Machine Learning Portfolio Project

Introduction

This notebook presents a machine learning project that predicts Global Active Power using household_power_consumption Dataset. The objective is to demonstrate the process from data loading and preprocessing to model training and evaluation.

Setup and Configuration

Import all necessary libraries .

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Import future statements to ensure compatibility across different Python versions
from __future__ import unicode_literals, print_function, division, absolute_import

# Import the required libraries
import matplotlib as mpl  # Matplotlib for plotting
import matplotlib.pyplot as plt  # Pyplot for easy plotting
import numpy as np  # Numpy for numerical operations
import seaborn as sns  # Seaborn for enhanced data visualization
import pandas as pd  # Pandas for data manipulation and analysis
import os  # OS module for operating system dependent functionality
from datetime import datetime  # Datetime module for handling dates and times

# Configure Matplotlib to set the default figure size and disable grid on axes
mpl.rcParams['figure.figsize'] = (10, 8)  # Set default figure size to 10 inches by 8 inches
mpl.rcParams['axes.grid'] = False  # Disable grid on axes by default
```

Loading the Dataset:

Load your dataset and display the first few rows.

```
file_path = '/content/drive/My Drive/household_power_consumption.txt'
df = pd.read_csv(file_path, sep=';')
df.head()
```

```
<ipython-input-3-b0376f579655>:2: DtypeWarning: Columns (2,3,4,5,6,7) have mixed type
  df = pd.read_csv(file_path, sep=';')
```

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_i
0	16/12/2006	17:24:00	4.216	0.418	234.840	
1	16/12/2006	17:25:00	5.360	0.436	233.630	
2	16/12/2006	17:26:00	5.374	0.498	233.290	
3	16/12/2006	17:27:00	5.388	0.502	233.740	
4	16/12/2006	17:28:00	3.666	0.528	235.680	

Data Preprocessing

Description:

This section should detail the steps taken to prepare the data for analysis. It typically includes handling missing values, data type conversions, filtering columns, and any calculations for new features.

Preprocess the data to make it suitable for analysis. This includes dealing with missing values, normalizing the data, encoding categorical variables, and other necessary steps.

```
# Combine 'Date' and 'Time' columns into a single 'Datetime' column
# Convert the combined string to a datetime object, specifying that the day comes first in the date format
df['Datetime'] = pd.to_datetime(df['Date'] + ' ' + df['Time'], dayfirst=True)

# Display the first few rows of the DataFrame to verify the new 'Datetime' column
df.head()
```

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_i
0	16/12/2006	17:24:00	4.216	0.418	234.840	
1	16/12/2006	17:25:00	5.360	0.436	233.630	
2	16/12/2006	17:26:00	5.374	0.498	233.290	

```
# Make a copy of the original DataFrame for safety, preserving the original data
datafram = df.copy()

# Drop the 'Date' and 'Time' columns from the DataFrame as they are now redundant
# This is because we have already combined them into the 'Datetime' column
df = df.drop(columns=['Date', 'Time'])


# Convert the 'Global_active_power' column to numeric type, coercing errors to NaN
df['Global_active_power'] = pd.to_numeric(df['Global_active_power'], errors='coerce')

# Convert the 'Global_reactive_power' column to numeric type, coercing errors to NaN
df['Global_reactive_power'] = pd.to_numeric(df['Global_reactive_power'], errors='coerce')

# Convert the 'Voltage' column to numeric type, coercing errors to NaN
df['Voltage'] = pd.to_numeric(df['Voltage'], errors='coerce')

# Convert the 'Global_intensity' column to numeric type, coercing errors to NaN
df['Global_intensity'] = pd.to_numeric(df['Global_intensity'], errors='coerce')

# Convert the 'Sub_metering_1' column to numeric type, coercing errors to NaN
df['Sub_metering_1'] = pd.to_numeric(df['Sub_metering_1'], errors='coerce')

# Convert the 'Sub_metering_2' column to numeric type, coercing errors to NaN
df['Sub_metering_2'] = pd.to_numeric(df['Sub_metering_2'], errors='coerce')



# Set the 'Datetime' column as the index of the DataFrame
df = df.set_index('Datetime')

# Display information about the DataFrame's index, columns, and data types
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2075259 entries, 2006-12-16 17:24:00 to 2010-11-26 21:02:00
Data columns (total 7 columns):
 #   Column                 Dtype
---  ------                 -----
 0   Global_active_power    float64
 1   Global_reactive_power  float64
 2   Voltage                float64
 3   Global_intensity       float64
 4   Sub_metering_1         float64
 5   Sub_metering_2         float64
 6   Sub_metering_3         float64
dtypes: float64(7)
memory usage: 126.7 MB
```

```
# Check for null values in DataFrame 'df' and return a DataFrame of boolean values where 'True' indicates a null value
null_check = df.isnull()

# Sum up the null values for each column and return a Series with the sums
null_counts = null_check.sum()

# Output the total count of null values for each column
print(null_counts)
```

```
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
Sub_metering_3           25979
dtype: int64
```

```
# Compute the rolling mean with a window size of 3000 and at least 1 non-null value
#df = df.rolling(window=3000, min_periods=1).mean()
numeric_columns = [col for col in df.columns if df[col].dtype != 'object']
df[numeric_columns] = df[numeric_columns].rolling(window=3000, min_periods=1).mean()


# Count the number of missing (NaN) values in each column after applying the rolling mean
# This helps to identify any missing values introduced by the rolling mean calculation
df.isnull().sum()
```

```
Global_active_power      7625
Global_reactive_power    7625
Voltage                  7625
Global_intensity         7625
Sub_metering_1           7625
Sub_metering_2           7625
Sub_metering_3           7625
dtype: int64
```

```
# This is formatted as code
```

** Exploratory Data Analysis (EDA)**

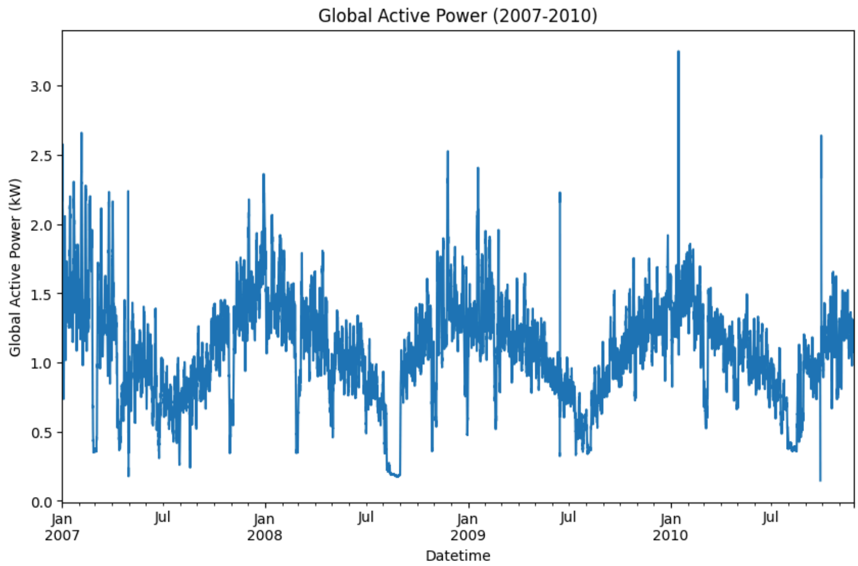
Description:

Explain the purpose of EDA in your project, which could include identifying patterns, anomalies, or relationships in the data. Use visualizations to support the analysis.

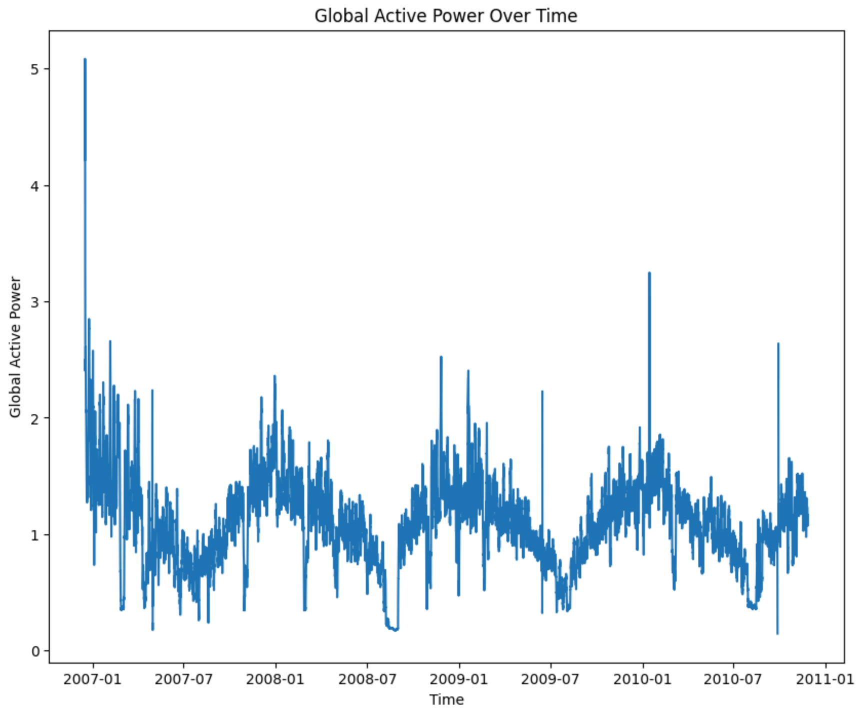
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2075259 entries, 2006-12-16 17:24:00 to 2010-11-26 21:02:00
Data columns (total 7 columns):
 #   Column                 Dtype  
---  ------                 -----  
 0   Global_active_power    float64
 1   Global_reactive_power  float64
 2   Voltage                float64
 3   Global_intensity       float64
 4   Sub_metering_1         float64
 5   Sub_metering_2         float64
 6   Sub_metering_3         float64
dtypes: float64(7)
memory usage: 126.7 MB
```

```
df_2007 = df.loc['2007-01-01':'2010-12-30']

# Plot using matplotlib
p_2007 = df_2007['Global_active_power']
plt.figure(figsize=(10, 6))
p_2007.plot()
plt.title('Global Active Power (2007-2010)')
plt.xlabel('Datetime')
plt.ylabel('Global Active Power (kW)')
plt.show()
```



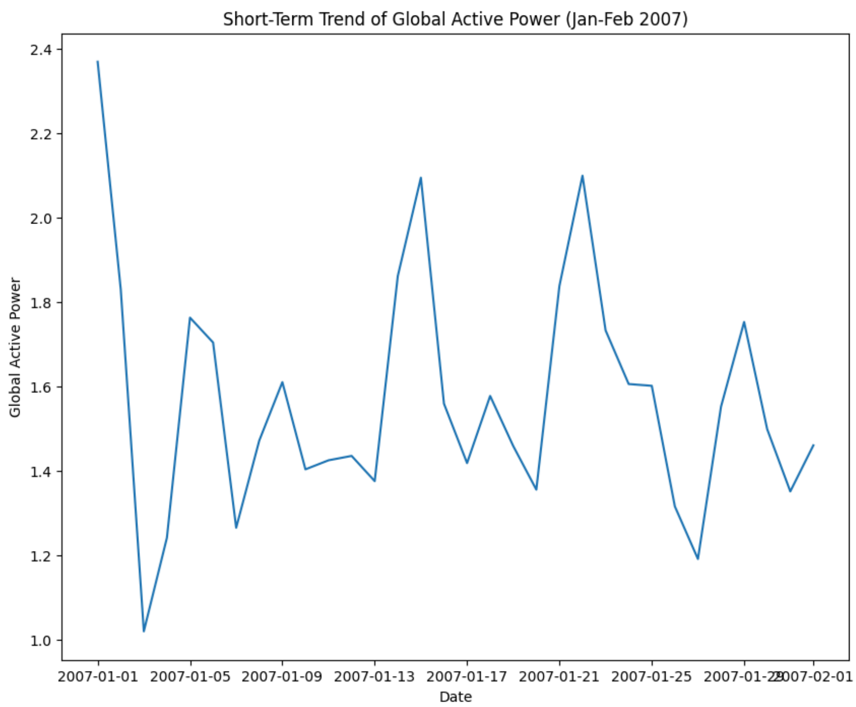

```
import matplotlib.pyplot as plt
def visualize_resampled_data(df):
  # Resample data
  resampled_data_h = df['Global_active_power'].resample('H').mean()
  resampled_data_d = df['Global_active_power'].resample('D').mean()
  resampled_data_m = df['Global_active_power'].resample('M').mean()
  # Plot Resampled Data set
  plt.figure(figsize=(10, 6))
  plt.plot(resampled_data_h)
  plt.plot(resampled_data_d)
  plt.plot(resampled_data_m)
  plt.title('Resampled Global Active Power (Hourly Mean)')
  plt.xlabel('Datetime')
  plt.ylabel('Global Active Power (kW)')
  plt.show()
  visualize_resampled_data(df)


# Plotting the 'Global_active_power' column from the DataFrame 'df'
plt.plot(df['Global_active_power'])

# Adding title to the plot
plt.title('Global Active Power Over Time')

# Adding labels to the axes
plt.xlabel('Time')
plt.ylabel('Global Active Power')

# Displaying the plot
plt.show()
```




```
# Resampling the DataFrame 'df' to daily frequency ('1d') and calculating the mean for each day
df = df.resample('1d').mean()

# Displaying the first 100 rows of the resampled DataFrame to examine the changes
df.head(100)

# Explanation:
# - Resampling data to a lower frequency (daily, in this case) helps in reducing the number of data points
#   and aggregating the data for easier analysis.
# - By resampling to daily frequency and calculating the mean, we get average values for each day,
#   which can provide a clearer overview of trends and patterns in the data.
# - Displaying the first 100 rows helps in verifying the changes made by resampling and ensuring data integrity.
```

Datetime	Global_active_power	Global_reactive_power	Voltage	Global_intensity
2006-12-16	3.648148	0.138562	234.728176	15.627769
2006-12-17	2.574466	0.107634	239.167078	10.999294
2006-12-18	2.130852	0.134341	240.202567	9.050809
2006-12-19	1.659651	0.128231	240.898049	7.014547
2006-12-20	1.329093	0.106785	241.846311	5.609023
...	...	...	...	...
2007-03-21	1.485748	0.108218	240.811469	6.266218
2007-03-22	1.430311	0.103795	241.107669	6.021466

```
# Plotting the 'Global_active_power' data for the specified date range ('2007-01-01' to '2007-02-01')
plt.plot(df['Global_active_power']['2007-01-01':'2007-02-01'])

# Adding title to the plot
plt.title('Short-Term Trend of Global Active Power (Jan-Feb 2007)')

# Adding labels to the axes
plt.xlabel('Date')
plt.ylabel('Global Active Power')

# Displaying the plot
plt.show()

# Explanation:
# - Selecting a subset of the 'Global_active_power' data for the specified date range ('2007-01-01' to '2007-02-01').
# - Plotting this subset helps in visualizing the short-term trend of global active power consumption during January and February 2007.
# - Adding a title and labels to the plot enhances clarity and understanding.
# - Displaying the plot allows for visual inspection of the short-term trend in the data.
```



Variance:

Variance tells us how much the numbers in a dataset spread out from their average. If the variance is low, it means the numbers are close to the average. If the variance is high, it means the numbers are more spread out.

Example:

Let's say we have a dataset of test scores: [70, 75, 80, 85, 90].

First, we find the average (mean) score: (70 + 75 + 80 + 85 + 90) / 5 = 80. Then, we find the difference between each score and the average: [-10, -5, 0, 5, 10]. We square each difference to get rid of negative values: [100, 25, 0, 25, 100]. Finally, we find the average of these squared differences, which gives us the variance. In this case, the variance is 50.

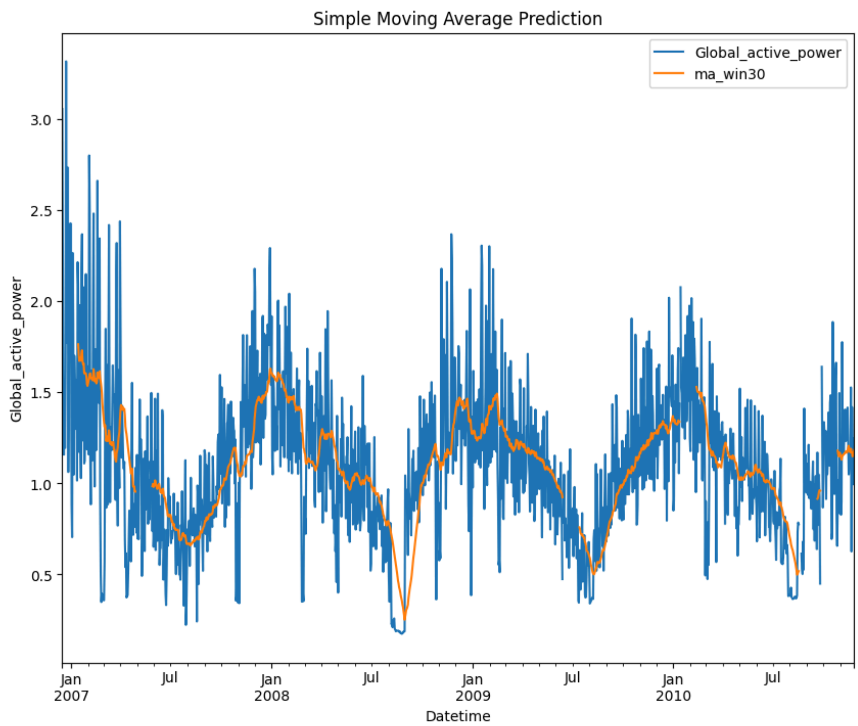
Overall, plotting both the original data and its variance side by side allows for a comprehensive exploration of the dataset, providing insights into both the overall behavior and the variability of the data over time.

```
data_gap = datafram.set_index('Datetime')
data_gap.info()
```

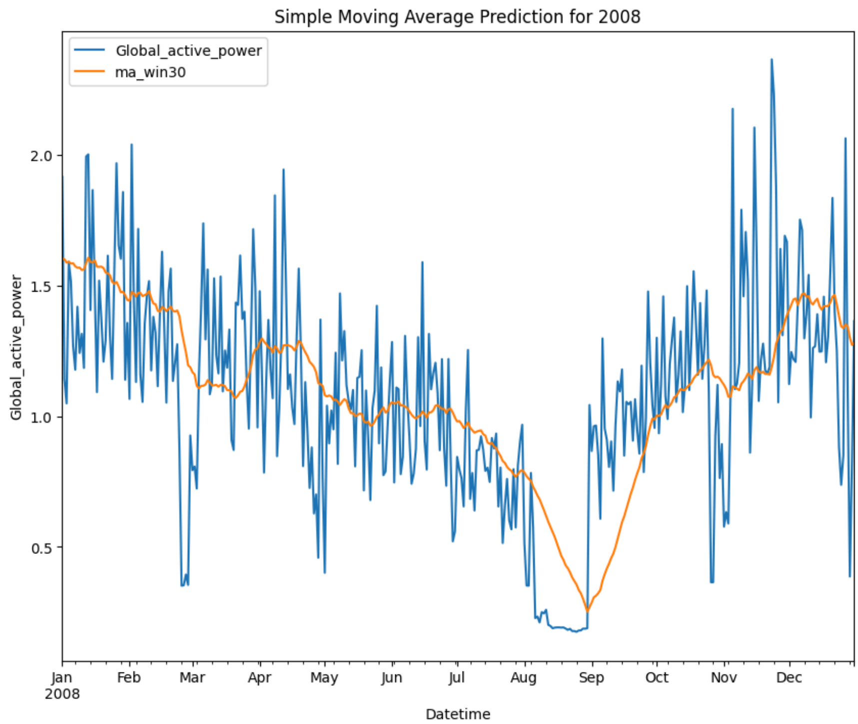
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2075259 entries, 2006-12-16 17:24:00 to 2010-11-26 21:02:00
Data columns (total 9 columns):
 #   Column                 Dtype
---  ------                 -----
 0   Date                   object
 1   Time                   object
 2   Global_active_power    object
 3   Global_reactive_power  object
 4   Voltage                object
 5   Global_intensity       object
 6   Sub_metering_1         object
 7   Sub_metering_2         object
 8   Sub_metering_3         float64
dtypes: float64(1), object(8)
memory usage: 158.3+ MB
```

```
# prompt: drop all column from data_gap except index and global active power, then convert global active power to float

data_gap = data_gap.drop(columns=[col for col in data_gap.columns if col != 'Global_active_power'])
data_gap['Global_active_power'] = data_gap['Global_active_power'].replace('?', np.nan)
data_gap['Global_active_power'] = data_gap['Global_active_power'].astype(float)
```

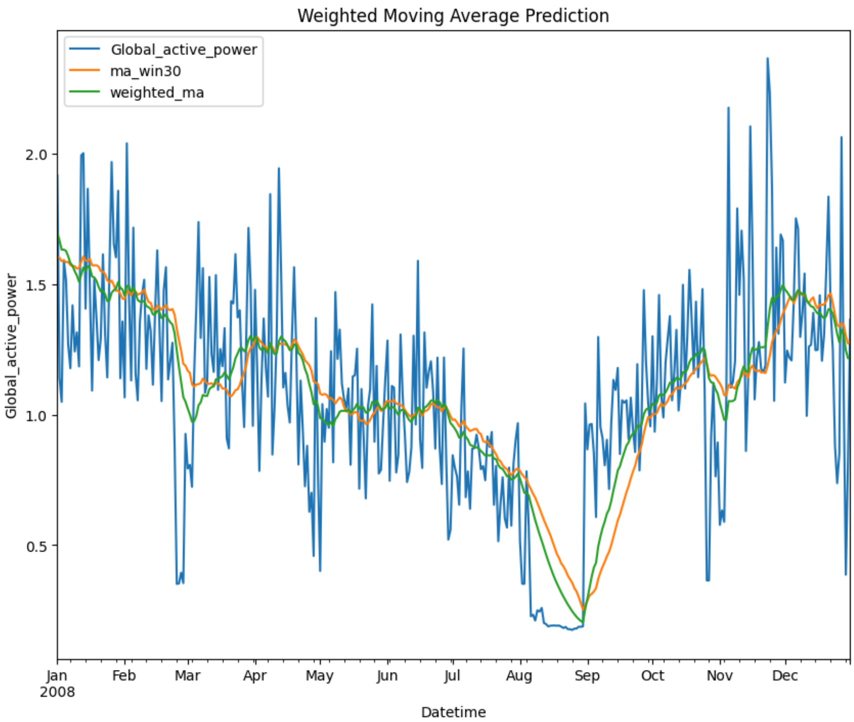
```
data_gap = data_gap.resample('1d').mean()
data_gap['ma_win30'] = data_gap['Global_active_power'].rolling(window=30).mean()
data_gap.plot()
plt.ylabel('Global_active_power')
plt.title('Simple Moving Average Prediction')
plt.show()
```



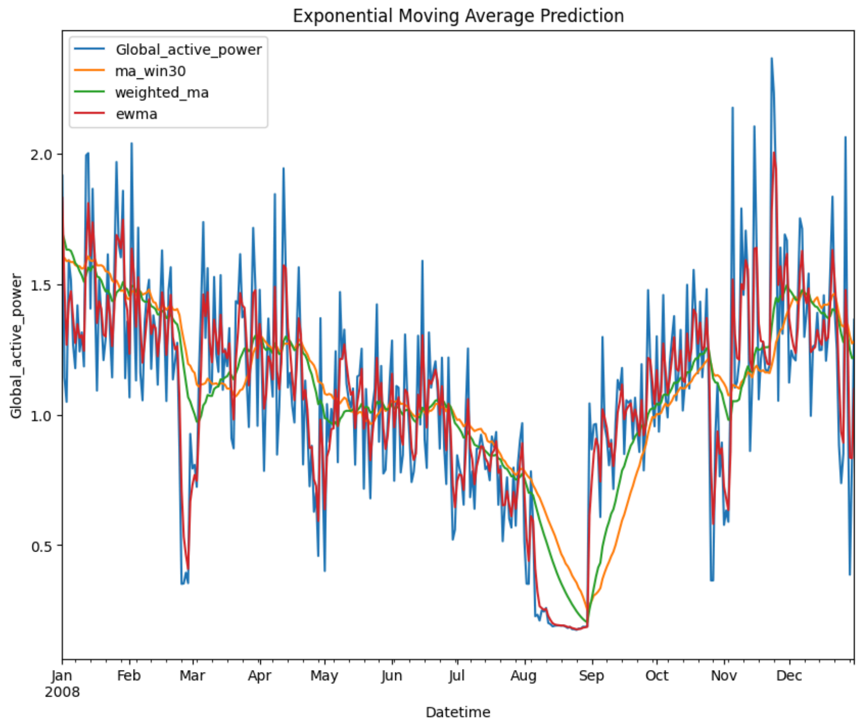
```
# prompt: only show values of one year
data_gap['2008-01-01':'2008-12-31'].plot()
plt.ylabel('Global_active_power')
plt.title('Simple Moving Average Prediction for 2008')
plt.show()
```



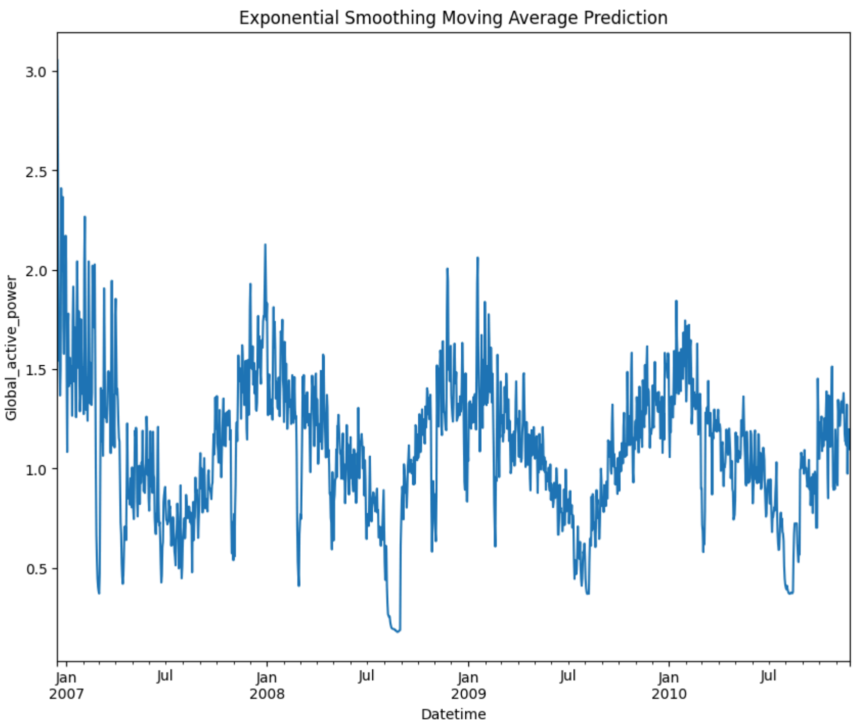

```
# prompt: apply weighted moving average
data_gap['weighted_ma'] = data_gap['Global_active_power'].rolling(window=30).apply(lambda x: np.average(x, weights=np.arange(1, len(x)+
data_gap['2008-01-01':'2008-12-31'].plot()
plt.ylabel('Global_active_power')
plt.title('Weighted Moving Average Prediction')
plt.show()
```



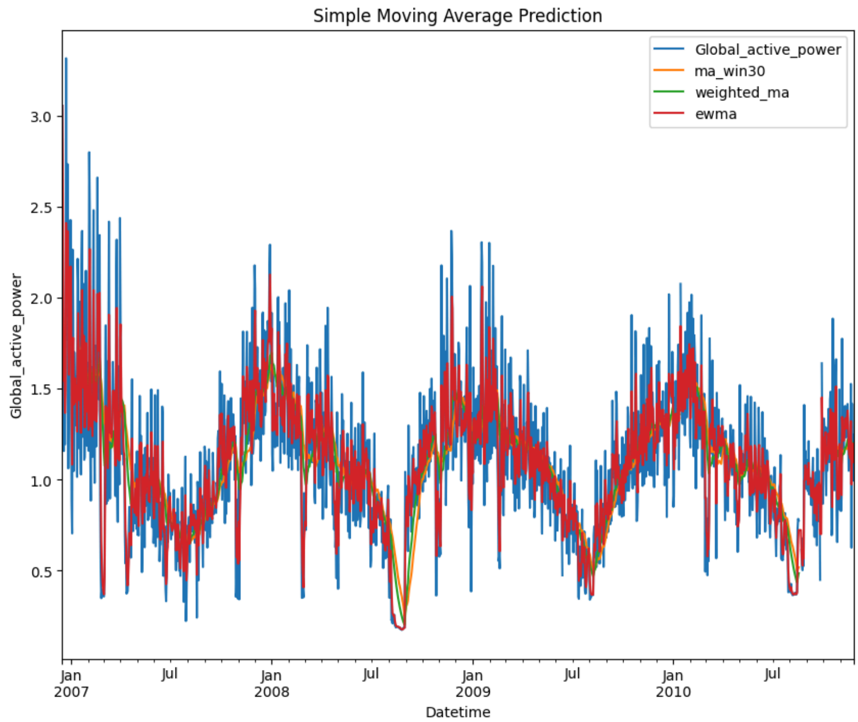

```
# prompt: apply exponential weighted moving average

import matplotlib.pyplot as plt
data_gap['ewma'] = data_gap['Global_active_power'].ewm(alpha=0.5).mean()
data_gap['2008-01-01':'2008-12-31'].plot()
plt.ylabel('Global_active_power')
plt.title('Exponential Moving Average Prediction')
plt.show()
```



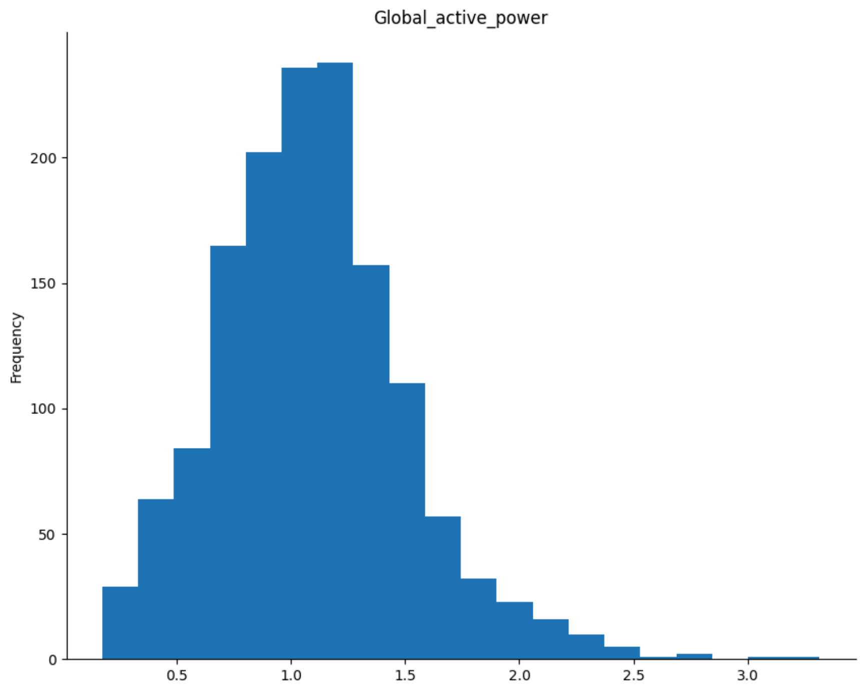

```
# prompt: apply exponential smoothing moving average

import matplotlib.pyplot as plt
data_gap['Global_active_power'].ewm(alpha=0.5).mean().plot()
plt.ylabel('Global_active_power')
plt.title('Exponential Smoothing Moving Average Prediction')
plt.show()
```



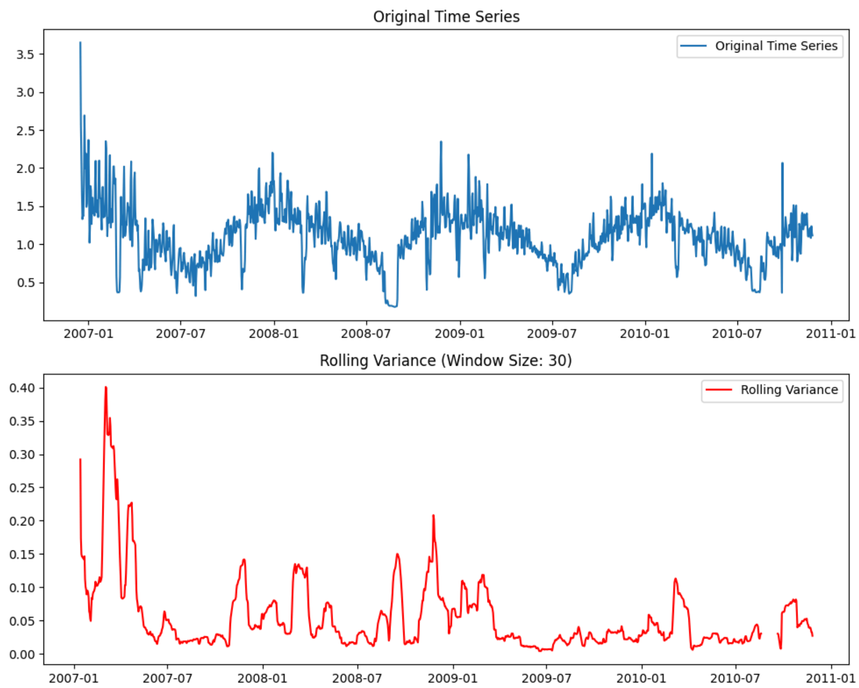

```
data_gap = data_gap.resample('1d').mean()
data_gap['ma_win30'] = data_gap['Global_active_power'].rolling(window=30).mean()
data_gap.plot()
plt.ylabel('Global_active_power')
plt.title('Simple Moving Average Prediction')
plt.show()
```




```
from matplotlib import pyplot as plt
data_gap['Global_active_power'].plot(kind='hist', bins=20, title='Global_active_power')
plt.gca().spines[['top', 'right',]].set_visible(False)
```

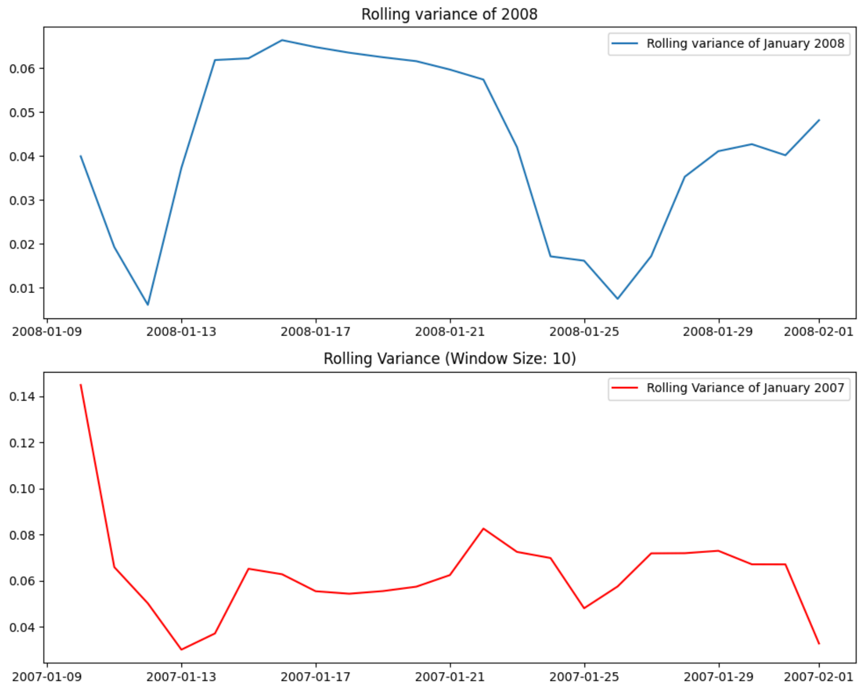


```
# Define the size of the rolling window for calculating variance
window_size = 30

# Calculate rolling variance of the 'Global_active_power' data using the specified window size
data = df['Global_active_power'].rolling(window=window_size).var()

# Create subplots for original time series and rolling variance
plt.subplot(2, 1, 1)  # Creating a subplot grid of 2 rows and 1 column, and selecting the first subplot
plt.plot(df['Global_active_power'], label='Original Time Series')  # Plotting the original time series
plt.title('Original Time Series')  # Adding title to the subplot
plt.legend()  # Adding legend to display label

# Plot rolling variance
plt.subplot(2, 1, 2)  # Selecting the second subplot
plt.plot(data, label='Rolling Variance', color='red')  # Plotting the rolling variance
plt.title(f'Rolling Variance (Window Size: {window_size})')  # Adding title to the subplot
plt.legend()  # Adding legend to display label

# Adjust layout to prevent overlapping of subplots
plt.tight_layout()

# Explanation:
# - The code calculates the rolling variance of the 'Global_active_power' data using a specified window size.
# - It then creates subplots to visualize both the original time series and the rolling variance.
# - The first subplot displays the original time series data.
# - The second subplot displays the rolling variance of the data with a specified window size.
# - Titles and legends are added to provide context and distinguish between the plots.
# - The 'plt.tight_layout()' function adjusts the layout to prevent overlapping of subplots and improve readability.
```

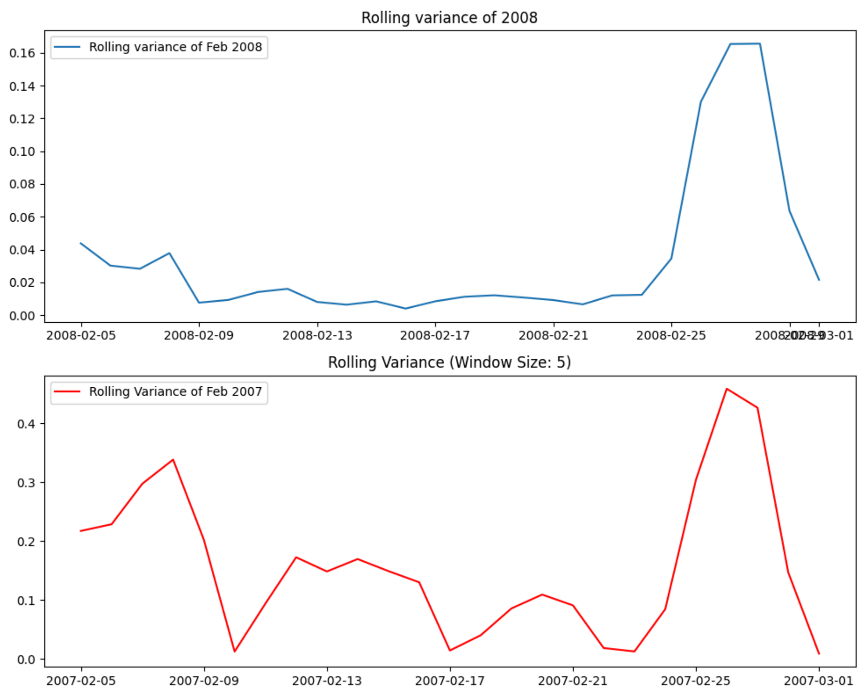


Short Term Analysis of Data

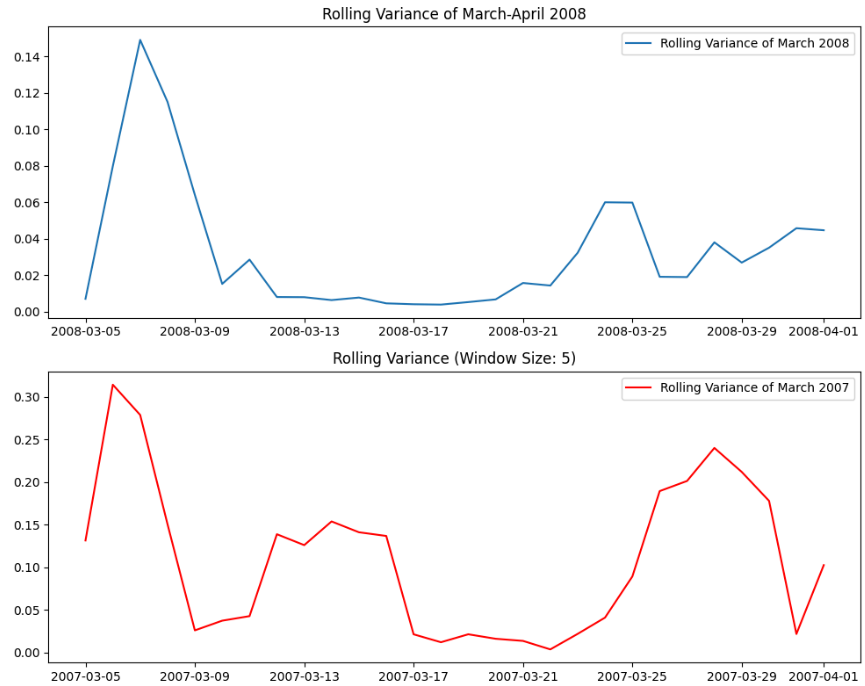
```
# Set the window size for rolling variance calculation
window_size = 10

# Calculate rolling variance for January 2007 data
data_jan = df['Global_active_power']['2007-01-01':'2007-2-1'].rolling(window=window_size).var()

# Calculate rolling variance for January 2008 data
data_ja = df['Global_active_power']['2008-01-01':'2008-2-1'].rolling(window=window_size).var()

# Create a subplot for the first graph (January 2008)
plt.subplot(2, 1, 1)
plt.plot(data_ja, label='Rolling variance of January 2008')  # Plot the rolling variance data
plt.title('Rolling variance of 2008')  # Set the title of the subplot
plt.legend()  # Display the legend

# Create a subplot for the second graph (January 2007)
plt.subplot(2, 1, 2)
plt.plot(data_jan, label='Rolling Variance of January 2007', color='red')  # Plot the rolling variance data
plt.title(f'Rolling Variance (Window Size: {window_size})')  # Set the title of the subplot
plt.legend()  # Display the legend

# Adjust the layout to prevent overlapping of subplots
plt.tight_layout()
```



```
# Set the window size for rolling variance calculation
window_size = 5

# Calculate rolling variance for February 2007 data
data_jan = df['Global_active_power']['2007-02-01':'2007-3-1'].rolling(window=window_size).var()

# Calculate rolling variance for February 2008 data
data_ja = df['Global_active_power']['2008-02-01':'2008-3-1'].rolling(window=window_size).var()

# Create a subplot for the first graph (February 2008)
plt.subplot(2, 1, 1)
plt.plot(data_ja, label='Rolling variance of Feb 2008')  # Plot the rolling variance data
plt.title('Rolling variance of 2008')  # Set the title of the subplot
plt.legend()  # Display the legend

# Create a subplot for the second graph (February 2007)
plt.subplot(2, 1, 2)
plt.plot(data_jan, label='Rolling Variance of Feb 2007', color='red')  # Plot the rolling variance data
plt.title(f'Rolling Variance (Window Size: {window_size})')  # Set the title of the subplot
plt.legend()  # Display the legend

# Adjust the layout to prevent overlapping of subplots
plt.tight_layout()
```

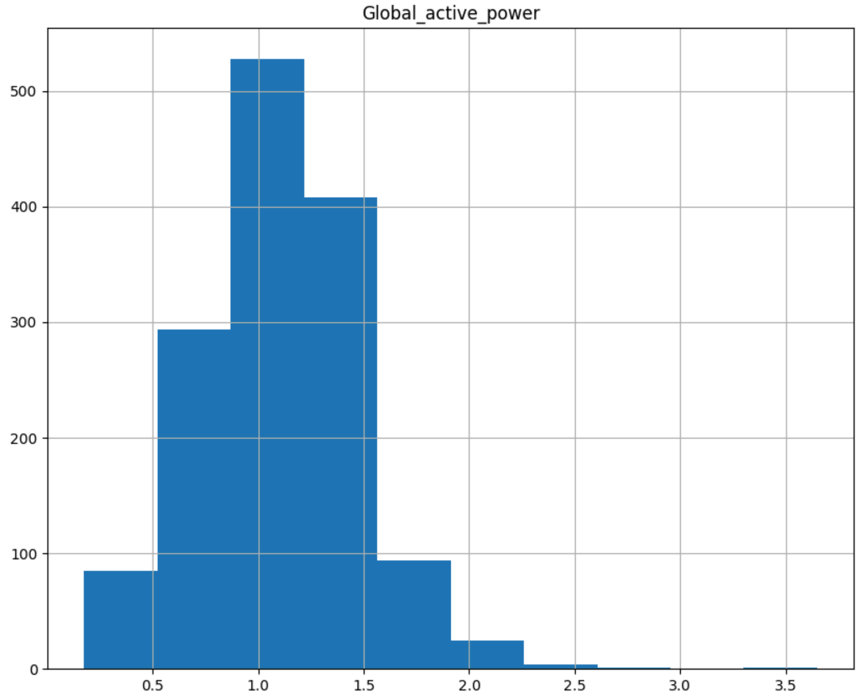


```
# Define the window size for calculating rolling variance
window_size = 5

# Calculate rolling variance for March-April 2007 data
data_jan = df['Global_active_power']['2007-03-01':'2007-04-01'].rolling(window=window_size).var()

# Calculate rolling variance for March-April 2008 data
data_ja = df['Global_active_power']['2008-03-01':'2008-04-01'].rolling(window=window_size).var()

# Create subplots for rolling variance of March-April 2008 data
plt.subplot(2, 1, 1)  # Selecting the first subplot
plt.plot(data_ja, label='Rolling Variance of March 2008')  # Plotting rolling variance for March-April 2008
plt.title('Rolling Variance of March-April 2008')  # Adding title to the subplot
plt.legend()  # Adding legend to display label

# Create subplots for rolling variance of March-April 2007 data
plt.subplot(2, 1, 2)  # Selecting the second subplot
plt.plot(data_jan, label='Rolling Variance of March 2007', color='red')  # Plotting rolling variance for March-April 2007
plt.title(f'Rolling Variance (Window Size: {window_size})')  # Adding title to the subplot
plt.legend()  # Adding legend to display label

# Adjust layout to prevent overlapping of subplots
plt.tight_layout()

# Explanation:
# - The code calculates the rolling variance for March-April 2007 and March-April 2008 data separately.
# - It creates subplots to visualize the rolling variance of each time period.
# - The first subplot displays the rolling variance for March-April 2008 data.
# - The second subplot displays the rolling variance for March-April 2007 data.
# - Titles and legends are added to provide context and distinguish between the plots.
```

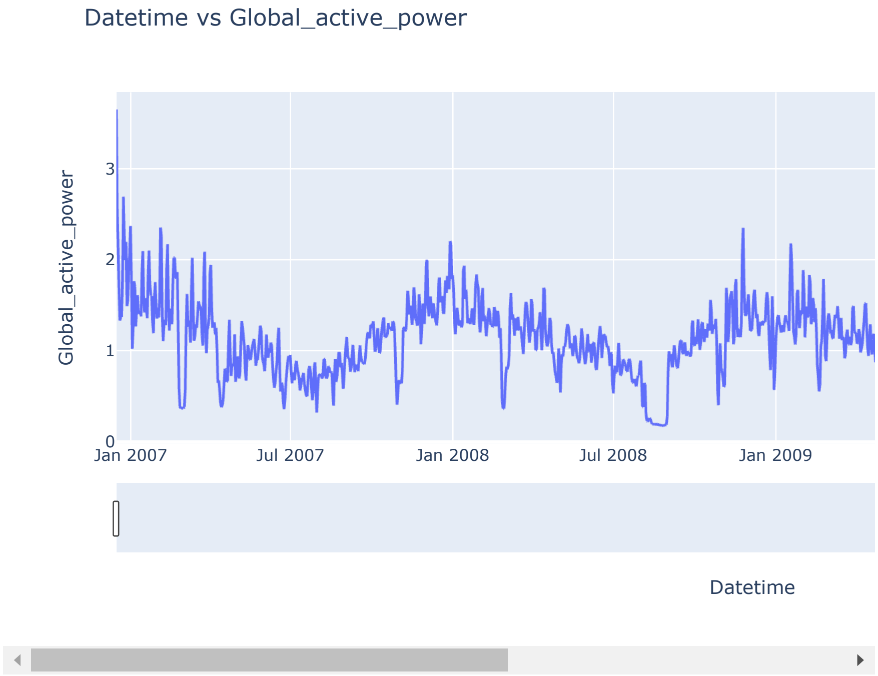



```
# Plotting a histogram to visualize the distribution of 'Global_active_power' values
df[['Global_active_power']].hist()

# Explanation:
# - This code generates a histogram to explore the distribution of the 'Global_active_power' variable.
# - Histograms are useful for understanding the frequency distribution of a continuous variable,
```

```
array([[<Axes: title={'center': 'Global_active_power'}>]], dtype=object)
```

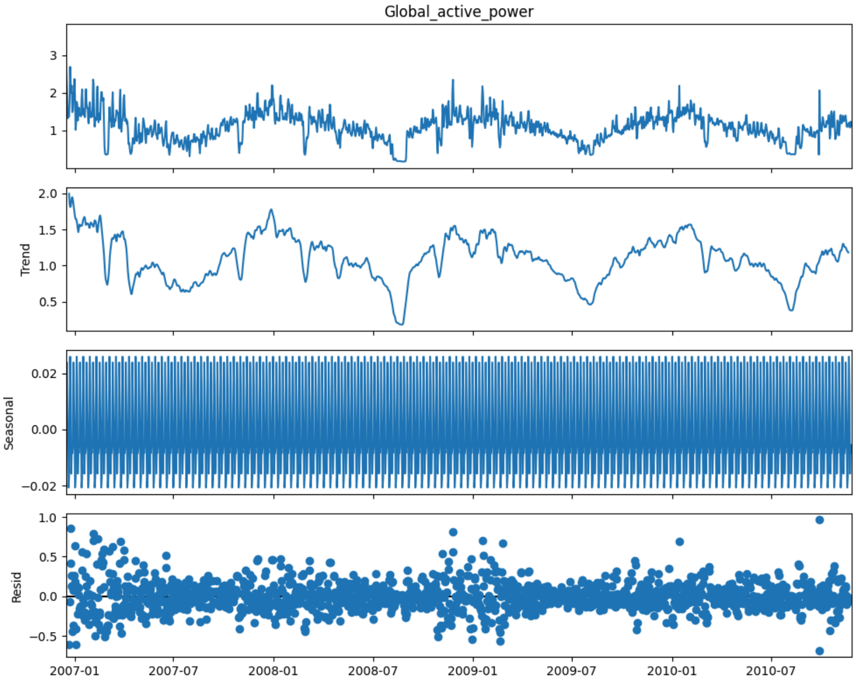



```
# Reset the index of the DataFrame to convert the 'Datetime' index back into a column
dff = df.reset_index()

# Import the Plotly Express library for interactive plotting
import plotly.express as px

# Create a line plot using Plotly Express
fig = px.line(dff, x='Datetime', y='Global_active_power', title='Datetime vs Global_active_power')

# Enable the range slider on the x-axis for zooming and panning in the plot
fig.update_xaxes(rangeslider_visible=True)

# Display the interactive plot
fig.show()
```




```
!pip install statsmodels
import statsmodels.api as sm  # Importing the seasonal decomposition function from the statsmodels library
df['Global_active_power'] = df['Global_active_power'].fillna(df['Global_active_power'].mean())
# Perform seasonal decomposition of the 'Global_active_power' time series data
# Model: 'additive' - assumes that the seasonal and trend components are additive
# Period: 12 - specifies the length of the seasonal cycle (e.g., if the data is daily and periodic, use 7 for weekly, 12 for monthly, e
res = sm.tsa.seasonal_decompose(df['Global_active_power'], model='additive', period=12)

# Plot the seasonal decomposition results
resp = res.plot()

# Display the plot
plt.show()

# Explanation:
# - Seasonal decomposition is a technique used to separate a time series into its constituent components: trend, seasonal, and residual
# - The 'sm.tsa.seasonal_decompose()' function from the statsmodels library performs this decomposition.
# - Model: 'additive' assumes that the seasonal and trend components are additive, meaning the observed data is the sum of trend, seaso
# - Period: 30 specifies the length of the seasonal cycle, indicating that the data exhibits a pattern that repeats every 30 time point
# - The result of the decomposition is stored in the 'res' variable.
# - The 'res.plot()' function generates a plot displaying the original time series, along with the decomposed trend, seasonal, and resi
# - The 'plt.show()' function displays the plot.
```

```
Requirement already satisfied: statsmodels in /usr/local/lib/python3.10/dist-package:
Requirement already satisfied: numpy>=1.22.3 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: scipy!=1.9.2,>=1.8 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pandas!=2.1.0,>=1.4 in /usr/local/lib/python3.10/dist
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/d
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from
```



(p), **(d)**, and **(q)** are the key parameters that define the autoregressive, differencing, and moving average components, respectively, of an ARIMA model, which is commonly used for time series forecasting and analysis. Adjusting these parameters helps capture the underlying patterns and characteristics of the time series data

From onward we are going to find the max value of p, d, and q as they are the parameter of Arima Model

Adfuller

The adfuller function, part of the statsmodels library in Python, is used for conducting the Augmented Dickey-Fuller (ADF) test. The ADF test is a statistical hypothesis test commonly used in time series analysis to determine whether a unit root is present in the time series data.

A unit root suggests that a time series is non-stationary, meaning its statistical properties such as mean and variance change over time. Stationarity is an important concept in time series analysis because many forecasting models assume that the time series data is stationary, i.e., it has constant statistical properties over time.

The ADF test evaluates the null hypothesis that a unit root is present in the time series data, indicating non-stationarity. If the p-value resulting from the ADF test is less than a chosen significance level (e.g., 0.05), then the null hypothesis is rejected, suggesting that the time series is stationary. Conversely, if the p-value is greater than the significance level, the null hypothesis cannot be rejected, indicating that the time series is non-stationary.

```
from statsmodels.tsa.stattools import adfuller  # Importing the ADF test function from statsmodels library

# Defining a function to check stationarity using the Augmented Dickey-Fuller (ADF) test
def check_stationarity(df):
  # Perform the ADF test on the input time series data
  result = adfuller(df)

  # Print ADF statistic and p-value
  print('ADF statistic:', result[0])
  print('p-value:', result[1])

  # Check if p-value is less than or equal to 0.05 (common significance level)
  if result[1] <= 0.05:
    print('The series is stationary.')  # If p-value is less than or equal to 0.05, the series is considered stationary
  else:
    print('The series is not stationary.')  # If p-value is greater than 0.05, the series is considered non-stationary

# Calling the function to check stationarity of 'Global_active_power' data
check_stationarity(df['Global_active_power'])
```

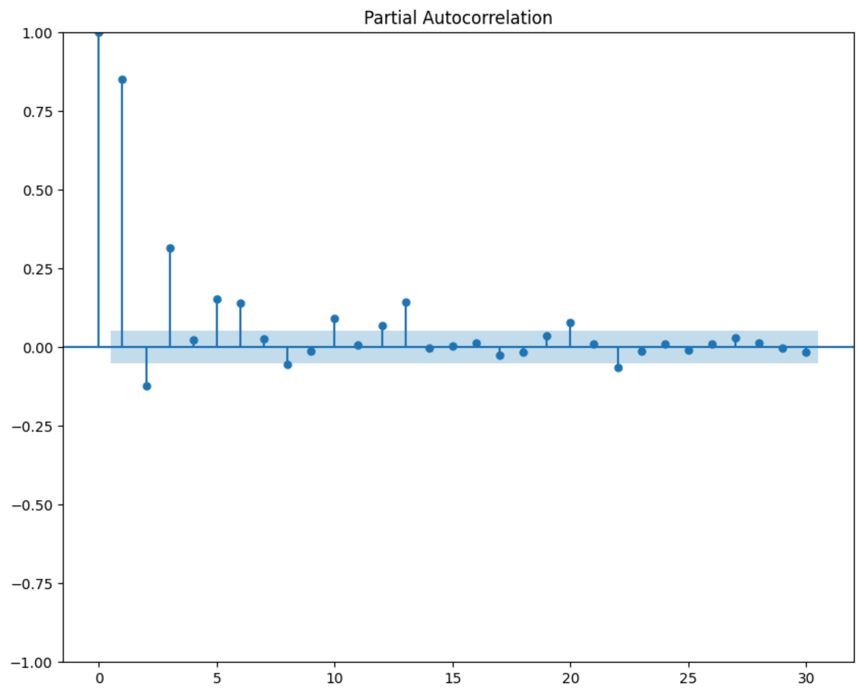
```
ADF statistic: -3.7147563558861165
p-value: 0.003910316001793592
The series is stationary.
```

through acf we are going to find the p

```
from statsmodels.tsa.stattools import acf, pacf  # Importing the autocorrelation function (ACF) and partial autocorrelation function (P

# Calculate the autocorrelation function (ACF) of the 'Global_active_power' time series data
x_acf = pd.DataFrame(acf(df['Global_active_power']))
```

```
p=2
```

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf  # Importing the functions for plotting ACF and PACF from statsmodels lib

# Plot the partial autocorrelation function (PACF) of the 'Global_active_power' time series data
# lags: Number of lags to include in the PACF plot (in this case, up to lag 30)
# alpha: Significance level for confidence intervals (default is 0.05)
plot_pacf(df['Global_active_power'], lags=30, alpha=0.05)

# Display the PACF plot
plt.show()

# Explanation:
# - The 'plot_pacf' function from the statsmodels library is used to plot the partial autocorrelation function (PACF) of a time series.
# - PACF measures the correlation between a time series and its lagged values, while controlling for the effect of shorter lag values.
# - The 'lags' parameter specifies the number of lagged values to include in the PACF plot. Here, it's set to 30.
# - The 'alpha' parameter sets the significance level for confidence intervals. A default value of 0.05 is commonly used.
# - The plot helps in identifying significant partial autocorrelations, which can be used to determine the 'q' value (the number of lag
# - In an ARIMA model, the 'q' value is typically chosen based on significant PACF values that fall outside the confidence intervals.
```



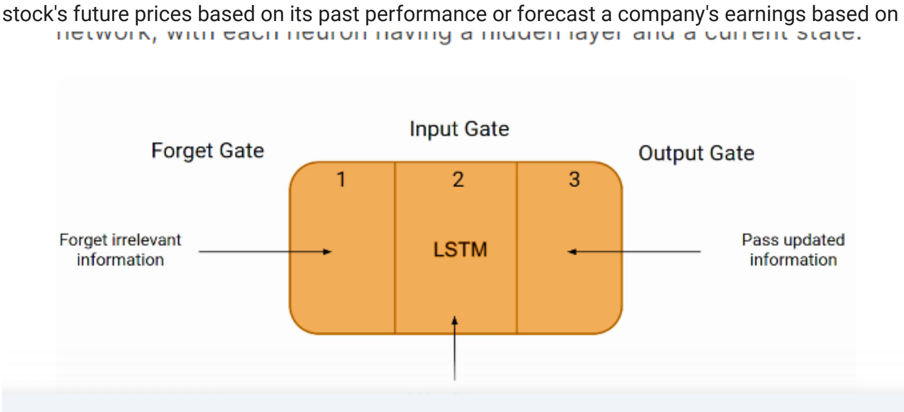
Model Implementation

Describe the model used for forecasting, including any specific configurations and the reasoning behind the choice of this model. Detail the training process.

What Is an Autoregressive Integrated Moving Average (ARIMA)?

An autoregressive integrated moving average, or ARIMA, is a statistical analysis model that uses time series data to either better understand the data set or to predict future trends.

A statistical model is autoregressive if it predicts future values based on past values. For example, an ARIMA model might seek to predict a stock's future prices based on its past performance or forecast a company's earnings based on past periods.



Understanding Autoregressive Integrated Moving Average (ARIMA)

An autoregressive integrated moving average model is a form of regression analysis that gauges the strength of one dependent variable relative to other changing variables. The model's goal is to predict future securities or financial market moves by examining the differences between values in the series instead of through actual values.

An ARIMA model can be understood by outlining each of its components as follows:

Autoregression (AR): refers to a model that shows a changing variable that regresses on its own lagged, or prior, values.

Integrated (I): represents the differencing of raw observations to allow the time series to become stationary (i.e., data values are replaced by the difference between the data values and the previous values).

Moving average (MA): incorporates the dependency between an observation and a residual error from a moving average model applied to lagged observations.

ARIMA Parameters

Each component in ARIMA functions as a parameter with a standard notation. For ARIMA models, a standard notation would be ARIMA with p, d, and q, where integer values substitute for the parameters to indicate the type of ARIMA model used. The parameters can be defined as: 1

p: the number of lag observations in the model, also known as the lag order. d: the number of times the raw observations are differenced; also known as the degree of differencing. q: the size of the moving average window, also known as the order of the moving average.

```
# Installing the pmdarima library using pip
!pip install pmdarima
```

```
Collecting pmdarima
  Downloading pmdarima-2.0.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.manylinux_2_28_x86_64.whl (2.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 21.7 MB/s eta 0:00:00
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.4.2)
Requirement already satisfied: Cython!=0.29.18,!=0.29.31,>=0.29 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (3.0.10)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.25.2)
Requirement already satisfied: pandas>=0.19 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (2.0.3)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.2.2)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.11.4)
Requirement already satisfied: statsmodels>=0.13.2 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (0.14.2)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (2.0.7)
Requirement already satisfied: setuptools!=50.0.0,>=38.6.0 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (67.7.2)
Requirement already satisfied: packaging>=17.1 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (24.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima) (2.8
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima) (2024.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.22->pmdarima)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.10/dist-packages (from statsmodels>=0.13.2->pmdarima) (0.5.6)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from patsy>=0.5.6->statsmodels>=0.13.2->pmdarima) (1
Installing collected packages: pmdarima
Successfully installed pmdarima-2.0.4
```

```
from pmdarima.arima import auto_arima  # Importing the AutoARIMA function from the pmdarima library
from sklearn.metrics import mean_squared_error  # Importing the mean squared error function from scikit-learn

# Calculating the size of the training set as 80% of the total data
train_size = int(len(df) * 0.8)

# Splitting the data into training and testing sets
train, test = df.iloc[:train_size], df.iloc[train_size:]

# Printing the shape of the training set
print(train.shape)

# Explanation:
# - The 'auto_arima' function from the pmdarima library is used for automatically selecting the best ARIMA model parameters.
# - The 'mean_squared_error' function from scikit-learn is used to calculate the mean squared error, which is a measure of the model's
# - The 'train_size' variable is calculated as 80% of the total length of the DataFrame 'df', which will be used to split the data into
# - The 'train' DataFrame contains the first 80% of the data, which will be used for model training.
# - The 'test' DataFrame contains the remaining 20% of the data, which will be used for model evaluation.
# - The shape of the training set is printed to verify the number of rows and columns in the training data.
```

```
(1153, 7)
```

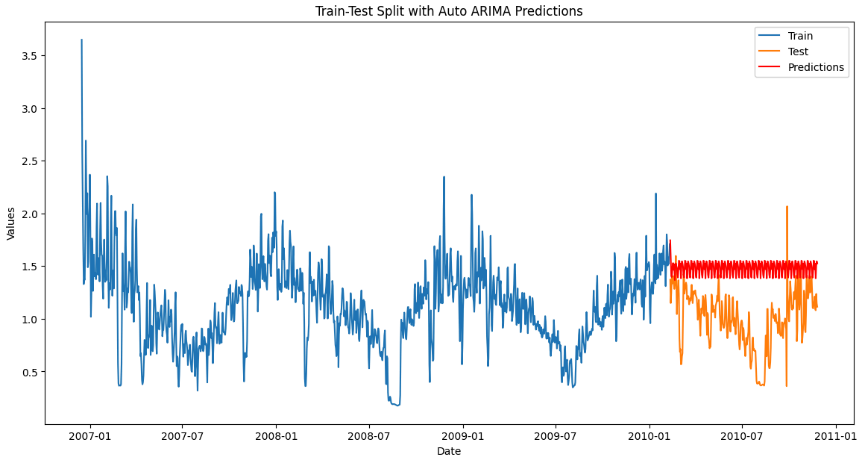
```
model = auto_arima(train['Global_active_power'], start_P=1, start_q=1, max_p = 2, max_q = 2, m=12, start_p = 0 , seasonal=True, d=0,D=1
```

```
Performing stepwise search to minimize aic
 ARIMA(0,0,1)(1,1,1)[12] intercept   : AIC=-335.276, Time=7.20 sec
 ARIMA(0,0,0)(0,1,0)[12] intercept   : AIC=1000.275, Time=0.49 sec
 ARIMA(1,0,0)(1,1,0)[12] intercept   : AIC=-117.396, Time=7.02 sec
 ARIMA(0,0,1)(0,1,1)[12] intercept   : AIC=-336.681, Time=9.31 sec
 ARIMA(0,0,0)(0,1,0)[12]             : AIC=998.516, Time=0.64 sec
 ARIMA(0,0,1)(0,1,0)[12] intercept   : AIC=-25.456, Time=4.87 sec
 ARIMA(0,0,1)(0,1,2)[12] intercept   : AIC=-335.420, Time=24.08 sec
 ARIMA(0,0,1)(1,1,0)[12] intercept   : AIC=-237.134, Time=9.55 sec
 ARIMA(0,0,1)(1,1,2)[12] intercept   : AIC=-335.860, Time=33.07 sec
 ARIMA(0,0,0)(0,1,1)[12] intercept   : AIC=723.427, Time=1.85 sec
 ARIMA(1,0,1)(0,1,1)[12] intercept   : AIC=inf, Time=14.60 sec
 ARIMA(0,0,2)(0,1,1)[12] intercept   : AIC=-590.083, Time=8.72 sec
```

```
 ARIMA(0,0,2)(0,1,0)[12] intercept   : AIC=-138.630, Time=2.76 sec
 ARIMA(0,0,2)(1,1,1)[12] intercept   : AIC=-592.201, Time=8.91 sec
 ARIMA(0,0,2)(1,1,0)[12] intercept   : AIC=-450.406, Time=5.89 sec
 ARIMA(0,0,2)(2,1,1)[12] intercept   : AIC=-590.458, Time=21.30 sec
 ARIMA(0,0,2)(1,1,2)[12] intercept   : AIC=-590.783, Time=36.78 sec
 ARIMA(0,0,2)(0,1,2)[12] intercept   : AIC=-592.379, Time=19.58 sec
 ARIMA(1,0,2)(0,1,2)[12] intercept   : AIC=inf, Time=36.29 sec
 ARIMA(1,0,1)(0,1,2)[12] intercept   : AIC=inf, Time=34.89 sec
 ARIMA(0,0,2)(0,1,2)[12]             : AIC=-593.954, Time=6.87 sec
 ARIMA(0,0,2)(0,1,1)[12]             : AIC=-591.641, Time=3.40 sec
 ARIMA(0,0,2)(1,1,2)[12]             : AIC=-592.359, Time=16.74 sec
 ARIMA(0,0,2)(1,1,1)[12]             : AIC=-593.775, Time=4.85 sec
 ARIMA(0,0,1)(0,1,2)[12]             : AIC=-336.925, Time=4.02 sec
 ARIMA(1,0,2)(0,1,2)[12]             : AIC=inf, Time=27.95 sec
 ARIMA(1,0,1)(0,1,2)[12]             : AIC=inf, Time=22.27 sec

Best model:  ARIMA(0,0,2)(0,1,2)[12]
Total fit time: 373.970 seconds
```

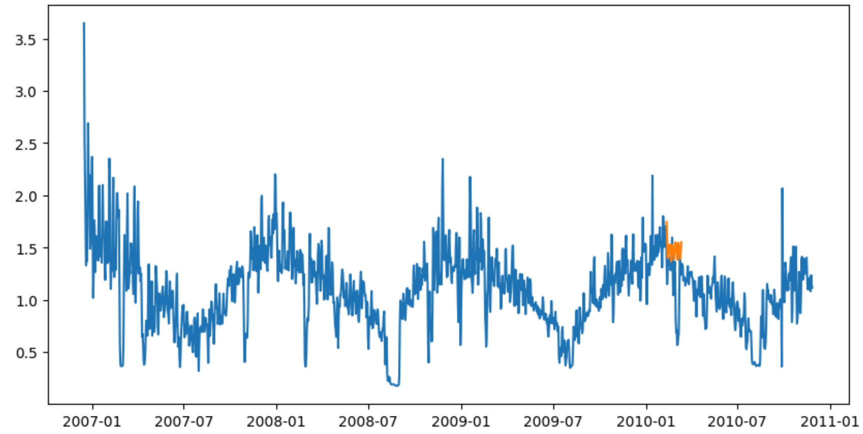
```
predictions = model.predict(n_periods=len(test))
# Evaluate the model
mse = mean_squared_error(test['Global_active_power'], predictions)
print(f'Mean Squared Error: {mse}')
# mae = mean_absolute_error(test['Global_active_power'], predictions)
# Visualize the results
plt.figure(figsize=(14, 7))
plt.plot(train['Global_active_power'], label='Train')
plt.plot(test['Global_active_power'], label='Test')
plt.plot(test.index, predictions, label='Predictions', color='red')
plt.xlabel('Date')
plt.ylabel('Values')
plt.title('Train-Test Split with Auto ARIMA Predictions')
plt.legend()
plt.show()
```

```
Mean Squared Error: 0.30848257337146007
```




```
prediction = model.predict(n_periods =30)
plt.figure(figsize = (10,5))
plt.plot(df['Global_active_power'], label = 'Actual')
plt.plot(prediction, label = 'predict')
```

```
[<matplotlib.lines.Line2D at 0x7e82cc1491e0>]
```



Analysis of ARIMA prediction.

```python
import pandas as pd

# Create a new DataFrame to store the results
results_df_arima = pd.DataFrame({
    'Time': test.index,
    'Test': test['Global_active_power'],
    'Predicted': predictions,
    'Predicted at 30': prediction
})
# Display the results
print(results_df_arima.to_string())
```

```
                 Time      Test  Predicted  Predicted at 30
2010-02-11 2010-02-11  1.370129   1.745159         1.745159
2010-02-12 2010-02-12  1.149385   1.576304         1.576304
2010-02-13 2010-02-13  1.246715   1.409485         1.409485
2010-02-14 2010-02-14  1.413374   1.396758         1.396758
2010-02-15 2010-02-15  1.335080   1.456625         1.456625
2010-02-16 2010-02-16  1.347082   1.526677         1.526677
2010-02-17 2010-02-17  1.404992   1.527453         1.527453
2010-02-18 2010-02-18  1.325865   1.464802         1.464802
2010-02-19 2010-02-19  1.283419   1.517004         1.517004
2010-02-20 2010-02-20  1.315336   1.368110         1.368110
2010-02-21 2010-02-21  1.446671   1.492608         1.492608
2010-02-22 2010-02-22  1.596793   1.526251         1.526251
2010-02-23 2010-02-23  1.339367   1.501842         1.501842
2010-02-24 2010-02-24  1.044249   1.485576         1.485576
2010-02-25 2010-02-25  1.228324   1.418424         1.418424
2010-02-26 2010-02-26  1.301737   1.390658         1.390658
2010-02-27 2010-02-27  1.366274   1.462267         1.462267
2010-02-28 2010-02-28  1.225704   1.551805         1.551805
2010-03-01 2010-03-01  0.813236   1.546372         1.546372
2010-03-02 2010-03-02  0.686661   1.470142         1.470142
2010-03-03 2010-03-03  0.711656   1.523276         1.523276
2010-03-04 2010-03-04  0.567246   1.384221         1.384221
2010-03-05 2010-03-05  0.573142   1.503286         1.503286
2010-03-06 2010-03-06  0.642575   1.545237         1.545237
2010-03-07 2010-03-07  0.674001   1.525040         1.525040
2010-03-08 2010-03-08  1.006776   1.494715         1.494715
2010-03-09 2010-03-09  1.425788   1.418424         1.418424
2010-03-10 2010-03-10  1.402596   1.390658         1.390658
2010-03-11 2010-03-11  1.238604   1.462267         1.462267
2010-03-12 2010-03-12  1.280396   1.551805         1.551805
2010-03-13 2010-03-13  1.310846   1.546372              NaN
2010-03-14 2010-03-14  1.343244   1.470142              NaN
2010-03-15 2010-03-15  1.316062   1.523276              NaN
2010-03-16 2010-03-16  1.184744   1.384221              NaN
2010-03-17 2010-03-17  1.205209   1.503286              NaN
2010-03-18 2010-03-18  1.216987   1.545237              NaN
2010-03-19 2010-03-19  1.206704   1.525040              NaN
2010-03-20 2010-03-20  1.137021   1.494715              NaN
2010-03-21 2010-03-21  1.166188   1.418424              NaN
2010-03-22 2010-03-22  1.193298   1.390658              NaN
2010-03-23 2010-03-23  1.172552   1.462267              NaN
2010-03-24 2010-03-24  1.246277   1.551805              NaN
2010-03-25 2010-03-25  1.322536   1.546372              NaN
```

```
2010-03-26 2010-03-26  1.209062   1.470142             NaN
2010-03-27 2010-03-27  1.149051   1.523276             NaN
2010-03-28 2010-03-28  1.223614   1.384221             NaN
2010-03-29 2010-03-29  1.261695   1.503286             NaN
2010-03-30 2010-03-30  1.236454   1.545237             NaN
2010-03-31 2010-03-31  1.257302   1.525040             NaN
2010-04-01 2010-04-01  1.269737   1.494715             NaN
2010-04-02 2010-04-02  1.223701   1.418424             NaN
2010-04-03 2010-04-03  1.211248   1.390658             NaN
2010-04-04 2010-04-04  1.194900   1.462267             NaN
2010-04-05 2010-04-05  1.105225   1.551805             NaN
2010-04-06 2010-04-06  0.990835   1.546372             NaN
2010-04-07 2010-04-07  1.090923   1.470142             NaN
```

```
# prompt: write a code to calculate and print MSE and MAE for predicted values

from sklearn.metrics import mean_squared_error, mean_absolute_error

# Calculate the mean squared error
mse_arima = mean_squared_error(test['Global_active_power'], predictions)

# Calculate the mean absolute error
mae_arima = mean_absolute_error(test['Global_active_power'], predictions)

# Print the results
print(f'Mean Squared Error (MSE): {mse_arima}')
print(f'Mean Absolute Error (MAE): {mae_arima}')
```

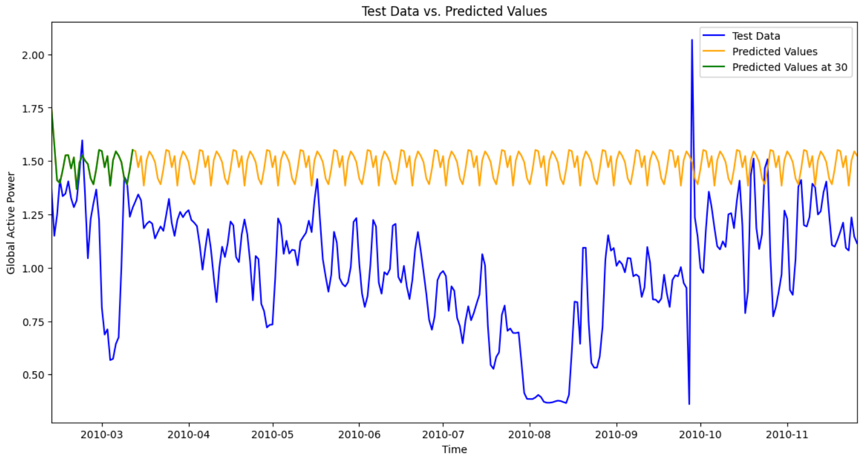
```
Mean Squared Error (MSE): 0.30848257337146007
Mean Absolute Error (MAE): 0.4850205480219283
```

```
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from ipywidgets import interactive
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Assuming results_df is already defined with relevant columns: 'Time', 'Test', 'Predicted', 'Predicted at 30'

# Extract relevant columns from results_df
time = results_df_arima['Time']
test_data = results_df_arima['Test']
predicted = results_df_arima['Predicted']
predicted_30 = results_df_arima['Predicted at 30']

# Calculate MSE and MAE
mse_predicted = mean_squared_error(test_data, predicted)
mae_predicted = mean_absolute_error(test_data, predicted)
# mse_predicted_30 = mean_squared_error(test_data, predicted_30)
# mae_predicted_30 = mean_absolute_error(test_data, predicted_30)

print(f"MSE for Predicted Values: {mse_predicted}")
print(f"MAE for Predicted Values: {mae_predicted}")
# print(f"MSE for Predicted Values at 30: {mse_predicted_30}")
# print(f"MAE for Predicted Values at 30: {mae_predicted_30}")

# Create the plot
def plot_data(x_min=0, x_max=len(time)-1):
    plt.figure(figsize=(14, 7))

    # Plot the test data
    plt.plot(time, test_data, label='Test Data', color='blue')

    # Plot the predicted values
    plt.plot(time, predicted, label='Predicted Values', color='orange')

    # Plot the predicted values at 30
    plt.plot(time, predicted_30, label='Predicted Values at 30', color='green')

    # Add labels and title
    plt.xlabel('Time')
    plt.ylabel('Global Active Power')
    plt.title('Test Data vs. Predicted Values')

    # Add legend
    plt.legend()

    # Set x-axis limits for zooming
    plt.xlim(time.iloc[x_min], time.iloc[x_max])

    plt.show()

# Interactive widget for the range slider
interactive_plot = interactive(plot_data, x_min=(0, len(time)-1, 1), x_max=(0, len(time)-1, 1))
output = interactive_plot.children[-1]
output.layout.height = '450px'
interactive_plot
```

```
MSE for Predicted Values: 0.30848257337146007
MAE for Predicted Values: 0.4850205480219283
```

x_min 0

x_max 288



ARIMA ANALYSIS END HERE

What is LSTM? Introduction to Long Short-Term Memory

Introduction

Long Short-Term Memory Networks is a deep learning, sequential neural network that allows information to persist. It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN. LSTM was designed by Hochreiter and Schmidhuber that resolves the problem caused by traditional rnns and machine learning algorithms. LSTM Model can be implemented in Python using the Keras library.

Let’s say while watching a video, you remember the previous scene, or while reading a book, you know what happened in the earlier chapter. RNNs work similarly; they remember the previous information and use it for processing the current input. The shortcoming of RNN is they cannot remember long-term dependencies due to vanishing gradient. LSTMs are explicitly designed to avoid long-term dependency problems.

The Logic Behind LSTM

The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten. In the second part, the cell tries to learn new information from the input to this cell. At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp. This one cycle of LSTM is considered a single-time step.

These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell. The first gate is called Forget gate, the second gate is known as the Input gate, and the last one is the Output gate. An LSTM unit that consists of these three gates and a memory cell or lstm cell can be considered as a layer of neurons in traditional feedforward neural network, with each neuron having a hidden layer and a current state.

LSTM gates, LSTM Models Just like a simple RNN, an LSTM also has a hidden state where H(t-1) represents the hidden state of the previous timestamp and Ht is the hidden state of the current timestamp. In addition to that, LSTM also has a cell state represented by C(t-1) and C(t) for the previous and current timestamps, respectively.

Here the hidden state is known as Short term memory, and the cell state is known as Long term memory. Refer to the following image.

LSTM memory It is interesting to note that the cell state carries the information along with all the timestamps.

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

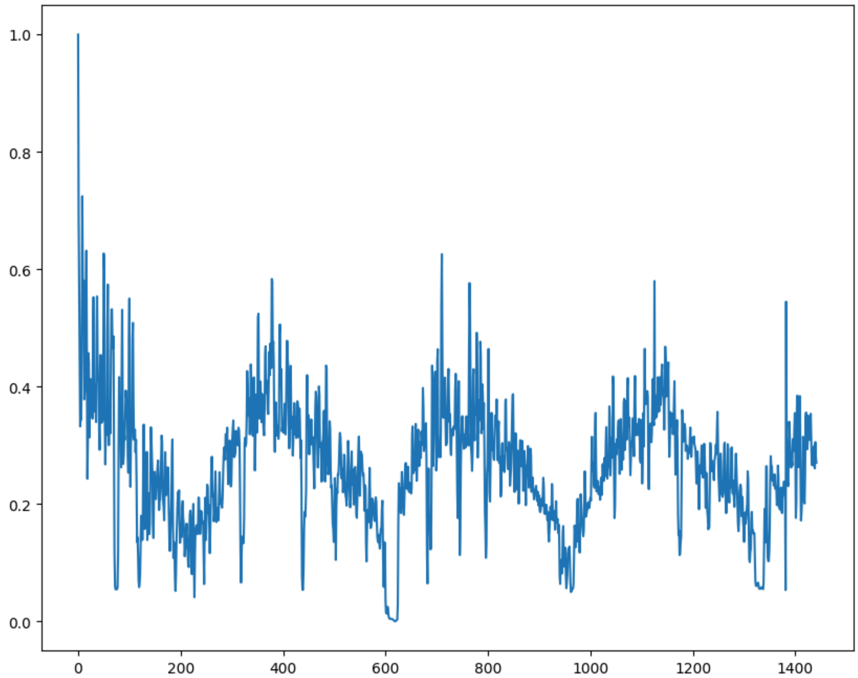
Double-click (or enter) to edit

```
df1=df['Global_active_power']
```

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))
df1 = scaler.fit_transform(np.array(df1).reshape(-1,1))
```

```
plt.plot(df1)
```

```
[<matplotlib.lines.Line2D at 0x7e82cc1de1a0>]
```




```
training_size = int(len(df1)*0.65)
test_size = len(df1)-training_size
train = df1[0:training_size]
train.shape
```

```
(937, 1)
```

```
test = df1[training_size:len(df1)]
print(training_size,"  ",test_size)
```

```
937    505
```

```
import numpy
def create_dataset(dataset, time_step=1):
  datax,datay= [],[]
  for i in range(len(dataset)-time_step-1):
    a=dataset[i:(i+time_step),0]
    datax.append(a)
    datay.append(dataset[i+time_step,0])
  return numpy.array(datax), numpy.array(datay)
```

```
time_step=100
x_train , y_train = create_dataset(train,time_step)
x_test, y_test=create_dataset(test, time_step)
```

```
x_train.shape
```

```
(836, 100)
```

```
x_train= x_train.reshape(x_train.shape[0], x_train.shape[1],1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1],1)
print(x_test)
```

```
[[[0.15075412]
  [0.16502131]
  [0.14408894]
  ...
  [0.25998763]
  [0.2649147 ]
  [0.28464391]]

 [[0.16502131]
  [0.14408894]
  [0.07726015]
  ...
  [0.2649147 ]
  [0.28464391]
  [0.36634038]]

 [[0.14408894]
  [0.07726015]
  [0.06367757]
  ...
  [0.28464391]
  [0.36634038]
  [0.34320546]]

 ...

 [[0.06603806]
  [0.12428257]
  [0.19167163]
  ...
  [0.28597446]
  [0.2983088 ]
  [0.26412513]]

 [[0.12428257]
  [0.19167163]
  [0.19089739]
  ...
  [0.2983088 ]
  [0.26412513]
  [0.26065283]]

 [[0.19167163]
  [0.19089739]
  [0.13482776]
  ...
  [0.26412513]
  [0.26065283]
  [0.30531983]]]
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM


model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(100,1)))
model.add(LSTM(50, return_sequences=True))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')


model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 lstm (LSTM)                 (None, 100, 50)           10400     
                                                                 
 lstm_1 (LSTM)               (None, 100, 50)           20200     
                                                                 
 lstm_2 (LSTM)               (None, 50)                20200     
                                                                 
 dense (Dense)               (None, 1)                 51        
                                                                 
=================================================================
Total params: 50851 (198.64 KB)
Trainable params: 50851 (198.64 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=100, batch_size=64, verbose=1)
```

```
Epoch 1/100
14/14 [==============================] - 20s 818ms/step - loss: 0.0162 - val_loss: 0.0082
Epoch 2/100
14/14 [==============================] - 6s 417ms/step - loss: 0.0067 - val_loss: 0.0041
Epoch 3/100
14/14 [==============================] - 6s 458ms/step - loss: 0.0060 - val_loss: 0.0040
Epoch 4/100
14/14 [==============================] - 9s 623ms/step - loss: 0.0059 - val_loss: 0.0043
Epoch 5/100
14/14 [==============================] - 10s 707ms/step - loss: 0.0060 - val_loss: 0.0039
Epoch 6/100
14/14 [==============================] - 8s 510ms/step - loss: 0.0060 - val_loss: 0.0039
Epoch 7/100
14/14 [==============================] - 8s 587ms/step - loss: 0.0057 - val_loss: 0.0040
Epoch 8/100
14/14 [==============================] - 6s 413ms/step - loss: 0.0056 - val_loss: 0.0038
Epoch 9/100
14/14 [==============================] - 3s 185ms/step - loss: 0.0057 - val_loss: 0.0037
Epoch 10/100
14/14 [==============================] - 3s 186ms/step - loss: 0.0054 - val_loss: 0.0037
Epoch 11/100
14/14 [==============================] - 3s 230ms/step - loss: 0.0056 - val_loss: 0.0037
Epoch 12/100
14/14 [==============================] - 4s 309ms/step - loss: 0.0054 - val_loss: 0.0037
Epoch 13/100
14/14 [==============================] - 3s 186ms/step - loss: 0.0053 - val_loss: 0.0037
Epoch 14/100
14/14 [==============================] - 3s 181ms/step - loss: 0.0056 - val_loss: 0.0041
Epoch 15/100
14/14 [==============================] - 3s 184ms/step - loss: 0.0055 - val_loss: 0.0038
Epoch 16/100
14/14 [==============================] - 3s 193ms/step - loss: 0.0052 - val_loss: 0.0036
Epoch 17/100
14/14 [==============================] - 4s 306ms/step - loss: 0.0052 - val_loss: 0.0036
Epoch 18/100
14/14 [==============================] - 3s 214ms/step - loss: 0.0052 - val_loss: 0.0044
Epoch 19/100
14/14 [==============================] - 3s 183ms/step - loss: 0.0061 - val_loss: 0.0045
Epoch 20/100
14/14 [==============================] - 3s 234ms/step - loss: 0.0053 - val_loss: 0.0035
Epoch 21/100
14/14 [==============================] - 3s 209ms/step - loss: 0.0050 - val_loss: 0.0035
Epoch 22/100
14/14 [==============================] - 4s 304ms/step - loss: 0.0049 - val_loss: 0.0034
Epoch 23/100
14/14 [==============================] - 3s 190ms/step - loss: 0.0048 - val_loss: 0.0034
Epoch 24/100
14/14 [==============================] - 3s 184ms/step - loss: 0.0048 - val_loss: 0.0034
Epoch 25/100
14/14 [==============================] - 3s 185ms/step - loss: 0.0047 - val_loss: 0.0034
Epoch 26/100
14/14 [==============================] - 3s 205ms/step - loss: 0.0048 - val_loss: 0.0038
Epoch 27/100
14/14 [==============================] - 4s 306ms/step - loss: 0.0054 - val_loss: 0.0047
Epoch 28/100
14/14 [==============================] - 3s 196ms/step - loss: 0.0053 - val_loss: 0.0035
Epoch 29/100
14/14 [==============================] - 3s 184ms/step - loss: 0.0050 - val_loss: 0.0033
```

```
import tensorflow as tf
train_pred = model.predict(x_train)
test_pred = model.predict(x_test)
```

```
27/27 [==============================] - 4s 63ms/step
13/13 [==============================] - 1s 37ms/step
```

```
train_pred = scaler.inverse_transform(train_pred)
test_pred = scaler.inverse_transform(test_pred)
```

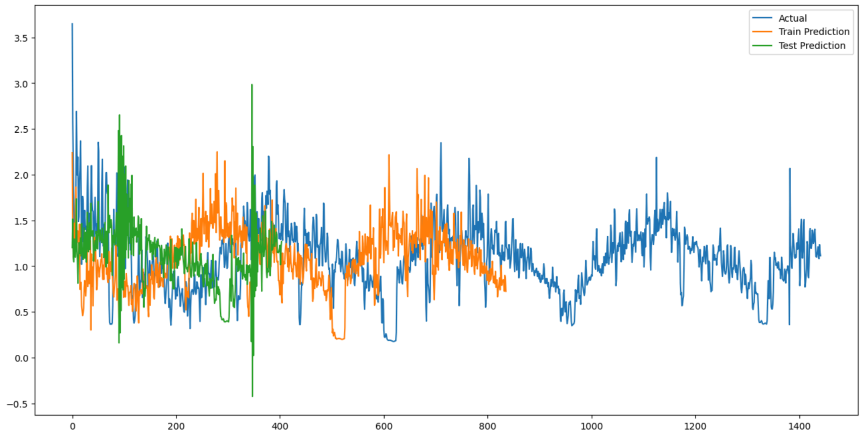
```
import math
from sklearn.metrics import mean_squared_error
math.sqrt(mean_squared_error(y_train, train_pred))
```

```
0.8551467037672137
```

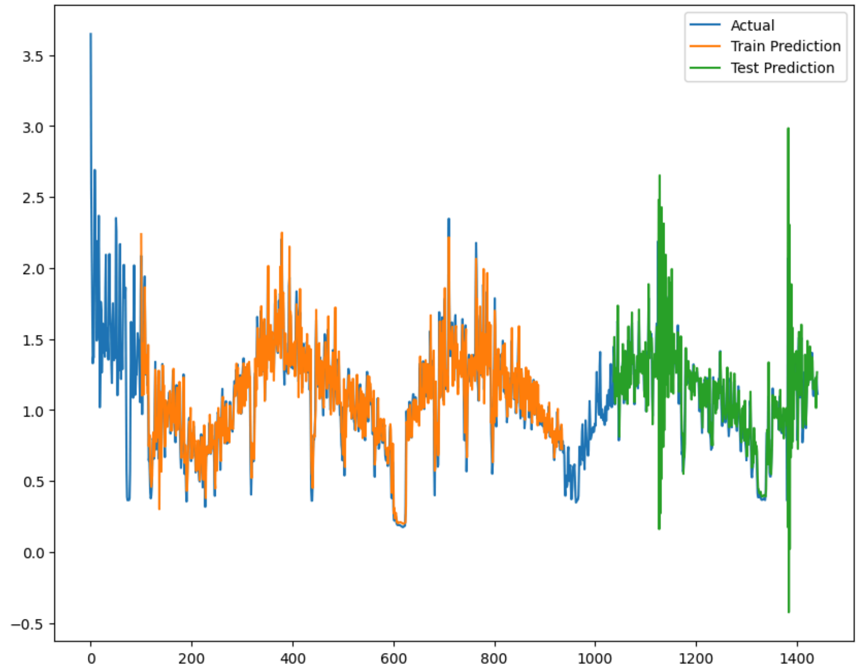
```
math.sqrt(mean_squared_error(y_test, test_pred))
```

```
0.903278776555784
```

```
# prompt: show legend in
# plt.plot(scaler.inverse_transform(df1))
# plt.plot(train_predict_plot)
# plt.plot(test_predict_plot)
# plt.show()

import matplotlib.pyplot as plt
plt.figure(figsize=(16, 8))
plt.plot(scaler.inverse_transform(df1), label='Actual')
plt.plot(train_pred, label='Train Prediction')
plt.plot(test_pred, label='Test Prediction')
plt.legend()
plt.show()
```




```
look_back=100
train_predict_plot = numpy.empty_like(df1)
train_predict_plot[:,:]=np.nan
train_predict_plot[look_back:len(train_pred)+look_back,:]=train_pred

test_predict_plot = numpy.empty_like(df1)
test_predict_plot[:,:]=np.nan
test_predict_plot[len(train_pred)+(look_back*2)+1:len(df1)-1,:]=test_pred
math.sqrt(mean_squared_error(y_test, test_pred))
plt.plot(scaler.inverse_transform(df1), label='Actual')
plt.plot(train_predict_plot, label='Train Prediction')
plt.plot(test_predict_plot, label='Test Prediction')
plt.legend()
plt.show()
```




```
x_input=test[405:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()


# demonstrate prediction for next 10 days
from numpy import array

lst_output=[]
n_steps=100
i=0
while(i<30):

    if(len(temp_input)>100):
        #print(temp_input)
        x_input=np.array(temp_input[1:])
        print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)
        x_input = x_input.reshape((1, n_steps, 1))
        #print(x_input)
        yhat = model.predict(x_input, verbose=0)
        print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat[0].tolist())
        temp_input=temp_input[1:]
        #print(temp_input)
        lst_output.extend(yhat.tolist())
        i=i+1
    else:
        x_input = x_input.reshape((1, n_steps,1))
        yhat = model.predict(x_input, verbose=0)
        print(yhat[0])
        temp_input.extend(yhat[0].tolist())
        print(len(temp_input))
        lst_output.extend(yhat.tolist())
        i=i+1


print(lst_output)
```

```
[0.30179492]
101
1 day input [0.26429842 0.26429842 0.16590271 0.10896857 0.10256287 0.10263852
 0.11787519 0.15778648 0.2479586  0.28135398 0.26045738 0.26404051
 0.23996535 0.24658456 0.2421599  0.23115366 0.25062441 0.2499752
 0.22587886 0.22824036 0.22544463 0.19798425 0.21084114 0.26542255
 0.24408893 0.19466299 0.19434514 0.19042695 0.19566352 0.22798174
 0.20251641 0.18448134 0.22090313 0.22730432 0.22584919 0.23838311
```

```
0.21694336 0.21020444 0.05336634 0.54486168 0.3052322  0.27924972
0.23713284 0.23040239 0.29191688 0.34007232 0.31889202 0.29131727
0.26641598 0.26210159 0.27302258 0.26547755 0.30942493 0.31105969
0.29080046 0.3274941  0.35489909 0.2944174  0.17614708 0.2060974
```