INFOH423 – Project

Hack my Ride

Data Mining Project 2021/22

Mahmoud SAKR and Jean-philippe HUBINONT

Group members :

- Thibault LANTHIEZ - 000541104
- Ishaan BINDAL - 000537572
- Rezeda SULTYEVA - 000537752
- Youssef BENAHMID - 000484014
- Abdelghani El HELLAJ - 000550457

Table of contents

1. Introduction

Société des Transports Intercommunaux de Bruxelles - Maatschappij voor het Intercommunaal Vervoer (STIB-MIVB) is the local public transport operator in Brussels and is accountable for the organisation of the Brussels' metro, trams, and buses. It is this dynamic and complex transport system which is the focus of this project. The project utilizes geographic transport modelling and data mining techniques to simplifying this complexity in a way that captures the essence of the following transport problems:

1. Vehicle Speed - Effective computation of vehicle speed is central to providing advanced traveller information and managing transportation systems. Vehicle speed is also a critical factor in reducing private car use and prioritising public transport as a means to bring down congestion, air pollution, and greenhouse gas emissions.

2. Vehicle Delays – Delays in public transport are key to assessing the vulnerability of a transport network. It can increase passengers' travel time as well as increase costs for the service provider (due to overtime payments to employees, decreased ridership levels and fines resulting from contractual disagreements between service provider and authority).

3. Arrival Time – Having information about the arrival time is fundamental for efficient public transport operation and dispatching decisions. It also has many positive effects on the users of public transport services such as lower uncertainty, increased ease of use and sense of security, increased willingness to pay, improved overall image of service provider and consequently may lead to an increased ridership in public transport.

4. GPS Tracking – Vehicle tracking service is the basis for implementing mobility solutions for problems such as traffic congestion, and pollution generated by transit vehicles, which increases in high congestion situations.

The organization of the rest of the report will start with a description of dataset loading and preprocessing in Section 2, followed by description of the data exploration activity in Section 3. Next, in Section 4, we discuss the results of the 5 questions of this project. We will also present our web application which can be used to interact with the results. Lastly, the report will conclude in the last section.

2. Description of dataset loading and preprocessing

For the entirety of this project we have used Python. It has the advantage of being easy to program and has a large number of libraries which can be used for data treatments.
Firstly, we used the popular Python library *pandas* to put our data files in a dataframe format. It behaves like SQL queries which are easy to understand and are complete. Pandas also has the capability to deal with millions of rows of data and allows complex data procedures to be performed such as *map* and *apply* to process each row of a dataframe. For all the questions of this project, we used the function *read_csv* in order to read data in a tabular format.

Further, in order to process shapefiles, we used *geopandas*. It is another python library which can deal with maps and data positions. We can choose the coordinate systems like latitude/longitude or a cartesian plane of Brussel in it. For position type, we compute distances using the *shapely* framework. It is composed of multiple methods such as the euclidean distance or finding the nearest point between a linestring (which includes various data points) and a point.

Our IDE (Integrated development environment) for this project was *Jupyter*. With this software, we were able to easily organize our codes in a cell format. The different outputs in the notebook were highly interpretable and efficient to manage.

Finally, on another IDE (like Atom/Visual Studio Code), we developed a Web Application with the framework *Streamlit*. It is a Python framework which allows us to create simple applications with widgets in order to interact with our results. We will discuss this part in detail during the presentation of our results.

In the next section, we will present the different data files that we have used during this project. We will also explore these files by showing you some visualizations.

3. Data exploration activity

3.1 GTFS

The General Transit Feed Specification (GTFS) defines a common format for public transportation schedules and associated geographic information. It allows public transit agencies to publish their static transit data in an interoperable way.

We have 9 GTFS files. Six of the files are mandatory for GTFS structure:

- agency.txt
- calendar.txt

- routes.txt
- stops.txt
- stop_times.txt
- trips.txt

While the remaining three are optional and help with scheduling exceptions (calendar_dates.txt), mapping routes geographically (shapes.txt), and translating names (translations.txt).

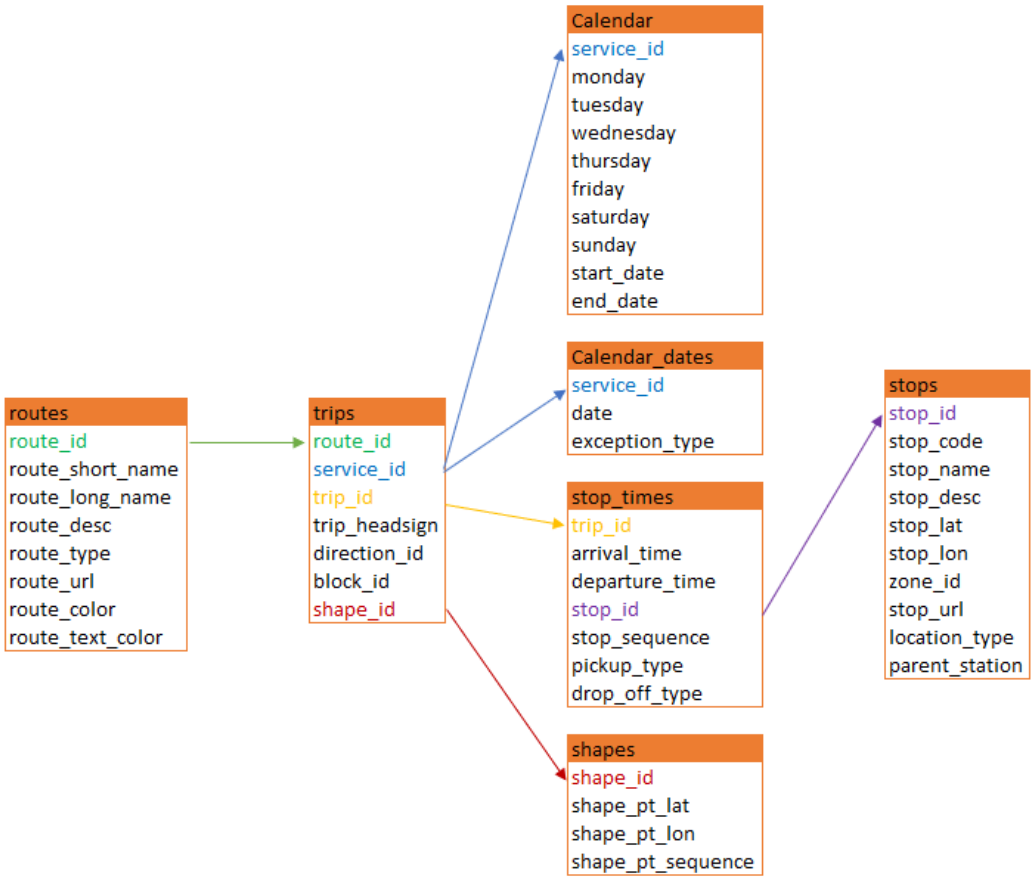
GTFS files are in a series of comma-delimited text files and represent a fixed-route schedule. They have become the most popular world-wide data format to describe fixed-route transit services.

The contents of both calendar.txt, and calendar_dates.txt files contain schedule information. The stops.txt contains information about the name, ID, and location of every stop. The routes.txt file contains information about the transit agency's routes. Information about the order of the stops for a particular route according to a particular schedule is provided in the trips.txt and stop_times.txt files. Spatial representation of a route alignment to be accurately drawn on a map is included in the shapes.txt file.

GTFS Files	N° variables	Contents and description	Required/optional
agency.txt	6	Information about the transit agency.	Required
calendar.txt	10	Defines service categories. Each category indicates the days that service starts and ends as well as the days that service is available.	Required
calendar_dates.txt	3	Lists exceptions for the service categories.	Optional
routes.txt	8	Information about a transit organization's routes. A route is a group of trips that are displayed to riders as a single service.	Required
shapes.txt	4	Provides rules for drawing lines on a map to represent a transit organization's routes.	Optional
stop_times.txt	7	Times that a vehicle arrives at and departs from individual stops for each trip.	Required
stops.txt	10	Information about individual locations where vehicles pick up or drop off riders.	Required
translations.txt	3	Defines fare information for a translation.	Optional
trips.txt	7	Information about scheduled service along a particular route. A trip is a sequence of two or more stops that are made at regularly scheduled intervals.	Required

General Transit Feed Specification (GTFS) table

The following figure demonstrates the General Transit Feed Specification (GTFS) tables relations:



General Transit Feed Specification (GTFS) table relations

3.2 Shapefiles

The Shapefiles available contain the information about the spatial structure of the STIB/MIVB network: route of the lines and position of the stops of the network, that is the basic itineraries "to" and "from". These different lines will help give good variance of delays for all the lines.

We used two shapefiles : "ACTU_LINES" and "ACTU_STOPS".

Let's see ACTU_LINES:

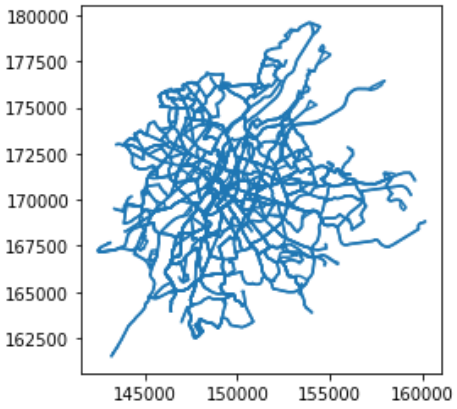
	LIGNE	VARIANTE	COLOR_HEX	Date_debut	Date_fin	geometry
0	001m	1	#C4008F	01/09/2021	06/03/2022	LINESTRING Z (146633.5 170956.3999999985 0, 14...
1	001m	2	#C4008F	01/09/2021	06/03/2022	LINESTRING Z (156746.700000003 170167 0, 15668...
2	002m	1	#F57000	01/09/2021	06/03/2022	LINESTRING Z (147305.5 172526.8999999985 0, 14...
3	002m	2	#F57000	01/09/2021	06/03/2022	LINESTRING Z (147370.5 172498.5 0, 147415.7999...
4	003t	1	#B5BA05	01/09/2021	06/03/2022	LINESTRING Z (148550 176641.3000000007 0, 1485...
...	...	...	...	...	...	...
169	213b	2	#991F36	01/09/2021	06/03/2022	LINESTRING Z (144054.200000003 169925.5 0, 144...
170	216b	1	#80C29C	01/09/2021	06/03/2022	LINESTRING Z (149229 170664.5 0, 149214.5 1706...
171	216b	2	#80C29C	01/09/2021	06/03/2022	LINESTRING Z (144443.3999999985 173553.1999999...
172	218b	1	#17A345	01/09/2021	06/03/2022	LINESTRING Z (149173.5 170683.3999999985 0, 14...
173	218b	2	#17A345	01/09/2021	06/03/2022	LINESTRING Z (147730 176277.1000000015 0, 1477...

Pandas dataframe for the file ACTU_LINE.shp

We have 6 columns. The following information is available for each variable:

- LIGNE: commercial line number (tram, bus or metro)
- VARIANTE: variant number of the line (1 (to) or 2 (from))
- COLOR_HEX: official colour code associated to the commercial line
- Date_debut: start date of the represented network
- Date_fin: end date of the represented network
- geometry: geometry field

With *Geopandas*, it's easy to represent all the lines in a map. Let's see what the different routes of STIB look like:



Line routes of the file ACTU_LINES.shp

It's difficult to analyse in depth this map but it gives a global view of the reartition of the lines (bus, tram, metro) in Brussels.

Now, let's see what's inside ACTU_STOPS:

	Code_Ligne	Variante	succession	stop_id	descr_fr	descr_nl	alpha_fr	alpha_nl	coord_x	coord_y	mode	numero_lig	terminus	geometry
0	012b	1	1	9600B	BRUSSELS AIRPORT	BRUSSELS AIRPORT	Brussels Airport	Brussels Airport	157950.0	176429.0	B	12	BRUSSELS CITY	POINT (157950 176429)
1	012b	1	2	3017	BOURGET	BOURGET	Bourget	Bourget	154334.0	174200.0	B	12	BRUSSELS CITY	POINT (154334 174200)
2	012b	1	3	5048	DA VINCI	DA VINCI	Da Vinci	Da Vinci	152934.0	173976.0	B	12	BRUSSELS CITY	POINT (152934 173976)
3	012b	1	4	2695	GENEVE	GENEVE	Genève	Genève	152428.0	172606.0	B	12	BRUSSELS CITY	POINT (152428 172606)
4	012b	1	5	2250	MEISER	MEISER	Meiser	Meiser	152045.0	171508.0	B	12	BRUSSELS CITY	POINT (152045 171508)
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4171	097t	2	25	6427F	JANSON	JANSON	Janson	Janson	148985.0	168336.0	T	97	LOUISE	POINT (148985 168336)
4172	097t	2	26	6430F	JANSON	JANSON	Janson	Janson	149019.0	168413.0	T	97	LOUISE	POINT (149019 168413)
4173	097t	2	27	5066F	FAIDER	FAIDER	Faider	Faider	149107.0	168847.0	T	97	LOUISE	POINT (149107 168847)
4174	097t	2	28	5068F	STEPHANIE	STEFANIA	Stéphanie	Stefania	149184.6	169237.5	T	97	LOUISE	POINT (149184.6 169237.5)
4175	097t	2	29	6361	LOUISE	LOUIZA	Louise	Louiza	149034.0	169497.0	T	97	LOUISE	POINT (149034 169497)

Pandas dataframe for the file ACTU_STOPS.shp

The shapefile of the stops consists of all the stops for each line of the STIB/MIVB network covering both directions. Each stop is multiplied according to the number of lines serving this stop. The stop is graphically represented by a local point in terms of stop line position of the vehicle.

The following information is available for each object:

- code_ligne: commercial line number (bus, tram or metro)
- VARIANTE: variant number of the line (1 (to) or 2 (from))
- succession: represents the stop position in the succession order of the stops along the itinerary
- stop_id: stop number (internal identifier, 5 characters)
- descr_fr: contains the functional name of the stop in capitals and in French
- descr_nl: contains the functional name of the stop in capitals and in Dutch
- alpha_fr: contains the official name of the stop in small letters and in French
- alpha_nl: contains the official name of the stop in small letters and in Dutch
- coord_x: coordinate X
- coord_y: coordinate Y

- mode: represents the mode (B, T or M)
- geometry: geometry field
- numero_lig: line number
- terminus: destination stop

3.3 Vehicle position

One of the formats used for data representation in this project was JSON format (JavaScript Object Notation) which is a popular data format used for representing structured data. Generally, it is common to transmit and receive data between a server and web application in JSON format.

In our case, the Vehicle Position Real-Time API of STIB-MIVB was invoked in order to collect the data regarding vehicle locations. A JSON file contains raw data not suitable for data analysis without transferring it to a readable format. That is why the first step was to explore JSON files in order to get an idea of the structure of data. For this purpose we used the json module of Python which was built for decoding and encoding JSON data.

This data file gives us access to information concerning the positions of vehicles in real time, on which stop a vehicle has passed and how many vehicles from a transit line are currently driving on the network. We will use this data to track the vehicle position and when they are passing stops.

Let's see the structure of JSON data:

{'data':
[{'time': '1631270420474',
'Responses': [{'lines': [{'lineId': '1',
'vehiclePositions': [{'directionId': '8161',
'distanceFromPoint': 0,
'pointId': '8092'},
{'directionId': '8731', 'distanceFromPoint': 0, 'pointId': '8141'},
{'directionId': '8161', 'distanceFromPoint': 1, 'pointId': '8032'},
{'directionId': '8161', 'distanceFromPoint': 1, 'pointId': '8132'},
{'directionId': '8731', 'distanceFromPoint': 0, 'pointId': '8091'},
{'directionId': '8161', 'distanceFromPoint': 1, 'pointId': '8292'},
{'directionId': '8731', 'distanceFromPoint': 1, 'pointId': '8041'},
{'directionId': '8731', 'distanceFromPoint': 0, 'pointId': '8281'},
{'directionId': '8161', 'distanceFromPoint': 0, 'pointId': '8733'}]},
{'lineId': ...,
'vehiclePositions': [{'directionId': ...,
'distanceFromPoint': ...,
'pointId': ...}...]}

This file describe the following variables:

- '*data*' is a JSON object.
- '*time*' is the time in milliseconds (unix epoch) at which the API was invoked.
- '*Responses*' is the array containing the result of the 9 API calls, each of these calls returns for the given line IDs all their vehicle positions.
- '*vehiclePositions*' is information about vehicle positions containing the following information:
- '*directionId*' is the identifier of the terminal stop.
- '*distanceFromPoint*' is the distance in meters between the vehicle and the last traversed stop.
- '*pointId*' is the identifier of the last stop traversed by a vehicle.

Since the represented JSON data is hierarchical and nested, the second step was to normalize the data and to import it into the data frame. Normalizing was done by the above-mentioned json module. The most convenient way of creating a data frame was using pandas module which is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Besides, we needed to check the quality of the data before applying data mining algorithms. That is the reason why as a part of data preprocessing we detected the errors in JSON data. As a result of this step we discovered that our data contains incorrect and incomplete data. For example, some lists were empty or some values were 'None'. That led us to a data cleaning process where we removed erroneous data.

After that we created a data frame consisting of cleaned data. For example, for one of the files the data frame looks like the one at the figure below, consisting of the columns ”directionId”, “distanceFromPoint”, “pointId”, “time”, “lineId”:

	directionId	distanceFromPoint	pointId	time	lineId
0	8731	0	8051	1631264056110	1
1	8161	0	8731	1631264056110	1
2	8731	0	8111	1631264056110	1
3	8731	0	8021	1631264056110	1
4	8161	0	8733	1631264056110	1
...	...	...	...	...	...
672795	2382	60	5953	1631292728222	98
672796	1951	0	2384	1631292728222	98
672797	2382	0	1951	1631292728222	98
672798	2382	0	2382	1631292728222	98
672799	2382	709	2610	1631292728222	98

672800 rows × 5 columns

Pandas dataframe for the file ‘VehiclePosition05.json’

As soon as we disposed of the cleaned data frame we could easily transform it into a csv file for further data analysis (question n°2).

4 Results

4.1 Analyze the vehicle speed over the different network segments, how it varies across segments and over time. Present this in a suitable visual way.

4.1.1 Methods and computations

For this first question, we were asked to compute the speed over different segments of the transport company of Brussels (STIB). For this, we computed the speed between each stop for each vehicle id. Further, we computed the average speed for a line. You will see our results at the end of this part.

But now, let us focus on the process and methods in order to compute the speed.

First of all, for the speed, we need two components: the travelling time and the distance.

The travelling time could be computed quickly since we had in the file named *stop_times.txt* the stop time at each stop for each vehicle id. However, these times were not the exact time, but the predicted times for example, the times which were written in the calendar.

It is important to know that for this file the departure time and the arrival time are the same. Hence, we did not know precisely how long the vehicle had stopped and how long it was moving. This means that our computation of the vehicle speed included the stop time. Therefore, it is underestimated.

<u>What we did to get the time:</u>

First of all, we had this file :

```
stop = pd.read_csv('stop_times.txt')
stop
```

	trip_id	arrival_time	departure_time	stop_id	stop_sequence	pickup_type	drop_off_type
0	112387248235954071	21:07:00	21:07:00	4014	1	0	0
1	112387248235954071	21:09:00	21:09:00	3231	2	0	0
2	112387248235954071	21:10:08	21:10:08	3232	3	0	0
3	112387248235954071	21:11:00	21:11:00	3233	4	0	0
4	112387248235954071	21:11:43	21:11:43	3239	5	0	0
...	...	...	...	...	...	...	...
2820504	113028649236519600	07:29:00	07:29:00	6427F	17	0	0
2820505	113028649236519600	07:30:00	07:30:00	6430F	18	0	0
2820506	113028649236519600	07:31:35	07:31:35	5066F	19	0	0
2820507	113028649236519600	07:33:00	07:33:00	5068F	20	0	0
2820508	113028649236519600	07:34:00	07:34:00	6361	21	0	0

2820509 rows × 7 columns

Pandas dataframe for the file stop_times.txt

We talked about this file in the presentation of all the GTFS files**.**
In this file, we had for each *trip_id* various *stop_id* and corresponding *arrival/departure time*. We did data treatments on this dataframe. Indeed, we shifted by 1 row the variable *arrival_time* in order to create a new variable named '*arrival_time_of_previous_stop*'.

By shifting, for each *trip_id*, we had the first row which was NaN (Not Available). Indeed, the first stop does not include previous stop time. Henceforth, all the other values correspond to the previous row.

Here is a example for a trip_id in the dataframe:

	trip_id	stop_id	stop_sequence	arrival_time	arrival_time_of_previous_stop
13900	112489651235954071	2831	1	08:05:00	NaN
13901	112489651235954071	2269	2	08:06:22	08:05:00
13902	112489651235954071	2587	3	08:07:11	08:06:22
13903	112489651235954071	1030	4	08:09:00	08:07:11
13904	112489651235954071	2271	5	08:11:14	08:09:00
13905	112489651235954071	2088	6	08:12:22	08:11:14
13906	112489651235954071	2237	7	08:14:25	08:12:22
13907	112489651235954071	2238	8	08:16:01	08:14:25
13908	112489651235954071	2239	9	08:17:13	08:16:01
13909	112489651235954071	2898	10	08:19:00	08:17:13
13910	112489651235954071	6459	11	08:21:00	08:19:00
13911	112489651235954071	2655	12	08:25:00	08:21:00

New dataframe of all the trip routes

In the future, we will refer to routes of trip_id by the dataframe. This dataframe is at the center of our results for many questions of this report.

Next, with this new dataframe, we were able to compute the travelling time between each *stop_id* by calculating the difference between the two time variables (*arrival_time* - *arrival_time_of_previous_stop*). In order to do this computation, we cautiously put those variables in the datetime format. We used the *apply* function to apply treatments and computations for each row of this dataframe.

What we did to get the distance:

Now, let us see how we have computed the precise distances between each stop station in the route of each vehicle id (previous dataframe).

Starting from the previous dataframe, we joined it with the file named *trips.txt* by the key variable *trip_id*. Hence, we had the exact route for each vehicle. Moreover, with this new file, we also had other variables like the line id, the terminal station and the id of each stop crossed.

Afterward, with the file named *ACTU_STOPS.shp*, we were able to find the exact position of each stop. The position was not defined with latitude/longitude but with coordinates relative to the Coordinate Reference System (CRS) of the city of Bruxelles.

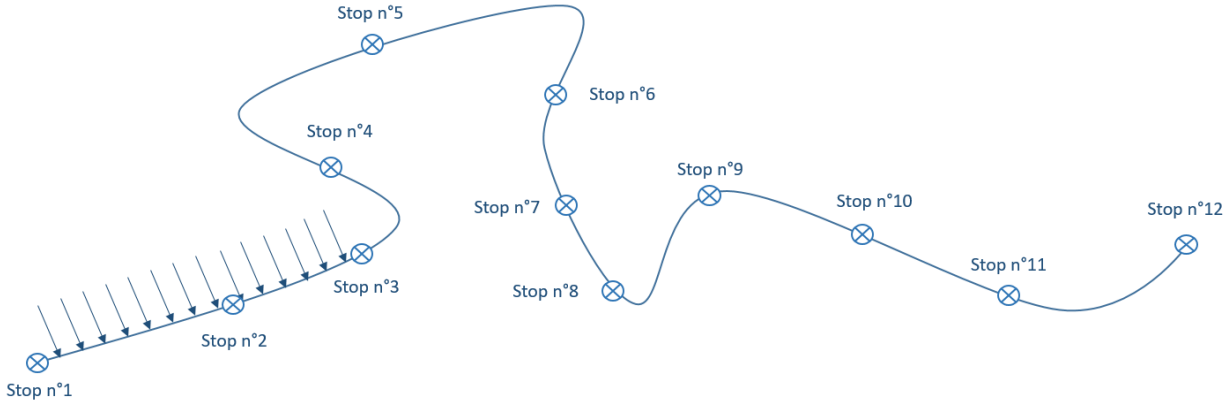
Similar to the transformation for the time, for each *trip_id* we shifted by 1 row the coordinate positions in order to create a new variable '*position_of_last_stop*'. Hence, we were able to

compute the distance by comparing for each row the position of the actual stop with the position of the previous stop.

However, it was just an euclidean distance (straight line) between two points. It does not give the actual distance of the route of the vehicle. For the actual distance we needed to find the precise route of a vehicle between two stop stations. For this, we used the file named *ACTU_LINES.shp* where for each line we had a huge number of coordinates where the vehicle of the line crossed.

In order to determine the different stops in this file, we found the point in the line (in *ACTU_LINES.shp)* that was the nearest from the stop coordinates found in *ACTU_STOPS.shp*. Then, to compute the exact distance between two stops, we computed the sub-distances between all the sub-points in the line.

Let's look at this schema to understand the situation:



Schema for a line in *ACTU_LINES.shp*

You see in the image a line with different stop stations. It's not an actual line, it's just a schema to summarize the situation. The arrows of this schema are all the sub-points of the file *ACTU_LINES.shp* for a specific line. You have arrows all over the line (not fully present in this schema). Hence, when we found the arrow which was the nearest from a stop. Then, we were able to compute precisely the distance between each stop of this line, by computing the distance between all the arrows from one stop to another. It provides the exact route of the vehicle and not just a straight line between stops.

Finally, at that step, we had the distances and the travelling time. With that information, we were able to compute the speed in km/h.

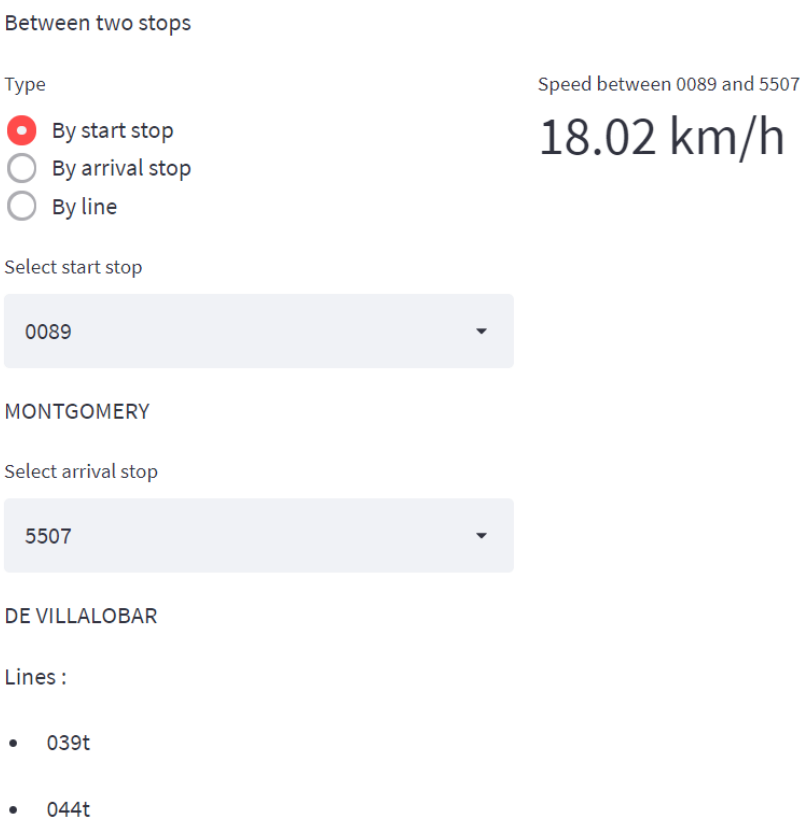
4.1.2 Results

Now let us look at the results of this first analysis.

All the results will be present to you in depth during our presentation. As you know, we have developed a web application in an interactive dashboard format. Then, it's more suitable to present our results face to face.

But in the meantime, let us look at some results. We have created two types of statistics for speed: speed between two stops and the average speed along a line.

First, in our web application, we have the average speed between two stop stations. Let us see an example for the average speed between Montgomery and De Villalobar.

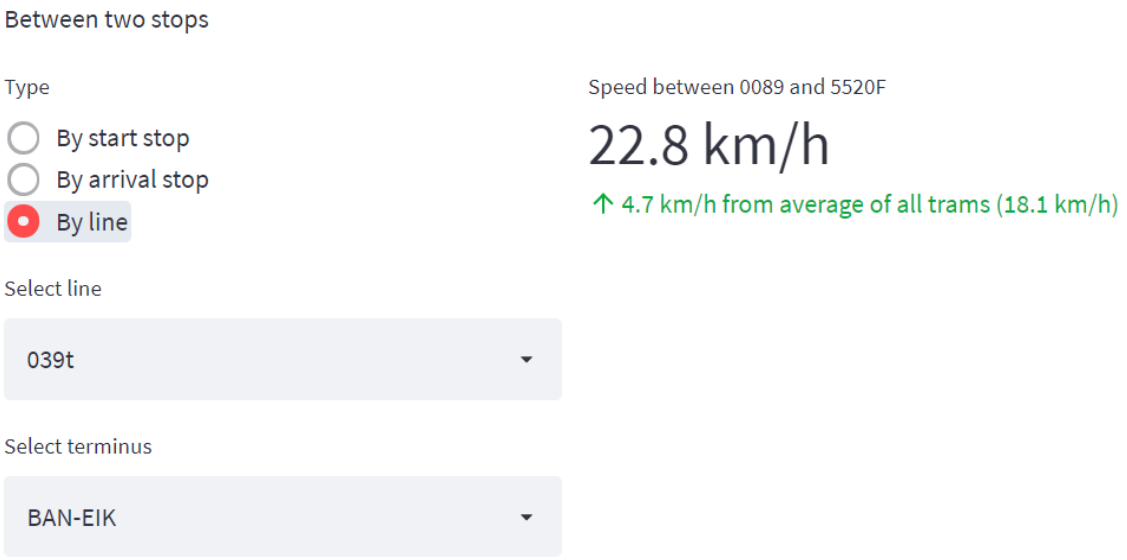


Screenshot from the application

For this example, we found two lines which follow this route (e.g from Montgomery to De Villalobar). It's the lines n°39 and n°44 (tram). As you can see the average speed between these two stop stations is 18.02 km/h.

Moreover, in our application, we have two modes for computation of the speed between stops. First, '*by start stop*' where we indicate a departure station. Then the app shows all the arrival stops accessible from the departure stop.
Second, '*by arrival stop*' where we indicate an arrival stop station. Then the app shows all the start stops which go to this departure stop.

Finally, our app can show the average speed for a line. This is accessible with the mode named *'By line*'. This speed is computed by aggregating the average speeds for all the stop present from the first stop to the last stop of a line.



Screenshot from the application

For this example, we chose the tram line n°39 in the direction of Ban-Eik. The app shows that the average speed of this line is 22.8 km/h.
Also note that the average speed for a bus in Brussels is 21.53 km/h, for metro it is 28.85 km/h and for tram itis 18.10 km/h. Hence, the app can show if a specific line speed is higher or lower than the average speed for the vehicle type (bus, metro, tram). In this particular example, the speed is higher (by 4.7 km/h) than the average speed for all the trams.

Finally, we note that for calculating the speed between two stops, we were not able to exclude the time when the vehicle was stopped. Therefore, all the calculations are underestimated.

4.2 Analyze the vehicle delays at the different stops, how it varies across stops, and over time. Present this in a suitable visual way.

For this question, our approach was to work with a joined data from GTFS static and GTFS Vehicule Position.
The first file is taken from the previous question, which we later name *gtfs_df*.
The second file is a concatenated version of Vehicle position file that compiles the 13 vehicle positions files. Later, at the first stage of the analysis, this file will be named *Vp_df*.

Vp_df has 19 421 883 rows and 6 columns while *gtfs_df* has 2 496 558 rows and 33 columns. These files were joined in a global file named *Final_df.* We will present later in this part how we have joined these two files.

Obviously, the joined file has millions of rows and many columns, therefore, working with pandas dataframes will slow down the process. Therefore, we used *PySpark* from *Spark* to calculate the delay as it is better in handling Big data queries.

We first convert each data part into RDD data frames from spark context, and later join them together.

Then, the joint was conducted at *stop_id* level (for *gtfs_df* side*)* and at *pointId* level (for *Vp_df* side).

After some data cleaning and field selection, we ended with the following dataframe:

trip_id	lineId	stop_id	directionId	distanceFromPoint	descr_fr	VpTime	arrival_time	Code_Ligne
112537487236801504	80	1159	4952	117	GAULOIS	08:00:02	05:21:15	080b
112537489237642504	80	1159	4952	117	GAULOIS	08:00:02	06:25:15	080b
112537490236801504	80	1159	4952	117	GAULOIS	08:00:02	21:47:52	080b
112537493236801504	80	1159	4952	117	GAULOIS	08:00:02	23:46:15	080b
112537496237642504	80	1159	4952	117	GAULOIS	08:00:02	06:53:15	080b
112537500236801504	80	1159	4952	117	GAULOIS	08:00:02	24:06:15	080b
112537504236801504	80	1159	4952	117	GAULOIS	08:00:02	08:35:34	080b
112537509236801504	80	1159	4952	117	GAULOIS	08:00:02	07:23:15	080b
112537513236801504	80	1159	4952	117	GAULOIS	08:00:02	07:53:15	080b
112537514236801504	80	1159	4952	117	GAULOIS	08:00:02	12:30:52	080b
112537515236801504	80	1159	4952	117	GAULOIS	08:00:02	13:30:52	080b
112537516236801504	80	1159	4952	117	GAULOIS	08:00:02	05:53:15	080b
112537518236801504	80	1159	4952	117	GAULOIS	08:00:02	15:10:52	080b
112537519236801504	80	1159	4952	117	GAULOIS	08:00:02	09:06:34	080b
112537520237642504	80	1159	4952	117	GAULOIS	08:00:02	12:09:52	080b
112537521237642504	80	1159	4952	117	GAULOIS	08:00:02	10:48:52	080b
112537522237642504	80	1159	4952	117	GAULOIS	08:00:02	10:06:52	080b
112537524236801504	80	1159	4952	117	GAULOIS	08:00:02	11:49:52	080b
112537525236801504	80	1159	4952	117	GAULOIS	08:00:02	10:27:52	080b
112537526236801504	80	1159	4952	117	GAULOIS	08:00:02	09:46:52	080b

only showing top 20 rows

Joint dataframe used for delay calculation

Delay calculation:

For delay calculation, we calculated the delay per each stop in an entire trip of a vehicle.
The trip selected here is trip number *112537487236801504*, this goes through line 80 taken by a bus.

The stops served by the bus in this trip are displayed below:

Stop names in Fr	Delay in sec
HEYDENBERG	4
DIABLOTINS	80
CARENE	36
PORTE DE NAMUR	1
BORDET STATION	100
FEVRIER	23
JULES BORDET	133
DEGROOFF	83
CIM. DE BRUXELLES	49
MUSEUM	18
MERODE	8
GAULOIS	47
TRONE	19
DA VINCI	109
LEMAN	26
COLONEL BOURG	11
JOURDAN	332
IDALIE	67
MONTGOMERY	32
PADUWA	2
MEUDON	6
GEORGES HENRI	23
SCIENCE	97

Bus delay per stops in the selected trip

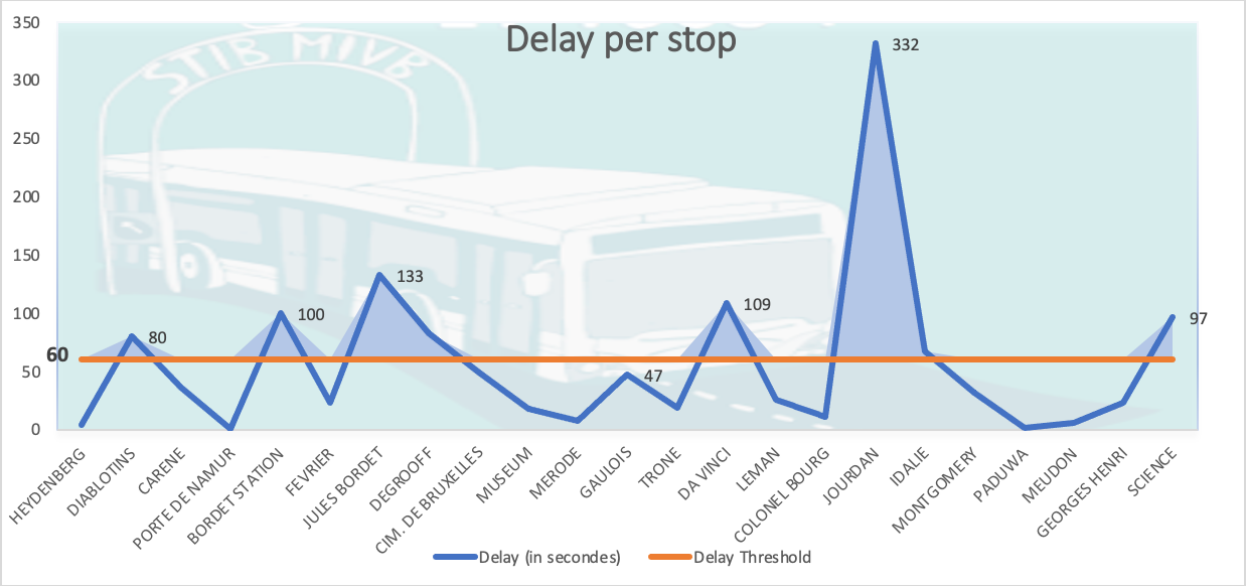
Particularly, the delay was calculated by taking the minimum absolute value of the difference time between the real time vehicle position and the scheduled time in the planning when the Bus is in the stop itself (i.e. *distanceFromPoint = 0*). We are indeed aware that the bus could arrive earlier instead of being late, we ran out of time to further explore both delay and time advance.

The same analysis could be replicated on any trip served by any other vehicle type. We showed the answer filtered on the trip_id mentioned above as our chosen way to better visualize our answer.

In terms of STIB clients', the delay is only sensitive after a certain time. Indeed, if the bus comes only 5 seconds late, this most of the time will not even be noticed. However, if the delay is one minutes or more, then it becomes "emotional waiting" for the client.

We took the initiative to set 1min as a "real delay", we therefore can distinguish "noisy delay" from "real delay".

Following this rule, the chart below shows that during its trip, the bus was only late 6 times with a major delay of 5,5 min in *Jourdan*.



Visualized delay and “real delay” (i.e. after threshold) of the bus per stops in the selected trip

4.3 Given a vehicle start time, do arrival time forecasting at a given stop in the route of this vehicle. You should be able to test the accuracy of your forecasting by randomly splitting the given dataset in disjoint training and testing subsets.

For this question, we reused our dataset from the first question in which we calculated for each trip the distance between each of its stops.

Let us remind that the dataframe looks like this:

	trip_id	arrival_time	departure_time	stop_id	stop_sequence	trip_headsign	Code_Ligne	descr_fr	descr_nl	terminus	...
0	112360041233813600	05:32:00	05:32:00	0089	1	BAN-EIK	039t	MONTGOMERY	MONTGOMERY	BAN-EIK	...
1	112360041233813600	05:33:02	05:33:02	5501	2	BAN-EIK	039t	GJ MARTIN	GJ MARTIN	BAN-EIK	...
2	112360041233813600	05:34:00	05:34:00	5502	3	BAN-EIK	039t	LEOPOLD II	LEOPOLD II	BAN-EIK	...
3	112360041233813600	05:35:41	05:35:41	5503	4	BAN-EIK	039t	JULES CESAR	JULIUS CAESAR	BAN-EIK	...
4	112360041233813600	05:37:11	05:37:11	5504	5	BAN-EIK	039t	CHIEN VERT	GROENE HOND	BAN-EIK	...
...	...	...	...	...	...	...	...	...	...	...	...
2496553	113549953238322602	10:57:03	10:57:03	3018	2	BRUSSELS AIRPORT	012b	BOURGET	BOURGET	BRUSSELS AIRPORT	...
2496554	113549953238322602	11:05:00	11:05:00	9600B	3	BRUSSELS AIRPORT	012b	BRUSSELS AIRPORT	BRUSSELS AIRPORT	BRUSSELS AIRPORT	...
2496555	113549954238322602	11:25:00	11:25:00	5044	1	BRUSSELS AIRPORT	012b	DA VINCI	DA VINCI	BRUSSELS AIRPORT	...
2496556	113549954238322602	11:27:03	11:27:03	3018	2	BRUSSELS AIRPORT	012b	BOURGET	BOURGET	BRUSSELS AIRPORT	...
2496557	113549954238322602	11:35:00	11:35:00	9600B	3	BRUSSELS AIRPORT	012b	BRUSSELS AIRPORT	BRUSSELS AIRPORT	BRUSSELS AIRPORT	...

2496558 rows × 32 columns

Dataframe from question 1

We came up with two methods to solve this question. First, we can look at an existing trip in the data frame. Second, we can use a Machine Learning model to make predictions on an existing trip but with a new schedule.

In the first method, the user indicates the departure time, a start stop, an arrival stop and a line id. Using this information we can find one or more vehicles which make this trip at a particular time. Then, we were able to look at the arrival time.

This method has the advantage of not making errors in the computation of the arrival time since we used existing trips between two stops which had already occurred. However, this is not the smartest way because any future user will be only able to select a time which is present in the data frame. Hence, a Machine Learning model can deal with new departure times and it is much more flexible. You forgot this model in our web application. Instead, we have used the following method.

Next, we used methods based on Machine Learning algorithms. We try using different variables in our model. At the end, we selected the following variables :

- Time start
- Start stop
- Arrival stop
- Duration for the trip (in seconds): this is the variable to predict

For the data preprocessing step, we transformed the *stop_id* for the stop and the arrival stop in integer type e.g ‘0089’ became 89 and ‘5281G’ became 5281. For the time start, we also transformed it into integer e.g number of seconds from 1 January 1900. These transformations were necessary for the models.

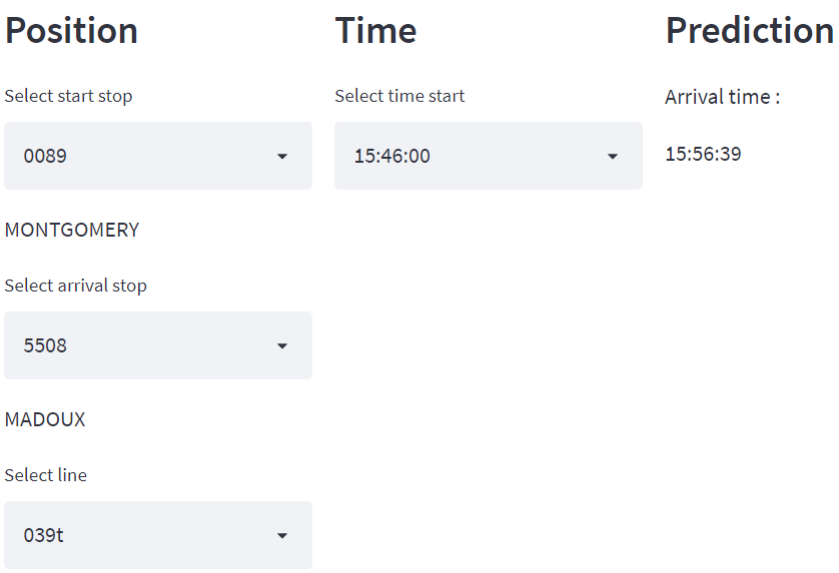
Our first model was a Linear Regression. We used *Sklearn* to load the model. Then we tried two types of splitting. The first one is a random splitting throughout the dataset. The size of the dataframe was 2.5 millions rows so we kept 5% to test and 95% to train the model. After training the Linear Regression model. We test it on the test set. We got a MAE (Mean Absolute Error) of 32 seconds.

For the second method of splitting, we isolated one line for the test set and we used the other lines for the training (like a k-fold split but with the lines). We did the process of split, train and test for each line. The mean error (MAE) through each line was 31.5 seconds. 31.5 seconds was the absolute gap between predicted time and real time. The standard deviation through the different splits was nearly 0.5 seconds.

Then, we also used Random Forest with the two same techniques of splitting. It did work better than the previous model, with a MAE of 14.5 seconds for the random splitting and 21.5 seconds for the leave one line out splitting.

Finally, we used the Deep Learning method: Neural Network. For that, we used the *Tensorflow* framework. For this method, the results were not as good as the previous ones. The MAE was around 50 seconds of error. In the future, we have kept the Random Forest for the prediction in our web application.

Further, we could have used and tested numerous other algorithms like using these 3 algorithms in an *Ensemble Learning predictor*. We could also try to fully optimize these methods by tweaking the hyper-parameters of the models. However, we thought it would be more interesting to put our model in life. In order to do that, we have created a web application in where the user can interact with the model by indicating a start stop, arrival stop, a date and a time start.



Part 3 of our web application

For example, here we have compared the predicted time (figure above) with the web site of STIB for the same trip (same start time, from Montgomery to Madoux with the tram n°39):

Example from the STIB website

The prediction is not bad: only 18 seconds less than the real time.
We will try other predictions as part of the demonstration of our app during the presentation.

For the Machine Learning model, we trained a model using the training dataset (95% of the rows of our global dataset). In this dataset, we had the *stop_id*, the *departure time* from the stop_id, the *next stop_id* and the *arrival time* to the stop_id.

If a trip_id crossed 3 stops, we had 2 records in our dataset :

1. Stop id n°1, departure time, Stop id n°2, arrival time
2. Stop id n°2, departure time, Stop id n°3, arrival time

The model worked well when we evaluated it on data which were side by side in a trip route (example: travel time from stop n°1 to stop n°2). However, when we tested our model on stops which were not side by side (example: travel time from stop n°1 to stop n°8) the model did not perform well.

Hence, we used a trick to solve the problem of performance. In order to predict the travel time from stop n°1 to stop n°4, for example, we did 3 predictions:

1. Prediction of travel time from stop n°1 to stop n°2
2. Prediction of travel time from stop n°2 to stop n°3
3. Prediction of travel time from stop n°3 to stop n°4

Using this method, the predictions were closer to the training data frame since we only made predictions on stop stations which were side by side in the route of the vehicle.
Finally, we summed the different times and printed the prediction.

4.4 The GPS tracks are for real people moving in Brussels. In fact they are from Mahmoud and Jean-Philippe. You are asked to infer the mode of transport of each of these tracks (bus, tram, etc)

For this question, we were provided some 9 trips from the professors. For each trip, we got the GPS coordinates and the time. These trips have been made by bus, tram, metro or other (by walking, car, etc).

We were asked to infer on the trip mode of transportation.

First we have computed for each row in the GPS tracks, the nearest *stop_id*, by comparing the coordinates with the position of the different stop stations (found in question n°1).

A first method was to look for each *TrackId:*

- A trip in the calendar that begin when the TrackId begin (at a specific stop station)
- A trip in the calendar that finish when the TrackId finish (at a specific stop station)

But this solution did not work. We couldn't have decent results.

Our final solution was to keep the nearest stop for each row of the TrackGPS file. Then we deduced the line (for example '046b'). Hence we got a transport mode.

Then for each *TrackId* we look at the first 5 rows and the last 5 rows, and we count the repartition of the transport mode.
If a transport mode is present more than 70% in the first 5 rows and more than 70% in the last 5 rows, we predict this transport mode (*bus*, *tram* or *metro*). Else, we predict *other* (driving car or

walking). We chose 5 rows because it’s the time when the professor starts the recording of his sensor. But it can be more than 5 rows. The thing is that we didn't know exactly when the record started, if it is when the vehicle started moving or before. In our future presentation, we will show you different values for the number of rows taken into account.

Let us see our results for this question.

TrackId : 1
- First 5 rows
  - Bus : 5 (100.0%) -> (prediction)
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

TrackId : 3
- First 5 rows
  - Bus : 3 (60.0%)
  - Tram : 1 (20.0%)
  - Metro : 1 (20.0%)

-> prediction : other
- Last 5 rows
  - Bus : 3 (60.0%)
  - Tram : 2 (40.0%)

-> prediction : other

TrackId : 4
- First 5 rows
  - Bus : 5 (100.0%) -> (prediction)
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

TrackId : 5
- First 5 rows
  - Bus : 3 (60.0%)
  - Tram : 2 (40.0%)

-> prediction : other
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

TrackId : 6
- First 5 rows
  - Bus : 4 (80.0%) -> (prediction)
  - Tram : 1 (20.0%)

- Last 5 rows
  - Bus : 4 (80.0%) -> (prediction)
  - Tram : 1 (20.0%)

TrackId : 7
- First 5 rows
  - Bus : 5 (100.0%) -> (prediction)
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

TrackId : 8
- First 5 rows
  - Bus : 5 (100.0%) -> (prediction)
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

TrackId : 10
- First 5 rows
  - Bus : 5 (100.0%) -> (prediction)
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

TrackId : 11
- First 5 rows
  - Bus : 4 (80.0%) -> (prediction)
  - Tram : 1 (20.0%)
- Last 5 rows
  - Bus : 5 (100.0%) -> (prediction)

Then TrackId 1, 4, 6, 7, 8, 9, 10 are bus and 3,5 are other.

We had other ideas, but didn't have the time to try it. For example, we can also compare the departure time with the line in the calendar, or compare the speed of question n°1 with the speed for the data of this file. Lastly, it could also be good to try to make a Machine Learning model like a KNN (k-nearest neighbors) for example.

4.5 Think your own of a valuable analysis on this data

For this part, we did a web application composed of dashboards for each question.

For that, we have used the framework *Streamlit* which is a Python framework where we can easily make web applications. You can see the documentation of Streamlit by clicking here.

With this application everybody who is extern to the project can have a summary of our results for each question. We have made this Web application to present our results in an interactive way. It's more fun and interesting than a static war with numerous graphs and plots.

5. A presentation and a demo of your solution

We showed you many figures from our application through this report. We will make a presentation of our application during the presentation, more in depth. You will not have access to this app in your computer to this application because we didn't have the time to "put it in production" e.g accessible from a web link.
We will present to you the app by viewing a localhost link on one of our computers.

6. Conclusion

STIB needs to improve some points in their database. For instance, the json files where we have records for every 30 seconds aren't labeled with the trip_id or any identifier of the vehicle. It was very challenging to find the trip and their real stop schedules (i.e. it is not clear which schedule a vehicle is supposed to follow).

Another related challenge we came across is selecting the planning or the schedule for a certain vehicle. The workaround used is to base our approach on the smallest time difference compared to the planned arrival time when the moving vehicle is at stop.
We therefore suggest populating a *vehicle_id* field in the real time vehicle position to make the matching task with the static GTFS files easier.

If any further analysis is to be done based on this work, we suggest verifying the delay per vehicle type (i.e. metro, bus, tram). We expect metros to have the least amount of delay and buses to have the highest.
Similarly, the data files we received didn't include the *alerts_file.* For the *trip_id* we used, this indeed could have helped analyze any effect of *Detour* for a certain trip_id and compare the delay and the speed with a trip without any *Detour* to see how the amount of delay and speed differ. Moreover, this could also be reflected in the case where the vehicle position is far from its assigned trip in *trip_shape.*

Moreover, the trips data and their calendar schedule reflected the same values for both departure time and arrival time. This was very likely done assuming theoretically no amount of time between the moment the vehicle arrives at stop and the moment it departures from it.
It would be more precise for the computation of the speed. Here, the speed is under-estimated between the time when the vehicle was at the stop and then wasn't moving.

Accurately modelling the vehicle speed/delays along with enhanced information on departure and arrivals in public transport systems can help directly improve current passenger service and operating efficiency. Moreover, it can pave the way for controlling future technology automated vehicles, which will share the road infrastructure with other vehicles.