Ole Kristian Westby, INF161

Bike traffic prediction project

Hello! In this PDF I will go over all my methodological choices, expectations and results. I will also explain my thinking, known issues/bugs, things I would change, and ideas.

This project is split into four parts:

- Data preparation and exploratory analysis
- Model selection, training
- Prediction
- Website

As you read through this report, I recommend having INF161project.py open on the side and watching as I talk through it, it will follow the same order that I do here.

Table of contents:

Data preparation

This project uses data from Statens Vegvesen and Geofysisk institutt. The goal is to create a model that can predict the volume of cyclists at a given time over Nygårdsbroen. For this part, it was all about taking all of that raw data (weather, traffic), and only focusing on what I subjectively decided would be important for the model.

I recognize that throughout the years there have been some times where people might have used the bikes more/less frequently because of certain factors.

- Covid-19 likely kept more people home, especially in peak times. Less people using bicycles to get to work as they had to work from home. Only interested in peak covid-19 times though.

- 2017 UCI Road World Championships. I've checked the routes and don't see that any bikes passed Nygårdsbroen.

We've been given weather data from 2010 all the way to 2023 as well as traffic data from 2015 to 2023. The libraries needed are numpy, pandas, os and matplotlib.pyplot.

I start by defining the directory for all our raw weather data, and I decide that instead of having to write up the names of every single .csv file (they're quite lengthy), I decide to use the power of **os** to create a list of every file in the directory that ends in .csv.

Next I define a list of interesting columns, these are interesting because I decided that I think only these will really have an effect on whether people use their bike that day.

Next, we'll append all those datasets in the list, with only the interesting columns into one big dataframe. This dataframe has 709220 rows of data.

	Dato	Tid	Globalstraling	Solskinstid	Lufttemperatur	Vindstyrke	Vindkast
709216	2023-06-30	23:10	0.7	0.0	13.7	2.3	3.6
709217	2023-06-30	23:20	0.7	0.0	13.6	1.9	3.3
709218	2023-06-30	23:30	0.8	0.0	13.6	1.7	3.0
709219	2023-06-30	23:40	0.8	0.0	13.6	1.9	3.3
709220	2023-06-30	23:50	0.8	0.0	13.5	1.9	3.0

After this, it is important to realize that because the traffic data only goes from 2015-2023 and the weather data is from 2010-2023, this effectively means we're missing a lot of traffic data for the earlier years. This could be solved in many ways, one way is to impute the data with a median, average, etc.. but I have decided that the best approach here is to simply filter the entire weather data for everything 2015 and later. This results in 446753 rows of data.

```
...              Dato    Tid  Globalstraling  Solskinstid  Lufttemperatur  \
    0      2015-01-01  00:00            18.1          0.0             6.6
    1      2015-01-01  00:10            19.5          0.0             6.6
    2      2015-01-01  00:20            20.2          0.0             6.6
    3      2015-01-01  00:30            20.2          0.0             6.6
    4      2015-01-01  00:40            20.2          0.0             6.7
    ...           ...    ...             ...          ...             ...
    446749 2023-06-30  23:10             0.7          0.0            13.7
    446750 2023-06-30  23:20             0.7          0.0            13.6
    446751 2023-06-30  23:30             0.8          0.0            13.6
    446752 2023-06-30  23:40             0.8          0.0            13.6
    446753 2023-06-30  23:50             0.8          0.0            13.5

            Vindstyrke  Vindkast
    0              4.2       NaN
    1              4.0       NaN
    2              3.1       NaN
    3              3.7       NaN
    4              2.9       NaN
    ...            ...       ...
    446749         2.3       3.6
    446750         1.9       3.3
    446751         1.7       3.0
    446752         1.9       3.3
    446753         1.9       3.0

    [446754 rows x 7 columns]
```

But, something important here is that we see on the column Vindkast, that there are rows with value NaN. This indicates missing data. There's also the value "9999.99" which indicates missing data. I want clean, full data, and who knows how many rows are missing data in one or more columns. We'll replace the 9999.99s with NaN and count this. Let's find out..

```
# Replace the missing data rows, "9999.99" indicates missing as well.
merged_weather_df.replace(9999.99, np.nan, inplace=True)

# Counting how many missing-data-rows
rows_missing_data = merged_weather_df[merged_weather_df.isna().any(axis=1)].shape[0]

print(rows_missing_data)
✓ 0.0s
11459
```

It appears there are 11459 rows with missing data. Again, we're left with some options on how to handle this, do we impute it or do we delete it? I went with deletion. The reason for this is 11459 only comes out to roughly ~2,6% of the merged dataframe. That is something we can afford to get rid of, and it wouldn't affect the models accuracy *as much*. So we drop them.

The next thing we want to do is convert the two columns, "Dato" and "Tid" to get a single datetime column. The reason for this is because the datetime column will be much easier to work with when we send it to the model, and we're able to pick out individual time features from that object, e.g hour, month, day etc..

Now, it's time to look at the traffic data, and this one is really messy! (a lot of unnecessary information). Reading the csv file for the traffic data and tailing it, gives us this:

	Trafikkregistreringspunkt	Navn	Vegreferanse	Fra	Til	Dato	Fra tidspunkt	Til tidspunkt	Felt	Trafikkmengde	Dekningsgrad (%)|A timer totalt|Antall ti inkludert|Antall ti ugyldig|Ikke gy lengde|Lengdekvalitets (%)|< 5,6m|>= 5,6m|5,(7,6m|7,6m - 12,5m|12,: 16,0m|>= 16,0m|16,(24,0m|>= 24
348635	17510B2483952	Gamle Nygårdsbru sykkel	KV256 S2D1 m75	2023-07-01T23:00+02:00	2023-07-02T00:00+02:00	2023-07-01	23:00	00:00	1	NaN	|||||
348636	17510B2483952	Gamle Nygårdsbru sykkel	KV256 S2D1 m75	2023-07-01T23:00+02:00	2023-07-02T00:00+02:00	2023-07-01	23:00	00:00	2	NaN	|||||
348637	17510B2483952	Gamle Nygårdsbru sykkel	KV256 S2D1 m75	2023-07-01T23:00+02:00	2023-07-02T00:00+02:00	2023-07-01	23:00	00:00	Totalt i retning Danmarksplass	NaN	|||||
348638	17510B2483952	Gamle Nygårdsbru sykkel	KV256 S2D1 m75	2023-07-01T23:00+02:00	2023-07-02T00:00+02:00	2023-07-01	23:00	00:00	Totalt i retning Florida	NaN	|||||
348639	17510B2483952	Gamle Nygårdsbru sykkel	KV256 S2D1 m75	2023-07-01T23:00+02:00	2023-07-02T00:00+02:00	2023-07-01	23:00	00:00	Totalt	NaN	|||||

.. a mess!

It is clear there is A LOT of information we don't need/want for the project. It is also worth noting that the amount of rows isn't that far away from our weather dataframe, so that is nice.

We start by filtering the dataframe where "Felt" == "Totalt". The reason for this is because we want to know the total number of cyclists in each direction. We're not trying to predict the amount of cyclists headed to Florida, we want to know how many pass the bridge. This will allow us to focus on that.

We'll also turn "Dato" and "Fra tidspunkt" into a datetime object. We'll drop these two columns afterwards, as well as drop all rows with missing data. This gives us a finished traffic dataframe that looks like this:

Datotid	Trafikkmengde
2015-07-16 15:00:00	-
2015-07-16 16:00:00	101
2015-07-16 17:00:00	79
2015-07-16 18:00:00	56
2015-07-16 19:00:00	45
...	...
2022-12-31 19:00:00	0.0
2022-12-31 20:00:00	0.0
2022-12-31 21:00:00	3.0
2022-12-31 22:00:00	5.0
2022-12-31 23:00:00	1.0

[65361 rows x 1 columns]

Let's go back to the weather dataframe. We notice here that the weather dataframe is displaying every 10 minutes, while the traffic dataframe is displaying every hour. What should we do here? I decide that the logical answer is to just resample the dataframe into hourly, with the mean. This won't mess up the data either, it'll stay consistent.

Datotid	Globalstraling	Solskinstid	Lufttemperatur	Vindstyrke	Vindkast
2023-06-30 14:00:00	532.633333	6.0	16.433333	4.316667	6.7
2023-06-30 15:00:00	369.983333	0.766667	16.3	4.25	7.3
2023-06-30 16:00:00	173.916667	0.166667	15.533333	4.2	6.55
2023-06-30 17:00:00	36.65	0.0	13.533333	3.65	6.35
2023-06-30 18:00:00	117.6	2.0	13.583333	2.433333	4.2
2023-06-30 19:00:00	28.88	0.0	13.68	2.86	4.38
2023-06-30 20:00:00	7.966667	0.0	13.283333	2.316667	3.9
2023-06-30 21:00:00	1.816667	0.0	13.466667	3.25	5.2
2023-06-30 22:00:00	0.416667	0.0	13.55	2.65	4.05
2023-06-30 23:00:00	0.75	0.0	13.616667	2.05	3.35

Now we're ready to merge them together. I notice Trafikkmengde is currently an object, so I decided to convert it to an integer.

The final dataframe now looks like this:

	Datotid	Trafikkmengde	Globalstraling	Solskinstid	Lufttemperatur	Vindstyrke	Vindkast
65351	2022-12-31 14:00:00	11	3.333333	0.0	3.566667	2.566667	4.55
65352	2022-12-31 15:00:00	13	-0.1	0.0	3.133333	1.533333	3.2
65353	2022-12-31 16:00:00	6	-0.85	0.0	2.533333	0.85	1.55
65354	2022-12-31 17:00:00	2	-0.2	0.0	1.883333	0.983333	1.75
65355	2022-12-31 18:00:00	8	0.133333	0.0	2.016667	1.4	2.75
65356	2022-12-31 19:00:00	0	-0.4	0.0	1.766667	1.283333	2.45
65357	2022-12-31 20:00:00	0	-0.15	0.0	0.666667	1.7	3.15
65358	2022-12-31 21:00:00	3	-1.75	0.0	0.483333	0.833333	1.6
65359	2022-12-31 22:00:00	5	-0.933333	0.0	0.516667	2.166667	4.2
65360	2022-12-31 23:00:00	1	-3.983333	0.0	0.316667	0.466667	1.1

Beautiful..

I was originally worried we wouldn't have enough data to train on, but it seems with 65360 rows, that is more than enough to get some good predictions!

Data exploration

With our dataframe now finally ready, I am curious about a couple things. For example, what's the most popular time of day, and day. What's the best temperature, and the best wind speed?

Let's find out!

We can find the best time by using idxmax() on a list of grouped "Datotid" rows with the mean traffic volumes. idxmax() will return the row label with the maximum value (meaning the most popular time). This is **16:00.** Which makes sense, I sort of expected this considering people are coming home from work..

Now let's find the average cyclists per day. I predict one of the weekend days (where people don't work). We'll find this by resampling the dataframe by days to find the total cyclists per day. Then we can get the average per day of the week using the mean and dayofweek. This is the result:

```
The average cyclists per day:
 Datotid
Monday       1540.933162
Tuesday      1589.442159
Wednesday    1557.789203
Thursday     1459.317949
Friday       1286.800000
Saturday      537.066667
Sunday        485.200514
Name: Trafikkmengde, dtype: float64
```

As predicted.. big surprise! (not)

It is clear that more people leave their bike at home (or at least don't travel over the Nygårdsbroen bridge on Sundays.

We can repeat this method (idxmax and mean) and find the best temperature is 26.7°C. Now this was a surprise, and I think it was because I didn't expect that to be the best temperature. I sort of think that it is a bit too hot to go cycling? Either I've made a mistake here, or there might be some outlier here that brought up this temperature a lot. I wasn't able to pinpoint it. (It wasn't SykkelVM either, that was in September 2017 and this would be too hot for that time of year).

Repeating the method with wind speed we find the best wind speed is 3,42 m/s. This sounds perfectly reasonable to me. Then again, this could simply be similar to the average wind speed and most Norwegians don't really care for the wind speed (unless it's really high). A quick Google search confirms that it is probably the average or something. I don't know how accurate this website is: https://weatherspark.com/y/52849/Average-Weather-in-Bergen-Norway-Year-Round (in mph, so I had to convert).

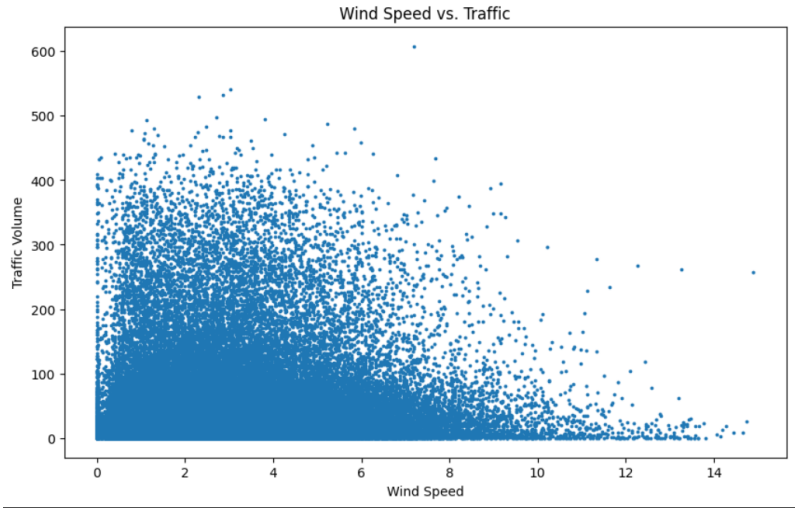
Additionally I also checked for Covid-19 but wasn't happy with my implementation, and I wasn't sure I had done it correctly. The result I found however was:

```
-----------------------------------------------------
Traffic for March-April each year to visualize Covid-19:
 Datotid
2016    238.075163
2017    158.312329
2018    164.756164
2019    185.210959
2020    143.049180
2021    178.216438
2022    617.741667
Name: Trafikkmengde, dtype: float64
```

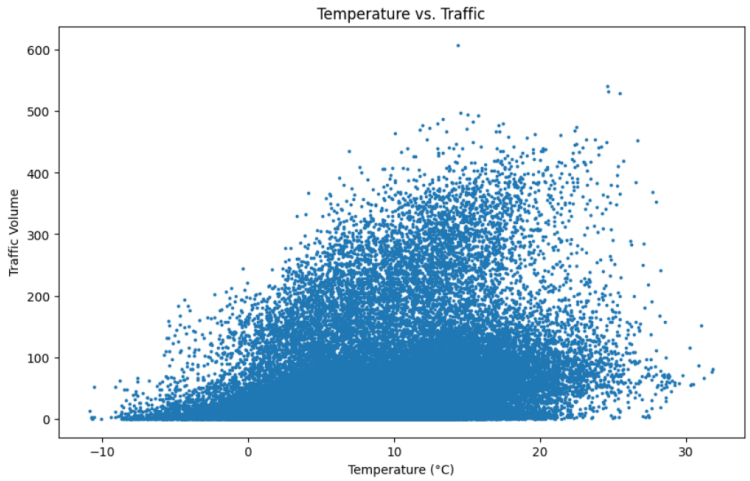
Here's the code as well (not included in my deliverables anymore..)

```
# Covid-19
march_april_data = ready_df[(ready_df['Datotid'].dt.month.isin([3, 4]))]
daily_bicyclists = march_april_data.resample('D', on='Datotid')['Trafikkmengde'].sum()
avg_bicyclists_per_year = daily_bicyclists.groupby(daily_bicyclists.index.year).mean()
print(avg_bicyclists_per_year)
print("-----------------------------------------------------")
```

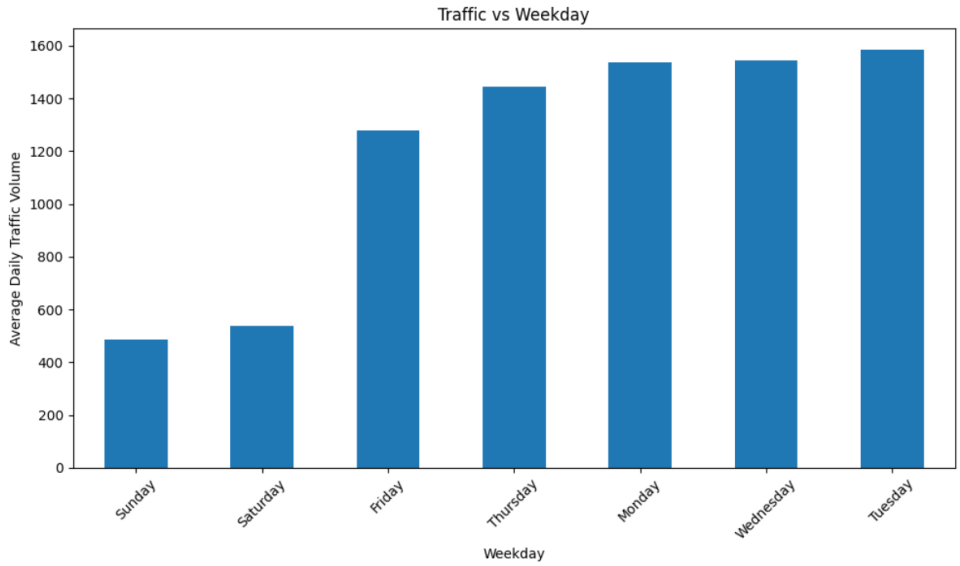
I then write the dataframe to a new file ready_data.csv. But before I end this part, I do some quick visualizations.



As we can see in the figure above, each point represents a day. It's important to think of the volume of that day as an indicator as to how many people went cycling, and not the amount of points anywhere on the figure. For example, over 14 m/s is a wind speed we rarely get, and hence so little data points. We can count 8 days with more than 14 m/s and 7/8 of those days kept people inside, except one wild day. Majority of people will leave their bike when we get over 8 m/s, which is considered significant.



Norway is a pretty cold country, so there's why we see so many data points under 20 celsius. When it's cold, and we're nearing 0 degrees and less, we see a lot less people leaving their bike at home. Especially when we're in -10.



Sunday being the least popular day wasn't a big surprise anywhere.

Model selection and training

This part is all about using the dataset we've created in the previous part and training a model on this data, finding the best possible model (both efficiency and root mean squared error), and finding the best hyperparameters.

We start by defining the location of our ready_data.csv file. We'll read the csv file into a dataframe using pandas (so we import pandas).

Before we continue, I decided to check for missing data one more time, and found some missing data again (I thought I got rid of it all).

```
# Missing data
missing_data = df.isnull().sum()
missing_data
```

```
Datotid             0
Trafikkmengde       0
Globalstraling    403
Solskinstid       403
Lufttemperatur    403
Vindstyrke        403
Vindkast          403
dtype: int64
```

Regarding these missing values, we're gonna fix this by imputing them with the median because the median is less sensitive to outliers. This will artificially impute these missing values with the median of the other values so it's sort of accurate. There are other strategies to do this, and they're not necessarily false either, but this is the way I'm doing it, plus it has worked earlier.

Next, we convert the "Datotid" column into a datetime format, and extract month, day and hour from the dataframe. This will help us later for predictions, and especially when the user is inputting datetime details on the website (soon to come).

	Datotid	Trafikkmengde	Globalstraling	Solskinstid	Lufttemperatur	Vindstyrke	Vindkast	Month	DayOfWeek	Hour
0	2015-07-16 15:00:00	50	504.400000	7.233333	13.900000	4.083333	6.70	7	3	15
1	2015-07-16 16:00:00	101	432.833333	8.116667	13.733333	4.333333	7.20	7	3	16
2	2015-07-16 17:00:00	79	378.400000	10.000000	13.866667	3.933333	6.55	7	3	17
3	2015-07-16 18:00:00	56	212.583333	10.000000	13.216667	4.233333	7.15	7	3	18
4	2015-07-16 19:00:00	45	79.750000	10.000000	12.683333	2.950000	5.45	7	3	19

Now that our data is clean, and *really ready*, we can start with data splitting. This essentially follows what I've been doing in the lab04 assignment. We'll need an additional library from this so we **import train_test_split** from **sklearn.model_selection**.

We're going to define two variables here, X and y. X is gonna be the entire dataframe, except "Datotid" (obsolete now), and "Trafikkmengde". This means that X now contains Month, DayOfWeek, Hour and all the weather information. Next, y is gonna be only "Trafikkmengde".

Once this is done, we're gonna define four variables here, X_train, X_val, y_train, y_val. These are split into two equal (0.5 test size) sets of data where one is training set and the other one validation. The training set contains 32,680 samples and the validation set contains 32,681 samples.

```
# How many samples?
X_train.shape, X_val.shape
```

```
((32680, 8), (32681, 8))
```

Next, we need to select a model, and we'll need to import numpy, as well as the first model and MSE. Let's try Linear Regression as our first model. ~0s runtime.

The project description asks for RMSE which is simply the square root of MSE. With LinearRegression we get a RMSE of ~61,7

```
# Calulate RMSE (root of mean squared error)
lr_rmse = np.sqrt(mean_squared_error(y_val, lr_predict))

lr_rmse
```

```
61.68697933176709
```

Before we continue checking different models, I wish to stop here and think a bit. Is ~61.7 a good RMSE here? I think in order to evaluate this, we need to check how many cycles on average each day. Let's check this.

```
count    65361.000000
mean        50.379905
std         69.782243
min          0.000000
25%          5.000000
50%         25.000000
75%         64.000000
max        608.000000
Name: Trafikkmengde, dtype: float64
```

Given this, it is clear that an RMSE of ~61,7 is pretty high, especially since the 75th percentile is 64 cycles. On average we're gonna be really off with this. This model won't do, we need to explore some more.. We can try the Lasso Regression model from the lab works. ~0.1s runtime

We'll repeat the exact same code, but with the Lasso model instead, and we get ~61,7 which is roughly the same as the Linear model.

```
# RMSE
lasso_rmse = np.sqrt(mean_squared_error(y_val, lasso_predict))
lasso_rmse
```

```
61.693154529589826
```

Next I try RandomForest ~4,6s runtime.

```
# RMSE
rf_rmse = np.sqrt(mean_squared_error(y_val, rf_predict))
rf_rmse
```

```
27.336702706600718
```

This is with n_estimators=10. I tested it with 100 estimators and got ~26,14. With 1000 estimators, it was ~25,89.

This is definitely a big improvement, and the current leader.

Will try another one, GradientBoosting, ~0.5s runtime

```
# RMSE
gb_rmse = np.sqrt(mean_squared_error(y_val, gb_predict))
gb_rmse
```

```
53.0983290926551
```

Hmm, not better. Okay, out of the three models we tested so far, RandomForest is winning. I'll try some more.. Next will try ElasticNet, ~0s runtime

```
# RMSE
en_rmse = np.sqrt(mean_squared_error(y_val, y_predict))
en_rmse
```

```
61.754404052183446
```

Let's try a different type of model, KNeighborsRegressor. ~1 second runtime

```
# RMSE
kn_rmse = np.sqrt(mean_squared_error(y_val, y_predict))
kn_rmse
```

```
49.66251270465994
```

We've now tried the following models: LinearRegression, Lasso, RandomForestRegressor, GradientBoostingRegressor, ElasticNet and KNeighborsRegressor. The ultimate winner is RandomForest.

Note: had it been allowed in the project description, I would probably try out some other libraries as well e.g XGBoost or some neural networks e.g Keras.

The next logical step is to find the best parameters to use with RandomForestRegressor. To achieve this, we can use GridSearchCV. This lets us specify a model, and a parameter grid, and it will run on every single combination of parameters until it finds the best estimators, and best params. This process took me between 29-35 minutes to run each time. This is why I have commented that entire code out in INF161project.py.

This was the result though:

```
grid_search = GridSearchCV(estimator=rf,
                           param_grid=param_grid,
                           cv=3,
                           n_jobs=-1,
                           verbose=2,
                           scoring='neg_mean_squared_error')

grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_

best_rf = grid_search.best_estimator_

rf_predict = best_rf.predict(X_val)

rf_rmse = np.sqrt(mean_squared_error(y_val, rf_predict))

rf_rmse
best_rf
```

✓ 34m 57.5s

```
Fitting 3 folds for each of 144 candidates, totalling 432 fits
```

```
RandomForestRegressor
RandomForestRegressor(min_samples_leaf=4, n_estimators=200, random_state=1)
```

Prediction

Now that we've successfully found our model and its parameters, it is time to do the next part of this project. That is, we give the model the weather data from 01.01.2023 to 01.07.2023 and have it predict traffic volume given the weather data.

To do this, we'll start by importing pandas, numpy and the model.

We'll have two variables for the paths to the input data, and output data. We will read the *Florida_2023-01-01_2023-07-01_1688719120.csv* file into a dataframe. Now, we'll have to repeat some steps from the preparation part, for example converting the two "Dato" and "Tid" columns into one "Datotid" column and also have it be a datetime object.

We'll then drop the obsolete columns "Dato" and "Tid".

Then we'll move the "Datotid" column to make it the leftmost column. Note that this isn't really necessary, but I still choose to do it for visualization where I might want to look at the dataframe. It is much cleaner and tidier when the first column you see is that. (I have to do this at the end anyways because the project description is asking for the columns in that order).

Now, this weather dataframe, just like the others, have missing data. So let's replace all "9999.99" values with NaN and fill them all with the median of that column (we can't drop them here, we need to predict on each timestamp).

Then, we remember the weather dataset had a bunch of columns that weren't necessary and the model won't look at them for predictions, in fact it will crash if we have it predicted with them included. So we will define a variable interesting_cols which contains all the columns that the model needs. We'll also remember that we need to extract the month, dayofweek and hour from "Datotid" and create new columns of those, then drop the "Datotid" column. (The model needs these three to predict).

Then.. we give the model the data and use predict(). We create a new column "Prediksjon" and set it to be the predicted data. Then because the assignment description only asks for the date, time and prediction columns, we have to retain only those. We'll also have to split the "Datotid" column back into "Dato" and "Tid" as per the description, then reorder them.

We save the dataframe to predictions.csv and voila! It is done. I have looked at the predictions, and they seem realistic as well. So I believe it has worked.

Website

The last part, which I found the most fun, is to create a website (hosted locally) that lets a user fill in a date and time, as well as optional weather data, then get a prediction on their screen of how many cyclists are predicted at that time.

All of the files for this can be found in the .zip file website.zip. You'll see that it contains in the main directory: app.py, model.py and three directories: static, templates and ready_data.

We'll start with the simplest one, model.py.

Model.py is like the name suggests, literally just a copy of the model we've found and trained earlier. It is done this way because it was asked for the website files to be zipped, and all required files needed to run the server needed to be included. I'm not going to explain it any further since I have already done so earlier. The TLDR: app.py calls for the model when receiving requests.

App.py is our flask server. You'll immediately notice an interesting function get_or_default(). This function takes three parameters: field, form, default_value. It basically does this: whenever the user sends in a request for prediction, it will check every single field. If the user has given a value, we will **get** that value *value = form.get(field)* and return the float version of the value if it exists, if not it will return the default_value, in this case: median. Let's say the user is sending in a request but leaves Globalstraling empty. The function will be called and since the field is empty, *else statement*, it will default to the median version instead. If however, the user does decide to input something, it will simply stay as the float version of that. The rest of this file is simple, we save all the input data as input_data, feed it to our model, and round it (the user is probably not interested in 500 decimal points), then return it to the user. I have also made some changes with app.run() to force it to run on localhost:8080, although it should run on all local addresses anyways. Make sure the 8080 port is open on your network!

static/styles.css contains styling. I made this file with an online editor to save some time, figured it was fine because it was only the styling and it is my work regardless.

templates/index.html contains the website. It is pretty self explanatory, but basically we've created a bunch of input-groups. We've also made sure to make the date and time a required form, but other fields are optional.

ready_data/ready_data.csv is just the dataset that the model is trained on, and is necessary for the model to work.

Lastly, you will see a picture of the webpage on the next page.
PS: This was probably my favorite project since I started my degree in 2020, kudos.

Bike Traffic Prediction

Date & Time:

05 / 17 / 2040 , 01 : 32 PM

Globalstraling:

5

Solskinstid:

8

Lufttemperatur:

17

Vindstyrke:

7

Vindkast:

10

Predict

The model predicts: 77.84 bicyles over Nygårdsbroen with given parameters.

Made by

Ole Kristian Westby

INF161 H23 | owe009@uib.no