



You:

write code in pandas that can poll operations technology sensor on one minute intervals and robustly forecast the sensor value one hour into the future continuously. App can cache data for best forecasting methods if needed but needs to have code built into limiting the amount of cached data so memory does not fill up. also continuously provide ability to print the forecasted error to actual sensor value



Here's a basic implementation using pandas and the statsmodels library to create a simple ARIMA model for time series forecasting. The implementation polls data at one-minute intervals and forecasts the sensor value one hour into the future. It also provides the ability to print the forecasted error when the actual value becomes available:

1. First, ensure you have the necessary libraries installed:

```
bash
pip install pandas statsmodels
```

2. Then, you can use the following code:

```
python
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
import time

class SensorDataForecast:

    def __init__(self):
        self.data_cache = pd.Series()
        self.CACHE_LIMIT = 1440 # Let's keep 24 hours of data as an example, adjust as needed

    def poll_sensor_data(self):
        # Simulate reading sensor data; replace with actual data fetching code
        return np.random.randn()

    def fetch_and_store_data(self):
        new_data = self.poll_sensor_data()
        current_time = pd.Timestamp.now()
        self.data_cache[current_time] = new_data

        # Limit cache size
        if len(self.data_cache) > self.CACHE_LIMIT:
            self.data_cache = self.data_cache[-self.CACHE_LIMIT:]

    def forecast_next_hour(self):
        # Here, a simple ARIMA model is used; you might want to choose a better-suited model depending on the data nature
        model = ARIMA(self.data_cache, order=(5,1,0)) # These parameters can be fine-tuned
        model_fit = model.fit(dispatch=0)
        forecast = model_fit.forecast(steps=60) # Forecasting 60 minutes into the future
        return forecast

    def calculate_forecast_error(self, actual, forecast):
        return actual - forecast

    def run_forecasting_cycle(self):
        while True:
            self.fetch_and_store_data()
            if len(self.data_cache) > 60: # Wait until we have at least 1 hour of data to start forecasting
                forecasted_values = self.forecast_next_hour()

                # Assuming one forecasted value for each minute, get the last forecasted value
                last_forecasted_value = forecasted_values[-1]

                # Wait for an hour to get the actual sensor value at that time
                time.sleep(3600)

                actual_value = self.poll_sensor_data()
                error = self.calculate_forecast_error(actual_value, last_forecasted_value)
                print(f"Forecasted value: {last_forecasted_value}, Actual value: {actual_value}, Error: {error}")

            time.sleep(60) # Poll every minute
```

```
# Execute
forecast_app = SensorDataForecast()
forecast_app.run_forecasting_cycle()
```

A few things to note:

- This code uses a simple ARIMA model. Depending on the nature of your sensor data, you might need to use a more complex or different forecasting model.
- I've simulated polling the sensor with the `poll_sensor_data` function that returns a random value. You should replace this function's content with the actual logic to poll the sensor's data.
- The code waits an hour after forecasting to fetch the actual sensor value and calculate the error. This might not be the most efficient approach; in a real-world scenario, you might want to use some asynchronous approach or threading to handle this more elegantly.
- The cached data size is limited to the last 24 hours (1440 minutes) in this example, but you can adjust the `CACHE_LIMIT` as required.



You:

can you provide an alternate method that doesn't need to be tuned?



ChatGPT:

Certainly! If you're looking for a method that doesn't require manual tuning like the ARIMA model, one popular approach is to use Prophet by Facebook. Prophet is designed to work well out of the box without requiring domain-specific knowledge.

Here's how you can use Prophet for your forecasting:

1. First, ensure you have the necessary libraries installed:

```
bash
pip install pandas fbprophet
```

2. Implement the forecasting:

```
python
import pandas as pd
from fbprophet import Prophet
import time

class SensorDataForecast:

    def __init__(self):
        self.data_cache = pd.DataFrame(columns=['ds', 'y'])
        self.CACHE_LIMIT = 1440 # Let's keep 24 hours of data as an example, adjust as needed

    def poll_sensor_data(self):
        # Simulate reading sensor data; replace with actual data fetching code
        return float(pd.np.random.randn())

    def fetch_and_store_data(self):
        new_data = self.poll_sensor_data()
        current_time = pd.Timestamp.now()
        self.data_cache = self.data_cache.append({'ds': current_time, 'y': new_data}, ignore_index=True)

        # Limit cache size
        if len(self.data_cache) > self.CACHE_LIMIT:
            self.data_cache = self.data_cache.iloc[-self.CACHE_LIMIT:]

    def forecast_next_hour(self):
        model = Prophet(yearly_seasonality=False, daily_seasonality=False)
        model.add_seasonality(name='hourly', period=60, fourier_order=8)
        model.fit(self.data_cache)

        future = model.make_future_dataframe(periods=60, freq='min')
        forecast = model.predict(future)

        # Return the forecasted values for the next hour
        return forecast['yhat'].tail(60).values
```

```

def calculate_forecast_error(self, actual, forecast):
    return actual - forecast

def run_forecasting_cycle(self):
    while True:
        self.fetch_and_store_data()
        if len(self.data_cache) > 60: # Wait until we have at least 1 hour of data to start forecasting
            forecasted_values = self.forecast_next_hour()

            # Assuming one forecasted value for each minute, get the last forecasted value
            last_forecasted_value = forecasted_values[-1]

            # Wait for an hour to get the actual sensor value at that time
            time.sleep(3600)

            actual_value = self.poll_sensor_data()
            error = self.calculate_forecast_error(actual_value, last_forecasted_value)
            print(f"Forecasted value: {last_forecasted_value}, Actual value: {actual_value}, Error: {error}")

            time.sleep(60) # Poll every minute

# Execute
forecast_app = SensorDataForecast()
forecast_app.run_forecasting_cycle()

```

Here, Prophet handles the forecasting. We added hourly seasonality since you mentioned the polling happens every minute, and we're interested in hourly forecasts.

Please note that while Prophet doesn't require manual tuning like ARIMA, it still has some hyperparameters and settings that you might want to explore, especially if you notice that the forecasts can be improved for your specific use-case. The above code assumes there's no strong yearly or daily seasonality in your sensor data, but this is something you might want to revisit based on the actual patterns in your data.

 You:

would results be better if you limited self.CACHE_LIMIT = 1440 to 2 weeks or data or even a month?

 ChatGPT:

The optimal amount of historical data (or cache size) to use for forecasting can vary based on the nature of the data and the specific forecasting model being employed.

In the context of time series forecasting:

- 1. Seasonality:** If your sensor data has weekly or monthly seasonality, then limiting the cache to just 24 hours might miss out on capturing those patterns. For instance, if there's a recurring pattern every week, then using at least a week of data would help in capturing that pattern. If there's a monthly pattern, then using a month of data would be beneficial.
- 2. Data Stability:** If the nature of the data changes over long periods, using older data might actually degrade the forecasting quality. In such cases, you'd want your model to be more responsive to recent changes and might prefer a shorter window of historical data.
- 3. Computational Overhead:** Prophet works by fitting a model to the historical data. If you're using a very large dataset, the model fitting process could become computationally intensive, and you might experience a delay. It's essential to balance the amount of data with the computational resources and latency requirements you have.
- 4. Memory Consumption:** Storing more data will, of course, use more memory. Depending on the environment this code runs in, you might have memory limitations to consider.

Given these factors:

- **2 Weeks:** If you have reasons to believe there might be weekly patterns (e.g., weekends vs. weekdays), storing and using two weeks of data can help the model capture two instances of that pattern, which can improve forecast accuracy.
- **1 Month:** If there's potential monthly seasonality or if the data is relatively stable throughout the month, then using a month of data could provide better results. However, you'll need to monitor the computational and memory overhead.

In summary, there isn't a one-size-fits-all answer. The optimal cache size depends on the nature of your data and the specific requirements and constraints of your application. If unsure, it might be helpful to experiment with different cache sizes and compare forecast accuracy and performance to determine the best setting for your use case.

 You:

I want to use it on a commercial buildings electricity meter which definitely fluctuates in occupancy and temperature outdoors. Does the model refit every minute?



Given the context that the forecasting is for a commercial building's electricity meter, which fluctuates with factors like occupancy and outdoor temperature, the insights become clearer.

1. Seasonality and Data Patterns:

- There are likely daily patterns (day vs. night).
- Weekly patterns might exist (weekdays vs. weekends).
- Monthly or yearly patterns might occur due to holidays, business cycles, or weather changes across seasons.
- External regressors like outdoor temperature can be important.

2. Data Length:

- Given the daily and weekly patterns, a month of data is a good starting point. This way, the model can capture 4-5 instances of the weekly pattern.

3. Model Refitting: In the provided code, the model is refitted every minute with the most recent data. This refitting can be computationally expensive, especially with larger datasets. Depending on the available computational resources and the latency requirement for the forecast, you might want to consider less frequent refitting, such as every hour or even daily.

Given this context, here's a modified plan:

1. Use Prophet with both daily and weekly seasonality enabled.
2. Consider incorporating external regressors like outdoor temperature if you have that data.
3. Refit the model less frequently to reduce computational overhead.

Here's a modified version of the previous example to illustrate:

```
python
import pandas as pd
from fbprophet import Prophet
import time

class SensorDataForecast:

    def __init__(self):
        self.data_cache = pd.DataFrame(columns=['ds', 'y'])
        self.CACHE_LIMIT = 43200 # Roughly 30 days at one-minute intervals

    def poll_sensor_data(self):
        # Simulate reading electricity meter data; replace with actual data fetching code
        return float(pd.np.random.randn())

    def fetch_and_store_data(self):
        new_data = self.poll_sensor_data()
        current_time = pd.Timestamp.now()
        self.data_cache = self.data_cache.append({'ds': current_time, 'y': new_data}, ignore_index=True)

        # Limit cache size
        if len(self.data_cache) > self.CACHE_LIMIT:
            self.data_cache = self.data_cache.iloc[-self.CACHE_LIMIT:]

    def forecast_next_hour(self):
        model = Prophet(daily_seasonality=True, yearly_seasonality=False, weekly_seasonality=True)
        model.fit(self.data_cache)

        future = model.make_future_dataframe(periods=60, freq='min')
        forecast = model.predict(future)

        # Return the forecasted values for the next hour
        return forecast['yhat'].tail(60).values

    def calculate_forecast_error(self, actual, forecast):
        return actual - forecast

    def run_forecasting_cycle(self):
        while True:
            self.fetch_and_store_data()
            if len(self.data_cache) > 1440: # Wait until we have at least 1 day of data to start forecasting
```

```

# Refit the model, say, every hour instead of every minute
if pd.Timestamp.now().minute == 0: # Just an example, you can decide the refitting frequency
    forecasted_values = self.forecast_next_hour()

    # Assuming one forecasted value for each minute, get the last forecasted value
    last_forecasted_value = forecasted_values[-1]

    # Wait for an hour to get the actual sensor value at that time
    time.sleep(3600)

    actual_value = self.poll_sensor_data()
    error = self.calculate_forecast_error(actual_value, last_forecasted_value)
    print(f"Forecasted value: {last_forecasted_value}, Actual value: {actual_value}, Error: {error}")

time.sleep(60) # Poll every minute

# Execute
forecast_app = SensorDataForecast()
forecast_app.run_forecasting_cycle()

```

Lastly, if you have data like outdoor temperature, you can use that as an external regressor in Prophet. You'd add the temperature data to the `data_cache` DataFrame and use `model.add_regressor('temperature_column_name')` to inform Prophet to use it in forecasting. Remember, including such regressors will require you to provide future values for them when forecasting. If accurate future values aren't available (e.g., for outdoor temperature), you might use short-term weather forecasts to provide estimates.

 You:

is the model refit every time it polls for a new value?

 ChatGPT:

Yes, in the given code, the model is refitted every time it polls for a new value. Specifically, every minute when new data is fetched, the model attempts to refit and forecast. This frequent refitting can be computationally intensive, especially as the dataset grows.

If you want to optimize this behavior, you can choose to refit the model less frequently. One common approach for such a use case might be to refit the model every hour or even every few hours, depending on the computational resources available and the volatility of the data.

To refit the model, for example, every hour, you can modify the `run_forecasting_cycle` method:

```

python
def run_forecasting_cycle(self):
    while True:
        self.fetch_and_store_data()

        if len(self.data_cache) > 1440: # Wait until we have at least 1 day of data to start forecasting

            # Refit the model every hour instead of every minute
            if pd.Timestamp.now().minute == 0: # Refit only when the minute is 0 (i.e., on the hour)
                forecasted_values = self.forecast_next_hour()

                # Assuming one forecasted value for each minute, get the last forecasted value
                last_forecasted_value = forecasted_values[-1]

                # Wait for an hour to get the actual sensor value at that time
                time.sleep(3600)

                actual_value = self.poll_sensor_data()
                error = self.calculate_forecast_error(actual_value, last_forecasted_value)
                print(f"Forecasted value: {last_forecasted_value}, Actual value: {actual_value}, Error: {error}")

            time.sleep(60) # Poll every minute

```

With this modification, the model will only refit once every hour while continuing to poll new data every minute. Adjusting the refitting frequency helps balance between computational efficiency and forecast accuracy.

 You:

does the model still forecast a new value every minute but only refit the model every hour?

 ChatGPT:

I apologize for the oversight. In the previous modification, the model refits and forecasts once every hour. If you wish to forecast a new value every minute while refitting the model only every hour, we'll need to adjust the code further.

Here's a modified version of the `run_forecasting_cycle` method to accomplish that:

```
python
def run_forecasting_cycle(self):
    while True:
        self.fetch_and_store_data()

        if len(self.data_cache) > 1440: # Wait until we have at least 1 day of data to start forecasting

            # Refit the model every hour
            if pd.Timestamp.now().minute == 0:
                self.model = Prophet(daily_seasonality=True, yearly_seasonality=False, weekly_seasonality=True)
                self.model.fit(self.data_cache)

            # Forecast every minute regardless of when the model was last refitted
            forecasted_values = self.forecast_next_hour()

            # Assuming one forecasted value for each minute, get the last forecasted value
            last_forecasted_value = forecasted_values[-1]

            # Wait for an hour to compare the forecasted value with the actual sensor value
            if pd.Timestamp.now().minute == 59:
                time.sleep(60)
                actual_value = self.poll_sensor_data()
                error = self.calculate_forecast_error(actual_value, last_forecasted_value)
                print(f"Forecasted value: {last_forecasted_value}, Actual value: {actual_value}, Error: {error}")

            time.sleep(60) # Poll every minute
```

In this modified method:

- The model refits at the start of each hour (i.e., when the current minute is 0).
- A new forecast is generated every minute.
- The forecasted value for the end of the next hour (i.e., the last forecasted value) is compared to the actual value at the end of that hour.

This way, you'll get a new forecast every minute using the most recently refitted model, but the model itself will only be refitted once every hour.

 You:

can you write up a github README for this?

 ChatGPT:

Certainly! Here's a sample README for your project.

Electricity Meter Forecasting

This repository contains a tool for forecasting the electricity usage of a commercial building using Prophet, a forecasting tool provided by Facebook. The tool continuously polls electricity meter readings at one-minute intervals, caching these readings, and uses them to make short-term forecasts.

Features

- **Continuous Polling:** The tool polls the electricity meter every minute to update its dataset.
- **Efficient Forecasting:** Forecasts the electricity usage one hour into the future every minute.
- **Model Refitting:** Refits the forecasting model every hour to account for new patterns and changes.
- **Error Calculation:** Continuously prints the error between forecasted and actual electricity usage.

- **Cache Management:** Automatically manages cached data to prevent memory overflow, keeping the most recent readings.

Prerequisites

- Python 3
- pandas
- fbprophet

Setup and Installation

1. Clone this repository:

```
bash
git clone https://github.com/yourusername/electricity-meter-forecasting.git
cd electricity-meter-forecasting
```

2. Install the required packages:

```
bash
pip install pandas fbprophet
```

3. Update the `poll_sensor_data` method to match your sensor data fetching mechanism.

Running the Forecaster

Execute the main script:

```
bash
python main.py
```

This will start the polling and forecasting process. The tool will print the forecast error for each hour.

Contributing

Pull requests are welcome. For major changes, please open an issue first to discuss what you'd like to change.

License

[MIT](#)

Remember, this README assumes a certain structure and naming convention for your project. You'll need to adjust file names, paths, or other specific details to match your actual project setup.

 You:

how do you bold in git readme?

 ChatGPT:

In a GitHub README, which uses Markdown syntax, you can bold text using two asterisks (**) or two underscores (__) before and after the text.

For example:

```
markdown
**This text is bold**
```

or

```
markdown
__This text is also bold__
```

When rendered on GitHub, both of the above examples will appear as:

This text is bold

You can incorporate this into your README or other Markdown documents on GitHub to emphasize certain parts of the text.
