Prediction Report

Atalay Özmen and Hande Şirikçi

TUM School of Computation, Information and Technology

July 10, 2023

1 Problem Definition

Our goal in this part was to train our time series forecasting model for room count prediction by invoking a function in the IoT platform using FaaS, obtaining the trained model from our VM that runs on Google Cloud, and send predictions to Kibana and IoT platform every five minutes.

2 Training of the Prediction Model

Our time series forecasting is done using Facebook's Prophet. Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects.

We thought Prophet was a good choice for our purpose since our data has both daily and weekly seasonality. Daily seasonality comes from the fact that we have non-zero room count only between 8:00 AM and 7:00 PM, and weekly seasonality comes from the fact that we do not have lectures on the weekends.

2.1 FaaS Setup

In our FaaS function in the IoT platform, training of our model is done. We managed to get stable and correct data since 12.06.2023 using our device, so the training is only done with the data after this date. This data for training the model is received from Elasticsearch.

In order to have the packages we want in the FaaS, we used a Dockerfile for defining the packages we need, and pushed it to DockerHub. Then the URL for this DockerHub was passed to the FaaS function parameters. We have used the Dockerfile given to us by the course instructors, but also added an extra line for Prophet.

```
FROM openwhisk/actionloop-python-v3.7

RUN pip install --no-cache-dir scipy
    statsmodels pandas requests elasticsearch
     opentelemetry-exporter-jaeger
    opentelemetry-sdk opentelemetry-sdk minio
     cython pystan

RUN pip install --no-cache-dir --upgrade
    prophet
```

After receiving the data, it should be in a format where we can give it to Prophet so that it can run the prediction. Because of this, we must parse the data. Prophet wants a Dataframe with columns named ds and y, where ds is datastamp and y is the room count in our case. We also did some timestamp localization for the unix epoch to get the correct time in Kibana, which isn't involved in pseudocode below. Parsing is done like below:

```
def parse_data(json_data):
    time_values = []
    sensor_values = []

    for data in json_data
        timestamp = data.timestamp
        data_value = data.value
        time_values.append(timestamp)
        sensor_values.append(data_value)

    df = pd.DataFrame(
        {
            "ds": time_values,
            "y": sensor_values
        }
    )
    return sorted_df
```

After getting the data ready for Prophet, we use the library to train our model.

First we initialize our Prophet object. The parameter changepoint_prior_scale defines how the model reacts to sudden changes. By default the value is 0.05, but since the room count is a pretty dynamic value, we increased it to 20 for better fitting. This number was obtained by experimenting.

Smaller number for this parameter resulted in way too small predictions, and a higher number would result in getting non-zero data between at night between 7:00 PM and 8:00 AM. Therefore, this number seemed decent.

We also passed in the parameters for daily seasonality and weekly seasonality. This parameters represents the fourier order. When we pass a higher fourier order, our model can adapt to fast changes in the data better. Changes in our data happen very frequently, and is very dynamic. This is why these parameters are necessary.

```
    model = Prophet(changepoint_prior_scale
    =15, daily_seasonality=600,
    weekly_seasonality=80)
```

We added two extra regressors. One regressor is for daily hours, where we indicate that we have room count between 9:00 AM and 18:00 PM.

```
    mask = (df['ds'].dt.hour >= 8) & (df['ds'
    ].dt.hour <= 19)
    df['subdaily_mask'] = np.where(mask, 1,
    0)
    model.add_regressor('subdaily_mask')
```

We also added another regressor for similar purpose, to indicate we don't have room count on the weekends.

```
    df['weekend_regressor'] = np.where(df['ds
    '].dt.dayofweek >= 5, 1, 0)
    model.add_regressor('weekend_regressor')
```

After we set our regressors and Prophet object, we pass the dataframe to our model object's fit function and get the trained model. Then, we serialize our trained model using pickle library and save it to Minio for later usage in VM.

```
    model.fit(df)
    bytes_file = pickle.dumps(model)

    #Storing Model
    result = minioClient.put_object(
        bucket_name="group2-iot",
        object_name="model2.sav",
        data=io.BytesIO(bytes_file),
        length=len(bytes_file)
    )
```

3 Edge VM Setup

For the Edge setup, we have used Google Cloud's compute engine and provisioned a g1-small type VM. To install the packages and write the code to be ran on the VM, we SSH'd into the VM from Google Cloud Platform.

In the VM, we were responsible for sending our predictions to Kibana and IoT platform every 5 minutes. We have implemented a Python script that downloads the latest model from Minio, makes a prediction about the room count after current time + 15 minutes and send it to Kibana and IoT Platform.

In Kibana, sending the data happens with indexes. However, in IoT platform, we have used our device JSON configuration to store the data. With the API of IoT platform, we are only able to get and update data. In order to define our data scheme, we use the JSON configuration of our device in IoT platform like this.

```
{
    "prediction":"12"
}
```

The pseudocode for explaining the Python script in the VM is like this:

```
serialized_model = Minio.get("model.sav")
model = serialized_model.unserialize()
prediction = model.predict(current_time + 15
    minutes)
kibana.send_data(prediction)
IoTPlatform.update_data("prediction",
    prediction)
```

3.1 CronJob

This script needs to run every 5 minutes. To ensure that this happens, we had two ideas: we could use a sleep in the function for five minutes and run it in an infinite loop. But this seemed kinda bad. Then we stumbled upon CronJob.

CronJob is for creating jobs in a scheduled manner. In our case, we needed to run this script for every five minutes. By running the crontab -e command in the terminal, we enter the editor where we can define our scheduling. For scheduling for every 5 minutes, the code is written as below.

```
    */5 * * * * python3 ~/main.py
```

The */5 at the start defines that the task should run every 5 minutes, 12:00, 12:05, etc. The other stars are for hours, days, weeks, months and years, which indicate that this should run every day of the year. After we set this up, we can just close our VM terminal, and it would handle running the code for us 24/7.

4 Device Setup

In the device, a new task was set up that fetches data from the IoT platform using the C Library for IoT platform. The fetching is done every 5 minutes, where we get the prediction for fifteen minutes into the future. Simplified pseudocode is as below.

```
void vUpdatePrediction()
{
    for (;;)
    {
        prediction = platform_api.retrieveVal
    ("prediction")
        sleep(300) //sleep for 5 minutes
    }
}
```

5 Comparison of Predictions and Reference Values

We have made predictions between 26.06.2023 8:00 AM and 30.06.2023 6:00 PM, and compared the result with the reference values given to us. We have made predictions in two time resolutions, with 5 minutes and 4 hours.

For comparing the reference values with prediction values for the 4 hour resolution, we had to filter the reference values to take counts with timestamps 8:00 12:00 16:00 etc. since the original reference values are in five minute intervals.

We have covered mean squared error (MSE), root mean squared error(RMSE) and symmetric mean absolute percentage error (sMAPE). Originally, we were asked to do MAPE, but since we have actual values that are zero, and MAPE includes division by zero, this was not possible. sMAPE is a slightly modified version of MAPE, that allows us to work with zero values.

For mean squared error, we calculate the difference between each predicted value and its corresponding actual value. Then, we square each of these differences. After squaring the differences, we calculate the average of these squared differences by summing them up and dividing by the total number of values. This average is the mean squared error. Now, if there are larger differences between predicted and actual values, squaring them will result in even larger values. In other words, MSE gives more weight to larger differences compared to smaller ones. This is because squaring amplifies the effect of larger differences, making them contribute significantly to the overall error.

Root mean squared error on the other hand does not amplify the higher differences, but gives us the average difference between the actual and predicted value.

sMAPE is another metric that quantifies the average percentage difference between predicted and actual values. Unlike other error metrics, sMAPE considers the relative magnitude of the difference, rather than the absolute difference. It calculates the absolute difference between each predicted value and its corresponding actual value. Then, it sums up these absolute differences and divides them by the sum of the absolute values of the predicted and actual values. Finally, this value is multiplied by 200 to express it as a percentage.

Resolution	**MSE**	**RMSE**	**SMAPE**
5 Minute	245.19	15.65	44.19
4 Hour	270.46	16.44	84.91

Table 1 Error Metrics

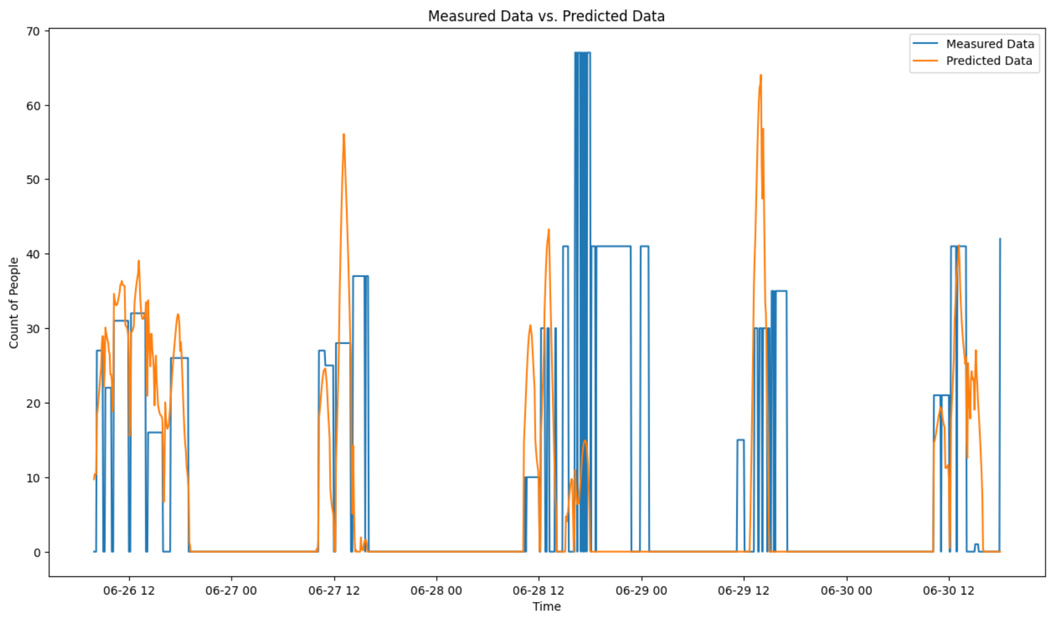

Figure 1 Comparison of our predictions and reference values on 5 minute resolution

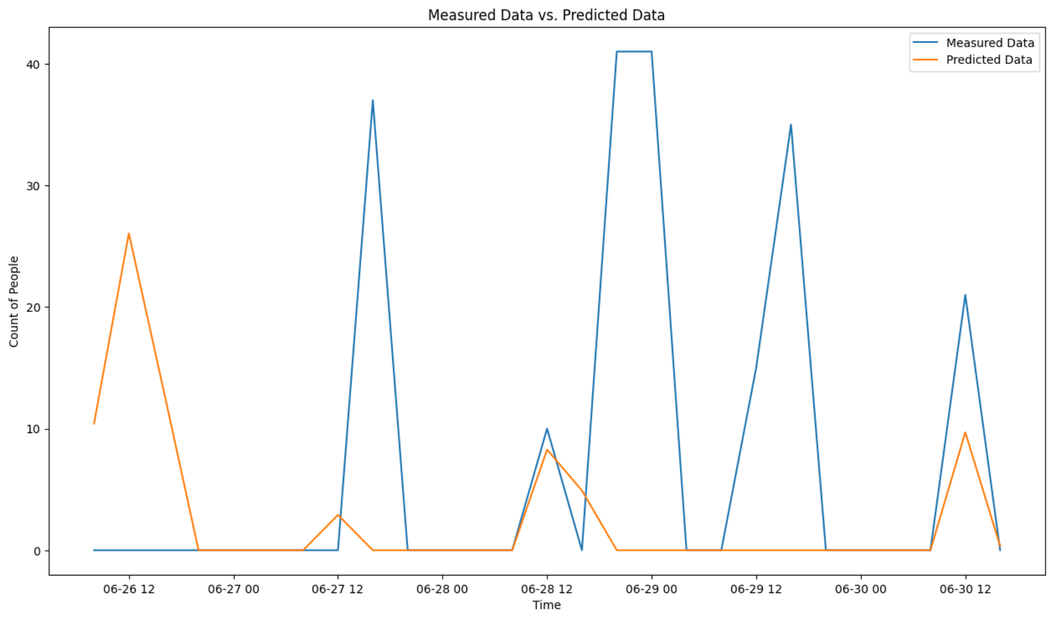

Figure 2 Comparison of our predictions and reference values on 4 hour resolution

6 Metrics

In order to do monitoring, we needed to set up OpenTelemetry and Jaeger. OpenTelemetry is responsible for collecting data from our code, and then uses an exporter to export data to Jaeger. Jaeger does the job of visualization.

Using OpenTelemetry, we have looked at the time it takes for the model to be trained. We also looked at the size of our model at it's best form.

Time to train	3.29 minutes
Time to save	137.12 milliseconds
Size of model	710.5 KiB

Table 2 Performance Metrics

7 Discussion

For the 5 minute resolution, our predictions showed relatively good results for most days except 28.06. During this day, reference values show room count being non-zero until midnight 00:00, which is very unexpected since usually, the non-zero data started at 8:00 AM and ended at 7:00 PM and our trained model got used to this. Because of this, the error metrics are highly affected, especially mean squared error, where higher differences between actual and predicted data are punished more severely.

For the 4 hour resolution, our model was not that successful. Since room count can be huge at one point, and zero after 5 minutes, during the filtering, we might have coincidentally picked the zero values from the measurement data. This is clear during 26.06, where measurement data showed zero all day after filtering the data for 4 hour intervals. Because of these reasons, our predictions weren't that successful with 4 hour resolution.

Another reason for the more than expected error rates are the fact that our device could not measure the room count totally accurately during the previous dates. This lead to our model training with data that was not totally accurate. In our sensor report, it can be seen that especially on the day 28.06, the room count measured by our device has huge differences with the reference data. During that day, the room count went up to 71 in reference data, however, our measured data showed zero most of this day. Because of this, the data that we trained our model with was not totally accurate in the first place, which caused it to give suboptimal predictions.

To prove my points, I also did error checking by taking the same predictions that I had, but this time comparing it with the data that my device measured instead of the reference data given by the course instructors between 26.06 and 30.06 and I got much lower error rates.

Resolution	MSE	RMSE	SMAPE
5 Minute	58.76	7.66	72.43

Table 3 Performance Metrics

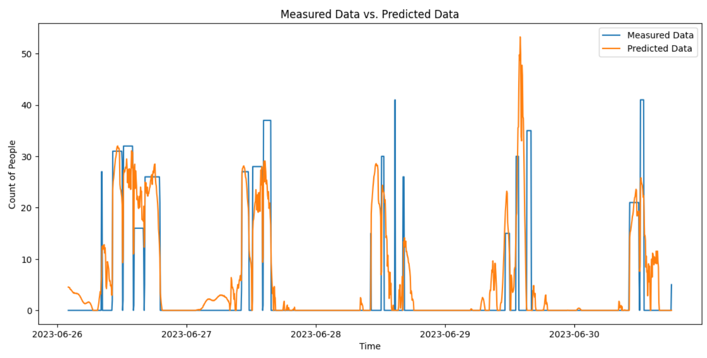

Figure 3 Comparison of our predictions and our measurements