Linear Regression Analysis and Prediction for IoT

This notebook holds the Assignment 3.1 for Module 3 in AAI 530, Data Analytics and the Internet of Things. In this assignment, you will use linear regression to make predictions for simulated "streaming" data. The work that you do in this assignment will build on the linear regression predictions that you saw in your text book and in this week's lab session. Be sure to answer the analysis questions thoroughly, as this is a large part of the assignment for this week.

General Assignment Instructions

These instructions are included in every assignment, to remind you of the coding standards for the class. Feel free to delete this cell after reading it.

One sign of mature code is conforming to a style guide. We recommend the Google Python Style Guide. If you use a different style guide, please include a cell with a link.

Your code should be relatively easy-to-read, sensibly commented, and clean. Writing code is a messy process, so please be sure to edit your final submission. Remove any cells that are not needed or parts of cells that contain unnecessary code. Remove inessential `import` statements and make sure that all such statements are moved into the designated cell.

When you save your notebook as a pdf, make sure that all cell output is visible (even error messages) as this will aid your instructor in grading your work.

Make use of non-code cells for written commentary. These cells should be grammatical and clearly written. In some of these cells you will have questions to answer. The questions will be marked by a "Q:" and will have a corresponding "A:" spot for you. *Make sure to answer every question marked with a* `Q:` *for full credit.*

In [210…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

#suppress scientific notation in pandas
pd.set_option('display.float_format', lambda x: '%.5f' % x)
```

In [211…
```
from datetime import datetime as dt
from sklearn.metrics import mean_squared_error
```

Load and prepare your data

We'll be using the cleaned household electric consumption dataset from Module 2 in this assignment. I recommend saving your dataset by running df.to_csv("filename") at the end of the last assignment so that you don't have to re-do your cleaning steps. If you are not confident in your own cleaning steps, you may ask your instructor for a cleaned version of the data. You will not be graded on the cleaning steps in this assignment, but some functions may not work if you use the raw data.

We need to turn our datetime column into a numeric value to be used as a variable in our linear regression. In the lab session, we created a new column of minutes and just incremented the value by 10 since we knew that the readings occurred every 10 minutes. In this dataset, we have readings every minute, but we might have some missing rows depending on how you cleaned your data. So instead we will convert our datetime column to something called unix/epoch time, which is the number of seconds since midnight on 1/1/1970.

TODO: load your data and convert the datetime column into epoch/unix time

In [212…
```
# Load the dataset
data_set = pd.read_csv('household_power_clean.csv', encoding='latin')

# Display the top 5 records in the dataset
data_set.head()
```

Out[212…

	Unnamed: 0	Date	Time	Global_active_power	Global_reactive_power	Voltag
0	0	2006-12-16	17:24:00	4.21600	0.41800	234.8400
1	1	2006-12-16	17:25:00	5.36000	0.43600	233.6300
2	2	2006-12-16	17:26:00	5.37400	0.49800	233.2900
3	3	2006-12-16	17:27:00	5.38800	0.50200	233.7400
4	4	2006-12-16	17:28:00	3.66600	0.52800	235.6800

In [ ]:

```
# Convert the string in 'Datetime' column to datetime objects
data_set['Datetime'] = pd.to_datetime(data_set['Datetime'])

# Create a new column 'Unix' by converting 'Datetime' to Unix timestamp
data_set['Unix'] = data_set['Datetime'].apply(lambda x: int(x.timestamp()))

# Print the total number of rows in the dataset
total_records = data_set.shape[0]
print("Total records:", total_records)

# Display the top 5 records in the dataset with the new column 'Unix'
data_set.head()
```

```
Total records: 2049280
```

Out[ ]:

	Unnamed: 0	Date	Time	Global_active_power	Global_reactive_power	Voltag
0	0	2006-12-16	17:24:00	4.21600	0.41800	234.8400
1	1	2006-12-16	17:25:00	5.36000	0.43600	233.6300
2	2	2006-12-16	17:26:00	5.37400	0.49800	233.2900
3	3	2006-12-16	17:27:00	5.38800	0.50200	233.7400
4	4	2006-12-16	17:28:00	3.66600	0.52800	235.6800

Predicting Global Active Power

We will follow the code from the Chapter 9 in our textbook and the recorded lab session from this week to predict the Global Active Power (GAP) with linear regression.

First we will create our x (time) and y (GAP) training variables, and then define our model parameters.

Q: What is ph? What is mu?

A: ph is the Prediction horizon that represents the specific future time interval for which we aim to forecast an outcome based on historical data. For example, if our goal is to predict the values for the next five minutes, an hours, or the next day, that duration is defined by the ph mu is the forgetting factor that represents the weight given to historical versus more recent data in our forecasts. The value ranges from 0 to 1. A value closer to 1 means we are heavily reliant on older historical data, leading to stable and smooth predictions. But, a value closer to 0 means the forecast is going to be more responsive and adaptive.

TODO: Set the ph to be 5 minutes--consider the units that our time column is measured in.

In [257…

```
# Time Series
ts = pd.DataFrame(data_set.Unix)
# Y Series - dependent variable
ys = pd.DataFrame(data_set.Global_active_power)

# Prediction horizon is set to 5 minutes
ph = 5
# ph/data resolution - Data resolution is the time interval between consecut
ph_index = ph / 1
# # Forgetting factor
mu = 0.9

#let's limit the number of samples in our model to 5000 just for speed
n_s = 5000

# Arrays to hold predicted values
tp_pred = np.zeros(n_s-1)
yp_pred = np.zeros(n_s-1)
```

Q: With mu = 0.9, how much weight will our first data point have on the last (5000th) prediction in our limited dataset?

A: Weight of the first data point on the 5000th prediction: 1.8126113170475857e-229

TODO: Following the code from Chapter 10 and the lab session, use linear regression to predict a rolling GAP for our dataset. Store these predictions in the tp_pred and yp_pred lists created above for visualization.

In [258…

```
# import numpy as np
# from sklearn.linear_model import LinearRegression

# Initialize variables
first_data_point_weight_on_5000th = 0

# At every iteration of the for loop a new data sample is acquired
for i in range(2, n_s+1):  # start out with 2 leading datapoints

    # Get x and y data "available" for our prediction
    ts_tmp = np.array(data_set['Unix'][:i]).reshape(-1, 1)  # Convert and re
    ys_tmp = data_set['Global_active_power'][:i]
    ns = len(ys_tmp)

    # Initialize weights with all values set to mu
    weights = np.ones(ns) * mu
    for k in range(ns):
        # Adjust weights to be downweighted according to their timestep away
        weights[k] = mu ** (ns - k - 1)
```

```
    # Initialize weights with exponential decay from the first to the last p
    weights = np.array([mu ** (k) for k in range(ns - 1, -1, -1)])

    # Perform linear regression on "available" data using the mu-adjusted we
    lm_tmp = LinearRegression()
    model_tmp = lm_tmp.fit(ts_tmp, ys_tmp, sample_weight=weights)

    # Store model coefficients and intercepts to compute prediction
    m_tmp = model_tmp.coef_[0]  # slope
    q_tmp = model_tmp.intercept_  # intercept

    # Use ph to make the model prediction according to the prediction time
    latest_time = ts_tmp[-1, 0]  # Get the latest timestamp
    tp = latest_time + ph * 60
    yp = m_tmp * tp + q_tmp  # Calculate the predicted power

    # Capture and print weights for the first few data points at the 5000th
    if i == 5000:
        first_data_point_weight_on_5000th = weights[0]
        print("Weights for the first few data points at the 5000th step:", w
        print("Weight of the first data point on the 5000th prediction:", fi

    if i in [1000, 2000, 3000, 4000, 5000]:
        print(f"At i={i}, model coefficient (m_tmp) = {m_tmp:.10f}, intercep
        print(f"Model prediction at i={i} for latest time {latest_time}: {yp

    tp_pred[i-2] = tp
    yp_pred[i-2] = yp

print("number of samples: ", ns)
```

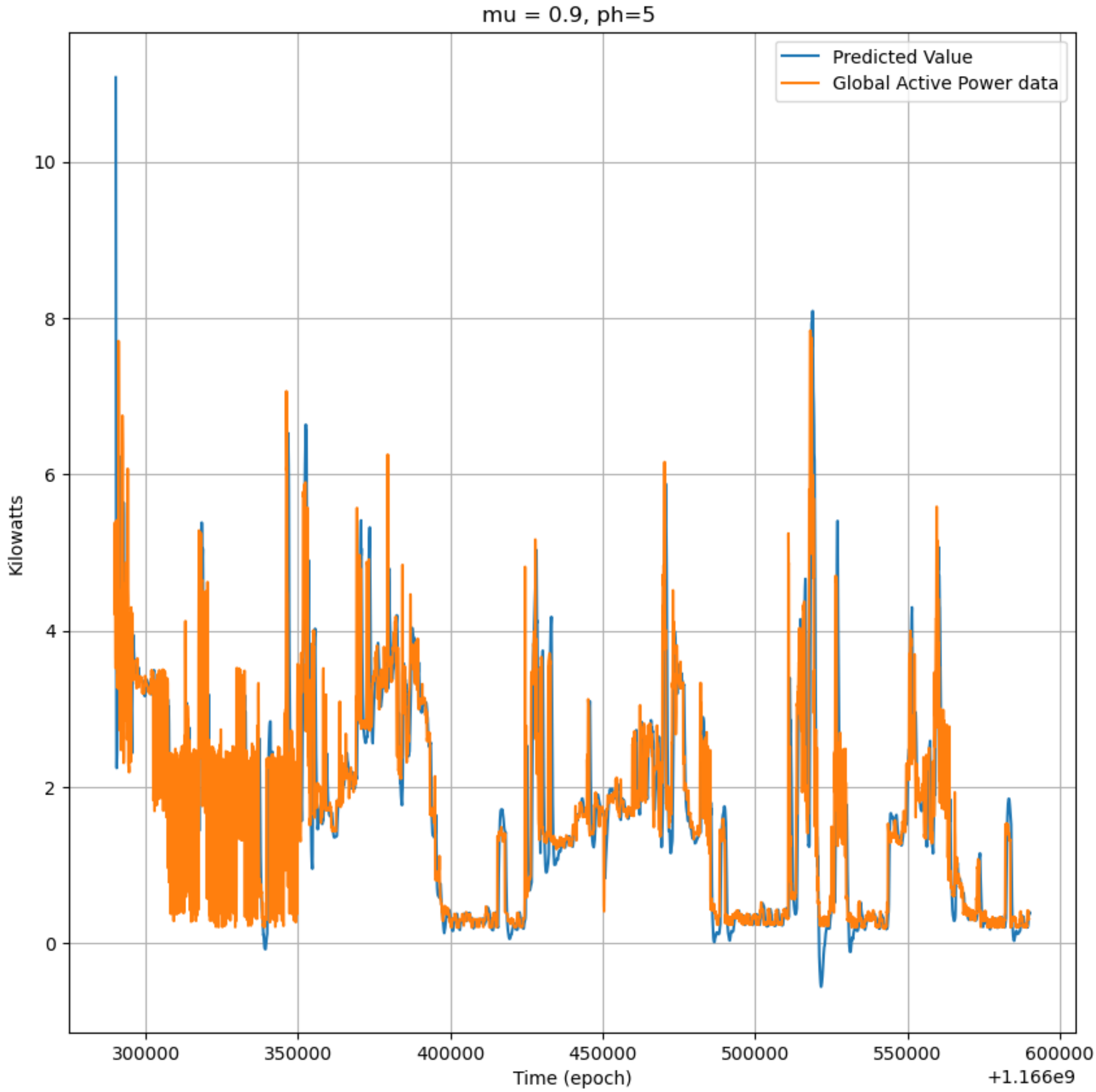
```
At i=1000, model coefficient (m_tmp) = -0.0001537945, intercept (q_tmp) = 17
9379.8501788810
Model prediction at i=1000 for latest time 1166349780: 1.6741979679
At i=2000, model coefficient (m_tmp) = 0.0000156915, intercept (q_tmp) = -18
302.4766463962
Model prediction at i=2000 for latest time 1166409780: 0.2973214251
At i=3000, model coefficient (m_tmp) = 0.0013566094, intercept (q_tmp) = -15
82440.1786136210
Model prediction at i=3000 for latest time 1166469780: 4.0641277418
At i=4000, model coefficient (m_tmp) = -0.0001730065, intercept (q_tmp) = 20
1818.9739590921
Model prediction at i=4000 for latest time 1166529780: 1.6458991628
Weights for the first few data points at the 5000th step: [1.81261132e-229
2.01401257e-229 2.23779175e-229 2.48643528e-229
 2.76270586e-229 3.06967318e-229 3.41074798e-229 3.78971998e-229
 4.21079997e-229 4.67866664e-229]
Weight of the first data point on the 5000th prediction: 1.8126113170475857e
-229
At i=5000, model coefficient (m_tmp) = 0.0000709433, intercept (q_tmp) = -82
761.3893597900
Model prediction at i=5000 for latest time 1166589780: 0.3809734717
number of samples:  5000
```

Now let's visualize the results from our model.

In [259…

```
fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

ax.plot(tp_pred, yp_pred, label='Predicted Value')
ax.plot(ts.iloc[0:n_s], ys.iloc[0:n_s], label='Global Active Power data')

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```

Global Active Power Prediction



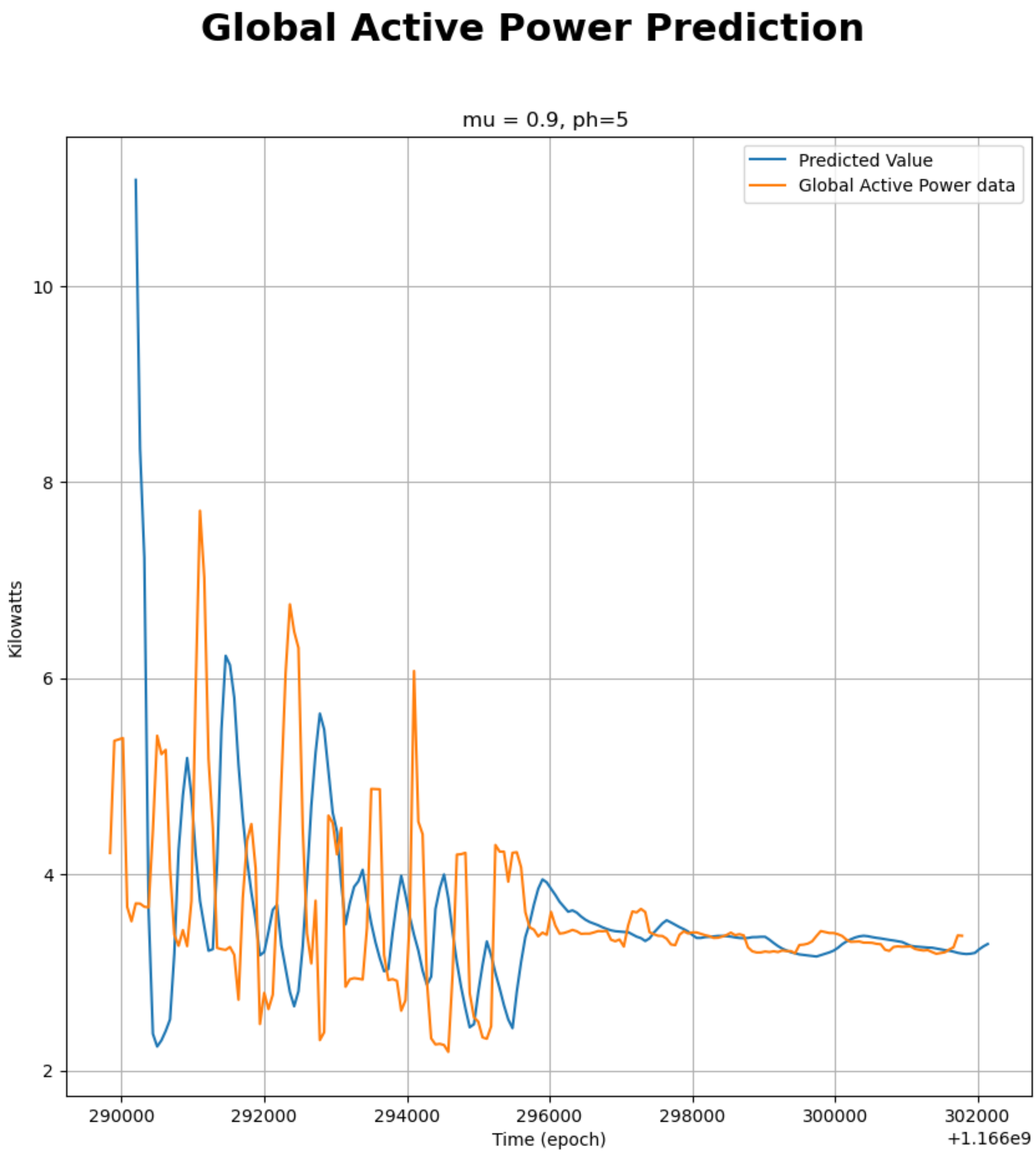
It's difficult to tell how the model is performing from this plot.

TODO: Modify the code above to visualize the first and last 200 datapoints/predictions (can be in separate charts) and compute the MSE for our predictions.

In [260…

```
#Plot first 200 data points/predictions

fig, ax = plt.subplots(figsize=(10, 10))
```

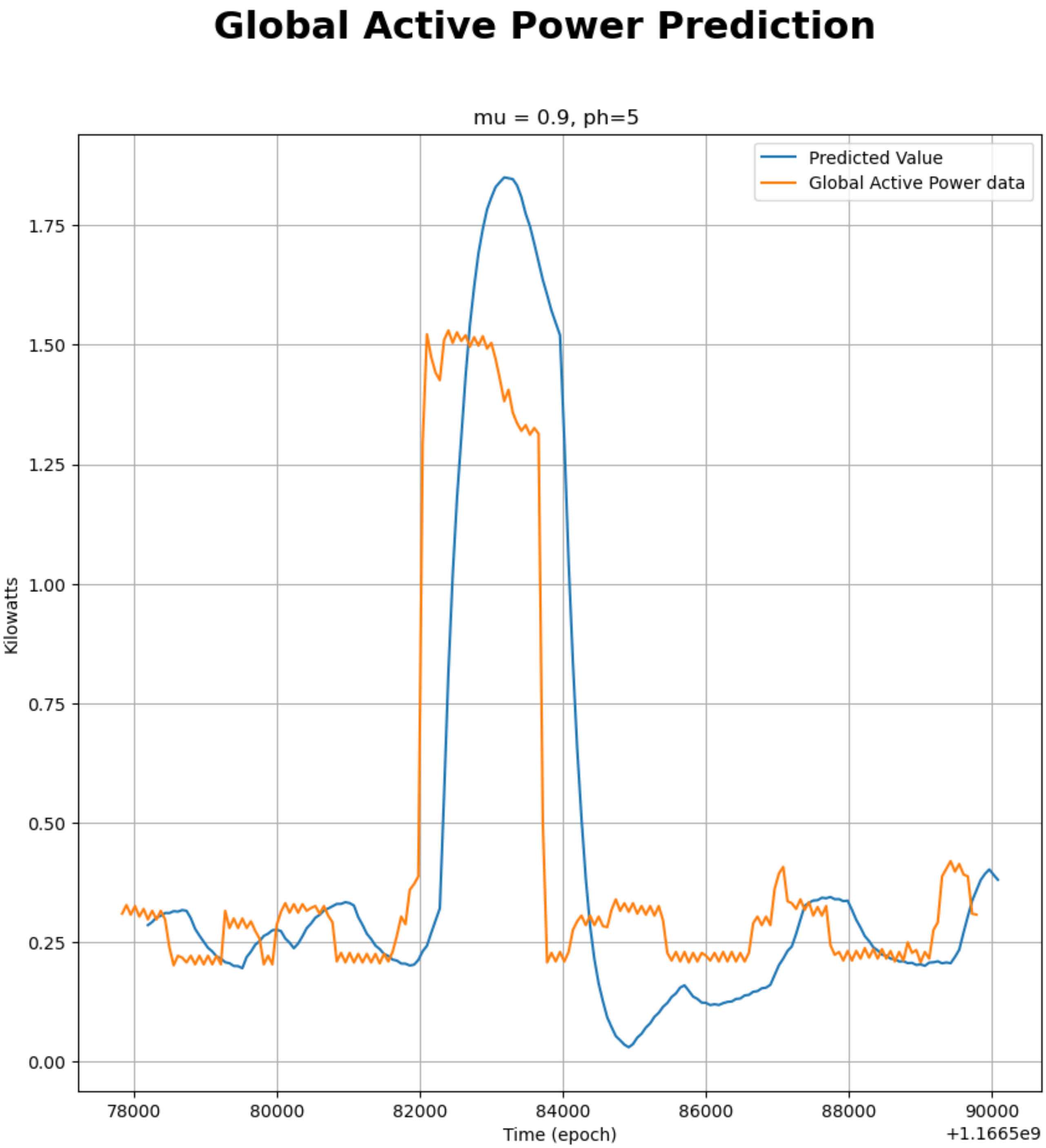
```
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

# Plot only the first 200 points for predicted values and actual data
ax.plot(tp_pred[:200], yp_pred[:200], label='Predicted Value')
ax.plot(ts.iloc[0:200], ys.iloc[0:200], label='Global Active Power data')

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```



In [261…

```
#Plot last 200 data points/predictions

fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

# Plot only the specified range for predicted values and actual data
# Specify the exact indices for the range 4800 to 5000
ax.plot(tp_pred[4800:5000], yp_pred[4800:5000], label='Predicted Value')
```

```
ax.plot(ts.iloc[4800:5000], ys.iloc[4800:5000], label='Global Active Power c

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```



In [262…

```
# Convert ph_index to integer to use in indexing
```

```
ph_index = int(ph_index)

end_index = ph_index + 5000
# Adjust end_index to not exceed the length of data arrays
end_index = min(end_index, len(ys['Global_active_power']), len(yp_pred))

# Calculate MSE using adjusted indices
mse_overall = mean_squared_error(ys['Global_active_power'][ph_index:end_inde
print("Overall MSE from ph_index to end_index:  ", mse_overall)

# Calculate MSE for the first 200 data points
mse_first_200 = mean_squared_error(ys.iloc[:200], yp_pred[:200])
print("MSE for the first 200 data points:       ", mse_first_200)

# Calculate MSE for the last 200 data points
mse_last_200 = mean_squared_error(ys.iloc[-200:], yp_pred[-200:])
print("MSE for the last 200 data points:        ", mse_last_200)
```

```
Overall MSE from ph_index to end_index:   0.18255777528889577
MSE for the first 200 data points:        0.576241217265719
MSE for the last 200 data points:         1.4339341162284445
```

Q: How did our model perform? What do you observe on the charts? Is there a difference between the early and the late predictions? What does the MSE tell you?

A:

First 200 Predictions:

- Initially High Predictions: Predictions start off higher as the model is still adapting to the scale and trends of the actual Global Active Power (GAP) data.
- Learning Process: As more data is processed, the model begins to learn and adjust, leading to predictions that gradually converge closer to the actual GAP values. The lower MSE for these predictions suggests that the model is starting to provide better predictions towards the second half of this segment, reflecting effective initial learning.

Last 200 Predictions:

- Closer Alignment: Predictions and actual GAP data are much closer, indicating the model's improved tuning from processing more data.
- Smoothed Predictions Despite Spikes: Although the overall predictions are smoother, suggesting a well-tuned model, the MSE is higher in this segment. This might be due to spikes, missing data, or outliers within these 200 records, which could distort the error metric despite generally closer predictions.

The overall MSE of 0.18255777528889577 suggests that the predictions were quite accurate, indicating that the linear regression model performed well with the dataset. This low MSE value demonstrates that the model's predictions were close to the actual data points, affirming that linear regression was a suitable choice for this analysis.

TODO: Re-run the prediction code with mu = 1 and mu = 0.01. Use the cells below to produce charts for the first and last 200 points and to compute the MSE for each of these sets of predictions.

In [263…

```
#Re-run prediction code for mu = 1

# Time Series
ts = pd.DataFrame(data_set.Unix)
# Y Series - dependent variable
ys = pd.DataFrame(data_set.Global_active_power)

# Prediction horizon is set to 5 minutes
ph = 5
# ph/data resolution - Data resolution is the time interval between consecut
ph_index = ph / 1
# # Forgetting factor
mu = 1

#let's limit the number of samples in our model to 5000 just for speed
n_s = 5000

# Arrays to hold predicted values
tp_pred = np.zeros(n_s-1)
yp_pred = np.zeros(n_s-1)

# At every iteration of the for loop a new data sample is acquired
for i in range(2, n_s+1):# start out with 2 leading datapoints

    # Get x and y data "available" for our prediction

    # Convert Timestamps to Unix time (seconds since the epoch) and reshape
    ts_tmp = data_set['Unix'][:i]

    ts_tmp = np.array(ts_tmp).reshape(-1, 1)  # Proper reshaping for use in

    ys_tmp = data_set['Global_active_power'][:i]
    ns = len(ys_tmp)

    # Initialize weights with all values set to mu
    weights = np.ones(ns) * mu

    for k in range(ns):
        # adjust weights to be downweighted according to their timestep away
```

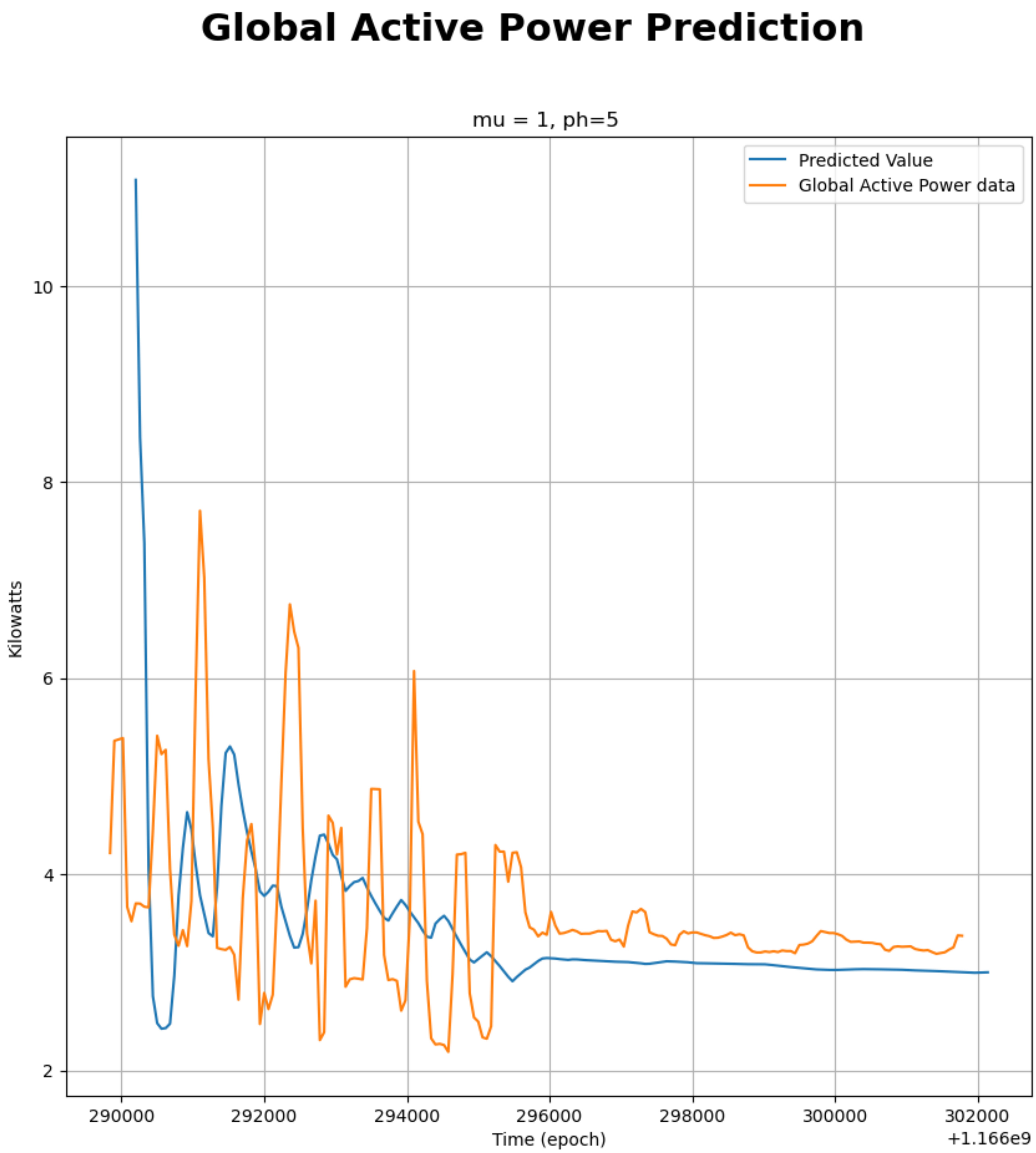
```
        weights[k] = mu ** (ns - k - 1)

        # Print weights for the first 5 records at specific intervals of 'i'
        # if i in [2000, 4000] and k in [2000, 4000]:
        #     print(f"At i={i}, k={k}, Weight = {weights[k]}")

        weights = np.flip(weights, 0)

    # perform linear regression on "available" data using the mu-adjusted we
    lm_tmp = LinearRegression()
    model_tmp = lm_tmp.fit(ts_tmp, ys_tmp, sample_weight=weights)

    # store model coefficients and intercepts to compute prediction
    m_tmp = model_tmp.coef_[0]  # slope
    q_tmp = model_tmp.intercept_  # intercept

    # use ph to make the model prediction according to the prediction time
    latest_time = ts_tmp[-1, 0]  # Get the latest timestamp
    tp = latest_time + ph * 60
    yp = m_tmp * tp + q_tmp  # Calculate the predicted power

    if i in [1000, 2000, 3000, 4000, 5000]:
        print(f"At i={i}, model coefficient (m_`tmp) = {m_tmp:.10f}, interce
        print(f"Model prediction at i={i} for l`atest time {latest_time}: {y

    tp_pred[i-2] = tp
    yp_pred[i-2] = yp

print("number of samples: ", ns)
```

```
At i=1000, model coefficient (m_`tmp) = -0.0000334346, intercept (q_tmp) = 3
8997.8856893903
Model prediction at i=1000 for l`atest time 1166349780: 1.4067058639
At i=2000, model coefficient (m_`tmp) = -0.0000116059, intercept (q_tmp) = 1
3538.8738146553
Model prediction at i=2000 for l`atest time 1166409780: 1.6242178464
At i=3000, model coefficient (m_`tmp) = -0.0000081679, intercept (q_tmp) = 9
528.9786107162
Model prediction at i=3000 for l`atest time 1166469780: 1.3260097783
At i=4000, model coefficient (m_`tmp) = -0.0000065735, intercept (q_tmp) = 7
669.3423644171
Model prediction at i=4000 for l`atest time 1166529780: 1.1115584820
At i=5000, model coefficient (m_`tmp) = -0.0000063637, intercept (q_tmp) = 7
424.5328857355
Model prediction at i=5000 for l`atest time 1166589780: 0.7618819092
number of samples:  5000
```

In [264…

```
#Plot first 200 data points/predictions for mu = 1

fig, ax = plt.subplots(figsize=(10, 10))
```

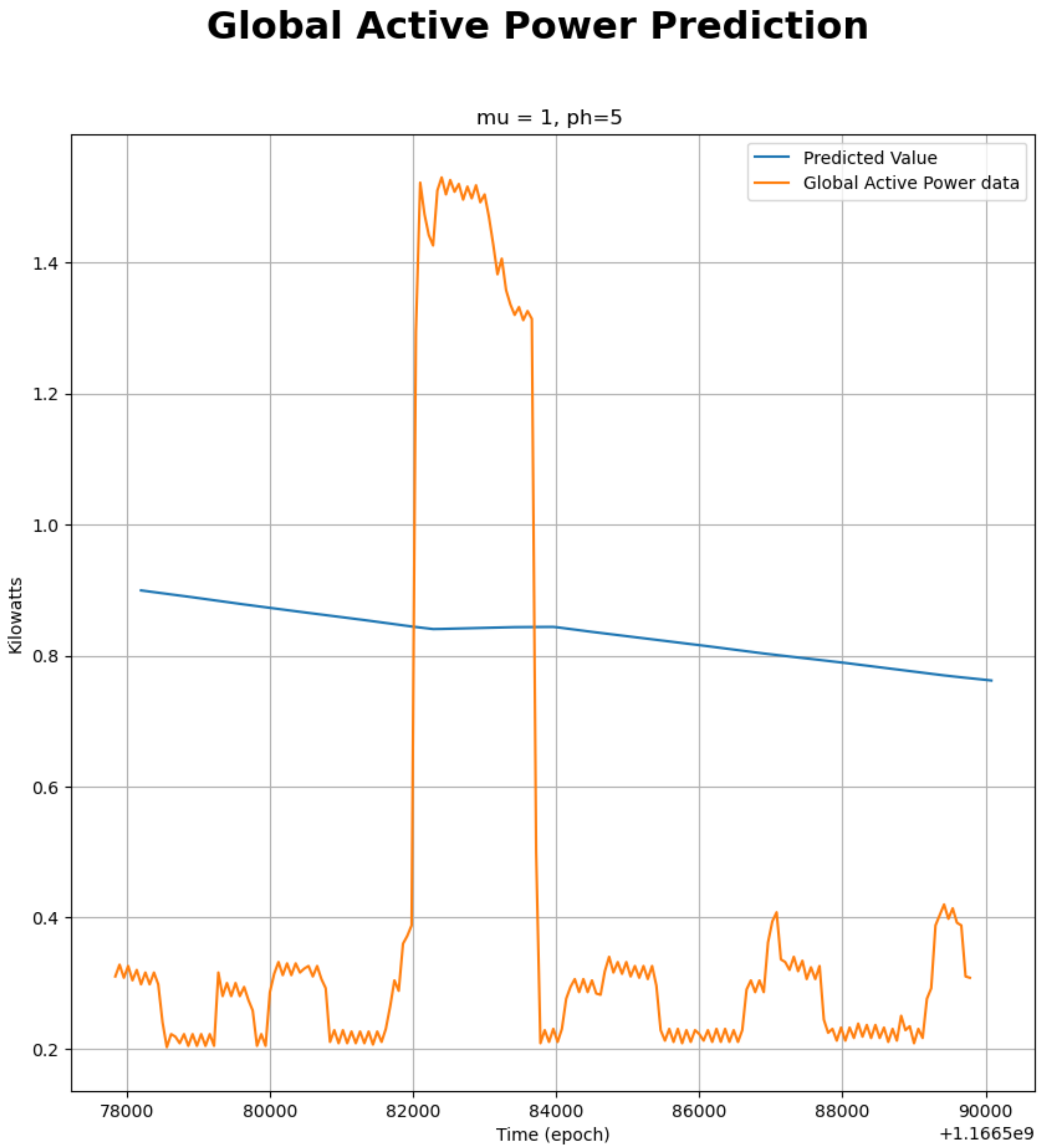
```
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bold
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

# Plot only the first 200 points for predicted values and actual data
ax.plot(tp_pred[:200], yp_pred[:200], label='Predicted Value')
ax.plot(ts.iloc[0:200], ys.iloc[0:200], label='Global Active Power data')

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```



In [265…

```
#Plot last 200 data points/predictions for mu = 1

fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

# Plot only the specified range for predicted values and actual data
# Specify the exact indices for the range 4800 to 5000
ax.plot(tp_pred[4800:5000], yp_pred[4800:5000], label='Predicted Value')
```

```
ax.plot(ts.iloc[4800:5000], ys.iloc[4800:5000], label='Global Active Power c

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```



In [266…

```
#Calculate MSE of predictions for mu = 1
```

```
# Convert ph_index to integer to use in indexing
ph_index = int(ph_index)

end_index = ph_index + 5000
# Adjust end_index to not exceed the length of data arrays
end_index = min(end_index, len(ys['Global_active_power']), len(yp_pred))

# Calculate MSE using adjusted indices
mse_overall = mean_squared_error(ys['Global_active_power'][ph_index:end_inde
print("Overall MSE from ph_index to end_index:  ", mse_overall)

# Calculate MSE for the first 200 data points
mse_first_200 = mean_squared_error(ys.iloc[:200], yp_pred[:200])
print("MSE for the first 200 data points:       ", mse_first_200)

# Calculate MSE for the last 200 data points
mse_last_200 = mean_squared_error(ys.iloc[-200:], yp_pred[-200:])
print("MSE for the last 200 data points:        ", mse_last_200)
```

```
Overall MSE from ph_index to end_index:   1.3812271202636723
MSE for the first 200 data points:        0.8258088082960455
MSE for the last 200 data points:         0.5699185016375299
```

In [267…

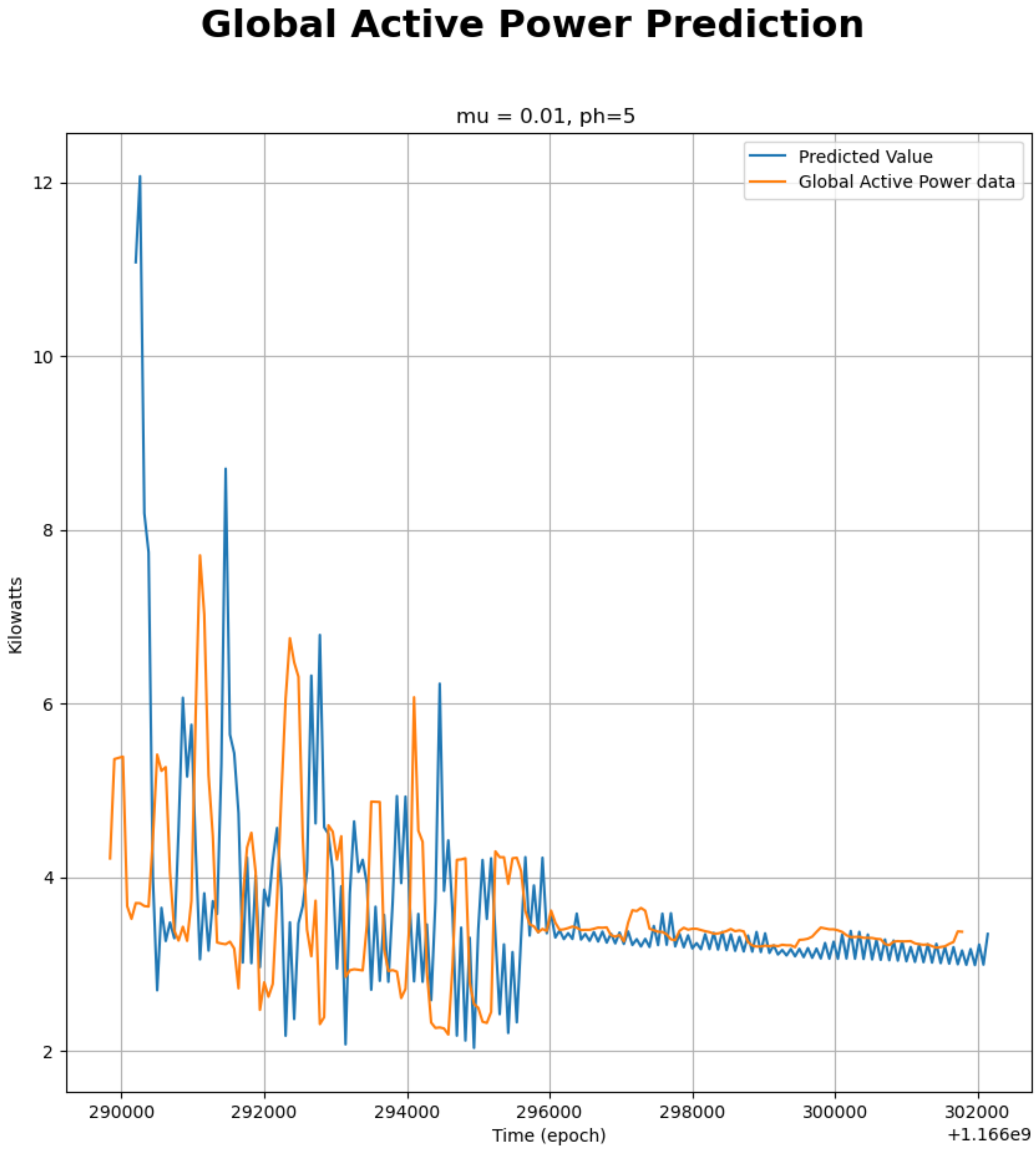
```
#Re-run prediction code for mu = 0.01

# Time Series
ts = pd.DataFrame(data_set.Unix)
# Y Series - dependent variable
ys = pd.DataFrame(data_set.Global_active_power)

# Prediction horizon is set to 5 minutes
ph = 5
# ph/data resolution - Data resolution is the time interval between consecut
ph_index = ph / 1
# # Forgetting factor
mu = 0.01

#let's limit the number of samples in our model to 5000 just for speed
n_s = 5000

# Arrays to hold predicted values
tp_pred = np.zeros(n_s-1)
yp_pred = np.zeros(n_s-1)

# At every iteration of the for loop a new data sample is acquired
for i in range(2, n_s+1):# start out with 2 leading datapoints

    # Get x and y data "available" for our prediction
```

```
    # Convert Timestamps to Unix time (seconds since the epoch) and reshape
    ts_tmp = data_set['Unix'][:i]

    ts_tmp = np.array(ts_tmp).reshape(-1, 1)  # Proper reshaping for use in

    ys_tmp = data_set['Global_active_power'][:i]
    ns = len(ys_tmp)

    # Initialize weights with all values set to mu
    weights = np.ones(ns) * mu

    for k in range(ns):
        # adjust weights to be downweighted according to their timestep away
        weights[k] = mu ** (ns - k - 1)

        # Print weights for the first 5 records at specific intervals of 'i'
        # if i in [2000, 4000] and k in [2000, 4000]:
        #     print(f"At i={i}, k={k}, Weight = {weights[k]}")

        weights = np.flip(weights, 0)

    # perform linear regression on "available" data using the mu-adjusted we
    lm_tmp = LinearRegression()
    model_tmp = lm_tmp.fit(ts_tmp, ys_tmp, sample_weight=weights)

    # store model coefficients and intercepts to compute prediction
    m_tmp = model_tmp.coef_[0]  # slope
    q_tmp = model_tmp.intercept_  # intercept

    # use ph to make the model prediction according to the prediction time
    latest_time = ts_tmp[-1, 0]  # Get the latest timestamp
    tp = latest_time + ph * 60
    yp = m_tmp * tp + q_tmp  # Calculate the predicted power

    if i in [1000, 2000, 3000, 4000, 5000]:
        print(f"At i={i}, model coefficient (m_`tmp) = {m_tmp:.10f}, interce
        print(f"Model prediction at i={i} for l`atest time {latest_time}: {y

    tp_pred[i-2] = tp
    yp_pred[i-2] = yp

print("number of samples: ", ns)
```

```
At i=1000, model coefficient (m_`tmp) = -0.0000425949, intercept (q_tmp) = 4
9681.7988425210
Model prediction at i=1000 for l`atest time 1166349780: 1.2002657708
At i=2000, model coefficient (m_`tmp) = -0.0000158610, intercept (q_tmp) = 1
8501.8231330780
Model prediction at i=2000 for l`atest time 1166409780: 1.4395095877
At i=3000, model coefficient (m_`tmp) = -0.0000104914, intercept (q_tmp) = 1
2239.0833062307
Model prediction at i=3000 for l`atest time 1166469780: 1.1844243199
At i=4000, model coefficient (m_`tmp) = -0.0000081000, intercept (q_tmp) = 9
449.8341587007
Model prediction at i=4000 for l`atest time 1166529780: 0.9836930412
At i=5000, model coefficient (m_`tmp) = -0.0000073709, intercept (q_tmp) = 8
599.4398397026
Model prediction at i=5000 for l`atest time 1166589780: 0.6574306357
number of samples:  5000
```

In [272…

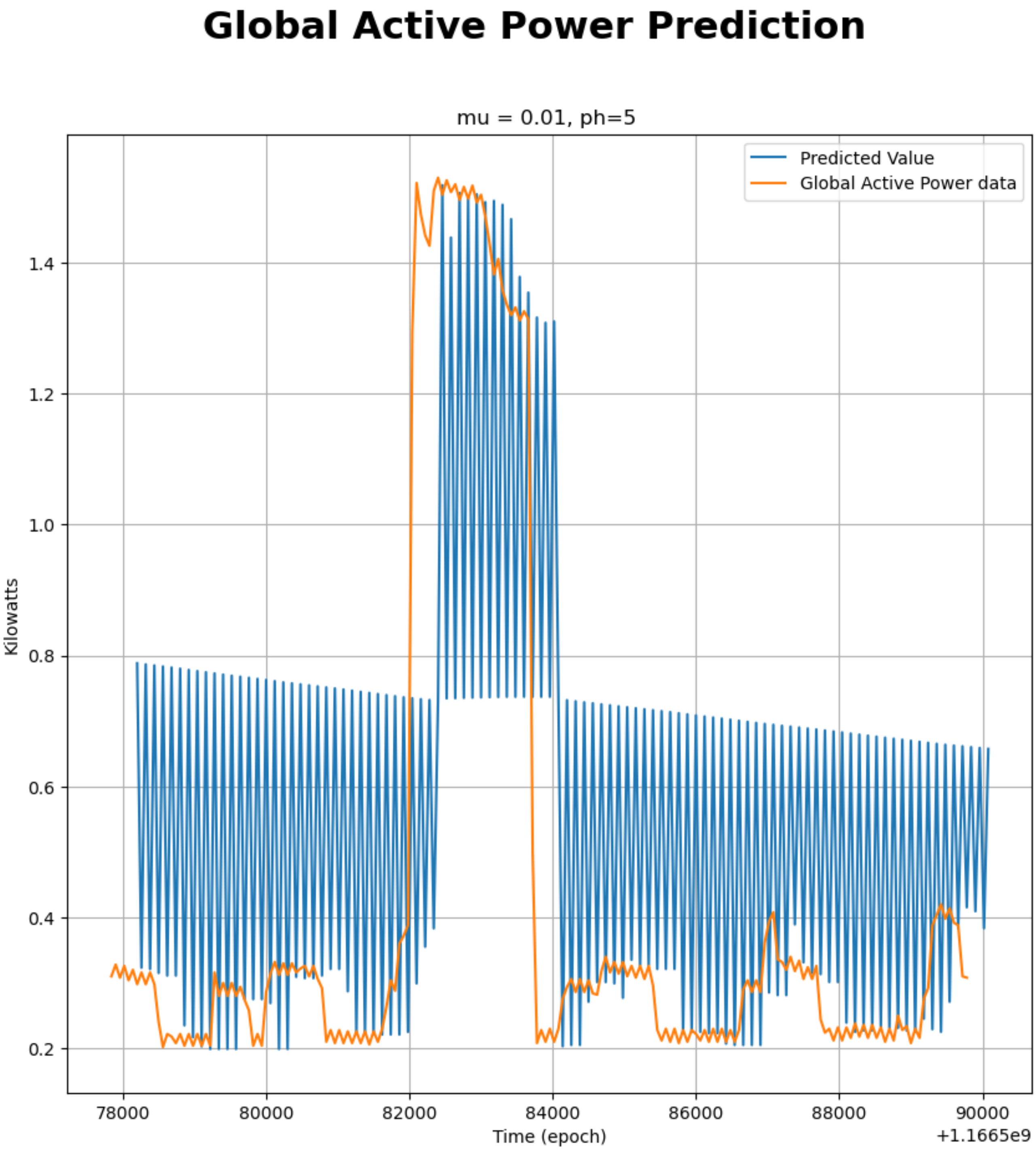
```
#Plot first 200 data points/predictions for mu = 0.01

fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

# Plot only the first 200 points for predicted values and actual data
ax.plot(tp_pred[:200], yp_pred[:200], label='Predicted Value')
ax.plot(ts.iloc[0:200], ys.iloc[0:200], label='Global Active Power data')

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```



In [273…

```
#Plot last 200 data points/predictions for mu = 0.01

fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title('mu = %g, ph=%g ' %(mu, ph))

# Plot only the specified range for predicted values and actual data
# Specify the exact indices for the range 4800 to 5000
ax.plot(tp_pred[4800:5000], yp_pred[4800:5000], label='Predicted Value')
```

```
ax.plot(ts.iloc[4800:5000], ys.iloc[4800:5000], label='Global Active Power c

ax.set_xlabel('Time (epoch)')
ax.set_ylabel('Kilowatts')
ax.legend()

plt.grid(True)
plt.show()
```



In [274…
```
#Calculate MSE of predictions for mu = 0.01
```

```
# Convert ph_index to integer to use in indexing
ph_index = int(ph_index)

end_index = ph_index + 5000
# Adjust end_index to not exceed the length of data arrays
end_index = min(end_index, len(ys['Global_active_power']), len(yp_pred))

# Calculate MSE using adjusted indices
mse_overall = mean_squared_error(ys['Global_active_power'][ph_index:end_inde
print("Overall MSE from ph_index to end_index:  ", mse_overall)

# Calculate MSE for the first 200 data points
mse_first_200 = mean_squared_error(ys.iloc[:200], yp_pred[:200])
print("MSE for the first 200 data points:       ", mse_first_200)

# Calculate MSE for the last 200 data points
mse_last_200 = mean_squared_error(ys.iloc[-200:], yp_pred[-200:])
print("MSE for the last 200 data points:        ", mse_last_200)
```

```
Overall MSE from ph_index to end_index:   0.7308118539078153
MSE for the first 200 data points:        0.7108107129675991
MSE for the last 200 data points:         1.0407316150812502
```

Q: How did our mu = 1 model perform? What do you observe on the charts? Is there a difference between the early and the late predictions? What does the MSE tell you?

A:

- The model's Mean Squared Error (MSE) indicates that performance is generally lower with MU set to 1.0 compared to an MU of 0.9. This reflects a less effective handling of the data variability over the entire dataset.
- The predictions in the initial 200 data points show a gap between the predicted and actual Global Active Power (GAP) values. This suggests that the model, highly reliant on the most recent data, fails to adapt quickly to changes, leading to inaccuracies early in the prediction phase.
- In the last 200 points, the predictions continue to heavily reflect historical data trends. The persistence of this reliance is evident as the predicted values diverge noticeably from the actual GAP data, ignoring recent fluctuations or anomalies.
- The MSE is notably higher for the overall predictions when MU is set to 1.0, signaling a poorer fit across the dataset. However, it slightly improves in the last 200 predictions, indicating some stabilization in the model's output as it continues processing more data.

Q: How did our mu = 0.01 model perform? What do you observe on the charts? Is

there a difference between the early and the late predictions? What does the MSE tell you?

A:

- Improved MSE: The Mean Squared Error (MSE) for MU = 0.01 shows that this setting provides a generally better fit than an MU of 1.0, suggesting that a higher emphasis on recent data improves prediction accuracy.
- The predictions in the initial 200 data points show a rapid adjustment to the data, aligning predicted values closely with the actual Global Active Power (GAP) after a brief period of initial input processing.
- In the last 200 points, the model shows an increased ability to respond to fluctuations, effectively capturing spikes and sudden changes in the GAP.
- The MSE is lower for the overall predictions when MU is set to 0.01, signaling a better fit across the dataset.

Q: Which of these three models is the best? How do you know? Why does this make sense based on the mu parameter used?

A:

- Optimal MU Setting: Among the tested MU values (0.01, 0.9, and 1.0), the model with MU = 0.9 demonstrated the best performance. The corresponding charts and Mean Squared Error (MSE) values indicate that this setting strikes an effective balance between relying on historical data and adapting to recent data.
- Balanced Data Consideration: The superior performance of MU = 0.9 suggests that neither a complete reliance on historical data nor an exclusive focus on recent changes is ideal for accurate predictions. Instead, a balanced approach that incorporates both elements can lead to better prediction accuracy and model stability.

Q: What could we do to improve our model and/or make it more realistic and useful?

A:

- Data Cleaning and Analysis: Initial observations suggest that the presence of spikes and missing records could be influencing the prediction accuracy. A thorough cleaning of the data set to correct or remove erroneous entries and fill gaps where feasible will likely improve model performance.
- Understanding Data Dynamics: Since the data represents power usage recorded at one-minute intervals, it is crucial to understand the context behind fluctuations.

Identifying patterns related to time-specific increases—such as higher energy consumption during certain hours—can inform more targeted predictive models.

- Adjustment of Time Intervals: Expanding the granularity of data from one minute to longer intervals, like hourly data, might reduce noise and highlight more substantial trends, aiding in smoother and potentially more accurate predictions. This change could help mitigate the impact of minute-to-minute variability and focus on longer-term trends.
- Further Experiments with MU Values: Given the promising results with an MU of 0.9, exploring adjacent values such as 0.85 or 0.95 might help refine the balance between sensitivity to recent data and historical trends. These experiments can reveal the optimal MU setting for the specific characteristics of this dataset.

TODO: Add voltage data as a second variable to our model and re-run the prediction code. Then visualize the first and last 200 points and compute the MSE

In [319…

```
# add voltage to the x-variables in our dataset

# Time Series
ts = pd.DataFrame({
    'Unix': data_set['Unix'],
    'Voltage': data_set['Voltage']
})
# Y Series - dependent variable
ys = pd.DataFrame(data_set['Global_active_power'])

# Prediction horizon is set to 5 minutes
ph = 5
# # Forgetting factor
mu = 0.9

#let's limit the number of samples in our model to 5000 just for speed
n_s = 5000

# Arrays to hold predicted values
tp_pred = np.zeros(n_s-1)
yp_pred = np.zeros(n_s-1)

print(ts.head())
```

```
         Unix   Voltage
0  1166290030 228.81616
1  1166289848 234.18391
2  1166290068 239.30812
3  1166290008 225.17943
4  1166289987 239.40875
```

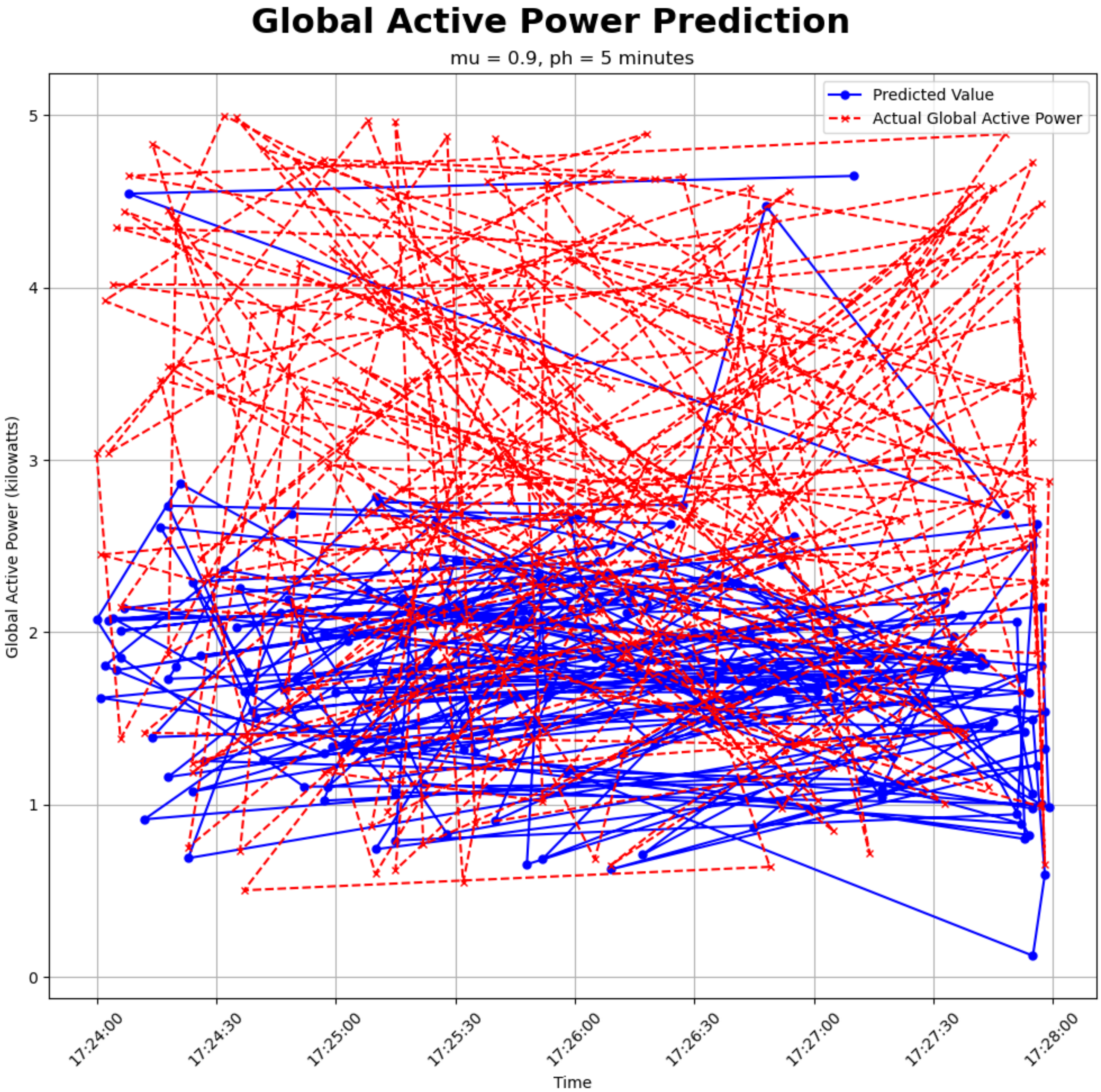
In [320…

```
import warnings
warnings.filterwarnings(action='ignore', category=UserWarning, message='.*X

# Loop through data
for i in range(2, n_s+1):

    # Select data up to current index
    ts_tmp = ts.iloc[:i]
    ys_tmp = ys.iloc[:i].values.ravel()  # Flatten array to 1D

    # print(f"At iteration {i}, TS data: \n{ts_tmp}")
    # print(f"At iteration {i}, YS data: \n{ys_tmp}\n")

    # Initialize weights with the basic mu value
    weights = np.ones(len(ys_tmp)) * mu

    # Adjust weights based on the forgetting factor
    for k in range(len(ys_tmp)):
        weights[k] = mu ** (len(ys_tmp) - k - 1)

        # Print weights for the first 5 records at specific intervals of 'i'
        # if i in [2000, 4000] and k in [2000, 4000]:
        #     print(f"At i={i}, k={k}, Weight = {weights[k]}")

    weights = np.flip(weights, 0)

    # print(f"At iteration {i}, weights: {weights}")

    # Fit linear regression with sample weights
    lm_tmp = LinearRegression()
    model_tmp = lm_tmp.fit(ts_tmp, ys_tmp, sample_weight=weights)

    # Predict the next value
    latest_time = ts_tmp.iloc[-1, 0]  # Latest time from 'Unix'
    latest_voltage = ts_tmp.iloc[-1, 1]  # Latest voltage
    tp = latest_time + ph * 60  # Add ph minutes to latest time
    yp = lm_tmp.predict([[tp, latest_voltage]])[0]  # Predict using both tim

    # Store predictions
    tp_pred[i-2] = tp
    yp_pred[i-2] = yp

    # Optional: print outputs at milestones
    if i % 1000 == 0:
        print(f"At i={i}, coefficients={lm_tmp.coef_}, intercept={lm_tmp.int
        print(f"Prediction at i={i} for time {tp}: {yp}")

print("Predictions and model fitting completed.")
```

```
At i=1000, coefficients=[-0.00578576  0.04996498], intercept=6747866.9081948
07
Prediction at i=1000 for time 1166290341: 1.536048011854291
At i=2000, coefficients=[-0.00578576  0.04996498], intercept=6747866.9081948
07
Prediction at i=2000 for time 1166290216: 1.6114943362772465
At i=3000, coefficients=[-0.00578576  0.04996498], intercept=6747866.9081948
08
Prediction at i=3000 for time 1166290263: 1.5860263612121344
At i=4000, coefficients=[-0.00578576  0.04996498], intercept=6747866.9081948
07
Prediction at i=4000 for time 1166290156: 2.4952686531469226
At i=5000, coefficients=[-0.00578576  0.04996498], intercept=6747866.9081948
06
Prediction at i=5000 for time 1166290208: 2.444668048992753
Predictions and model fitting completed.
```

In [ ]:

```
# Plot first 200 data points/predictions for the expanded dataset

# Convert Unix timestamps to a readable format, if necessary, or ensure they
ts['Readable_Time'] = pd.to_datetime(ts['Unix'], unit='s')

fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title(f'mu = {mu}, ph = {ph} minutes')

# Define the range for plotting directly in the indexing
start_index = 0
end_index = 200

ax.plot(ts['Readable_Time'].iloc[start_index:end_index], yp_pred[start_index
ax.plot(ts['Readable_Time'].iloc[start_index:end_index], ys.iloc[start_index

ax.set_xlabel('Time')
ax.set_ylabel('Global Active Power (kilowatts)')
ax.legend()

plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

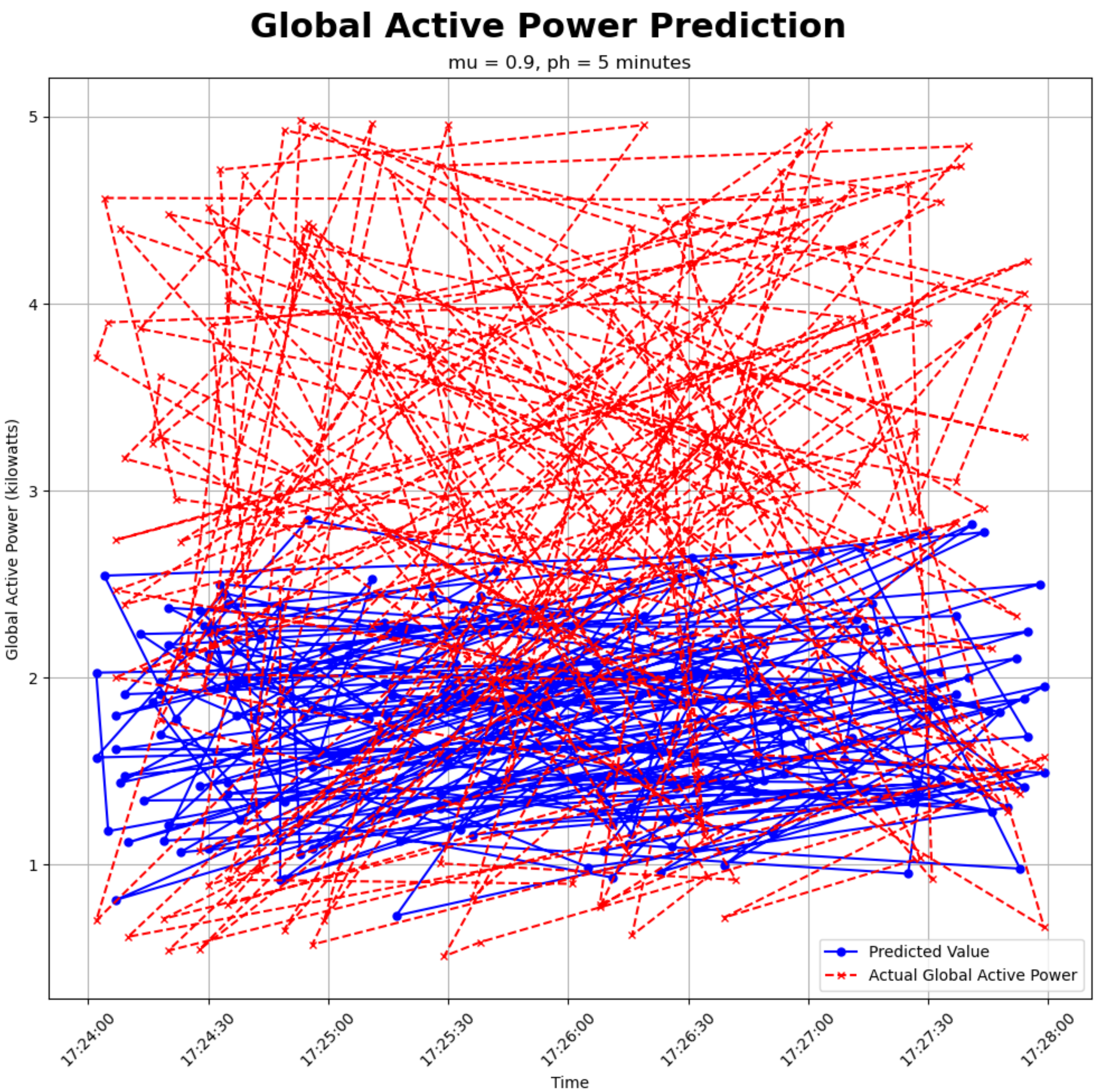


In [350…

```
#Plot last 200 data points/predictions for the expanded data

print(n_s)

fig, ax = plt.subplots(figsize=(10, 10))
fig.suptitle('Global Active Power Prediction', fontsize=22, fontweight='bolc
ax.set_title(f'mu = {mu}, ph = {ph} minutes')

# Define the range for plotting directly in the indexing
start_index = 4800
end_index = 5000 - 1

ax.plot(ts['Readable_Time'].iloc[start_index:end_index], yp_pred[start_index
```

```
ax.plot(ts['Readable_Time'].iloc[start_index:end_index], ys.iloc[start_index

ax.set_xlabel('Time')
ax.set_ylabel('Global Active Power (kilowatts)')
ax.legend()

plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

5000



In [353…

```
#Calculate MSE of predictions for the expanded data
```

```
# Convert ph_index to integer to use in indexing
ph_index = int(ph_index)

end_index = ph_index + 5000
# Adjust end_index to not exceed the length of data arrays
end_index = min(end_index, len(ys['Global_active_power']), len(yp_pred))

# Calculate MSE using adjusted indices
mse_overall = mean_squared_error(ys['Global_active_power'][ph_index:end_inde
print("Overall MSE from ph_index to end_index:  ", mse_overall)

# Calculate MSE for the first 200 data points
mse_first_200 = mean_squared_error(ys.iloc[:200], yp_pred[:200])
print("MSE for the first 200 data points:       ", mse_first_200)

# Calculate MSE for the last 200 data points
mse_last_200 = mean_squared_error(ys.iloc[-200:], yp_pred[-200:])
print("MSE for the last 200 data points:        ", mse_last_200)
```

```
Overall MSE from ph_index to end_index:   2.919662979208407
MSE for the first 200 data points:        2.9442307979200852
MSE for the last 200 data points:         3.1310257859056985
```

Q: How did the model performed when you added the voltage data? How does it compare to the models without it?

A:

- Overall MSE: A value of 2.92 suggests that on average, the square of the error between the predicted and actual global active power is quite high. This indicates poor model performance across the entire dataset.
- The analysis of mean squared error (MSE) and predictive accuracy clearly indicates that the model performs better without the inclusion of voltage as a feature. The MSE values and the visual comparison of prediction charts show that the model without voltage not only yielded lower error rates but also displayed better alignment with the actual global active power data. This suggests that adding voltage to the model may have introduced noise or complexity that detracted from its predictive capability.
- Given these observations, it is advisable to reconsider the utility of including voltage as a predictive feature. This involves evaluating the feature's relevance and contribution to the model and assessing whether linear regression is the appropriate method for handling this type of data complexity. It may be necessary to explore alternative modeling approaches or adjust the current model's complexity to better accommodate the additional feature if it is deemed essential for other reasons.

There are lots of other ways that we could try to improve our model while still using linear regression.

TODO: Choose one alternative model and re-run the prediction code. Some ideas include:

- Use a moving average as the response variable
- Make your prediction based on the time of day instead of as a continuous time series
- Use a moving window to limit your predictions instead of using a mu factor

Q: Describe your alternative model and why it might improve your model

A:

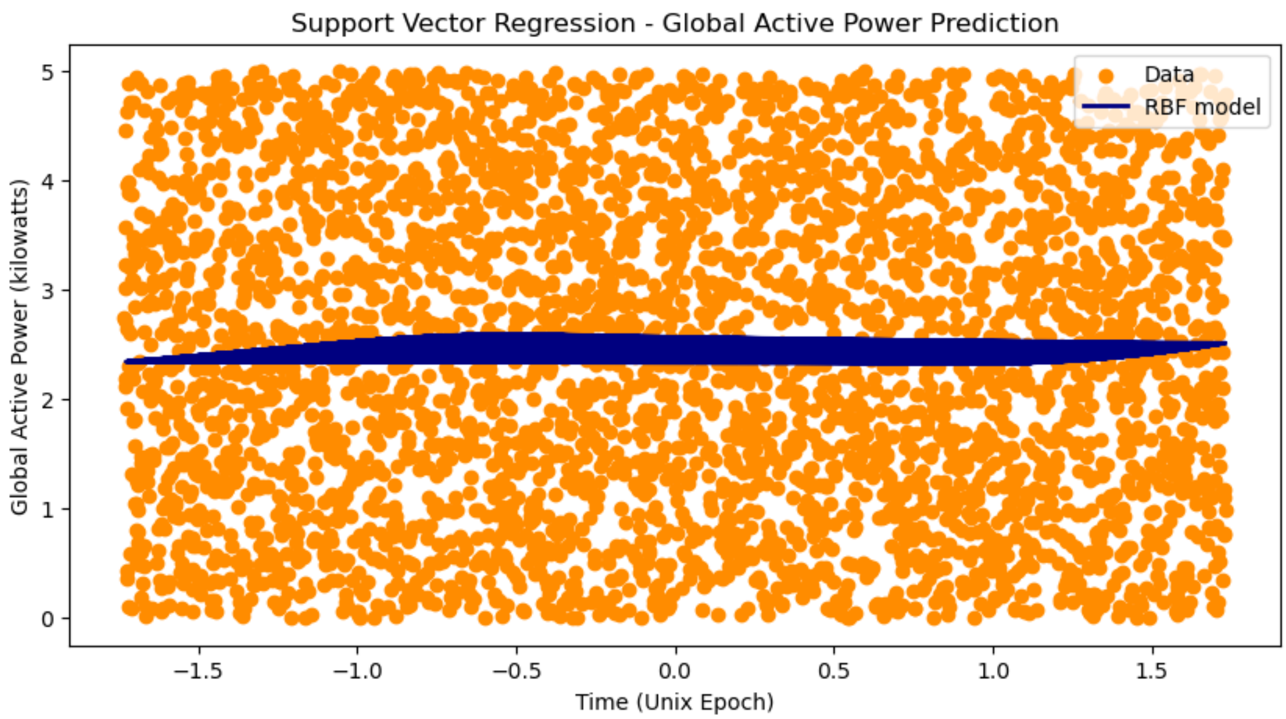
- Exploring Alternative Models: Employing various models can enhance prediction accuracy through fitting and training nuances. For instance, using Support Vector Regression (SVR) improved the Mean Squared Error (MSE), but the dispersed plot indicates that this model may not yield the most reliable predictions.
- Model Performance Variability: Even when some models, like Random Forest Regression, provide a reasonable MSE, their plot representation may not accurately reflect the data, suggesting limitations in their predictive capabilities.

In [419…

```
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Prepare the data
X = data_set[['Unix']].values  # Using only Unix timestamps for now
y = data_set['Global_active_power'].values

# Standardize the features
scaler_x = StandardScaler()
X_scaled = scaler_x.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0

# Create the SVR model
svr = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)

# Fit the model
svr.fit(X_train, y_train)
```

```
# Make predictions
y_pred = svr.predict(X_test)

# Calculate MSE
mse = mean_squared_error(y_test, y_pred)
print("MSE:", mse)

# Plotting the results
plt.figure(figsize=(10, 5))
plt.scatter(X_train, y_train, color='darkorange', label='Data')
plt.plot(X_test, y_pred, color='navy', lw=2, label='RBF model')
plt.xlabel('Time (Unix Epoch)')
plt.ylabel('Global Active Power (kilowatts)')
plt.title('Support Vector Regression - Global Active Power Prediction')
plt.legend()
plt.show()
```

```
MSE: 2.1788483517922552
```

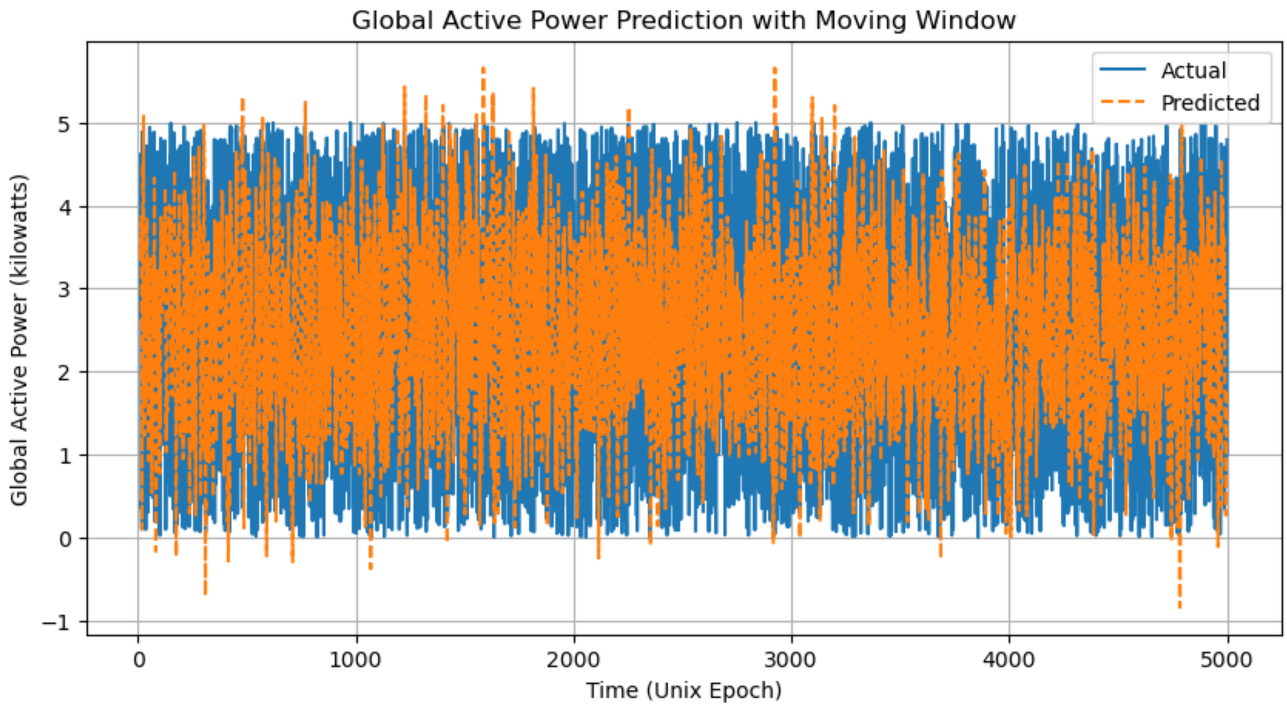


In [398…

```
#re-run the prediction code here

warnings.filterwarnings(action='ignore', category=DeprecationWarning, messag

window_size = 10  # number of observations to consider for each prediction
n_s = 5000  # total number of observations to simulate

# Sample data generation (replace with your actual data loading)
np.random.seed(0)
data_set = pd.DataFrame({
    'Unix': np.linspace(1, n_s, n_s),
```

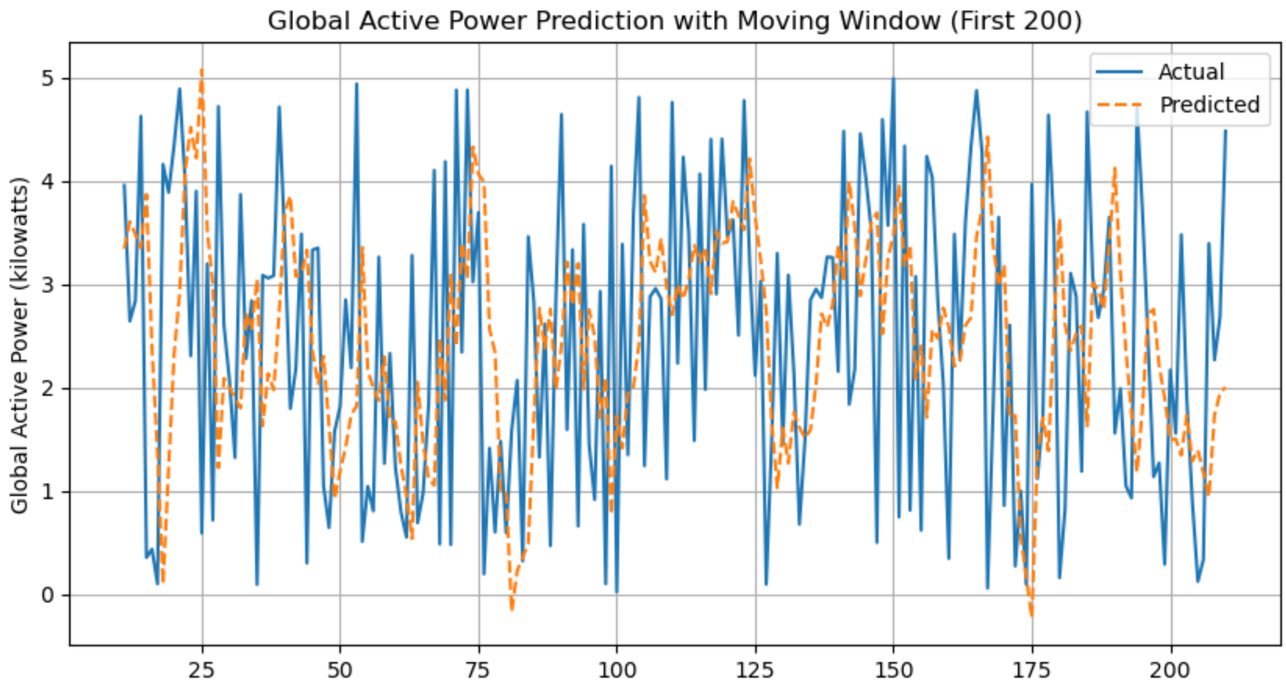
```
    'Global_active_power': np.random.rand(n_s) * 5  # Random power usage val
})

# Prepare the arrays for storing predictions
yp_pred = np.zeros(n_s - window_size)

# Prediction model
model = LinearRegression()

# Perform rolling window predictions
for i in range(window_size, n_s):
    # Select data from the window
    start_index = i - window_size
    end_index = i
    ts_tmp = data_set['Unix'][start_index:end_index].values.reshape(-1, 1)
    ys_tmp = data_set['Global_active_power'][start_index:end_index]

    # Fit model
    model.fit(ts_tmp, ys_tmp)

    # Predict the next point
    next_timestamp = data_set['Unix'][i].reshape(-1, 1)
    yp_pred[i - window_size] = model.predict(next_timestamp)

# Calculate Mean Squared Error
mse_value = mean_squared_error(data_set['Global_active_power'][window_size:r
print(f"The MSE for the predictions is: {mse_value}")

# Plotting the results for visualization
plt.figure(figsize=(10, 5))
plt.plot(data_set['Unix'][window_size:n_s], data_set['Global_active_power'][
plt.plot(data_set['Unix'][window_size:n_s], yp_pred, label='Predicted', line
plt.title('Global Active Power Prediction with Moving Window')
plt.xlabel('Time (Unix Epoch)')
plt.ylabel('Global Active Power (kilowatts)')
plt.legend()
plt.grid(True)
plt.show()
```

```
The MSE for the predictions is: 3.062768296804339
```



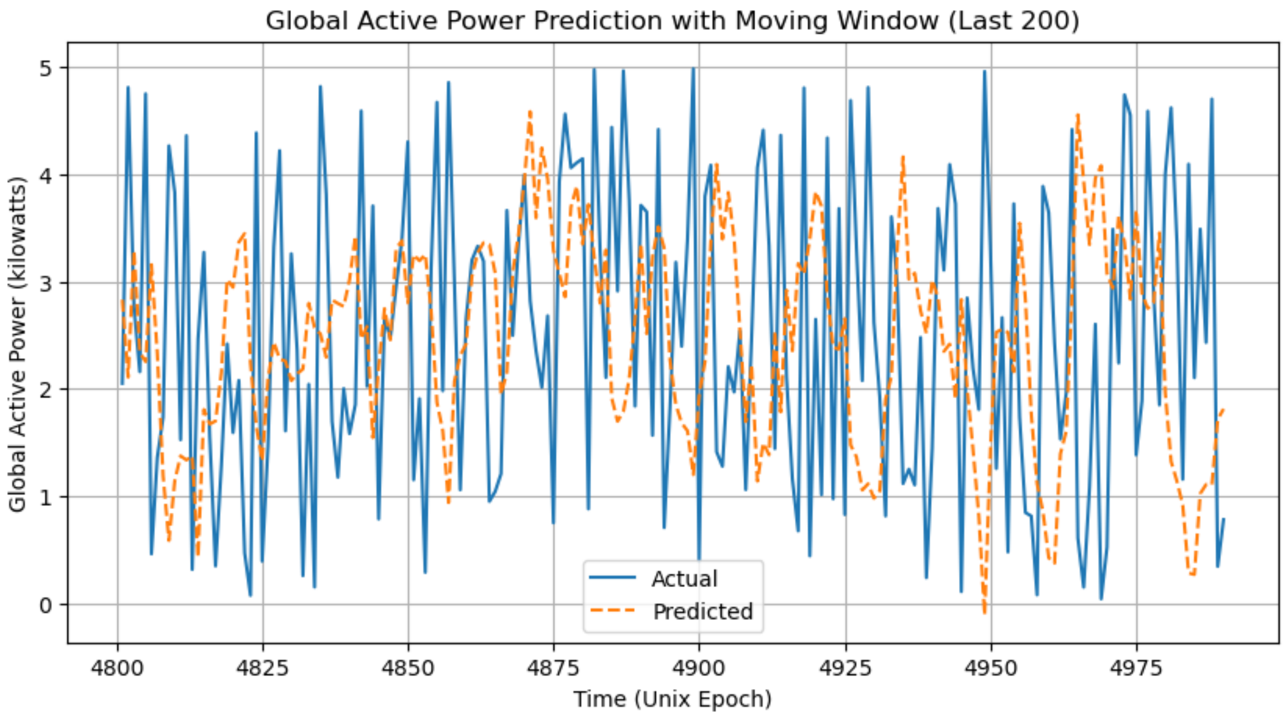
In [ ]:

```
#Plot first 200 data points/predictions for alternative model

plt.figure(figsize=(10, 5))

# Plot first 200 data points
plt.plot(data_set['Unix'][window_size:window_size+200], data_set['Global_act
plt.plot(data_set['Unix'][window_size:window_size+200], yp_pred[:200], label
plt.title('Global Active Power Prediction with Moving Window (First 200)')
plt.xlabel('Time (Unix Epoch)')
plt.ylabel('Global Active Power (kilowatts)')
plt.legend()
plt.grid(True)
```



In [396…

```
#Plot last 200 data points/predictions for alternative model

plt.figure(figsize=(10, 5))

# Define the range for plotting directly in the indexing
start_index = 4800
end_index = 4990

# Plot last 200 data points
plt.plot(data_set['Unix'][start_index:end_index], data_set['Global_active_po
plt.plot(data_set['Unix'][start_index:end_index], yp_pred[start_index:end_in

plt.title('Global Active Power Prediction with Moving Window (Last 200)')
plt.xlabel('Time (Unix Epoch)')
plt.ylabel('Global Active Power (kilowatts)')
plt.legend()
plt.grid(True)
plt.show()
```



In [418…

```
#Calculate MSE of predictions for alternative model

# Calculate MSE using adjusted indices
mse_value = mean_squared_error(data_set['Global_active_power'][window_size:r
print("The MSE for the predictions is:      ", mse_value)

# Calculate MSE for the first 200 data points
mse_first_200 = mean_squared_error(data_set['Global_active_power'][window_si
print("MSE for the first 200 data points:   ", mse_first_200)

# Calculate MSE for the last 200 data points
if len(yp_pred) >= n_s - 200:
    last_200_start_index = len(yp_pred) - 200
else:
    last_200_start_index = max(0, len(yp_pred) - 200)
mse_last_200 = mean_squared_error(data_set['Global_active_power'][n_s-200:n_
print("MSE for the last 200 data points:    ", mse_last_200)
```

```
The MSE for the predictions is:       3.062768296804339
MSE for the first 200 data points:    2.8941888947538836
MSE for the last 200 data points:     2.7769091055008244
```

Q: Did your alternative model improve on our previous results? What else could you do to improve the model while still using linear regression?

A:

- Alternative Model Performance: While employing the moving window approach in the alternative model showed that smaller window sizes could closely track fluctuations, resulting in visually more accurate predictions on plots, this did not correspondingly lower the Mean Squared Error (MSE). This suggests a trade-off where higher responsiveness to data spikes increases plot accuracy but not necessarily error metrics.

- Improving Model Accuracy: Enhancing the model's performance, even when sticking with linear regression, could involve more meticulous data cleaning to remove noise and anomalies that contribute to spikes. Understanding the underlying causes of fluctuations within the data can lead to more accurate modeling and prediction, as clean and well-understood data typically yields better forecasting outcomes.

It's worth noting that the results we're getting int his assignment are based on a pretty short predictive horizon of 5 minutes. If we were to increase our predictive horizon, our results would likely be worse and there would be more room for optimizing and improving the predictions of our model.