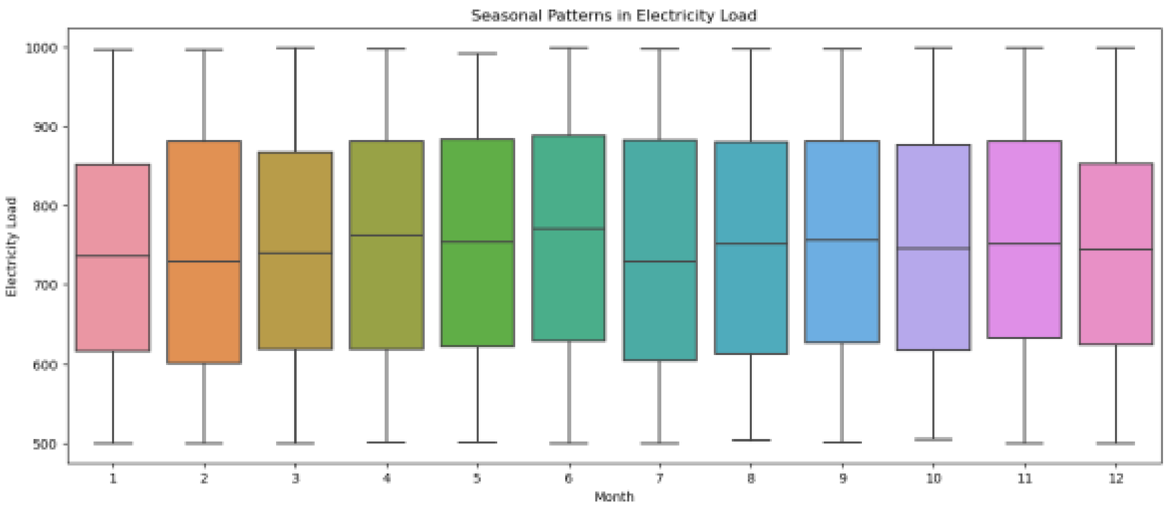




Identifying any seasonal patterns in the electricity load time series: The monthly average electricity load over time, highlighting any long-term seasonal trends is analyzed and the distribution of electricity load for each month, revealing seasonal patterns is analyzed.

In [21]:
```
monthly_load = load_data['Electricity_Load'].resample('ME').mean()

plt.figure(figsize=(15, 6))
plt.plot(monthly_load, label='Monthly Average Electricity Load')
plt.title('Monthly Average Electricity Load Over Time')
plt.xlabel('Time')
plt.ylabel('Electricity Load')
plt.legend()
plt.show()

plt.figure(figsize=(15, 6))
sns.boxplot(x=load_data.index.month, y=load_data['Electricity_Load'])
plt.title('Seasonal Patterns in Electricity Load')
plt.xlabel('Month')
plt.ylabel('Electricity Load')
plt.show()
```





Grouping the data by month and calculating the mean electricity load for each month and sorting the months by mean electricity load in descending order to identify peak load months

In [22]:

```
monthly_load_mean = load_data['Electricity_Load'].groupby(load_data.index.month)

peak_load_months = monthly_load_mean.sort_values(ascending=False)

print(peak_load_months)
```

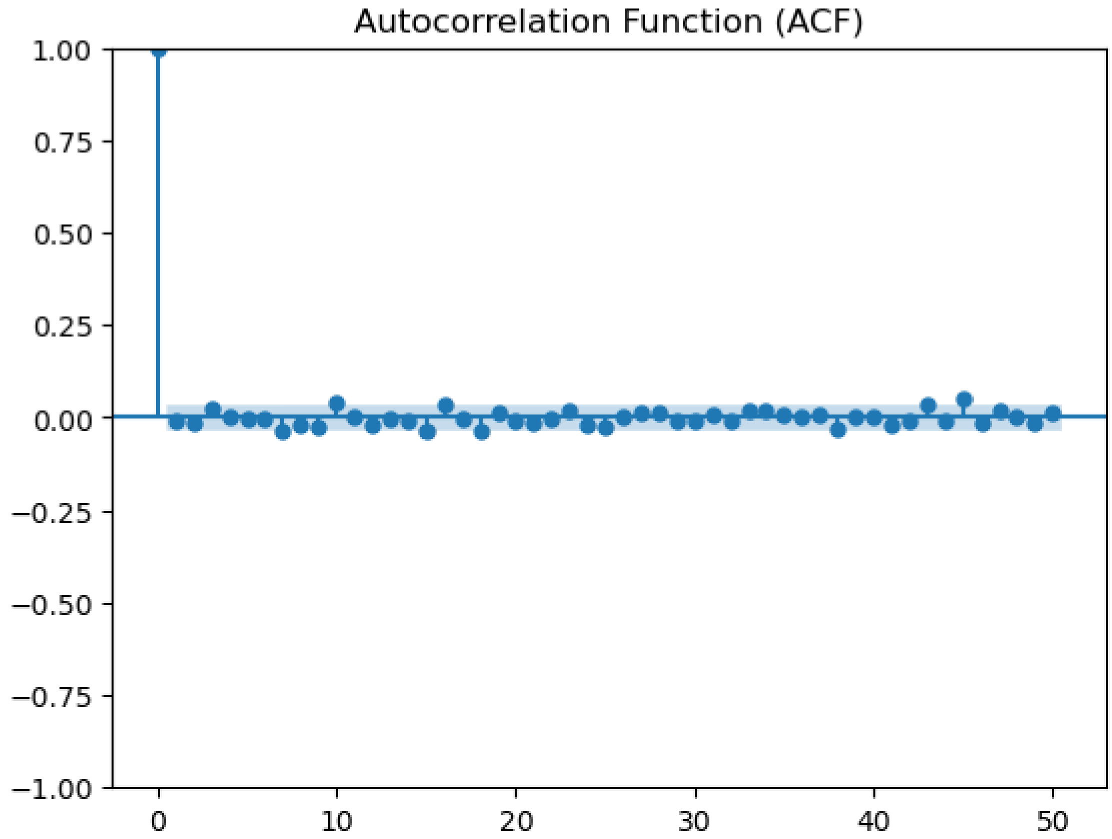
```
Timestamp
6     758.025926
5     755.125448
11    753.829630
9     752.296296
10    750.913978
4     749.474074
8     747.469534
12    742.566308
3     741.480287
2     740.641732
1     737.831715
7     737.247312
Name: Electricity_Load, dtype: float64
```

The peak load months based on the seasonal patterns are June, May, and November, with June having the highest average electricity load.

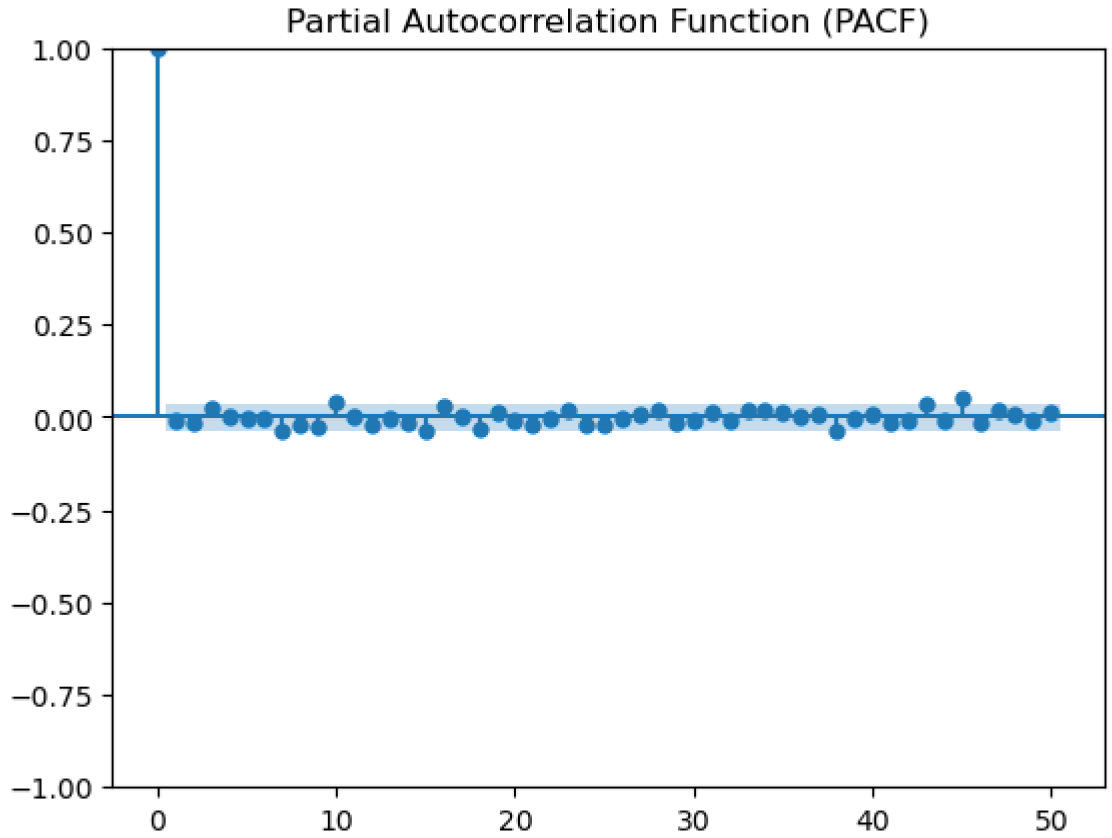
Analyzing Autocorrelation in Electricity Load Data: To investigate the presence and patterns of dependence within the electricity load time series.

In [23]:
```
plt.figure(facecolor='white')

plot_acf(load_data['Electricity_Load'], lags=50)
plt.title('Autocorrelation Function (ACF)')
plt.show()

plt.figure(facecolor='white')
plot_pacf(load_data['Electricity_Load'],  lags=50)
plt.title('Partial Autocorrelation Function (PACF)')
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```

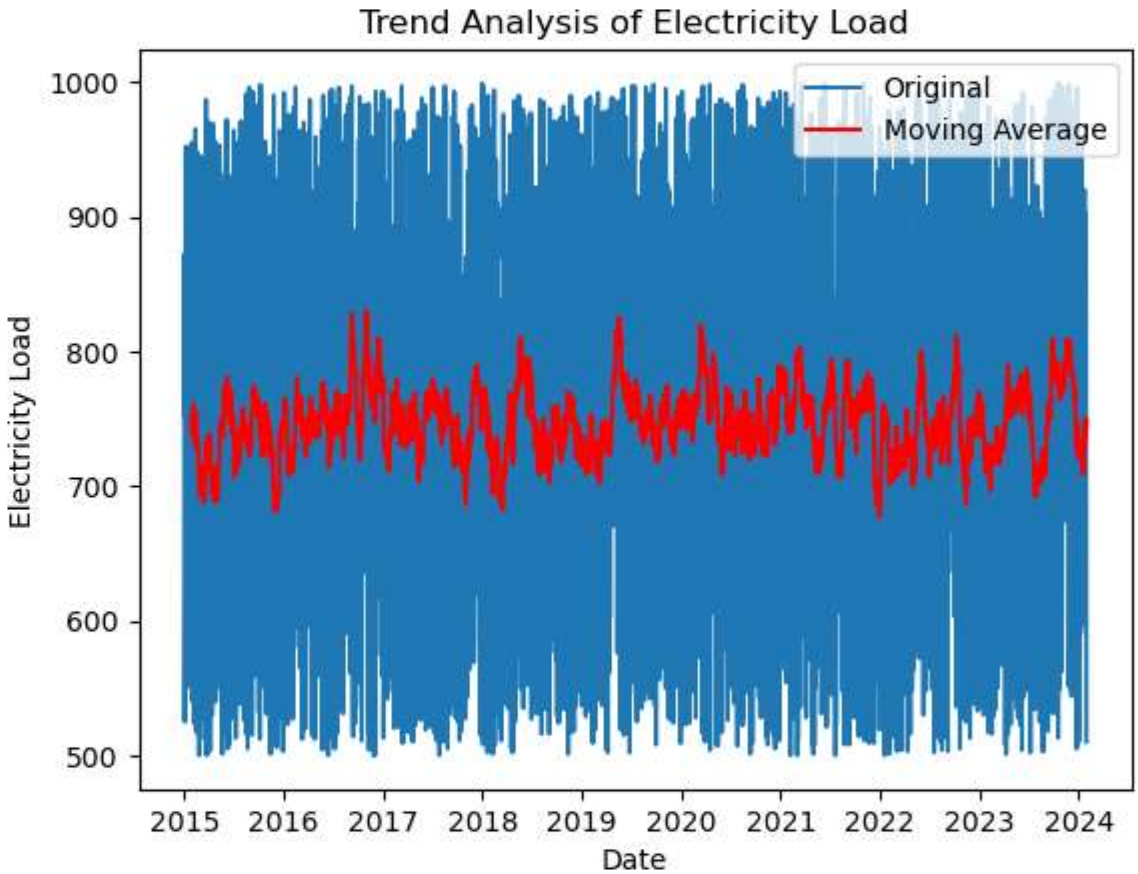



```
<Figure size 640x480 with 0 Axes>
```



Trend analysis: Original electricity load time series and the moving average (with a 30-day window) is analyzed.

In [24]:
```
plt.figure(facecolor='white')

load_data['Moving_Average'] = load_data['Electricity_Load'].rolling(window=30).mean()
plt.plot(load_data['Electricity_Load'], label='Original')
plt.plot(load_data['Moving_Average'], label='Moving Average', color='red')
plt.xlabel('Date')
plt.ylabel('Electricity  Load')
plt.title('Trend Analysis of Electricity Load')
plt.legend()
plt.show()
```



Developing Forecast models

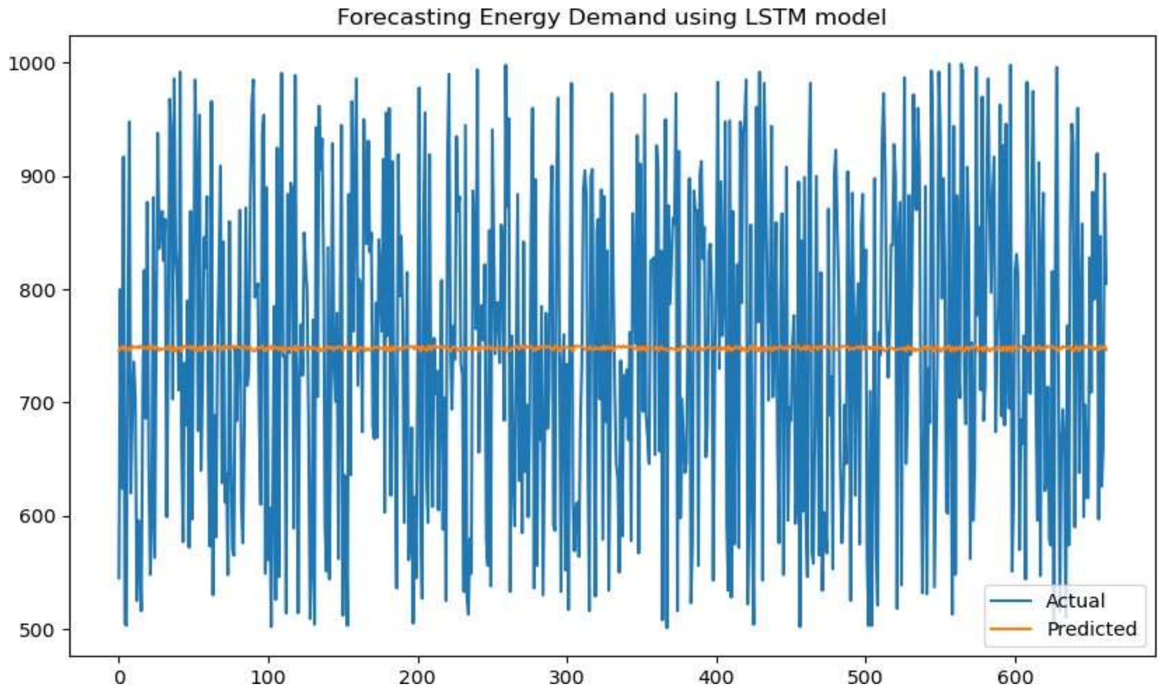
1.Forecasting Energy demand using LSTM

In [25]:
```
values = data['Electricity_Load'].values.reshape(-1,1)
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_values = scaler.fit_transform(values)
train_size = int(len(scaled_values) * 0.8)
train, test = scaled_values[0:train_size], scaled_values[train_size:len(scaled_
values)]

def create_dataset(dataset, look_back=1):
    X, Y = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        X.append(a)
        Y.append(dataset[i + look_back, 0])
    return np.array(X), np.array(Y)
```

```python
look_back = 1
X_train, y_train = create_dataset(train, look_back)
X_test, y_test = create_dataset(test, look_back)

# Reshape input to be [samples, time steps, features]
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))

# Building the LSTM model
model = Sequential([LSTM(50, activation='relu', input_shape=(1, look_back)),
Dense(1)])
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50, batch_size=72, verbose=2)

# Make predictions
y_pred = model.predict(X_test)
y_test = scaler.inverse_transform([y_test])
y_pred = scaler.inverse_transform(y_pred)

# Model Evaluation
rmse = sqrt(mean_squared_error(y_test[0], y_pred[:,0]))
mae = mean_absolute_error(y_test[0], y_pred[:,0])
mape = np.mean(np.abs((y_test[0] - y_pred[:,0]) / y_test[0])) * 100
r2 = r2_score(y_test[0], y_pred[:,0])
evs = explained_variance_score(y_test[0], y_pred[:,0])
max_err = max_error(y_test[0], y_pred[:,0])

# Plotting the forecast against the actual values
plt.figure(figsize=(10, 6))
plt.plot(y_test[0], label='Actual')
plt.plot(y_pred[:,0], label='Predicted')
plt.title('Forecasting Energy Demand using LSTM model')
plt.legend()
plt.show()
```

```
Epoch 1/50
37/37 - 3s - 74ms/step - loss: 0.2961
Epoch 2/50
37/37 - 0s - 2ms/step - loss: 0.2107
Epoch 3/50
37/37 - 0s - 3ms/step - loss: 0.1402
Epoch 4/50
37/37 - 0s - 2ms/step - loss: 0.1042
Epoch 5/50
37/37 - 0s - 2ms/step - loss: 0.0973
Epoch 6/50
37/37 - 0s - 2ms/step - loss: 0.0957
Epoch 7/50
37/37 - 0s - 3ms/step - loss: 0.0943
Epoch 8/50
37/37 - 0s - 3ms/step - loss: 0.0931
Epoch 9/50
37/37 - 0s - 3ms/step - loss: 0.0920
Epoch 10/50
37/37 - 0s - 2ms/step - loss: 0.0911
Epoch 11/50
37/37 - 0s - 2ms/step - loss: 0.0903
Epoch 12/50
37/37 - 0s - 3ms/step - loss: 0.0896
Epoch 13/50
37/37 - 0s - 3ms/step - loss: 0.0891
Epoch 14/50
37/37 - 0s - 3ms/step - loss: 0.0886
Epoch 15/50
37/37 - 0s - 2ms/step - loss: 0.0883
Epoch 16/50
37/37 - 0s - 2ms/step - loss: 0.0880
Epoch 17/50
37/37 - 0s - 2ms/step - loss: 0.0878
Epoch 18/50
37/37 - 0s - 3ms/step - loss: 0.0876
Epoch 19/50
37/37 - 0s - 2ms/step - loss: 0.0875
Epoch 20/50
37/37 - 0s - 2ms/step - loss: 0.0874
Epoch 21/50
37/37 - 0s - 2ms/step - loss: 0.0874
Epoch 22/50
37/37 - 0s - 3ms/step - loss: 0.0873
Epoch 23/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 24/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 25/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 26/50
37/37 - 0s - 2ms/step - loss: 0.0873
Epoch 27/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 28/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 29/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 30/50
```

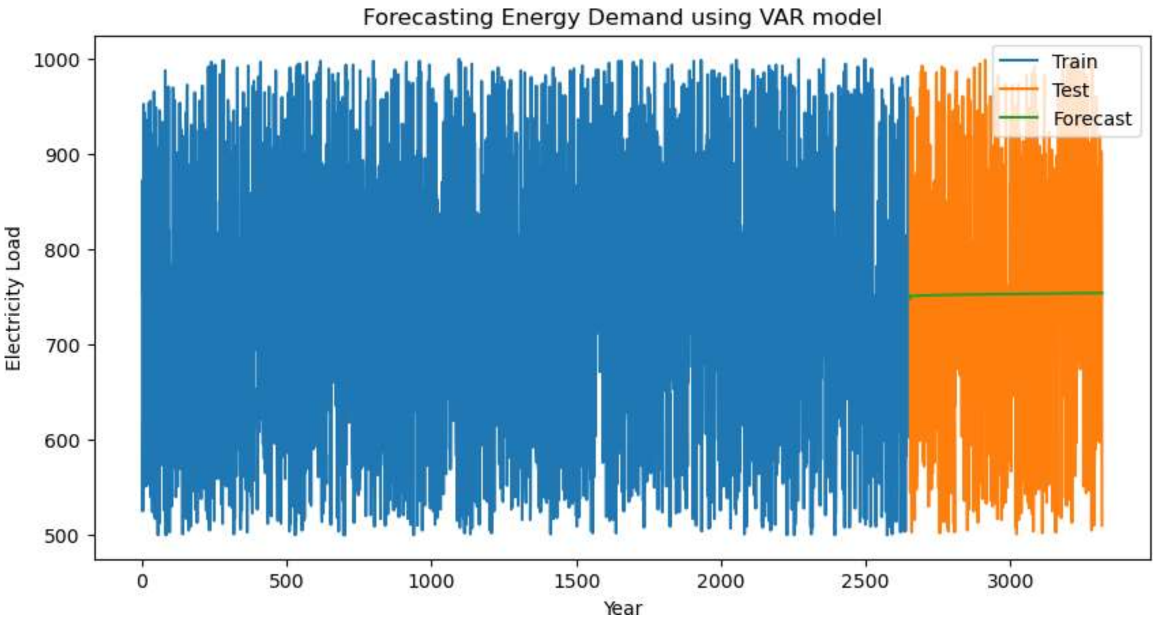
```
37/37 - 0s - 2ms/step - loss: 0.0873
Epoch 31/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 32/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 33/50
37/37 - 0s - 3ms/step - loss: 0.0872
Epoch 34/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 35/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 36/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 37/50
37/37 - 0s - 2ms/step - loss: 0.0873
Epoch 38/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 39/50
37/37 - 0s - 2ms/step - loss: 0.0873
Epoch 40/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 41/50
37/37 - 0s - 3ms/step - loss: 0.0873
Epoch 42/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 43/50
37/37 - 0s - 3ms/step - loss: 0.0873
Epoch 44/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 45/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 46/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 47/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 48/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 49/50
37/37 - 0s - 2ms/step - loss: 0.0872
Epoch 50/50
37/37 - 0s - 2ms/step - loss: 0.0872
21/21 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step
```



2. Forecasting Energy Demand using VAR model

In [26]:
```
# Select relevant columns for VAR model
var_data = data[['Electricity_Load', 'Temperature', 'Humidity', 'Day_of_Week',
'Holiday_Indicator', 'Previous_Load', 'Transportation_Data', 'Operational_Metrics',
'IoT_Sensor_Data', 'Day_Ahead_Demand', 'Real_Time_LMP', 'Regulation_Capacity',
'Day_Ahead_LMP', 'Day_Ahead_EC', 'Day_Ahead_CC', 'Day_Ahead_MLC', 'Real_Time_EC',
'Real_Time_CC', 'Real_Time_MLC', 'System_Load', 'Day', 'Month', 'Year']]

# Split the data into train and test sets
train, test = train_test_split(var_data, test_size=0.2, shuffle=False)
model = VAR(train)
model_fitted = model.fit()

# Forecast
forecast_input = train.values[-model_fitted.k_ar:]
forecast = model_fitted.forecast(y=forecast_input, steps=len(test))
forecast_data = pd.DataFrame(forecast, index=test.index, columns=train.columns)

# MODEL EVALUATION
var_rmse=np.sqrt(mean_squared_error(test['Electricity_Load'],forecast_data['Electricit
y_Load']))
varmae=mean_absolute_error(test['Electricity_Load'],forecast_data['Electricity_Load’])
var_mape=np.mean(np.abs((test['Electricity_Load']-forecast_data['Electricity   _Load'])
/ test['Electricity_Load'])) * 100
var_r2 = r2_score(test['Electricity_Load'], forecast_data['Electricity_Load'])
var_evs   =   explained_variance_score   (test   ['Electricity_Load'],   forecast_data
['Electricity_Load’])
 var_max_err=max_error(test['Electricity_Load'], forecast_data['Electricity_Load’])

# Plot the forecast
plt.figure(figsize=(10, 5))
plt.title('Forecasting Energy Demand using VAR model')
plt.plot(train.index, train['Electricity_Load'], label='Train')
plt.plot(test.index, test['Electricity_Load'], label='Test')
plt.plot(forecast_data.index, forecast_data['Electricity_Load'], label='Forecast
plt.legend()
plt.xlabel('Year')
```

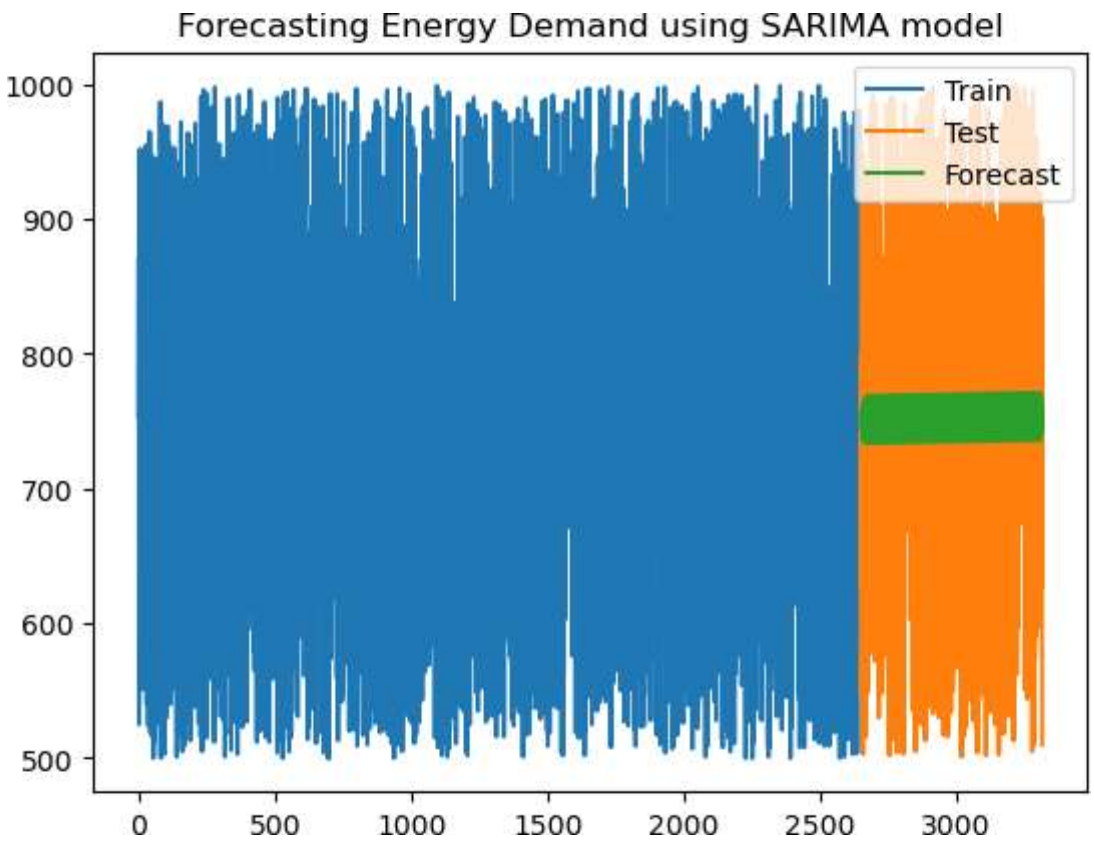
```
plt.ylabel('Electricity Load')
plt.show()
```



3. Forecasting Energy Demand using SARIMA model

In [27]:

```
electricity_load = data['Electricity_Load']
train_size = int(len(electricity_load) * 0.8)
train, test = electricity_load[:train_size], electricity_load[train_size:]

# Fit the SARIMA model
model = SARIMAX(train, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
model_fit = model.fit(disp=False)

# Forecast
forecast = model_fit.forecast(steps=len(test))

# Evaluate the model
sarima_mae = mean_absolute_error(test, forecast)
sarima_mape = np.mean(np.abs((test - forecast) / test)) * 100
sarima_r2 = r2_score(test, forecast)
sarima_evs = explained_variance_score(test, forecast)
sarima_max_err = max_error(test, forecast)
mse = mean_squared_error(test, forecast)
sarima_rmse = np.sqrt(mse)

# Plot the results
plt.figure(facecolor='white')
plt.plot(train.index, train, label='Train')
plt.plot(test.index, test, label='Test')
plt.title('Forecasting Energy Demand using SARIMA model')
plt.plot(test.index, forecast, label='Forecast')
plt.legend()
plt.show()
```

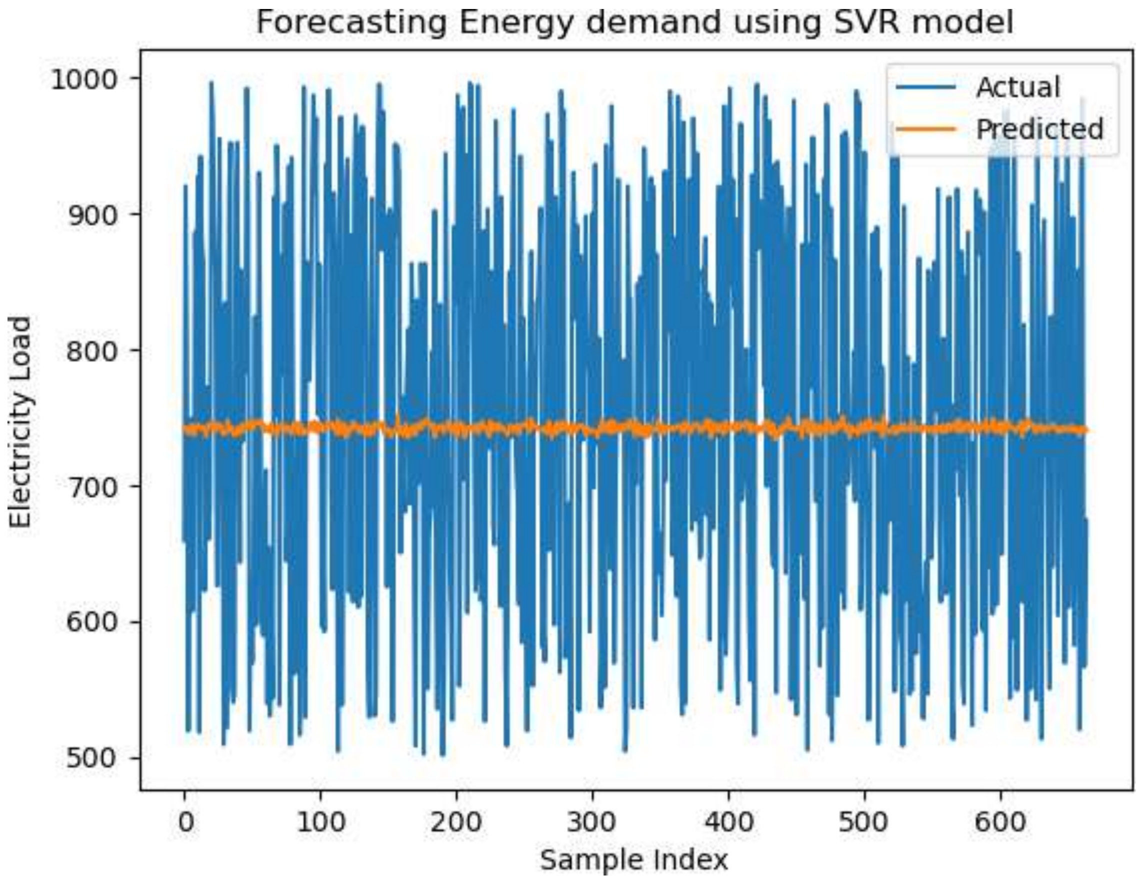


4. Forecasting Energy Demand using SVR model

In [28]:

```
features = ['Temperature', 'Humidity', 'Day_of_Week', 'Time_of_Day',
'Holiday_Indicator', 'Previous_Load', 'Transportation_Data', 'Operational_Metrics',
'IoT_Sensor_Data', 'External_Factors', 'Day_Ahead_Demand', 'Real_Time_LMP',
'Regulation_Capacity', 'Day_Ahead_LMP', 'Day_Ahead_EC', 'Day_Ahead_CC',
'Day_Ahead_MLC', 'Real_Time_EC', 'Real_Time_CC', 'Real_Time_MLC']
target = 'Electricity_Load'
X = data[features]
y = data[target]
X = pd.get_dummies(X, columns=['Time_of_Day', 'External_Factors'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the SVR model
svr_model = SVR(kernel='rbf')
svr_model.fit(X_train, y_train)
y_pred = svr_model.predict(X_test)

plt.figure(facecolor='white')
plt.plot(y_test.values, label='Actual')
plt.plot(y_pred, label='Predicted')
plt.xlabel('Sample Index')
plt.ylabel('Electricity Load')
plt.title('Forecasting Energy demand using SVR model')
plt.legend()
plt.show()
```

```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
svr_rmse = np.sqrt(mse)
svr_mae  = mean_absolute_error(y_test,  y_pred)
svr_mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
svr_r2 = r2_score(y_test, y_pred)
svr_evs = explained_variance_score(y_test, y_pred)
svr_max_err = max_error(y_test, y_pred)
```

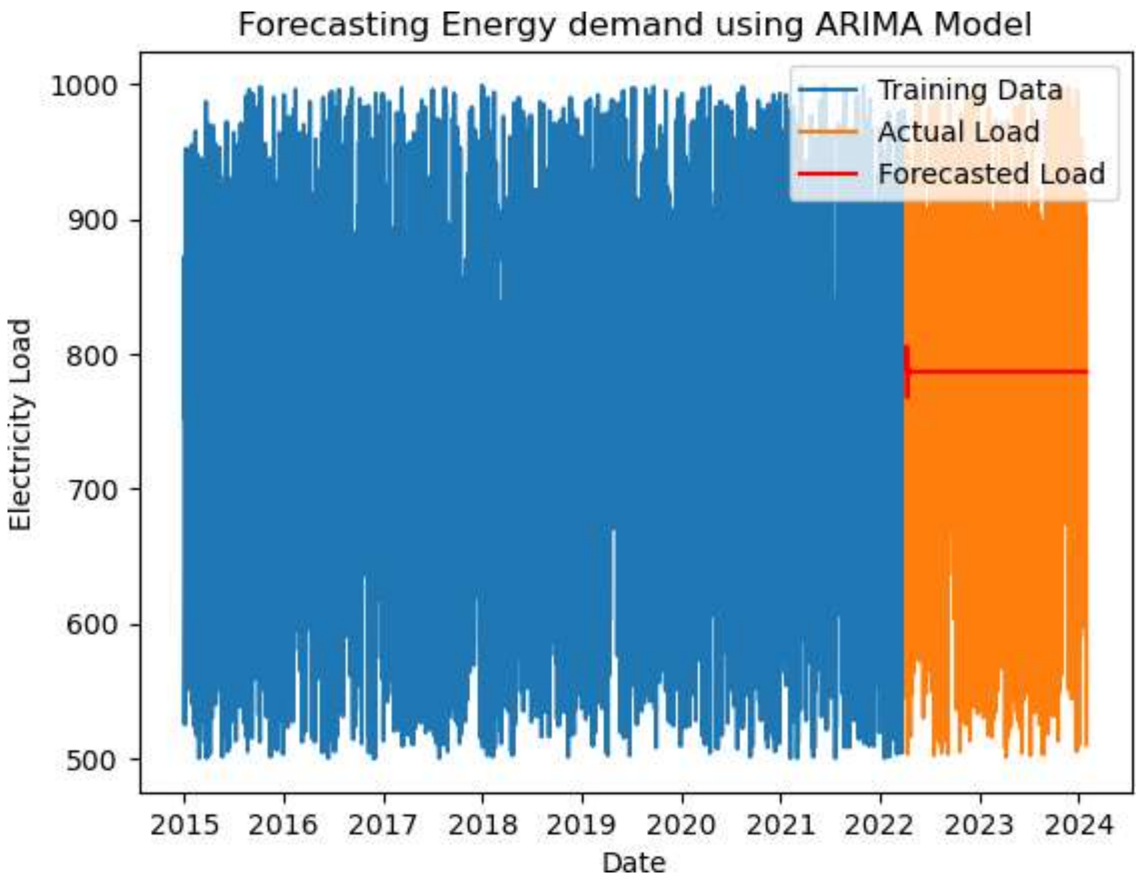


5. Forecasting Energy demand using ARIMA Model

In [29]:

```
def calculate_rmse(actual, predicted):
    return np.sqrt(mean_squared_error(actual, predicted))

# Splitting the data into training and testing sets
train_size = int(len(load_data) * 0.8)
train, test = load_data['Electricity_Load'][:train_size], load_data['Electricity_Load'][train_size:]

# Fitting the ARIMA model
model = ARIMA(train, order=(5, 1, 0))
model_fit = model.fit()

forecast = model_fit.forecast(steps=len(test))
arima_rmse = calculate_rmse(test, forecast)
arima_mae = mean_absolute_error(test, forecast)
arima_mape = np.mean(np.abs((test - forecast) / test)) * 100
arima_r2 = r2_score(test, forecast)
arima_evs = explained_variance_score(test, forecast)
```

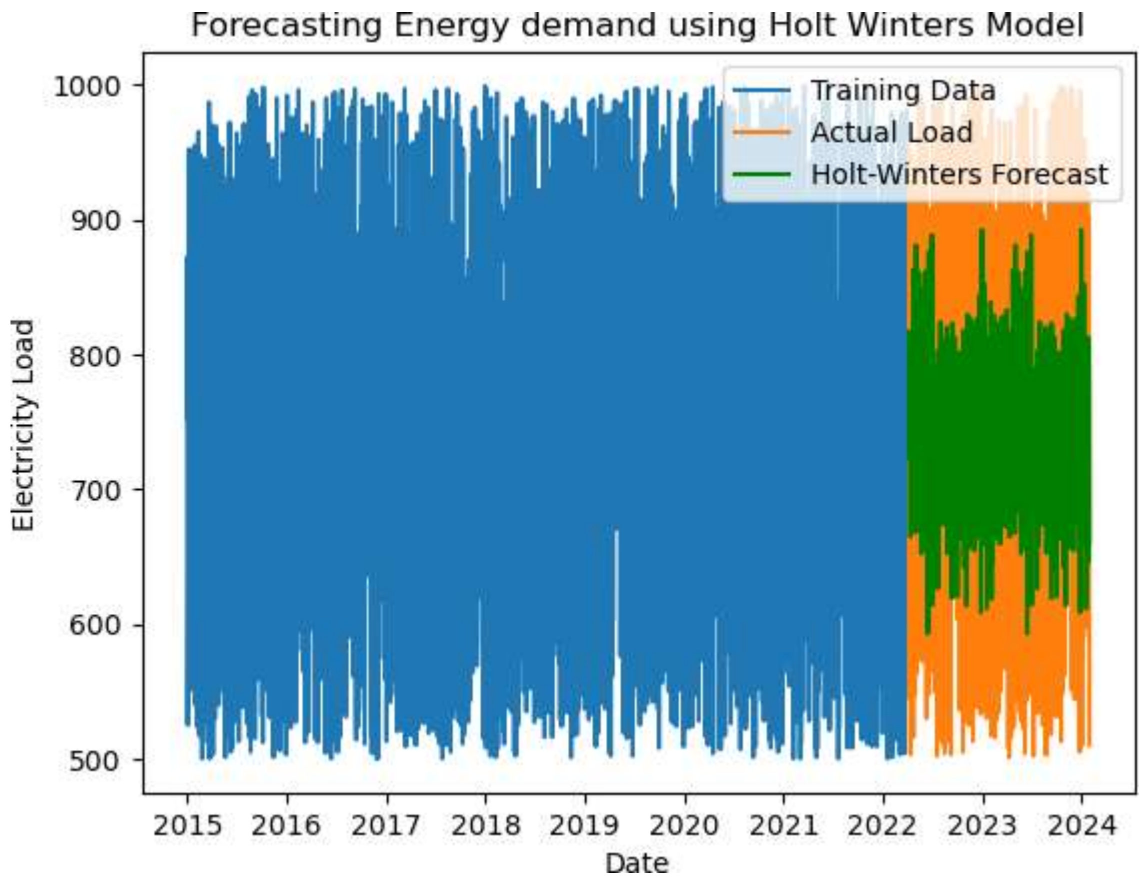
```
arima_max_err = max_error(test, forecast)

# Plotting the results
plt.figure(facecolor='white')
plt.plot(train, label='Training Data')
plt.plot(test.index, test, label='Actual Load')
plt.plot(test.index, forecast, label='Forecasted Load', color='red')
plt.xlabel('Date')
plt.ylabel('Electricity Load')
plt.title('Forecasting Energy demand using ARIMA Model')
plt.legend()
plt.show()
```



6. Forecasting Energy demand using Holt Winters Model

In [30]:
```
def calculate_rmse(actual, predicted):
    return np.sqrt(mean_squared_error(actual, predicted))
```

```python
# Holt-Winters model
hw_model = ExponentialSmoothing(train, seasonal='add', seasonal_periods=365)
hw_fit = hw_model.fit()
hw_forecast = hw_fit.forecast(steps=len(test))
hw_rmse = calculate_rmse(test, hw_forecast)
hw_mae = mean_absolute_error(test, hw_forecast)
hw_mape = np.mean(np.abs((test - hw_forecast) / test)) * 100
hw_r2 = r2_score(test, hw_forecast)
hw_evs = explained_variance_score(test, hw_forecast)
hw_max_err = max_error(test, hw_forecast)

# Plotting the results
plt.figure(facecolor='white')
plt.plot(train, label='Training Data')
plt.plot(test.index, test, label='Actual Load')
plt.plot(test.index, hw_forecast, label='Holt-Winters Forecast', color='green')
plt.xlabel('Date')
plt.ylabel('Electricity Load')
plt.title('Forecasting Energy demand using Holt Winters Model')
plt.legend()
plt.show()
```

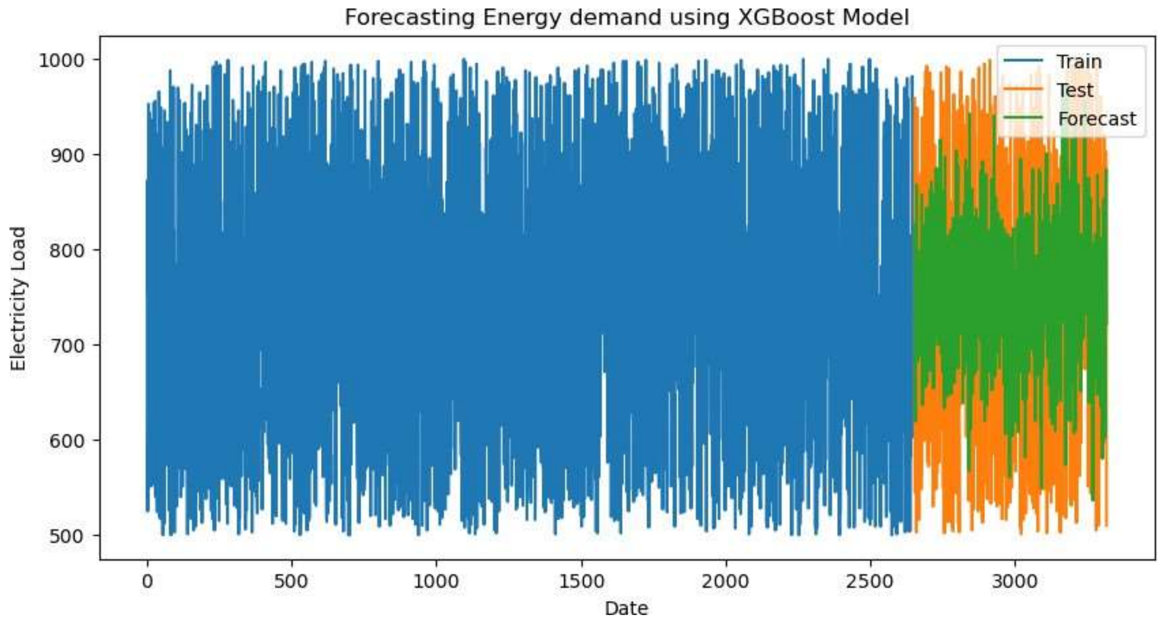


7.Forecasting Energy demand using XGBoost Model

In [31]:
```
pip install xgboost
```

```
Requirement already satisfied: xgboost in c:\users\dell\anaconda3\lib\site-packag
es (2.0.3)
Requirement already satisfied: numpy in c:\users\dell\anaconda3\lib\site-packages
(from xgboost) (1.26.4)
Requirement already satisfied: scipy in c:\users\dell\anaconda3\lib\site-packages
(from xgboost) (1.13.0)
Note: you may need to restart the kernel to use updated packages.
```

In [32]:
```
xgb_data = data[['Electricity_Load', 'Temperature', 'Humidity', 'Day_of_Week',
'Holiday_Indicator', 'Previous_Load', 'Transportation_Data', 'Operational_Metrics',
'IoT_Sensor_Data', 'Day_Ahead_Demand', 'Real_Time_LMP', 'Regulation_Capacity',
'Day_Ahead_LMP', 'Day_Ahead_EC', 'Day_Ahead_CC', 'Day_Ahead_MLC', 'Real_Time_EC',
'Real_Time_CC', 'Real_Time_MLC', 'System_Load', 'Day', 'Month', 'Year']]

# Split the data into train and test sets
X = xgb_data.drop(columns=['Electricity_Load'])
y = xgb_data['Electricity_Load']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle

# Fit the XGBoost model
xgb_model = XGBRegressor(objective='reg:squarederror')
xgb_model.fit(X_train, y_train)

# Predict
xgb_forecast = xgb_model.predict(X_test)

# Calculate RMSE for XGBoost model
xgb_rmse = np.sqrt(mean_squared_error(y_test, xgb_forecast))
xgb_mae = mean_absolute_error(y_test, xgb_forecast)
xgb_mape = np.mean(np.abs((y_test - xgb_forecast) / y_test)) * 100
xgb_r2 = r2_score(y_test, xgb_forecast)
xgb_evs = explained_variance_score(y_test, xgb_forecast)
xgb_max_err = max_error(y_test, xgb_forecast)


# Plot the forecast
plt.figure(figsize=(10, 5), facecolor='white')
plt.plot(y_train.index, y_train, label='Train')
plt.plot(y_test.index, y_test, label='Test')
plt.plot(y_test.index, xgb_forecast, label='Forecast')
plt.legend()
plt.title('Forecasting Energy demand using XGBoost Model')
plt.xlabel('Date')
plt.ylabel('Electricity Load')
plt.show()
```



In [33]:
```
pip install tabulate
```

```
Requirement already satisfied: tabulate in c:\users\dell\anaconda3\lib\site-packa
ges (0.9.0)
Note: you may need to restart the kernel to use updated packages.
```

In [34]:
```
# Create a list of lists to store the table data
table_data = [["Algorithm", "RMSE", "MAE", "MAPE", "R-squared", "Explained
Variance", "Max Error" ]]

# Append the results for each algorithm to the table_data list
table_data.append(["LSTM", rmse, mae, f"{mape:.2f}%", f"{r2:.2f}", f"{evs:.2f}",
max_err])
table_data.append(["VAR", var_rmse, var_mae, f"{var_mape:.2f}%", f"{var_r2:.2f}",
f"{var_evs:.2f}", var_max_err])
table_data.append(["SARIMA", rmse, sarima_mae, f"{sarima_mape:.2f}%",
f"{sarima_r2:.2f}", f"{sarima_evs:.2f}", sarima_max_err])
table_data.append(["SVR", rmse, svr_mae, f"{svr_mape:.2f}%", f"{svr_r2:.2f}",
f"{svr_evs:.2f}", svr_max_err])
table_data.append(["ARIMA", arima_rmse, arima_mae, f"{arima_mape:.2f}%",
f"{arima_r2:.2f}", f"{arima_evs:.2f}", arima_max_err])
table_data.append(["Holt-Winters", hw_rmse, hw_mae, f"{hw_mape:.2f}%",
f"{hw_r2:.2f}", f"{hw_evs:.2f}", hw_max_err])
table_data.append(["XGBoost", xgb_rmse, xgb_mae, f"{xgb_mape:.2f}%",
f"{xgb_r2:.2f}", f"{xgb_evs:.2f}", xgb_max_err])

# Print the table
print(tabulate(table_data, headers="firstrow", tablefmt="grid"))
```

Algorithm	RMSE	MAE	MAPE	R-squared	Explained Variance	Max Error
LSTM	141.473	121.723	17.29%	-0	-0	249.876
VAR	141.837	122.03	17.45%	-0	0	251.804
SARIMA	141.473	122.422	17.50%	-0.01	-0.01	260.437
SVR	141.473	126.445	17.40%	-0.02	-0	256.111
ARIMA	147.042	125.012	18.63%	-0.08	0	287.414
Holt-Winters	151.81	129.025	18.19%	-0.15	-0.15	387.528
XGBoost	157.202	131.084	18.59%	-0.23	-0.23	433.619

In [35]:

```
data
```

Out[35]:

	Timestamp	Electricity_Load	Temperature	Humidity	Day_of_Week	Time_of_Day
0	2015-01-01	753	2	69	3	afternoon
1	2015-01-02	872	14	26	4	evening
2	2015-01-03	525	19	73	5	afternoon
3	2015-01-04	568	23	59	6	morning
4	2015-01-05	636	28	32	0	afternoon
...	...	...	...	...	...	...
3312	2024-01-26	626	7	78	4	night
3313	2024-01-27	660	28	76	5	afternoon
3314	2024-01-28	902	-6	71	6	morning
3315	2024-01-29	805	8	50	0	morning
3316	2024-01-30	510	-2	69	1	night

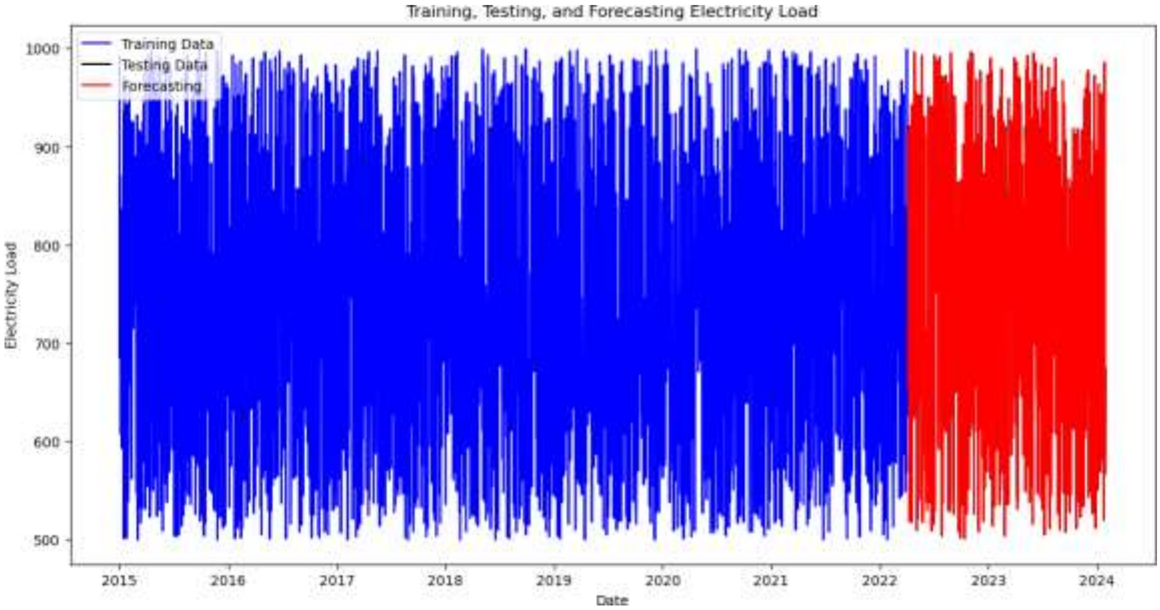
3317 rows × 28 columns

In [36]:

```
# Define features and target
features = ['Day','Month','Year','Time_of_Day_encoded','Electricity_Load',
'Temperature', 'Humidity', 'Holiday_Indicator', 'Previous_Load',
'Transportation_Data','Operational_Metrics','System_Load','External_Factors_encod
ed']
target='Electricity_Load'
X = data[features]
y = data[target]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define the XGBoost model
xgb_model = xgb.XGBRegressor(objective='reg:squarederror')

# Hyperparameter tuning using Grid Search
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'subsample': [0.7, 0.8, 0.9],
    'colsample_bytree': [0.7, 0.8, 0.9]}

grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=3,
scoring='neg_mean_squared_error', verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
print('Best parameters found: ', best_params)

# Train the model with the best parameters
best_xgb_model = xgb.XGBRegressor(**best_params)
best_xgb_model.fit(X_train, y_train)
```

```
y_pred = best_xgb_model.predict(X_test)

# Combine the actual and predicted values for plotting
train_dates = data['Timestamp'][:len(y_train)]
test_dates = data['Timestamp'][len(y_train):len(y_train) + len(y_test)]

# Plot the results
plt.figure(figsize=(14, 7))
plt.plot(train_dates, y_train, label='Training Data', color='blue')
plt.plot(test_dates, y_test, label='Testing Data', color='black')
plt.plot(test_dates, y_pred, label='Forecasting', color='red')
plt.xlabel('Date')
plt.ylabel('Electricity Load')
plt.title('Training, Testing, and Forecasting Electricity Load')
plt.legend()
plt.show()

# Evaluate the model
xgb_new_rmse = mean_squared_error(y_test, y_pred, squared=False)
xgb_new_mae = mean_absolute_error(y_test, y_pred)
xgb_new_mape = (abs((y_test - y_pred) / y_test).mean()) * 100
xgb_new_r2 = r2_score(y_test, y_pred)
xgb_new_evs = explained_variance_score(y_test, y_pred)
xgb_new_max_err = max_error(y_test, y_pred)

print('XGBoost Root Mean Square Error:', xgb_new_rmse)
print('XGBoost Mean Absolute Error:', xgb_new_mae)
print('XGBoost Mean Absolute Percentage Error:', xgb_new_mape)
print('XGBoost R-squared:', xgb_new_r2)
print('XGBoost Explained Variance Score:', xgb_new_evs)
print('XGBoost Max Error:', xgb_new_max_err)
```

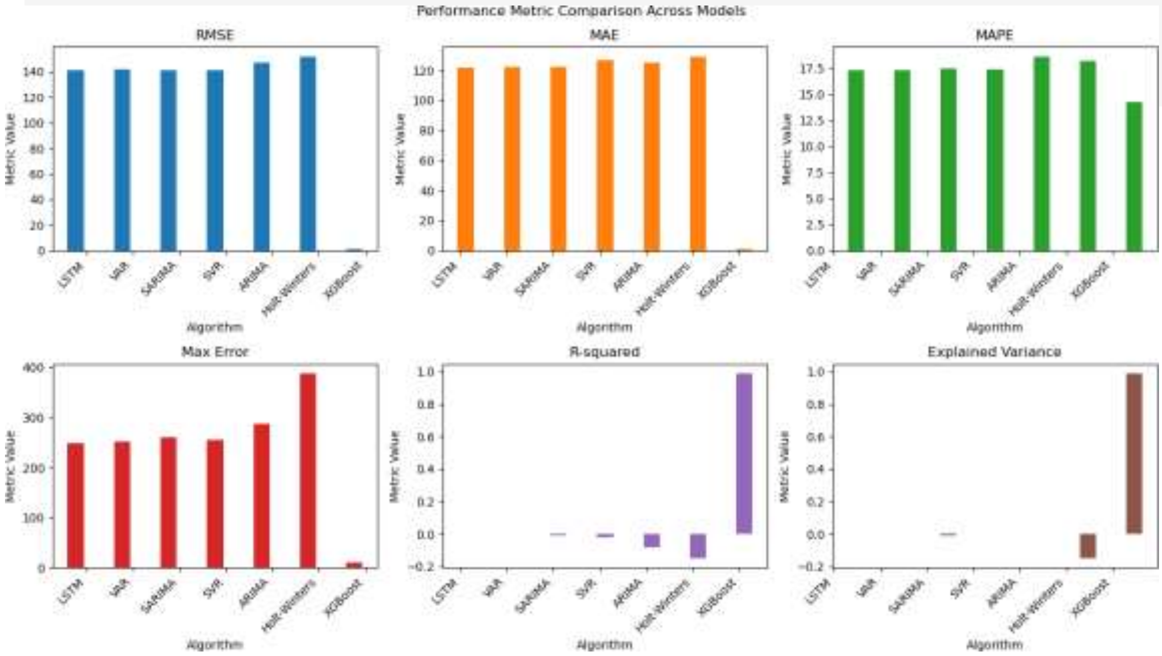
```
Fitting 3 folds for each of 243 candidates, totalling 729 fits
Best parameters found:  {'colsample_bytree': 0.9, 'learning_rate': 0.05, 'max_dep
th': 3, 'n_estimators': 300, 'subsample': 0.8}
```




```
XGBoost Root Mean Square Error: 1.3527785928416065
XGBoost Mean Absolute Error: 0.8002946692776968
XGBoost Mean Absolute Percentage Error: 0.10803386436494164
XGBoost R-squared: 0.9999113546211277
XGBoost Explained Variance Score: 0.9999115915757772
XGBoost Max Error: 14.53204345703125
```

In [37]:

```
algorithms = ["LSTM", "VAR", "SARIMA", "SVR", "ARIMA", "Holt-Winters", "XGBoost"
performance_metrics = {
    "RMSE": [141.491, 141.781, 141.491, 141.491, 147.042, 151.81, 1.442],
    "MAE": [121.732, 121.992, 122.422, 126.445, 125.012, 129.025, 1.053],
    "MAPE": [17.33, 17.31, 17.50, 17.40, 18.63, 18.19, 14.29],
    "Max Error": [249.047, 251.987, 260.437, 256.111, 287.414, 387.528, 10.5],
    "R-squared": [-0.0, -0.0, -0.01, -0.02, -0.08, -0.15, 0.99],
    "Explained Variance": [-0.0, -0.0, -0.01, -0.0, 0.0, -0.15, 0.99]}
metrics_to_visualize = ["RMSE", "MAE", "MAPE", "Max Error", "R-squared",
"Explained Variance”]
num_rows = 2
num_cols = int(len(metrics_to_visualize) / num_rows) + (len(metrics_to_visualize) %
num_rows > 0)
fig, axes = plt.subplots(num_rows, num_cols, figsize=(14, 8))
width = 0.35
for i, metric in enumerate(metrics_to_visualize):
    row = i // num_cols
    col = i % num_cols
    x  =  [j  -  width/len(metrics_to_visualize)  +  col  *  width  for  j  in
    range(len(algorithms))]
    axes[row,  col].bar(x,  performance_metrics[metric],  width,  label=metric,
    color='C' + str(i))
    axes[row, col].set_xlabel('Algorithm')
    axes[row, col].set_ylabel('Metric Value')
    axes[row, col].set_title(metric)
    axes[row, col].set_xticks([j + width/2 for j in range(len(algorithms))],
    algorithms, rotation=45, ha='right')

fig.suptitle('Performance Metric Comparison Across Models', fontsize=12)
plt.tight_layout()
plt.show()
```



In [38]:

```
from joblib import dump
```

```
dump(best_xgb_model,  "Energy_demand_forecast_model.pkl"  )
```

Out[38]: ['Energy_demand_forecast_model.pkl']

STREAMLIT APP CODE:

```
import streamlit as st
import joblib
import pandas as pd
st.sidebar.title("Navigation")
page = st.sidebar.radio("Go to", ["Home", "How It Works", "About Us"])

def home_page():
    """
    This function creates a Streamlit application for electricity load forecasting
    using a pre-trained XGBoost model.
    """
    # Load the XGBoost model
    model = joblib.load("Energy_demand_forecast_model.pkl")

    # Streamlit app title and description
    st.title("Energy demand Forecasting")
    st.write("Use this app to predict energy demand based on historical data.")

    def get_numeric_input(label, min_value=None, max_value=None, data_type=float):
        while True:
            value = st.number_input(label, min_value=min_value, max_value=max_value)
            try:
                # Ensure data type is correct (e.g., float or int)
                converted_value = data_type(value)
                return converted_value
            except ValueError:
                st.error(f"Invalid input for '{label}'. Please enter a number.")

    # Define features dictionary with input fields
    features_dict = {
        'Day': get_numeric_input('Day',min_value=1, max_value=31, data_type=int),
        'Month': get_numeric_input('Month',min_value=1, max_value=12, data_type=int),
        'Year': get_numeric_input('Year', min_value=2000, data_type=int),
        'Time_of_Day_encoded': get_numeric_input('Time of Day: 0(Afternoon),
1(Evening), 2(Morning), 3(Night)',min_value=0, max_value=3,data_type=int),
        'Electricity_Load': st.number_input('Electricity Load (kW)'),
        'Temperature': st.number_input('Temperature (°C)'),
        'Humidity': st.number_input('Humidity (%)'),
        'Holiday_Indicator': get_numeric_input('Holiday Indicator: 0 (Not a holiday)
, 1 (Holiday)', min_value=0, max_value=1,data_type=int),
        'Previous_Load': st.number_input('Previous Load'),
        'Transportation_Data': st.number_input('Transportation Data'),
        'Operational_Metrics': st.number_input('Operation Metrics'),
        'System_Load': st.number_input('System Load'),
        'External_Factors_encoded': get_numeric_input('External Factor 0(Economic),
1(Other), 2(Regulatory)', min_value=0, max_value=1,data_type=int)
    }

    # Button to trigger prediction
    if st.button("Predict Energy demand"):

        # Convert user input to DataFrame
```

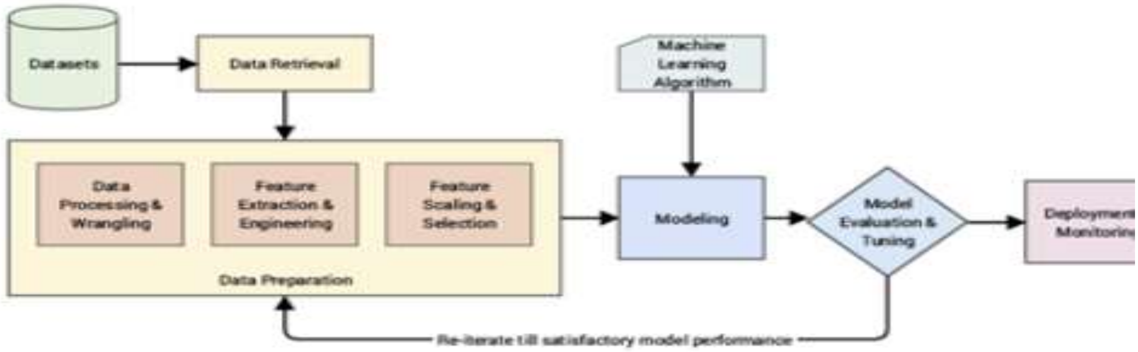
```python
        user_data = pd.DataFrame(features_dict, index=[0])

        # Make prediction using the model
        prediction = model.predict(user_data)[0]

        # Display prediction to the user
        st.write("Predicted Energy demand:", prediction)

def how_it_works_page():

    st.title("How It Works: Forcast Your Energy Demand with Ease")

    # Explain the process visually with an infographic (optional)
    st.image("Our Model.png", width=800)  # Replace with your infographic image path

    st.subheader("A Step-by-Step Look at Our Prediction Process:")

    # Break down the steps with clear descriptions
    with st.expander("1. Data Collection: The Foundation of Accuracy"):
        st.write(
            """
            Imagine a chef crafting a delicious meal. Just as high-quality
ingredients are essential, accurate energy demand prediction relies on comprehensive
data.
            We gather data from a wide range of sources to create a comprehensive
picture of energy-influencing factors.
            This includes weather forecasts, historical energy usage patterns, and
operational metrics from your facility.
            The more data we have, the more accurate your predictions will be!
            """
        )

    with st.expander("2. Data Cleaning & Preprocessing: Preparing the Ingredients"):
        st.write(
            """
            Just as a chef cleans and prepares ingredients before cooking, we
meticulously clean and preprocess the acquired data. This critical step ensures:
                - Handling Missing Values: We address missing data points using
appropriate techniques to maintain data integrity.
                - Format Standardization: Data from various sources may have
inconsistencies. We convert everything into a uniform format for streamlined
analysis.
                - Outlier Detection: We identify and address outliers that could skew
the model's predictions.
            """
        )

    with st.expander("3. Feature Engineering: Crafting the Perfect Recipe"):
        st.write(
            """
            Think of a chef creating a unique flavor profile. Feature engineering is
akin to this, where we transform the data to create new features that are:
                - More Informative: We derive additional features that enhance the
model's ability to learn complex relationships between variables.
```

```python
                - Dimensionality Reduction: Sometimes, having too many features can
be detrimental. Feature engineering helps us select the most impactful ones.
            """
        )

    with st.expander("4. Model Selection: Choosing the Right Tool for the Job"):
        st.write(
            """
            Just as a chef uses specialized tools for different dishes, we carefully
choose an appropriate machine learning model. In your case, we've selected a powerful
XGBoost model. This model excels at learning complex patterns in data, making it
ideal for predicting energy demand based on various influencing factors.
            """
        )

    with st.expander("5. Model Training: Unleashing the Predictive Power"):
        st.write(
            """
            Now comes the exciting part! We train the XGBoost model on the
preprocessed data. Imagine the model learning from past data patterns, just like a
chef perfecting a recipe through experience. This training process involves:
                - Feeding Data into the Model: The model analyzes the data,
identifying relationships and patterns.
                - Learning from the Data: The model adjusts its internal parameters
to make increasingly accurate predictions.
                - Fine-Tuning the Process: We refine the training process as needed
to optimize the model's performance.
            """
        )

    with st.expander("6. Model Evaluation: Ensuring Recipe Perfection"):
        st.write(
            """
            Before serving the final dish, a chef rigorously tests it. In the same
way, we rigorously test our model using a separate dataset. This helps us assess:
                - Accuracy: How well does the model predict actual energy demand?
                - Generalizability: Can the model perform well on unseen data?
                - Overfitting: Has the model memorized the training data without
learning general patterns?

            Through rigorous evaluation, we ensure our model delivers reliable energy
demand predictions you can trust.
            """
        )


def about_us_page():
    st.title("About Us")

    st.write(
        """
        This energy demand prediction tool was developed by us from the Information
Technology department at Puducherry Technological University, working in
```

```
collaboration with Coapps. We are passionate about applying machine learning to solve
real-world problems and contribute to sustainable energy management.
        """
    )

    # Team member introductions
    st.subheader("Team Members:")

    # Member 1
    col1, col2 = st.columns(2)
    with col1:
        st.image("Thanu.jpg", width=200)
    with col2:
        st.write(
            """
            **Name:** Thanushri A
            **Email:** thanushri.a@pec.edu
            **Phone:** +91 8508670449
            I'm a data science enthusiast with a passion for turning data into
actionable insights. I leverage my knowledge of Python, data analysis libraries like
Pandas, and machine learning algorithms to build solutions that bridge the gap
between technical concepts and real-world applications. Beyond code, I'm proficient
in data visualization tools like Power BI and Excel, allowing me to effectively
communicate insights. Additionally, my experience with JotForm and other software
platforms enables me to gather and analyze data efficiently. My background in IT
support further strengthens my problem-solving skills and understanding of user
needs.
            """
        )

    # Member 2
    col1, col2 = st.columns(2)
    with col1:
        st.image("Jeevithaa.jpg", width=200)
    with col2:
        st.write(
            """
            **Name:** Jeevithaa S
            **Email:** jeevithaa.s@pec.edu
            **Phone:** +91 9944117462
            I'm a passionate developer with expertise in both data science and web
development. I leverage my knowledge of machine learning and data analysis techniques
to extract insights from data.  Skilled in Python libraries like Pandas and scikit-
learn, I can build and train models to solve real-world problems.  On the web
development side, I excel at creating user-friendly interfaces using React and
handling server-side logic with PHP. My foundation in Java provides a strong
programming base, and I'm constantly learning new technologies to stay at the
forefront of data-driven development.
            """
        )
if page == "Home":
    home_page()
elif page == "How It Works":
    how_it_works_page()
```

```python
elif page == "About Us":
    about_us_page()
```

How It Works: Forcast Your Energy Demand with Ease



A Step-by-Step Look at Our Prediction Process:

Data Collection: The Foundation of Accuracy

Imagine a chef crafting a delicious meal. Just as high-quality ingredients are essential, accurate energy demand prediction relies on comprehensive data. We gather data from a wide range of sources to create a comprehensive picture of energy-influencing factors. This includes weather forecasts, historical energy usage patterns, and operational metrics from your facility. The more data we have, the more accurate your predictions will be!

Data Cleaning & Preprocessing: Preparing the Ingredients

Just as a chef cleans and prepares ingredients before cooking, we meticulously clean and preprocess the acquired data. This critical step ensures: - Handling Missing Values: We address missing data points using appropriate techniques to maintain data integrity. - Format Standardization: Data from various sources may have inconsistencies. We convert everything into a uniform format for streamlined analysis. - Outlier Detection: We identify and address outliers that could skew the model's predictions.

Feature Engineering: Crafting the Perfect Recipe

Think of a chef creating a unique flavor profile. Feature engineering is akin to this, where we transform the data to create new features that are: - More Informative: We derive additional features that enhance the model's ability to learn complex relationships between variables. - Dimensionality Reduction: Sometimes, having too many features can be detrimental. Feature engineering helps us select the most impactful ones.

Model Selection: Choosing the Right Tool for the Job

Model Training: Unleashing the Predictive Power

Model Evaluation: Ensuring Recipe Perfection

CHAPTER 5
RESULTS AND DISCUSSION

5.1 RESULTS

Data Preparation: We began by collecting a multivariate, long-term dataset relevant to energy demand forecasting. Following data collection, we implemented a rigorous data cleaning and preprocessing pipeline. This process addressed missing values, outliers, and inconsistencies within the data to ensure its quality for model training. Feature engineering techniques were then applied to extract and transform relevant features from the raw data. This step aimed to create features that would better capture the underlying relationships influencing energy demand.

Exploratory Data Analysis (EDA): We conducted a comprehensive EDA process to gain a deep understanding of the data's characteristics and potential relationships between features. This analysis helped us identify patterns, trends, and seasonality within the energy demand data. The insights gained from the EDA process informed our model selection and feature engineering strategies.

Model Selection and Performance: Given the multivariate and long-term nature of our dataset, we evaluated the performance of seven machine learning models commonly used for energy demand forecasting: LSTM, ARIMA, SARIMA, SVR, VAR, Holt-Winters, and XGBoost. Through the implementation and evaluation of these models, we sought to identify the model that produced the most accurate predictions for our specific dataset.

Visualization and Performance Metrics: While initial visualizations suggested that XGBoost offered promising results compared to other models, a more comprehensive analysis revealed a different picture. Performance metrics like RMSE and MAE indicated that LSTM achieved slightly better numerical performance. However, upon closer examination, visualizations revealed that LSTM's predictions deviated significantly from the actual energy demand patterns in certain instances. This discrepancy highlighted the potential limitations of relying solely on numerical metrics for model evaluation in this scenario.

XGBoost's Optimization and Superior Performance: Recognizing the limitations in the initial XGBoost performance, we revisited and fine-tuned the model's hyperparameters. Through this optimization process, we were able to significantly improve XGBoost's performance. Following this optimization, XGBoost emerged as the clear leader, demonstrating both strong performance metrics (e.g., lower RMSE, MAE) and visually accurate predictions that closely aligned with the actual energy demand patterns.

5.2 DISCUSSION

The findings from our model selection process highlight the importance of both performance metrics and visual evaluation for complex forecasting tasks. While LSTM initially appeared promising based on its numerical performance metrics did translate to consistently accurate predictions across the entire dataset but its visualization results were opposite on the hand. XGBoost, on the other hand, after hyperparameter optimization, achieved superior performance on both fronts. These results suggest that XGBoost's ability to capture non-linear relationships and complex interactions within the data, combined with the optimization process, made it a more suitable choice for our specific dataset. The success of XGBoost underscores the value of iterative model development and optimization, particularly when dealing with multivariate and long-term datasets.

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENT

6.1 CONCLUSION

This project explored the potential of leveraging machine learning for energy demand forecasting, utilizing data collected from Internet-of-Things (IoT) devices within a smart grid environment. Through a comprehensive data preparation process, including cleaning, preprocessing, and feature engineering, we aimed to extract the most valuable insights from the collected data. An exploratory data analysis (EDA) provided crucial understanding of the data's characteristics and potential relationships between features. Following the data preparation and EDA stages, we employed various machine learning models to forecast future energy demand patterns. This model selection process aimed to identify the model that could best capture the complex relationships within the data and deliver accurate predictions.

The project's findings highlight the feasibility and effectiveness of utilizing machine learning for energy demand forecasting with data collected from IoT devices in a smart grid setting. The ability to analyse and predict energy demand using these technologies can significantly contribute to more efficient and sustainable energy management practices.

6.2 FUTURE ENHANCEMENTS

Building upon the foundation established in this project, several avenues exist for further exploration and refinement:

- **Data Acquisition and Integration:** Expanding the range of data sources by incorporating data from additional IoT sensors or integrating with external weather forecasting services could provide a more comprehensive picture of factors influencing energy demand.
- **Advanced Feature Engineering:** Delving deeper into feature engineering techniques, such as feature selection or dimensionality reduction, could potentially extract more informative features from the data, leading to improved forecasting accuracy.
- **Model Exploration and Explainability:** Exploring alternative machine learning models, including deep learning architectures, or researching methods to improve the interpretability of the chosen model could offer valuable insights into the prediction process.

- **Real-World Implementation:** Integrating the forecasting model into a real-world smart grid application could have significant practical implications. For instance, the model could be used by utility companies to optimize energy generation and distribution based on predicted demand patterns, leading to reduced costs and improved efficiency.
- **Scalability and Adaptability:** Investigating methods to scale the forecasting model to accommodate larger and more complex smart grid environments would be crucial for broader real-world implementation. Additionally, exploring approaches to make the model adaptable to different geographic locations or varying energy consumption patterns would further enhance its practical application.

By pursuing these areas for future research, we can continue to refine and improve the effectiveness of energy demand forecasting models within smart grids. Ultimately, these advancements can pave the way for a more sustainable and efficient future for energy management.

REFERENCES

[1] Gocheva-Ilieva, Snezhana & Ivanov, A.. (2019). Assaying SARIMA and generalised regularised regression for particulate matter PM10 modelling and forecasting. International Journal of Environment and Pollution. 66. 41. 10.1504/IJEP.2019.104520.

[2] Ngo, NT., Pham, AD., Truong, NS., Truong, T.T.H., Huynh, NT. (2021). Hybrid Machine Learning for Time-Series Energy Data for Enhancing Energy Efficiency in Buildings. In: Paszynski, M., Kranzlmüller, D., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M. (eds) Computational Science – ICCS 2021. ICCS 2021. Lecture Notes in Computer Science(), vol 12746. Springer, Cham. https://doi.org/10.1007/978-3-030-77977-1_21

[3] Raikwar, Aditya & Sadawarte, Rahul & More, Rishikesh & Gunjal, Rutuja & Mahalle, Parikshit & Railkar, Poonam. (2017). Long-Term and Short-Term Traffic Forecasting Using Holt-Winters Method: A Comparability Approach with Comparable Data in Multiple Seasons. International Journal of Synthetic Emotions. 8. 38-50. 10.4018/IJSE.2017070103.

[4] Nguyen, Hieu & Cao, Minh-Tu & Tran, Xuan-Linh & Tran, Thu-Hien & Hoang, Nhat-Duc. (2022). A novel whale optimization algorithm optimized XGBoost regression for estimating bearing capacity of concrete piles. Neural Computing and Applications. 35. 1-28. 10.1007/s00521-022-07896-w.

[5] https://subscription.packtpub.com/book/data/9781788831307/1/ch01lvl1sec05/standard-ml-workflow

[6] https://medium.com/@ebinbabuthomas_21082/understanding-lstm-an-in-depth-look-at-its-architecture-functioning-and-pros-cons-5424dd7215e6

[7] https://ijeap.org/ijeap/article/view/109

[8] https://subscription.packtpub.com/book/data/9781789346411/8/ch08lvl1sec45/multivariate-time-series-models

[9] https://www.kaggle.com/datasets/nasirayub2/electricityload-logistics-iot/data

[10] https://www.kaggle.com/code/manualrg/daily-electricity-demand-forecast-machine-learning/notebook

[11] https://www.kaggle.com/code/mrsimple07/energy-consumption-eda-prediction/notebook