Scalable analytics - Bike count forecasting in Brussels

Quivron Loic, Tribel Pascal,Mehdi MOUTON,Damiano BERTOLDO

May 15, 2023

1 Introduction

Brussels Mobility Bike Count API is an API that provides the number of bikes passing in front of multiple sensors placed in Brussels. It gives multiple information, including the number of bikes and their speed, every 15 minutes.

This data can be used in order to make prediction on the number of bikes that will pass in front of each sensor, using embedding models that predict a quantity at time $t+1$ using the historical data with

$$\hat{y}(t+1) = f(y(t), ..., y(t-n)) \tag{1}$$

In the real world situation, this data keeps getting updated every 15 minutes. New data can help models to improve their predictive performances, therefore, *streaming* fashion data collecting is suitable to propose almost real-time improving models. In this paper, we propose different models, presented in section 3, computed on a data flow generated using *Spark framework*, presented in section 2. Then, we compare the performance of those models in section 4.

2 Data streaming

Data streaming is implemented using *Spark*. It flows data from a *Producer* to a *Consumer*. The data used in this paper starts on the 15^{th} of April 2022 and stops on the 31^{st} of March 2023. This makes a total of 320 days, or $320 \times 24 \times 4 = 30720$ rows for each sensor. The data coming from the API is pre-processed in the first part of this project, and the missing rows are completed with 0 for number of bikes and -1 for the speed. The completed data is stored in a file called *data.csv*.

2.1 Producer

The producer (in the *TimeSerie_Producer* notebook) collects the data and computes a few columns depending on the date, that will be useful for the models of section 3, including:

- Day: the number of the day in the year
- Day of week: the number of the day in the week (1 for Monday, 2 for Tuesday, ...)
- Week of year: the number of the day in the week
- Year

The following data are also included in the *dataframe* that the producer builds:

- Date
- Timegap: a number between 1 and 96 representing the quarter of an hour of a given row
- Sensor: the identifier of the sensor having produced a given row
- Count and Average speed

Finally, the Producer builds a column called *Year_week* which allows to batch the data by week. The data is sent every two seconds to the consumer, by batches built on a week basis. This choice is arbitrary.

2.2 Consumer

Once the data is sent by the Producer, the Consumers (see *TimeSerie_Consumer* and *RLS_Ultimate_Consumer* notebooks) receive the data and compute the different models (se section 3) on each received batch.

2.3 Distributed system

Our architecture processes the data by serializing it on a sensor basis. This allows to expand the size of the architecture when the number of sensors increases, but not when the frequency of data collection increases.

In order to compute the different models (see section 3), the states are composed of multiple values. For the persistence and average models, the state is made of:

- The key of the state, which is the sensor name
- The prediction function to use (persistence or average)[1]
- The n last observations
- The Sum of Squared Error (SSE)
- The number of observations (to compute the MSE from the SSE)

For the RLS model, we add some other parameters to the state:

- β, the coefficients of the linear model
- V, the covariance matrix
- err, the last prediction error
- $\hat{y}$, the last prediction

3 Models

We propose three families of models, presented by increasing complexity:

- A persistence model
- An average model
- Five Least-Square models

3.1 Persistence model

A *persistence model* always output the last value received as prediction for the next step, like

$$\hat{y}(t) = y(t-1) \tag{2}$$

There is no parameter to update other than the value to output.

3.2 Average Model

An *average model* outputs the average of the n last received values as prediction for the next steps, with

$$\hat{y} = \frac{\sum_{i=1}^{n} y(t-i)}{n} \tag{3}$$

In section 4, we compare the impact of n on the quality of the prevision. Persistence model is a particular case of Average Model, where $n = 1$.

[1] We have two functions for those computations, even if the first one is a particular case of the second one. We made a separate function for the sake of efficiency.

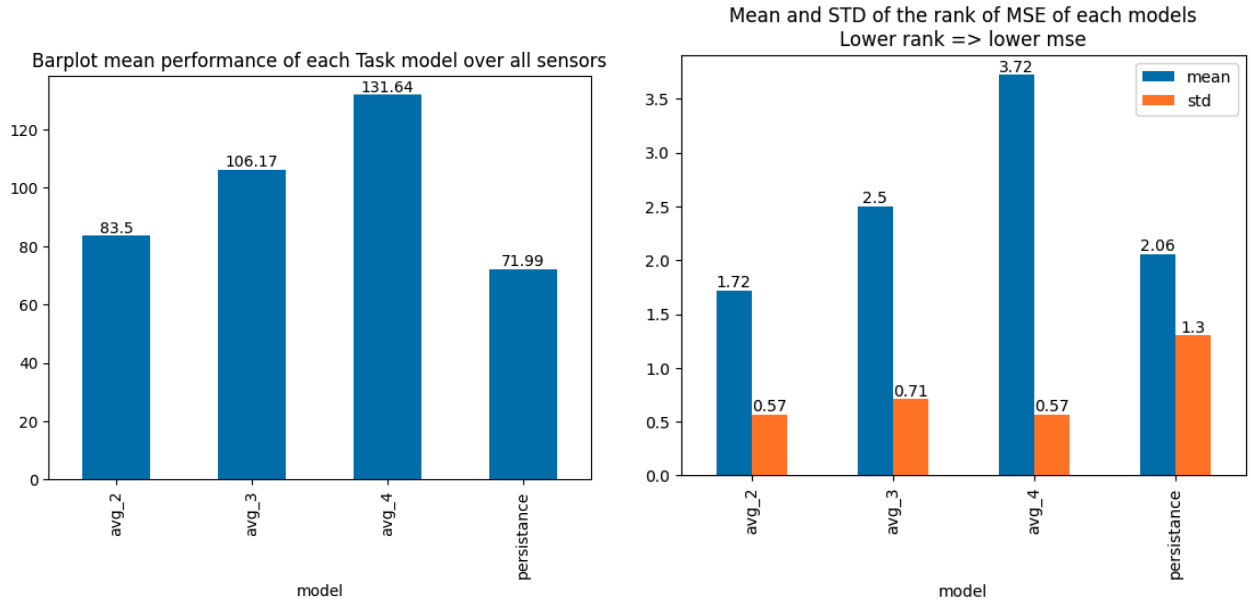


Figure 1: Comparison of the MSE, and the mean and standard deviation of the rank of the MSE for three average models ($n = \{2, 3, 4\}$) and the persistence model

3.3 Least-Square Models

Least-Square is a method to build *linear models*. Linear models on inputs of size n are functions of the form

$$\hat{y} = \beta_0 + \sum_{i=1}^{n} \beta_i y(t-i) \tag{4}$$

and *Least-Square* aims at finding the best values for β to minimize the *Mean Squared Error* between the prediction and the expected output.

3.3.1 With time features

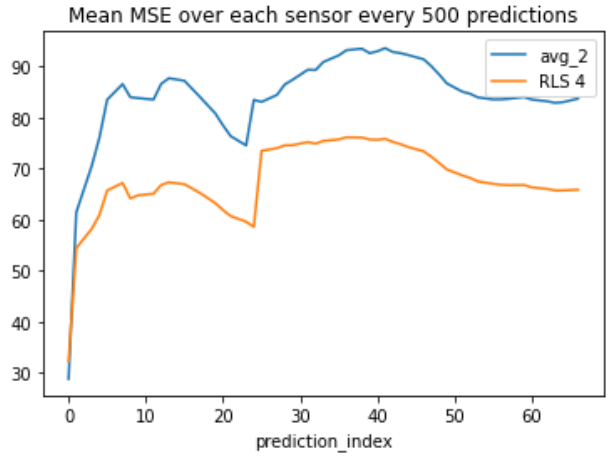
We have tried to improve the function above by adding variables for the time stamp ts (i.e. the quarter of an hour where the measure has been taken), the day in the week dw and the week in the year y, which leads to the new function

$$\hat{y} = \beta_0 + \sum_{i=1}^{n} \beta_i y(t-i) + \beta_{n+1} ts + \beta_{n+2} dw + \beta_{n+3} y \tag{5}$$

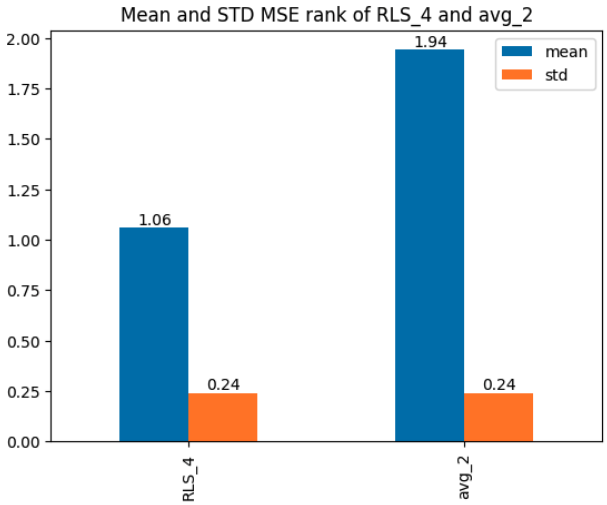
4 Models comparison

Figures 1 shows on the left side the mean MSE value of the *persistence* model and the *average* model with $n = \{2, 3, 4\}$. On the right side is shown the mean and the standard deviation of the rank of each of those models over all sensors (a lower rank indicating lower MSE and thus a better model for the same sensor). While on the left plot, the *persistence* exhibits a lower mean MSE than *avg_2*, *avg_2* has a lower mean rank in the right plot as well as a much lower standard deviation (shown in the orange bars). *avg_2* therefore seems to be the overall model for those reasons.

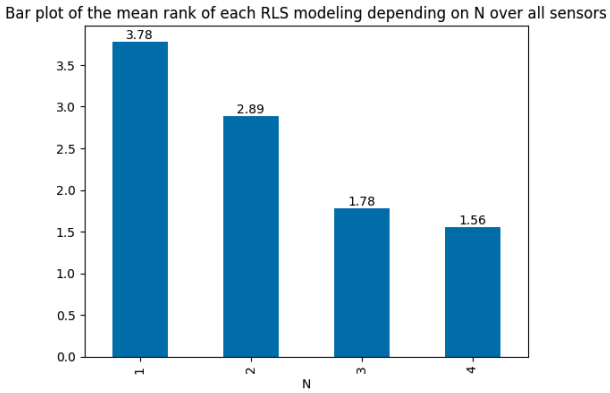
Figure 3a shows the the mean rank MSE of each RLS models depending on different N. $N = 4$ is the one with lowest rank. For this reason a comparison between the best RLS model and the best *Average* model was made and plotted in Figure 2b. $RLS4$ clearly outperforms *Average* model with $n = 2$.



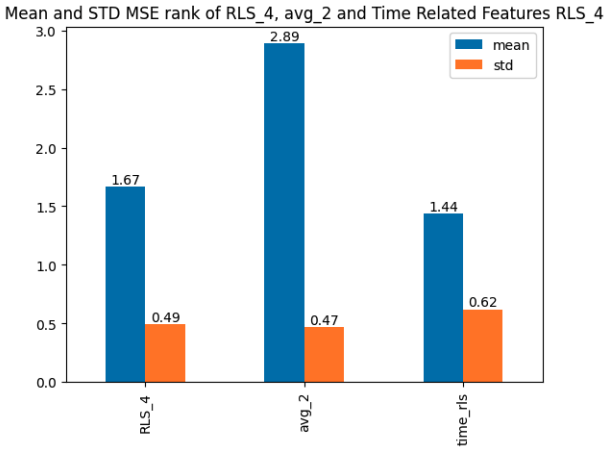
(a) Mean of the MSE the RLS models for $n = 4$, and the average model with $n = 2$



(b) ean rank of the RLS models for $n = 4$, and the average model with $n = 2$



(a) Mean rank of the RLS models for $n = \{1, 2, 3, 4\}$



(b) Mean rank of the RLS models for $n = \{1, 2, 3, 4\}$, and the model with $n = 4$ with added time features

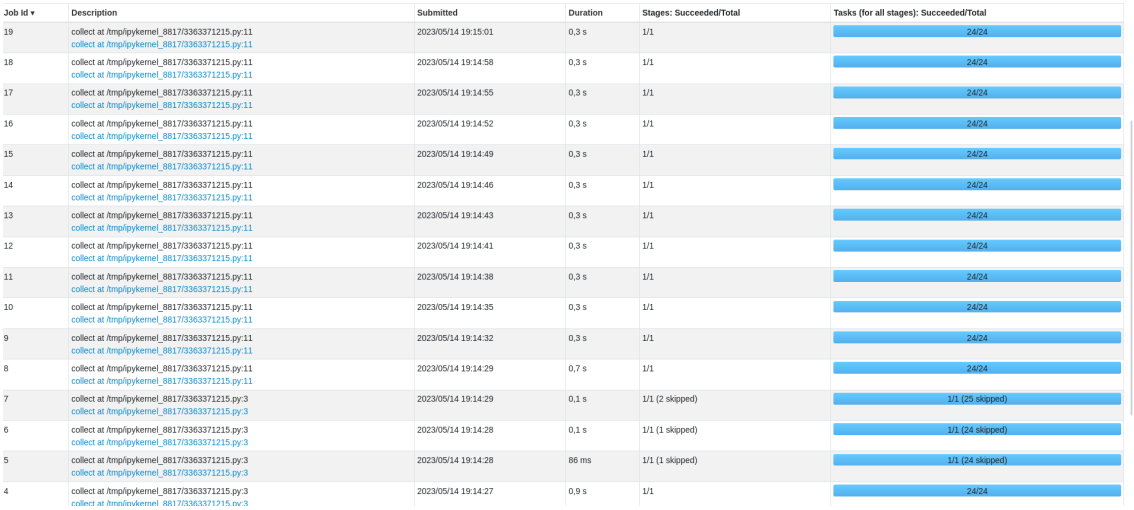
We show in figure 2a that the variation of MSE over time is comparable between RLS with $N = 4$ and *Average* with $n = 2$, but RLS always outperforms *Average*. The curve of RLS is also smoother, which means that the MSE for this model is less sensitive to the variation of data through time.

Finally, If we add time features as proposed in section 3, as shown in figure 3b, the performances seem not to improve: the mean is slightly lower but the standard deviation is higher. Therefore, we suppose that those information have either no linear impact on the number of bikes that pass, or that they have no impact at all, or that the information is redundant with the time series used for the prediction.

5 Scalability

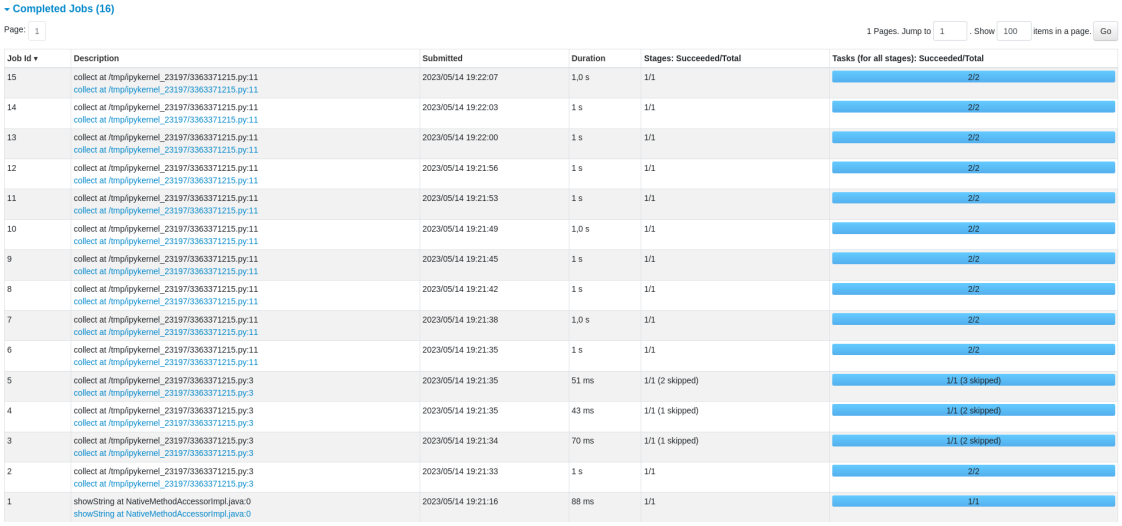
To test the scalability of our implementation we conducted an experiment by launching the persistent model concurrently over all sensors two times. The first one by allowing Spark to use as many cores as possible (24) and one where we limit Spark to use 2 cores.

Figure 4 shows a screenshot of the dashboard using 24 cores, while Figure 5 shows the same screenshot but using 2 cores for the same task (the persistence model). Clearly we can observe that our implementations scales with the number of cores as the Job duration goes from 1 second with 2 cores to 0.3 seconds with 24



Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
19	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:15:01	0,3 s	1/1	24/24
18	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:58	0,3 s	1/1	24/24
17	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:55	0,3 s	1/1	24/24
16	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:52	0,3 s	1/1	24/24
15	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:49	0,3 s	1/1	24/24
14	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:46	0,3 s	1/1	24/24
13	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:43	0,3 s	1/1	24/24
12	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:41	0,3 s	1/1	24/24
11	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:38	0,3 s	1/1	24/24
10	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:35	0,3 s	1/1	24/24
9	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:32	0,3 s	1/1	24/24
8	collect at /tmp/ipykernel_8817/3363371215.py:11 collect at /tmp/ipykernel_8817/3363371215.py:11	2023/05/14 19:14:29	0,7 s	1/1	24/24
7	collect at /tmp/ipykernel_8817/3363371215.py:3 collect at /tmp/ipykernel_8817/3363371215.py:3	2023/05/14 19:14:29	0,1 s	1/1 (2 skipped)	1/1 (25 skipped)
6	collect at /tmp/ipykernel_8817/3363371215.py:3 collect at /tmp/ipykernel_8817/3363371215.py:3	2023/05/14 19:14:28	0,1 s	1/1 (1 skipped)	1/1 (24 skipped)
5	collect at /tmp/ipykernel_8817/3363371215.py:3 collect at /tmp/ipykernel_8817/3363371215.py:3	2023/05/14 19:14:28	86 ms	1/1 (1 skipped)	1/1 (24 skipped)
4	collect at /tmp/ipykernel_8817/3363371215.py:3 collect at /tmp/ipykernel_8817/3363371215.py:3	2023/05/14 19:14:27	0,9 s	1/1	24/24

Figure 4: Spark Task Dashboard with 24 cores



▾ Completed Jobs (16)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
15	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:22:07	1,0 s	1/1	2/2
14	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:22:03	1 s	1/1	2/2
13	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:22:00	1 s	1/1	2/2
12	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:56	1 s	1/1	2/2
11	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:53	1 s	1/1	2/2
10	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:49	1,0 s	1/1	2/2
9	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:45	1 s	1/1	2/2
8	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:42	1 s	1/1	2/2
7	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:38	1,0 s	1/1	2/2
6	collect at /tmp/ipykernel_23197/3363371215.py:11 collect at /tmp/ipykernel_23197/3363371215.py:11	2023/05/14 19:21:35	1 s	1/1	2/2
5	collect at /tmp/ipykernel_23197/3363371215.py:3 collect at /tmp/ipykernel_23197/3363371215.py:3	2023/05/14 19:21:35	51 ms	1/1 (2 skipped)	1/1 (3 skipped)
4	collect at /tmp/ipykernel_23197/3363371215.py:3 collect at /tmp/ipykernel_23197/3363371215.py:3	2023/05/14 19:21:35	43 ms	1/1 (1 skipped)	1/1 (2 skipped)
3	collect at /tmp/ipykernel_23197/3363371215.py:3 collect at /tmp/ipykernel_23197/3363371215.py:3	2023/05/14 19:21:34	70 ms	1/1 (1 skipped)	1/1 (2 skipped)
2	collect at /tmp/ipykernel_23197/3363371215.py:3 collect at /tmp/ipykernel_23197/3363371215.py:3	2023/05/14 19:21:33	1 s	1/1	2/2
1	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/05/14 19:21:16	88 ms	1/1	1/1

Figure 5: Spark Task Dashboard with 2 cores

cores.

6 Conclusion

In this work, we have presented 9 models, to forecast the number of bikes passing in front of sensors in Brussels, in a streaming data way. Four of them were averaging historical data to predict the next one, with a particular case called persistence. Five of them used *RLS* algorithm, with various history sizes, and with use of time-related features.

We have shown that *RLS* with $n = 4$ without use of time-related features was the most performative. In the future, more complex models, like non-linear ones, or models using more information, like the speed of the bikes, the weather, *etc.* could improve the prediction performances.