Technical University of Munich

Informatics 10 – Chair of Computer Architecture and Parallel Systems (Prof. Schulz)

Master Praktikum: IoT (Internet of Things) (IN2106, IN4224)

MILESTONE 2

prediction models, report

Group: 2

Students and matriculation number:

Nadija Borovina (03742897)

Nedžad Hadžiosmanović (03742896)

Lecturers:

Gerndt, Hans Michael, Prof. Dr.

Podolski Vladimir, M. Sc.

Chenyuan Zhao

Munich, July 2021.

Before starting to address what was done by our group for the task of building prediction models, we would like to signal out that because of the lack of space that we have in this report, we made a detailed documentation of everything that we done. This documentation includes the full Python code that we wrote for this task, along with detailed explanations, and it can be found on the following Google Drive link (for which you will not need an invite, as it is shared with everybody who has a link to it):

https://drive.google.com/drive/folders/1255CfLR2tnjlS_VY9q1KSYBCXSCfD_N1?usp=sharing

Throughout this report, you will often encounter references to what is written in the script by phrases like 'this was performed in "*Section XY*" in the script'.

1. Tidying up the dataset

As a part of tasks covered during milestone 1, we had to build a system which would be able to receive messages through the internet about people entering and leaving. After our system receives a message, it has to interpret it's meaning, and take actions. These actions influenced the counter that our system outputs. Another responsibility that our system had was to publish to the IoT platform the count each time it changes together with a timestamp at which this action was registered by our system, and in addition to that to push the counter and timestamp every 15 minutes, no matter if the counter was changed in the 15 minutes or not.

When we pulled the data from the platform as the first part of the task of making prediction models (look at "*Section 1.1*" in the script), we were surprised to see that some unexpected publishes were done. In the further text we give two reasons such unexpected behavior occurred, and what caused it.

Firstly, due to the fact that at one point our group had two systems running (i.e. both ESP32 chips with our program flashed on them) we were doing two publishes for every enter or leave action, as well as doing two publishes every 15 minutes.

Secondly, another problem which we noticed while examining published data is the fact that because of breaking of internet we had a count which is greater than zero in periods of a day and in which we would not expect to have a count different from zero, and this as well caused some enters to be missed by our system.

Because of the described problems with the data, before making any prediction models, we had to tidy up the dataset, to be able to have correct inputs to prediction models.

For most of the prediction models to work properly, it has to be fed with **equally spaced data**, i.e. the timestamps of observations had to be equally spaced.

When the task for making prediction models was published as part of this praktikum, it was not defined what should be the distance between two consecutive observations. Our group decided to make a program which would search the dataset that we have, and find two observations which had the smallest difference in

timestamps between them, and according to this value to add data to the dataset to make it equally spaced (look at "*Section 1.3.2.*" in the script). Even on the first run, our program recognized that there were two observations which happened with one second in between, so to make the dataset equally spaced we changed the dataset by adding new observations, so that between each observation we had one second. To do this, between every two observations that had a space between them less than one second, we added new observations by setting the timestamp through the code, and copying the count from the observation which had the lower timestamp of the two. By using this approach, even when having a little data collected, we soon realized that we were generating a huge number of new observations, and that this will possibly cause problems when there is more data generated.

During the following two weeks it was suggested by the lecturers that having data which is equally spaced with one second is going to cause us problems while building the models (which it did, but we will not write about this here, because this is a section of the report dedicated to tidying the dataset), and it was suggested to all groups to try to make some other spacings such as 1 minute, 15 minutes, 30 minutes and 60 minutes.

Making the data equally spaced by using the suggested time between two consecutive observations was much easier because we could use already existing functions to perform these actions, as they did not require generating new data (except the spacing of 1 minute).

Our group decided to make a separate dataframe with the data that we collected and each of them having one of the suggested spacings between consecutive observation (look at "*Section 1.3.3.*", "*Section 1.3.4.*", "*Section 1.3.5.*" and "*Section 1.3.6.*" in the script). We did this so that we are ready for when the spacing between consecutive observations was finally set by the lecturers.

Later on it was agreed that the spacing of 15 minutes between consecutive observations should be used, so for the training of the models we used this dataset, leaving the four datasets unused.

2. Performing test on the dataset

Another thing we had to do before starting to make prediction models is to examine the properties of the dataset that we had.

We firstly visualized our dataset and immediately realized that our dataset has **seasonality** on weekly level (look at "*Section 2.1.1.*"). This is an important conclusion which we will leverage in the later part of the script while building the prediction models.

When examining time series data we are interested in knowing is the dataset which we working with **stationary** or not, because many prediction models (including the ones which we used for making predictions) make an assumption that the data with which they are working with is stationary. In order to find out is the dataset with which we are working with is stationary or not, we can use multiple methods, such as:

1. **Look at Plots**: You can review a time series plot of your data and visually check if there are any obvious seasonality (look at "*Section 2.1.1.*" in the script).
2. **Summary Statistics**: A quick and dirty check to see if your time series is non-stationary is to review summary statistics. You can split your time series into two (or more) partitions and compare the mean and variance of each group. If they differ and the difference is statistically significant, the time series is likely non-stationary (look at "*Section 2.1.2.*" in the script).
3. **Statistical Tests**: You can use statistical tests to check if the expectations of stationarity are met or have been violated. (look at "*Section 2.1.4.*" in the script). A well-known test for examining stationarity in our data is the so-called Augmented Dicky-Fuller test (ADF). According to this test, the null hypothesis and alternative hypothesis are given as:
    - **Null Hypothesis (H0)**: If failed to be rejected, it suggests the time series has a unit root, meaning it is non-stationary. It has some time dependent structure.
    - **Alternate Hypothesis (H1)**: The null hypothesis is rejected; it suggests the time series does not have a unit root, meaning it is stationary. It does not have time-dependent structure.

    (look at "*Section 2.1.4.*" in the script)

Our group used all three methods to check for seasonality of our dataset, and at the end they all gave the same answer, that the dataset was stationary. At this point we had everything ready for making the prediction models, as the dataset that we were working with was equally spaced, and in addition it was stationary.

3. **Building prediction models**

The first step of building models is to make splits of our dataset into training set D_T and test set D_t. A very important fact when working with time series data (which is a type of sequential data), is that when making these splits we have to make sure that each of the sets has data that is sequential inside of it. Making splits in such a way is different compared to how the splits are made when working with unsequential data, as in that case we make the splits by randomly sampling observations from the dataset. So, we performed the split to the two datasets by assigning 90% of the dataset to be the training set, and the remaining 10% to be the test set from (look at the "*Section 2.3.*" in the script). In *Figure 1* below you can see a visualization made in the script, that shows the data that we collected, and how the split of the dataset is performed.

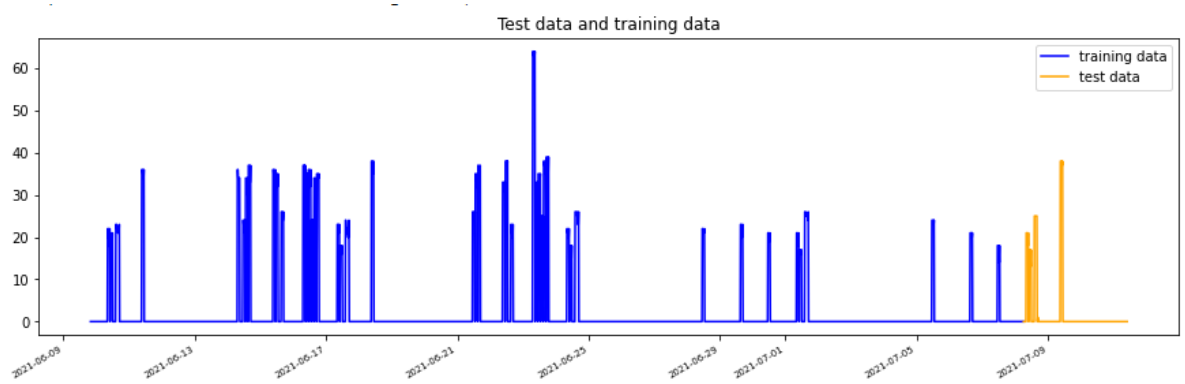

Figure 1: Split of the dataset in training set and test set.

We finally came to the part of the task in which we were building prediction models. Our group made multiple prediction models:

1. **AR** (look at "*Section 2.4.1.*", "*Section 2.4.2.*" and all of their subsections in the script)
2. **MA** (look at "*Section 2.4.3.*", "*Section 2.4.4.*" and all of their subsections in the script)
3. **ARMA** (look at "*Section 2.4.5.*" and all of their subsections in the script)
4. **ARIMA** (look at "*Section 2.4.6.*", "*Section 2.4.7.*", "*Section 2.4.8.*" and all of their subsections in the script)
5. **SARIMAX** (look at "*Section 2.4.9.*", "*Section 2.4.10.*" and all of their subsections in the script)
6. **LSTM** (look at "*Section 2.4.11.*" and all of their subsections in the script)
7. **PROPETH** (look at "*Section 2.4.12.*" and all of their subsections in the script)

In the list of prediction models which is given above, models are given in the order in which we tried them out. As it can be seen from the list, we started from very simple models, moving on to more complex ones.

Because in the setting of the milestone 2 it is said that we should write about only three models that we built, that we find the best out the models that we built, in the further part of the text we are going to discuss about SARIMAX, LSTM and PROPHET models in more detail.

3.1 SARIMAX prediction model (and ARIMA prediction model with set "*seasonal_order*" attribute)

There are two sections that are dedicated to building a SARIMAX model in the script, and those are "*Section 2.4.9.*" and "*Section 2.4.10.*". In "*Section 2.4.9.*" we are building a ARIMA model with "*seasonal_order*" parameter set, while in "*Section 2.4.10.*" a SARIMAX model is coded.

Preferred model of the two would be the SARIMAX model, but the problem with this model was that it was too big to be built on our laptops. At that point we contacted the lecturers for advice on what to do about the issue, as our laptops were simply not powerful enough to build such a model. We were told that we would need a machine that has at least 8GB of RAM on it, which none of the devices that we own had, so we decided to try out working on Google Colabs. We adjusted the code slightly so it can work on Google Colabs, on which we had 12GB of RAM, but we were still not able to solve the issue, i.e. we were not able to train the model. The reason why such a model is very hard to train is the weekly seasonality which we have inside of our dataset. Remembering that we are working with a dataset with equally spaced observations which are 15 minutes apart, **4*24*7** data samples from such the dataset are observations concerning one week of the data, which is why we had to set the attribute "*seasonal_order=(0, 0, 0, 4*24*7)*", as there are 4 observations in an hour, and 4*24 observations in a day. In addition to this, we are using an already existing function "*auto_arima*" to help us specify the best suited values for attribute "*order*" in the "*SARIMAX*" function. More precisely, the "*auto_arima*" function helps us choose attributes "*p*" (defining the number of AR parameters), "*d*" (defining the number of differences) and "*q*" (defining the number of MA parameters). Using this function is great in a sense that it will give the best suited values of these parameters, but there is a downside to it, as it is very expensive to compute. The function performs stepwise AIC in choosing the best suited parameters for the model (i.e. the model itself). To try to make the function as quick as possible, a user can specify the maximum number that each of the parameters that this function outputs can take, and therefore narrowing the number of possible combinations of parameters.

As we were not able to build the SARIMAX model described in the paragraph above, we taught that there was no point in building a SARIMAX with different seasonality, just to make it work, so as an alternative we build a ARIMA model with set "*seasonal_order*" attribute. This model is built in "*Section 2.4.9.*" and its subsections. The reason why we chose this model as an alternative to the SARIMAX model is because we were able to set the seasonality of our data correctly by setting "*seasonal_order=(0, 0, 0, 4*24*7)*", while the attribute "*order*" is set by using the "*auto_arima*" function discussed in previous paragraph. Thew model did not do well in making predictions, which was kind of expected, taking into consideration that when we have seasonal data that we should use SARIMAX instead of ARIMA model.

3.1.1. Our ARIMA model with "*seasonal_order*" parameter set, performing predictions on the training set data

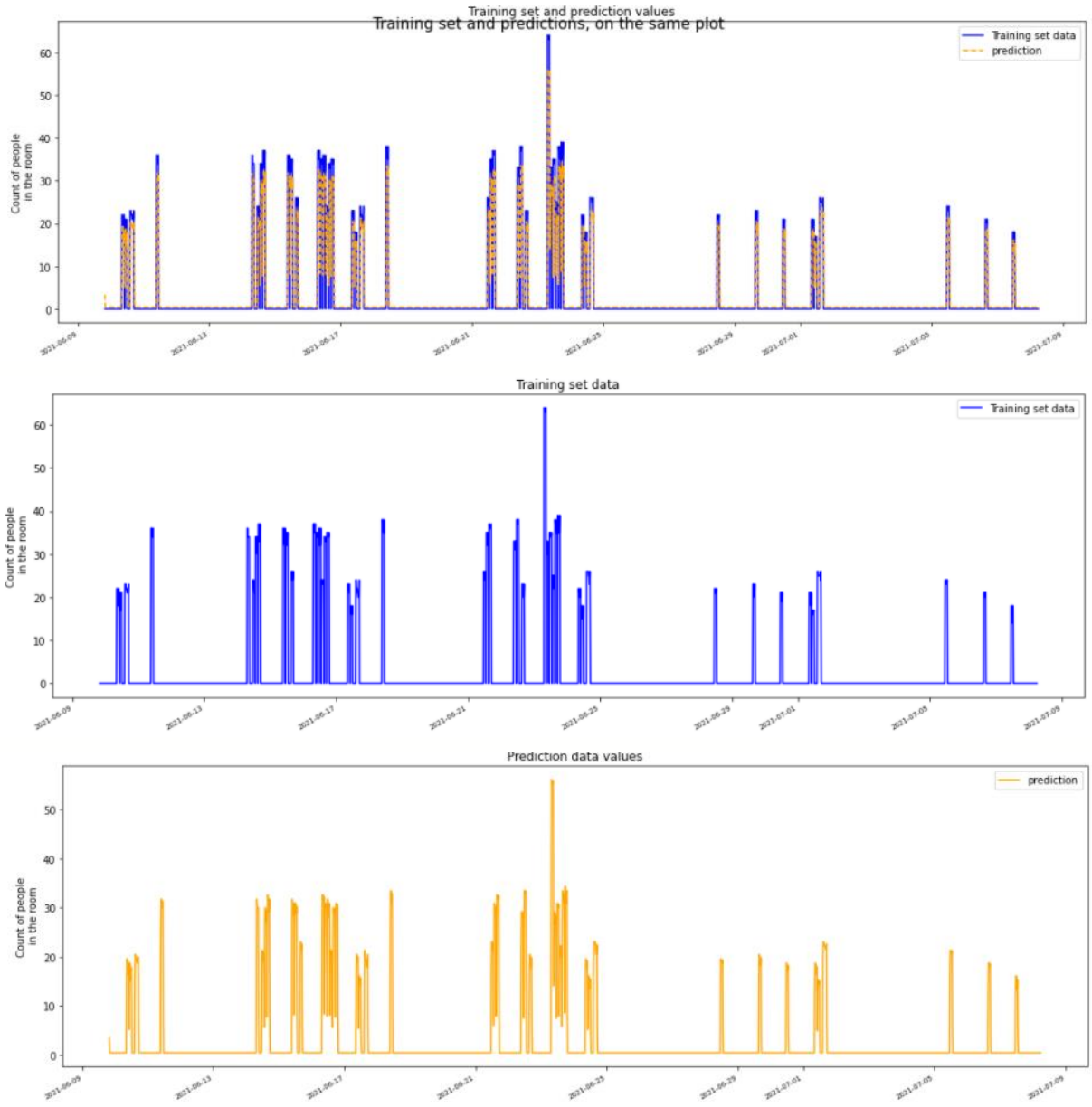

Figure 2: Three plots in which the performance of our ARIMA model (with set "*seasonal_order*" parameter) is shown, where we are making predictions for the data on which the model was trained on. The first plot is a plot in which we can see both the training data and the predictions that the model made, while on the second and third plot we can see separately the training set data and predictions made by the model, respectively.

From *Figure 2* we are able to see that the predictions that the ARIMA model with specified "*seasonal_order*" parameter for the data on which it was trained on. From what is seen on the graphs, one may argue that the

model overfit to the data on which it was trained on, as it is making predictions which almost perfectly follow the training set data.

3.1.2. Our ARIMA model with "*seasonal_order*" parameter set, performing predictions on the test set data

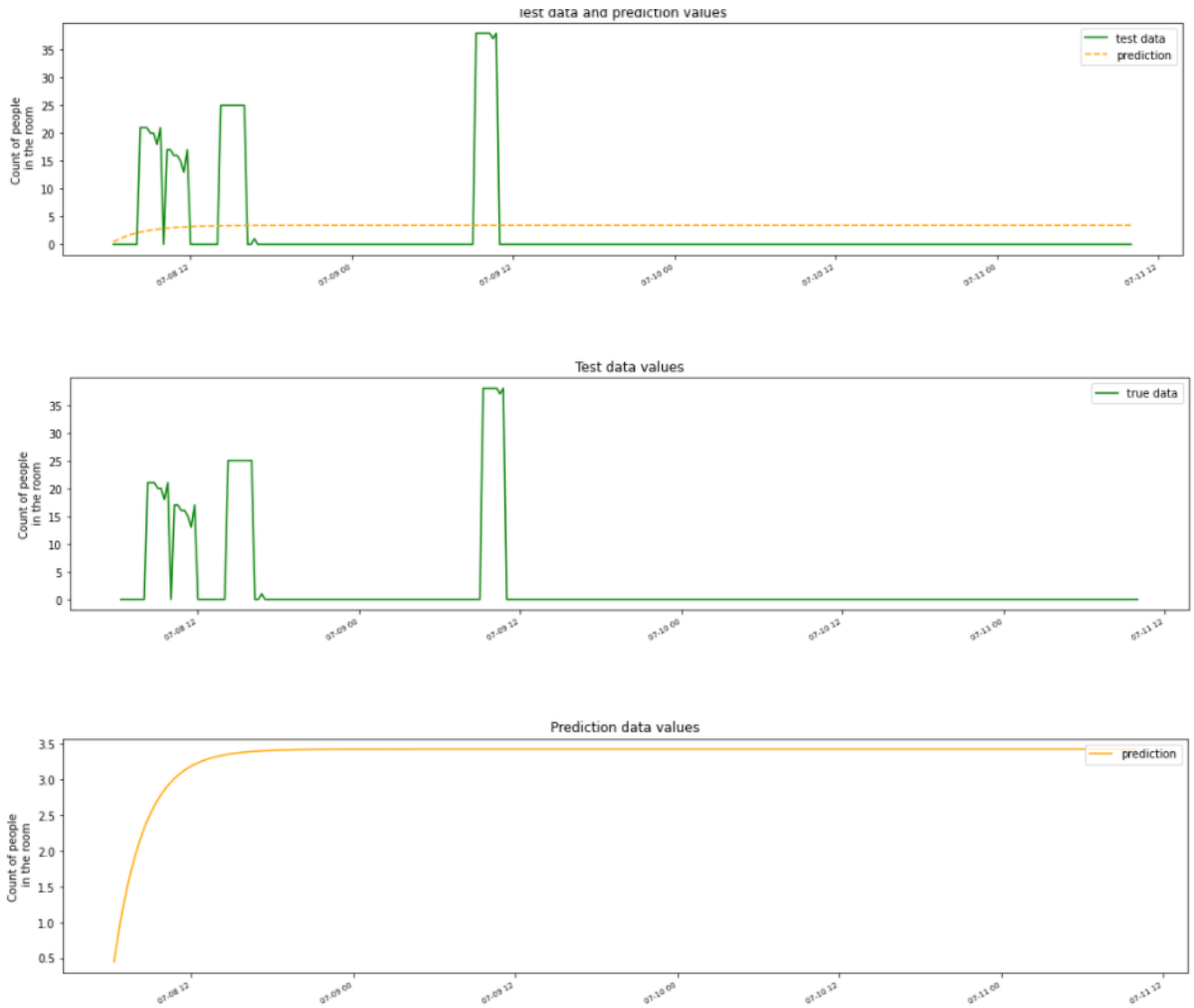

Figure 3: Three plots in which the performance of our ARIMA model (with set "*seasonal_order*" parameter) is shown, where we are making predictions for the test set data. The first plot is a plot in which we can see both the test data and the predictions that the model made, while on the second and third plot we can see separately the test set data and predictions made by the model, respectively.

On *Figure 3* we can see that the ARIMA model does not perform well when applied to unseen data (in contrast to the results that it showed on the data on which it was trained on).

3.1.3. Accuracy of our ARIMA model with "*seasonal_order*" parameter set

Accuracy score for our ARIMA model with specified "*seasonal_order*" on train set is surprisingly good. Almost 85% of the values are correctly predicted.

```
print("Percentage of correctly classified samples is: " + str(accuracy_score
```

```
Percentage of correctly classified samples is: 85.59657218193803%
```

While on the other hand, on test data accuracy is close to zero.

3.2 LSTM prediction model

There is one section that is dedicated to building a LSTM model in the script, and it is the "*Section 2.4.11.*".

When building the Long Short Term Memory model, what we were essentially doing is that we were making a recurrent neural network. RNN-s have an advantage over vanilla neural networks because the are able to handle input sequences. LSTM is a type of RNN, but which has an advantage over vanilla RNN-s in a sense that it introduced highway of gradients and system of gates which help in solving the problem of vanishing gradient during the backpropagation in which the network is looking for partial derivatives with respect to every model parameter in the network. On the other hand, standard RNN-s would encounter vanishing gradient problems as the dependencies on previous observations become bigger.

First attribute of the "*LSTM*" function (called "*units*")is specifying the number of units, which specifies the dimension of the output vector from the LSTM layer of the network.

Also, we set parameter "*input_shape=(1, look_back)*", where "*look_back=4*7*24*". The first argument of the attribute "*input_shape*" (i.e. the "*1*") specifies how many timestamps do we have in a single input, while the second argument of the attribute "*look_back*" specifies how many features are used for one timestamp. As it can be noticed, we are using 4*7*24 'feature', but what does this mean in our use-case is that for each input we are looking at 4*7*24 observations from the training set.

At the end, we added a Dense output layer with a single neuron. This layer acts as the output layer of the network, meaning that we built a model which as an input gets a sequence (i.e. many inputs), and as an output it gives a single value (i.e. one output.)

The optimizer which we chose for the process of training for this neural network is Adaptive Moment Estimation (or in short Adam). The reason why we chose this optimizer is because it most often used optimizer nowadays, because of the fact that uses both first moment, and the second moment when doing update steps, which essentially means that we do not go down the loss/cost function in each direction with same steps (this could be taught as if we had different constant multiplying partial derivatives of the gradients during the update step of model parameters), and in addition to this, if by looking at previous update steps the optimizer concludes that we have been taking steps in the same direction, we age going to make bigger steps in that direction to reach the local minimum (which may or may not be a global minimum at the same time) faster. Specifying Adam to be the used optimizer is done by specifying an attribute "*optimizer='adam'*".

In addition to this we chose to use Mean Square Error as the loss/cost function for the network, which is essentially the L2-loss/cost function which in addition averages the final result by dividing it with the number of samples used, and thus making it invariant to the number of samples.

For the number of epochs we chose 20 (done by setting "*epochs=20*") purely because of the fact that the neural network model would need less time/iterations to converge. In addition to this we choose the batch size to be equal to 1 (i.e. the network weights are updated after each training example), which can have the effect of faster learning, but also adds instability to the learning process as the weights widely vary with each batch.

We left the default activation function which is TanH, but as this activation function was used we had to perform a normalization of our input samples, in order to avoid having vanishing gradient problems because of the derivate w.r.t. the activation function while performing backpropagation.

By specifying "*verbose=2*" we are not actually influencing the model itself, but rather the output that is going to be shown to us during while fitting the model. By setting this attribute to be equal to 2 we will be shown both the number of epochs which is performed currently, along with the progress bar (which is showing how the training is progressing).

3.2.1. Our LSTM model performing predictions on the training set data

On *Figure 4* we can see a graph of how does our LSTM model perform when trying it out on the data on which it was trained. The predictions are given in orange, training data is given in blue, and the test data is represented with green in the graph.

At this point you might notice that there is something rather unusual that happens on the graph, and that is that there are no predictions for the beginning part of the training data. This is a consequence which happened because we set the "*look_back*" parameter to eb equal to 4*24*7, because this means that in order to make an input to the neural network we will need to input a sequence of observations 4*24*7 long. This is not possible for the first 4*24*7 observations, as they simply do not have enough preceding observations.

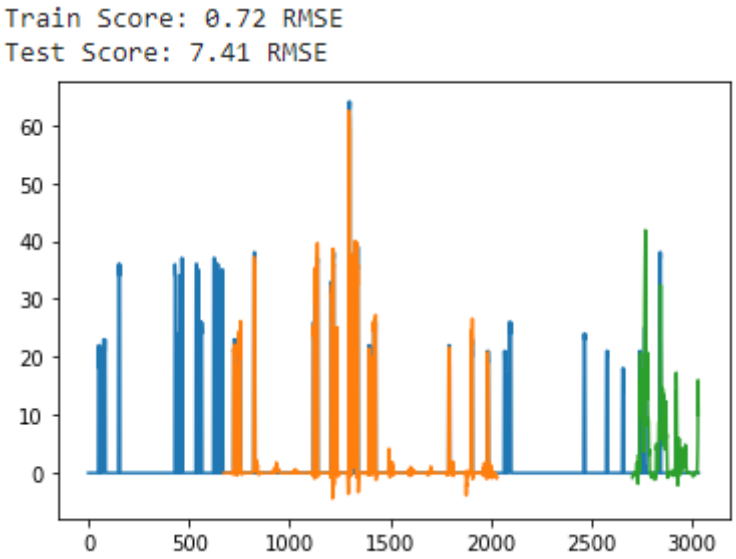

Figure 4: The performance of our LSTM model on the training set data

3.2.2. Our LSTM model performing predictions on (simulated) test set data

As described above, we would be encountering a problem if we tried to simply use our model for prediction on the test set data, simply because it has too small number of training set data. The reason for the training set data being so small is not due to a bas split of the dataset, but rather generally lacking observation. Here it should be noted that the data that was collected by our group was lost two times, with the first time being right after the finishing of the first milestone, and the second loss of pushed observations to the platform happened at the beginning of June.

Because of the explained difficulty, we had to find a workaround in order to see how would our model perform on data on which it was not trained on. We came to an idea that as the model needs at least 4*24*7 observation to make a single prediction, we expanded our test set data with the last 4*24*7 observations from the training set. As the model is not going to make a prediction for any of these observations, by adding these observations we did not harm the prediction procedure, moreover, by doing so we made it able for our model to perform predictions on the test set data.

On the Figure 5 you can see how our LSTM model performed on the test set data.

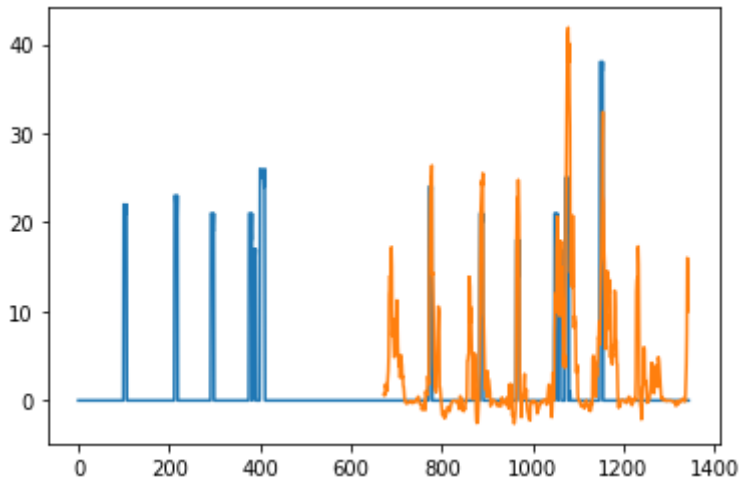

Figure 5: Predictions made on the test set data by the LSTM model, which was expanded with 4*24*7 observations from the training set data

3.3 PROPHET prediction model

The third prediction model which we chose to build for this task is the Facebook's PROPHET model, and the code for this model can be examined by looking at the "*Section 2.4.11.*" in the script.

A thing which is specific for PRPOPHET model is that the input to the model's function "fit" and "predict" strictly have to have two columns with specific names, "*ds*" (which contains time series) and "*y*" (which contains observation). So, the first step which we did, we adapted the training set data, as well as the test set data to these requirements, so we can use them with the model.

As discussed in the previous parts of this report, in the preprocessing steps of our dataset we concluded that we have weekly seasonality in our dataset, which is why we set "*weekly_seasonality=True*", but in the same phase of examining the dataset we realized that we did not have any other kind of seasonality, which is why we set attribute of "Prophet" function as "*yearly_seasonality=False*".

In addition, we set an attribute as "*interval_width=0.95*", with which we specified the width of the uncertainty interval when making predictions.

With parameter "*growth='logistic'*" we are specifying that we have a trend inside of our data which is logistic, rather than the default set liner trend. Now, as we specified the trend to be logistic inside our data, we must specify the lower-most value that the model can take, as well as the upper-most value the model can take. We make sure that this requirement is met by specifying columns "*cap*" and "*floor*" of the data that we are using for building the model, to which we assign a constant value. This is also done for the test set data in the later part of the code.

3.3.1. Our PROPHET model performing predictions on training set data

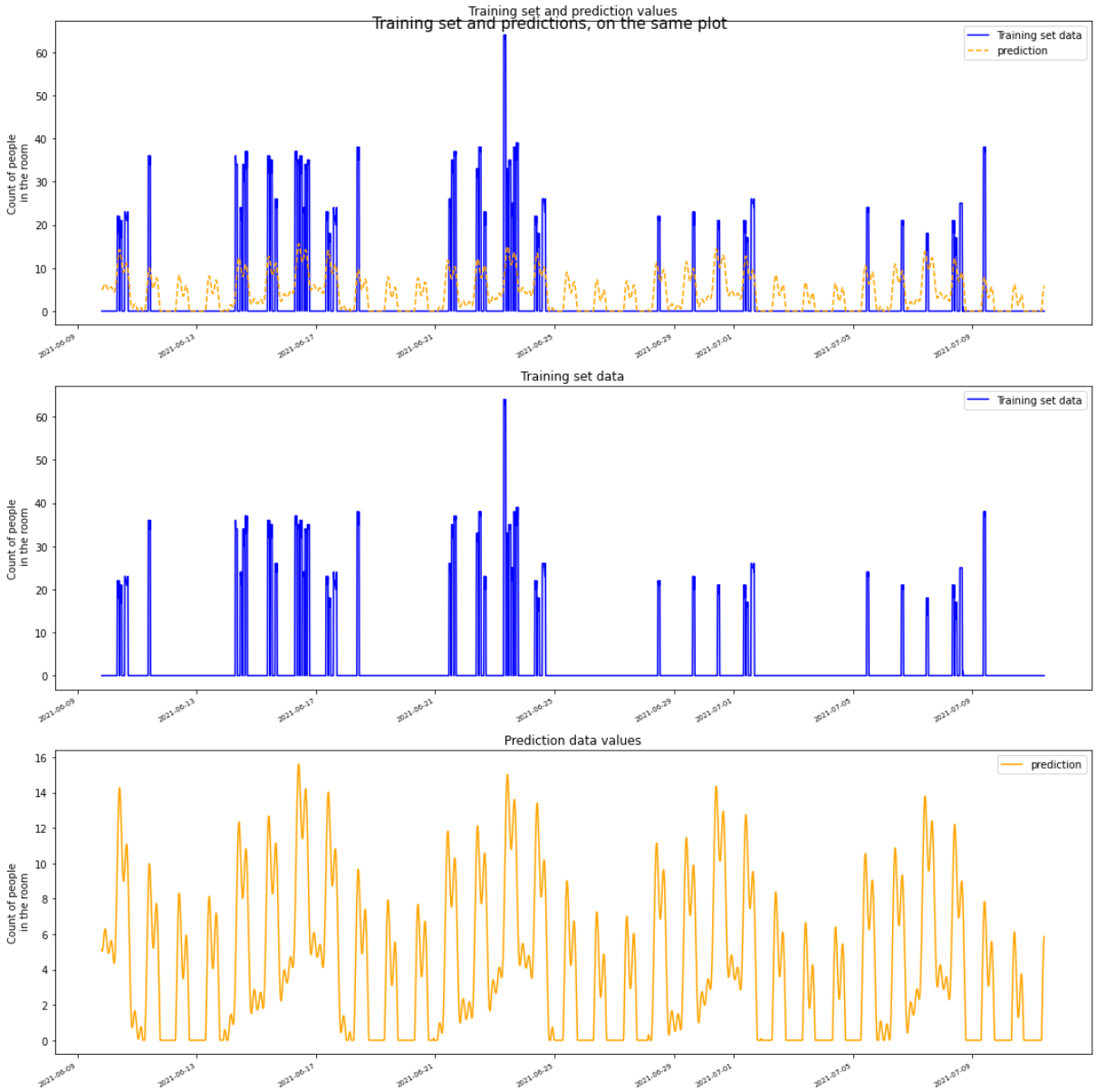

Figure 6: Three plots in which the performance of our PROPHET model is shown, where we are making predictions for the data on which the model was trained on. The first plot is a plot in which we can see both the training set data and the predictions that the model made, while on the second and third plot we can see separately the training set data and predictions made by the model, respectively.

In *Figure 6* you can see the predictions that our PROPHET models make on the data on which it was trained on. From the plots we can see that the model did definitely not overfit to the training set data, but as well that it was not making very good predictions for the test set data.

Accuracy score of our PROPHET model is very low (which can be seen from the screenshot below) as it is only around 14% of samples that are correctly predicted.

```
from sklearn.metrics import accuracy_score

print("Precentage of correctly classified samples is: " + str(accuracy_score
```

```
Precentage of correctly classified samples is: 13.744232036914964%
```

3.3.2. Our PROPHET model performing predictions on test set data

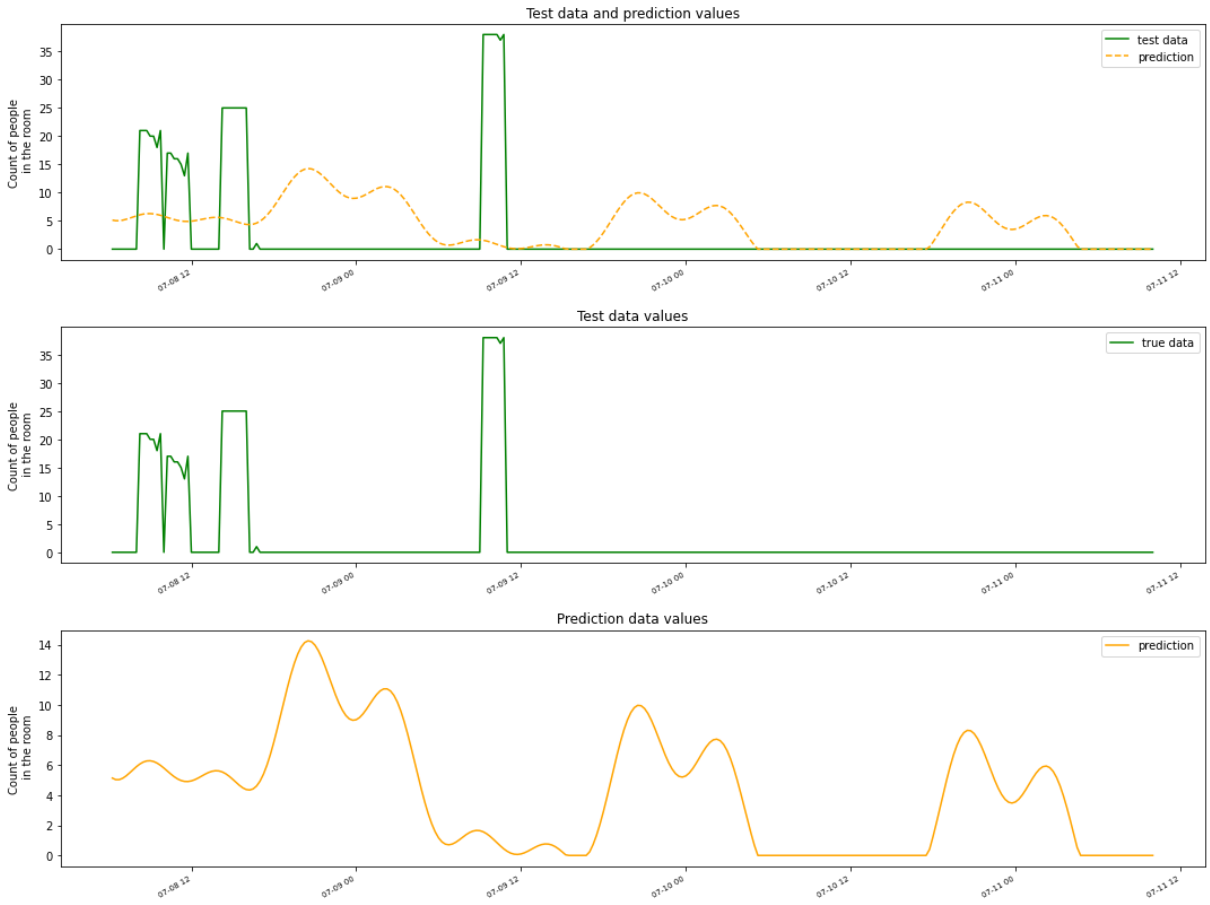

Figure 7: Three plots in which the performance of our PROPHET model is shown, where we are making predictions for the data on which the model was trained on. The first plot is a plot in which we can see both the training data and the predictions that the model made, while on the second and third plot we can see separately the training set data and predictions made by the model, respectively.

On *Figure 7* we can observe that the PROPHET model that we built doesn't show the best results on the test set data either, but better then the data that the ARIMA model with specified "*seasonal_order*" does.

4. Conclusion

To finally conclude, by performing analysis of the performance of the three models that we chose to present in this report, we realized that the LSTM model was by far showing the best results, which is why the LSTM model is the model that we would single out as the best model that we made. The ARIMA model was showing good results on the data on which it was trained one, but had very bad results when used on unseen data. If we used a SARIMAX model as the one which was described in the report, we would surely achieve much better results, but because of the issue of not having a strong enough machine to build our model on it, we were not able to train such a model. Lastly, the PROPHET model showed not so encouraging results when used on the data on which it was trained on, but when using it on unseen data, it showed better results than the ARIMA model that we built.