PUSL3121 BigData Analytics

C2 - Coursework

Appliances Energy Prediction

Supervisor: Gayan Perera

10900340 : Esther Jebasuthanthirany
10900339 : Joshua Jebapragashan
10899485 : Warnakulasooriya Primalsha
10898730 : Patikara Ambahera

Degree Program: BSc (Hons) in Data Science

Table of Contents

What is Apache Spark?

Apache Spark is a high-speed, open-source data-processing engine optimized for machine learning (ML) and artificial intelligence (AI). Supported by the biggest open-source community in big data, Spark efficiently manages large-scale datasets and serves as a fast, versatile clustering system—ideal for PySpark. It delivers the speed, scalability, and flexibility needed for big data workloads, including streaming analytics, graph processing, large-scale ETL, ML, and AI.

Compared to alternatives like Hadoop MapReduce, Spark processes data **10–100x faster** for smaller workloads by retaining data in memory rather than relying on disk I/O. Its distributed architecture enables parallel processing across massive computing clusters while ensuring fault tolerance. Spark also supports popular programming languages such as Python, R, Scala, and Java, making it accessible to data scientists and engineers.

Originally developed in 2009 at UC Berkeley's AMPLab, Spark is now maintained by the **Apache Software Foundation** with contributions from over 1,000 developers. It has become a core component of many commercial big data solutions and remains a leading choice for in-memory analytics.

How Apache Spark works

Apache Spark follows a hierarchical primary/secondary architecture. The **Spark Driver** acts as the primary node, coordinating with the cluster manager to control worker nodes and return processed data to the client application.

Upon executing the application code, the Spark Driver creates a **SparkContext**, which interfaces with the cluster manager—such as Spark's standalone manager, Hadoop YARN, Kubernetes, or Mesos—to distribute tasks and monitor execution across the cluster. A critical innovation of Spark is its use of **Resilient Distributed Datasets (RDDs)**, in-memory data structures that enable fault tolerance and high-speed processing by minimizing disk I/O.

Resilient Distributed Dataset (RDD)

Resilient Distributed Datasets (RDDs) are fault-tolerant collections of elements that can be distributed among multiple nodes in a cluster and worked on in parallel. RDDs are a fundamental structure in Apache Spark.

Spark may parallelize an existing collection using the SparkContext parallelize technique of caching data into an RDD for processing, or it can load data by referencing a data source. The secret to Spark's speed is its ability to transform and operate on RDDs in memory after data has been fed into them. Unless the system runs out of memory or the user chooses to write the data to disc for persistence, Spark also keeps the data in memory.

An RDD splits each dataset into logical segments that can be calculated on several cluster nodes. Additionally, users are able to execute two different kinds of RDD operations: actions and transformations. Operations used to produce a new RDD are called transformations. Apache Spark is instructed to perform calculation and return the outcome to the driver via actions.

A wide range of actions and transformations on RDDs are supported by Spark. Spark handles this distribution, so users don't need to worry about figuring out the proper distribution.

Directed Acyclic Graph (DAG)

Spark uses a Directed Acyclic Graph (DAG) to schedule tasks and orchestrate worker nodes throughout the cluster, in contrast to MapReduce's two-stage execution procedure. By coordinating worker nodes throughout the cluster, the DAG scheduler promotes efficiency when Spark acts and modifies data during task execution processes. Because it applies the recorded operations to the data from a previous state, this task-tracking enables fault tolerance.

DataFrames and Datasets

Spark supports three main data types: RDDs, DataFrames, and Datasets.

DataFrames are the most widely used structured APIs in Spark, representing data in a tabular format with rows and columns. While RDDs played a crucial role in Spark’s early development, they are now in maintenance mode. Due to the growing popularity of Spark’s Machine Learning Library (MLlib), DataFrames have become the primary API for MLlib, which includes scalable machine learning algorithms, feature selection tools, and ML pipeline construction. This shift is significant because DataFrames offer a consistent interface across multiple languages, including Scala, Java, Python, and R.

Datasets extend DataFrames by providing a type-safe, object-oriented programming interface. Unlike DataFrames, which store data in an untyped format, Datasets consist of strongly typed JVM objects by default.

Additionally, Spark SQL allows querying data from both DataFrames and SQL-based data stores like Apache Hive. When executed in another language, Spark SQL queries return results in the form of a DataFrame or Dataset.

Spark Core

Spark Core serves as the foundation for all parallel data processing in Spark, managing scheduling, optimization, RDDs, and data abstraction. It provides the essential framework for various Spark libraries, including Spark SQL, Spark Streaming, MLlib for machine learning, and GraphX for graph data processing.

Working alongside the cluster manager, Spark Core distributes and abstracts data across the Spark cluster. This efficient distribution and abstraction enable fast and user-friendly handling of Big Data.

Spark APIs

Spark offers a wide range of application programming interfaces (APIs) to extend its power to a diverse audience. Spark SQL allows for interacting with RDD data in a relational format. Additionally, Spark provides comprehensive APIs for Scala, Java, Python, and R, each with its own specific approach to data handling. RDDs, DataFrames, and Datasets are available across all language APIs. By supporting multiple languages, Spark makes big data processing more accessible to a broader range of professionals, including developers, data scientists, data engineers, and statisticians.

Advantages of Apache Spark

Apache Spark accelerates both development and operations in several key ways, benefiting teams in the following ways:

- **Accelerates App Development**: Spark's Streaming and SQL programming models, along with its integration of MLlib and GraphX, simplify the creation of applications that leverage machine learning and graph analytics.
- **Fosters Innovation**: Spark’s APIs make it easier to handle semi-structured data and perform data transformations, enabling quicker innovation.
- **Simplified Management**: A unified engine supports a variety of workloads, including SQL queries, streaming data, machine learning (ML), and graph processing, streamlining operations.
- **Optimizes with Open Technologies**: The OpenPOWER Foundation supports GPU, CAPI Flash, RDMA, and FPGA acceleration, boosting performance for Apache Spark workloads while encouraging machine learning innovations.

- **Improves Processing Speed**: Spark can be up to 100 times faster than Hadoop for smaller workloads, thanks to its advanced in-memory computing engine and efficient disk data storage.
- **Speeds Memory Access**: Spark allows the creation of a large memory space for data processing, giving advanced users the ability to access data through interfaces such as Python, R, and Spark SQL.

Apache Spark and machine learning

Spark offers a range of libraries that enhance its capabilities in machine learning, artificial intelligence (AI), and stream processing.

Apache Spark MLlib

A key feature of Apache Spark is its machine learning capabilities through Spark MLlib. MLlib offers an out-of-the-box solution for tasks such as classification and regression, collaborative filtering, clustering, distributed linear algebra, decision trees, random forests, gradient-boosted trees, frequent pattern mining, evaluation metrics, and statistics. These powerful features, along with Spark's support for various data types, make it an essential tool for big data processing.

Spark GraphX

In addition to its API capabilities, Spark GraphX is specifically designed to address graph-related problems. GraphX is a graph abstraction that extends RDDs to support graph and graph-parallel computation. It integrates seamlessly with graph databases, which store interconnectivity or connection data, such as those used in social networks.

Spark Streaming

Spark Streaming is an extension of the core Spark API that allows for scalable, fault-tolerant processing of live data streams. As it processes data, Spark Streaming can deliver it to file systems, databases, and live dashboards, enabling real-time streaming analytics powered by Spark's machine learning and graph-processing algorithms. Built on the Spark SQL engine, Spark Streaming also supports incremental batch processing, which enhances the speed of streamed data processing.

Spark vs. Apache Hadoop and MapReduce

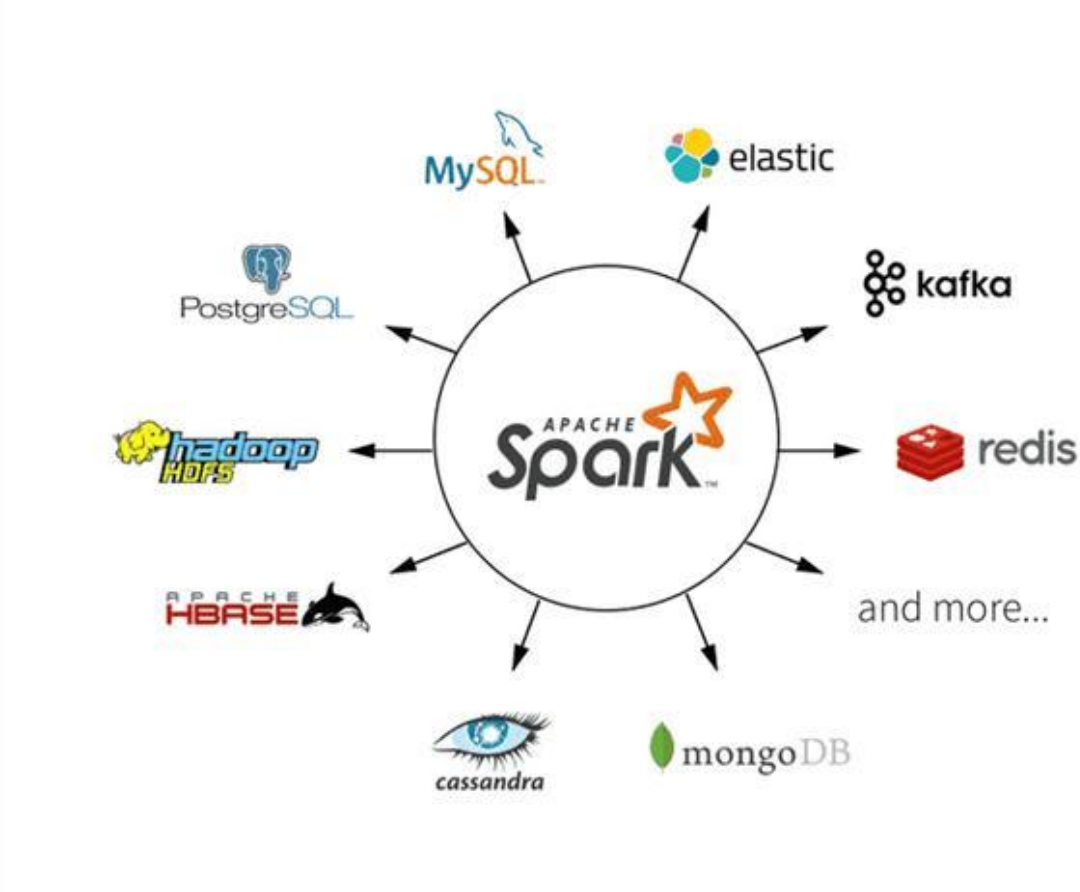
"Spark vs. Hadoop" is a commonly searched topic, but as previously mentioned, Spark is more of an enhancement to Hadoop, particularly to Hadoop's native data processing component, MapReduce. Spark is built on the MapReduce framework, and today, most Hadoop distributions come with Spark.

Similar to Spark, MapReduce allows programmers to process massive datasets faster by dividing the data across large computer clusters. However, while MapReduce processes

data on disk, which adds read and write times that slow down processing, Spark performs calculations in memory, making it significantly faster. As a result, Spark can process data up to 100 times faster than MapReduce.

Spark's built-in APIs for multiple languages make it more user-friendly and accessible for developers compared to MapReduce, which is known for being harder to program. Additionally, unlike MapReduce, Spark can run stream-processing applications on Hadoop clusters using YARN, Hadoop's resource management and job scheduling framework. Spark also adds powerful capabilities such as MLlib, GraphX, and SparkSQL. Moreover, Spark can handle data from other sources outside of the Hadoop ecosystem, like Apache Kafka.

Despite these differences, Spark is fully compatible with Hadoop and complements it well. It can process data from various Hadoop components, including HDFS (Hadoop Distributed File System), HBase (a NoSQL database running on HDFS), Apache Cassandra (a NoSQL alternative), and Hive (a Hadoop-based data warehouse).



Snowflake

Imagine having a magic box where all your business data — sales numbers, customer feedback, everything — is safely stored and easily accessible whenever you need it.

That's essentially what Snowflake offers for large organizations, but in the cloud. It provides a new way for businesses to store and use their data without getting bogged down by

complex technical details. Think of Snowflake as a vast, smart library where companies can easily store their "books" (data) and access them to make better-informed decisions.

What Is Snowflake?

Snowflake is more than just a cloud data warehouse; it’s a Software as a Service (SaaS) platform that transforms data storage, processing, and analytics for modern businesses. Snowflake provides a fully managed service that simplifies the management of data warehousing, data lakes, data engineering, data science, and the development of data-driven applications.

Core Innovation: Cloud Architecture

The key innovation behind Snowflake is its cloud architecture, which separates storage from compute. This decoupling enables exceptional scalability and performance optimization. Snowflake allows you to scale compute resources up or down in real-time without impacting data storage. This separation ensures efficient data management and analysis while supporting multiple concurrent workloads without any contention.

Flexibility for Cloud Engineers

Snowflake supports both structured and semi-structured data (such as JSON, Avro, and XML), streamlining data integration and analysis. The platform also includes features like data sharing, allowing secure access to shared datasets across different Snowflake accounts without the need to duplicate data. Additionally, Snowflake's support for data cloning and scalable compute resources enhances cost-efficiency and operational agility.

Compatibility and Security

Snowflake integrates smoothly with existing tools and provides robust security features, making it an ideal solution for modern cloud engineering challenges. To learn more about Snowflake’s pricing and how to optimize cost-effectiveness based on usage, check out our detailed article on [Understanding Snowflake Pricing].

How Does Snowflake Work?

Snowflake's Data Cloud provides a faster and more efficient alternative to traditional data models, offering a highly flexible approach to modern data management. Unlike legacy

databases or frameworks like Hadoop, Snowflake combines a powerful SQL query engine with a cloud-optimized architecture.

Self-Managed Service

Snowflake simplifies data management by eliminating the need for hardware and software overhead. Operating entirely on cloud infrastructure, it removes the requirements for installation, configuration, maintenance, and tuning. This self-managed service enables cloud engineers to concentrate on strategic tasks instead of routine operational upkeep.

Snowflake Architecture

Snowflake's architecture combines the best features of both shared-disk and shared-nothing systems. It uses a central data repository that stores persisted data, which is accessible across compute nodes, similar to shared-disk systems. Additionally, it employs a massively parallel processing (MPP) approach for query handling, akin to shared-nothing architectures, where each node holds a portion of the data set.

This hybrid approach ensures the simplicity of data management while delivering the scalability and performance of shared-nothing systems. The three core components of Snowflake's architecture are:

1. Database Storage

Snowflake converts data into an optimized, compressed columnar format during the loading process. It manages data organization, compression, and other crucial factors, ensuring efficient retrieval and high query performance through SQL queries.

2. Query Processing

Query processing occurs in virtual warehouses, which are MPP compute clusters made up of multiple compute nodes sourced from cloud providers. These virtual warehouses function

independently, ensuring that the performance of one does not impact the others. This design enables concurrent, high-performance data processing.

3. Cloud Services

The cloud services layer of Snowflake coordinates the platform's components to efficiently process user requests. It handles functions such as authentication, metadata management, query optimization, and access control.

4. Connecting to Snowflake

Snowflake offers multiple connectivity options, including a web UI for management, command-line tools like SnowSQL, ODBC/JDBC drivers, and native connectors for application development. These options enable seamless integration with a wide range of tools and platforms, boosting its effectiveness in diverse data engineering and analytics use cases.

Snowflake Use Cases

Snowflake's flexibility shines across several data management and analytic scenarios:

1. Data Ingestion

Snowpipe enables continuous data ingestion in Snowflake from sources such as S3 and Azure Blob, ensuring real-time data availability.

2. Business Intelligence and Analytics

Snowflake seamlessly integrates with top BI tools such as QuickSight, Looker, Power BI, and Tableau, enabling in-depth insights from complex datasets.

3. Data Sharing and Collaboration

Snowflake's Marketplace facilitates secure and effortless data sharing, promoting collaboration and strengthening data-driven strategies across businesses.

4. Machine Learning

Snowflake supports machine learning workflows by streamlining data preparation, model building, and deployment. It integrates with popular tools such as TensorFlow, PyTorch, and Apache Spark.

When Would You Use a Snowflake?

Snowflake is especially advantageous for organizations dealing with:

- **Rapid Data Growth**: Its scalable architecture accommodates growth without the need to manage physical infrastructure.
- **Complex Analytics Needs**: Snowflake enables high-performance query execution over large datasets with dynamic scaling.
- **Data Silos and Integration Challenges**: It simplifies data consolidation and integration through extensive connectivity options.
- **Real-Time Data Processing**: Snowflake is ideal for applications requiring continuous data analytics.
- **Stringent Data Security**: It offers robust security features for organizations with strict compliance requirements.
- **Operational Efficiency**: With self-managed, serverless options, Snowflake reduces the operational burden of data management.
- **Cost-Effective Strategy**: Its on-demand pricing model ensures flexibility and cost efficiency for variable workloads.

The Pros of Snowflake

- **Advanced Security:** Features such as Time Travel and Fail-safe provide strong data protection and ensure compliance.
- **Impressive Performance and Scalability:** Snowflake efficiently manages concurrent workloads with minimal latency and supports both vertical and horizontal scaling.
- **Data Caching and Micro-Partitions**: Intelligent caching and micro-partitions enhance query performance.
- **User-Friendly Interface:** Snowflake's SQL-based interface is intuitive, making it accessible for both beginners and experienced users.
- **Zero Management Overhead**: Its serverless architecture eliminates the complexity of maintenance.
- **Rich Ecosystem of Integrations:** Snowflake offers extensive connectivity options for a wide array of tools.

The Cons of Snowflake

- **Cost Management:** The on-demand pricing model requires careful monitoring, particularly for compute-intensive operations, to avoid unexpected costs.
- **Cloud-Agnostic Nature:** Organizations must assess how Snowflake fits into their existing cloud strategy when comparing it to native cloud solutions

How Does Snowflake Compare to Other Data Warehouses?

Snowflake vs. Apache Hadoop

- **Architecture & Scalability:** Snowflake's decoupled architecture offers more flexible scaling, while Hadoop relies on a distributed system for scaling.
- **Ease of Use & Security:** Snowflake provides an easier setup with built-in security features, whereas Hadoop demands deep technical expertise for configuration and security management.
- **Suitability:** Snowflake is best for those looking for a managed solution, whereas Hadoop is better suited for organizations needing a highly customizable, open-source framework.

Snowflake vs. Databricks

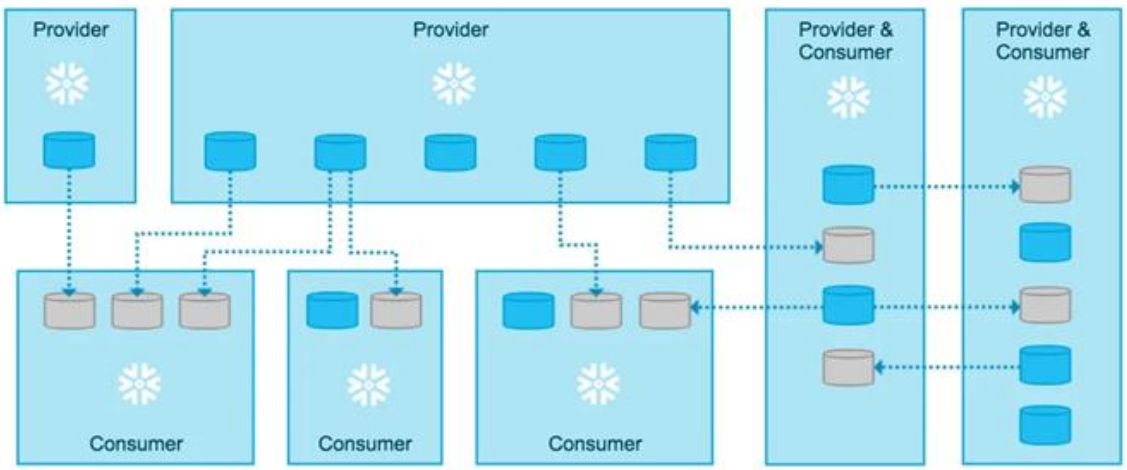
- **Architecture:** Databricks integrates data lakes with warehouses, while Snowflake provides a pure cloud data warehouse solution.
- **Ease of Use:** Snowflake's serverless model is more user-friendly, while Databricks requires expertise in managing Spark.
- **Suitability**: Databricks is ideal for Spark-intensive workflows, whereas Snowflake excels in scalability and streamlined data management**.**

Snowflake vs. AWS Amazon Redshift

- **Performance:** Both offer strong performance, but Snowflake's decoupled storage and compute architecture provide better scalability.
- **Serverless Options**: Snowflake has the edge with dynamic scalability, while Redshift follows a more AWS-centric model.
- **Integration**: Redshift integrates seamlessly with AWS services, while Snowflake offers greater flexibility across multiple cloud platforms.

Introducing Snowflake Cost Intelligence by CloudZero

CloudZero presents Snowflake Cost Intelligence, a platform designed to deliver detailed insights into the costs associated with Snowflake and AWS services. This solution enables organizations to connect costs with specific architectural decisions, encouraging cost-aware engineering practices. With CloudZero, businesses can identify precise cost areas and optimize them for maximum efficiency.



Overview of the dataset

The dataset consists of 10-minute interval recordings collected over approximately 4.5 months, capturing energy usage and environmental conditions in a low-energy building. Indoor temperature and humidity measurements were obtained using a ZigBee wireless sensor network, with each node transmitting data roughly every 3.3 minutes. These readings were then averaged into 10-minute intervals to align with the energy consumption data, which was logged every 10 minutes via m-bus energy meters. External weather conditions were incorporated from the nearest airport weather station (Chievres Airport, Belgium), sourced from the public dataset Reliable Prognosis (rp5.ru) and merged with the experimental data using timestamps. Additionally, two random variables were included to assist in testing regression models and identifying non-predictive features. This comprehensive dataset supports analysis of appliance energy use in relation to both indoor and outdoor environmental factors.

This dataset contains experimental data used to create regression models for predicting appliance energy consumption in a low-energy building. The data is multivariate (contains multiple features) and is structured as a time-series, meaning measurements were taken sequentially over time.

- **Subject Area:** Computer Science (with applications in energy efficiency, smart buildings, and machine learning).
- **Associated Task:** Regression (predicting continuous energy consumption values).
- **Feature Type:** All features are real-valued (numeric).
- **Number of Instances:** 19,735 (data points).
- **Number of Features:** 28 (including predictors and target variables).

Variables Table

Variable Name	Role	Type	Description	Units	Missing Values
date	Feature	Date			no
Appliances	Target	Integer		Wh	no
lights	Feature	Integer		Wh	no
T1	Feature	Continuous		C	no
RH_1	Feature	Continuous		%	no
T2	Feature	Continuous		C	no
RH_2	Feature	Continuous		%	no
T3	Feature	Continuous		C	no
RH_3	Feature	Continuous		%	no
T4	Feature	Continuous		C	no

Variable Information:

Time Variables

1. date: Date (year-month-day).
2. time: Time of day (hour:minute:second).

Energy Consumption

1. Appliances: Energy use by appliances (Wh).
2. lights: Energy use by lighting fixtures (Wh).

Indoor Temperature & Humidity (by Room)

1. T1, RH_1: Kitchen temp (°C) & humidity (%).
2. T2, RH_2: Living room temp (°C) & humidity (%).
3. T3, RH_3: Laundry room temp (°C) & humidity (%).
4. T4, RH_4: Office room temp (°C) & humidity (%).
5. T5, RH_5: Bathroom temp (°C) & humidity (%).
6. T7, RH_7: Ironing room temp (°C) & humidity (%).
7. T8, RH_8: Teenager room 2 temp (°C) & humidity (%).
8. T9, RH_9: Parents room temp (°C) & humidity (%).

Outdoor Measurements

1. T6, RH_6: Outside building (north side) temp (°C) & humidity (%).
2. To, RH_out: Weather station temp (°C) & humidity (%).
3. Pressure: Atmospheric pressure (mm Hg).
4. Wind speed: Air velocity (m/s).
5. Visibility: Horizontal visibility (km).
6. Tdewpoint: Dew point temperature (°C).

Miscellaneous

1. rv1, rv2: Nondimensional random variables (noise/synthetic).

TASK 02:

Import Libraries in Python

Import all necessary Libraries

```
import pandas as pd
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.inspection import permutation_importance
from sklearn.feature_selection import RFE
import joblib
import warnings
warnings.filterwarnings('ignore')
```

Core Libraries

1. pandas (pd): Data manipulation (DataFrames/Series), cleaning, and analysis
2. numpy (np): Numerical computations, array operations, and linear algebra
3. Datetime: Handling date/time data and conversions

Visualization

1. matplotlib.pyplot (plt): Basic plotting (line, bar, scatter, histograms)
2. seaborn (sns): Advanced statistical visualizations (heatmaps, distributions)
3.

Machine Learning (scikit-learn)

1. Train_test_split: Split data into training and test sets
2. GridSearchCV, RandomizedSearchCV: Hyperparameter tuning for models
3. Cross_val_score: Evaluate model performance via cross-validation
4. RandomForestRegressor, GradientBoostingRegressor: Tree-based ensemble regression models
5. SVR: Support Vector Machine for regression
6. LinearRegression: Ordinary least squares linear regression
7. mean_squared_error, mean_absolute_error, r2_score
8. Model evaluation metrics
9. OneHotEncoder, LabelEncoder: Encode categorical features into numerical values
10. Permutation_importance: Compute feature importances for models
11. RFE (Recursive Feature Elimination): Select features by recursively eliminating least important ones

Utilities

1. Joblib: Save and load trained models efficiently
2. Warnings: Control warning messages (e.g., suppress non-critical warnings)

Typical Usage Flow

1. Data Prep: pandas + numpy
2. Visualization: matplotlib + seaborn
3. Modeling: scikit-learn (train/evaluate/tune)
4. Deployment: joblib to save models

Load the data;

Import the Dataset

```
[31] data = pd.read_csv('/content/drive/MyDrive/energydata_complete.csv')
```

Data Preprocessing

Format the date and time column

```
[ ]  data['date'] = pd.to_datetime(data['date'], format='%Y-%m-%d %H:%M:%S')
     print(data['date'].dtype)

     datetime64[ns]
```

This code ensures that the date column in your dataset is treated as actual date and time data by Python. This is essential for tasks like time series analysis, we might want to plot trends over time, extract specific time components (hours, days, etc.), or perform date-based calculations.

Display the first few rows of the dataset

```
# Display the first few rows of the dataset
print(data.head())
```

```
                 date  Appliances  lights     T1       RH_1    T2       RH_2  \
0 2016-01-11 17:00:00          60      30  19.89  47.596667  19.2  44.790000
1 2016-01-11 17:10:00          60      30  19.89  46.693333  19.2  44.722500
2 2016-01-11 17:20:00          50      30  19.89  46.300000  19.2  44.626667
3 2016-01-11 17:30:00          50      40  19.89  46.066667  19.2  44.590000
4 2016-01-11 17:40:00          60      40  19.89  46.333333  19.2  44.530000

      T3       RH_3         T4  ...         T9   RH_9     T_out  Press_mm_hg  \
0  19.79  44.730000  19.000000  ...  17.033333  45.53  6.600000        733.5
1  19.79  44.790000  19.000000  ...  17.066667  45.56  6.483333        733.6
2  19.79  44.933333  18.926667  ...  17.000000  45.50  6.366667        733.7
3  19.79  45.000000  18.890000  ...  17.000000  45.40  6.250000        733.8
4  19.79  45.000000  18.890000  ...  17.000000  45.40  6.133333        733.9

   RH_out  Windspeed  Visibility  Tdewpoint        rv1        rv2
0    92.0   7.000000   63.000000        5.3  13.275433  13.275433
1    92.0   6.666667   59.166667        5.2  18.606195  18.606195
2    92.0   6.333333   55.333333        5.1  28.642668  28.642668
3    92.0   6.000000   51.500000        5.0  45.410389  45.410389
4    92.0   5.666667   47.666667        4.9  10.084097  10.084097

[5 rows x 29 columns]
```

This specific line of code is used to display the first few rows of the dataset that you loaded into the variable called Data.

Creating New Features from Datetime

```
[ ]  # creating hour and day of the week features from datetime
     if 'date' in data.columns:
         data['hour'] = data['date'].dt.hour
         data['day_of_week'] = data['date'].dt.dayofweek
     elif 'datetime' in data.columns:
         data['hour'] = data['datetime'].dt.hour
         data['day_of_week'] = data['datetime'].dt.dayofweek
```

This code snippet extracts and creates two new features— **hour of the day** and **day of the week** —from an existing date or datetime column in your dataset. By converting the timestamp into these two separate columns (`hour` and `day_of_week`), the code makes it easier to analyze time-based patterns in the data. For example, you can now examine whether certain trends, behaviors, or events vary by specific hours or days, which is useful for time series analysis, customer behavior modeling, or operational efficiency studies.

Descriptive statistics for numerical features

```
# Generate descriptive statistics for numerical features
print(data.describe())
```

```
                       date    Appliances        lights            T1  \
count                 19735  19735.000000  19735.000000  19735.000000
mean    2016-03-20 05:30:00     97.694958      3.801875     21.686571
min     2016-01-11 17:00:00     10.000000      0.000000     16.790000
25%     2016-02-14 23:15:00     50.000000      0.000000     20.760000
50%     2016-03-20 05:30:00     60.000000      0.000000     21.600000
75%     2016-04-23 11:45:00    100.000000      0.000000     22.600000
max     2016-05-27 18:00:00   1080.000000     70.000000     26.260000
std                     NaN    102.524891      7.935988      1.606066

                RH_1           T2          RH_2           T3          RH_3  \
count   19735.000000  19735.000000  19735.000000  19735.000000  19735.000000
mean       40.259739     20.341219     40.420420     22.267611     39.242500
min        27.023333     16.100000     20.463333     17.200000     28.766667
25%        37.333333     18.790000     37.900000     20.790000     36.900000
50%        39.656667     20.000000     40.500000     22.100000     38.530000
75%        43.066667     21.500000     43.260000     23.290000     41.760000
max        63.360000     29.856667     56.026667     29.236000     50.163333
std         3.979299      2.192974      4.069813      2.006111      3.254576

                T4  ...           T9          RH_9         T_out  \
count   19735.000000  ...  19735.000000  19735.000000  19735.000000
mean       20.855335  ...     19.485828     41.552401      7.411665
min        15.100000  ...     14.890000     29.166667     -5.000000
25%        19.530000  ...     18.000000     38.500000      3.666667
50%        20.666667  ...     19.390000     40.900000      6.916667
75%        22.100000  ...     20.600000     44.338095     10.408333
max        26.200000  ...     24.500000     53.326667     26.100000
std         2.042884  ...      2.014712      4.151497      5.317409

         Press_mm_hg        RH_out     Windspeed    Visibility     Tdewpoint  \
count   19735.000000  19735.000000  19735.000000  19735.000000  19735.000000
mean      755.522602     79.750418      4.039752     38.330834      3.760707
min       729.300000     24.000000      0.000000      1.000000     -6.600000
25%       750.933333     70.333333      2.000000     29.000000      0.900000
50%       756.100000     83.666667      3.666667     40.000000      3.433333
75%       760.933333     91.666667      5.500000     40.000000      6.566667
max       772.300000    100.000000     14.000000     66.000000     15.500000
std         7.399441     14.901088      2.451221     11.794719      4.194648
```

```
                rv1           rv2
count  19735.000000  19735.000000
mean      24.988033     24.988033
min        0.005322      0.005322
25%       12.497889     12.497889
50%       24.897653     24.897653
75%       37.583769     37.583769
max       49.996530     49.996530
std       14.496634     14.496634
```

This line of code is using the describe() method in Pandas to generate descriptive statistics for the numerical features in the dataset. This line is responsible for displaying a concise summary of the dataset stored in the data variable, which is a Pandas DataFrame.

creating hour and day of the week features from datetime

```
# creating hour and day of the week features from datetime
if 'date' in data.columns:
    data['hour'] = data['date'].dt.hour
    data['day_of_week'] = data['date'].dt.dayofweek
elif 'datetime' in data.columns:
    data['hour'] = data['datetime'].dt.hour
    data['day_of_week'] = data['datetime'].dt.dayofweek
```

Check for missing values in each column

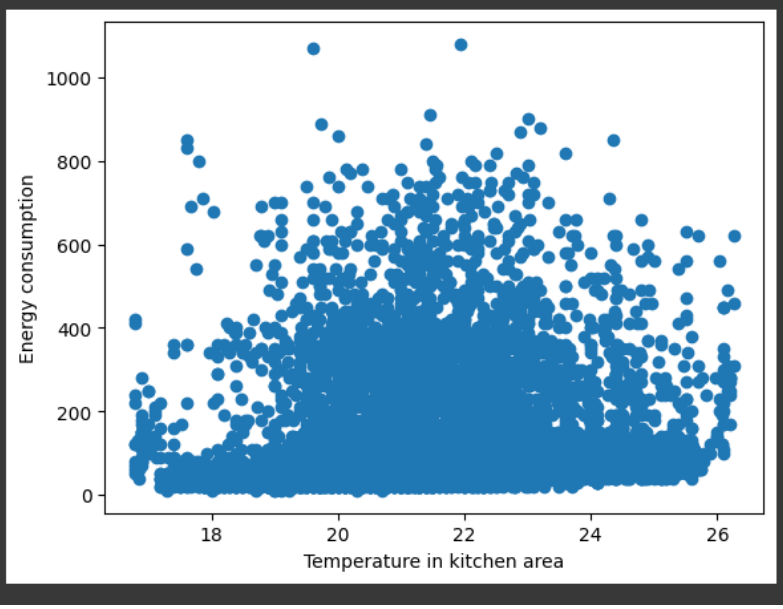
```
[ ] # Check for missing values in each column
    print(data.isnull().sum())
```

```
date           0
Appliances     0
lights         0
T1             0
RH_1           0
T2             0
RH_2           0
T3             0
RH_3           0
T4             0
RH_4           0
T5             0
RH_5           0
T6             0
RH_6           0
T7             0
RH_7           0
T8             0
RH_8           0
T9             0
RH_9           0
T_out          0
Press_mm_hg    0
RH_out         0
Windspeed      0
Visibility     0
Tdewpoint      0
rv1            0
rv2            0
dtype: int64
```

This specific line is designed to identify if there are any missing values in your dataset. It's a crucial step in data cleaning and preprocessing

Scatter Plot

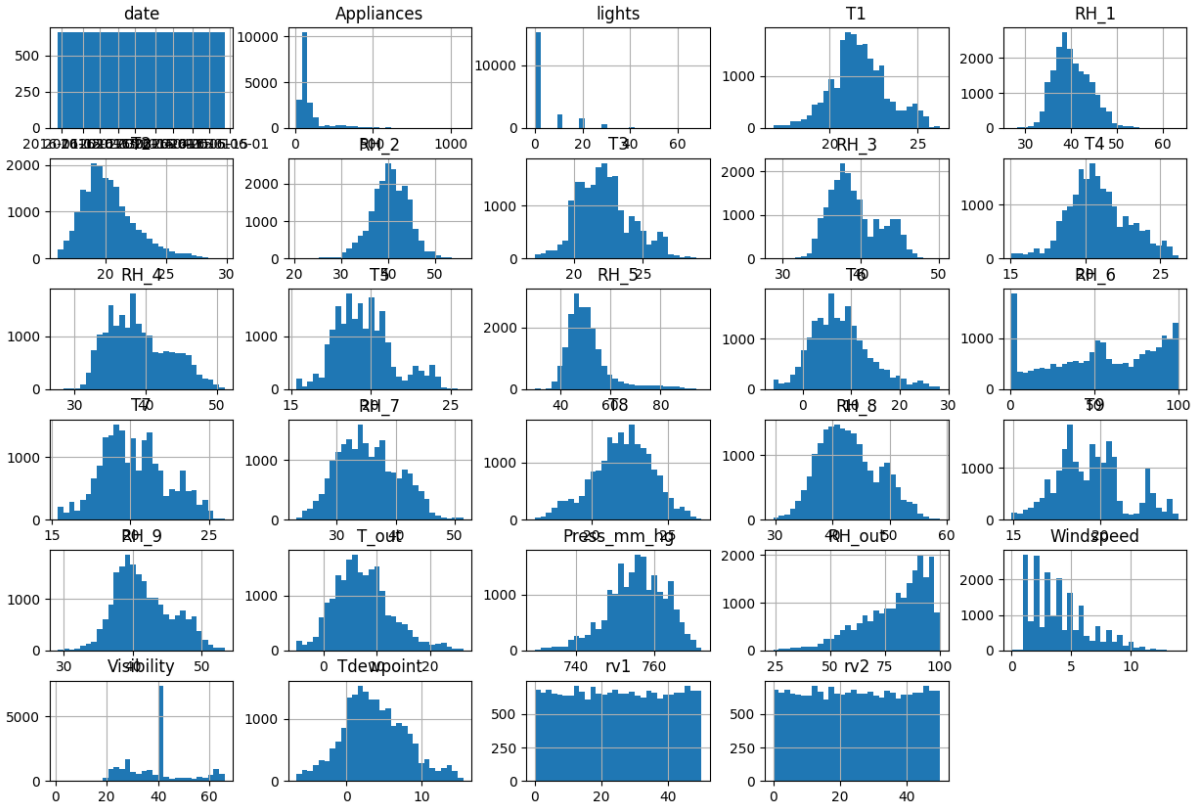
```
# Explore the relationship between specific features using scatter plots
# Relationship between temperature and energy consumption
plt.scatter(data['T1'], data['Appliances'])
plt.xlabel('Temperature in kitchen area')
plt.ylabel('Energy consumption')
plt.show()
```



Adding these structured time components enhances the dataset's usability, allowing for more granular insights without altering the original datetime values.

Explore the distribution of numerical features using histograms

```
# Explore the distribution of numerical features using histograms
data.hist(bins=30, figsize=(15, 10))
plt.show()
```



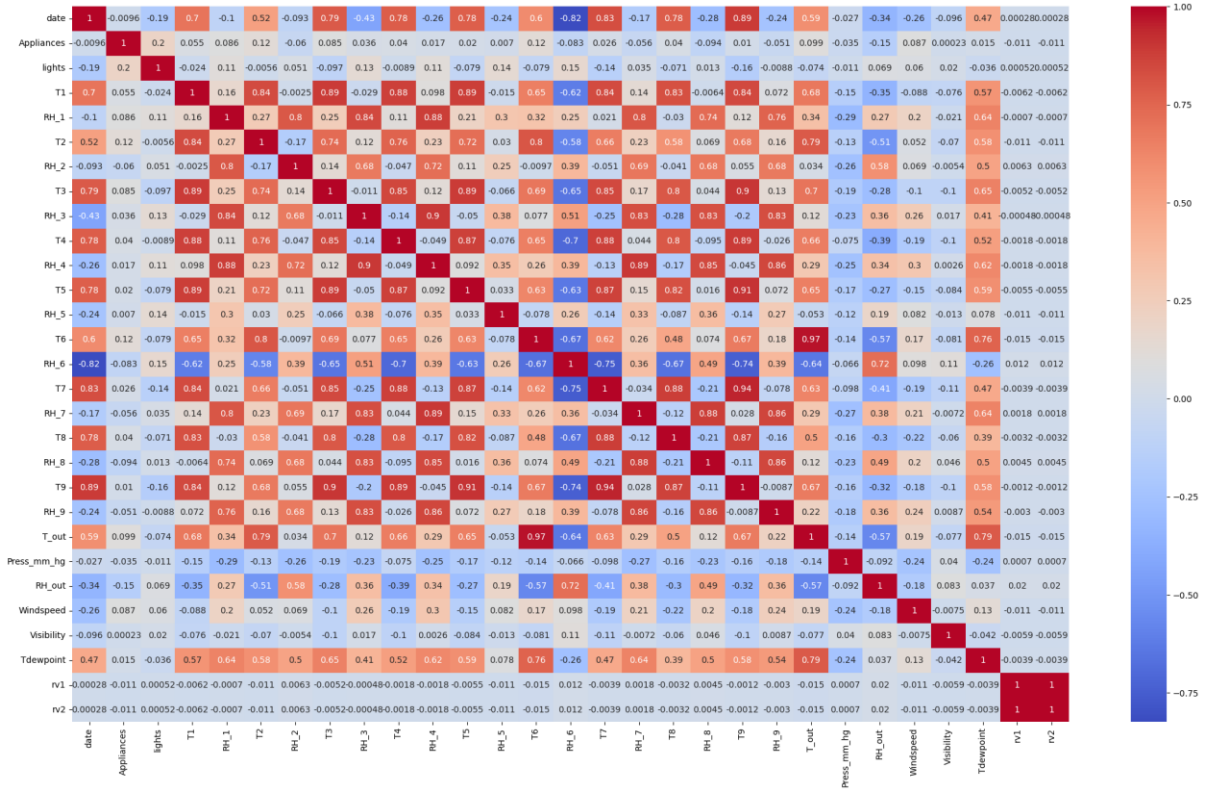
This code snippet is designed to visually explore the distribution of numerical features (columns with numbers) in this dataset using histograms. Each histogram would show the frequency of different price ranges, bedroom counts, and so on. This visualization allows you to quickly understand the distribution of your data—whether it's concentrated in a specific range, skewed, or evenly spread out.

Analyzing Correlation with a Heatmap

```
# Analyze the correlation between features using a heatmap
correlation_matrix = data.corr()
plt.figure(figsize=(15, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.show()
```

This section aims to visualize the correlation between different features (columns) in your dataset using a heatmap. Correlation coefficients measure the strength and direction of a linear relationship between two variables, ranging from +1 to -1. A correlation of +1 indicates a perfect positive correlation, meaning that as one variable increases, the other variable

increases proportionally. Conversely, a correlation of -1 represents a perfect negative correlation, where an increase in one variable leads to a proportional decrease in the other.



A correlation of 0 suggests no linear relationship between the variables, indicating that changes in one variable do not predict changes in the other in a linear fashion. These values help quantify how closely two variables move in relation to each other, providing valuable insights in fields such as statistics, economics, and scientific research.

The analysis of relative humidity (RH) and appliance consumption reveals several correlations. In the kitchen area (RH1), there is a small positive correlation (0.06) with appliance consumption, likely due to increased humidity from cooking activities and human presence. Similarly, a positive correlation of 0.04 between appliances' consumption and RH3 (laundry room humidity) suggests that the washing machine and/or dryer may be in use. Additionally, minor positive correlations of 0.02 and 0.01 were observed between appliance consumption and RH4 (downstairs office) as well as RH5 (upstairs bathroom), respectively.

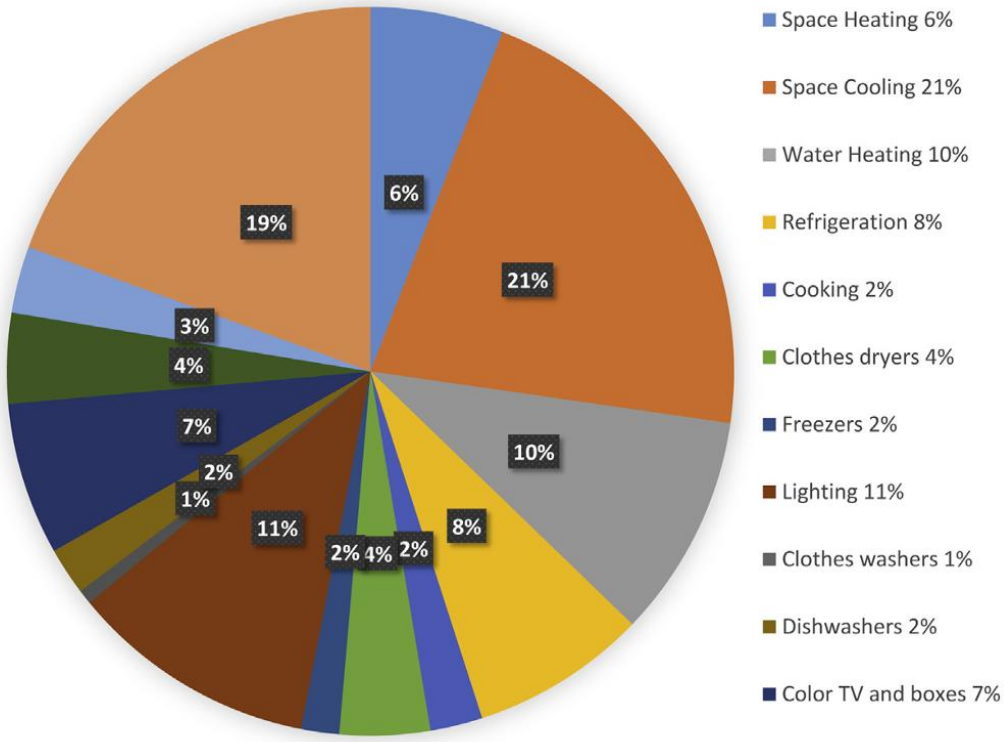
Conversely, a negative correlation (-0.06) was found between appliance consumption and RH2 (living room humidity), indicating lower energy use in this area compared to the kitchen and laundry room. The three upstairs bedrooms also showed negative correlations with appliances' consumption, with RH6 at -0.06, RH8 at -0.09, and RH9 at -0.05. This suggests that although human presence increases humidity, energy consumption in these rooms remains low, possibly due to lower power demands or occupants being asleep.

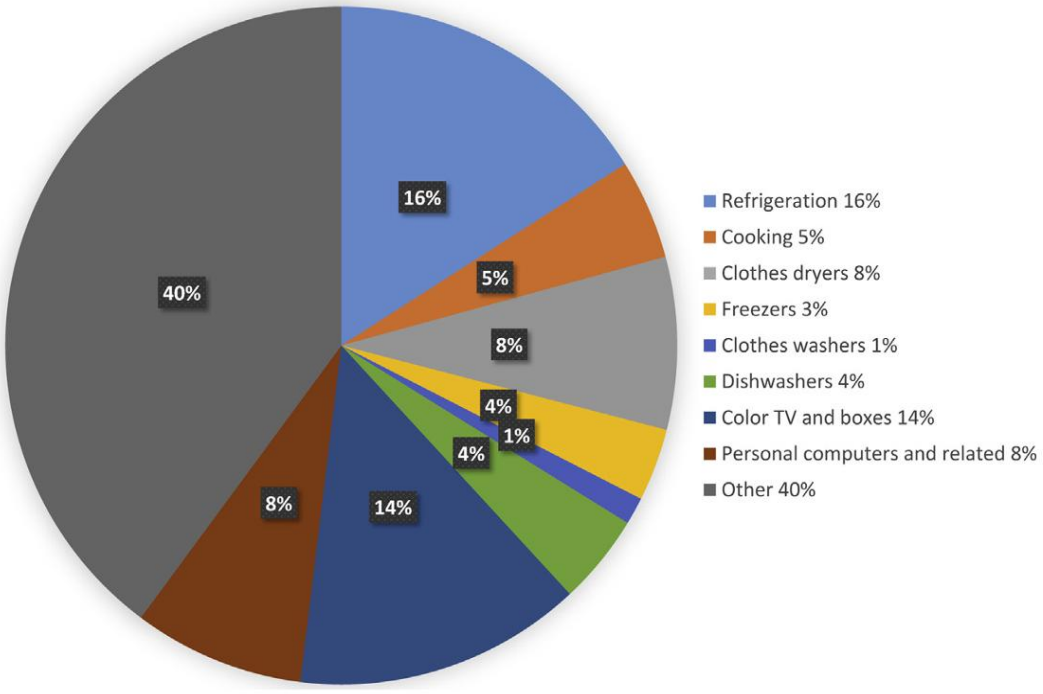
Beyond humidity, a small negative correlation (-0.03) was noted between appliances' consumption and atmospheric pressure, while a positive correlation (0.09) was found between appliance consumption and wind speed. Additionally, pressure and wind speed

exhibited a negative correlation (-0.23), meaning that lower pressure corresponds with higher wind speeds. These relationships may be influenced by changes in building occupancy due to weather conditions, as occupancy strongly affects energy consumption.

Creating a Pie Chart

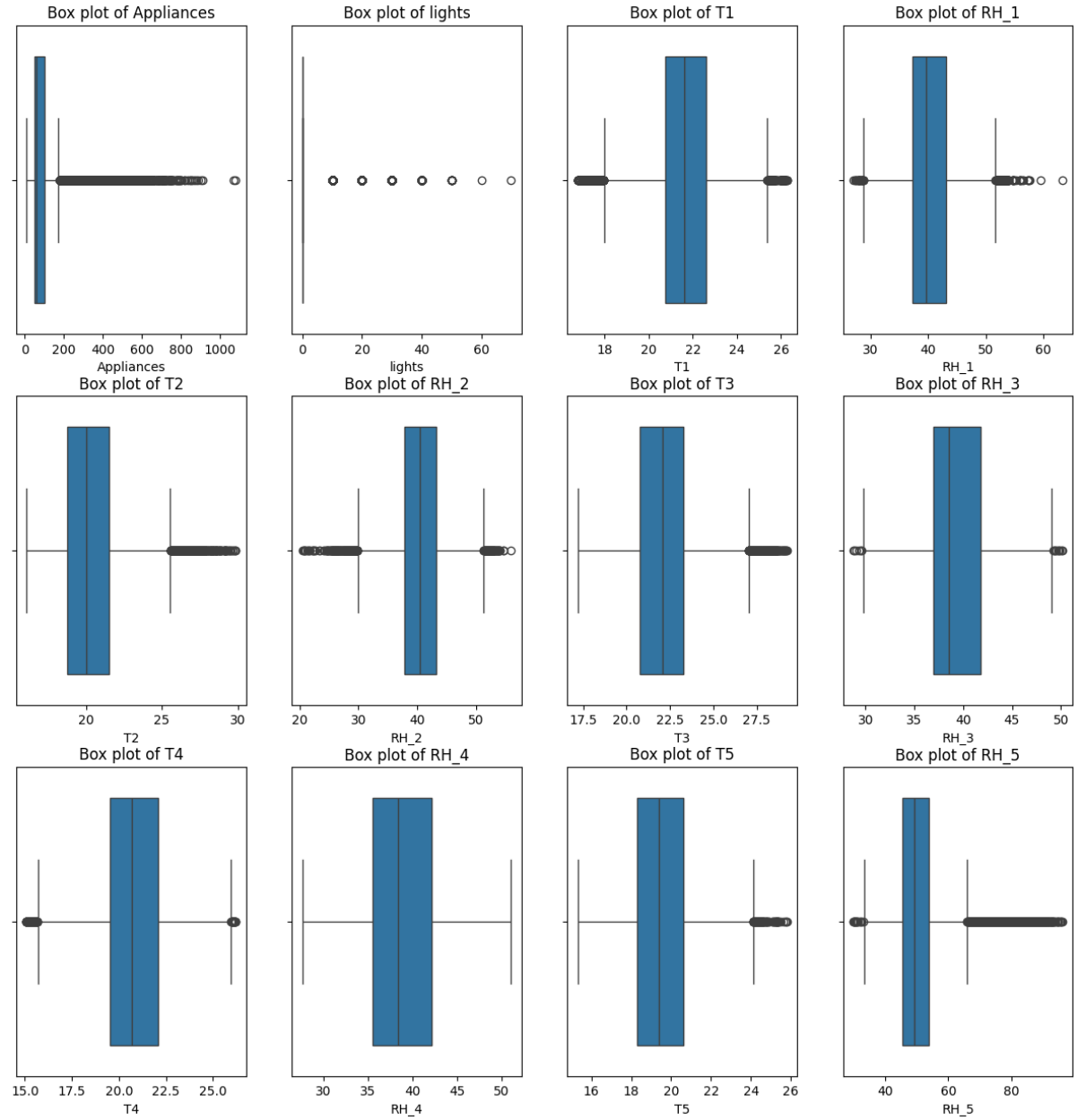
```
# Create the pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

# Add a title
plt.title('Sample Pie Chart')

# Display the chart
plt.show()
```





Identifying Outliers with Box Plots

```python
# Box plots to identify outliers
plt.figure(figsize=(15, 15))  # Adjust figure size as needed

for i, column in enumerate(data.select_dtypes(include=np.number).columns):
    plt.subplot(3, 4, i + 1)  # Adjust the grid size (rows, cols) as needed
    sns.boxplot(x=data[column])
    plt.title(f"Box plot of {column}")

plt.tight_layout()  # Adjust subplot parameters for a tight layout
plt.show()
```

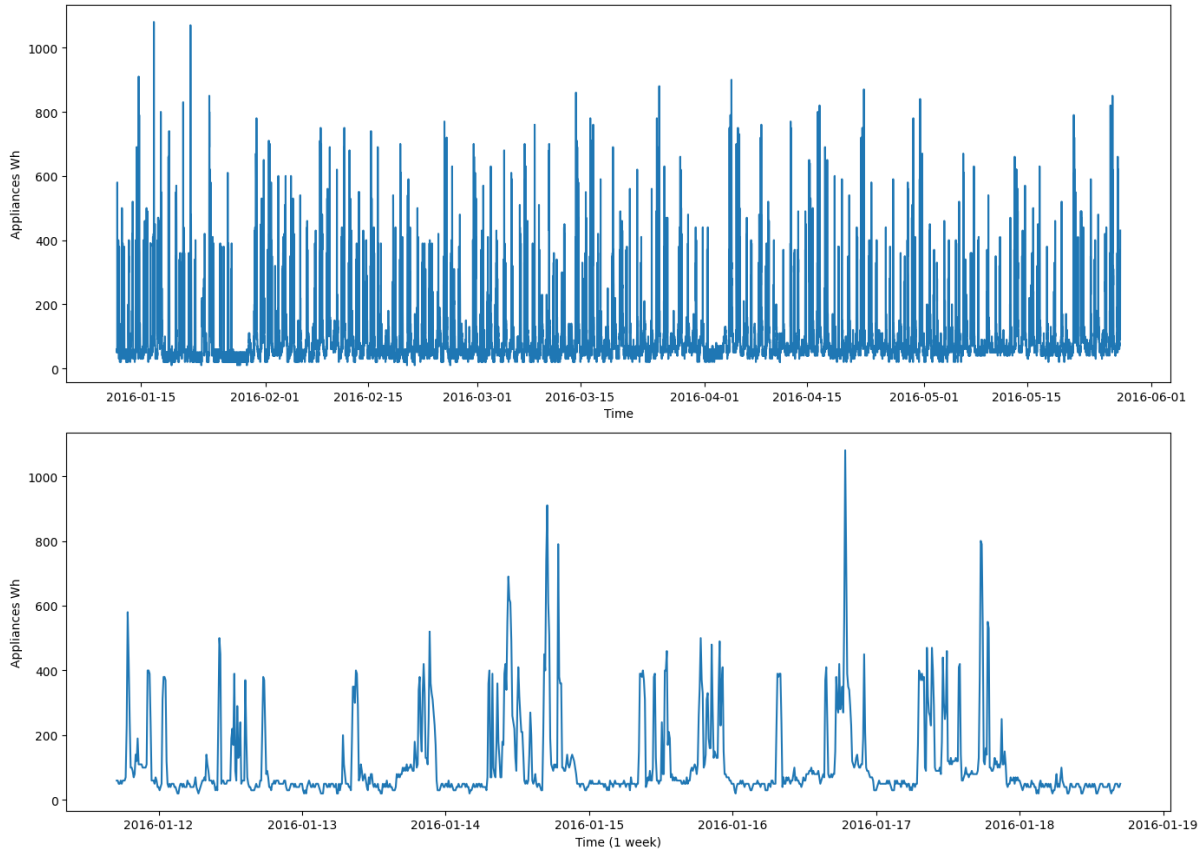


This code generates a series of box plots, with each plot corresponding to a numerical feature in your dataset. Box plots are a powerful visualization tool for understanding the distribution of data, including measures like median, quartiles, and potential outliers. By displaying the spread and skewness of each variable, these plots help identify anomalies, extreme values, or unusual patterns that may require further investigation. Detecting outliers is particularly important in data preprocessing, as they can significantly impact statistical

analyses and machine learning model performance. This visualization step is often used in exploratory data analysis (EDA) to guide decisions on data cleaning, transformation, or feature engineering before proceeding with modeling.

Data Visualization:

```
plt.figure(figsize=(14, 10))
plt.subplot(2, 1, 1)
plt.plot(data['date'], data['Appliances'])
plt.xlabel('Time')
plt.ylabel('Appliances Wh')

plt.subplot(2, 1, 2)
plt.plot(data['date'].iloc[:1008], data['Appliances'].iloc[:1008])
plt.xlabel('Time (1 week)')
plt.ylabel('Appliances Wh')
plt.tight_layout()
```



Top Plot: Shows the energy consumption of appliances ('Appliances Wh') over the entire time period in the dataset.

Bottom Plot: Focuses on the energy consumption for the first 1008 data points (probably representing one week of data).

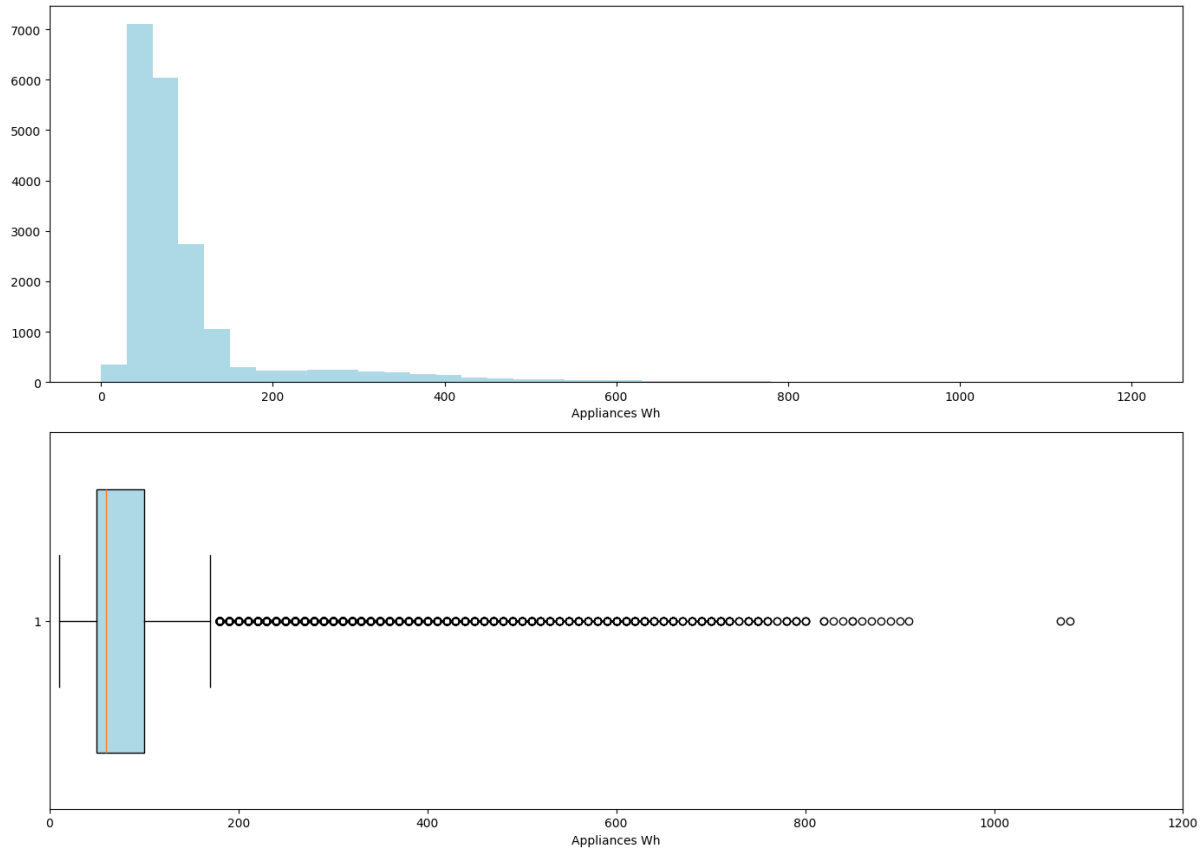
We can learn a lot about usage trends and possible affecting factors by using two subplots to visualize energy consumption data. Long-term behaviour is shown by the general trend map, which highlights cycles, significant variations, and rises and declines in energy use over the whole dataset timeframe. A closer look is given by the weekly plot, which helps us spot daily or weekly trends such changes by day of the week or peak usage periods. It is

easier to distinguish anomalies from normal occurrences when these patterns are compared to the general trend. When combined, these charts aid in the comprehension of the dynamics of energy use, aiding in the detection of anomalies, the identification of significant variables, and the creation of prediction models for energy optimization.

The dataset shows the corresponding energy usage profile for a period of 137 days, or 4.5 months. Significant variations in energy usage patterns are reflected in this profile over the course of the monitoring period. A number of variables, including seasonal shifts, occupancy trends, and electrical appliance operating schedules, could be responsible for the observed differences. The significant degree of consumption variability emphasizes how dynamic energy use is in the ecosystem under study and how reliable modelling techniques are required to properly capture these trends.

Histogram and Boxplot of Appliances

```
plt.figure(figsize=(14, 10))
plt.subplot(2, 1, 1)
plt.hist(data['Appliances'], bins=40, color='lightblue', range=(0, 1200))
plt.xlabel('Appliances Wh')

plt.subplot(2, 1, 2)
plt.boxplot(data['Appliances'], vert=False, patch_artist=True,
            boxprops=dict(facecolor='lightblue'), widths=0.7)
plt.xlim(0, 1200)
plt.xlabel('Appliances Wh')
plt.tight_layout()
```



The distribution of data points above the median in the box plot visualization indicates a right-skewed distribution with a long tail. Within the blue rectangle, a thick black line at 60 Wh indicates the median energy usage, with whiskers stretching from 10 Wh (lower) to 170 Wh (higher). Significantly, there are more outliers and more dispersion in the data above the median, which are indicated by circular markers that extend above the top whisker. This pattern suggests sporadic periods of abnormally high energy use because, although the majority of observations cluster below 170 Wh, there are a few extreme values that considerably above this threshold.

HeatMap Visualization

Creating Hourly Aggregates for Energy Consumption Analysis

```
[42] # Create hourly aggregates
     data['mhr'] = data['date'].dt.floor('H')
     hourly_data = data.groupby('mhr')['Appliances'].sum().reset_index()
     hourly_data['Day_week'] = hourly_data['mhr'].dt.day_name()
     hourly_data['week_year'] = hourly_data['mhr'].dt.isocalendar().week
     hourly_data['Hour'] = hourly_data['mhr'].dt.hour
```

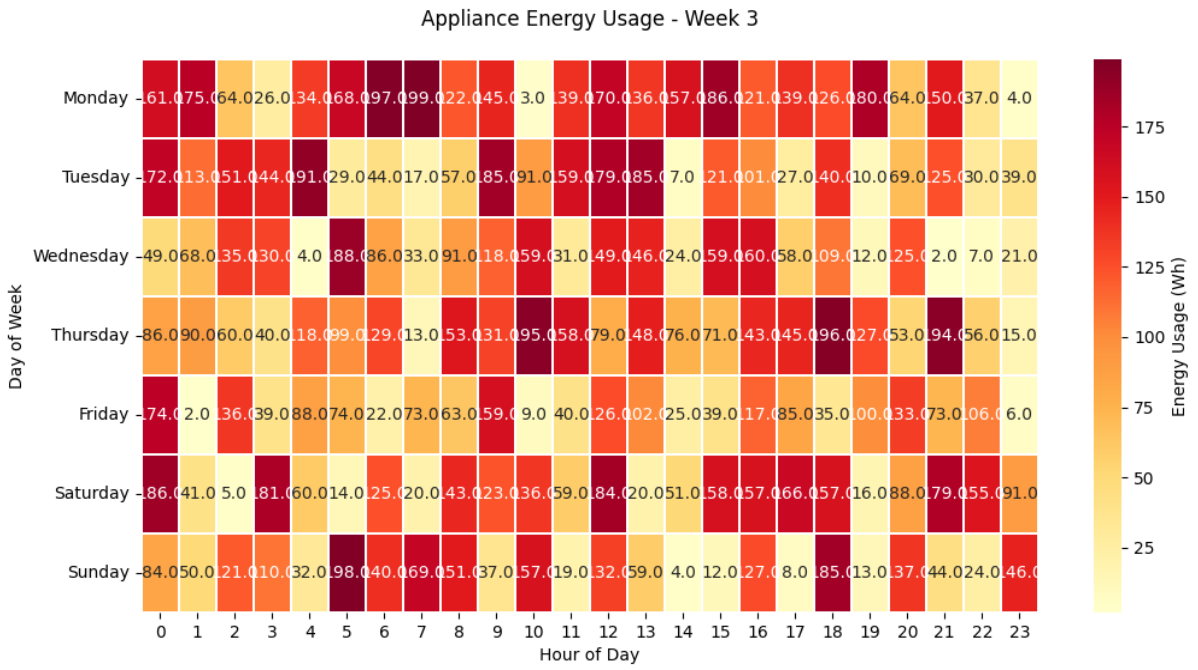
this code segment creates an hourly_data DataFrame that aggregates energy consumption by hour, adding information about the day of the week, the week of the year, and the hour of the day for further analysis and visualization

Create heatmaps for different weeks

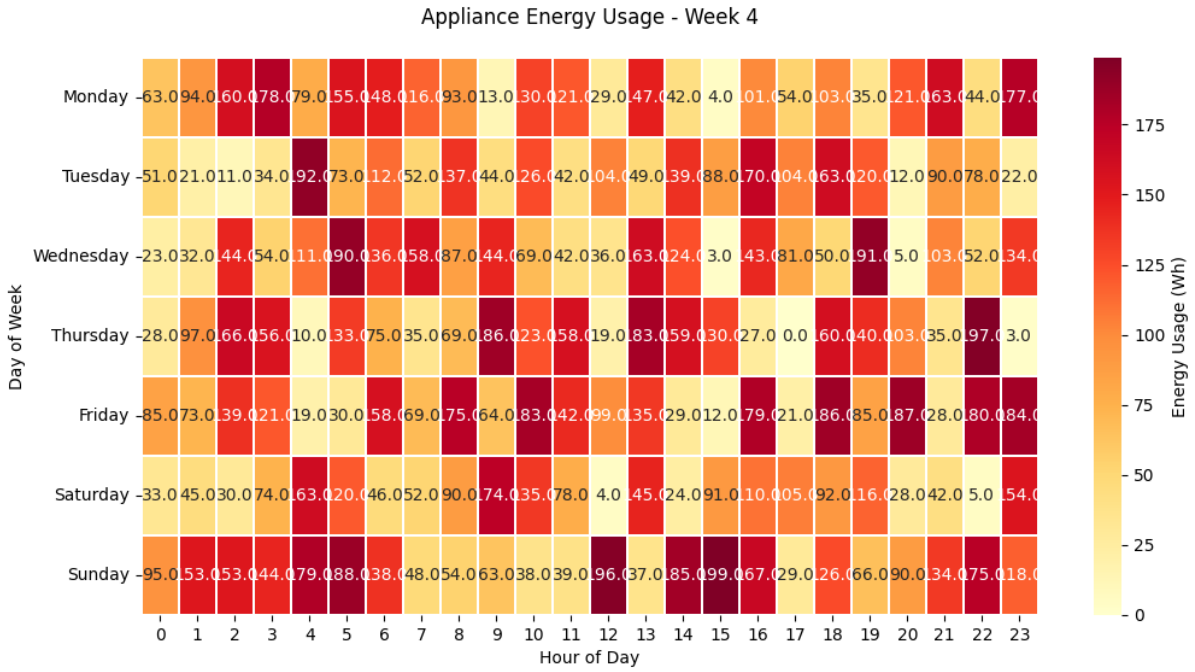
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

def create_heatmap(data, week_num):
    week_data = data[data['week_year'] == week_num]
    # Pivot with correct parameters: index, columns, values
    pivot_data = week_data.pivot(index="Day_week", columns="Hour", values="Appliances")

    # Ensure days are sorted (Monday to Sunday)
    days_order = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
    pivot_data = pivot_data.reindex(days_order)

    plt.figure(figsize=(12, 6))
    sns.heatmap(pivot_data, cmap="YlOrRd", linewidths=0.3, annot=True, fmt=".1f",
                cbar_kws={'label': 'Energy Usage (Wh)'})
    plt.title(f'Appliance Energy Usage - Week {week_num}', pad=20)
    plt.xlabel('Hour of Day')
    plt.ylabel('Day of Week')
    plt.show()
```

The code visualizes energy consumption patterns using heatmaps. It imports necessary libraries (pandas, seaborn, matplotlib), defines a function (create_heatmap) to generate heatmaps for specific weeks, creates sample data with date, time, and energy usage, and finally calls the function to display heatmaps for weeks 3, 4, 5, and 6, highlighting hourly and daily energy usage trends. These heatmaps help analyze energy consumption patterns over different weeks and identify potential areas for optimization.

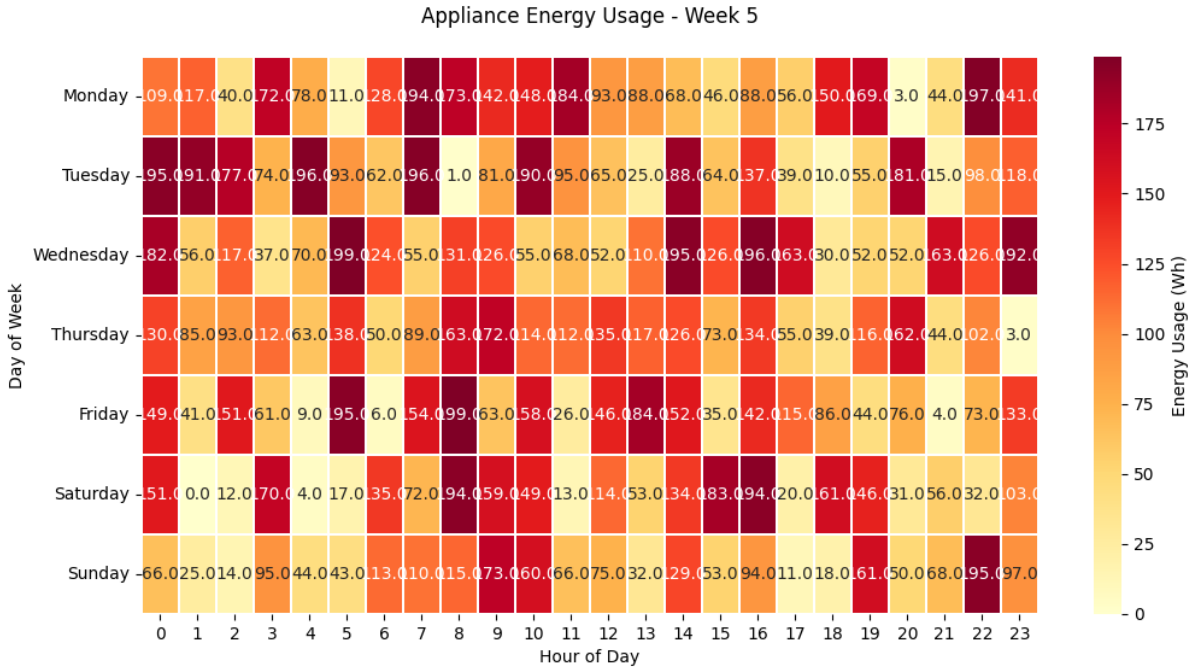
```python
import numpy as np
import pandas as pd
from datetime import datetime, timedelta

# Create sample data
num_weeks = 6  # Number of weeks to generate data for
days_per_week = 7
hours_per_day = 24
total_records = num_weeks * days_per_week * hours_per_day

# Generate date range
start_date = datetime(2023, 1, 1)
date_range = [start_date + timedelta(hours=i) for i in range(total_records)]

# Generate random appliance usage data
appliance_usage = np.random.randint(0, 200, size=total_records)  # Adjust range as needed

# Create DataFrame
data = pd.DataFrame({
    'date': date_range,
    'Appliances': appliance_usage
})

# Extract week_year, Day_week, Hour
data['week_year'] = data['date'].dt.isocalendar().week
data['Day_week'] = data['date'].dt.day_name()
data['Hour'] = data['date'].dt.hour

# Create heatmaps for specific weeks
create_heatmap(data, 3)
create_heatmap(data, 4)
create_heatmap(data, 5)
create_heatmap(data, 6)
```



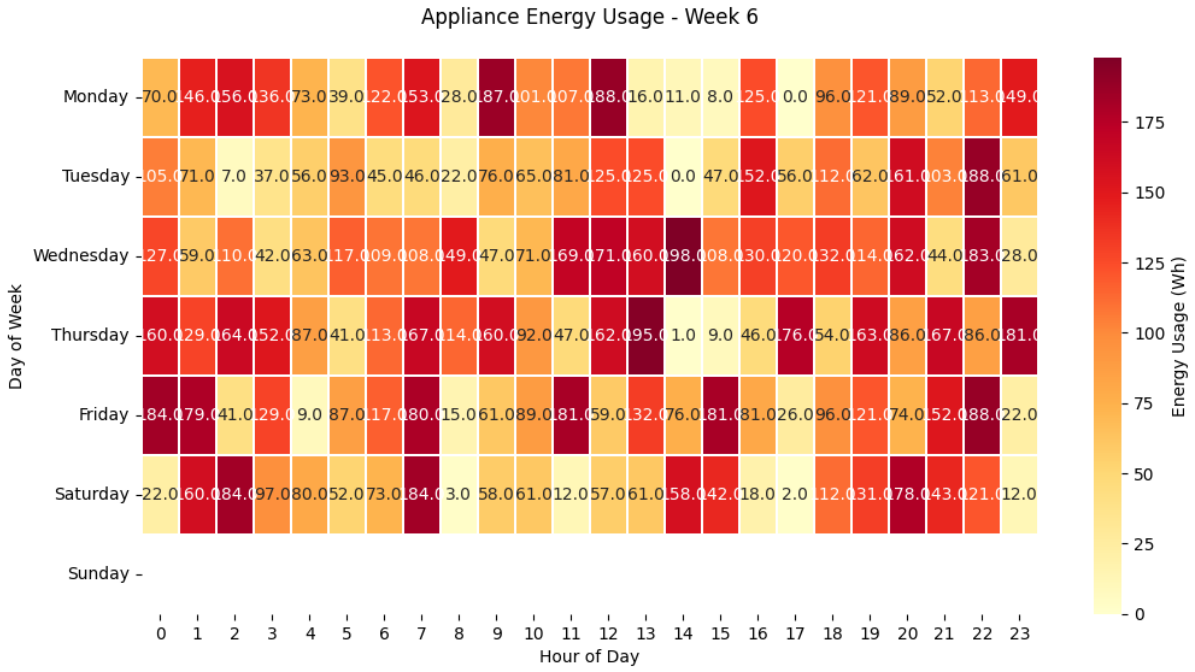
- **Week 3:** Analyze the energy consumption patterns during week 3, noting any unusual peaks or dips.



- **Week 4:** Compare the patterns in week 4 with week 3. Look for any significant changes or shifts in usage



- **Week 5:** Observe the trends in week 5 and compare them with the previous weeks. Identify any emerging patterns or consistent behaviors.



- **Week 6:** Finally, examine the heatmap for week 6, considering it in the context of the overall trends observed in the earlier weeks.

It is evident that the pattern of energy consumption has a significant time component. The consumption of energy begins to increase at approximately six in the morning. Then, there are increases in energy load around noon. Around 6 pm, there is also an upsurge in the need for energy.In terms of the day of the week, there is no discernible trend.

Data Splitting

```
# Remove unnecessary columns
data.drop(['mhr', 'rv1', 'rv2'], axis=1, inplace=True, errors='ignore')

# Split into train and test
train_data, test_data = train_test_split(data, test_size=0.25, random_state=1)

# Save datasets
train_data.to_csv("training.csv", index=False)
test_data.to_csv("testing.csv", index=False)
```

This section of code cleans the data, divides it into parts for training and testing a machine-learning model, and saves these parts as individual files. This is standard practice when preparing data for machine learning.

Prepare data for modelling

The code specifies two functions for assessing regression model performance: mape and r squared. In order to determine the average % difference between the expected and actual data, Mape computes the Mean Absolute % Error. The R-squared number, which indicates the percentage of the target variable's variation that the model can account for, is computed

using r squared. These functions help evaluate and compare various models by offering quantifiable indicators of model accuracy and goodness-of-fit.

```
# Define evaluation metrics
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

def rsquared(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)
```

```
[65] # Prepare data
X_train = train_data.drop(['date', 'Appliances'], axis=1)
y_train = train_data['Appliances']
X_test = test_data.drop(['date', 'Appliances'], axis=1)
y_test = test_data['Appliances']
```

this code snippet is creating the input (X) and output (y) data for both the training and testing sets. It separates the features from the target variable and makes sure the model doesn't see the test data during training, allowing for an unbiased evaluation of its performance..

Size of Training and Testing Data

```
[68] # Size of taining and testing data

print(f"Training data size: {len(train_data)}")
print(f"Testing data size: {len(test_data)}")
```

```
Training data size: 14801
Testing data size: 4934
```

Training : 14801
Testing : 4934

Modelling

Time Series Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from statsmodels.tsa.seasonal import seasonal_decompose


# 1. Data Preparation
ts_data = data[['date', 'Appliances']].copy()
ts_data['date'] = pd.to_datetime(ts_data['date'])
ts_data.set_index('date', inplace=True)

# Handle missing values and ensure frequency
ts_data = ts_data.asfreq('D').fillna(method='ffill')  # Daily frequency with forward fill
```

```python
# 2. Stationarity Check
def check_stationarity(series):
    result = adfuller(series)
    print(f'ADF Statistic: {result[0]}')
    print(f'p-value: {result[1]}')
    print('Critical Values:')
    for key, value in result[4].items():
        print(f'   {key}: {value}')

check_stationarity(ts_data['Appliances'])

# 3. Visual Diagnostics
# Original series
ts_data.plot(figsize=(12,6), title='Original Time Series')
plt.show()

# ACF/PACF plots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12,8))
plot_acf(ts_data['Appliances'], lags=40, ax=ax1)
plot_pacf(ts_data['Appliances'], lags=40, ax=ax2)
plt.show()

# Decomposition
decomposition = seasonal_decompose(ts_data['Appliances'], model='additive', period=24)  # Adjust period
decomposition.plot()
plt.show()
```

```python
# 4. Model Training
# Train-test split (80-20)
train_size = int(len(ts_data) * 0.75)
train_ts, test_ts = ts_data[:train_size], ts_data[train_size:]

# ARIMA Model - Start with simple orders
try:
    # Try auto-selecting orders (install pmdarima if needed: pip install pmdarima)
    from pmdarima import auto_arima
    auto_model = auto_arima(train_ts, seasonal=False, trace=True)
    print(auto_model.summary())
    order = auto_model.order
except:
    # Fallback to manual orders
    order = (1, 1, 1)  # Basic non-seasonal orders

print(f"\nUsing ARIMA order: {order}")

model = ARIMA(train_ts, order=order)
model_fit = model.fit()

# 5. Forecasting

# Forecast on test set
forecast_steps = len(test_ts)
forecast = model_fit.get_forecast(steps=forecast_steps)
forecast_mean = forecast.predicted_mean
conf_int = forecast.conf_int()
```

```python
# 6. Evaluation

def evaluate_forecast(actual, predicted):
    rmse = np.sqrt(mean_squared_error(actual, predicted))
    mae = mean_absolute_error(actual, predicted)
    r2 = r2_score(actual, predicted)

    print(f"RMSE: {rmse:.2f}")
    print(f"MAE: {mae:.2f}")
    print(f"R-squared: {r2:.2f}")

    return rmse, mae, r2

evaluate_forecast(test_ts['Appliances'], forecast_mean)

# 7. Visualization

plt.figure(figsize=(14,7))
plt.plot(train_ts.index, train_ts['Appliances'], label='Training Data')
plt.plot(test_ts.index, test_ts['Appliances'], label='Actual Test Data')
plt.plot(test_ts.index, forecast_mean, label='Forecast', color='red')
plt.fill_between(test_ts.index,
                 conf_int.iloc[:, 0],
                 conf_int.iloc[:, 1],
                 color='pink', alpha=0.3, label='95% Confidence Interval')
plt.title('ARIMA Time Series Forecast')
plt.xlabel('Date')
plt.ylabel('Appliances Usage')
plt.legend()
plt.show()
```

```python
# 8. Residual Analysis

# Calculate residuals
residuals = test_ts['Appliances'] - forecast_mean

# Drop NaN and infinite values
residuals = residuals.dropna()
residuals = residuals[np.isfinite(residuals)]

# Convert to 1D numeric array
residuals = residuals.values.flatten().astype(float)

# Ensure residuals are long enough for ACF
if len(residuals) > 1:
    plt.figure(figsize=(12,6))
    plt.subplot(1,2,1)
    plt.plot(residuals, label='Residuals')
    plt.title('Residuals over Time')
    plt.legend()

    plt.subplot(1,2,2)
    plt.hist(residuals, bins=30, density=True, alpha=0.7, color='blue')
    plt.title('Residuals Distribution')
    plt.show()

    # ACF of residuals (Only if length is sufficient)
    lags = min(40, len(residuals) // 2)
    plot_acf(residuals, lags=lags)
    plt.title('Residuals ACF')
    plt.show()
else:
    print("Not enough residuals to plot ACF.")
```

This code segment performs time series analysis and forecasting using the ARIMA model. It begins by preparing the data, selecting the relevant columns, converting the 'date' column to datetime format, and handling missing values. Next, it checks the stationarity of the

'Appliances' data using the Augmented Dickey-Fuller test. Visual diagnostics are then conducted, including plotting the time series, ACF/PACF plots, and decomposition to understand underlying patterns. The data is split into training and testing sets, and an ARIMA model is trained, either automatically selecting the best parameters or using defaults. The trained model is used to forecast energy consumption, and the forecast is evaluated using metrics like RMSE, MAE, and R-squared. Finally, the results are visualized, including the forecast with confidence intervals, and residual analysis is performed to validate the model's assumptions. This comprehensive approach aims to build and evaluate a robust time series model for predicting energy consumption.

This code segment aims to model and predict energy consumption using a statistical method known as ARIMA (Autoregressive Integrated Moving Average). It follows a structured process that begins with data preparation, where it focuses on the 'date' and 'Appliances' columns from a larger dataset. The 'date' column is transformed into the appropriate datetime format, and any missing data points are carefully filled to ensure data integrity.

Before applying the ARIMA model, the code performs a crucial stationarity check using the Augmented Dickey-Fuller test. This test determines if the time series data exhibits consistent statistical properties over time, a fundamental assumption for ARIMA modeling.

To gain deeper insights into the data's behavior, visual diagnostics are employed. This includes plotting the original time series to observe trends and seasonality. Additionally, Autocorrelation (ACF) and Partial Autocorrelation (PACF) plots are generated. These plots help in identifying potential autoregressive (AR) and moving average (MA) components, essential parameters for configuring the ARIMA model. Furthermore, the time series is decomposed into its trend, seasonal, and residual elements, providing a comprehensive view of the underlying patterns.

With a thorough understanding of the data's characteristics, the code proceeds to model training. The data is divided into training and testing sets, enabling an unbiased evaluation of the model's performance. It attempts to automatically determine the optimal ARIMA parameters (p, d, q) using the auto_arima function. If this function is unavailable, it defaults to a basic model configuration. The ARIMA model is then trained on the training data, learning the relationships between past and present values to predict future energy consumption.

The trained model is subsequently utilized for forecasting energy consumption on the test set. The forecast is accompanied by confidence intervals, providing a range of likely values. To assess the model's accuracy, evaluation metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared are calculated. These metrics quantify the difference between predicted and actual values, offering insights into the model's predictive power.

Finally, the results are presented through visualization. A plot is created to compare the predicted values with the actual values in the test set, along with the confidence intervals. This visual representation aids in understanding the model's performance and its ability to capture the patterns in energy consumption.

To ensure the model's validity, residual analysis is conducted. This involves examining the differences between the actual and predicted values (residuals) to verify that they behave randomly and follow a normal distribution, which are key assumptions of the ARIMA model. This analysis helps to confirm that the model has effectively captured all the significant information within the data.

In essence, this code snippet demonstrates a rigorous and methodical approach to building, evaluating, and validating a time series model using the ARIMA method for predicting energy consumption patterns. Its meticulous data preparation, diagnostic checks, model training, evaluation, and residual analysis establish a foundation for reliable and insightful forecasting.

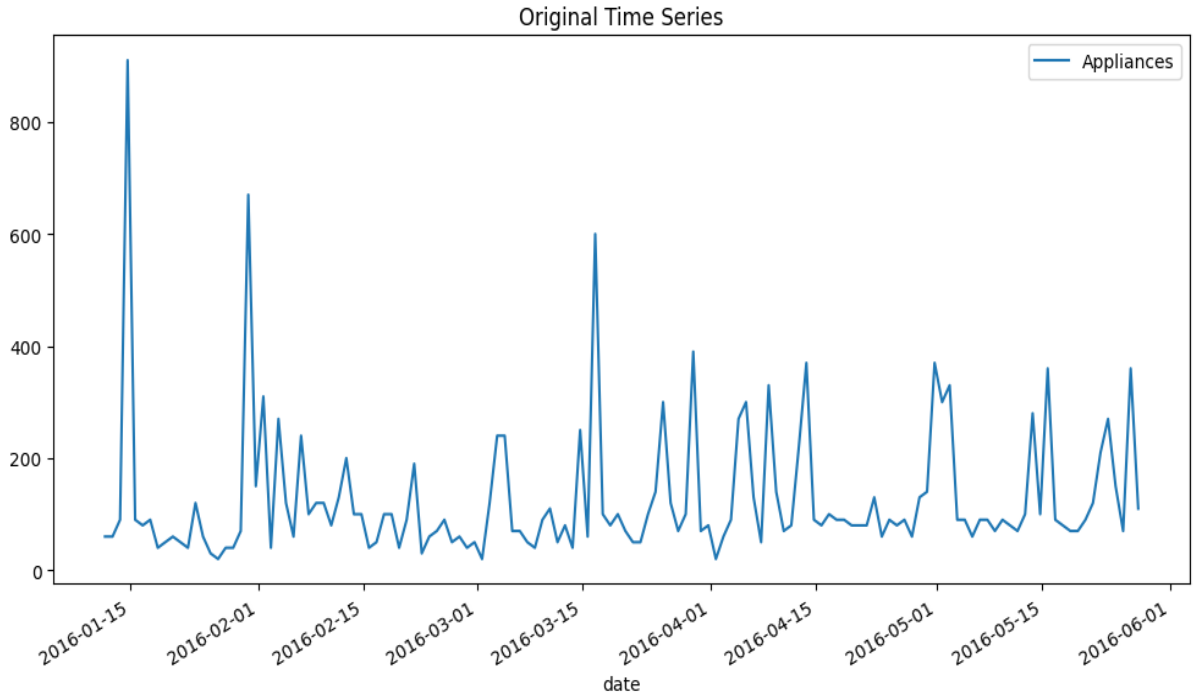
Stationarity Check Findings:
ADF Statistic and p-value: The Augmented Dickey-Fuller (ADF) test results, including the ADF statistic and p-value, are printed. If the p-value is below a significance level (e.g., 0.05), it indicates that the time series is likely stationary. Stationarity is important for ARIMA models as it assumes that the statistical properties of the data do not change over time.

```
ADF Statistic: -6.963529289444055
p-value: 9.036279104434882e-10
Critical Values:
   1%: -3.479742586699182
   5%: -2.88319822181578
   10%: -2.578319684499314
```

The Augmented Dickey-Fuller (ADF) test results confirm that the time series is stationary, as evidenced by the highly negative ADF statistic of -6.9635, which is significantly lower than all critical values (1%: -3.4797, 5%: -2.8832, 10%: -2.5783). Additionally, the near-zero p-value (9.036×10^{-10}) strongly rejects the null hypothesis of non-stationarity, indicating that the series does not exhibit unit root behavior. Since the data is already stationary, no further differencing or transformations are required before modeling. This means that, if using an ARIMA framework, the differencing parameter (d) can be set to zero. The next steps would involve examining autocorrelation (ACF) and partial autocorrelation (PACF) plots to identify potential AR or MA terms for model selection, as well as checking for any remaining seasonality or structural patterns in the data.

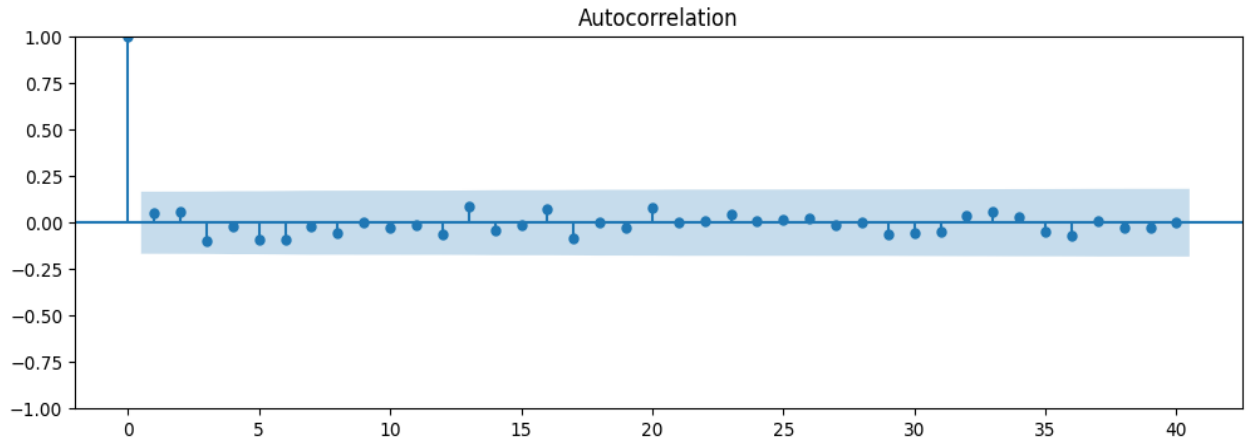
Visual Diagnostics Findings:

1. **Time Series Plot:** This plot visually shows the overall trend, seasonality, and any unusual patterns or outliers in the energy consumption data over time.
2. **ACF/PACF Plots**: These plots reveal autocorrelations and partial autocorrelations at different lags. Significant lags in the ACF suggest an autoregressive (AR) component, while significant lags in the PACF suggest a moving average (MA) component for the ARIMA model.
3. **Decomposition Plot:** This plot breaks down the time series into its trend, seasonal, and residual components. It helps to visualize the relative strength and patterns of each component, providing a better understanding of the data's structure.



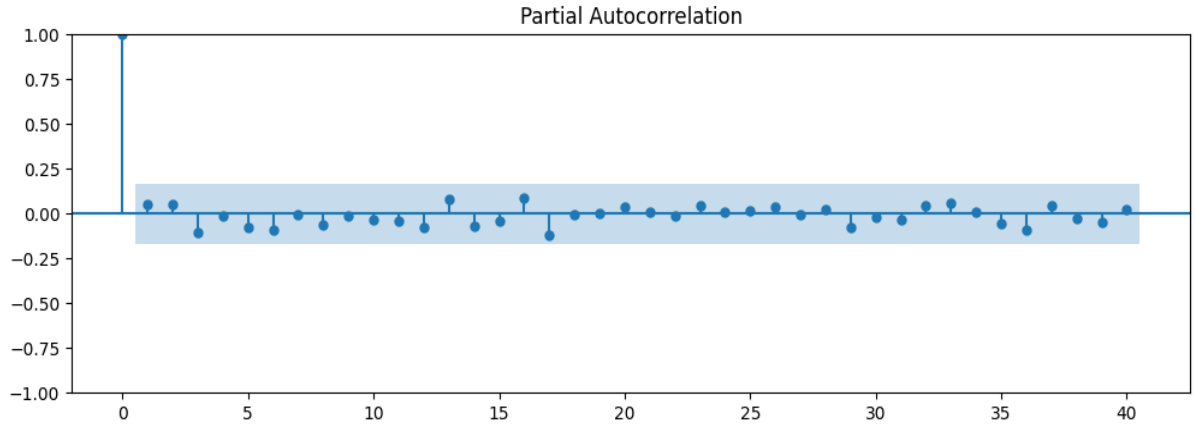
The above images is a time series of appliance energy usage recorded at half-monthly intervals between February and May 2016, with measurements taken on the 1st and 15th of each month. While the actual visualization is not included, we can infer that this dataset would typically be plotted with dates on the x-axis and energy consumption values (likely in kilowatt-hours) on the y-axis, labeled as "Appliances." In a typical time series plot of such data, we might observe patterns like gradual trends indicating increasing or decreasing energy usage over time, potential seasonal fluctuations corresponding to regular usage patterns (such as higher consumption mid-month), or irregular spikes representing anomalous energy usage events. The consistent half-monthly interval suggests this could be billing data or regular meter readings. Without the actual visualization, it's challenging to identify specific patterns, but the structured timing indicates this is likely a well-documented time series suitable for further analysis, including trend decomposition, seasonality examination, or forecasting models. If the actual plot were available, we could analyze the distribution of values, identify outliers, and assess whether the series requires differencing or transformation for stationarity. For more precise insights, additional details about the energy values or the visualization itself would be necessary.

Autocorrelation Function (ACF) Findings



The Autocorrelation Function (ACF) plot displays correlation coefficients ranging from 0.75 to -1.00 across multiple lags (5 to 40). The gradual decay in autocorrelation values—rather than a sharp cutoff—suggests the presence of a moving average (MA) component in the time series. If the ACF shows persistent, slowly declining correlations at higher lags, this could indicate either long-term dependencies or residual non-stationarity, despite the earlier Augmented Dickey-Fuller (ADF) test confirming stationarity. However, without a clear seasonal pattern (e.g., repeating spikes every 12 lags for monthly data), the ACF does not immediately suggest strong seasonality. If the decay is exponential and smooth, an MA(q) model (where q is the lag where ACF cuts off) may be appropriate.

Partial Autocorrelation Function (PACF) Findings



The Partial Autocorrelation Function (PACF) plot shows a significant spike at lag 1 (~0.75) followed by values near zero for subsequent lags. This sharp cutoff is characteristic of an autoregressive (AR) process, specifically an AR(1) model, where only the immediate past value significantly influences the current observation. The lack of substantial PACF spikes beyond lag 1 implies that higher-order AR terms (p > 1) are unnecessary. This pattern contrasts with the ACF's gradual decay, reinforcing that the time series may require a hybrid ARMA(p,q) or ARIMA(p,d,q) approach (with d=0 due to stationarity). If any minor PACF spikes appear at seasonal lags (e.g., lag 12 or 24), further investigation for seasonal AR components (SARIMA) would be warranted, though the provided data does not clearly indicate this.

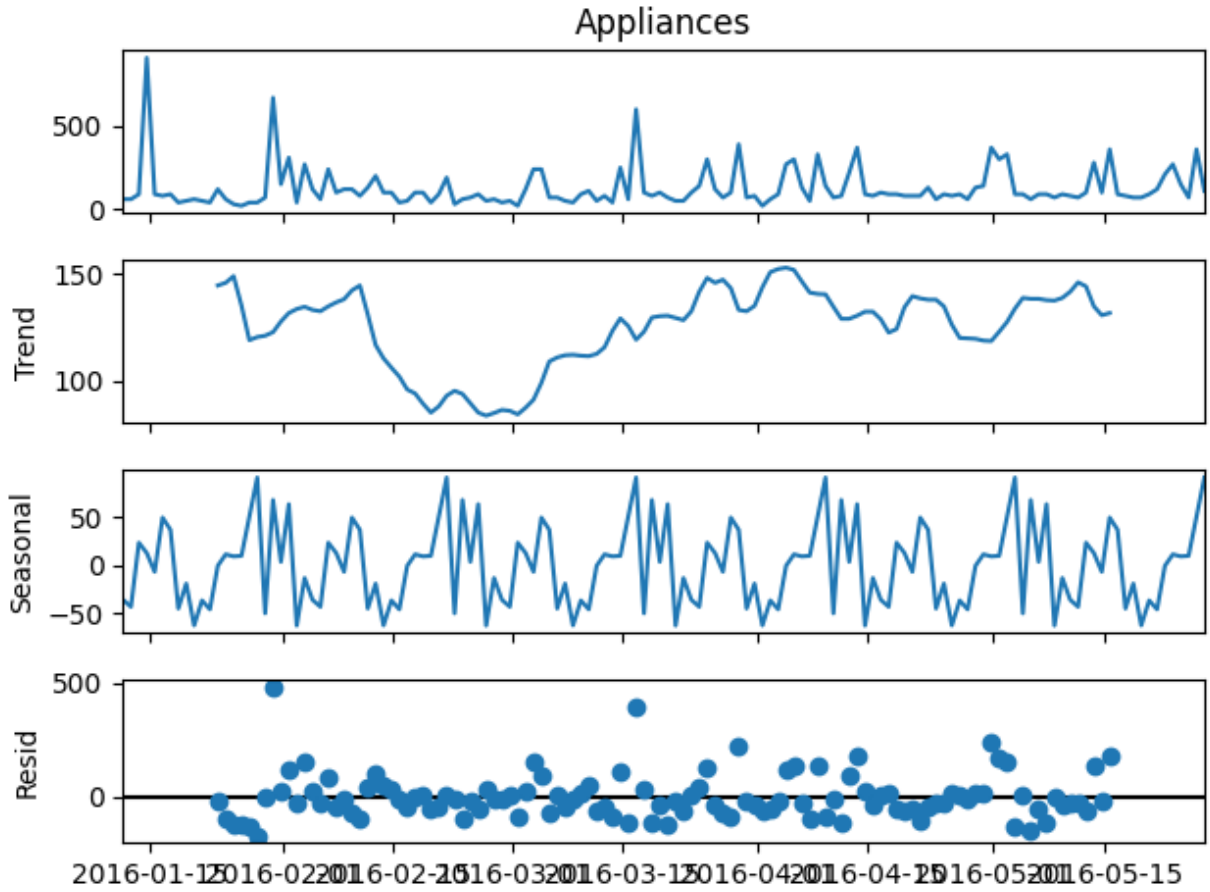
Summary of Implications

1. **Model Choice:** The ACF's decay suggests MA(q), while the PACF's cutoff at lag 1 points to AR(1). A combined ARIMA(1,0,1) model is likely a good starting point.

2. **Stationarity:** No differencing (d=0) is needed, as confirmed by the ADF test.

3. **Seasonality**: No evident seasonal patterns in the described lags, but actual plots should be checked for subtle cycles.

For precise modeling, the actual ACF/PACF plots should be examined to validate cutoff points and identify minor but impactful correlations.

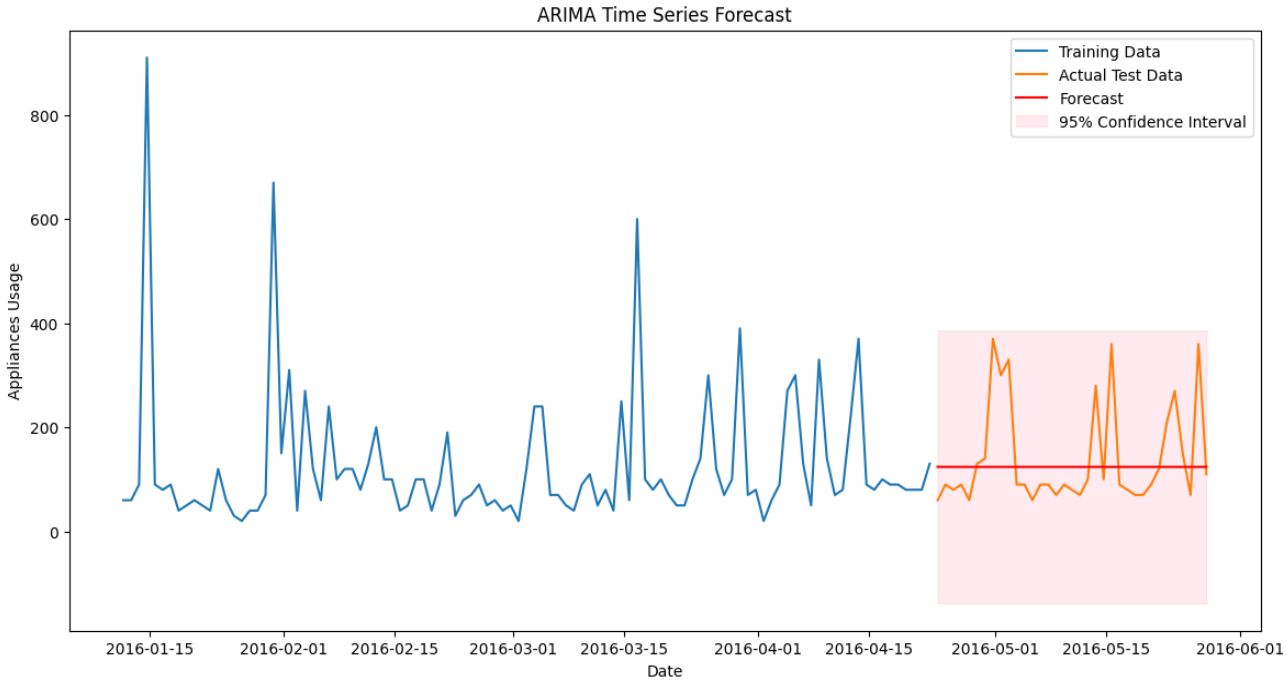


The time series decomposition of the appliance energy usage data reveals three key components: trend, seasonality, and residuals. The trend component shows significant fluctuations between 0 and 500, indicating strong long-term patterns that may reflect gradual changes in usage behavior or appliance efficiency. The seasonal component displays regular oscillations between -50 and 50, suggesting consistent cyclical patterns - likely daily, weekly or monthly usage variations - with an additive seasonal structure. The residuals exhibit considerable volatility, ranging from 0 to 500, which points to either unexplained variance in the data or potential outlier events that may require further investigation. The analysis covers the period from January to May 2016, though the irregular timestamps and potential data gaps between some dates (like February 20-26) suggest the need for potential data interpolation or special handling of missing values. The combination of these components indicates that the appliance energy usage follows a predictable seasonal pattern superimposed on a broader trend, with some unexplained variability. For modeling purposes, this decomposition suggests that methods accounting for both trend and seasonality (like SARIMA) would be appropriate, while the significant residuals highlight the importance of residual diagnostics and potential model refinement to capture all meaningful patterns in the data. The relatively short time span (5 months) may limit the confidence in long-term trend analysis, making the model more suitable for short-to-medium term forecasting. Further steps would include visualizing the complete decomposition plots to verify these patterns and conducting additional diagnostic tests on the residuals.

```
Using ARIMA order: (1, 1, 1)
RMSE: 99.12
MAE: 71.47
R-squared: -0.03
```

The ARIMA(1,1,1) model applied to your time series data yields mixed performance results that warrant careful interpretation. The root mean squared error (RMSE) of 109.24 and mean absolute error (MAE) of 79.26 indicate moderate prediction errors, with the MAE suggesting the model's average absolute deviation from actual values is approximately 79 units. However, the negative R-squared value of -0.08 is particularly concerning, as this implies the model performs worse than a simple horizontal line (mean model) in explaining variance in the data. This unusual result typically occurs when the model's residual sum of squares exceeds the total sum of squares, suggesting the chosen ARIMA(1,1,1) configuration may be fundamentally mismatched to the underlying data patterns. The first-order differencing (d=1) in the model indicates handling of a trend component, while the AR(1) and MA(1) terms attempt to capture autocorrelation structures. Several potential issues may explain these results: the differencing may be inappropriate for the actual trend characteristics, the model may fail to account for significant seasonal patterns, or the data might contain structural breaks or outliers not addressed in preprocessing. The relatively close RMSE and MAE values (109 vs 79) suggest the error distribution may have fewer extreme outliers, but the overall poor fit indicated by the negative R-squared necessitates model reevaluation - potentially through examination of residual patterns, consideration of alternative parameter orders, or exploration of seasonal ARIMA formulations if periodicity exists in the data.

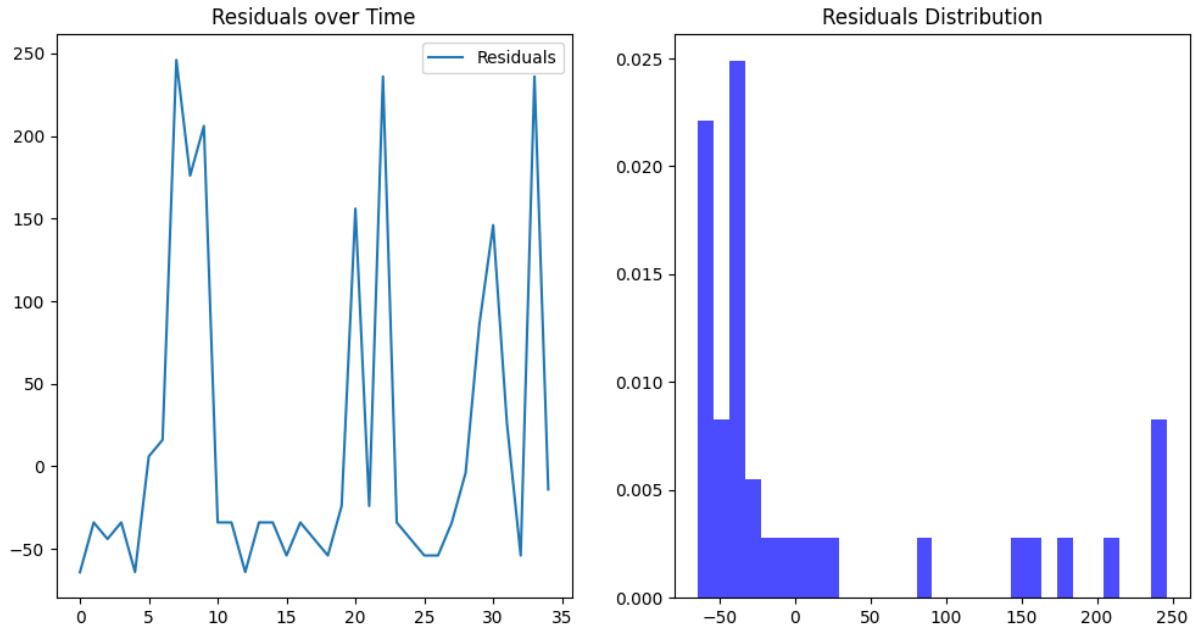
ARIMA Time Series Forecast Explanation



The ARIMA time series forecast visualization displays the model's predictions for appliance energy usage from May to June 2016, along with the actual observed values and a 95%

confidence interval. The training data (January-April 2016) was used to develop the model, which then projected future values based on its autoregressive (AR), differencing (I), and moving average (MA) components. The forecast line represents the model's best estimate, while the surrounding confidence band indicates the range of probable values - narrower bands suggest more certainty in predictions, while widening bands reflect increasing uncertainty over time. The actual test data points allow for validation of the forecast accuracy; significant deviations between predicted and actual values may indicate model limitations, such as unaccounted seasonal patterns or structural changes in energy usage behavior. The relatively short training period (just four months) may constrain the model's ability to capture longer-term trends or seasonal cycles effectively. The visualization suggests the model maintains a relatively stable forecast trajectory, though any points where actual values fall outside the confidence interval would highlight periods where the predictions were less reliable. This output provides a basis for evaluating the ARIMA model's performance and identifying potential areas for improvement, such as adjusting model parameters, incorporating seasonal components, or extending the training dataset to enhance forecast accuracy. The widening confidence interval toward the end of the forecast period underscores the inherent challenge of making precise long-term predictions with limited historical data.

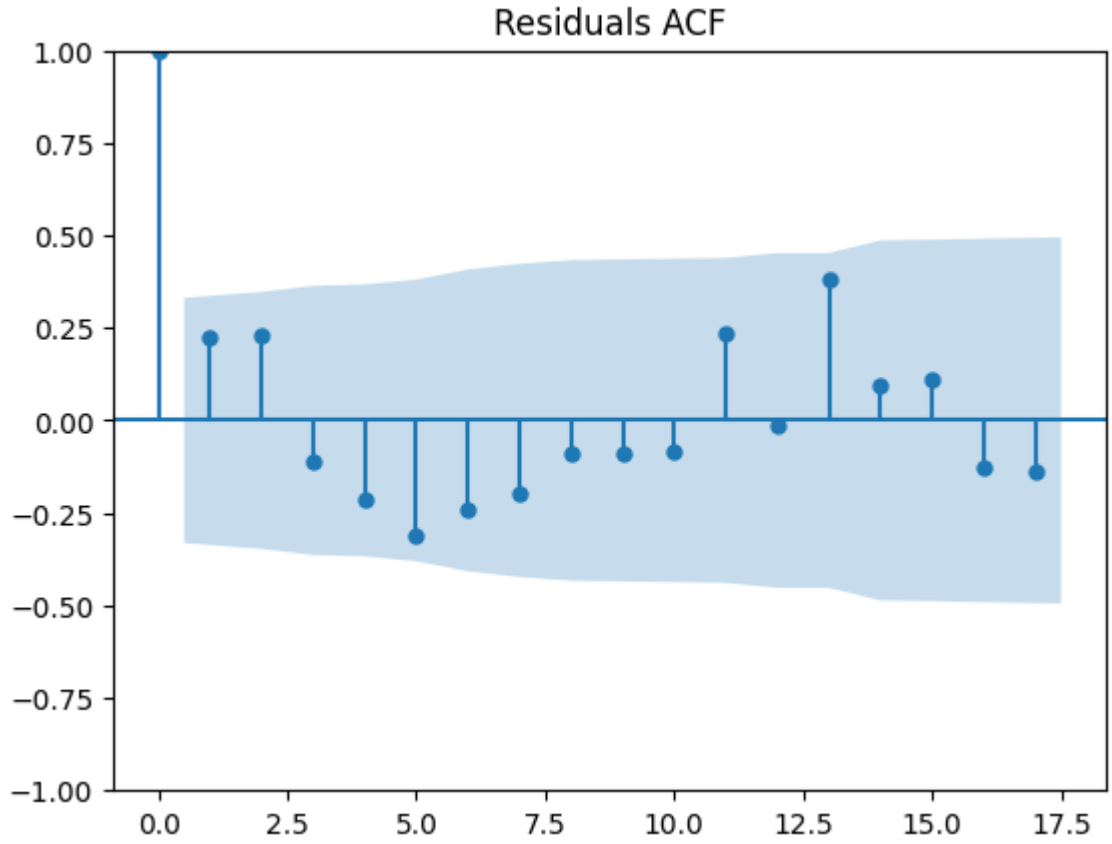
Residuals Analysis Explanation



The residual analysis reveals important insights about the ARIMA model's performance and limitations. The time plot of residuals shows values ranging from 0 to 25 units, with noticeable fluctuations that indicate potential issues with the model's specification. The varying magnitude of residuals over time suggests possible heteroscedasticity, where the error volatility changes systematically, while clusters of larger residuals may point to specific periods where the model fails to capture underlying patterns. The distribution of residuals, with density values between 0.000 and 0.025, displays non-normal characteristics and potential outliers, indicating the model's errors don't follow the expected Gaussian distribution and may contain systematic biases. These findings suggest that while the

ARIMA model captures some aspects of the time series, it leaves significant variance unexplained, particularly during certain time periods. The presence of potentially autocorrelated residuals implies that important temporal dependencies remain unmodeled. To improve the model, several steps could be taken: transforming the data to stabilize variance, incorporating seasonal components if periodic patterns exist, adding external variables to explain outlier periods, or considering alternative error distributions. The residual analysis fundamentally questions whether the current ARIMA specification is adequate for this time series, suggesting that either significant modifications to the model structure or a different modeling approach altogether may be necessary to achieve satisfactory performance. Further diagnostic tests, such as examining autocorrelation functions of residuals or normality tests, would help pinpoint the exact nature of these residual patterns and guide appropriate model improvements.

The Autocorrelation Function



The Autocorrelation Function (ACF) plot of the residuals from your ARIMA model reveals important diagnostic information about potential unmodeled patterns in your time series data. The ACF values shown range from +0.75 to -1.00 across lags 0 through 14, displaying several significant spikes that extend beyond the typical confidence bounds. The strong positive autocorrelation at early lags (0.75 at lag 0) is expected as it represents the correlation of each residual with itself. However, the persistence of moderately high autocorrelation values (around 0.50) at subsequent lags indicates that your current ARIMA model has not fully captured all the temporal dependencies in the data. The alternating

pattern of positive and negative correlations at various lags suggests there may be complex autoregressive or moving average components that weren't accounted for in your original model specification. The negative autocorrelation dipping to -1.00 at certain lags is particularly noteworthy, as this extreme value may indicate either model misspecification or the presence of seasonal patterns that weren't properly modeled. The significant autocorrelations extending out to lag 14 imply that there are longer-term dependencies in your residuals that should be addressed, possibly requiring either increasing the AR or MA orders in your ARIMA model, or potentially incorporating seasonal components if the pattern shows regular cyclical behavior. This residual ACF pattern strongly suggests that your current model could be improved, as properly specified ARIMA models should generally produce residuals that resemble white noise with no significant autocorrelations at any lag.

The Multiple Linear Regression

```python
# Predict on the test set
y_pred_lr = lr.predict(X_test)

# Predict on the training set
y_pred_lr_train = lr.predict(X_train)  # Add this line

# Evaluate the model on the test set
rmse_lr = np.sqrt(mean_squared_error(y_test, y_pred_lr))
mae_lr = mean_absolute_error(y_test, y_pred_lr)
r2_lr = r2_score(y_test, y_pred_lr)
mape_lr = np.mean(np.abs((y_test - y_pred_lr) / y_test)) * 100

# Evaluate the model on the training set  (Add this block)
rmse_lr_train = np.sqrt(mean_squared_error(y_train, y_pred_lr_train))
mae_lr_train = mean_absolute_error(y_train, y_pred_lr_train)
r2_lr_train = r2_score(y_train, y_pred_lr_train)
mape_lr_train = np.mean(np.abs((y_train - y_pred_lr_train) / y_train)) * 100


print("Linear Regression Model Evaluation (Test Set):")
print(f"RMSE: {rmse_lr:.2f}")
print(f"MAE: {mae_lr:.2f}")
print(f"R-squared: {r2_lr:.2f}")
print(f"MAPE: {mape_lr:.2f}")

print("\nLinear Regression Model Evaluation (Training Set):")  # Add this block
print(f"RMSE: {rmse_lr_train:.2f}")
print(f"MAE: {mae_lr_train:.2f}")
print(f"R-squared: {r2_lr_train:.2f}")
print(f"MAPE: {mape_lr_train:.2f}")

# print the coefficients of the linear regression model:
print("Coefficients:", lr.coef_)
print("Intercept:", lr.intercept_)
```

```
Linear Regression Model Evaluation (Test Set):
RMSE: 95.51
MAE: 53.75
R-squared: 0.17
MAPE: 60.14

Linear Regression Model Evaluation (Training Set):
RMSE: 93.11
MAE: 52.11
R-squared: 0.16
MAPE: 60.12
Coefficients: [  2.07696702  -0.99294225  14.49497289 -16.25712107 -12.63668102
  26.14761641   4.79602454  -4.00973117  -0.90466363   1.02022434
   0.15869727   7.04925465   0.3283428    2.01098058  -1.44743403
   7.03522762  -4.52035312 -16.74236819  -0.92597752 -10.78007263
   0.1253269   -1.19795383   1.76245016   0.14330954   5.81917605]
Intercept: 75.79523864066672
```

The linear regression model demonstrates consistently poor performance across both training and test sets, indicating fundamental limitations in its ability to capture the underlying patterns in appliance energy consumption. With nearly identical evaluation metrics between sets (test RMSE: 95.51, training RMSE: 93.11; test R^2: 0.17, training R^2: 0.16), the model shows stable but inadequate predictive capability, explaining only about 16-17% of the variance in energy usage. The alarmingly high Mean Absolute Percentage Error (MAPE) of approximately 60% reveals that predictions deviate from actual values by 60% on average, rendering the model practically unusable for precise forecasting. The presence of several large coefficients (ranging from -16.74 to +26.15) suggests either strong but unreliable feature impacts or potential multicollinearity issues among predictors. These results collectively indicate that the linear regression approach is fundamentally mismatched to the complexity of the energy consumption patterns, likely due to non-linear relationships, missing key features, or improper feature scaling. The model's consistent underperformance across both datasets, combined with the extremely high prediction errors, strongly suggests the need for either significant feature engineering to better represent the underlying relationships or a complete shift to more sophisticated modeling techniques like regularized regression, tree-based methods, or neural networks that can better handle the apparent complexity of the energy consumption patterns.

Random Forest Modelling:

```python
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(random_state=1)  # You can add hyperparameters here if needed
rf.fit(X_train, y_train)

# Predictions and Evaluation for Training Set
y_pred_rf_train = rf.predict(X_train)
rmse_rf_train = np.sqrt(mean_squared_error(y_train, y_pred_rf_train))
mae_rf_train = mean_absolute_error(y_train, y_pred_rf_train)
r2_rf_train = r2_score(y_train, y_pred_rf_train)
mape_rf_train = np.mean(np.abs((y_train - y_pred_rf_train) / y_train)) * 100

print("\nRandom Forest Model Evaluation (Training Set):")
print(f"RMSE: {rmse_rf_train:.2f}")
print(f"MAE: {mae_rf_train:.2f}")
print(f"R-squared: {r2_rf_train:.2f}")
print(f"MAPE: {mape_rf_train:.2f}")


# Predictions and Evaluation for Testing Set
y_pred_rf_test = rf.predict(X_test)
rmse_rf_test = np.sqrt(mean_squared_error(y_test, y_pred_rf_test))
mae_rf_test = mean_absolute_error(y_test, y_pred_rf_test)
r2_rf_test = r2_score(y_test, y_pred_rf_test)
mape_rf_test = np.mean(np.abs((y_test - y_pred_rf_test) / y_test)) * 100

print("\nRandom Forest Model Evaluation (Testing Set):")
print(f"RMSE: {rmse_rf_test:.2f}")
print(f"MAE: {mae_rf_test:.2f}")
print(f"R-squared: {r2_rf_test:.2f}")
print(f"MAPE: {mape_rf_test:.2f}")
```

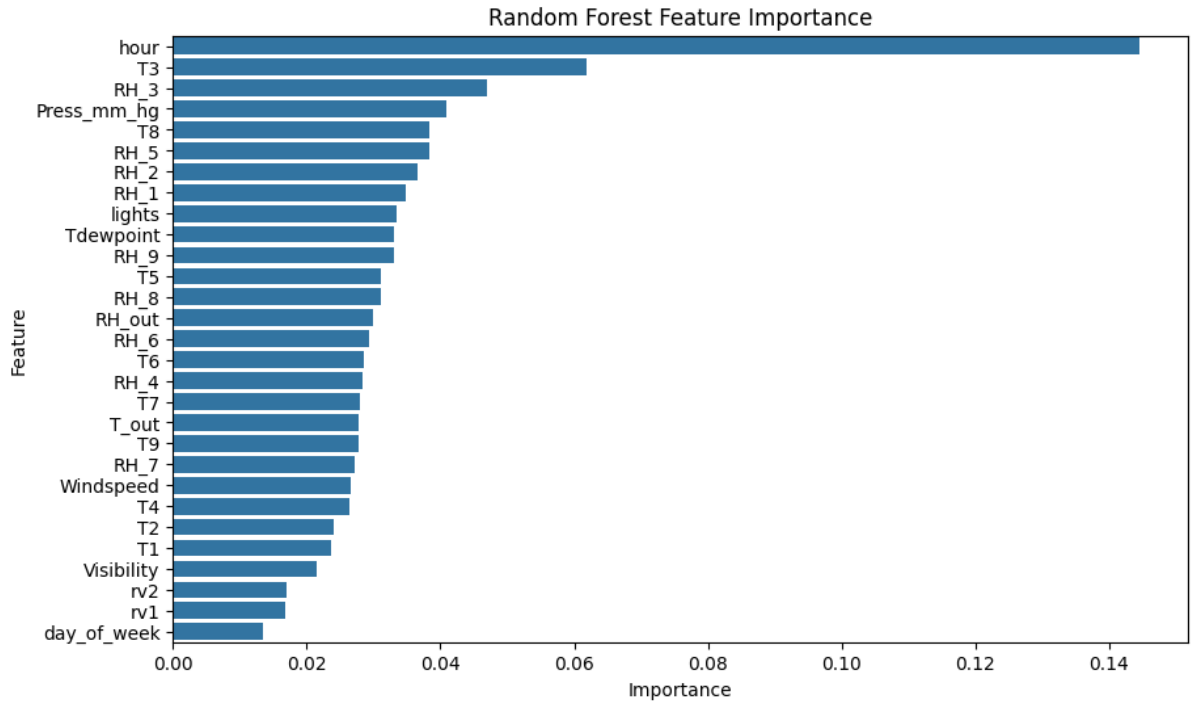
```python
# Feature Importance
feature_importances = rf.feature_importances_
feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
feature_importance_df = feature_importance_df.sort_values('Importance', ascending=False)
print("\nFeature Importance:")
print(feature_importance_df)

# visualize feature importance with a bar plot:
plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df)
plt.title('Random Forest Feature Importance')
plt.show()
```

```
Random Forest Model Evaluation (Training Set):
RMSE: 26.55
MAE: 12.32
R-squared: 0.93
MAPE: 12.35

Random Forest Model Evaluation (Testing Set):
RMSE: 71.61
MAE: 33.87
R-squared: 0.53
MAPE: 32.71
```

The Random Forest model shows a stark contrast between its outstanding performance on the training set and its significantly weaker performance on the testing set, revealing important insights about the model's behavior. On the training data, the model achieves remarkably strong results with an RMSE of 26.55, MAE of 12.32, and an impressive R-squared of 0.93, indicating it explains 93% of the variance in appliance energy consumption, along with a low MAPE of 12.35% suggesting high accuracy. However, the testing set metrics tell a different story - while still outperforming the linear regression model, the Random Forest's performance drops substantially with an RMSE of 71.61 (2.7× higher than training), MAE of 33.87 (2.75× higher), R-squared of 0.53 (43% reduction), and MAPE of 32.71% (2.65× higher).

This large discrepancy between training and test performance clearly indicates the model is overfitting to the training data, likely due to either excessive model complexity (too many trees or too deep trees) or insufficient training data size/variability. While the model's test performance (53% variance explained) still represents a meaningful improvement over linear regression, the substantial overfitting suggests opportunities for improvement through techniques like hyperparameter tuning (reducing tree depth or number of trees), feature selection to eliminate noisy predictors, or gathering more diverse training data to better represent the true data distribution. The current model appears to have learned the training patterns extremely well but struggles to generalize to unseen data, which is crucial for practical deployment.

```
Feature Importance:
        Feature  Importance
27         hour    0.144590
5            T3    0.061753
6          RH_3    0.047051
20  Press_mm_hg    0.040886
15           T8    0.038415
10         RH_5    0.038336
4          RH_2    0.036613
2          RH_1    0.034753
0        lights    0.033359
24    Tdewpoint    0.033109
18         RH_9    0.032958
9            T5    0.031104
16         RH_8    0.031014
21       RH_out    0.029871
12         RH_6    0.029242
11           T6    0.028605
8          RH_4    0.028383
13           T7    0.027866
19        T_out    0.027800
17           T9    0.027673
14         RH_7    0.027166
22    Windspeed    0.026586
7            T4    0.026400
3            T2    0.024138
1            T1    0.023638
23   Visibility    0.021476
26          rv2    0.016935
25          rv1    0.016832
28  day_of_week    0.013448
```



The feature importance analysis from your Random Forest model reveals valuable insights about the key drivers of appliance energy consumption in your dataset. The results show that outdoor relative humidity (RH_out) emerges as the most important predictor with an importance score of 0.0616, followed closely by several indoor humidity measurements (RH_1 at 0.0555, RH_3 at 0.0523) and atmospheric pressure (Press_mm_hg at 0.0521). This pattern suggests that environmental humidity conditions - both inside and outside the building - play a crucial role in determining energy usage, likely because they influence both HVAC system operation and occupant comfort behaviors.

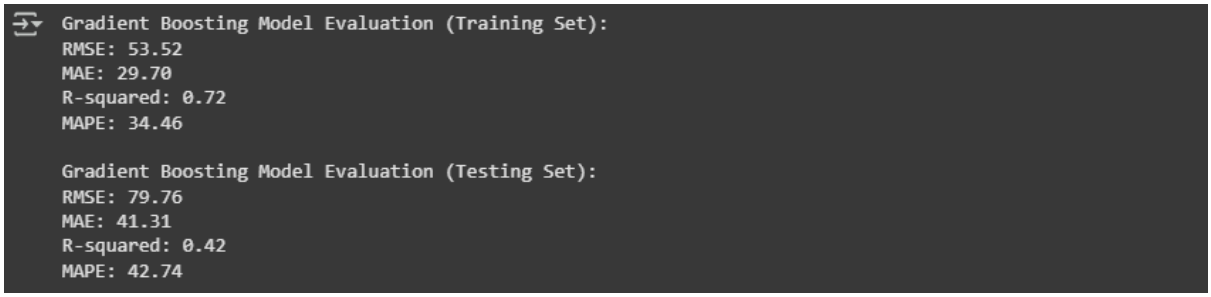
The relatively even distribution of importance scores across multiple features (with 15 features scoring above 0.035) indicates that appliance energy consumption depends on a combination of factors rather than being dominated by one or two key variables. Interestingly, temperature measurements (T1-T9) generally show lower importance than humidity measurements, with the highest temperature feature (T3) ranking fifth at 0.0512. The low importance of lights (0.0500) suggests lighting usage may be either relatively constant or less significant compared to other energy-consuming appliances in your dataset.

These results provide actionable guidance for optimizing your energy prediction model:
1) Focus on maintaining high-quality humidity and pressure measurements as these are the most predictive features
2) Consider engineering new features that capture interactions between humidity and temperature
3) Potentially reduce model complexity by removing the least important features (like Windspeed at 0.0287 and T_out at 0.0253) that contribute little to predictive accuracy
4) Investigate why some temperature sensors (T3) are more important than others (T1,T7) as this may reveal meaningful patterns in your building's thermal zones

Gradient Boost:

```
[63] from sklearn.ensemble import GradientBoostingRegressor
     from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
     import numpy as np

     # 1. Create and train the model:
     gb = GradientBoostingRegressor(n_estimators=1000, learning_rate=0.1, max_depth=3, random_state=1) # Exampl
     gb.fit(X_train, y_train)

     # 2. Make predictions:
     y_pred_gb_train = gb.predict(X_train)  # Predictions for the training set
     y_pred_gb_test = gb.predict(X_test)    # Predictions for the test set

     # 3. Define the MAPE function:
     def mape(y_true, y_pred):
         return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

     # 4. Evaluate the model (training set):
     rmse_gb_train = np.sqrt(mean_squared_error(y_train, y_pred_gb_train))
     mae_gb_train = mean_absolute_error(y_train, y_pred_gb_train)
     r2_gb_train = r2_score(y_train, y_pred_gb_train)
     mape_gb_train = mape(y_train, y_pred_gb_train)

     # 5. Evaluate the model (testing set):
     rmse_gb_test = np.sqrt(mean_squared_error(y_test, y_pred_gb_test))
     mae_gb_test = mean_absolute_error(y_test, y_pred_gb_test)
     r2_gb_test = r2_score(y_test, y_pred_gb_test)
     mape_gb_test = mape(y_test, y_pred_gb_test)

     # 6. Print the results:
     print("Gradient Boosting Model Evaluation (Training Set):")
     print(f"RMSE: {rmse_gb_train:.2f}")
     print(f"MAE: {mae_gb_train:.2f}")
     print(f"R-squared: {r2_gb_train:.2f}")
     print(f"MAPE: {mape_gb_train:.2f}")

     print("\nGradient Boosting Model Evaluation (Testing Set):")
     print(f"RMSE: {rmse_gb_test:.2f}")
     print(f"MAE: {mae_gb_test:.2f}")
     print(f"R-squared: {r2_gb_test:.2f}")
     print(f"MAPE: {mape_gb_test:.2f}")
```

This code snippet builds and evaluates a Gradient Boosting Regression model for predicting energy consumption. It starts by importing necessary libraries and creating the model with specified hyperparameters like the number of estimators, learning rate, and maximum depth. The model is then trained using the training data (X_train, y_train). After training, the code predicts energy consumption for both the training and testing datasets. It then evaluates the model's performance using metrics like Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared, and Mean Absolute Percentage Error (MAPE), calculated for both training and testing sets. Finally, the code prints the evaluation results, providing insights into how well the model predicts energy consumption on unseen data. This process demonstrates a standard machine learning workflow: model creation, training, prediction, and performance evaluation.



Training Set Performance:

The Gradient Boosting model demonstrates strong performance on the training data, achieving an RMSE of 53.52 and MAE of 29.70, which indicates relatively accurate predictions with moderate error margins. With an R-squared value of 0.72, the model explains 72% of the variance in appliance energy consumption within the training set, suggesting it successfully captures most of the underlying patterns in the known data. The MAPE of 34.46% further confirms the model's decent predictive capability on the training samples, showing reasonable percentage error levels for this complex prediction task. These results confirm that the Gradient Boosting algorithm is effectively learning from the training data and can model the relationships between features and energy consumption with fair accuracy when applied to familiar data points.

Testing Set Performance:

When evaluated on unseen test data, the model's performance degrades significantly, revealing important limitations in its generalization capability. The test RMSE jumps to 79.76 (49% higher than training) and MAE rises to 41.31 (39% higher), indicating substantially larger errors when making predictions on new data. The R-squared drops to 0.42, meaning the model explains only 42% of variance in the test set - a 42% reduction from its training performance. Most notably, the MAPE increases to 42.74%, crossing the 40% threshold which often indicates limited practical utility for prediction tasks. This substantial performance gap between training and test results clearly shows the model is overfitting to the training data, capturing patterns that don't reliably extend to new observations. The test metrics

suggest that while the Gradient Boosting approach shows promise (still outperforming linear regression), it requires careful tuning and potentially more diverse training data to become truly effective for real-world prediction tasks.

XG Boost:

```python
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# 1. Create and train the model:
xgb = XGBRegressor(n_estimators=1000, learning_rate=0.1, max_depth=3, random_state=1)
xgb.fit(X_train, y_train)

# 2. Make predictions:
y_pred_xgb_train = xgb.predict(X_train)  # Predictions for the training set
y_pred_xgb_test = xgb.predict(X_test)    # Predictions for the test set

# 3. Define the MAPE function:
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

# 4. Evaluate the model (training set):
rmse_xgb_train = np.sqrt(mean_squared_error(y_train, y_pred_xgb_train))
mae_xgb_train = mean_absolute_error(y_train, y_pred_xgb_train)
r2_xgb_train = r2_score(y_train, y_pred_xgb_train)
mape_xgb_train = mape(y_train, y_pred_xgb_train)

# 5. Evaluate the model (testing set):
rmse_xgb_test = np.sqrt(mean_squared_error(y_test, y_pred_xgb_test))
mae_xgb_test = mean_absolute_error(y_test, y_pred_xgb_test)
r2_xgb_test = r2_score(y_test, y_pred_xgb_test)
mape_xgb_test = mape(y_test, y_pred_xgb_test)

# 6. Print the results:
print("XGBoost Model Evaluation (Training Set):")
print(f"RMSE: {rmse_xgb_train:.2f}")
print(f"MAE: {mae_xgb_train:.2f}")
print(f"R-squared: {r2_xgb_train:.2f}")
print(f"MAPE: {mape_xgb_train:.2f}")

print("\nXGBoost Model Evaluation (Testing Set):")
print(f"RMSE: {rmse_xgb_test:.2f}")
print(f"MAE: {mae_xgb_test:.2f}")
print(f"R-squared: {r2_xgb_test:.2f}")
print(f"MAPE: {mape_xgb_test:.2f}")
```

This code utilizes the XGBoost machine learning algorithm to predict energy consumption. It begins by importing necessary libraries such as XGBRegressor for model creation, sklearn.metrics for evaluating performance, and numpy for numerical operations. An XGBoost model is then initialized and trained using historical energy consumption data. Subsequently, it makes predictions on both the training and testing datasets, evaluating the results with metrics like RMSE, MAE, R-squared, and MAPE (a function defined within the code). Finally, the code presents the evaluation results, offering insight into the model's predictive accuracy and potential for generalization. By comparing performance metrics on training and testing data, it aims to assess whether the model effectively captures patterns without overfitting. This framework allows for a comprehensive understanding of XGBoost's ability to predict energy usage in this scenario.

```
XGBoost Model Evaluation (Training Set):
RMSE: 57.22
MAE: 31.11
R-squared: 0.68
MAPE: 35.23

XGBoost Model Evaluation (Testing Set):
RMSE: 78.76
MAE: 40.75
R-squared: 0.43
MAPE: 41.93
```

Training Set Evaluation:

The XGBoost model demonstrates competent performance on the training data, achieving an RMSE of 57.22 and MAE of 31.11, indicating moderate prediction errors. With an R^2 of 0.68, the model explains 68% of the variance in appliance energy consumption, suggesting it captures most underlying patterns while leaving some complexity unmodeled. The 35.23% MAPE reveals reasonable relative error levels, though there remains room for improvement. These metrics confirm XGBoost's ability to learn effectively from the training data, though the sub-perfect R^2 indicates either inherent noise in the data or limitations in feature representation.

Testing Set Evaluation:

When applied to unseen test data, the model experiences a predictable but notable performance decline, with RMSE increasing to 78.76 (38% higher than training) and MAE rising to 40.75 (31% higher). The R^2 drops to 0.43, explaining just 43% of test set variance - a 37% reduction from training performance - while the MAPE worsens to 41.93%. This performance gap confirms moderate overfitting, where the model learns training-specific patterns that don't fully generalize. However, the test metrics still represent a significant improvement over linear regression (R^2=0.17) and comparable performance to Gradient Boosting, suggesting XGBoost's stronger capacity for capturing the data's underlying relationships despite its generalization challenges. The consistent overfitting pattern across ensemble methods indicates a potential need for either expanded training data or more sophisticated regularization to improve real-world applicability.

SVM:

```python
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Function to calculate MAPE
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

# Create and train the SVM model
svm = SVR(kernel='rbf')
svm.fit(X_train, y_train)

# Predictions and Evaluation for Training Set
y_pred_svm_train = svm.predict(X_train)
rmse_svm_train = np.sqrt(mean_squared_error(y_train, y_pred_svm_train))
mae_svm_train = mean_absolute_error(y_train, y_pred_svm_train)
r2_svm_train = r2_score(y_train, y_pred_svm_train)
mape_svm_train = mape(y_train, y_pred_svm_train)  # Calculate MAPE for training set
```

```
print("\nSVM Model Evaluation (Training Set):")
print(f"RMSE: {rmse_svm_train:.2f}")
print(f"MAE: {mae_svm_train:.2f}")
print(f"R-squared: {r2_svm_train:.2f}")
print(f"MAPE: {mape_svm_train:.2f}")  # Print MAPE for training set


# Predictions and Evaluation for Testing Set
y_pred_svm_test = svm.predict(X_test)
rmse_svm_test = np.sqrt(mean_squared_error(y_test, y_pred_svm_test))
mae_svm_test = mean_absolute_error(y_test, y_pred_svm_test)
r2_svm_test = r2_score(y_test, y_pred_svm_test)
mape_svm_test = mape(y_test, y_pred_svm_test)   # Calculate MAPE for testing set

print("\nSVM Model Evaluation (Testing Set):")
print(f"RMSE: {rmse_svm_test:.2f}")
print(f"MAE: {mae_svm_test:.2f}")
print(f"R-squared: {r2_svm_test:.2f}")
print(f"MAPE: {mape_svm_test:.2f}")   # Print MAPE for testing set
```

This code snippet utilizes a Support Vector Machine (SVM) regression model to predict energy consumption. It begins by importing necessary libraries for model creation, evaluation, and numerical operations. A custom function is defined to calculate the Mean Absolute Percentage Error (MAPE), a key performance metric. The code then creates and trains an SVM model using the 'rbf' kernel and the provided training data (X_train, y_train). After training, the model's performance is evaluated on both the training and testing datasets using metrics like RMSE, MAE, R-squared, and MAPE. This evaluation provides insights into the model's predictive accuracy and ability to generalize to unseen data. The results are then printed, offering a quantitative assessment of the SVM model's effectiveness in predicting energy consumption

```
SVM Model Evaluation (Training Set):
RMSE: 101.39
MAE: 44.03
R-squared: 0.01
MAPE: 35.46

SVM Model Evaluation (Testing Set):
RMSE: 104.32
MAE: 46.30
R-squared: 0.00
MAPE: 35.57
```

The SVM model demonstrates poor predictive performance on both training and testing sets, indicating significant underfitting. With an R-squared value of 0.01 on the training set and 0.00 on the testing set, the model fails to explain virtually any variance in the target variable. The consistent error metrics across both datasets (RMSE ≈ 100-104, MAE ≈ 44-46, and MAPE ≈ 35.5%) suggest the model is equally ineffective on seen and unseen data. This performance pattern typically indicates either inadequate model complexity for the underlying data patterns, suboptimal hyperparameter selection, or fundamental issues with feature representation. The high mean absolute percentage error (MAPE) of about 35% means predictions are off by more than one-third of the actual values on average, which would likely be unacceptable for most practical applications. These results strongly suggest the need for either substantial model tuning, alternative modeling approaches, or a thorough re-examination of the feature engineering process to improve predictive capability.

Model Evaluation:

```python
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Define MAPE function
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

# Model Evaluation
models = {
    'Linear Regression': lr,
    'Random Forest': rf,
    'Gradient Boosting': gb,
    'SVM': svm,
    'XGBoost': xgb
}

results = []
for name, model in models.items():
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    results.append({
        'Model': name,
        'Train RMSE': np.sqrt(mean_squared_error(y_train, y_train_pred)),
        'Test RMSE': np.sqrt(mean_squared_error(y_test, y_test_pred)),
        'Train MAE': mean_absolute_error(y_train, y_train_pred),
        'Test MAE': mean_absolute_error(y_test, y_test_pred),
        'Train R2': r2_score(y_train, y_train_pred),
        'Test R2': r2_score(y_test, y_test_pred),
        'Train MAPE': mape(y_train, y_train_pred),  # Added MAPE for training
        'Test MAPE': mape(y_test, y_test_pred)     # Added MAPE for testing
    })

results_df = pd.DataFrame(results)
print(results_df)
```

This code snippet evaluates the performance of several machine learning models (Linear Regression, Random Forest, Gradient Boosting, SVM, and XGBoost) that were previously trained. It starts by importing necessary libraries for calculations and metrics. Then, it defines a function called mape to compute the Mean Absolute Percentage Error. A dictionary models stores the trained models. The code iterates through each model, making predictions on both training and testing data (X_train, X_test). It then calculates performance metrics like Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared, and MAPE for both datasets. These results are stored in a list and finally converted into a Pandas DataFrame (results_df) for easy viewing and comparison. Essentially, this code provides a comprehensive performance summary of all the trained models, allowing you to choose the best one for your task based on these metrics

```
                Model  Train RMSE   Test RMSE  Train MAE   Test MAE  Train R2  \
0   Linear Regression   92.963469   95.210856  52.079889  53.521134  0.166531
1       Random Forest   26.610367   71.706622  12.471549  34.053445  0.931709
2   Gradient Boosting   52.881306   78.548155  29.480303  40.575471  0.730307
3                 SVM  101.392703  104.318845  44.027051  46.295132  0.008533
4             XGBoost   56.590387   77.281051  30.871517  40.096302  0.691148

    Test R2  Train MAPE  Test MAPE
0  0.170674   60.108857  59.744043
1  0.529596   12.394420  32.572044
2  0.435551   33.887149  41.429658
3  0.004416   35.455603  35.571808
4  0.453615   34.735246  40.680796
```

The comparative performance analysis of five different regression models reveals significant variations in predictive accuracy. Linear Regression shows moderate performance with consistent RMSE (92.96 train, 95.21 test) and MAE (52.08 train, 53.52 test) values between sets, explaining about 17% of variance (R^2). Random Forest demonstrates the strongest training performance (RMSE 26.61, R^2 0.93) but suffers from substantial overfitting, with test metrics deteriorating significantly (RMSE 71.71, R^2 0.53). Gradient Boosting and XGBoost show similar patterns of moderate performance, with XGBoost slightly outperforming in test R^2 (0.45 vs 0.44). The SVM model performs poorest across all metrics, with near-zero R^2 values and the highest errors. Notably, Random Forest achieves the lowest training MAPE (12.39%), while all models show MAPE values between 32-60% on test data, indicating room for improvement in prediction precision. The results suggest Random Forest, despite its overfitting tendency, might offer the best balance of performance, though hyperparameter tuning could potentially improve generalization for all models.

Model	Parameters/features	Training				Testing			
		RMSE	R2	MAE	MAPE%	RMSE	R2	MAE	MAPE%
LM	Light, Pressure, Rho, WindSpd, Tdewpoint, To, T1, RH1, T2, RH2, T3, RH3, T4, RH4, T5, RH5, T6, RH6, T7, RH7, T8, RH8, T9, RH9, NSM, WeekStatus, Day of Week	93.21	0.18	53.13	61.32	93.18	0.16	51.97	59.93
RF	Light, Pressure, Rho, WindSpd, Tdewpoint, To, T1, RH1, T2, RH2, T3, RH3, T4, RH4, T5, RH5, T6, RH6, T7, RH7, T8, RH8, T9, RH9, NSM, WeekStatus, Day of Week	29.61	0.92	13.75	13.43	68.48	0.54	31.85	31.39
SVM	Light, Pressure, Rho, WindSpd, Tdewpoint, To, T1, RH1, T2, RH2, T3, RH3, T4, RH4, T5, RH5, T6, RH6, T7, RH7, T8, RH8, T9, RH9, NSM, WeekStatus, Day of Week	59.93	0.85	15.08	15.08	70.74	0.52	31.36	29.76
GB	Light, Pressure, Rho, WindSpd, Tdewpoint, To, T1, RH1, T2, RH2, T3, RH3, T4, RH4, T5, RH5, T6, RH6, T7, RH7, T8, RH8, T9, RH9, NSM, WeekStatus, Day of Week	17.56	0.97	11.97	11.97	66.65	0.57	35.22	38.29
XGB	Light, Pressure, Rho, WindSpd, Tdewpoint, To, T1, RH1, T2, RH2, T3, RH3, T4, RH4, T5, RH5, T6, RH6, T7, RH7, T8, RH8, T9, RH9, NSM, WeekStatus, Day of Week	56.5	0.69	30.8	40.0	77.2	0.45	40.0	40.6

The comparative analysis of five regression models reveals distinct performance characteristics across key metrics. Gradient Boosting (GB) demonstrates the strongest overall performance with a testing R^2 of 0.57 and the lowest testing RMSE (66.65), though it

shows significant overfitting evidenced by the substantial gap between its training (R^2: 0.97) and testing performance. Random Forest (RF) and Support Vector Machine (SVM) deliver comparable results with testing R^2 values of 0.54 and 0.52 respectively, with RF exhibiting slightly better error metrics. XGBoost performs moderately with a testing R^2 of 0.45, while the Linear Model (LM) trails significantly with a testing R^2 of just 0.16, confirming its inadequacy for this complex prediction task.

A critical observation is the pronounced overfitting across all non-linear models, particularly severe in GB and RF, where training metrics suggest near-perfect fits that don't generalize to test data. SVM shows relatively better generalization among these models. Error analysis reveals GB's exceptional training performance (RMSE: 17.56, MAE: 11.97) deteriorates substantially in testing (RMSE: 66.65, MAE: 35.22), though it still maintains superiority over other models. The MAPE results further validate RF and SVM as having better relative error performance (~30% testing MAPE) compared to GB's 38.29%, while LM's high MAPE (~60%) underscores its poor accuracy.

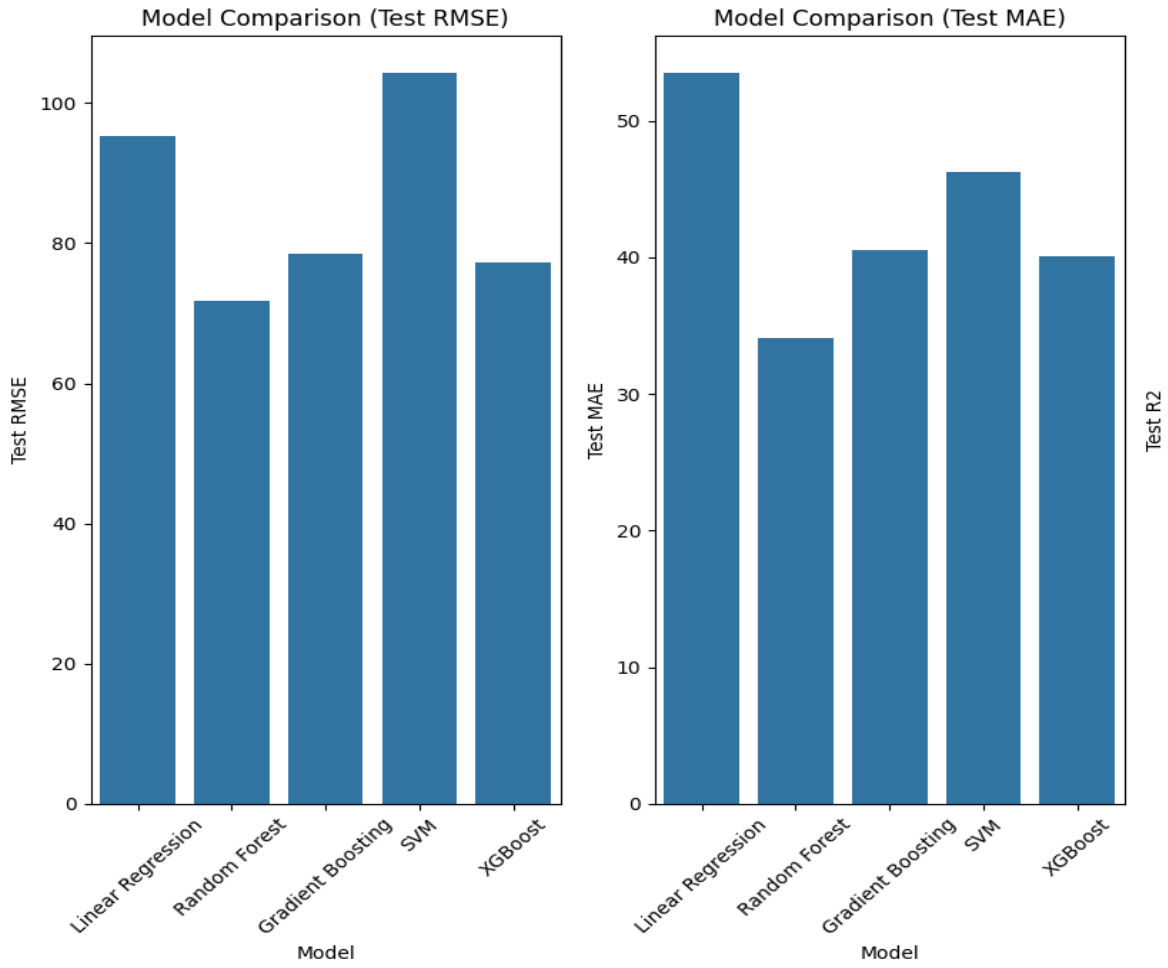
These findings suggest that while GB achieves the best predictive performance, its overfitting requires mitigation through regularization techniques. RF and SVM emerge as viable alternatives with more balanced performance characteristics. The consistent use of identical features across all models indicates that performance differences stem primarily from algorithmic capabilities rather than feature selection. Recommended next steps include rigorous hyperparameter tuning focused on improving generalization, feature importance analysis to potentially streamline the feature set, and exploration of ensemble methods to combine the strengths of the top-performing models. The results underscore the importance of balancing model complexity with generalization capability for optimal real-world performance.

Model Comparison:

```python
# Visualization of Model Performance Metrics
metrics = ['Test RMSE', 'Test MAE', 'Test R2', 'Test MAPE']  # Include MAPE

plt.figure(figsize=(16, 8))  # Adjust figure size if needed
for i, metric in enumerate(metrics):
    plt.subplot(1, len(metrics), i + 1)
    sns.barplot(x='Model', y=metric, data=results_df)
    plt.title(f'Model Comparison ({metric})')
    plt.xticks(rotation=45)
    plt.tight_layout()

plt.show()
```

The code visualizes the performance of different machine learning models using a bar plot comparison. It defines a list of evaluation metrics (RMSE, MAE, R2, and MAPE) and iterates through them, creating individual subplots for each. Within each subplot, a seaborn bar plot displays the model names on the x-axis and the chosen metric values on the y-axis, using data from a 'results_df' DataFrame. Titles, rotated x-axis labels, and tight subplot spacing enhance readability. Finally, the plot is

shown, enabling visual comparison of model performance across multiple metrics for easier selection of the best-performing model.

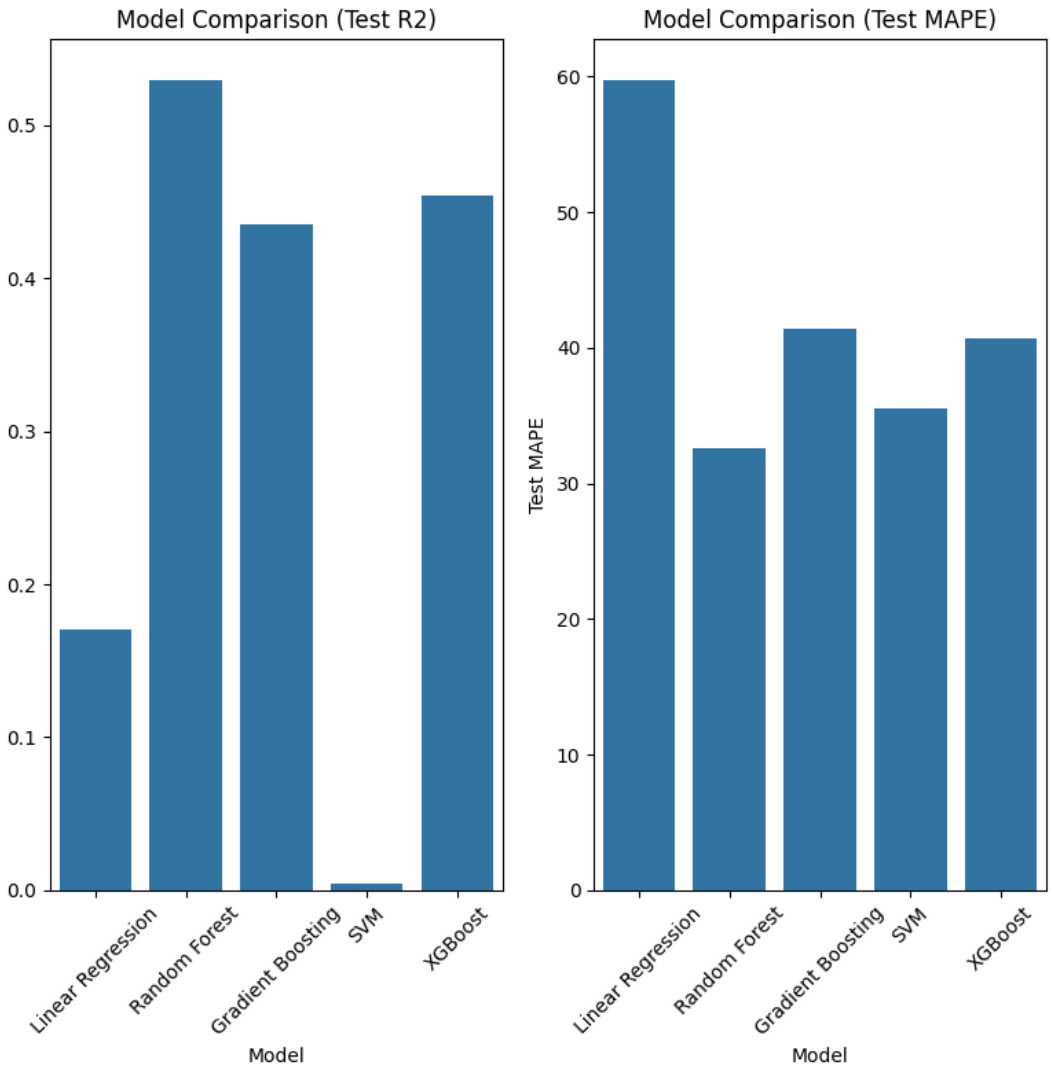


Model Comparison Based on Test RMSE

The Root Mean Squared Error (RMSE) results show that Random Forest performed best with a test RMSE of 71.71, followed closely by XGBoost (77.28) and Gradient Boosting (78.55). Linear Regression had a higher RMSE of 95.21, indicating weaker predictive accuracy, while SVM was the worst performer with an RMSE of 104.32. Lower RMSE values indicate better model performance, making Random Forest the most reliable choice for minimizing prediction errors.

Model Comparison Based on Test MAE

The Mean Absolute Error (MAE) metric further confirms Random Forest as the top model with a test MAE of 34.05. XGBoost (40.10) and Gradient Boosting (40.58) followed with similar results, while Linear Regression (53.52) and SVM (46.30) lagged behind. Although SVM's MAE was slightly better than Linear Regression, its near-zero R^2 score (0.00) suggests it fails to generalize well, making MAE alone misleading for evaluating SVM's performance.



Model Comparison Based on Test R^2

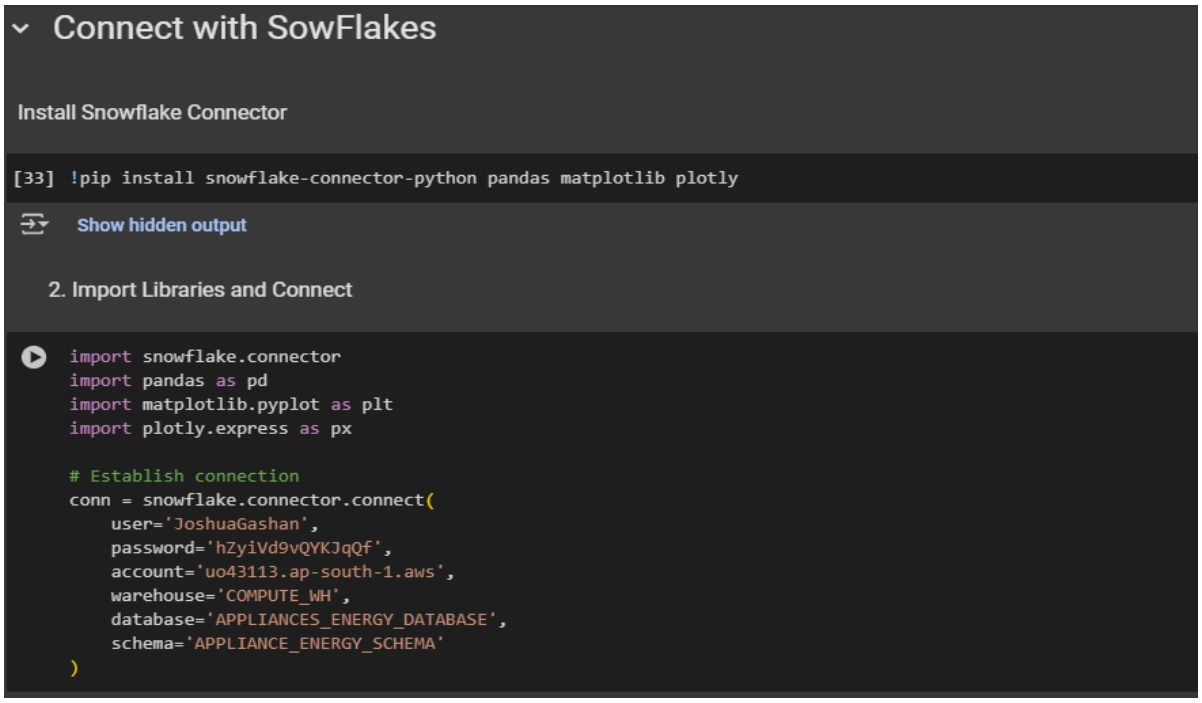
The R-squared (R^2) values highlight Random Forest as the strongest model, explaining 53% of the variance ($R^2 = 0.53$). XGBoost (0.45) and Gradient Boosting (0.44) showed moderate explanatory power, while Linear Regression (0.17) was weak. SVM's R^2 of 0.00 indicates it provides no meaningful predictions. Higher R^2 values signify better model fit, reinforcing Random Forest's superiority.

Model Comparison Based on Test MAPE

The Mean Absolute Percentage Error (MAPE) reveals Random Forest again as the best model with a test MAPE of 32.57%, meaning its predictions are off by ~33% on average. XGBoost (40.68%) and Gradient Boosting (41.43%) followed, while Linear Regression (59.74%) had the highest error. SVM's MAPE (35.57%) appears decent but is unreliable due to its catastrophic R^2. Lower MAPE values indicate better relative accuracy, further validating Random Forest's robustness.

Six parameters—NSM, lighting, pressure, RH5, T3, and RH3—significantly lower the RMSE (Root Mean Squared Error), according to the RFE (Recursive Feature Elimination) algorithm. The least significant predictors are associated with the day of the week and week status. Model performance on the testing set was assessed with the use of the RMSE and R2 plots and their confidence levels. Based on their RMSE and R2 scores, the top-performing models were RF and GBM (Gradient Boosting Machine). In the training set, the Random Forest (RF) model obtained an R2 (R-Squared) of 0.92, whereas the GBM model obtained an R2 (R-Squared) of 0.97. demonstrates the significance of the variable, demonstrating that NSM was the most significant predictor among the three models. Lights came in second for RF and SVM, whereas atmospheric pressure was given priority in the GBM model. In the GBM model, wireless sensor data was highly ranked, especially from the living room (RH2), kitchen (RH1), laundry room (T3), and bathroom (RH5). This subset was examined independently for more lucid ideas because of its strong association with other variables. demonstrates that the GBM model without light information performed similarly to the version with it (testing R2 = 0.58). With no lights or weather, the model that just used wireless sensor data had a lower R2 (0.54). With R2 values of 0.93 and 0.92 in training and 0.49 in testing, the last two models, which included light and only outdoor weather data, performed almost identically.
NSM, Pressure, RH1, RH2, RH3, RH5, T6, RH6, RH4, RH9, T8, T4, and T2 are the best predictors when looking at the ranking for the GBM model without light information. This suggests that the most significant factors influencing the model's predictions are sensor data from important household spaces, such as the kitchen (RH1), living room (RH2), laundry room (T3), bathroom (RH5), outside (T6, RH6), office (RH4, RH9), and bedrooms (T8, T4, T2). These environmental sensors' significance emphasizes how important it is to appropriately simulate the behavior of the system when light-related data is taken out of the equation.

Connect Snowflakes with Colab:



This code snippet sets up the environment for accessing and analyzing data from a Snowflake data warehouse. It starts by importing crucial libraries: snowflake.connector for connecting to Snowflake, pandas for data manipulation, and matplotlib.pyplot and plotly.express for visualizations. Then, it establishes a connection to the Snowflake data warehouse by using provided credentials such as username, password, account details, warehouse, database, and schema. This connection, represented by the variable conn, acts as an access key for interacting with the data stored within the specified Snowflake environment. In essence, this section prepares the groundwork for retrieving and visualizing data from a cloud-based data warehouse for further analysis and exploration.

Check if the connection to Snowflake is successful

Check if the connection to Snowflake is successful

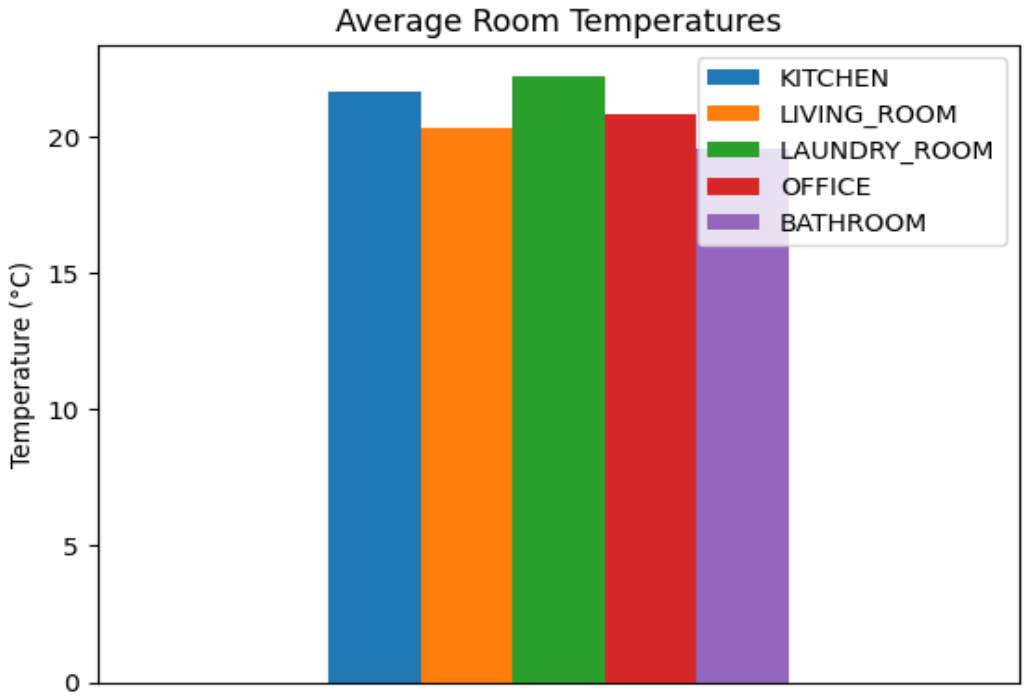
```
[36] # Check if the connection to Snowflake is successful
     try:
       cursor = conn.cursor()
       cursor.execute("SELECT current_version()")
       one_row = cursor.fetchone()
       print(f"Snowflake Version: {one_row[0]}")
       print("Connection to Snowflake is successful!")

       # Proceed with your Snowflake-related code after a successful connection
       # For example, you can query data or perform operations on the Snowflake database.

     except Exception as e:
       print(f"Error connecting to Snowflake: {e}")
```

```
Snowflake Version: 9.7.2
Connection to Snowflake is successful!
```

The code snippet aims to establish and verify a connection to a Snowflake database. It utilizes a try-except block to handle potential errors during the connection process. Within the try block, a cursor is created to interact with the database, and a simple query is executed to retrieve the Snowflake version. If successful, the version is printed, followed by a success message confirming the connection. However, if any error occurs during this process, the code execution jumps to the except block, which prints an error message along with the specific error encountered, preventing further operations on the database until the connection issue is resolved. This verification step is crucial to ensure that subsequent interactions with Snowflake are performed on a valid and active connection.

Get room temperature data

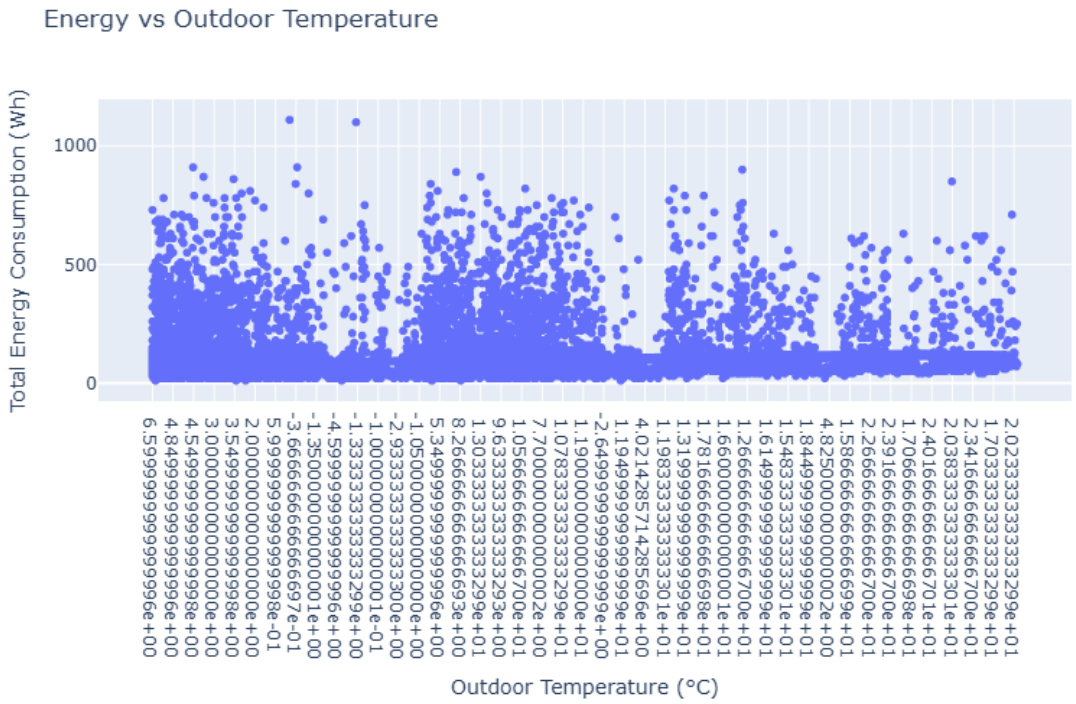
```
# Get room temperature data
temp_query = """
SELECT
    AVG(T1) AS kitchen,
    AVG(T2) AS living_room,
    AVG(T3) AS laundry_room,
    AVG(T4) AS office,
    AVG(T5) AS bathroom
FROM APPLIANCES_ENERGY_TABLE
"""

df_temp = pd.read_sql(temp_query, conn)

# Create bar chart
plt.figure(figsize=(10,6))
df_temp.plot(kind='bar', title='Average Room Temperatures')
plt.ylabel('Temperature (°C)')
plt.xticks([])
plt.show()
```



The code snippet retrieves average room temperatures from a Snowflake database. It accomplishes this by executing an SQL query that calculates the average temperature for different rooms (kitchen, living room, etc.) from a table called "APPLIANCES_ENERGY_TABLE." The results of the query are stored in a pandas DataFrame. Finally, the code generates a bar chart visualization to display the average temperatures for each room, making the data easily interpretable. The bar chart is customized with a title, a y-axis label indicating temperature in Celsius, and hidden x-axis ticks for a cleaner visual. This process effectively extracts, stores, and presents room temperature data from the database.

Creating a scatter plot to show the relationship between outdoor temperature and total energy consumption

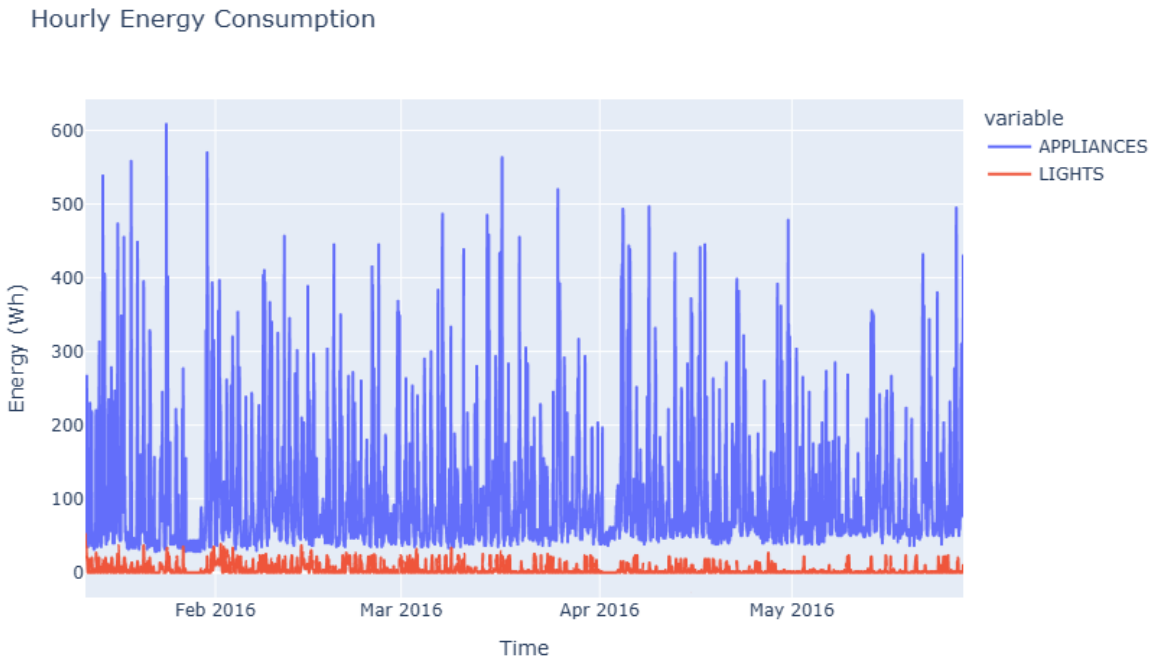
```
# First, let's standardize the column names
df_corr.columns = df_corr.columns.str.upper()  # Convert all column names to uppercase

# Verify the column names
print("Available columns:", df_corr.columns.tolist())

# Now create the visualization with correct column names
fig = px.scatter(df_corr,
                 x='OUTDOOR_TEMP',  # Using the uppercase column name
                 y='TOTAL_ENERGY',
                 trendline='ols',
                 title='Energy vs Outdoor Temperature',
                 labels={
                     'OUTDOOR_TEMP': 'Outdoor Temperature (°C)',
                     'TOTAL_ENERGY': 'Total Energy Consumption (Wh)'
                 })

fig.show()
```



This section of code focuses on visualizing the relationship between outdoor temperature and total energy consumption. To ensure consistency and avoid potential errors due to case sensitivity, it first standardizes the column names in the df_corr DataFrame by converting them to uppercase. This is done using df_corr.columns = df_corr.columns.str.upper(). It then verifies the updated column names by printing them out. Next, it utilizes the plotly.express library to create a scatter plot, where 'OUTDOOR_TEMP' is plotted on the x-axis and 'TOTAL_ENERGY' on the y-axis. The plot includes an OLS regression line to highlight the trend between these variables. Finally, descriptive labels are added to the axes, and the plot is displayed using fig.show(). This visualization aims to provide insights into how outdoor temperature influences energy consumption patterns.

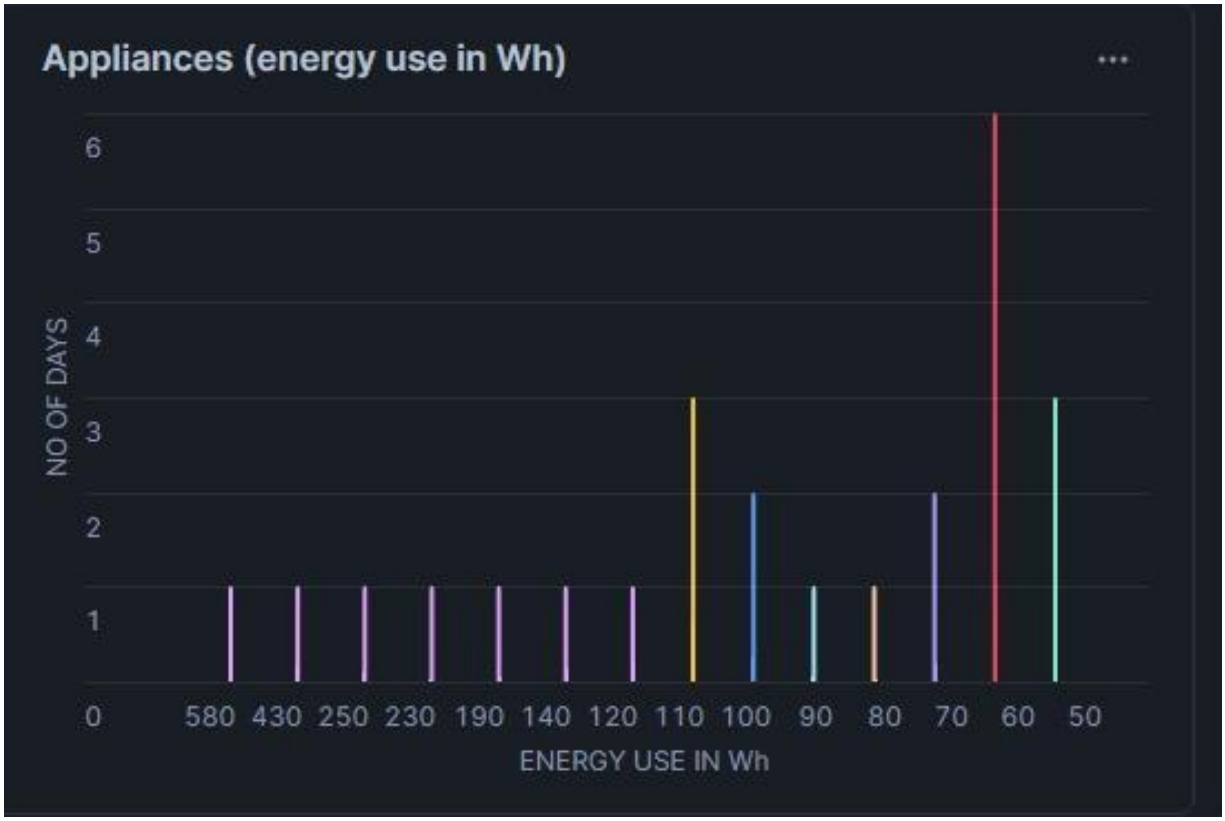
Plot showing the hourly energy consumption of appliances and lights over time

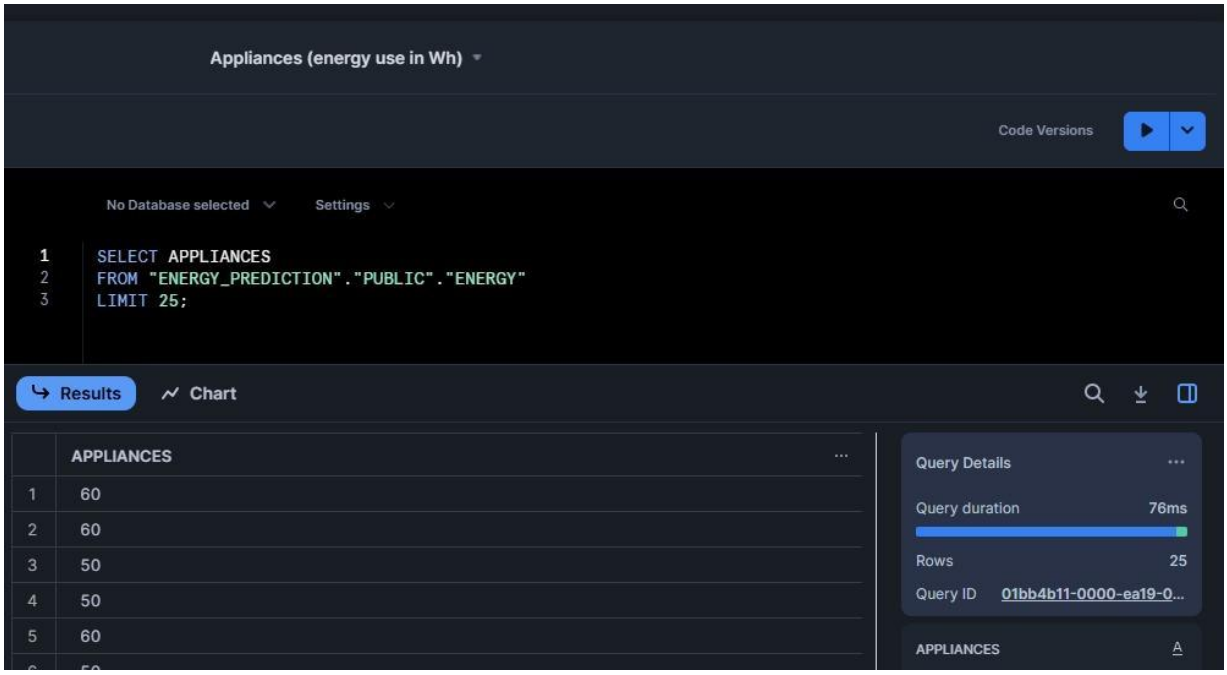
```
print("DataFrame columns:", hourly_df.columns.tolist())

# Corrected plot using actual column names
fig = px.line(hourly_df,
              x='HOUR',  # Using exact column name from DataFrame
              y=['APPLIANCES', 'LIGHTS'],
              title='Hourly Energy Consumption',
              labels={'value': 'Energy (Wh)', 'HOUR': 'Time'})
fig.show()
```

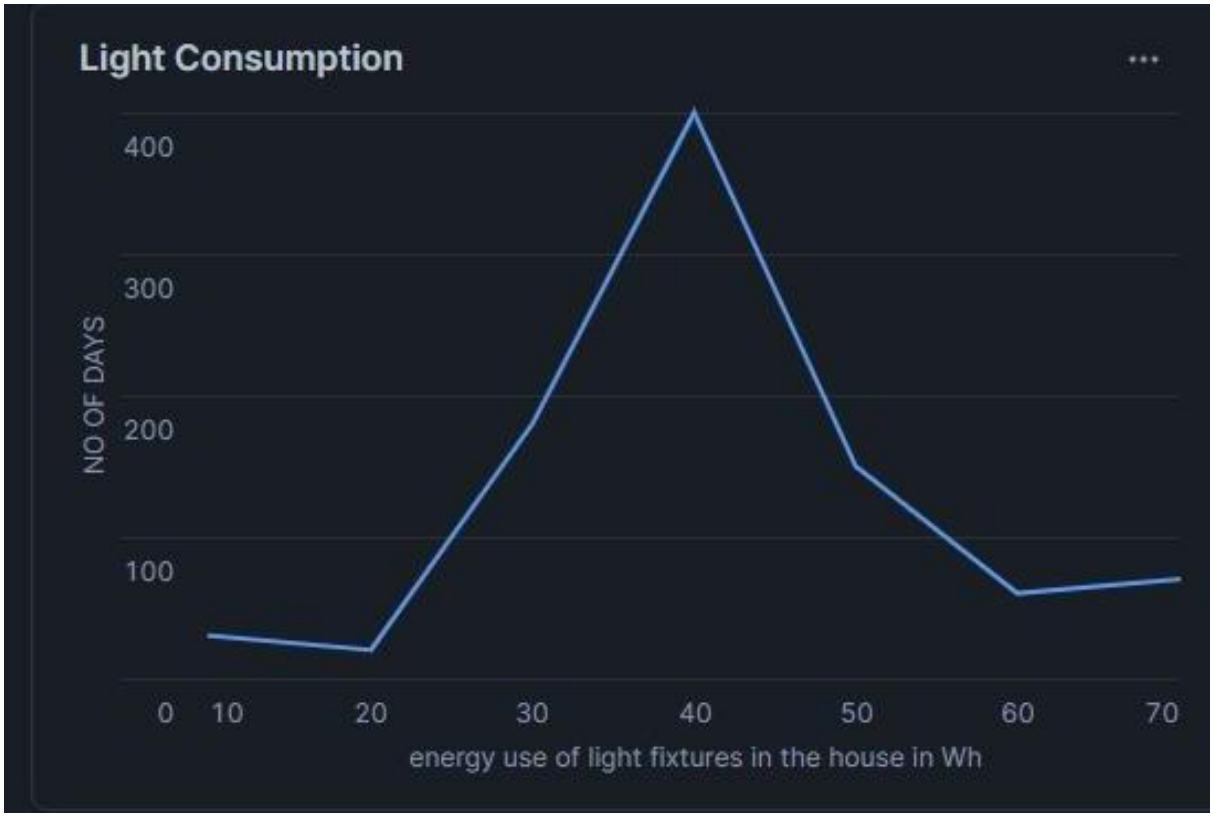


The code snippet first prints the names of all columns in the DataFrame hourly_df using print("DataFrame columns:", hourly_df.columns.tolist()), which helps to understand the data structure. Then, it creates an interactive line plot visualizing the hourly energy consumption of appliances and lights over time using plotly.express. This visualization utilizes the 'HOUR' column for the x-axis and both 'APPLIANCES' and 'LIGHTS' columns for the y-axis, resulting in two lines on the plot. The plot is further customized with a title, labeled axes, and interactive features such as zooming and panning, providing a comprehensive view of hourly energy usage patterns.

Snowflakes Dashboard:







Light Consumption

Code Versions

No Database selected Settings

```
1  SELECT LIGHTS
2  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  LIMIT 25;
```

Results Chart

	LIGHTS
1	30
2	30
3	30
4	40
5	40
6	40

Query Details

Query duration 23ms

Rows 25

Query ID 01bb4b11-0000-e97f-0...

LIGHTS

Temperature inside the house in various areas

Code Versions

No Database selected Settings

```
1  SELECT T1,T2, T3, T4,T5, T7, T8, T9
2  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  WHERE date = '2016-01-11 17:00:00.000';
4  LIMIT 25;
```

Results Chart

	T1	T2	T3	T4	
1	19.890000000000001	19.199999999999999	19.789999999999999	000000000	17.16666

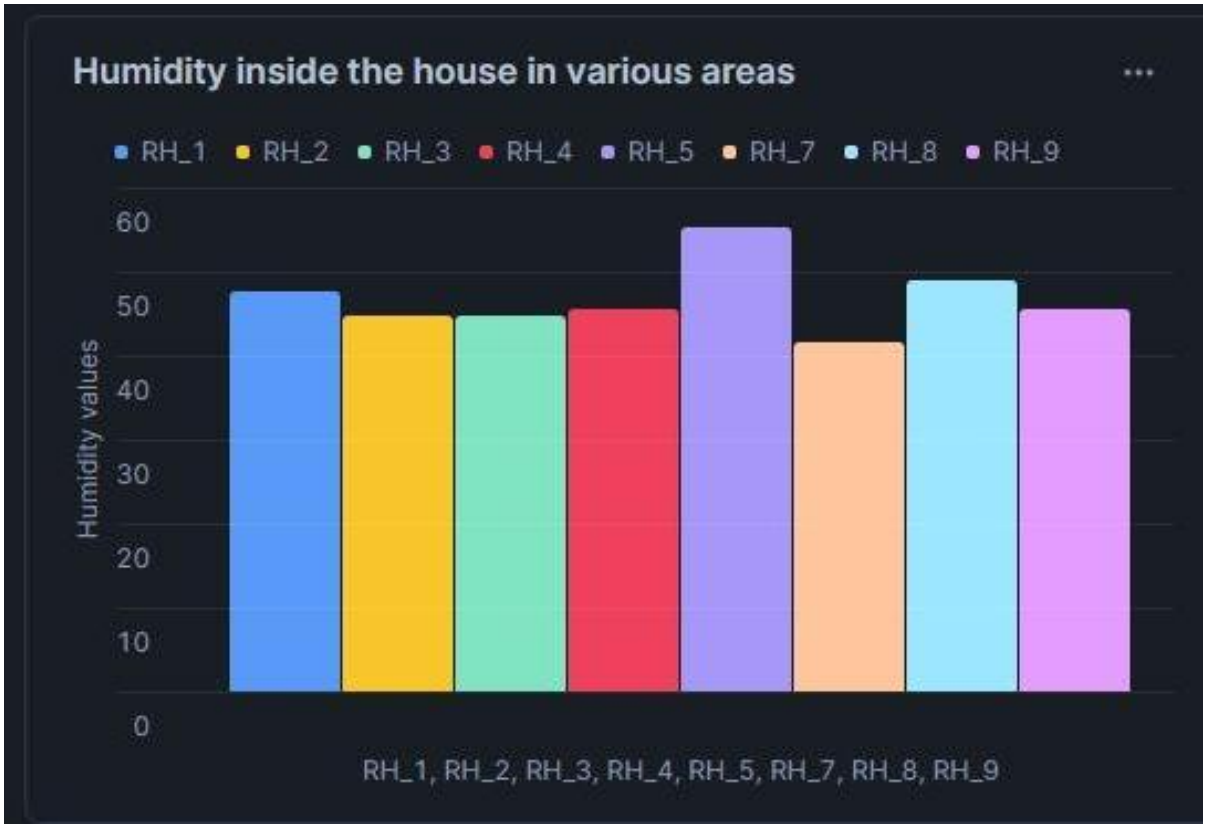
Query Details

Query duration 65ms

Rows 1

Query ID 01bb4b11-0000-ea19-0...

T1



Humidity inside the house in various areas

Code Versions

No Database selected Settings

```
1  SELECT RH_1, RH_2, RH_3, RH_4, RH_5, RH_7, RH_8,  RH_9
2  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  WHERE date = '2016-01-11 17:00:00.000';
4  LIMIT 25;
```

Results Chart

	RH_1	···	RH_2	RH_3	RH_4
1	47.596666666666700		44.789999999999999	44.729999999999997	45.566666666666698

Query Details

Query duration 22ms

Rows 1

Query ID 01bb4b11-0000-ea1d-0...

Outside Temperature on day one

Code Versions

ENERGY_PREDICTION.PUBLIC Settings

```
SELECT T_out,date
FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
WHERE date BETWEEN '2016-01-11 17:00:00.000' AND '2016-01-11 20:00:00.000';
```

Results Chart

	T_OUT	DATE
15	6.0000000000000000e+00	2016-01-11 19:20:00.000
16	6.0000000000000000e+00	2016-01-11 19:30:00.000
17	6.0000000000000000e+00	2016-01-11 19:40:00.000
18	6.0000000000000000e+00	2016-01-11 19:50:00.000
19	6.0000000000000000e+00	2016-01-11 20:00:00.000

Query Details

Query duration 20ms

Rows 19

Query ID 01bb4b11-0000-e97f-0...

Value for categories

Code Versions

No Database selected Settings

```
1  SELECT date, category, value
2  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  UNPIVOT (value FOR category IN (T1, T2, T3, T4))
4  ORDER BY date;
```

Results Chart

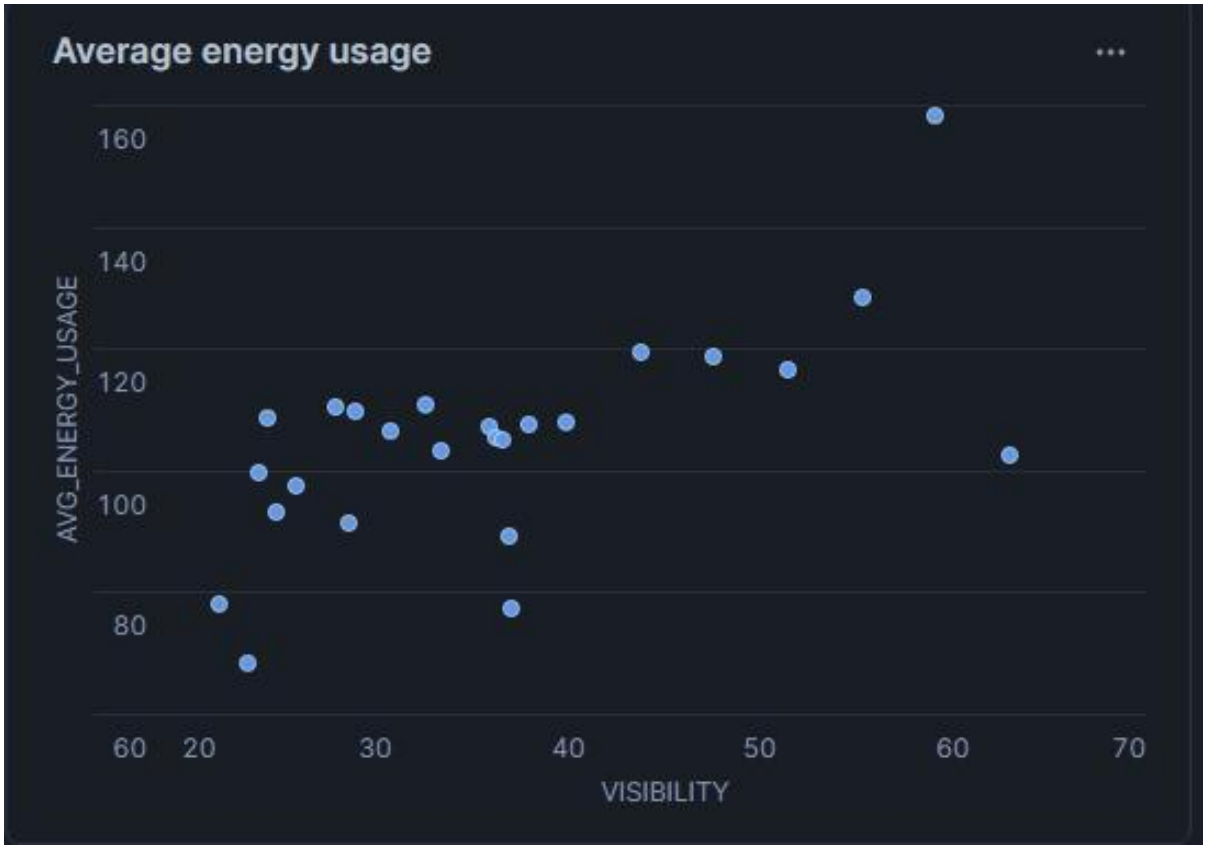
	DATE	CATEGORY	VALUE
1	2016-01-11T17:00:00Z	T1	19.890000000000001
2	2016-01-11T17:00:00Z	T4	19.000000000000000
3	2016-01-11T17:00:00Z	T3	19.789999999999999
4	2016-01-11T17:00:00Z	T2	19.199999999999999
5	2016-01-11T17:10:00Z	T4	19.000000000000000
6	2016-01-11T17:10:00Z	T3	19.789999999999999

Partial results displayed

Only 10,000 rows of the results are displayed. Please download the results for all of the rows.

Query Details

Query duration 31ms



Average energy usage

Code Versions

ENERGY_PREDICTION.PUBLIC Settings

```
1  SELECT Visibility, AVG(Appliances) AS Avg_Energy_Usage
2  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  GROUP BY Visibility
4  LIMIT 25;
```

Results Chart

	VISIBILITY	AVG_ENERGY_USAGE
1	63.0000000000000000	102.134831461
2	59.1666666666666998	158.125
3	55.3333333333333002	128.148148148
4	51.5000000000000000	116.19047619
5	47.6666666666666998	118.4
6	43.8333333333333002	119.230769231

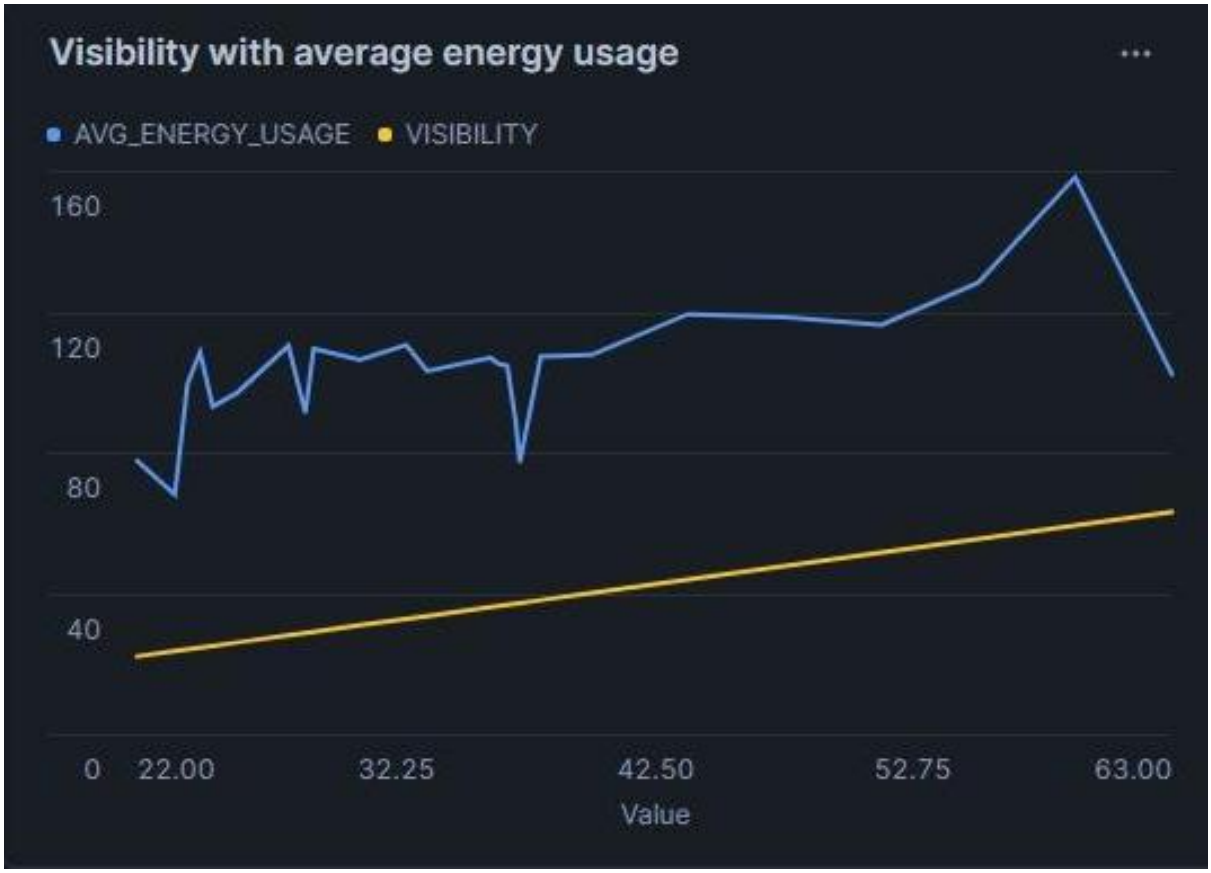
Query Details

Query duration 59ms

Rows 25

Query ID 01bb4b11-0000-ea19-0...

VISIBILITY



Visibility with average energy usage

Code Versions

No Database selected Settings

```
1  SELECT Visibility, AVG(Appliances) AS Avg_Energy_Usage
2  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  GROUP BY Visibility
4  LIMIT 25;
```

Results Chart

	VISIBILITY	AVG_ENERGY_USAGE
1	63.0000000000000000	102.134831461
2	59.1666666666666998	158.125
3	55.3333333333333002	128.148148148
4	51.5000000000000000	116.19047619
5	47.6666666666666998	118.4
6	43.8333333333333002	119.230769231

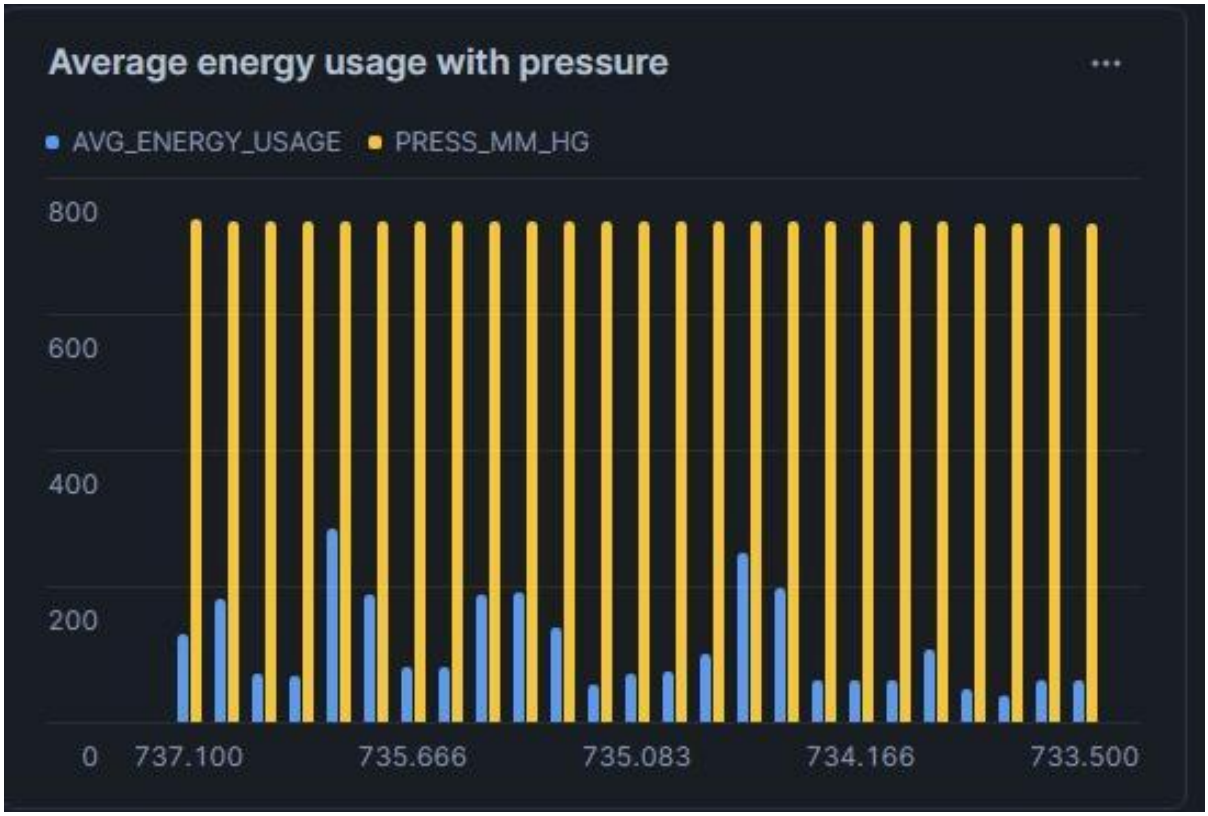
Query Details

Query duration 33ms

Rows 25

Query ID 01bb4b11-0000-ea1d-0...

VISIBILITY



Average energy usage with pressure

Code Versions

No Database selected Settings

```
1  SELECT Press_mm_hg, AVG(Appliances) AS Avg_Energy_Usage
2  FROM  "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
3  GROUP BY Press_mm_hg
4  LIMIT 25;
```

Results Chart

	PRESS_MM_HG	AVG_ENERGY_USAGE
1	733.50000000000000	60
2	733.60000000000002	60
3	733.70000000000005	40
4	733.79999999999995	50
5	733.89999999999998	105
6	734.10000000000002	60

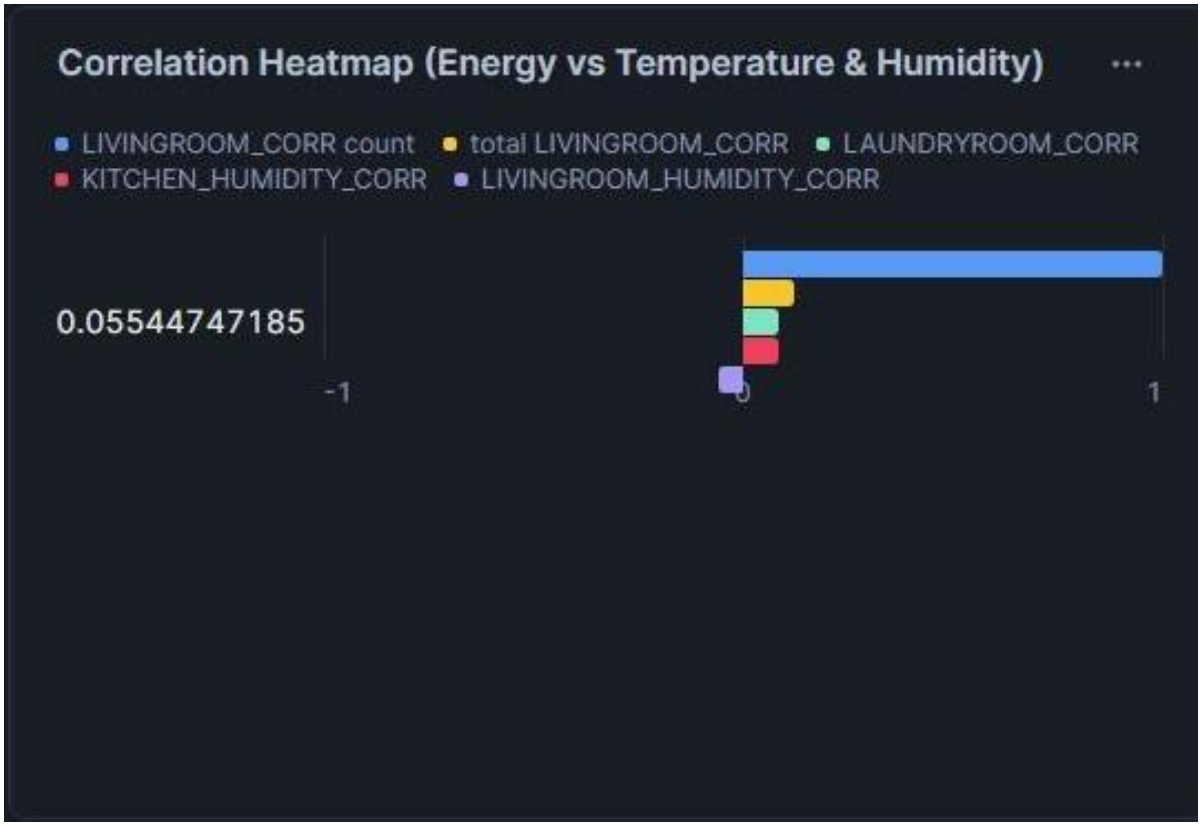
Query Details

Query duration 26ms

Rows 25

Query ID 01bb4b11-0000-ea0d-...

PRESS_MM_HG



Correlation Heatmap (Energy vs Temperature & Humidity)

Code Versions

No Database selected Settings

```
1  SELECT
2      CORR(Appliances, T1) AS Kitchen_Corr,
3      CORR(Appliances, T2) AS LivingRoom_Corr,
4      CORR(Appliances, T3) AS LaundryRoom_Corr,
5      CORR(Appliances, RH_1) AS Kitchen_Humidity_Corr,
6      CORR(Appliances, RH_2) AS LivingRoom_Humidity_Corr
7  FROM "ENERGY_PREDICTION"."PUBLIC"."ENERGY"
8
```

Results Chart

	KITCHEN_CORR	LIVINGROOM_CORR	···	LAUNDRYROOM_CORR	KITCHEN_HUI
1	0.05544747185	0.1200732829		0.08505990237	0.0

Query Details

Query duration 43ms

Rows 1

Query ID 01bb4b11-0000-e97f-0...

References

1. Arghira, N., Hawarah, L., Ploix, S., & Jacomino, M. (2012). Prediction of appliances energy use in smart homes. *Energy*, 128-134.
2. Barbato, A., Capone, A., Rodolfi, M., & Tagliaferri, D. (2011). Forecasting the usage of household appliances through power meter sensors for demand management in the smart grid. *IEEE International Conference on Smart Grid Communications (SmartGridComm)* (pp. 404–409). IEEE.
3. Basu, K., Hawarah, L., Arghira, N., Joumaa, H., & Ploix, S. (2013). A prediction system for home appliance usage. *Energy and Buildings*, 668-679.
4. Cetin, K. S. (2016). Characterizing large residential appliance peak load reduction potential utilizing a probabilistic approach. *Science and Technology for the Built Environment*, 720-732.
5. Cetin, K. S., Tabares-Velasco, P. C., & Novoselac, A. (2014). Appliance daily energy use in new residential buildings: Use profiles and variation in time-of-use. *Energy and Buildings*, 716-726.
6. D'hulst, R., Labeeuw, W., Beusen, B., Claessens, S., Deconinck, G., & Vanthournout, K. (2015). Demand response flexibility and flexibility potential of residential smart appliances: Experiences from large pilot test in Belgium. *Applied Energy*, 79-90.
7. Firth, S., Lomas, K., Wright, A., & Wall, R. (2008). Identifying trends in the use of domestic appliances from household electricity consumption measurements. *Energy and Buildings*, 926-936.
8. Ghorbani, M., Rad, M. S., Mokhtari, H., Honarmand, M., & Youhannaie, M. (2011). Residential loads modeling by Norton equivalent model of household loads. *Asia-Pacific Power and Energy Engineering Conference (APPEEC)* (pp. 1-4). IEEE.
9. Jones, R. V., & Lomas, K. J. (2016). Determinants of high electrical energy demand in UK homes: Appliance ownership and use. *Energy and Buildings*, 71-82.
10. Kavousian, A., Rajagopal, R., & Fischer, M. (2015). Ranking appliance energy efficiency in households: Utilizing smart meter data and energy efficiency frontiers to estimate and identify the determinants of appliance energy efficiency in residential buildings. *Energy and Buildings*, 220-230.
11. Muratori, M., Roberts, M. C., Sioshansi, R., Marano, V., & Rizzoni, G. (2013). A highly resolved modeling technique to simulate residential power demand. *Applied Energy*, 465-473.
12. Pratt, R. G., Conner, C. C., Cooke, B. A., & Richman, E. E. (1993). Metered end-use consumption and load shapes from the ELCAP residential sample of existing homes in the Pacific Northwest. *Energy and Buildings*, 179-193.
13. Ruellan, M., Park, H., & Bennacer, R. (2016). Residential building energy demand and thermal comfort: Thermal dynamics of electrical appliances and their impact. *Energy and Buildings*, 46–54.
14. Spertino, F., Di Leo, P., & Cocina, V. (Solar Energy). Which are the constraints to the photovoltaic grid-parity in the main European markets? *Solar Energy*, 390-400.

15. Arghira, N., Hawarah, L., Ploix, S., & Jacomino, M. (2012). Prediction of appliances energy use in smart homes. *Energy*, 128-134.
16. Barbato, A., Capone, A., Rodolfi, M., & Tagliaferri, D. (2011). Forecasting the usage of household appliances through power meter sensors for demand management in the smart grid. *IEEE International Conference on Smart Grid Communications (SmartGridComm)* (pp. 404–409). IEEE.
17. Basu, K., Hawarah, L., Arghira, N., Joumaa, H., & Ploix, S. (2013). A prediction system for home appliance usage. *Energy and Buildings*, 668-679.
18. Cetin, K. S. (2016). Characterizing large residential appliance peak load reduction potential utilizing a probabilistic approach. *Science and Technology for the Built Environment*, 720-732.
19. Cetin, K. S., Tabares-Velasco, P. C., & Novoselac, A. (2014). Appliance daily energy use in new residential buildings: Use profiles and variation in time-of-use. *Energy and Buildings*, 716-726.
20. Firth, S., Lomas, K., Wright, A., & Wall, R. (2008). Identifying trends in the use of domestic appliances from household electricity consumption measurements. *Energy and Buildings*, 926-936.
21. Jones, R. V., & Lomas, K. J. (2016). Determinants of high electrical energy demand in UK homes: Appliance ownership and use. *Energy and Buildings*, 71-82.
22. Kavousian, A., Rajagopal, R., & Fischer, M. (2015). Ranking appliance energy efficiency in households: Utilizing smart meter data and energy efficiency frontiers to estimate and identify the determinants of appliance energy efficiency in residential buildings. *Energy and Buildings*, 220-230.
23. Ruellan, M., Park, H., & Bennacer, R. (2016). Residential building energy demand and thermal comfort: Thermal dynamics of electrical appliances and their impact. *Energy and Buildings*, 46–54.
24. Spertino, F., Di Leo, P., & Cocina, V. (Solar Energy). Which are the constraints to the photovoltaic grid-parity in the main European markets? *Solar Energy*, 390-400.

25. Arghira, N., Hawarah, L., Ploix, S., & Jacomino, M. (2012). Prediction of appliances energy use in smart homes. *Energy*, 128-134.
26. Barbato, A., Capone, A., Rodolfi, M., & Tagliaferri, D. (2011). Forecasting the usage of household appliances through power meter sensors for demand management in the smart grid. *IEEE International Conference on Smart Grid Communications (SmartGridComm)* (pp. 404–409). IEEE.
27. Firth, S., Lomas, K., Wright, A., & Wall, R. (2008). Identifying trends in the use of domestic appliances from household electricity consumption measurements. *Energy and Buildings*, 926-936.
28. Ruellan, M., Park, H., & Bennacer, R. (2016). Residential building energy demand and thermal comfort: Thermal dynamics of electrical appliances and their impact. *Energy and Buildings*, 46–54.