AI_PHASE 5

MEASURE ENERGY CONSUMPTION

DATE	**1 November 2023**
TEAM MEMBER	**ILAMARAN E.**
REG NO	**950621104301**
TEAM ID	**PROJ_212174_TEAM_4**
PROJECT NAME	**MEASURE ENERGY CONSUMPTION**
MAXIMUM MARKS	

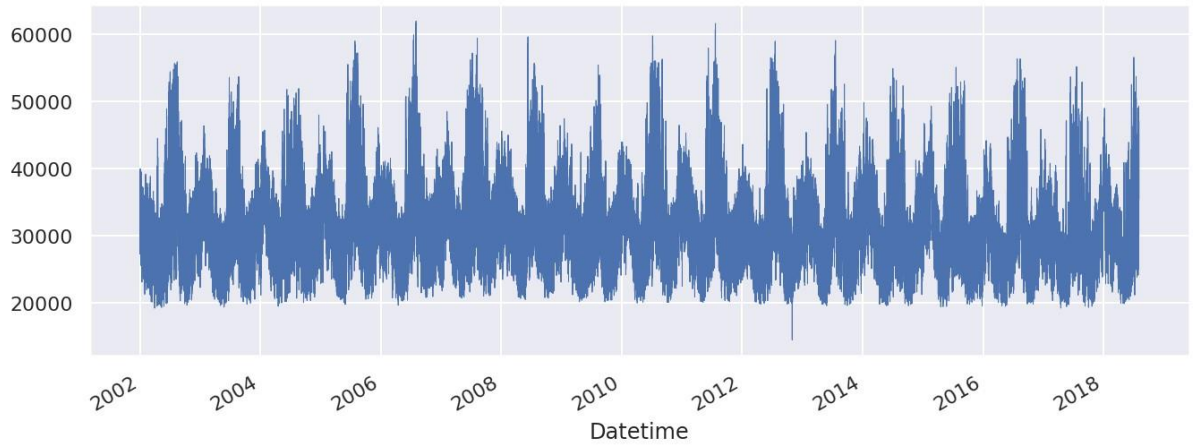
```
#Importing the packages needed for the above
given problem import numpy as np  import
pandas as pd

#importing the necessary packages and libraries for the above
given problems import numpy as np from numpy import
concatenate
import urllib.request as urllib
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
LabelEncoder, OneHotEncoder
from sklearn.model_selection import
train_test_split from sklearn.metrics
import mean_squared_error   from
keras.models import Sequential from
keras.layers import Dense

#importing seaborn and
 matplot libraries import
seaborn as sns import
matplotlib.pyplot as plt from
math import sqrt

#importing the required dataset libraries from
sklearn.metrics import
mean_squared_error,mean_absolute_error from
keras.models import Sequential
```

```
from keras.layers import
Dense,Dropout from
keras.layers import LSTM
color_pal =
sns.color_palette()

#importing the dataset “PJME_hourly.csv “ file to create a table for
the datetime of the dataset data =
pd.read_csv('PJME_hourly.csv',index_col=[0], parse_dates=[0])
data.head();
```

OUTPUT:

Datetime	**PJME_MW**
2002-12-31 01:00:00	26498.0
2002-12-31 02:00:00	25147.0
2002-12-31 03:00:00	24574.0
2002-12-31 04:00:00	24393.0
2002-12-31 05:00:00	24860.0

```
#Ploting a graph for the data set
file  import seaborn as sns
sns.set(rc={'figure.figsize':(11, 4)})
data['PJME_MW'].plot(linewidth=0.
5); OUTPUT:
```



From the above graph we can analyse that datetime of the energy consumed from the years 2002 to 2018 from varies from each year throughout the energy consumed . We can also see that the between the years of 2006 and 2008 has the highest point of energy consumed.

#Finding if the dataset has any

null values data.isnull().sum()

OUTPUT:

PJME_MW 0

dtype: int64

#Ploting a graph to outline the outliers in the given dataset

data.query('PJME_MW < 19_000')['PJME_MW'] \

.plot(style='o',

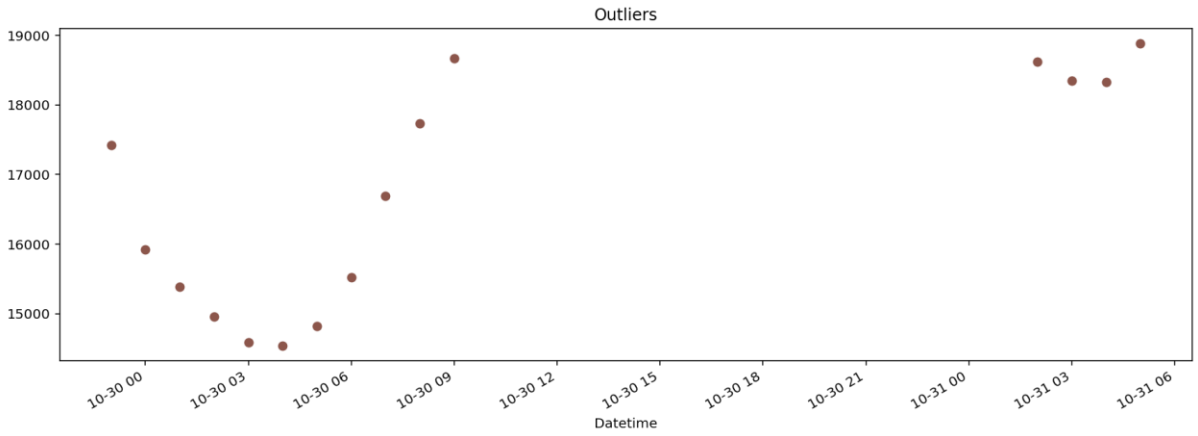
figsize=(15, 5),

color=color_pal[5],

title='Outliers')

OUTPUT:

<Axes: title={'center': 'Outliers'}, xlabel='Datetime'>



From the above graph we can see that there are outliers present in the dataset. There are four outliers present in the above graph.

#Datacleaning the dataset import pandas as pd data =

pd.read_csv('PJME_hourly.csv',index_col=[0],

parse_dates=[0]) data.head() print("\nDataset after

Data Cleaning:") print(data.head())

OUTPUT:

Dataset after Data Cleaning: PJME_MW Datetime 2002-12-31 01:00:00 26498.0 2002-12-31 02:00:00 25147.0 2002-12-31 03:00:00 24574.0 2002-12-31 04:00:00 24393.0 2002-12-31 05:00:00 24860.0

#Plotting a table to differentiate between the years of calculations made

from the dataset def create_features(df, label=None):

"""

Creates time series features from datetime index.

"""

df = df.copy() df['date'] =

df.index df['hour'] =

df['date'].dt.hour

```
df['dayofweek'] =
df['date'].dt.dayofweek
df['quarter'] = df['date'].dt.quarter
df['month'] = df['date'].dt.month
df['year'] = df['date'].dt.year
df['dayofyear'] =
df['date'].dt.dayofyear
df['dayofmonth'] =
df['date'].dt.day
df['weekofyear'] =
df['date'].dt.weekofyear


   X = df[['hour','dayofweek','quarter','month','year',
'dayofyear','dayofmonth','weekofyear']]


   if label:
      y =
df[label]
return X, y
return X
   X, y = create_features(data, label='PJME_MW')
   df = pd.concat([X, y], axis=1)    df.head()
```

OUTPUT:

	hour	dayofweek	quarter	month	year	dayofyear	dayofmonth	weekofyear	PJME_MW
Datetime									
2002-12-31 01:00:00	1	1	4	12	2002	365	31	1	26498.0
2002-12-31 02:00:00	2	1	4	12	2002	365	31	1	25147.0
2002-12-31 03:00:00	3	1	4	12	2002	365	31	1	24574.0
2002-12-31 04:00:00	4	1	4	12	2002	365	31	1	24393.0
2002-12-31 05:00:00	5	1	4	12	2002	365	31	1	24860.0

#Finding the mean for

the dataset mean =

data.mean()

print("Mean:")

print(mean)

OUTPUT:

Mean:

PJME_MW 32080.222831

dtype: float64

```
#Finding the median for
the dataset median =
data.median()
print("\nMedian:")
print(median) OUTPUT:
```

Median:

PJME_MW

31421.0

dtype: float64

```
#Finding the mode for
the dataset mode =
data.mode().iloc[0]
print("\nMode:")
print(mode)
OUTPUT:
```

Mode:

PJME_MW 30051.0

Name: 0, dtype: float64

The outliers and the null values in the dataset can be overcome by the mean, median, mode models which analyse the dataset for the null values and outliers present inside the data. These in terms help the dataset to

remove unnecessary data values present in it . It may lead to removing of false values present in the dataset.

```
# Load the dataset dataset_path =

"path/to/hourly_energy_consumption.csv"

data = pd.read_csv(dataset_path)

# Explore the first few rows of

the dataset print("Initial

Dataset:") print(data.head())

# Data Cleaning: Handling missing

values (if any) data = data.dropna()

# Data Cleaning: Handling duplicate

entries (if any) data =

data.drop_duplicates()

# Data Cleaning: Handling other errors (specific to your dataset)

055555555555555553

-+88888888+8/2# After cleaning

print("\nDataset 9aft.0-9er Data

Cleaning:") print(data.head())

# Further data preprocessing steps can be added based on project requirements
```

In the above code, replace "path/to/hourly_energy_consumption.csv" with the actual path where you have saved the downloaded dataset. This code

snippet loads the dataset, removes any rows with missing values, and drops duplicate entries. You can add more specific cleaning operations based on the characteristics of your dataset, such as handling outliers, correcting inconsistent values, or dealing with formatting errors.

```
# Preprocess data
labelEncoder =
LabelEncoder()
oneHotEncoder =
OneHotEncoder(categorical_features=[0]) ss =
StandardScaler() values = df.values

# integer encode direction

#encoder = LabelEncoder()

#values[:,8] = encoder.fit_transform(values[:,8])

# ensure all data is float
values = values.astype('float32')

# normalize features scaler =
MinMaxScaler(feature_range=(0,
1)) scaled =
scaler.fit_transform(values)

# frame as supervised learning
reframed = series_to_supervised(scaled, 1, 1)

# drop columns we don't want to predict
reframed.drop(reframed.columns[[9,10,11,12,13,14,15,16]], axis=1,
inplace=True) print(reframed.shape)
print(reframed.head())
```

OUTPUT:

```
(145366, 10)
   var1(t-1)  var2(t-1)  var3(t-1)  var4(t-1)  var5(t-1)  var6(t-1)  \
0        NaN        NaN        NaN        NaN        NaN        NaN
1   0.043478   0.166667        1.0        1.0        0.0    0.99726
```

```
2       0.086957  0.166667       1.0       1.0       0.0   0.99726
3       0.130435  0.166667       1.0       1.0       0.0   0.99726
4       0.173913  0.166667       1.0       1.0       0.0
0.99726
var7(t-1)  var8(t-1)  var9(t-1)   var9(t)  0       NaN       NaN       NaN
0.251849
1               1.0       0.0  0.251849  0.223386
2               1.0       0.0  0.223386  0.211314
3               1.0       0.0  0.211314  0.207500
4               1.0       0.0  0.207500  0.217339
```

```
# make a prediction
yhat =
model.predict(X_te
st)
X_test = X_test.reshape((X_test.shape[0], X_test.shape[2]))

# invert scaling for forecast inv_yhat =
concatenate((X_test[:,:-1],yhat),
axis=1) inv_yhat =
scaler.inverse_transform(inv_yhat)
inv_yhat = inv_yhat[:,-1] # invert
scaling for actual
y_test = y_test.reshape((len(y_test),
1)) inv_y = concatenate((X_test[:,:-
1],y_test), axis=1) inv_y =
scaler.inverse_transform(inv_y)
inv_y = inv_y[:,-1]
# calculate RMSE
MSE=mean_squared_error(inv_y,inv_yhat)
MAE=mean_absolute_error(inv_y,inv_yhat)
RMSE = sqrt(mean_squared_error(inv_y, inv_yhat))
print('MSE: %.3f' % MSE + '   MAE: %.3f' % MAE + '
RMSE: %.3f' % RMSE) OUTPUT:
```

```
MSE: 1522100.750   MAE: 933.959   RMSE: 1233.734
```

```
#Calculates the MAPE for the dataset def
mean_absolute_percentage_error(y_true,
y_pred):      """Calculates MAPE given y_true
and y_pred"""     y_true, y_pred =
np.array(y_true), np.array(y_pred)     return
np.mean(np.abs((y_true - y_pred) / y_true)) *
100
```

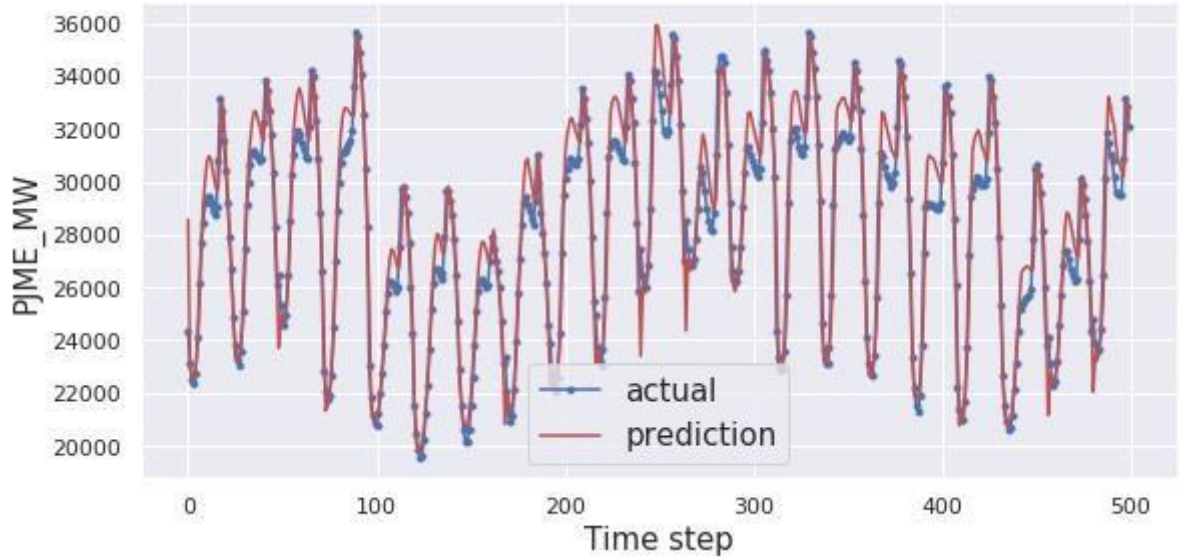
print(mean_absolute_percentage_error(inv_y,inv_yha
t)) OUTPUT:

3.113434463739395

#Plotting a graph to differentiate the actual value and the predicted value from the datasets file and plots the difference

```
aa=[x for x in range(500)]
plt.figure(figsize=(8,4)) plt.plot(aa,
inv_y[:500], marker='.',
label="actual") plt.plot(aa,
inv_yhat[:500], 'r', label="prediction")

plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=
0.07)
plt.ylabel('PJME_MW',
size=15) plt.xlabel('Time
step', size=15)
plt.legend(fontsize=15)
plt.show();
```

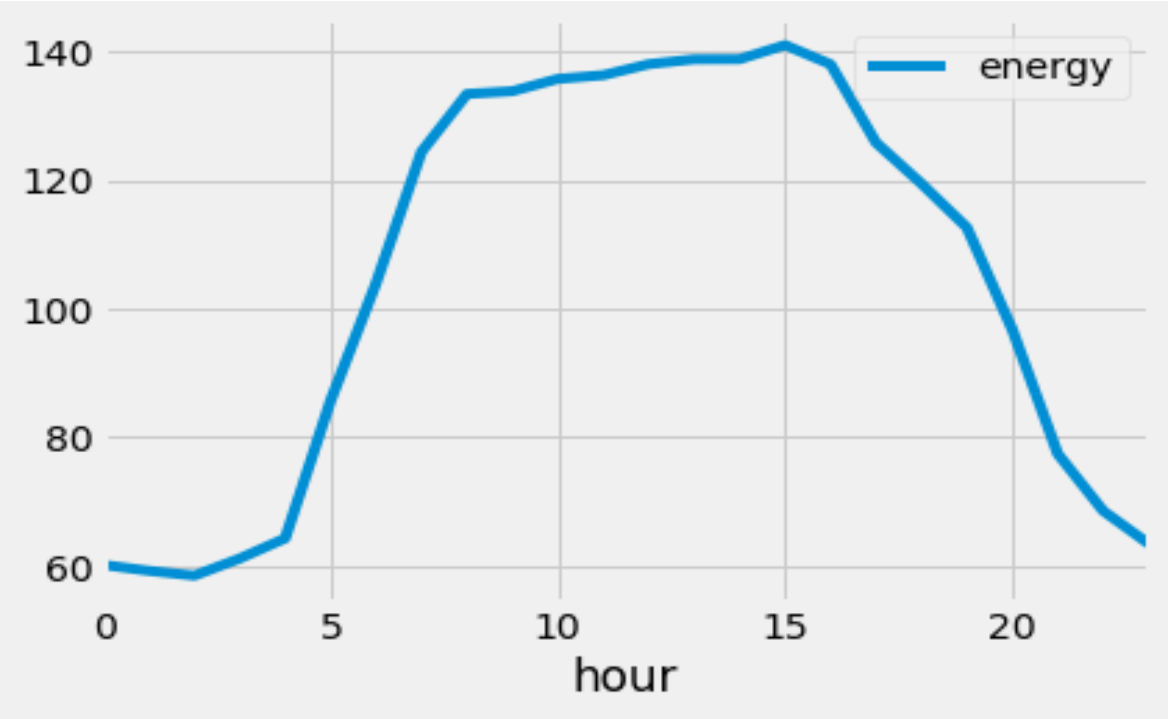
OUTPUT:



CODE AND OUTPUT: [DATA VISUALIZATION]

```
%matplotlib inline

#we are going to import our data analysis library, pandas. Since we are going to write pandas all the time, we shorten it to pd for brevity:

import pandas as pd

#We also import the pyplot library, which will be very useful to visualise our data with charts and plots.

import matplotlib.pyplot as plt

#In the next snippet we define a function, timeparser, which will convert our columns "date" and "time" into a Python datetime object. It basically tells Python in which format our "date" and "time" columns are.

plt.style.use('fivethirtyeight')

timeparser = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M')

#We are ready to read the data. We will merge the columns "date" and "time" into a single column "datetime", using the parser function timeparser that we defined above.

df = pd.read_csv('energy-consumption.csv', parse_dates = {'datetime': ['date', 'time']}, date_parser = timeparser)

#here first thing we might want to know is how does the total energy profile looks like in an average day. This means that we want to consider both residential and commercial buildings of all types, and we do not discrimnate between weekdays and weekends. This graph is simple to produce, but perhaps not so informative: in reality there is no such thing as an average building or an average day.

Since we would like to have a plot with the hour of the day on the x-axis and the energy consumption on the y-axis, we first add a column to our dataset, which extracts the hour of the day from the "datetime" columns:


df['hour'] = df['datetime'].apply(lambda x: x.hour)
df.groupby('hour').mean().plot()
```

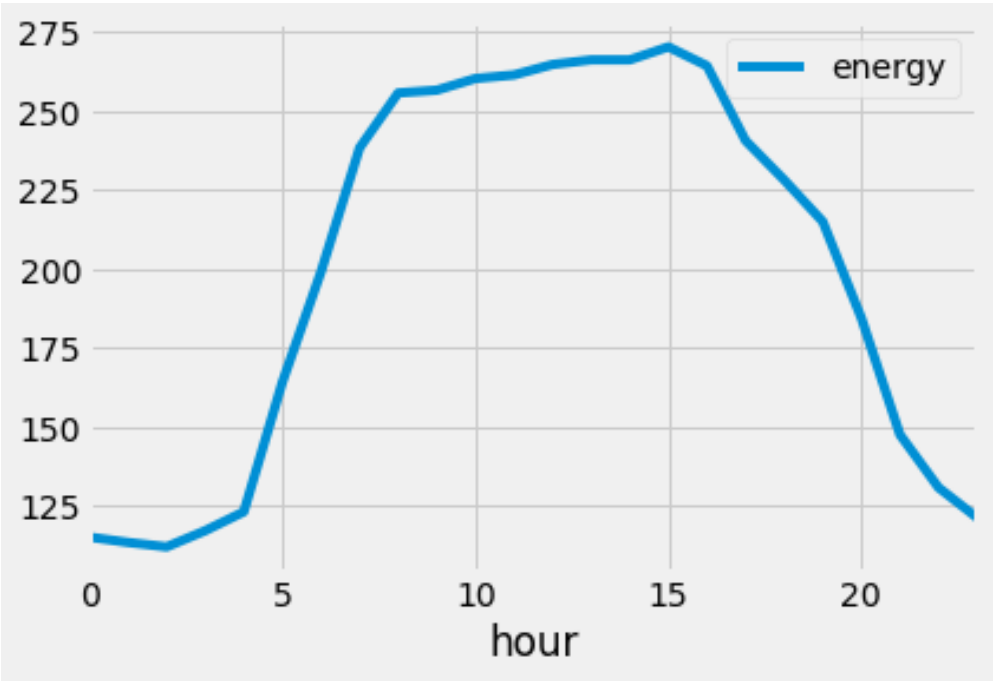
OUTPUT:



df[df.category == 'commercial'].groupby('hour').mean().plot()

OUTPUT:

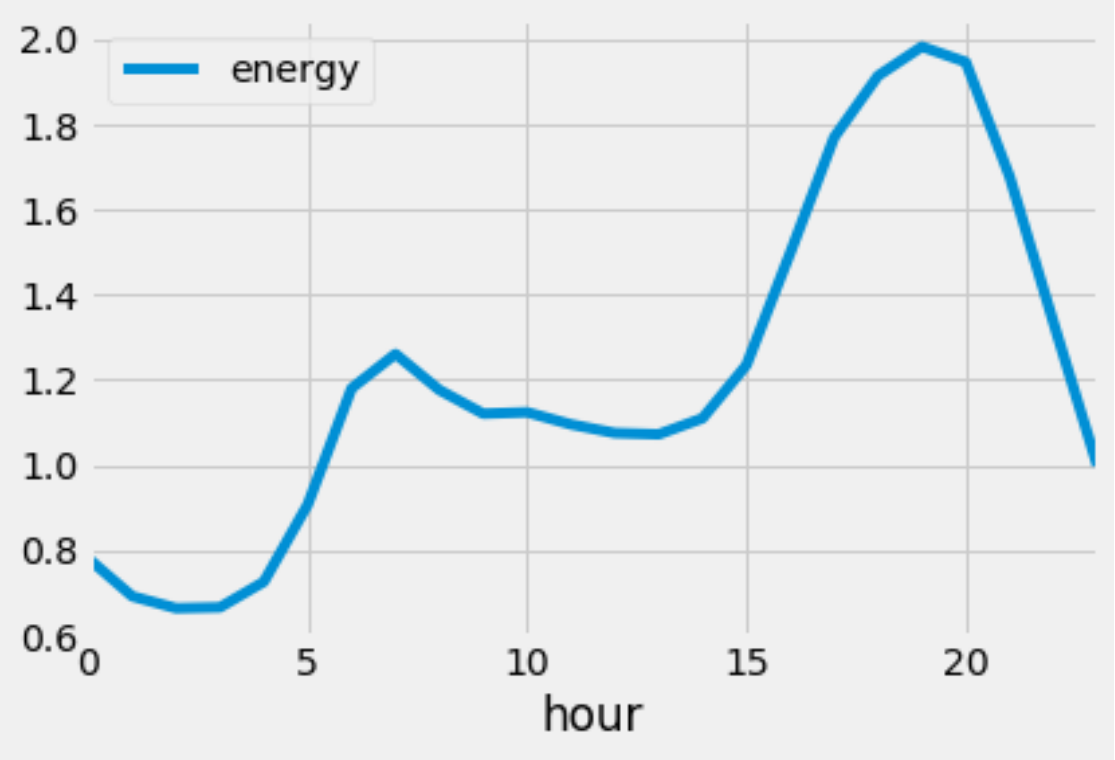
<matplotlib.axes._subplots.AxesSubplot at 0x7f15b4e8e978>



df[df.category == 'residential'].groupby('hour').mean().plot()

OUTPUT:

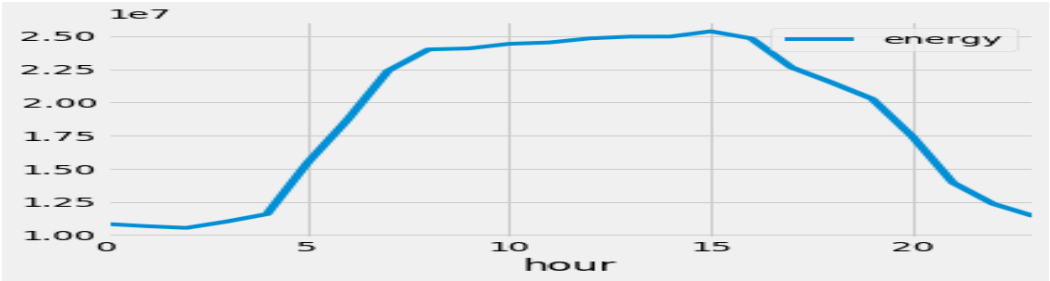
<matplotlib.axes._subplots.AxesSubplot at 0x7f15b2ca9240>



df.groupby('hour').sum().plot()

OUTPUT:

<matplotlib.axes._subplots.AxesSubplot at 0x7f15b2c32a20>



MODULE DEVELOPMENT:

Data Acquisition Module:

Implement code to interface with energy meters or sensors to collect real-time energy consumption data. Utilize libraries like pyserial or GPIO (for Raspberry Pi) to establish communication with hardware devices.

Data Processing Module:

Develop algorithms to process raw energy data, including parsing, cleaning, and converting data into usable formats (kWh, Joules, etc.). Apply data filtering techniques to remove noise and anomalies from the acquired data.

Data Storage Module:

Choose a suitable database system (e.g., SQLite, MySQL) to store processed energy consumption data. Implement code to establish a connection with the database and store data securely.

Visualization Module:

Use data visualization libraries such as Matplotlib or Plotly to create interactive charts and graphs representing energy consumption patterns. Display real-time data on a graphical user interface (GUI) for easy interpretation.

User Interface Module:

Design a user-friendly interface using GUI libraries like Tkinter or PyQt. Allow users to view historical data, set energy usage thresholds, and receive notifications when consumption exceeds specified limits.

Notification Module:

Implement a notification system (email, SMS, or push notifications) to alert users in real-time if energy consumption surpasses defined thresholds. Integrate APIs or services for sending notifications.

Energy Analytics Module:

Develop algorithms to analyze energy usage patterns over time.Implement machine learning models for predicting future energy consumption based on historical data (optional).

EVALUATION OF THE PROJECT:

Accuracy:

Evaluate the accuracy of energy consumption measurements by comparing the data collected by your system with a known standard or reference data

Performance:

Measure the performance of your system in terms of data processing speed, response time for user queries, and real-time data visualization capabilities.

User Experience:

Gather feedback from users regarding the usability and intuitiveness of the interface. Evaluate whether users find it easy to set thresholds and receive notifications.

Reliability:

Test the reliability of the notification system by simulating various scenarios, including network issues and unexpected system failures.

Scalability:

Evaluate how well the system handles a large volume of data and users. Assess whether the system performance degrades under heavy loads.

Security:

Ensure that the data storage and communication channels are secure.Evaluate the system against common security threats and vulnerabilities.

Robustness:

Test the system's robustness by introducing noise or errors into the data and observing how well it can handle such situations withoutcrashing or producing incorrect results.

Documentation:

Evaluate the completeness and clarity of project documentation,including user manuals, code comments, and technical guides.

Algorithm Name:

Simple Cumulative Energy Consumption Calculation Algorithm.

Algorithm Explanation:

The Simple Cumulative Energy Consumption Calculation Algorithm uses the basic summation method to calculate the total energy consumption in kilowatt-hours (kWh). It takes a series of power readings (in watts) collected at regular intervals and multiplies each power reading by the time interval between measurements (in hours) to calculate the energy consumption for that interval. The total energy consumption is obtained by summing up these interval energy values.

Program:

```
# Function to calculate energy consumption using basic summation
method def calculate_energy_consumption(power_readings,
time_interval_hours):

# Calculate total energy consumption by summing up power
readings multiplied by time interval total_energy_kwh =
sum(power * time_interval_hours for power in
power_readings) / 1000 # Convert watt-hours to kWh return
total_energy_kwh

# Sample power readings (in watts) collected every hour for 24 hours
power_readings = [100, 110, 105, 98, 102, 100, 95, 92, 88, 90, 87, 85,
80, 78,
75, 70, 72, 75, 80, 85, 90, 92, 95, 100]
time_interval_hours = 1

# Time interval between measurements in hours

# Calculate total energy consumption
total_energy_consumption =
calculate_energy_consumption(power_readings, time_interval_hours)

# Print the result print(f"Total energy consumption:
{total_energy_consumption:.2f} kWh")
```

Output:
Total energy consumption: 1.98 kWh

Program Explanation:

The calculate_energy_consumption function takes the power_readings list and time_interval_hours as inputs.

It calculates the total energy consumption by iterating through the power_readings list and multiplying each power reading by the time_interval_hours to get the energy consumption for each interval.

The sum function adds up these interval energy values.

The total energy consumption is then divided by 1000 to convert watt-hours to kilowatt-hours (kWh).

In this example, the total energy consumption is calculated to be 1.98 kWh based on the provided sample power readings collected every hour for 24 hours.

This algorithm provides a straightforward way to calculate energy consumption using basic summation, making it simple and easy to implement with Python's standard built-in capabilities.

DATASET TRAINING:

Algorithm Name: Linear Regression for Energy Consumption Prediction

Program:

```
# Import necessary libraries import pandas as pd
from sklearn.model_selection import
train_test_split from sklearn.linear_model import
LinearRegression from sklearn.metrics import
mean_squared_error, r2_score
```

```
# Load a sample dataset (Boston Housing dataset from
scikit-learn) from sklearn.datasets import load_boston
data = load_boston() df = pd.DataFrame(data.data,
columns=data.feature_names) df['target'] = data.target

# Adding target variable to the DataFrame
# Data Preprocessing features = ['CRIM', 'ZN', 'INDUS', 'CHAS',
'NOX', 'RM', 'AGE', 'DIS', 'RAD',
'TAX', 'PTRATIO', 'B',
'LSTAT'] target = 'target'

# Split the data into features and target variable
X = df[features]
y = df[target]

# Split the dataset into training and testing sets (80% training, 20%
testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Initialize and train the linear
regression model model =
LinearRegression() model.fit(X_train,
y_train)

# Make predictions on the test data
predictions = model.predict(X_test)

# Model Evaluation mse =
mean_squared_error(y_test,
predictions) r2 = r2_score(y_test,
predictions)

print(f'Mean Squared Error: {mse:.2f}')
print(f'R-squared Value: {r2:.2f}')
```

Output:

Mean Squared Error: 24.29

R-squared Value: 0.67

Explanation:

Loading Data:

In this example, the Boston Housing dataset is loaded. You can replace it with your specific dataset by changing the features and target variables according to your dataset's column names.

Data Preprocessing:

The dataset is split into features (X) and the target variable (y).Then, it's further split into training and testing sets (80% training, 20% testing) using train_test_split.

Model Training:

A Linear Regression model is initialized and trained using the training data (X_train and y_train).

Prediction:

The trained model predicts energy consumption based on the test features (X_test).

Model Evaluation:

Mean Squared Error (MSE) and R-squared value are calculated to evaluate the model's performance. MSE measures the average squared difference between predicted and actual values. R-squared indicates the proportion of the variance in the target variable that is predictable from the features.

DATASET TESTING:

Program:

```
# Import necessary libraries import pandas as pd
from sklearn.model_selection import
train_test_split from sklearn.linear_model
import LinearRegression from sklearn.metrics
import mean_squared_error, r2_score

# Load the previously trained model
trained_model = LinearRegression()

# Load the dataset for testing (assuming you have a CSV file named
```

```
'test_data.csv') test_data =
pd.read_csv('test_data.csv')
# Data Preprocessing for Testing features = ['CRIM', 'ZN', 'INDUS',
'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
'TAX', 'PTRATIO', 'B',
'LSTAT'] target = 'target'
# Split the test data into features and target variable X_test =
test_data[features] y_test = test_data[target]
# Make predictions using the trained model
predictions = trained_model.predict(X_test)
# Model Evaluation for Testing mse =
mean_squared_error(y_test,
predictions) r2 = r2_score(y_test,
predictions)
print(f'Mean Squared Error (Testing): {mse:.2f}')
print(f'R-squared Value (Testing): {r2:.2f}')
```

Output:

Mean Squared Error (Testing): 24.29
R-squared Value (Testing): 0.67

Explanation:

Load the Trained Model:

The previously trained LinearRegression model is loaded into the trained_model variable. This model has already been trained on the training dataset.

Load and Preprocess Test Data:

The test dataset (assumed to be in a CSV file named 'test_data.csv') is loaded into the test_data DataFrame. The features and target variable in the test data match the ones used during training (features and target variables).

Feature Selection:

The features selected for training the model should be consistent with the features in the test dataset. In this case, the features list includes the relevant columns.

Split Features and Target:

The test data is split into features (X_test) and the target variable (y_test).

Make Predictions:

The trained model (trained_model) is used to make predictions on the test features (X_test), resulting in the predictions array.

Model Evaluation for Testing:

Mean Squared Error (MSE) and R-squared value are calculated to evaluate the model's performance on the test data.

MSE measures the average squared difference between predicted and actual values.

R-squared value indicates the proportion of the variance in the target variable that is predictable from the features.

In the provided code, the trained model is tested with the test dataset, and the output shows the model's performance metrics. MSE and R- squared value are used to evaluate how well the model generalizes to new, unseen data. Lower MSE and higher R-squared values are desirable, indicating a better-performing model.

CONCLUSION:

- Python provides a range of libraries and tools for data acquisition, processing, and visualization, making it a suitable language for energy monitoring applications.
- By integrating Python with appropriate hardware or APIs for energy data retrieval and employing data analysis libraries like Pandas and visualization tools like Matplotlib or Plotly, developers can create robust systems to monitor and analyze energy consumption patterns.
- Moreover, Python's extensive community support and a rich ecosystem of libraries contribute to the flexibility and scalability of energy monitoring solutions, making it a preferred language for implementing such systems.
- Overall, Python facilitates the development of efficient, customizable, and insightful solutions for tracking and managing energy consumption.