Final Report
Technische Universität Berlin
Fakultät IV
Smart Communication Systems
DAI-Labor

Submission Topic:
Predictive QoS for Connected Automated Mobility

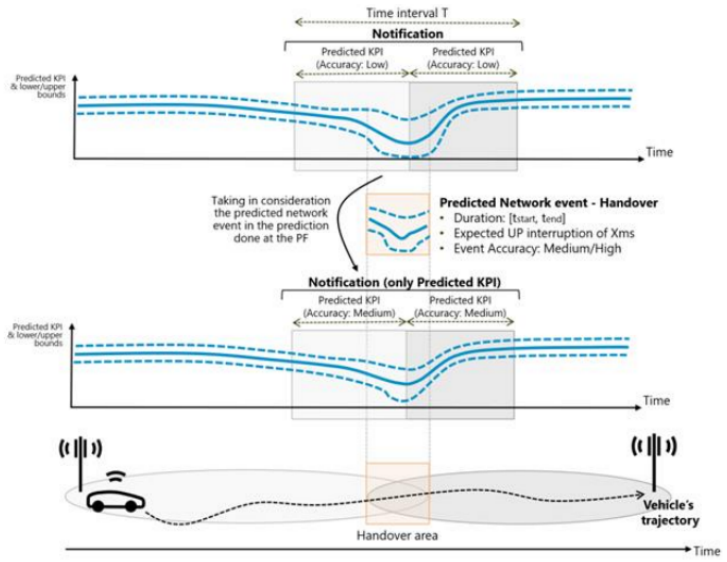
Onur Sen, Louis Andree, Yeji Shim
28.03.2023

DAI-Labor
TU Berlin

Table of Contents

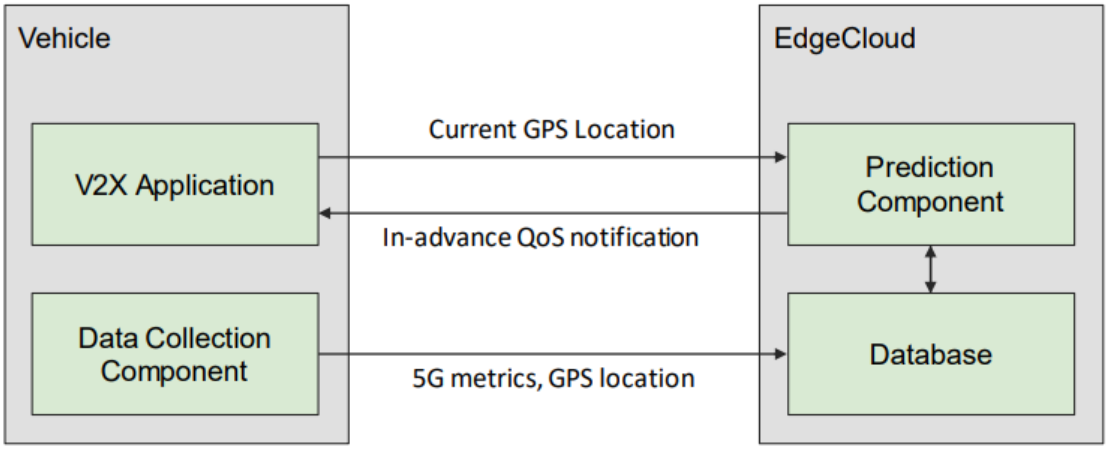
1. Introduction

Predictive QoS for CAM is aiming at supporting the advanced driving use cases by optimizing use of 5G networks to attain best connectivity for in vehicle CAM applications.



Predictive QoS is a mechanism that enables the mobile network to provide in-advance notifications about predicted QoS changes to interested consumers in order to take a proper action _before_ the predicted QoS Changes actually effects. Our objective is to write a program that is capable of making these **QoS predictions with in-advanced-notifications.

Our architecture for this project look like :



2. Database

After setting up Influx and defining the organization, you need to login to your Database (either via Cloud or locally).

The modem script helped us generate sample data at first. It gave us an overall idea of what the data looks like. But generating data for our predictions was not possible, since the data only had 20 different GPS locations (which were all located on the “straße des 17. Junis”). Nonetheless we generated 400 data points and sent them to Influxdb.

In order to change the credentials of your database, you can head to “\topic2-qos4cam\data-collection\src\settings.py” and change the url, org, token, bucket and series Name to your liking. The organization should be the same one you defined when registering to InfluxDB. The token will be generated by influxDB once you create a new bucket, which is required to send and receive data from the database. Note that there are two ways to work with the script, either by having an actual modem connect or by using sample data, which you can change by setting the methods “use_random_data” to either true or false.

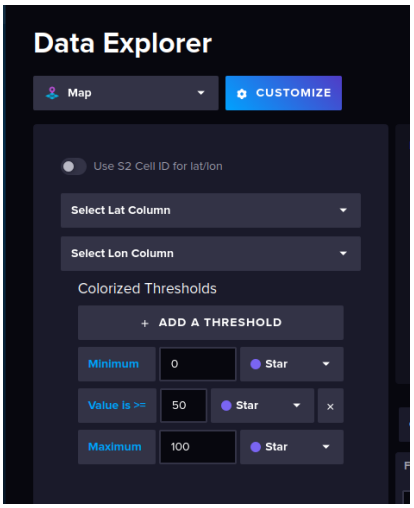
At ”\topic2-qos4cam\data-collection\src\Client.py” we defined the client and sent our data over to the database. This is all done via an api and other methods which Influx provides.

We send our data via a point data type and define each attitude we have measured from our sample files.

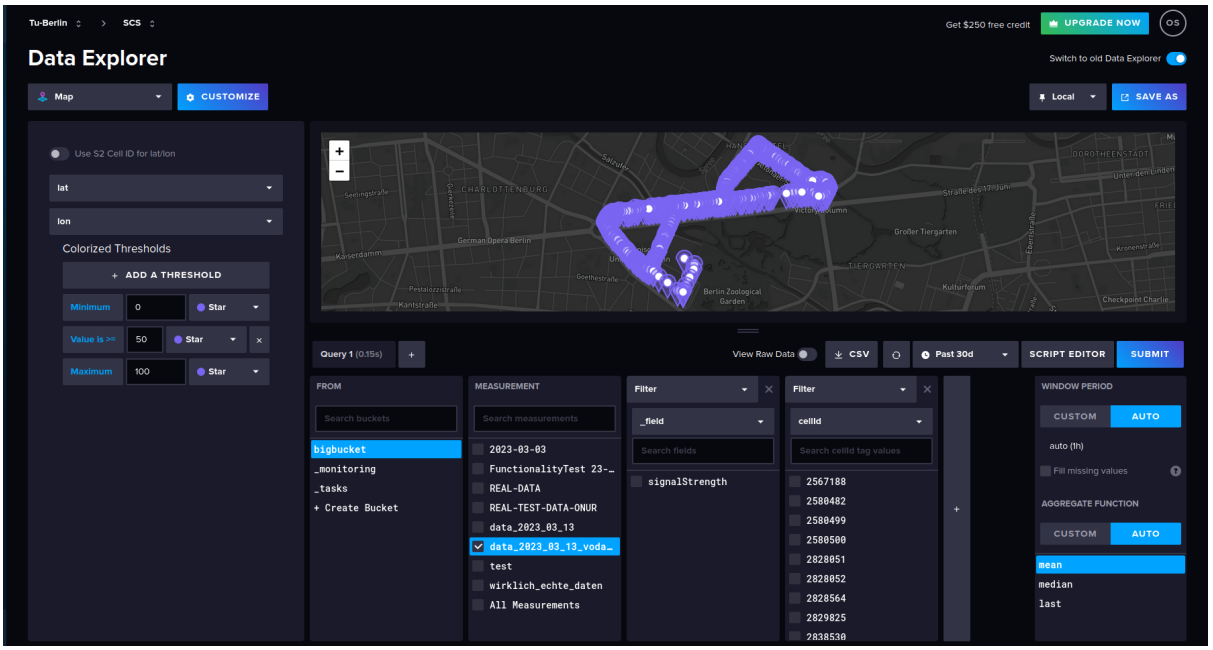
When defining the Point, you can choose between a “tag” and a “field”. In order to read the data in influx as GPS, you need to set latitude and longitude as fields. The difference with tags and fields is that tags are only metadata while fields are there to apply filters on.
In order to receive the data from InfluxDB, we used an API to gather the data directly.

You can view the code for our data receiver here. This generates a new CSV file, which you

can use to push into the prediction components. You can plot the latitude and longitude with the Cloud version and see the dots displayed on the map. To do that you first need to deactivate “Use S2 Cell Id for lat and lon” and set your lat long in the boxes below.



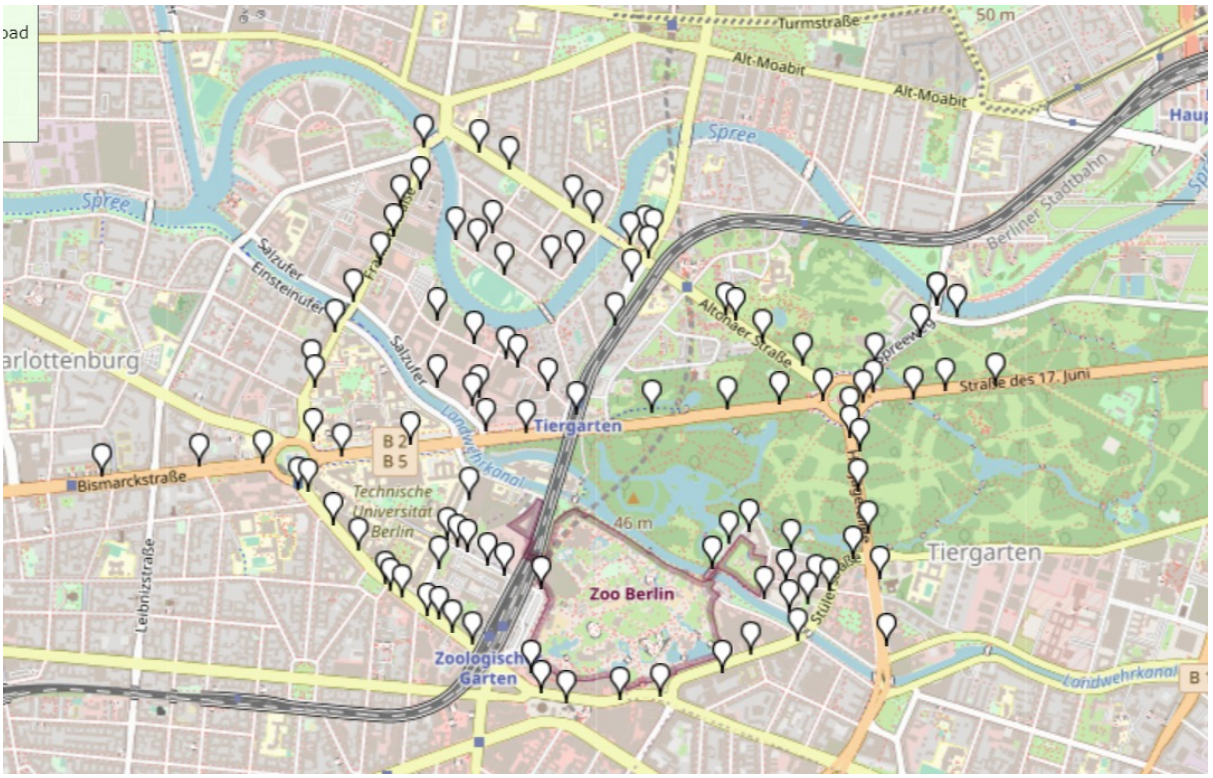
After you have done that, you should be able to see your GPS dots displayed on the map like that..



We were originally thinking about combining latitude and longitude as a S2 Cell id, which decomposes the latitude and longitude and creates a cell which describes the sphere as a hierarchy of cells. For more Info you can read this.

But since InfluxDB had issues recognizing these cells from our data and because we did not know how Include these types of data for our future predictions, we ditched this idea and continued with the old lat, lon approach.

After running our predictions on the sample data (from the script), we realized that the coverage area to make predictions was very small, since we only trained with GPS points located on one street. We then decided to create our own points and give each of those a random signal strength, latency, uplink and downlink (cell id was not included since they were still defined as 999999)



You can find the GPS file here.

Before inserting the data into our prediction component, we need to normalize the data first to prevent our model from overfitting/generating bad predictions.You can read this paper in order to familiarize yourself more with the concept of normalization.

To put it simply, we need to get rid of extreme values that might distort our data and also rewrite certain attributes to make it more “machine friendly”. In our prediction function, we have defined such a function that reads the different cell id’s (sorted by time) and then maps the different cell id’s from 10 digit id’s to 1,2,3,4 etc. So instead of reading the cell 321315132, we can map it as let’s say 1 and continue on with all different cells. Every time we get a new cell, we simply add it to our map and increment its id.

NOTE this can only work if you filter the data by time, since otherwise it would map cells that are apart from each other as neighbors. This should not happen, since we also want to know if a handover occurred between two cells. Having handovers between cells that are not next to each other would falsify our data and hence worsen our prediction. The switch is

determined by comparing the current cell id with the previous one. If the id is different, we set the switch for the data point to 1 and if it hasn't changed, we write it as 0.

We hope to find correlations between high latencies and the handover area, since most of the time, drop of qualities occur in these areas.

3. Cell Prediction

Since all we receive from the vehicle is its latitude and longitude, we need to predict which cell it is currently using. We used three different methods to predict the location

1 Neural Networks

2. Cluster analysis

3. Decision tree

After experimenting with all three, we quickly came to the conclusion that the decision tree based Model performed the best out of all and was the simplest to implement. We used the scikit-learn library to implement our tree algorithm.

We also ran into an Issue with this approach.

While looking at the output of our predicted cells, we saw that the first cells were guessed correctly but over the last entry points, they all were completely wrong. We quickly found out that this was because each base station has different cells integrated with different amounts. Our decision tree is only able to predict which cell was used based on the old information. But we did not take into account that other cells might connect at the same location over time (due to overload for instance).

To solve this, we could generalize all cells as base stations, but decided not to because this cell switch might be an important indicator for overload and can also be fed into the neural network. Since this would span our project, we skipped that part but wanted to emphasize on its usefulness for future latency predictions.

Since our downloaded InfluxDB csv was not filtered based on time, we had to adjust our files first (starting from oldest to newest) so that the decision tree functions properly. The unfiltered data looks like this and the filtered one like this.

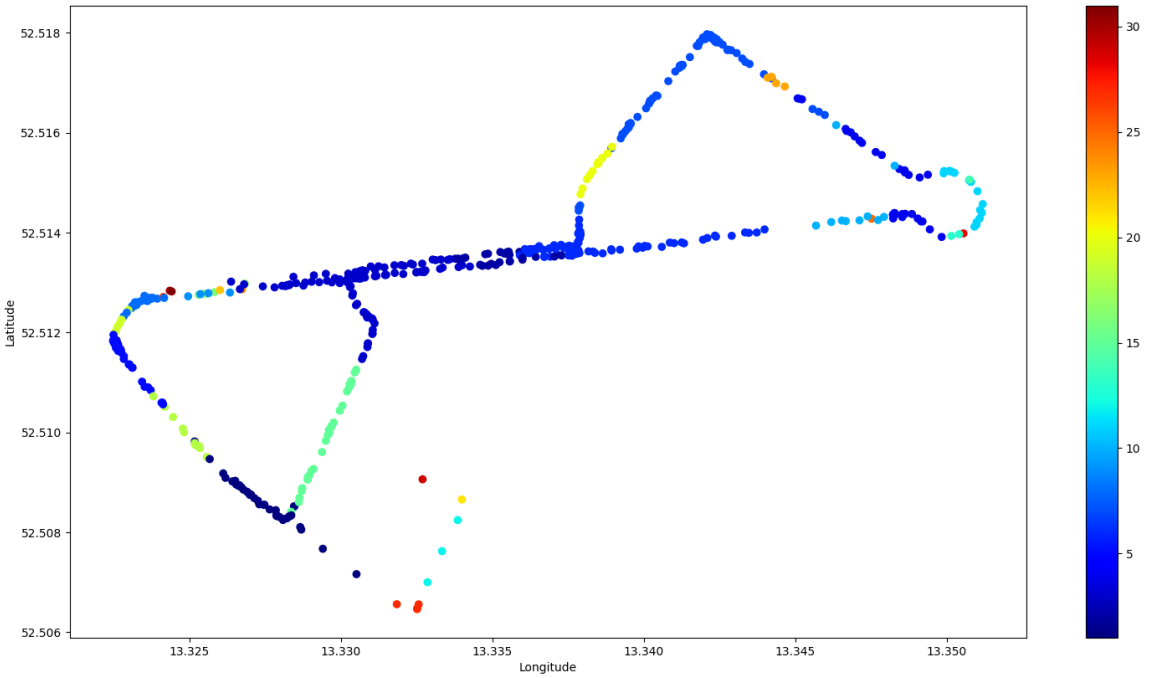
Additionally, we also need to take different network providers into consideration and their base stations. This makes cell prediction for all possible providers very complicated, since Telekom has their own base stations (located at different spots) with their own cells, which is different to the ones that vodafone uses.

Since our Telekom card had expired while collecting data, we switched to a vodafone sim card which subsequently produced different data to the one that we collected before. Due to these differences, we were not able to merge our old data with the new one or use them to test the accuracy of our prediction component. For that reason, we are only able to make predictions based on one provider only and cannot take other ones into consideration.

Unifying all these different base stations and cells could be a step forward towards better understanding overall network traffic and recognizing high latency patterns for all users.

For more info.

When we plot our lat, lon and cell id



4. Cell Prediction Component

We decided to make predictions on uplink, downlink and latency in 2 ways and compare with the prediction results.

1. Neural Network
2. Linear Regression

4.1 Keras Neural Network

While talking with Umuralp in regards to our prediction function, he recommended to us the

usage of keras for our neural networks, since it is very easy to implement and work with. For that we had to import Keras and Tensorflow libraries.

The most important parts in the NN were the layers, the hyperparameters, learning rates and batch size. With these parameters, we were able to define NN architecture and adjust it according to the data that you feed it with. NOTE Neural network predictions are only as good as your data is, which means that by adding more data into the training model, you need to change the architecture alongside to match their accuracy.

There is no real way to find out which architecture is the best for a certain case. In order to find the best predictions, you need to play around the layers, batch size, learning rates and watch out for overfitting at the same time. An example of overfitting can be seen here.

The inputs for our Neural Network are, the latitude, longitude, cell ID and the switch.

Why did we not include more you might ask? Since the car is only sending us its latitude and longitude, we need to predict the other inputs based on that and feed them into the neural network. We managed to predict the cellID with the help of a decision tree and the cell switch with regression. But when it comes to the SignalStrength for instance, it becomes very hard to predict it based only on the GPS location. We ran an experiment by trying to predict the Strength and StrengthLTE with a neural network, but the outcome was very bad. We therefore did not include any signals into our prediction as well as the time, since this would negatively impact our model. Frequency is also a component, which seemed complex to predict, which is why we also did not consider it as an input.

Nonetheless we wrote a neural network, which would predict the signal strength with latitude, longitude, cellId and switch. You can view our code here. This could work however if we would gather much more data (over several days) and feed everything into our model.

Implementing time as an input for neural networks on the other hand can be complex, since it depends on the specific use case and the nature of the time series data.

If the time series data has a regular time interval between each sample, such as hourly or daily data, then it can be relatively straightforward to include time as an input feature. Normalizing timestamps would be easy for this cause, however our data has irregular time distances, which makes it hard to normalize and therefore complex to include in the neural network. We also spoke with Umuralp in that regard and he agreed and told us we should Ignore the time as an input for the neural network.

That leaves us with only the latitude, longitude, cellId and the switch as inputs.

Model:

```
target = data[['uplink', 'downlink', 'latency']]
features = data[['lat', 'lon', 'cellId', 'switch']]

model = keras.Sequential([
    layers.BatchNormalization(input_shape=(4,)),
    layers.Dense(300, activation='relu', kernel_regularizer=keras.regularizers.l2(0.0005)),
    layers.Dropout(0.2),
    layers.BatchNormalization(),
    layers.Dense(160, activation='relu', kernel_regularizer=keras.regularizers.l2(0.0005)),
    layers.Dropout(0.2),
    layers.Dense(100, activation='relu', kernel_regularizer=keras.regularizers.l2(0.0005)),
    layers.Dropout(0.2),
    layers.BatchNormalization(),
    layers.Dense(100, activation='relu', kernel_regularizer=keras.regularizers.l2(0.0005)),
    layers.Dropout(0.2),
    layers.Dense(3),
    layers.Dense(150, activation='relu', kernel_regularizer=keras.regularizers.l2(0.002)),
    layers.Dropout(0.2),
    layers.BatchNormalization(),
    layers.Dense(50, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)),
    layers.Dropout(0.2),
    layers.Dense(3)
])
```

This is our Neural network architecture, with the above mentioned targets (what we want to predict) and features (what we feed it with). The input layer has a shape of (4,), which means that it takes in a feature vector of 4 dimensions as input (lat,lon,cellId,switch). The first layer after the input layer is a Batch Normalization layer(which helps in normalizing the inputs and improves speed and performance).

The next 6 layers have different amounts of neurons starting from 300 to 3. The reduction helps with overall performance and prevents overfitting. We use ReLu as our activation function, since the Rectified Linear Unit activation is a popular choice for many neural network architectures.

Additionally, we added a Dropout layer that randomly drops out 20% of the neurons during training. This can help in reducing overfitting by forcing the remaining neurons to learn more robust representations of the data.

As mentioned before, this architecture was designed randomly over time to adjust it towards our desired predictions.

For more info, you can read this article to understand more about the architecture for Neural Networks, since there is much more to talk about.

4.2 Linear Regression

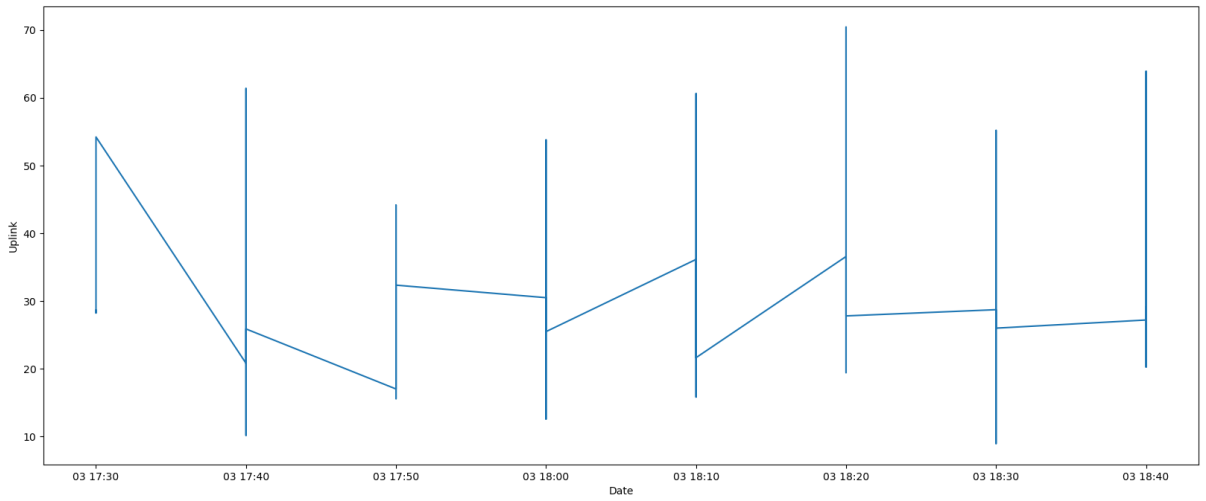
According to our decision to make predictions in 2 ways, which are machine-learning method as well as deep-learning method for more precise prediction results, we had chosen **ARIMA Model** as a machine learning method at first.

We implemented following libraries to build the ARIMA :

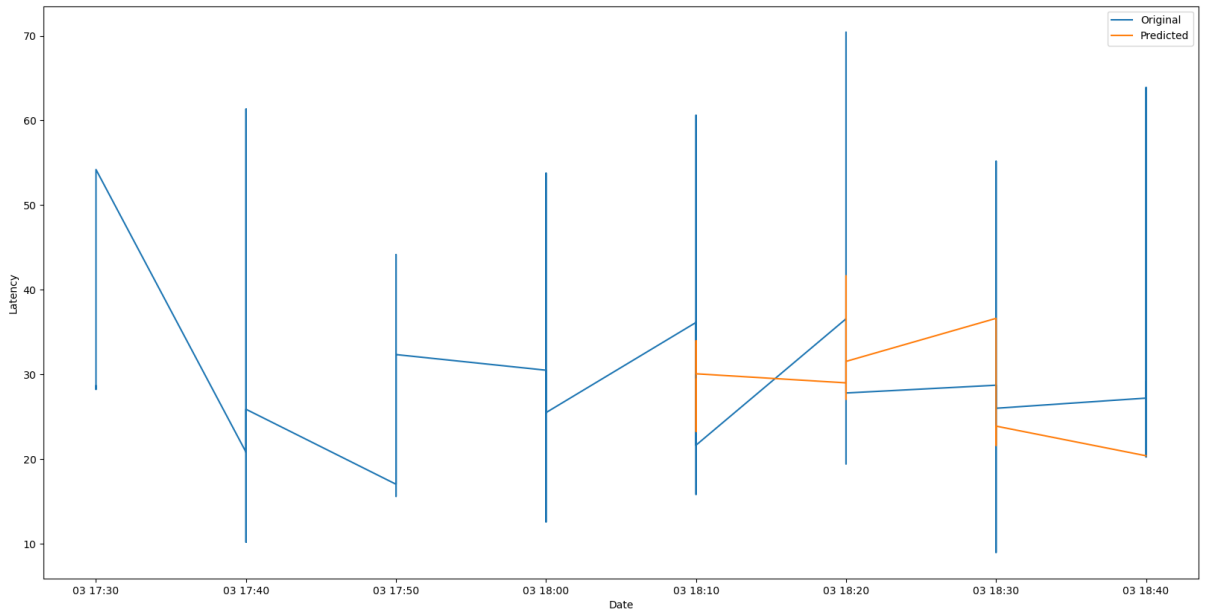
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

from pylab import rcParams
import statsmodels.api as sm
import warnings
import itertools
from statsmodels.tsa.ar_model import AR
from statsmodels.tsa.arima_model import ARIMA
```

In the process of building the ARIMA model, we created a graph of a time-series dataset to figure out how our data flow in time-order.

The unnormalzied data graph looks like: unnormalized_data_graph



When we make a prediction, for example, on latency with this data, a graph of predicted and original data looks like : latency prediction without normalization



This graph shows unorganized and bad-fitted flows, so that we had determined to normalize the data.

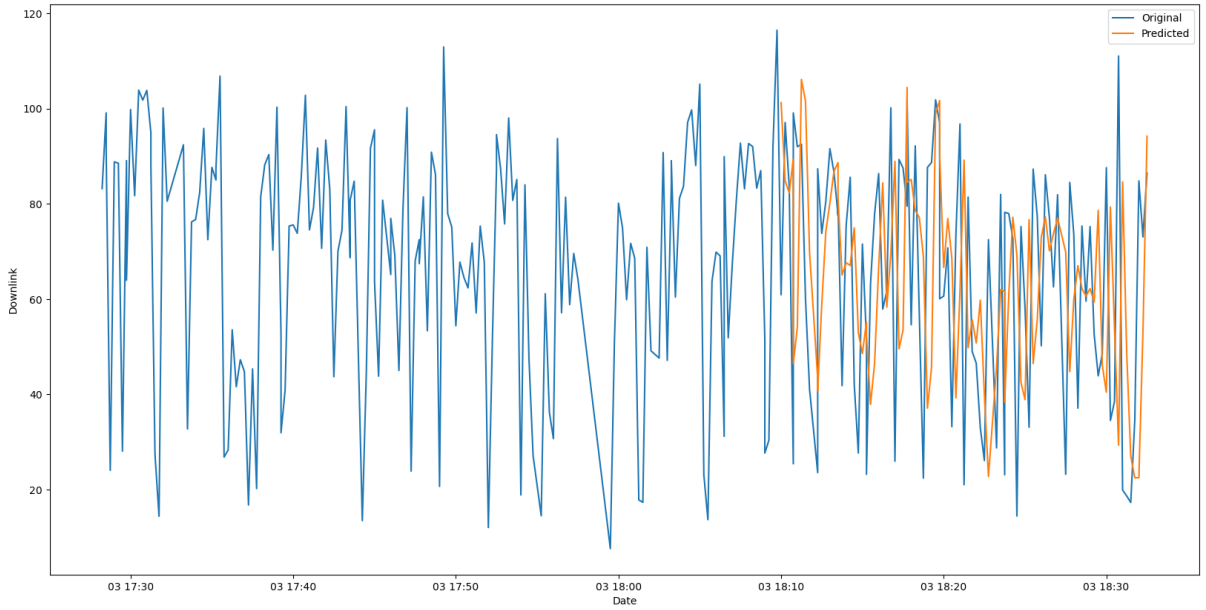
The data-normalization is the practice of organizing data entries to ensure they appear similar across all fields and records, making information easier to find, group and analyze.

Our codes for data-normalization look like :

```
data = data[data['_value'] != -32768]
data = data[data['frequency'] != -9999]
data = data[data['lat'] != -9999]
data = data[data['lon'] != -9999]
data = data[data['uplink'] != -1]
data = data[data['downlink'] != -1]
data = data[data['latency'] != -1]
data = data[data['cellId'] != -9999]
```

Now that the data is organized through the normalization, the ARIMA model was well-trained and the predictions look also well-forecasted : latency prediction after normalization.

For example, This graph below is a prediction result of downlink with the ARIMA model. The blue flow indicates original data and the orange one means the predicted downlink on the test data.



In addition, we measured the prediction performance of different algorithms (MAE, MSE, RMSE, R2, MAPE) in order to check how well the prediction is proceeded with these codes :

```
from sklearn import metrics

def mae(y_true, y_pred):
    return metrics.mean_absolute_error(y_true,y_pred) #MAE
def mse(y_true, y_pred):
    return metrics.mean_squared_error(y_true,y_pred) # MSE
def rmse(y_true, y_pred):
    return np.sqrt(metrics.mean_squared_error(y_true,y_pred))  # RMSE
def r2(y_true, y_pred):
    return metrics.r2_score(y_true,y_pred) # R2
def mape(y_true, y_pred):
    return np.mean(np.abs((y_pred - y_true) / y_true)) * 100 # MAPE

def get_score( y_true, y_pred):

    mae_val = mae(y_true, y_pred)
    mse_val = mse(y_true, y_pred)
    rmse_val = rmse(y_true, y_pred)
    r2_val = r2(y_true, y_pred)
    mape_val = mape(y_true, y_pred)

    score_dict = {
                "mae" :  mae_val,
                "mse" :  mse_val,
                "rmse" : rmse_val,
                "r2":    r2_val,
                "mape" : mape_val
               }
    return score_dict
```

```
get_score(np.array(test1), np.array(forecast) )
```

As results of measuring prediction performances with ARIMA, we found out the prediction performance is quite accurate.

However, we figured out that the predictions are made with this model only based on time-series, without regarding any important values (Longitude, Latitude, CellId, CellSwitch), which would have an effect on the QoS in reality.

Here is a detailed process and result with ARIMA .

As a solution to this issue, we chose to make predictions with the **VAR (Vector Autoregression) model** and took the values *(longitude, latitude, cellId, and switch)* into account.

We implemented following libraries to build the VAR :

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

from pylab import rcParams
import statsmodels.api as sm
import warnings
import itertools

from statsmodels.tsa.api import VAR
from statsmodels.tsa.stattools import adfuller
```

For better prediction with VAR model, we checked stationarity of dataset using ADF(Advanced Dickey-Fuller test) and differencing.

The codes look like:

```
#ADF(Advanced Dickey-Fuller test)

from statsmodels.tsa.stattools import adfuller
adfuller_test = adfuller(mydata['lon'], autolag= "AIC")
print("ADF test statistic: {}".format(adfuller_test[0]))
print("p-value: {}".format(adfuller_test[1]))

#Differencing by order 1
```

```
mydata_diff = mydata.diff().dropna()
```

In order to fit VAR Model, we found an appropriate order with AIC(Akaike's Information Criterion).

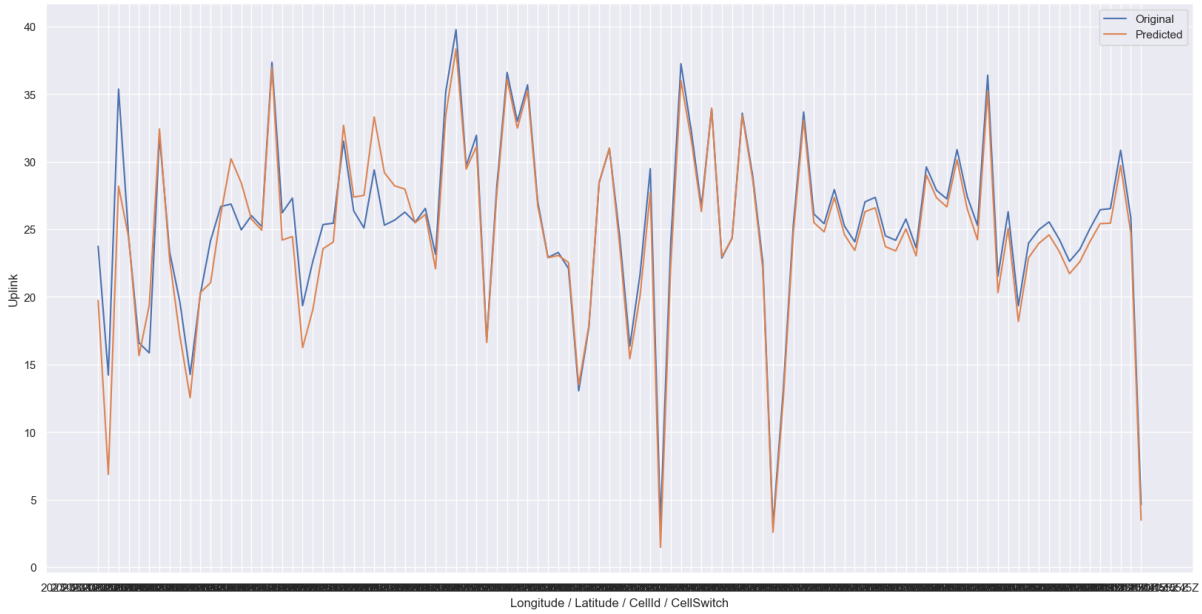
```
from statsmodels.tsa.api import VAR

from statsmodels.tsa.stattools import adfuller


forecasting_model = VAR(train)

results_aic = []

for p in range(1,30):

  results = forecasting_model.fit(p)

  results_aic.append(results.aic)
```

After finding a right lag order for each target, we fitted a VAR model with the lag order and made predictions on the test data.

Then, we measured prediction performance of different algorithms (MAE, MSE, RMSE, R2, MAPE) through the codes like the way in the ARIMA model.

Here is the detailed process and prediction results with the VAR Model.

The results seem to be precise and almost perfectly trained. For example, you can check a graph of the prediction on uplink here.



As you can see from the graph above, the prediction result is extremely accurate and the model is perfectly trained.

But, we came across another issue.
In our project, we aim to realize new predictions on the future of target variables (Uplink, Downlink, Latency) using the trained model. Also, the new data we will use for new prediction includes only four feature variables*(longitude, latitude, cellId, and switch)* and does not include target variables.
For this reason, we should have defined feature- and target-variables respectively.

However, the problem is :

In a VAR model, all variables are considered as both predictors and targets.

Therefore, there is no clear distinction between feature variables and target variables.

When we fit the VAR model, we should specify all the variables we want to include in the model, which means, for example, if we want to predict latency based on the inputs of longitude, latitude, cellId and switch, we should then include all five variables in the VAR model.

Because the feature- and target-variables are defined all together in the beginning and the model is trained based on it, we could not use the trained model for new prediction.

As a solution of the second issue with the VAR Model, we finally decided to change a model to a **Linear Regression.**

Linear regression is the most commonly used method of predictive analysis.

It uses linear relationships between a dependent variable (target) and one or more independent variables (predictors) to predict the future of the target.

Therefore, by using Linear regression, we were able to predict the target variables (latency, uplink and downlink) based on the feature variables (longitude, latitude, cellId and switch).

In addition, once we have trained the regression model, we made new predictions using the trained linear regression model on new data, which includes longitude, latitude, cellId, and switch values but does not include the latency, uplink, and downlink values, as already mentioned.

The Following is how we set up the linear regression model and the codes work :

1. We imported our dataset and defined feature- and target variables.

```
data = pd.read_csv('data_file.csv')

X = data[['longitude', 'latitude', 'celled', 'switch']] # Feature variables

y_lat = data['latency'] # Target variable - latency

y_ul = data['uplink'] # Target variable - uplink

y_dl = data['downlink'] # Target variable - downlink
```

2. We splitted train and test data of feature variables and each target variable using the '*train_test_split' library.*

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_lat_train, y_lat_test, y_ul_train, y_ul_test, y_dl_train, y_dl_test =
train_test_split(X, y_lat, y_ul, y_dl, test_size=0.2, random_state=42)
```

3. We fitted a linear regression model to predict target variables and made predictions on the test set using the *LinearRegression library* .

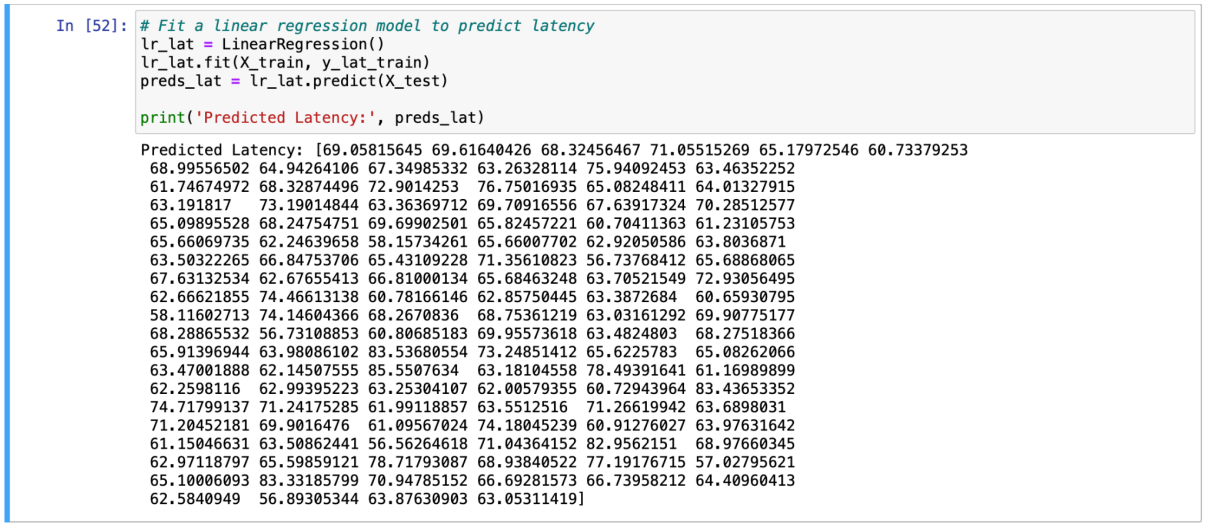
```
from sklearn.linear_model import LinearRegression

# codes for latency prediction
lr_lat = LinearRegression()
lr_lat.fit(X_train, y_lat_train)
preds_lat = lr_lat.predict(X_test)

# codes for uplink prediction
lr_ul = LinearRegression()
lr_ul.fit(X_train, y_ul_train)
preds_ul = lr_ul.predict(X_test)

#codes for downlink prediction
lr_dl = LinearRegression()
lr_dl.fit(X_train, y_dl_train)
preds_dl = lr_dl.predict(X_test)
```

4. We used the fitted regression equation to predict the future of target variables based on new data.

```
# Load and prepare new data
new_data = pd.read_csv('your_new_data_file.csv')
X_new = new_data[['longitude', 'latitude', 'celled', 'switch']]  # Feature variables

# Make predictions on the new data to predict latency
new_preds_lat = lr_lat.predict(X_new)

# Make predictions on the new data to predict uplink
new_preds_ul = lr_ul.predict(X_new)

# Make predictions on the new data to predict downlink
new_preds_dl = lr_dl.predict(X_new)
```

Through this process we obtained the prediction results of latency, uplink and downlink.

For example, the predicted latency on the test set :



In [52]:
```
# Fit a linear regression model to predict latency
lr_lat = LinearRegression()
lr_lat.fit(X_train, y_lat_train)
preds_lat = lr_lat.predict(X_test)

print('Predicted Latency:', preds_lat)
```

```
Predicted Latency: [69.05815645 69.61640426 68.32456467 71.05515269 65.17972546 60.73379253
 68.99556502 64.94264106 67.34985332 63.26328114 75.94092453 63.46352252
 61.74674972 68.32874496 72.9014253  76.75016935 65.08248411 64.01327915
 63.191817   73.19014844 63.36369712 69.70916556 67.63917324 70.28512577
 65.09895528 68.24754751 69.69902501 65.82457221 60.70411363 61.23105753
 65.66069735 62.24639658 58.15734261 65.66007702 62.92050586 63.8036871
 63.50322265 66.84753706 65.43109228 71.35610823 56.73768412 65.68868065
 67.63132534 62.67655413 66.81000134 65.68463248 63.70521549 72.93056495
 62.66621855 74.46613138 60.78166146 62.85750445 63.3872684  60.65930795
 58.11602713 74.14604366 68.2670836  68.75361219 63.03161292 69.90775177
 68.28865532 56.73108853 60.80685183 69.95573618 63.4824803  68.27518366
 65.91396944 63.98086102 83.53680554 73.24851412 65.6225783  65.08262066
 63.47001888 62.14507555 85.5507634  63.18104558 78.49391641 61.16989899
 62.2598116  62.99395223 63.25304107 62.00579355 60.72943964 83.43653352
 74.71799137 71.24175285 61.99118857 63.5512516  71.26619942 63.6898031
 71.20452181 69.9016476  61.09567024 74.18045239 60.91276027 63.97631642
 61.15046631 63.50862441 56.56264618 71.04364152 82.9562151  68.97660345
 62.97118797 65.59859121 78.71793087 68.93840522 77.19176715 57.02795621
 65.10006093 83.33185799 70.94785152 66.69281573 66.73958212 64.40960413
 62.5840949  56.89305344 63.87630903 63.05311419]
```

For example, the new predicted latency on the new dataset using the trained model :

```python
# Make predictions on the new data to predict latency
new_preds_lat = lr_lat.predict(X_new)

# Make predictions on the new data to predict uplink
new_preds_ul = lr_ul.predict(X_new)

# Make predictions on the new data to predict downlink
new_preds_dl = lr_dl.predict(X_new)

print('Predicted Latency:', new_preds_lat)
print('Predicted Uplink:', new_preds_ul)
print('Predicted Downlink:', new_preds_dl)
```

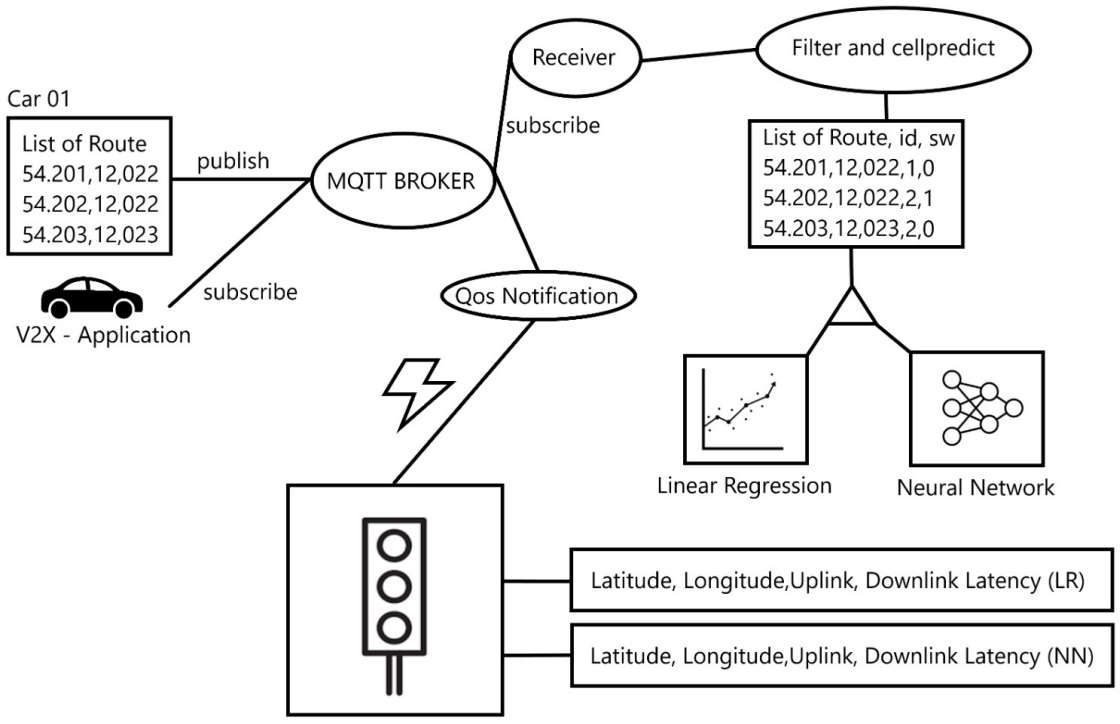
```
Predicted Latency: [49.81236598 49.96267275 58.68291637 51.2964335  51.51183287 60.77328956
 62.45330287 64.36809043 57.42646169 58.03405131 67.38089488 60.7395272
 70.07777611 63.36453992]
Predicted Uplink: [25.64417101 25.6889187  21.4131247  25.81926595 25.83200442 21.33990541
 21.20520285 20.98432397 25.16559057 25.0053974  20.44531821 24.5461998
 19.99167088 24.11129363]
Predicted Downlink: [123.43405295 121.55507077  96.6670032  117.10774888 115.33042899
  90.39989288  88.18169593  85.80871273 107.0532177  105.53344996
  81.19926001 101.71445417  77.34325896  98.0419097 ]
```

You also can check the predicted values of latency, uplink and downlink as a graph.

Based on the prediction results with LR and NN, we pushed the data to the Edge Cloud.

5. Edge Cloud

To give you an overall idea how our edge cloud works, or more specifically, how predictions are made, you can take a look at this.



The vehicle is sending us a route, where it is going to drive through in the next 10 seconds for example. The exchange between the vehicle and the cloud happens via the MQTT protocol. The car publishes his route and our Edge cloud clients subscribe to his topic. This route will be defined as a list, which contains latitude and longitude for each position on the route (sorted ofcourse). We take this route list and predict it's cell id and switch

List before filter, List after filter

```
lat,lon,cellId
52.5126298,13.3235621,7
52.5126951,13.3250856,0
52.5128518,13.3269739,0
52.5130216,13.3289266,0
52.5130085,13.3301497,0
52.5126168,13.3304715,0
52.5123295,13.3309436,0
52.5118202,13.3308792,0
52.5114415,13.3306646,0
52.5109974,13.3304501,0
52.5104489,13.3298922,0
52.5098482,13.3295488,0
52.5093128,13.3290553,0
52.5087642,13.3287334,0
```

```
lat,lon,cellId,switch
52.5126298,13.3235621,8,0
52.5126951,13.3250856,8,0
52.5128518,13.3269739,17,1
52.5130216,13.3289266,3,1
52.5130085,13.3301497,3,0
52.5126168,13.3304715,3,0
52.5123295,13.3309436,3,0
52.5118202,13.3308792,3,0
52.5114415,13.3306646,3,0
52.5109974,13.3304501,15,1
52.5104489,13.3298922,15,0
52.5098482,13.3295488,15,0
52.5093128,13.3290553,15,0
52.5087642,13.3287334,15,0
```

The 1st gps column is the current GPS location and current Cell. For our 2nd picture, we removed the current location, since it is not necessary to include for our predictions.

As you can see, we added a new column switch as a boolean and deleted the first entry of where the car is currently located (since the current location is not necessary to include into our prediction). After we have predicted the cell and the switch, we push this data into both our prediction components (NN, Linear Regression) and predict the uplink, downlink and latency for each position.

	lat	lon	uplink	downlink	latency
0	52.512630	13.323562	26.322571	111.489037	60.123032
1	52.512695	13.325086	26.342554	111.608284	60.193726
2	52.512852	13.326974	20.761669	88.185814	63.990719
3	52.513022	13.328927	24.503815	103.982559	64.311134
4	52.513008	13.330150	26.076946	104.026588	68.420753
5	52.512617	13.330471	26.183887	103.819336	68.921043
6	52.512329	13.330944	26.239344	103.731834	69.286949
7	52.511820	13.330879	26.349218	103.676140	69.991753
8	52.511441	13.330665	26.446703	103.719452	70.547516
9	52.510997	13.330450	22.385237	95.056381	64.080162
10	52.510449	13.329892	27.173832	116.207405	63.010429
11	52.509848	13.329549	27.164970	116.161072	62.982235
12	52.509313	13.329055	27.155466	116.111748	62.952114
13	52.508764	13.328733	27.147173	116.068436	62.925747

As you might have already noticed, we did not include the cell or the switch into our solutions, since that information is totally irrelevant for the car. Based on this scenario, we

can see at the 8th row that the latency is higher than usual (handover occurs in that area), so we can send a warning to the car and notify a reduction in the quality of service for that location.

The same will be done for the linear Regression solution. We take both predictions into account by using a “traffic light” warning concept, which works like this:

Green: Both Predictions show good QoS

Yellow: One of the predictions shows a drop in QoS

Red: Both Predictions show bad QoS

The vehicle will receive a notification based on the “light” that was generated for a certain position and warn the driver for an upcoming reduction of network quality.

6. Data Collection Component

The data collection component is supposed to collect the necessary cellular network data as well as current QoS measurements and current GPS location. You can find it here. In order to connect to the cellular network and collect the data we used a Quectel rm500Q-GL Board. We communicated with the board over a USB using the “*modem manager cli*” The main part of the script was already handed to us. We just had to include our Database and the script for measuring the latency, uplink and downlink. You can see below for further information. The script activates the necessary functions on the EVB and then enters an endless loop where we first run the speedtest then log the 5G and GPS data, combine it to a data point and send it to the database. The entry point to the script is the main.py file. Start it by running:

sudo python3 src/main.py

```
ection$ sudo python3 src/main.py
[sudo] password for louis:
{'latency': 76.58590815000986, 'download': 54.69681540181009, 'upload': 25.97402661494351
5}
FunctionalityTest\ 23-03-23,cellId=99999999,downlink=54.69681540181009,frequency=1880,lat
=52.51355,latency=76.58590815000986,lon=13.33815,networkProvider=Telekom,signalStrengthLT
E=-86,uplink=25.974026614943515 signalStrength=-82i
{"signalStrength": -82, "signalStrengthLTE": -86, "networkProvider": "Telekom", "cellId":
 99999999, "frequency": 35, "lat": 52.51355, "lon": 13.33815, "uplink": 25.97402661494351
5, "downlink": 54.69681540181009, "latency": 76.58590815000986}
{'latency': 76.48271390005927, 'download': 45.45600712345235, 'upload': 26.38522785973413
}
FunctionalityTest\ 23-03-23,cellId=99999999,downlink=45.45600712345235,frequency=1920,lat
=52.5142333333,latency=76.48271390005927,lon=13.34865,networkProvider=Telekom,signalStren
gthLTE=-88,uplink=26.38522785973413 signalStrength=-96i
{"signalStrength": -96, "signalStrengthLTE": -88, "networkProvider": "Telekom", "cellId":
 99999999, "frequency": 37, "lat": 52.5142333333, "lon": 13.34865, "uplink": 26.385227859
73413, "downlink": 45.45600712345235, "latency": 76.48271390005927}
```

You can customize the script in the settings.py file. The variables `token`, *`bucket`*, *`org`*, *`url`* and `seriesName` are for the database. The *measure_rate* is the interval between two measurements in seconds. With `save_to_file` you can determine whether you want to save the data points in a log file. Same for the database with the `save_to_database` option.The modem-script can also be run in dummy mode so the 5G and GPS data is generated randomly and transferred to the Database. To activate the dummy mode set `use_random_data` to True. The option `testSpeed` determines whether the speed test is run.

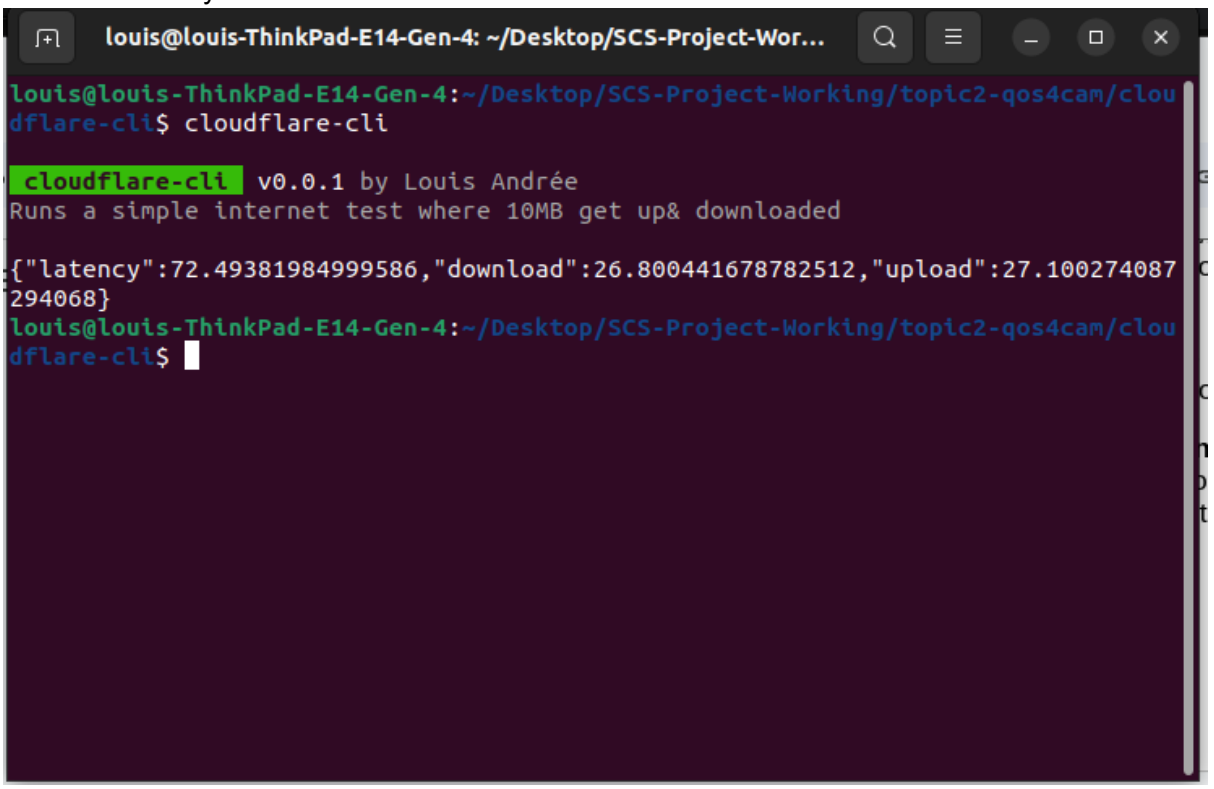
7. cloudflare-cli

To measure the up- & downlink speed we first used the tool speedtest-cli. But we soon realized that the missing configuration possibilities would limit the effective data collection progress. The main problem was that it is not possible to determine the size of the uploaded and downloaded data but it was fixed to ~200MB. Considering that we only had 5GB of fast mobile data we would have a very limited amount of data. Also the down and upload would have taken so long that it would be hard to justify that the result can be assigned to one GPS Point. So our current approach is based on publicly available source code of "speed-cloudflare-cli". We modified the code so it only uploads 5 MB of data. You can find the modified version here

To run the modemscript the cloudflare-cli has to be installed as a Command Line Interface(CLI). To do so follow these steps:

1. **Install nodeJS and npm** In Ubuntu you can run "sudo apt install node"
2. **install cli** Navigate to the downloaded folder in the terminal and run "sudo npm install -g" to install the cli globally on your machine.

3. **Test the cli:** To test if the cli is installed on your machine run the command cloudflare-cli in your terminal



8. V2X-application

This module simulates a car and transmits a route csv file(*Route1.csv*) containing gps points to the **prediction component** over MQTT. The script also subscribes to the MQTT topic where the responses for the request will be published. A response from the Keras model and Linear Regression are awaited and are also csv files. Both predictions are then merged to one big csv and saved in the folder results. The data could now be analyzed and evaluated by the traffic light principle described earlier. For increased security the QoS-level should be 2 and not 0 this needs to be updated in the future. To start the application navigate to the folder *v2x-application* and run

python3 src/car.py

9. Prediction-Component

The prediction component can be found here. The main parts are the server.py and PredictionModule.py for the operation. The server listens for incoming request over MQTT and passes the data to the PredictionModule.py where the trained Linear Regression, TensorFlow Keras and Decision Tree are are integrated to make predictions regarding the

cells the car will be connected to and the QoS. When you run the server.py you will be asked which models you want to load. You need to load a .joblib file for the decision tree, a .h5 keras model and 3 .pkl files for the Linear Regression(one for up-, downlink and latency. When they are set up properly the server is running and can respond to requests. The models you should load are inside the trained models folder.

The models you can load are generated and trained by DecisionTree.py, LinearRegression.py, TensorFlow.py and saved in the sub folders of the trained models folder. How these models are trained is explained in the topics above.

You can start the prediction component by running

python3 server.py

in the Folder of the prediction component.

10. References

Prasad, A. (2021). The Complete Guide to Time Series Analysis and Forecasting. Towards Data Science. Retrieved from

“https://towardsdatascience.com/the-complete-guide-to-time-series-analysis-and-forecasting-70d476bfe775”

Yadav, S. (2021). Designing Your Neural Networks. Towards Data Science. Retrieved from

“https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed”

Google LLC. (n.d.). S2 Geometry: S2Cell Hierarchy. Retrieved from

“https://s2geometry.io/devguide/s2cell_hierarchy.html”

Speedtest-CLI https://www.npmjs.com/package/speed-cloudflare-cli