Hack My Ride

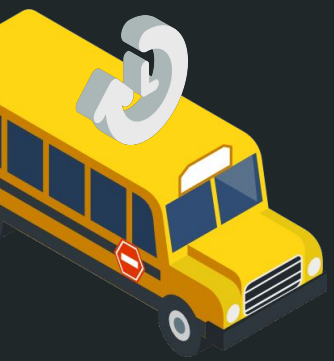
INFO-H-423 Data Mining

Students:

- Gega, Jezuela (ID 000561082)
- Lorencio Abril, Jose Antonio (ID 000559831)
- Bagus Wikacsono, Satria (ID 000558809)
- Yusupov, Sayyor (ID 000558672)

Professor:
Sakr, Mahmoud

STIB Representative:
Baret, Justine



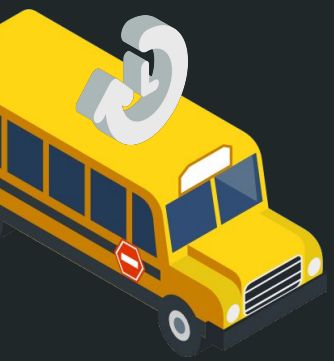
Overview

Overview

1. Understanding the data.
2. Preparing the data:
    a. Creating the schedules and classifying points for regularity or punctuality
    b. Computing SWT
    c. Dealing with 0/1 lines
    d. Approximating departure time
    e. Computing AWT
3. Analyzing the data:
    a. Computing EWT
    b. Computing Punctuality Index
    c. Monte-Carlo simulation: people arriving to a stop
    d. Sequential pattern mining
4. Conclusions
5. Dashboard time!

Tools Used

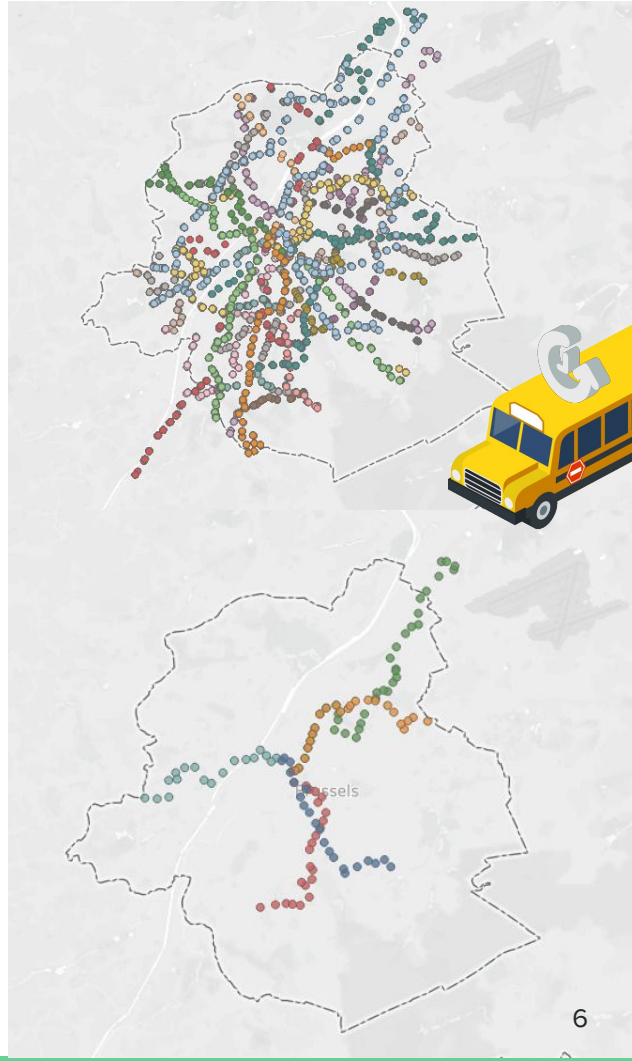
- Understanding, Analyzing Data: Python (Pandas, Numpy), PostGIS
- Processing: gtfs_kit
- Clustering: ckwarp, dbscan1d, jenkspy, sklearn
- Dashboard: Tableau
- Visualization: Python (Matplotlib), QGIS
- Pattern Mining: prefixspan
- Workspace: Google Colab, Google Compute Engine, Google Cloud Storage



Understanding the Data

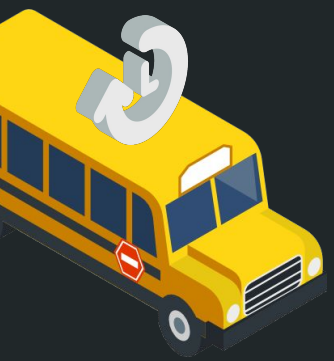
GTFS Data

- General Traffic Feed Specification data format
- Used to describe public transport:
  - routes
  - stops
  - stops_times
  - calendar and calendar_dates
  - shapes
  - agency and translation
  - trips
- Dealt with PostGIS and Python gtfs_kit
- gtfs3s goes from 2021-08-23 to 2021-09-19
- gtfs23s goes from 2021-09-20 to 2021-11-17



Measured data

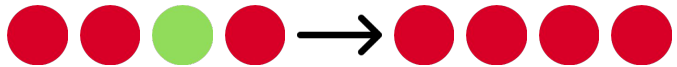
- Measured every ~30 seconds
- From 2021-09-06 around 10:00 AM to 2021-09-21 around 6:30 PM
- At each timestamp, all active trips report:
  - the timestamp
  - last stop_id visited
  - distance from that stop
  - the line (using the route_short_name)
  - direction of the bus
- Bus, night bus, metro and tram

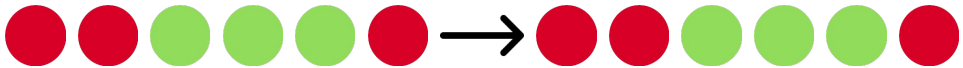


Processing the Data

Creating the schedules and computing SWT

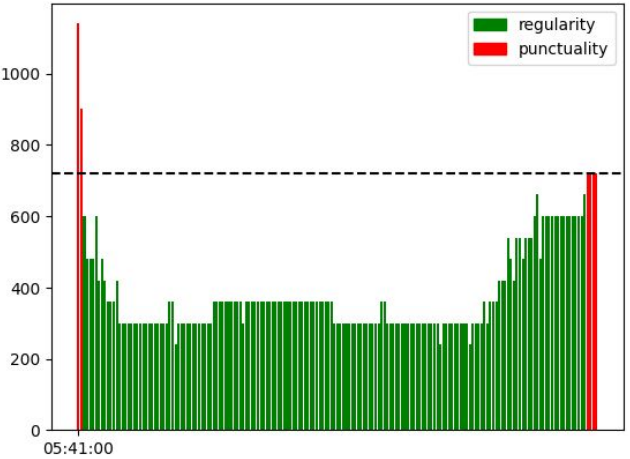
- Python library **gtfs_kit**
- Take from 6 Sept - 19 Sept from **gtfs3s** and 20 Sept - 21 Sept from **gtfs23s**
- Get the schedules for **each line in each day** automatically with gtfs_kit and more:
  - Modify the **stop_id** to match real data (e.g. 01234F → 1234)
  - Correct the scheduled **departure_time** (e.g.: 09-06 25:27:01 → 09-07 01:27:01)
  - Compute the **headway** by sorting properly, shifting and difference
  - Get **candidates** for regularity and punctuality assessment:
    i. If headway < 12: candidate for **regularity**
    ii. else candidate for **punctuality**
  - **Classify** candidates into classes:
    i. If candidate is regularity: change it if is isolated
    ii. Else: maintain as it is

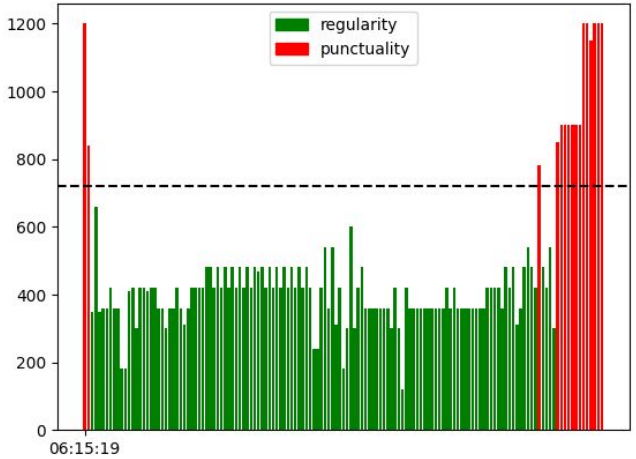


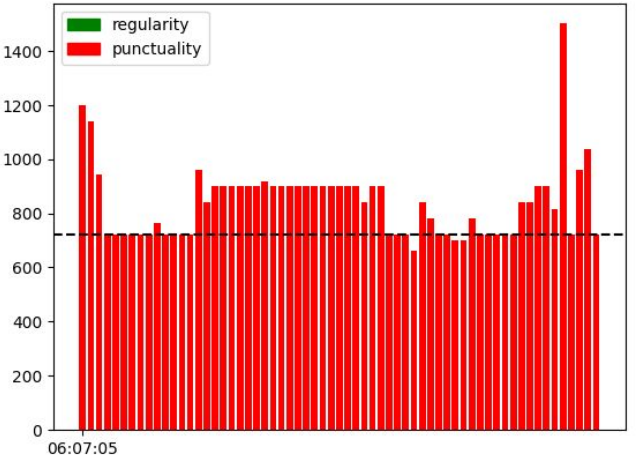


Computing SWT

If we focus on a particular stop, its classification over a day would look like this:







So we are interested now in computing the time intervals in which we will need to use regularity.

Computing SWT

For computing intervals:

- We have designed an **iterative algorithm**:
  a. Traverse schedules by stop_id, ordered by date, and datetime
  b. Keep track of the current interval information
  c. Identify when an interval ends, add this information to the dataframe, and restart the process with the next interval
- The **SWT** is calculated at the same time:
  a. Keep track of sum of headways and sum of headways squared
  b. When an interval ends:

$$SWT = SUM\_SQ / (2*SUM)$$

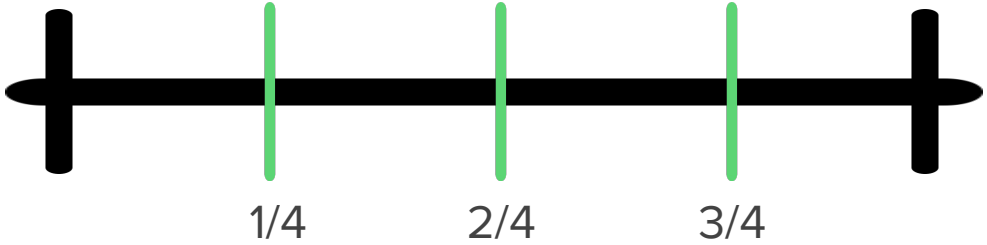
Preprocessing Real-Time Data

- Flattening from **JSON to CSV**. The structure of the JSON is:
  - timestamp in ‘Unix Epoch Time in Milliseconds’
    - Line (LineID)
      - Vehicle (directionID, distanceFromPoint, pointID)
- Set the right time-zone to **timestamps**
- Sort by **lineID**, **stopID**, **directionID**
- Removing technical stops: stopIDs/directionIDs that are not in gtfs data

Dealing with 0/1 lines

- **0/1 Lines** are those that do not contain distance to last stop, but only:
  - 0 if the vehicle is in the stop
  - 1 otherwise
- These are 1, 2, 5 and 6 (which happen to be metro lines).
- Our idea was to give an **estimated distance** to these records:
  - We compute distance between consecutive stops (**PostGIS**)
  - At **time** T, in **stop** S, for **line** L, we count how many vehicles have a 1. Say N
  - We assume they are **equally distributed along the line**, so if D = dist(S, S+1), the distances of the vehicles will be assumed to be

D/(N+1), 2D/(N+1), ..., ND/(N+1)



Approximating departure time

At this point, we have the distances at every timestamp of every vehicle to the last step it went through. But we need to know when the vehicle departed:

- We got the **average speed** of each type of vehicle from a STIB-MIVB report.
- This speed can be used to approximate when the vehicle left the stop:

approx_time = current_time - distance / speed

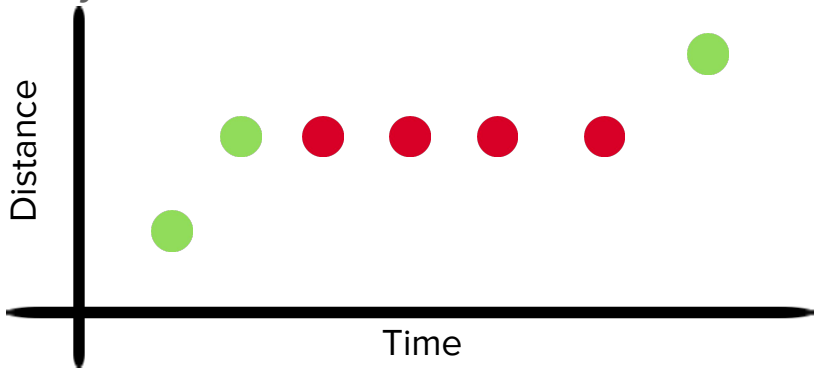
- Additionally, we interpolated the each non-zero distance data by calculating approximated time for -5 and +5 distance and then we sort the data based on the timestamp.
- Now, we have converted the dataset to a set of many approximated departure time.

Removing Consecutive Data

- We will delete consecutive data from the approximated departure time to minimize redundancy in the data.
- Consecutive data defined as follows:

data[i].distanceFromPoint = data[i+2].distanceFromPoint

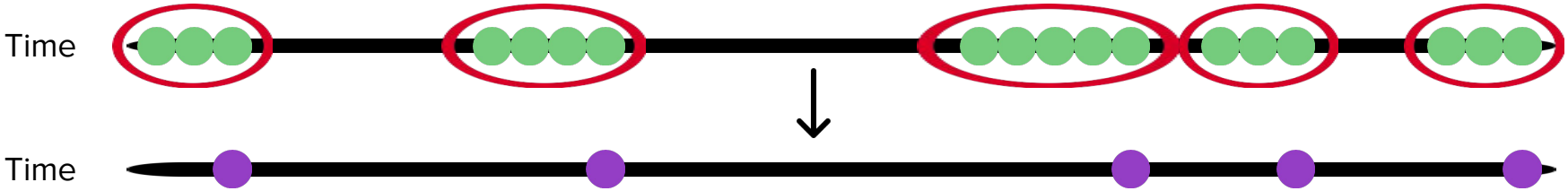
- Since we have interpolated the data for the approximation, each data point will produce 2 different rows, that's why we used index + 2 to detect the consecutive data.



Approximating departure time

We have to reduce the huge amount of departure times to match the schedules...

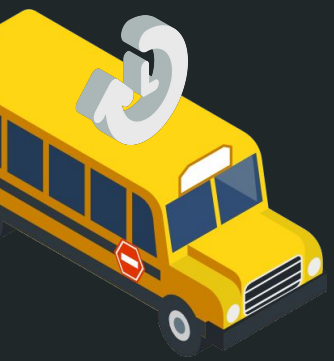
- From the schedules, we know the number of buses that go through each stop, S_j ➜ We filter out schedules that occurs before first and last approximated departure time
- Our idea is to use **clustering** to get, for each stop j, S_j clusters of time
- Then, we take the maximum of each cluster as departure time



Clustering method

- We have several criteria for our clustering method:
  - It must be fast enough to run on 1 dimensional data
  - The number of clusters should be close enough or the same as the number in the schedules
- We explored several algorithms for clustering
  - DBSCAN 1D ❌ (slow, number of clusters different from schedule, epsilon (max distance of neighbors) can be different for each line)
  - CKmeans 1D DP ✅ (fast, number of clusters the same as schedule)
  - Jenkin's Natural Breaks ❌ (fast, perform slightly worse than CKmeans)
  - Kernel Density Estimation ❌ (slow, number of clusters is really much smaller than we expect)
- Three different metrics is used to validate our clustering result

	Calinski-Harabasz	Davies-Bouldin	Silhouette
DBscan1D	1 137 938 883.98	0.09	0.91
CKmeans 1D DP	34 636 912.36	0.35	0.73
Jenkins' Natural Breaks	352 159.44	0.36	0.7
Kernel Density Estimation	239 180.41	0.41	0.64

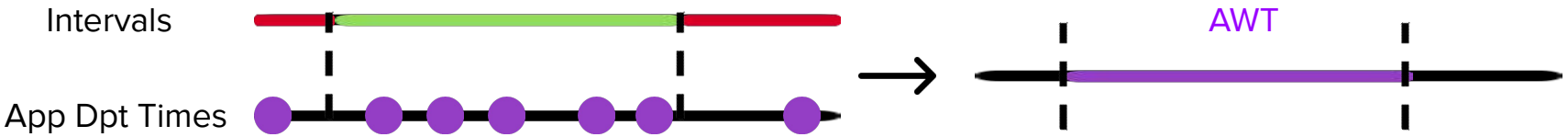


Analyzing the data

Computing AWT and EWT

Now that we have approximated the departure times for the measured data, we can compute the **Actual Waiting Times**:

- For each stop in each line, we order the timestamps and compute the **real headways**
- Once we have the real headways, we use the **intervals** computed from schedules. Focus on stop j:
  - If departure_time_j in interval k and interval k is ‘regularity’: we aggregate this values to compute AWT
  - The formula is the same as SWT. The difference is the matching of dept times with the intervals



Computing AWT and EWT

- Finally, to compute EWT, we just do

  EWT = SWT -AWT

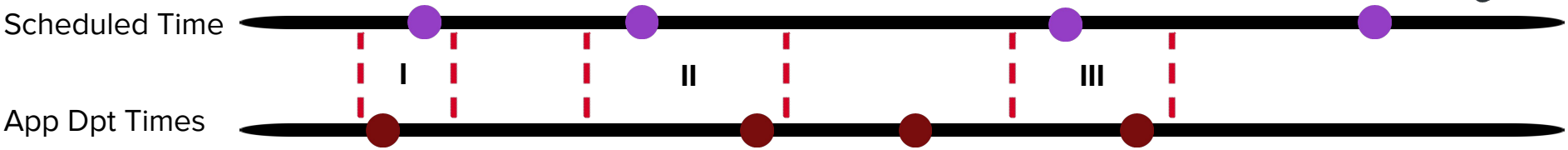
  for each interval.

Computing Punctuality Index

Punctuality is calculated by finding the nearest approximated departure time from the schedule.

- Filter schedule that has trip class defined as **punctuality** and also remove schedules that occur before first and last approximated departure time
- Then we pick **schedule_j** and find the first nearest approximated departure time from -1 minutes before until we find the first match.
- Then we calculated punctuality index with the formula

punctuality_j = (approximated_j - schedule_j) / headway_j



Monte-Carlo simulation: people arriving to a stop

We have simulated people arriving to a particular stop, to take a certain bus in a particular direction.

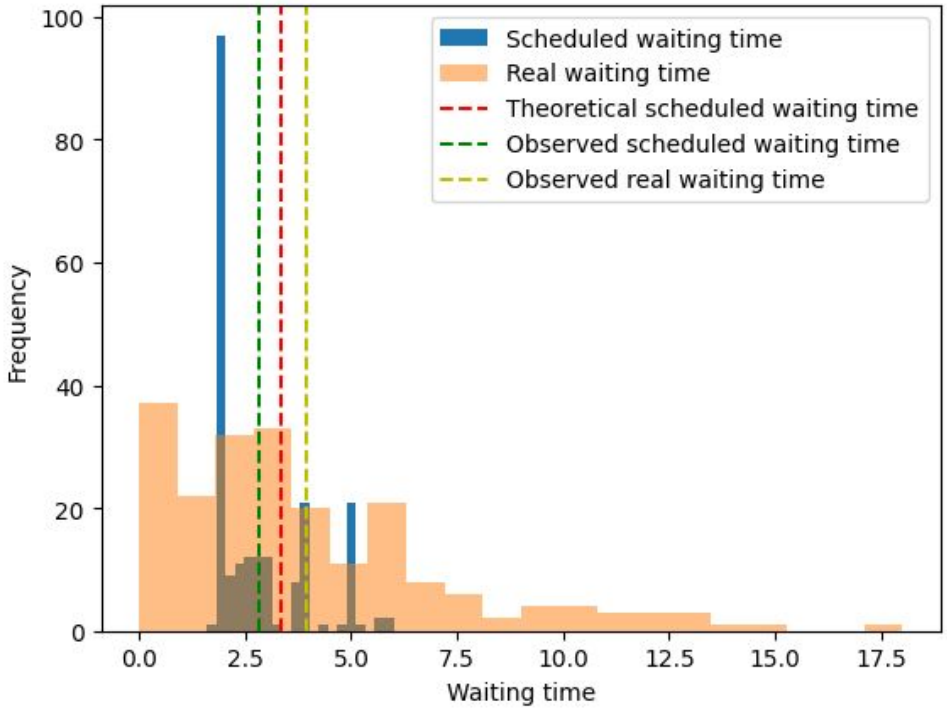
We have picked two scenarios from day 07th of September:

- **Regularity scenario**: line 71, stop TRÔNE, direction DE BROUCKÈRE, from 5:50 to 23:30
- **Punctuality scenario**: line 29, stop DAILLY, direction DE BROUCKÈRE, from 20:50 to 24:30

We have simulated 20.000 people arriving at each stop at a random time between the interval, generated with an uniform distribution.

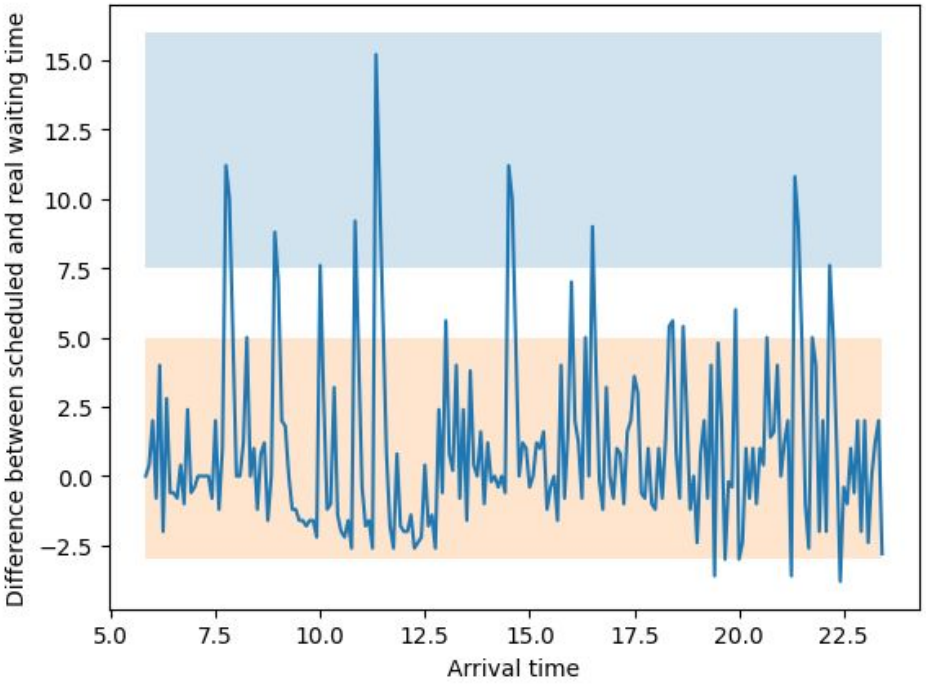
For each arriving person, we compute how much he/she expects to wait according to the schedule, and how much he/she actually waits.

Simulating people arriving at a stop: Regularity setup



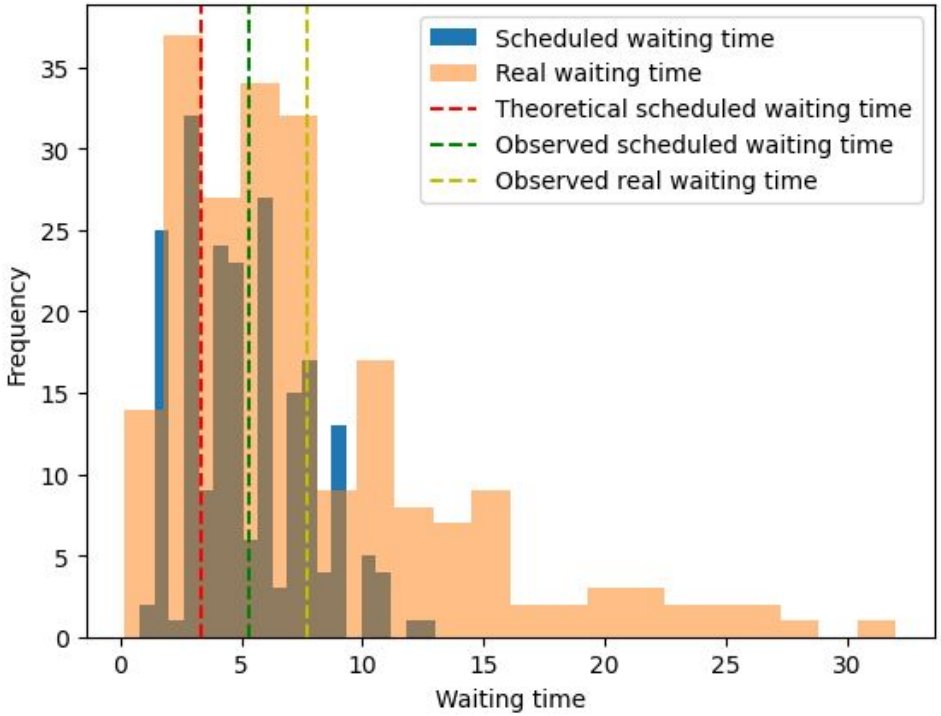
- Real waiting times are more **spread**:
  - More variable
  - 17.5 minutes? Something happened
- The **theoretical SWT** overestimates the average of the waiting times
- The **real waiting time** is bigger than both SWT and measured schedule waiting time

Simulating people arriving at a stop: Regularity setup



- If we plot the difference between the scheduled waiting time along the whole interval, and the real one, we can see:
  - Most of the time, people are waiting **less than 5 minutes** more than expected (orange zone)
  - There are some **problematic points** that would need further assessment, especially the point with more than 15 minutes of difference (blue zone)

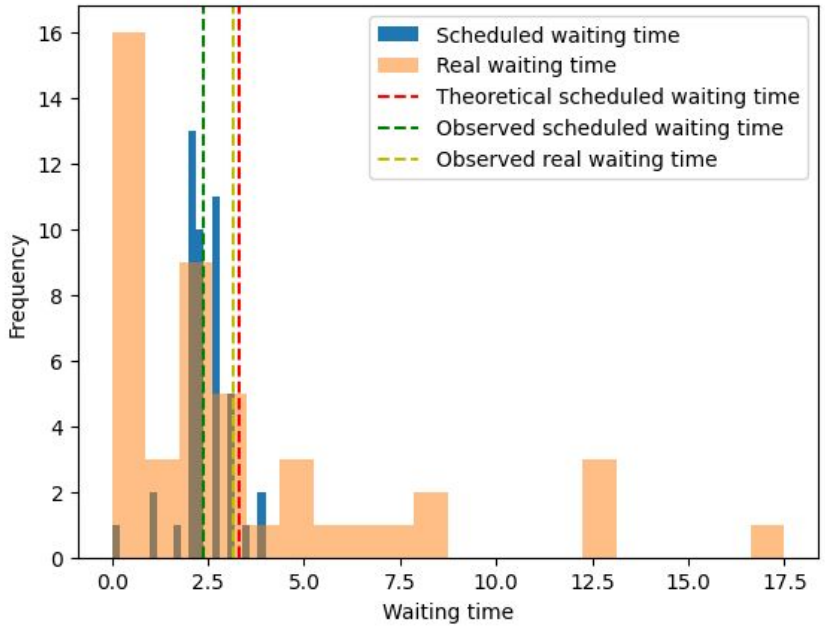
Simulating people arriving at a stop: Punctuality setup



- Again, real waiting time is **more spread** than schedule waiting time
- In this case, the SWT **underestimates** the average of schedule waiting times
- The **average of the real waiting time** is still over both the other values
- In this case, the **distribution of real times** is more condensed near the schedule one

Simulating people: focus on rush hours

We can model rush hours with a normal distribution with mean at 13:00 and std 30 min.



- In this case we observe that both the schedule waiting time and the real waiting time are smaller than the SWT: this is because the interval used for this was much longer.
- This indicate that **our intervals** should probably be subdivided, to better fit reality.
- Also, bus 71, at stop TRÔNE, direction DE BROUCKÈRE shows **good regularity**: only a few periods of 5 minutes show too high values for the real waiting time.

Sequence pattern mining

- We divide each day in **30 minutes intervals**.
- Only consider stops measured with **punctuality**.
- Only performed for **line 65** (the most punctuality-assessed one on our subset).
- At each 30 minutes interval, we compute the average difference between estimated departure time and schedule departure time.
  - If the **average is >10 min**, then we add the stop to the sequence.
  - When all intervals are computed, we **merge consecutive ones**.
  - Each group of intervals is thus a sequence, which is ordered by interval first, and then by the order of stops.
- We do the same for each day in the data, getting a database of **50 sequences**.
- Then, we run the **PrefixSpan** algorithm, to get frequent sequences of this kind.

Frequent Pattern Mining

We use the algorithm in the format **top-10**, getting the 10 most frequent patterns.

```
2980 : 35
2977 : 34
2977 6081 : 34
2980 2997 : 34
2997 : 34
3362 : 34
5361 : 34
6081 : 34
2980 6081 : 33
5361 6081 : 33
```

top 10 most frequent patterns

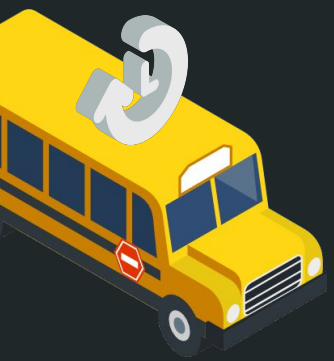
top 10 most frequent patterns with length >=4

```
2977 2980 2997 6081 : 33
2977 2980 2997 3361 : 32
2977 2980 2997 3361 6081 : 32
2977 2980 2997 5361 : 32
2977 2980 2997 5361 6081 : 32
2977 2980 3361 6081 : 32
2977 2997 3361 6081 : 32
2977 2997 5361 6081 : 32
2980 2997 3361 6081 : 32
2980 2997 5361 6081 : 32
```

```
2980 : 35
2977 : 34
2997 : 34
3362 : 34
5361 : 34
6081 : 34
2975 : 33
2976 : 33
2990 : 33
3361 : 33
```

top 10 most frequent patterns of length 1

- This is high indicator to pay **further attention to these sequences**.
- Specially:
  - Stops 2980, 2977, 2997, 6081, which appear together as the most frequent sequence, and they are also among the most frequent single sets



Conclusions

Limitations

- Departure Times are estimations based on external data (even if it is for STIB)
- Had to discard some data
  - current/ terminal stops of vehicles not in the scheduled data for that line
  - therefore, **14.7%** of scheduled trips could not be matched with real-time data
  - out of matched scheduled trips:
    - **4.9%** of punctuality records had **punc_idx** > 3, were truncated to 3
    - **25.4%** of regularity periods had **EWT** > 1000 seconds, were discarded
- More detailed information about technical stops is needed to correctly treat those cases
- A bus ID (or at least a trip ID) should be there! ☺

Can we do better?

- Subdivide **regularity intervals** in fixed time intervals (e.g. 30 minutes) or by some condition on a moving average (e.g. when the moving average change by a factor of X) or even using a clustering algorithm, over some transformation of the data.
- Analyze further some **assumptions**:
  - Regularity classification only depend on SWT?
  - Is our punctuality index a good punctuality indicator?
- Some **statistical analysis** could be also interesting:
  - How different is the distribution of real buses from that of scheduled buses?
- When dealing with real data, sometimes the **direction_id** (trip_headsign in schedule) is in a stop which is sooner than the usual ones. We removed those, but it is possible to match them, so they could also used in the analysis.

Thank you very much for your attention!!