MIVB–STIB: Analyzing Network Performance, Forecasting Delays and Predicting Passenger Behaviour

INFOH-423 – Data Mining

Students

Adam Broniewski | abro0001
Victor Diví | vdiv0001
Sergio Postigo | spos0001
Filip Sotiroski | fsot0001

Professor

Mahmoud Sakr

UNIVERSITÉ LIBRE DE BRUXELLES

December 2021

This report contains a static version of the python jupyter code and markdown required to complete the INFOH-423 Data Mining Project.

The report is broken up by section of the project:

- Assignment 1
- Assignment 2
- Assignment 3
- Assignment 4
- Visualization

The git repository is located at:
https://github.com/adambron-dot-com/STIB-Hach-My-Ride

assignment-1

December 20, 2021

1 Assignment 1

1.1 Calculating and Analyzing Vehicle Speed

1.2 What are we looking to accomplish?

Here we will be loading in files available from STIB API that provides timestamped locations of all vehicles on the STIB network. The data is provided in several reference files: - calendars and bus route data (.txt) - timestamp locations (.json) - bus stop and route locations are in shapefiles (.shp)

The data provided shows what line a bus is on, which direction it is heading (towards or away from the terminus), and which stop it is heading towards. A single snapshot in time provides all this info for every vehicle on the STIB network.

We will need to: 1. Load files in and manipulate them into a convenient working format (e.g. unpack the .json files) 2. Merge columns from different reference tables 3. Find a way to artificially "track" a vehicle along it's route 4. Calculate the vehicle speed using some segment of distance

Let's go!

[24]:
```
import datetime
import json
import os
from enum import Enum
from functools import reduce
from itertools import count
from operator import add
import numpy as np

import pandas as pd
import shapefile
from tqdm.notebook import tqdm
from numpy import inf

from scripts.helpers import *
```

1.3 Convert Raw JSON to CSV

Here we convert the raw JSON files containing vehiclePositions from STIB into a single CSV file. These files were generated by invoking the Vehicle Position Real-Time API of stib-mivb. Every 30 seconds, the API was called 9 times. Each call was for 10 lines IDs. The time attribute is the time in milliseconds (unix epoch) at which the API was invoked. The response array has the result of the 9 API calls.

The API calls were completed this way due to limitations on the number of lines that can be queried at a time. Multiple JSON files were created for redundancy's sake and for easier transfer between the original pull and project work.

We then iterate through the JSON files and extract the timestamp, lineId, directionId, distanceFromPoint, and pointId. This new CSV file becomes the base data for the remainder of the project.

Readsfrom: raw JSON files in `data/raw` folder (`data/raw/vehiclePosition*.json`)
Writes to: Single CSV file containing all the vehicle positions in `data` folder (`data/processed/assignment1/vehiclePositions.csv`)

[14]:
```
raw_json_files = [
    '../data/raw/vehicleTimestamp/vehiclePosition01.json',
    '../data/raw/vehicleTimestamp/vehiclePosition02.json',
    '../data/raw/vehicleTimestamp/vehiclePosition03.json',
    '../data/raw/vehicleTimestamp/vehiclePosition04.json',
    '../data/raw/vehicleTimestamp/vehiclePosition05.json',
    '../data/raw/vehicleTimestamp/vehiclePosition06.json',
    '../data/raw/vehicleTimestamp/vehiclePosition07.json',
    '../data/raw/vehicleTimestamp/vehiclePosition08.json',
    '../data/raw/vehicleTimestamp/vehiclePosition09.json',
    '../data/raw/vehicleTimestamp/vehiclePosition10.json',
    '../data/raw/vehicleTimestamp/vehiclePosition11.json',
    '../data/raw/vehicleTimestamp/vehiclePosition12.json',
    '../data/raw/vehicleTimestamp/vehiclePosition13.json'
]
vehicle_positions_csv = '../data/processed/assignment1/vehiclePositions.csv'
csv_header = ['Timestamp', 'LineId', 'DirectionId', 'DistanceFromPoint',␣
↪'PointId']
```

[15]:
```
with write_csv(vehicle_positions_csv) as writer:
    writer.writerow(csv_header)
    for raw_json_path in tqdm(raw_json_files):
        file = open(raw_json_path, 'r', encoding='utf8')
        data = json.load(file)['data']
        file.close()
        for time in data:
            timestamp = time['time']
            for response in time['Responses']:
                if response is None:
                    # Skip if response is empty
```

```
                    continue
                for line in response['lines']:
                    line_id = line['lineId']
                    for vehiclePosition in line['vehiclePositions']:
                        writer.writerow([
                            timestamp,
                            line_id,
                            vehiclePosition['directionId'],
                            vehiclePosition['distanceFromPoint'],
                            vehiclePosition['pointId'],
                        ])
```

```
  0%|          | 0/13 [00:00<?, ?it/s]
```

[16]:
```
vehicle_positions_df = pd.read_csv(vehicle_positions_csv)
vehicle_positions_df.head()
```

[16]:

	Timestamp	LineId	DirectionId	DistanceFromPoint	PointId
0	1630914886924	1	8161	1	8012
1	1630914886924	1	8162	0	8142
2	1630914886924	1	8162	0	8282
3	1630914886924	1	8731	0	8111
4	1630914886924	1	8162	1	8062

1.4 Shapefile to routes

Now we will transform the shapefiles by converting the raw stops Shapefiles from STIB to a single CSV file containing line routes. The ACTU_STOPS file provides information on the order in which a vehicle makes its stops. We will also merge the ACTU_STOPS shapefile with the stops.txt file to have access to GPS coordinates in 2 formats: - Belgian Lambert 1972 (EPSG 31370) - World Geodectic System 29814 (ESPG 4326)

This will be useful later on when calculating distances, as we can use euclidean distance calculations when dealing with Belgium Lambert 1972 projections.

Reads from: - Shapefiles in `data/raw/shapefiles` folder (`data/raw/shapefiles/ACTU_STOPS.*`) - `stops.txt` GTFS file in `data/raw/gtfs` folder

Writes to: Single CSV file containing all the line routes in `data` folder (`data/line_stops.csv`)

[13]:
```
csv_header = ['lineId', 'direction', 'stop_id', 'stop_id_int', 'name',␣
↪'name_ascii', 'lat', 'long', 'lambert_x',
              'lambert_y', 'order']
stops_shapefile_path = '../data/raw/shapefiles/ACTU_STOPS.shp'
stops_gtfs_path = '../data/raw/gtfs/stops.csv'
merged_stops_csv_path = '../data/processed/assignment1/line_stops.csv'
```

[14]:
```
stops_shapefile = shapefile.Reader(stops_shapefile_path)
# We take the first value of each field tuple (its name), and skip the first␣
↪field (DeletionFlag field, not relevant)
stop_fields = [field[0] for field in stops_shapefile.fields][1:]
shapefile_df = pd.DataFrame(stops_shapefile.records(), columns=stop_fields)
shapefile_df.head()
```

[14]:
```
  Code_Ligne  Variante  succession stop_id          descr_fr  \
0       012b         1           1   9600B  BRUSSELS AIRPORT
1       012b         1           2    3017           BOURGET
2       012b         1           3    5048          DA VINCI
3       012b         1           4    2695            GENEVE
4       012b         1           5    2250            MEISER

           descr_nl          alpha_fr          alpha_nl   coord_x   coord_y  \
0  BRUSSELS AIRPORT  Brussels Airport  Brussels Airport  157950.0  176429.0
1           BOURGET           Bourget           Bourget  154334.0  174200.0
2          DA VINCI          Da Vinci          Da Vinci  152934.0  173976.0
3            GENEVE            Genève            Genève  152428.0  172606.0
4            MEISER            Meiser            Meiser  152045.0  171508.0

  mode  numero_lig       terminus
0    B          12  BRUSSELS CITY
1    B          12  BRUSSELS CITY
2    B          12  BRUSSELS CITY
3    B          12  BRUSSELS CITY
4    B          12  BRUSSELS CITY
```

[15]:
```
shapefile_df.drop(columns=['descr_nl', 'alpha_nl', 'mode', 'numero_lig',␣
↪'terminus'], inplace=True)
renames = {'Code_Ligne': 'lineId',
           'Variante': 'direction',
           'succession': 'order',
           'descr_fr': 'name_ascii',
           'alpha_fr': 'name',
           'coord_x': 'lambert_x',
           'coord_y': 'lambert_y'}
shapefile_df.rename(columns=renames, inplace=True)
shapefile_df.sort_values(['lineId', 'direction', 'order'], inplace=True)
shapefile_df['stop_id_int'] = shapefile_df['stop_id'].apply(lambda stop_id:␣
↪int(stop_id[:4]))
shapefile_df.head()
```

[15]:
```
     lineId  direction  order stop_id        name_ascii            name  \
2987   001m          1      1    8733  GARE DE L'OUEST  Gare de l'Ouest
2988   001m          1      2    8742         BEEKKANT        Beekkant
2989   001m          1      3    8292     ETANGS NOIRS    Étangs Noirs
```

```
2990    001m          1      4     8282  COMTE DE FLANDRE  Comte de Flandre
2991    001m          1      5     8272  SAINTE-CATHERINE  Sainte-Catherine

      lambert_x  lambert_y  stop_id_int
2987   146633.5   170956.4         8733
2988   146776.5   171444.3         8742
2989   147492.7   171859.9         8292
2990   148013.6   171590.4         8282
2991   148539.5   171278.2         8272
```

[16]:
```
gtfs_stops_df = pd.read_csv(stops_gtfs_path)
gtfs_stops_df.dropna(axis=1, inplace=True)
gtfs_stops_df['stop_id_int'] = gtfs_stops_df['stop_id'].apply(lambda stop_id:␣
 ↪int(stop_id[:4]))
gtfs_stops_df.drop(columns=['stop_id', 'location_type', 'stop_name'],␣
 ↪inplace=True)
gtfs_stops_df.set_index('stop_id_int', inplace=True)
gtfs_stops_df.head()
```

[16]:
```
              stop_lat  stop_lon
stop_id_int
89           50.838006  4.408970
470          50.863666  4.329612
471          50.863732  4.329236
472          50.863543  4.329023
473          50.863418  4.330031
```

[17]:
```
joined_stops_df = shapefile_df.join(gtfs_stops_df, on='stop_id_int')
joined_stops_df.head()
```

[17]:
```
     lineId  direction  order stop_id         name_ascii              name  \
2987    001m          1      1     8733   GARE DE L'OUEST   Gare de l'Ouest
2988    001m          1      2     8742          BEEKKANT          Beekkant
2989    001m          1      3     8292      ETANGS NOIRS      Étangs Noirs
2990    001m          1      4     8282  COMTE DE FLANDRE  Comte de Flandre
2991    001m          1      5     8272  SAINTE-CATHERINE  Sainte-Catherine

      lambert_x  lambert_y  stop_id_int   stop_lat  stop_lon
2987   146633.5   170956.4         8733  50.848999  4.320948
2988   146776.5   171444.3         8742  50.853386  4.322974
2989   147492.7   171859.9         8292  50.857125  4.333143
2990   148013.6   171590.4         8282  50.854705  4.340542
2991   148539.5   171278.2         8272  50.851900  4.348012
```

[18]:
```
joined_stops_df[joined_stops_df['stop_lat'].isna()]
```

[18]:
```
      lineId  direction  order stop_id        name_ascii              name  \
3424   019t          1      1   5104F  GROOT-BIJGAARDEN  Groot-Bijgaarden
3467   019t          2     22   5169F  GROOT-BIJGAARDEN  Groot-Bijgaarden

       lambert_x  lambert_y  stop_id_int  stop_lat  stop_lon
3424   143429.4   172979.7         5104       NaN       NaN
3467   143385.5   172978.7         5169       NaN       NaN
```

[19]:
```python
file = open(merged_stops_csv_path, 'w', encoding='utf8')
joined_stops_df.to_csv(file, index=False)
file.close()
```

1.5 Drop incomplete data from CSV

Now we will clean the `vehiclePositions.csv` file created in previous section by comparing each datapoint to see if the stopId exists in the recently created `merged_stops.csv`. When we checked that file, there was only one undefined stop based on GPS position, so we are treating the GTFS file provided by STIB as being the most accurate source of truth.

When we check the stopIds that exist in `vehiclePositions.csv` against the `merged_stops.csv`, we will see that we lost 11.7% (~2.3M timestamps out of 19.4M) of our data. This first data drop is assumed to be due to the "technical stops" described in the STIB documentation that are not true passenger stops and can be ignored. The number of timestamps dropped is different for each line, and the percentage drop is shown.

Depending on the business case, it would be worthwhile evaluating data line by if a granular data analysis is required. For the purposes of this project, we are looking at the STIB network performance as a whole and did not dive further into the data drops.

Reads from: CSV file containing all the vehicle positions in `data` folder (`data/processed/assignment1/vehiclePositions.csv`) **Writes to**: CSV file containing filtered vehicle positions in `data` folder (`data/processed/assignment1/vehiclePositionsClean.csv`)

[20]:
```python
stops_csv_path = '../data/processed/assignment1/line_stops.csv'
positions_csv_path = '../data/processed/assignment1/vehiclePositions.csv'
cleaned_positions_csv_path = '../data/processed/assignment1/
 ↪vehiclePositionsClean.csv'
```

[21]:
```python
stops_df = pd.read_csv(stops_csv_path)
stop_ids = stops_df['stop_id_int']
stops_df.head()
```

[21]:
```
  lineId  direction  order stop_id        name_ascii              name  \
0   001m          1      1    8733   GARE DE L'OUEST   Gare de l'Ouest
1   001m          1      2    8742          BEEKKANT          Beekkant
2   001m          1      3    8292       ETANGS NOIRS       Étangs Noirs
3   001m          1      4    8282  COMTE DE FLANDRE  Comte de Flandre
```

```
4   001m          1      5    8272  SAINTE-CATHERINE  Sainte-Catherine

   lambert_x  lambert_y  stop_id_int   stop_lat  stop_lon
0   146633.5   170956.4         8733  50.848999  4.320948
1   146776.5   171444.3         8742  50.853386  4.322974
2   147492.7   171859.9         8292  50.857125  4.333143
3   148013.6   171590.4         8282  50.854705  4.340542
4   148539.5   171278.2         8272  50.851900  4.348012
```

[37]:
```python
positions_df = pd.read_csv(positions_csv_path)
positions_df.count()
```

[37]:
```
Timestamp            19421883
LineId               19421883
DirectionId          19421883
DistanceFromPoint    19421883
PointId              19421883
dtype: int64
```

[23]:
```python
cleaned_positions_df = positions_df[positions_df['DirectionId'].isin(stop_ids)␣
↪& positions_df['PointId'].isin(stop_ids)]
cleaned_positions_df.count()
```

[23]:
```
Timestamp            17153015
LineId               17153015
DirectionId          17153015
DistanceFromPoint    17153015
PointId              17153015
dtype: int64
```

[26]:
```python
original_counts = positions_df['LineId'].value_counts()
new_counts = cleaned_positions_df['LineId'].value_counts()
counts_df = original_counts.to_frame('original_counts').join(new_counts.
↪to_frame('new_counts'))
counts_df.head()
```

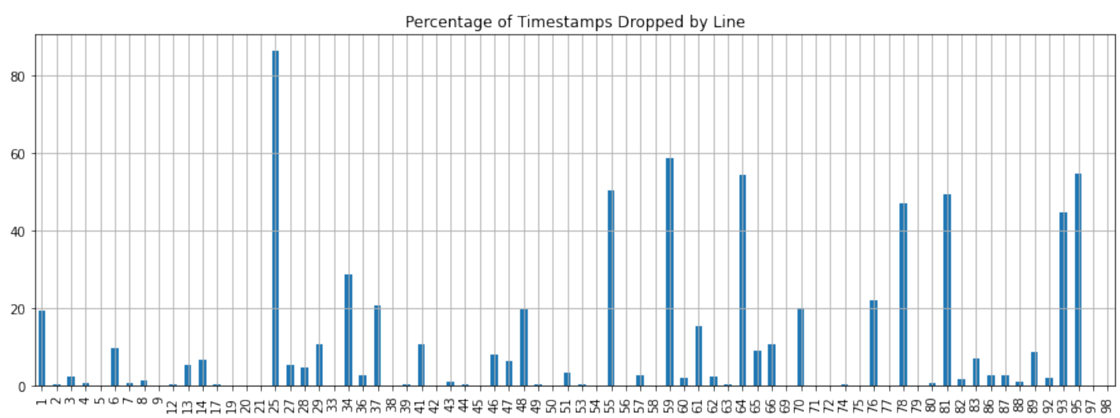
[26]:
```
    original_counts  new_counts
95           531447      240516
7            529144      526086
82           515924      506619
81           504996      256610
51           500110      483565
```

[31]:
```python
counts_df['positions_dropped'] = counts_df['original_counts'] -␣
↪counts_df['new_counts']
counts_df['percentage_dropped'] = counts_df['positions_dropped'] * 100 /␣
↪counts_df['original_counts']
```

```
counts_df.sort_index(inplace=True)
print(counts_df)
counts_df['percentage_dropped'].plot(kind='bar', figsize=(15,5), grid=True,␣
↪title="Percentage of Timestamps Dropped by Line")
```

```
    original_counts  new_counts  positions_dropped  percentage_dropped
1            305454      246596              58858           19.269023
2            215114      214341                773            0.359344
3            383543      373963               9580            2.497764
4            282947      281293               1654            0.584562
5            405897      405853                 44            0.010840
..              ...         ...                ...                 ...
92           486408      475895              10513            2.161354
93           402974      223230             179744           44.604367
95           531447      240516             290931           54.743182
97           276705      276501                204            0.073725
98            98755       98755                  0            0.000000

[74 rows x 4 columns]
```

[31]: <AxesSubplot:title={'center':'Percentage of Timestamps Dropped by Line'}>



[22]:
```
cleaned_positions_df.to_csv(cleaned_positions_csv_path, index=False)
```

1.6 Add direction to CSV

Here we add the direction that a vehicle is moving to `vehiclePositionsClean.csv` that was just created based on some information in the stops. For the majority of vehicle trips, the stop id on one side of the road is different from that on the other side of the road.

> **Example:** There will be 2 different IDs for stop "Thys" on bus route 95. One will be associated with a direction going towards city center, and one is associated with a direction going out of city center.

It could also be the case that the stop is the last stop of a direction, in this case, the direction is set depending on which line the stop is in.

For the 2 complex cases, a comparison is completed between the stop_id and destination_id and the direction is set depending on which of the two comes first in the route sequence.

Reads from: CSV file containing filtered vehicle positions in `data` folder (`data/processed/assignment1/vehiclePositionsClean.csv`) **Writes to**: CSV file containing filtered vehicle positions with direction in `data` folder (`data/processed/assignment1/vehiclePositionsCleanDirected.csv`)

[86]:
```python
def get_direction_from_line_stop_and_destination(line: Tuple[List[List[str]],␣
↪List[List[str]]], stop_id: str,
                                                  destination_id: str) -> int:
    # Simple case 1 -> destination_id is in one direction but not in the other:
    stops = ({stop[3] for stop in line[0]}, {stop[3] for stop in line[1]})
    if destination_id in stops[0] and destination_id not in stops[1]:
        return 0
    if destination_id not in stops[0] and destination_id in stops[1]:
        return 1

    # Simple case 2 -> destination_id is the last stop of a direction:
    if destination_id == line[0][-1][3]:
        return 0
    if destination_id == line[1][-1][3]:
        return 1

    # Simple case 3 -> stop_id is in one direction but not in the other:
    if stop_id in stops[0] and stop_id not in stops[1]:
        return 0
    if stop_id not in stops[0] and stop_id in stops[1]:
        return 1

    # Complex case 1 -> if stop_id != destination_id, return the direction in␣
↪which the stop with
    # id destination_id is after the stop with id stop_id
    if stop_id != destination_id:
        index_of_destination_0 = next(int(stop[8]) for stop in line[0] if␣
↪stop[3] == destination_id)
        index_of_stop_0 = next(int(stop[8]) for stop in line[0] if stop[3] ==␣
↪stop_id)
        return 0 if index_of_stop_0 < index_of_destination_0 else 1
```

```
    # Complex case 2 -> if stop_id == destination_id, return the direction in␣
↪which the stop with
    # id destination_id is further down the direction
    else:
        index_of_destination_0 = next(int(stop[8]) for stop in line[0] if␣
↪stop[3] == destination_id)
        index_of_destination_1 = next(int(stop[8]) for stop in line[1] if␣
↪stop[3] == destination_id)
        return 0 if index_of_destination_0 > index_of_destination_1 else 1
```

[87]:
```
def add_direction_to_csv():
    positions = read_csv_stream('../data/processed/assignment1/
↪vehiclePositionsClean.csv', skip_first=False)
    directed_positions, output_file = get_csv_writer('../data/processed/
↪assignment1/vehiclePositionsCleanDirected.csv')
    directed_positions.writerow([*next(positions), 'Direction'])
    grouped_lines = group_line_stops(read_csv_stream('../data/processed/
↪assignment1/line_stops.csv'))
    memory = {}
    for position in tqdm(positions):
        line_id = position[1]
        stop_id = position[4]
        destination_id = position[2]
        tuple_id = f'{line_id}-{stop_id}-{destination_id}'
        if tuple_id not in memory:
            memory[tuple_id] =␣
↪get_direction_from_line_stop_and_destination(grouped_lines[line_id], stop_id,
                                                                              ␣
↪destination_id)
        directed_positions.writerow([*position, memory[tuple_id]])
    output_file.close()
```

[88]:
```
add_direction_to_csv()
```

```
0it [00:00, ?it/s]
```

1.7 Split CSV into lines

For future manipulation, especially during the SARIMA time-series prediction modelling for arrival time at a stop, we will split the data in `vehiclePositionsCleanDirected.csv` into individual csv files based on each line. Given the size of the data set, this will also help in doing some exploratory testing on smaller subsets of data to ensure the approach being implemented makes sense.

Reads from: CSV file containing filtered vehicle positions with direction in `data` folder (`data/processed/assignment1/vehiclePositionsCleanDirected.csv`)

Writes to: CSV file per line containing filtered vehicle positions with direction in `data/processed/assignment1/vehiclePositionsPerLine` folder (`data/processed/assignment1/vehiclePositionsPerLine/vehiclePositions*.csv`)

[89]:
```
def split_csv_by_lines():
    files = {}
    positions = read_csv_stream('../data/processed/assignment1/
↪vehiclePositionsCleanDirected.csv', skip_first=False)
    output_dir = '../data/processed/assignment1/vehiclePositionsPerLine'
    if not os.path.exists(output_dir):
        os.mkdir(output_dir)
    header = next(positions)
    for line in tqdm(positions):
        line_id = line[1]
        if line_id not in files:
            files[line_id] = get_csv_writer(
                f'{output_dir}/vehiclePositions{line_id}.csv')
            files[line_id][0].writerow(header)
        files[line_id][0].writerow(line)
    for _, file in files.values():
        file.close()
```

[90]:
```
split_csv_by_lines()
```

```
0it [00:00, ?it/s]
```

1.8 Vehicle Matching

Now we have all the data needed to attempt to link several vehicle positions belonging to same physical vehicle. The general idea is that we will look at every vehicle position at a single timestamp, and compare that to every vehicle at the next timestamp. There are different changes that might have taken place with the stop and distance to stop columns, which will be used to match buses up. So we will: 1. Group positions by timestamp and sort groups 2. Iterate over groups finding matches 3. Match buses depending on stop and distance to stop

This approach means that anytime a vehicle approaches a technical stop (that was previously dropped when we matched our vehiclePosition dataset with the GTFS stopIDs data) we will "lose" that unique bus, and "gain" a new unique bus on the other side of the technical stop. This also means that a bus is "lost" whenever it finishes a sequence at a line terminus and a new bus is "gained" whenever it starts a route at the first stop in a sequence.

For the purpose of calculating distance between stops, this approach works well, as we do not need to know exactly which bus is where, just how long it takes to travel between stops.

1.8.1 The project looks at times and delays from the perspective of a public transport client, who does not care which unique bus arrives at which time, just that ANY bus arrives at the scheduled time

Reads from: CSV file per line containing filtered vehicle positions with direction in `data/processed/assignment1/vehiclePositionsPerLine` folder (`data/processed/assignment1/vehiclePositionsPerLine/vehiclePositions*.csv`)
Writes to: CSV file per line containing vehicle positions with `bus_id` in `data/processed/assignment1/csv_lines_linked` folder (`data/processed/assignment1/csv_lines_linked/vehiclePositions*.csv`)

[91]:
```
def split_positions_by_direction(positions: Iterable[List[str]]) ->␣
↪Tuple[List[List[str]], List[List[str]]]:
    line = ([], [])
    for position in positions:
        line[int(position[-1])].append(position)
    return line
```

[103]:
```
def get_index_of_stop_in_line(line, direction, stop_id):
    return get_index_of_stop_in_line_direction(line[direction], stop_id)


def get_index_of_stop_in_line_direction(line, stop_id):
    return next((int(stop[-1]) for stop in line if stop[3] == stop_id), -1)


class Match(Enum):
    WRONG = 1
    OK = 2
    TOO_FAR = 3
```

[100]:
```
def group_positions_by_timestamp(positions: Iterable[List[str]]) ->␣
↪List[Tuple[int, List[List[str]]]]:
    grouped_positions = []
    old_timestamp = -1
    current_timestamp_positions = []
    for position in positions:
        current_timestamp = int(position[0])
        if current_timestamp != old_timestamp:
            assert current_timestamp > old_timestamp
            grouped_positions.append((old_timestamp,␣
↪current_timestamp_positions))
            old_timestamp = current_timestamp
            current_timestamp_positions = []
        current_timestamp_positions.append(position)
    grouped_positions.append((old_timestamp, current_timestamp_positions))
    return grouped_positions[1:]
```

[101]:
```
def possible_match(first_position: List[str], second_position: List[str],
                   line: List[List[str]]) -> Match:
    first_stop_id = first_position[4]
    second_stop_id = second_position[4]
    # Both positions are in the same stop -> Compare using distance from that␣
↪stop
    if first_stop_id == second_stop_id:
        first_distance = int(first_position[3])
        second_distance = int(second_position[3])
        return Match.OK if first_distance <= second_distance else Match.WRONG
    # Positions are in different stops -> Compare using order of stops in␣
↪direction
    else:
        # Assert both positions have the same direction
        first_stop_index = get_index_of_stop_in_line_direction(line,␣
↪first_stop_id)
        second_stop_index = get_index_of_stop_in_line_direction(line,␣
↪second_stop_id)
        if second_stop_index - first_stop_index > 3:
            return Match.TOO_FAR
        return Match.OK if first_stop_index < second_stop_index else Match.WRONG
```

[102]:
```
def find_bus_matched_of_line_direction(positions: Iterable[List[str]], line:␣
↪List[List[str]], line_id: str,
                                       direction: int, writer):
    grouped_positions = group_positions_by_timestamp(positions)
    previous_positions = []
    bus_id = (f'{line_id}-{direction}-{i:06d}' for i in count())
    sorting_key = lambda vehicle_position:␣
↪get_index_of_stop_in_line_direction(line, vehicle_position[-2])

    for timestamp, current_positions in grouped_positions:
        sorted_positions = sorted(current_positions, key=sorting_key)
        while len(sorted_positions) > 0 and␣
↪get_index_of_stop_in_line_direction(line, sorted_positions[0][-2]) == -1:
            sorted_positions.pop(0)
        current_previous_position_index = 0
        current_position_index = 0
        while current_position_index < len(sorted_positions) and␣
↪current_previous_position_index < len(
                previous_positions):
            previous_position =␣
↪previous_positions[current_previous_position_index]
            current_position = sorted_positions[current_position_index]
            result = possible_match(previous_position, current_position, line)
            if result == Match.OK:
```

```
                current_position.append(previous_position[-1])
                current_previous_position_index += 1
                current_position_index += 1
            elif result == Match.WRONG:
                current_position.append(next(bus_id))
                current_position_index += 1
            elif result == Match.TOO_FAR:
                current_previous_position_index += 1
        for position in sorted_positions:
            if len(position) == 6:
                position.append(next(bus_id))
            writer.writerow(position)
        previous_positions = sorted_positions
```

[96]:
```
def find_bus_matches_of_line(file_path: str, output_path: str, line_id: str,
                             line: Tuple[List[List[str]], List[List[str]]]) ->␣
↪None:
    positions = read_csv_stream(file_path, skip_first=False)
    with write_csv(output_path) as linked_positions:
        linked_positions.writerow([*next(positions), 'BusId'])
        direction1, direction2 = split_positions_by_direction(positions)
        find_bus_matched_of_line_direction(direction1, line[0], line_id, 0,␣
↪linked_positions)
        find_bus_matched_of_line_direction(direction2, line[1], line_id, 1,␣
↪linked_positions)
```

[97]:
```
def find_bus_matches_of_lines():
    path = '../data/processed/assignment1/vehiclePositionsPerLine'
    output_path = '../data/processed/assignment1/csv_lines_linked'
    if not os.path.exists(output_path):
        os.mkdir(output_path)
    lines = group_line_stops(read_csv_stream('../data/processed/assignment1/
↪line_stops.csv'))
    for file in tqdm(os.listdir(path)):
        line_id = file[16:-4]
        line = lines[line_id]
        find_bus_matches_of_line(f'{path}/{file}', f'{output_path}/{file}',␣
↪line_id, line)
```

[104]:
```
find_bus_matches_of_lines()
```

```
  0%|          | 0/74 [00:00<?, ?it/s]
```

2 Calculate average time between stops

2.1 Pulling it all together

We now have all the building blocks we need to calculate the time between each stop. Here we will calculate the time between each stop for each line and then create and average of the times at 1hr intervals. Given the focus of the project for an analysis of the STIB network speed, this provides a "low-resolution" version of the data that can be used to draw some powerful initial conclusions with a more compact data set.

After initial analysis and conclusions are drawn, it is possible to come back and generate more granular data at smaller time intervals, look at specific lines with a very high data resolution, or even use a clustering method to develop a dynamic time averaging depending on when the most change in travel time between stops is detected.

For the current revision of the project, no additional granularity of data was used beyond the 1-hour intervals.

[105]:
```
def group_positions_by_vehicle(positions: List[List[str]]) -> Dict[str,␣
↪List[List[List[str]]]]:
    grouped_positions = {}
    for position in positions:
        bus_id = position[-1]
        if bus_id not in grouped_positions:
            grouped_positions[bus_id] = []
        grouped_positions[bus_id].append(position)
    return grouped_positions
```

[106]:
```
def get_hour_from_timestamp(timestamp: int) -> int:
    return datetime.datetime.fromtimestamp(timestamp // 1000).hour
```

[107]:
```
def get_vehicle_times_between_stops(vehicle_positions: List[List[str]],
                                    line: Tuple[List[List[str]],␣
↪List[List[str]]]) -> List:
    times = []
    previous_timestamp = int(vehicle_positions[0][0])
    previous_stop = vehicle_positions[0][4]
    for position in vehicle_positions:
        current_timestamp = int(position[0])
        current_stop = position[4]
        if current_stop != previous_stop:
            time_difference = (current_timestamp - previous_timestamp) // 1000
            first_hour = get_hour_from_timestamp(previous_timestamp)
            last_hour = get_hour_from_timestamp(current_timestamp)
            times.append([first_hour, previous_stop, current_stop,␣
↪time_difference])
            if first_hour != last_hour:
```

```
                times.append([last_hour, previous_stop, current_stop,␣
↪time_difference])
            previous_stop = current_stop
            previous_timestamp = current_timestamp
    return times
```

[108]:
```
def calculate_average_time_between_stops_of_line(positions: List[List[str]],␣
↪line_id: str,
                                                 line: Tuple[List[List[str]],␣
↪List[List[str]]], output):
    grouped_positions = group_positions_by_vehicle(positions)
    times = []
    for vehicle_id, vehicle_positions in grouped_positions.items():
        times += get_vehicle_times_between_stops(vehicle_positions, line)
    for direction in [0, 1]:
        for fromStop, toStop in zip(line[direction][:-1], line[direction][1:]):
            time_sum = [0 for _ in range(24)]
            time_sum_filtered = [0 for _ in range(24)]
            time_count = [0 for _ in range(24)]
            time_count_filtered = [0 for _ in range(24)]
            for time in times:
                if time[1] == fromStop[3] and time[2] == toStop[3]:
                    time_sum[time[0]] += time[3]
                    time_count[time[0]] += 1
                    if time[3] < 6000:  # Times larger than 10 minutes are␣
↪likely anomalies that shouldn't be counted
                        time_sum_filtered[time[0]] += time[3]
                        time_count_filtered[time[0]] += 1
            output[0].writerow(
                [line_id, fromStop[3], toStop[3],
                 *[f'{total / amount:.2f}' if amount > 0 else 0 for total,␣
↪amount in zip(time_sum, time_count)]])
            output[1].writerow([line_id, fromStop[3], toStop[3], *time_count])
            output[2].writerow(
                [line_id, fromStop[3], toStop[3], *[f'{total / amount:.2f}' if␣
↪amount > 0 else 0 for total, amount in
                                                    zip(time_sum_filtered,␣
↪time_count_filtered)]])
            output[3].writerow([line_id, fromStop[3], toStop[3],␣
↪*time_count_filtered])
```

[109]:
```
def calculate_average_time_between_stops():
    source_path = '../data/processed/assignment1/csv_lines_linked'
    output_path = '../data/processed/assignment1/average_time_between_stops.csv'
    output_path_filtered = '../data/processed/assignment1/
↪average_time_between_stops_filtered.csv'
```

```
    output_path_count = '../data/processed/assignment1/
↪average_time_between_stops_count.csv'
    output_path_filtered_count = '../data/processed/assignment1/
↪average_time_between_stops_filtered_count.csv'
    lines = group_line_stops(read_csv_stream('../data/processed/assignment1/
↪line_stops.csv'))
    with write_csv(output_path) as output, write_csv(output_path_filtered) as␣
↪output_filtered, write_csv(
            output_path_count) as output_count,␣
↪write_csv(output_path_filtered_count) as output_filtered_count:
        header = ['LineId', 'FromStop', 'ToStop', *[f'{i}' for i in range(24)],␣
↪'Day']
        output.writerow(header)
        output_filtered.writerow(header)
        output_count.writerow(header)
        output_filtered_count.writerow(header)
        for file in tqdm(os.listdir(source_path)):
            line_id = file[16:-4]
            line = lines[line_id]
            positions = read_csv_list(f'{source_path}/{file}')[1:]
            calculate_average_time_between_stops_of_line(positions, line_id,␣
↪line,
                                                         (output, output_count,␣
↪output_filtered, output_filtered_count))
```

[110]:
```
calculate_average_time_between_stops()
```

```
  0%|          | 0/74 [00:00<?, ?it/s]
```

[111]:
```
def calculate_daily_average_time_between_stops():
    source_path = '../data/processed/assignment1/csv_lines_linked'
    output_path = '../data/processed/assignment1/
↪daily_average_time_between_stops_filtered'
    lines = group_line_stops(read_csv_stream('../data/processed/assignment1/
↪line_stops.csv'))
    header = ['LineId', 'FromStop', 'ToStop', *[f'{i}' for i in range(24)]]

    class Dummy:
        def writerow(self, *args):
            pass

    dummy = Dummy()
    day_files = {}
    for file in tqdm(os.listdir(source_path)):
        line_id = file[16:-4]
```

```
        line = lines[line_id]
        positions = read_csv_list(f'{source_path}/{file}')[1:]
        day_positions = {}
        for position in positions:
            day = datetime.datetime.fromtimestamp(int(position[0]) / 1000).
↪date().strftime('%Y-%m-%d')
            if day not in day_positions:
                day_positions[day] = []
            day_positions[day].append(position)
        for day in day_positions:
            if day not in day_files:
                day_files[day] = get_csv_writer(f'{output_path}/{day}.csv')
                day_files[day][0].writerow(header)
            calculate_average_time_between_stops_of_line(day_positions[day],␣
↪line_id, line,
                                                         (dummy, dummy,␣
↪day_files[day][0], dummy))
    for writer, file in day_files.values():
        file.close()
```

[113]:
```
calculate_daily_average_time_between_stops()
```

```
  0%|          | 0/74 [00:00<?, ?it/s]
```

2.2 Calculating Speed Between Stops

Now that we have the time between stops, the next step to calculate the speed between stops will be to find the distance. To do this, we will iterate through the ACTU_LINES shapefile to find the nearest geometry points to each stop. We will then iterate through the shapefile to calculate the full distance from stop to stop. Once we have distance, we can merge distance and time to find Velocity = distance/time.

We will also prep some data for visualization by creating a GeoJSON array that can be passed into Kepler.gl to plot each individual line segment (stop-to-stop polyline).

Input Files: - line_stops.csv - ACTU_LINES.shp - shapes_lat_long.csv - average_time_between_stops_filtered.csv

Export Files: - stop_distance.csv - vehicle_speed.csv

[8]:
```
line_stops = pd.read_csv('../data/processed/assignment1/line_stops.csv')
sf_actu_lines = shapefile.Reader('../data/raw/shapefiles/ACTU_LINES.shp')
# here we initialize shape_records, which includes a combination of the shapes␣
↪and records from the shapefile. This combination will allow us to pull the␣
↪lambert coordinates from the shapes as while also accessing the record␣
↪information like line_id.
```

```
shape_records = sf_actu_lines.shapeRecords()
```

3 Shapefile Distance Calculation Function

Now that we have our libraries loaded and files imported, we will create a function that can calculate the distance between two points on a polyline. The start_point and end_point are indexes to tell us where we should start and stop calculating distance in the polyline. the line_segment is one of the shape elements that will be pulled from the shapefile. This calculation will be called later in an iterative for loop for each shape element withing shape_records.shape.

We will calculate the distance between each point in the shapefile using Pythagoreas' theorem, since the units in both line_stops and the shapefile are already provided in Belgium Lambert 1972 format, which projects the points onto a flat surface.

[9]:
```
def calculate_distance_between_polyline_points(start_point: int, stop_point:␣
↪int,
                                                line_segment: shapefile.Shape)␣
↪-> float:
    # initializing our total distance to 0
    total_distance = 0
    # we'll need to calculate the distance between each consecutive pair of␣
↪coordinates, and will iterate
    # from the start_point to the end_point. Each newly caluclated distance␣
↪between points will be added
    # to the sum total_distance and then returned.
    for index in range(start_point, stop_point - 1):
        current = index
        next = index + 1
        total_distance += sqrt(pow((line_segment.points[current][0] -␣
↪line_segment.points[next][0]), 2) + pow(
            (line_segment.points[current][1] - line_segment.points[next][1]),␣
↪2))
    return total_distance
```

4 Calculate Distance Between Stops

Time to get to work! Here we several nested for loops that are used to compare match up the line in the shapefile to the line in the line_stops. For each matching line, we will cycle through to project the stop location ONTO the polyline. This is required because a bus stop can be imagined to be on a sidewalk, while the polyline is moving along the road.

Once we have matched up our bus stop with the nearest polyline point, we move to the next stop and do the same. Having 2 stop locations projected, we can call the previously defined calculate_distance_between_polyline_points function to find the distance between these 2 stops.

The first iteration of the loop will result in a dummy value, as it does not have a real stop to pair with. All of these dummy values are dropped once the dictionary that stores all values is transformed into a dataframe.

The data frame will be accessed latter with the unique combination of [LineID + fromStopID + toStopID]. This combination will be different depending on which direction a vehicle is moving, as the stop id's are not the same on each side of a street.

We will also hold onto the index value for the polyline location in case we need it later on for future predictions.

integration: This is how things were integerated parameters: Here we dropped all stops before 4am because...

[10]:
```
# initialize a dictionary that will be used to make a dataframe and csv file␣
↪with the following format:
# |    LineId    |   FromStop  |   ToStop  |   distance    |     fromIndex    |  ␣
↪ toIndex   |
stop_distance = {'LineId': [], 'LineIdFormatted': [], 'LineId_GeoMerge': [],␣
↪'Type': [], 'Direction': [],
                 'FromStop': [], 'ToStop': [], 'distance': [], 'fromIndex': [],␣
↪'toIndex': [],
                 'FromStop_lat': [], 'FromStop_lon': [], 'ToStop_lat': [],␣
↪'ToStop_lon': []}

# Initializing variables that will be used in loops
last_pointID = 0
last_stop_id = 0
last_stop_lat = 0
last_stop_lon = 0
adjusted_stop_lat_GPS = 0
adjusted_stop_lon_GPS = 0

# look through each shape/record combo in the shape_records file. Each element␣
↪of shape_records represents a single line (metro, bus or tram)
for shape_record in shape_records:
    record = shape_record.record
    shape = shape_record.shape
    # look through each of the stops that exist in the line_stops csv. Here we␣
↪are going to only cycle through a subset of the line_stops where there is a␣
↪match on LineId and the direction to reduce computation time.
    for index, stop in line_stops[
        (line_stops['lineId'] == record['LIGNE']) & (line_stops['direction'] ==␣
↪record['VARIANTE'])].sort_values(
        by=['order']).iterrows():
        # Initializing variables that will be used in loops
        min_distance = 50
        adjusted_stop_lat = 50
```

```
        adjusted_stop_lon = 50
        current_pointID = 0
        current_stop_id = stop['stop_id_int']
        stop_lat = stop['lambert_x']
        stop_lon = stop['lambert_y']
        #After choosing a single stop from the line_stops file, we will compare␣
↪that stops lambert GPS position to each coordinate that makes up the␣
↪polyline in the current shape_records shape. We are finding the closest␣
↪location in the shape file to our bus stop location. This can be done using␣
↪euclidean distance calculation because the coordinates are in lambert␣
↪notation. Whichever location on the polyline is the closest becomes the␣
↪projected location of the bus stop using the if statement.
        for pointID in range(len(shape.points)):
            point_lat = shape.points[pointID][0]
            point_lon = shape.points[pointID][1]
            distance = sqrt(pow((point_lat - stop_lat), 2) + pow((point_lon -␣
↪stop_lon), 2))
            # if statement to compare distances and updated if shorter. It also␣
↪saves the polyline info for future use in predicting
            # which method of transport is being used.
            if distance < min_distance:
                min_distance = distance
                adjusted_stop_lat = point_lat
                adjusted_stop_lon = point_lon
                current_pointID = pointID
                adjusted_stop_lat_GPS = stop['stop_lat']
                adjusted_stop_lon_GPS = stop['stop_lon']
        # now we call a previously defined function to calculate the total␣
↪distance between the location projected during the previous for loop␣
↪iteration and the current loop iteration. We are able to do this because the␣
↪stops have been sorted by descending order from first to last. The first row␣
↪in the array will always be a dummy row and needs to be dropped afterwards.
        distance_between_stops =␣
↪calculate_distance_between_polyline_points(last_pointID, current_pointID,␣
↪shape)
        # we update our dictionary with all the values needed for distance␣
↪between stops.
        # we will also strip out the leading zeros and the trailing text␣
↪characters indicating (b,t,m for bus, tram and metro)
        stripped_line_id = stop['lineId'][:-1].strip("0")
        stop_distance['LineId'].append(stripped_line_id)
        stop_distance['LineIdFormatted'].append(f"line{stripped_line_id}")
        stop_distance['LineId_GeoMerge'].
↪append(f"{stop['lineId']}-{stop['direction']}")
        stop_distance['Type'].append(str(stop['lineId'][-1]))
        stop_distance['Direction'].append(stop['direction'])
```

```
        stop_distance['FromStop'].append(last_stop_id)
        stop_distance['ToStop'].append(current_stop_id)
        stop_distance['distance'].append(distance_between_stops)
        stop_distance['fromIndex'].append(last_pointID)
        stop_distance['toIndex'].append(current_pointID)
        stop_distance['FromStop_lat'].append(last_stop_lat)
        stop_distance['FromStop_lon'].append(last_stop_lon)
        stop_distance['ToStop_lat'].append(adjusted_stop_lat_GPS)
        stop_distance['ToStop_lon'].append(adjusted_stop_lon_GPS)
        # after calculating the distance, we update the last stop id, point,␣
↪and lat/lon to the currently being used before iterating through to the next␣
↪bus stop. The current point becomes the last point for the next calculation.
        last_stop_id = current_stop_id
        last_pointID = current_pointID
        last_stop_lat = adjusted_stop_lat_GPS
        last_stop_lon = adjusted_stop_lon_GPS
```

[11]:

```
# Opening JSON file
json_shapes_path = '../data/raw/shapes_lat_long.json'
with open(json_shapes_path, 'r') as file:
    line_shapes_geojson = json.load(file)

df_line_shapes_geojson = pd.json_normalize(line_shapes_geojson)
df_stop_distance = pd.DataFrame.from_dict(stop_distance)

sorted_geo_json = []
for line_id, polyline in df_line_shapes_geojson.iteritems():
    for stop_index, value in df_stop_distance.iterrows():
        if line_id == value['LineId_GeoMerge']:
            sorted_geo_json.append(polyline.values.__array__()) ␣
↪#df_line_shapes_geojson[line_id]
```

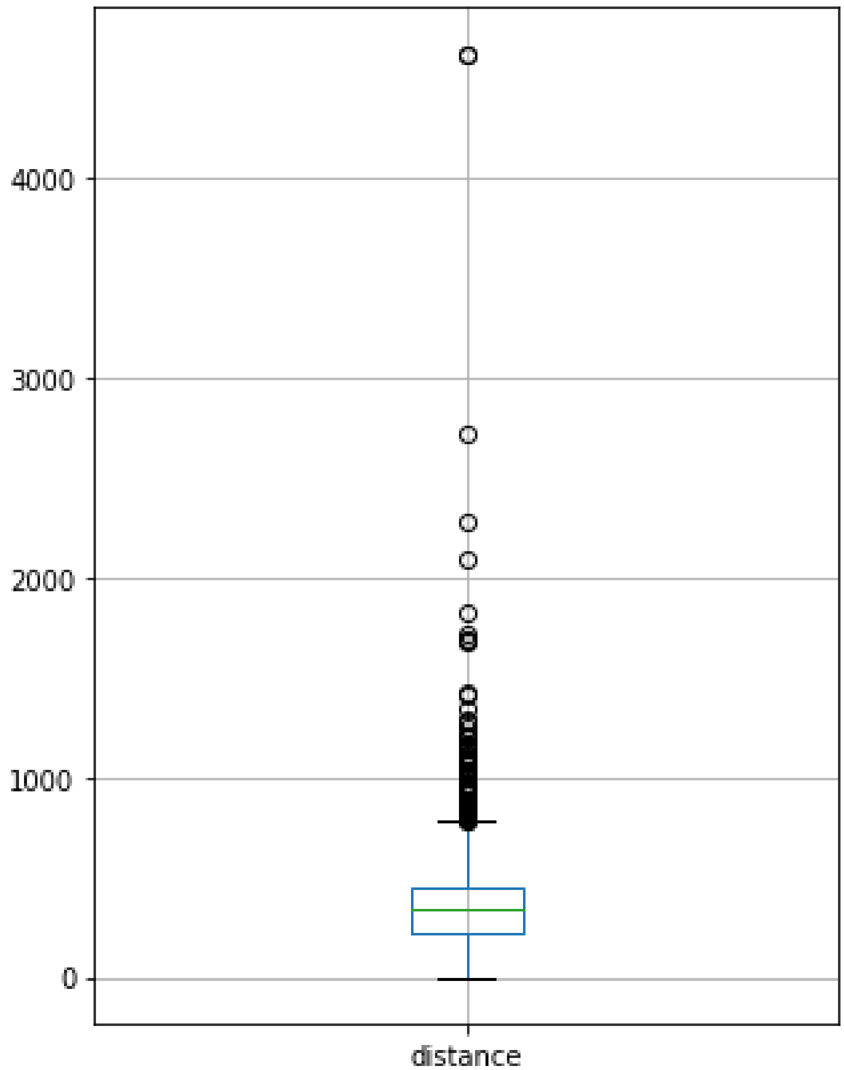
[12]:

```
df_line_shapes_geojson_transformed = df_line_shapes_geojson.transpose()[0].
↪apply(lambda x: {"type": "FeatureCollection",
  "features": [
    {"type": "Feature",
     "properties": {},
     "geometry": {
        "type": "LineString",
        "coordinates": x}
    }
                                                                              ␣
↪ )

df_stop_distance_merged = df_stop_distance.
↪merge(df_line_shapes_geojson_transformed, left_on='LineId_GeoMerge',
```

```
                                                        right_index=True).
 ↪rename(columns={0: 'geojson'})
# now we convert the dictionary to a Pandas DataFrame for easier manipulation␣
 ↪and drop to the first dummy row
df_stop_distance_merged.
 ↪drop(df_stop_distance_merged[df_stop_distance_merged['toIndex'] == 0].index,␣
 ↪inplace=True)
```

[14]:
```
print(df_stop_distance_merged[['LineId', 'Type', 'Direction', 'distance',␣
 ↪'FromStop', 'ToStop']].nlargest(5, 'distance'))
print(df_stop_distance_merged[['distance']].describe())
print(df_stop_distance_merged[['distance']].boxplot(figsize=(5,7)))
```

```
     LineId Type  Direction     distance  FromStop  ToStop
493      12    b          2  4623.389353      3018    9600
477      12    b          1  4614.768150      9600    3017
3992     98    b          1  2717.641454      2384    3810
4024     98    b          2  2277.669942      3851    2382
192       5    m          1  2092.387086      8642    8652
          distance
count  4511.000000
mean    337.976954
std     226.978543
min       0.000000
25%     224.625593
50%     335.533957
75%     447.653545
max    4623.389353
AxesSubplot(0.125,0.125;0.775x0.755)
```



4.1 Data Analysis

After viewing a box plot of distance, there are some obvious outliers that may need to be dropped or cleaned later on. Below, we explore the data's distribution after filtering out 1.5 * IQR (interquartile range) which is a typical measure of outlier. We will also look to see how many values this would remove.

After reviewing the data along with the actual stops, it was seen that the outliers are real data, as the longest distance between stops is on bus line 12 travelling to the airport, with a distance between stops in the 5 km range. This aligns well with the data, indicating there are no issues with our upper range date.

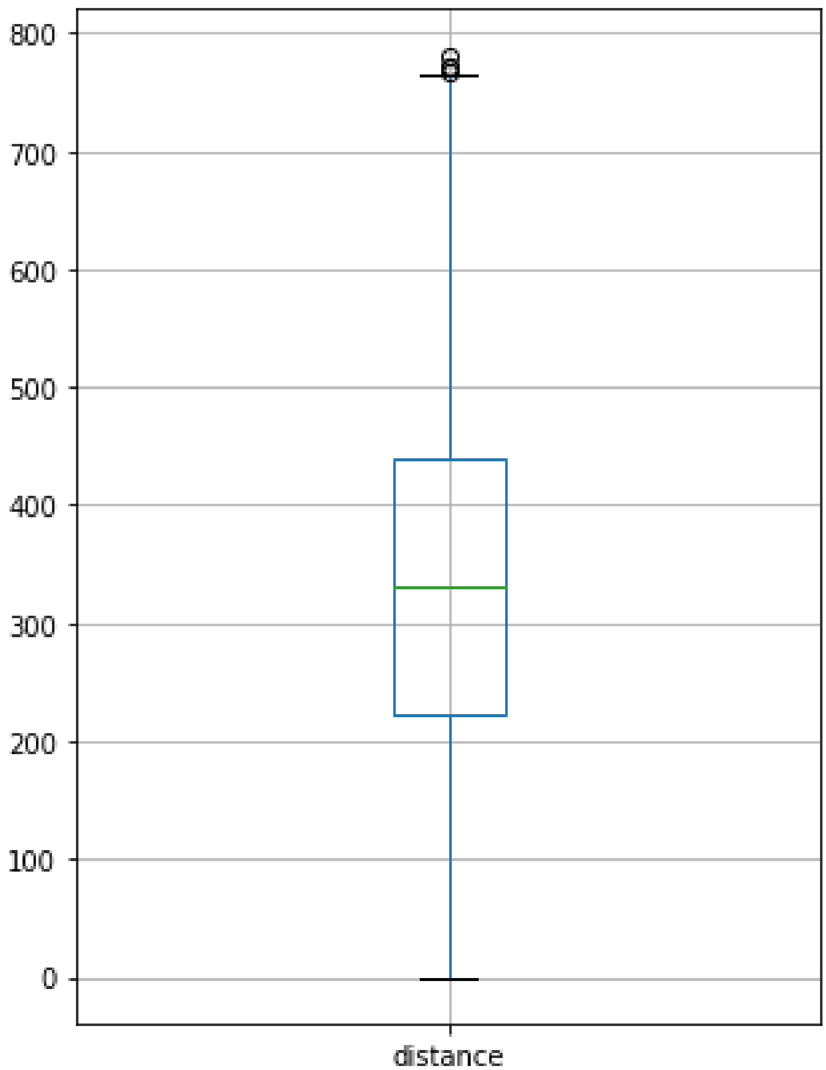
We can also see that there are 517 stop segments that have a distance of 0 km. This could be due to the distance between terminus stops at either end of a line, or potentially some other factor. We will see later on in the speed calculation how these distances resulted in a speed calculation of "0 km/h". For the first pass through the data to draw STIB network level conclusions, these values were not investigated and will be dropped latter on in the speed calculation. This would be a place to return to the data to do additional data exploration.

[16]:
```
# Remove outliers from a dataframe by column, including optional whiskers,␣
↪removing rows for which the column value are less than Q1-1.5IQR or greater␣
↪than Q3+1.5IQR.
def subset_by_iqr(df: pd.DataFrame, column: str, whisker_width=1.5) -> pd.
↪DataFrame:
    q1 = df[column].quantile(0.25)
    q3 = df[column].quantile(0.75)
    iqr = q3 - q1
    # Apply filter with respect to IQR, including optional whiskers
    filter = (df[column] >= q1 - whisker_width*iqr) & (df[column] <= q3 +␣
↪whisker_width*iqr)
    return df.loc[filter]

df_filtered = subset_by_iqr(df_stop_distance_merged, 'distance',␣
↪whisker_width=1.5)
print(df_filtered[['distance']].boxplot(figsize=(5,7)))
print("Count of unfiltered distances is: \n",␣
↪df_stop_distance_merged[['distance']].count().sum(), "\n")
print("Count of filtered distances is: \n", df_filtered[['distance']].count().
↪sum(), "\n")
print(f"Count of '0' distances is: \n", df_filtered[['distance']].isin([0]).
↪sum(), "\n")

df_filtered = df_filtered.loc[df_filtered['distance'] > 0]
df_filtered[['distance']].describe()
```

```
AxesSubplot(0.125,0.125;0.775x0.755)
Count of unfiltered distances is:
 4511

Count of filtered distances is:
 4409

Count of '0' distances is:
 distance    517
dtype: int64
```

[16]:

	distance
count	3892.000000
mean	362.893774
std	139.212075
min	5.414795
25%	261.809106
50%	354.137831
75%	455.402780
max	781.344789



[150]:

```
# finally, we export the distance to a csv file named stop_distance.csv
```

```
df_stop_distance_merged.to_csv(r'../data/processed/assignment1/stop_distance.
↪csv', index=False, header=True)
```

4.2 Calculate Speed

We now have the time and distance between stops that we need to calculate speed. Here we will merge the `stop_distance.csv` and `average_time_between_stops_filtered.csv` tables with a division to create a table of average hourly speeds between every stop segment on every line.

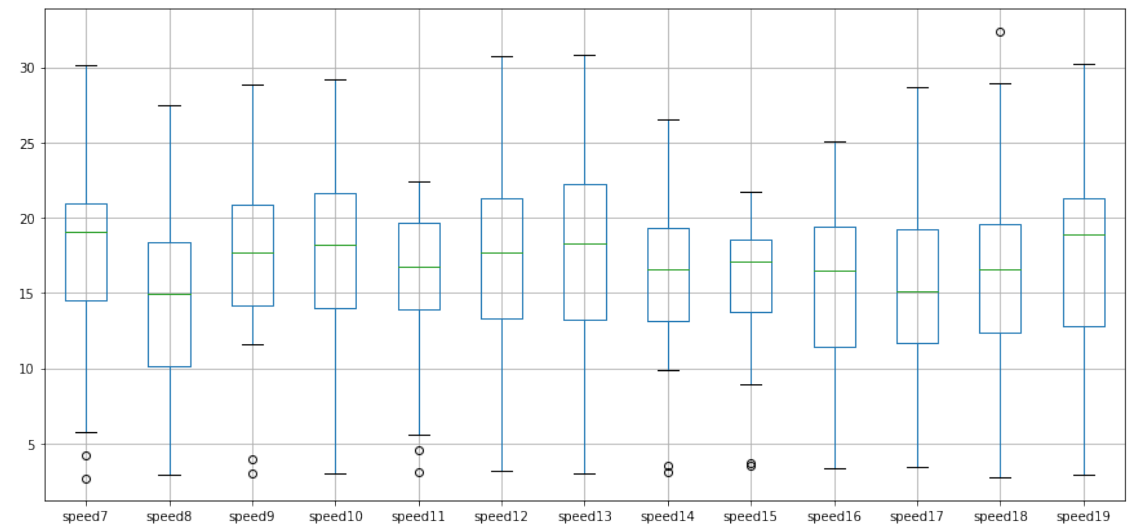
Some speeds resulted in a value of "inf" (distance/0 = infinity). This warrants some additional investigation for future iterations if a greater resolution of data is needed. For now, we will drop any speeds that are showing as "inf" and take a peek at how the speed data for line 72 varies over time with a series of hourly boxplots. We would hypothesize that the vehicle speed would decrease during morning and afternoon rush hour.

[36]:
```
stop_distance_path = '../data/processed/assignment1/stop_distance.csv'
stop_time_path = '../data/processed/assignment1/
↪average_time_between_stops_filtered.csv'

# initialize a dictionary that will be used to make a dataframe and csv file
stop_time = pd.read_csv(stop_time_path)
stop_distance = pd.read_csv(stop_distance_path)
stop_distance['FromStop'].astype('float_', copy=True, errors='raise')
stop_distance['ToStop'].astype('float_', copy=True, errors='raise')

def calculate_speed(time: pd.DataFrame, distance: pd.DataFrame) -> pd.DataFrame:
    merged_time_distance = time.merge(distance, how='left', on=['LineId',␣
↪'FromStop', 'ToStop'])
    for hour in range(0, 24):
        merged_time_distance[f"speed{hour}"] =␣
↪(merged_time_distance['distance'] / merged_time_distance[
            f"{hour}"]) * 3.6
    merged_time_distance.drop(columns=['distance', *[f'{i}' for i in␣
↪range(24)]], inplace=True)
    return merged_time_distance

df_speed = calculate_speed(stop_time, stop_distance)
```

[56]:
```
df_speed.replace([np.inf, -np.inf], np.nan, inplace=True)
df_speed.dropna(subset=["speed9"], how="all", inplace=True)

df_speed_speed_hour = df_speed[['LineId', 'speed7', 'speed8', 'speed9',␣
↪'speed10', 'speed11','speed12','speed13','speed14','speed15','speed16',␣
↪'speed17','speed18','speed19']]
df_speed_speed_hour_72 = df_speed_speed_hour[df_speed_speed_hour['LineId']==72]
```

```
df_speed_speed_hour_72 = df_speed_speed_hour_72[['speed7', 'speed8', 'speed9',␣
↪'speed10', 'speed11','speed12','speed13','speed14','speed15','speed16',␣
↪'speed17','speed18','speed19']]

print(df_speed_speed_hour_72.describe())
print(df_speed_speed_hour_72.boxplot(figsize=(15,7)))
```

```
          speed7     speed8     speed9    speed10    speed11    speed12  \
count  16.000000  25.000000  25.000000  25.000000  25.000000  25.000000
mean   16.906765  14.934410  17.439004  16.738013  15.589888  17.345880
std     7.612827   6.605189   6.499785   6.404245   5.363655   6.471373
min     2.710827   2.945260   2.995581   3.021969   3.103848   3.201245
25%    14.489430  10.172396  14.154441  14.008978  13.926953  13.322901
50%    19.041073  14.905394  17.725066  18.189287  16.703723  17.727997
75%    20.989779  18.400792  20.832077  21.667680  19.688451  21.252134
max    30.112057  27.452810  28.834675  29.156778  22.368208  30.690894

         speed13    speed14    speed15    speed16    speed17    speed18  \
count  25.000000  25.000000  25.000000  25.000000  25.000000  25.000000
mean   17.315772  16.263069  15.653817  15.334685  14.899769  16.851614
std     6.471386   5.776993   4.908359   5.635238   5.987488   6.930831
min     3.020178   3.120710   3.566193   3.333682   3.447917   2.718025
25%    13.235314  13.148890  13.780325  11.421791  11.683318  12.400202
50%    18.316111  16.528582  17.069845  16.505249  15.071444  16.567820
75%    22.257391  19.324886  18.554313  19.393990  19.268272  19.604508
max    30.807340  26.513174  21.732110  25.090903  28.693164  32.397645

         speed19
count  25.000000
mean   17.257796
std     6.433643
min     2.906916
25%    12.830122
50%    18.897960
75%    21.268058
max    30.212687
AxesSubplot(0.125,0.125;0.775x0.755)
```



[18]:
```
df_speed.to_csv(r'../data/processed/assignment1/vehicleSpeed.csv', index=False,␣
 ↪header=True)
```

We can see in the above data that the speed makes sense. > #### This data is sufficiently clean to draw conclusions between major segments or line to line comparisons for general STIB network speed analysis.

We will input this data later on into Kepler.gl for visualization and to compare sub-segments or line to line performance. Most importantly, the data is setup in a way that will allow a user to easily filter through results to see query results immediately.

If more in-depth analysis is desired, for example to look at the performance of a single critical line, it would be necessary to investigate the areas of 0 km/h speed, as well as to look at all the locatons where the speed was calculated as "inf".

assignment_2

December 20, 2021

1 Assignment 2

1.1 Calculating and Analyzing Stop Delays

1.2 What are we looking to accomplish?

Here we will again be loading in files available from STIB API as well as some files that were processed earlier during speed analysis.

Assumptions needed to be made to create a manageable data set for further predictions. An early approach was to match a buses actual arrival time at a stop to the nearest scheduled time, however this resulted in very symmetric delay data — a bus was considered early or late an equal number of times. New limits were put in place that reflect client expectations. A STIB client does not care WHICH bus they are getting on.

For example, if a bus is meant to arrive at 12:10, 12:20 and 12:30, it doesn't matter that a bus that was supposed to arrive at 12:20 actually arrives at 12:30. To manage these scenarios, there is a lower bound in place for how early a bus is allowed to be (current it is set to `200 secounds`). An upper bound is set at `1400 secounds` based on a visual heatmap analysis of the delay times to see choose a reasonable time outside of outliers.

To begin, we will import required libraries and define some helper functions before doing some additional transformation of the data to suit our needs.

[2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from datetime import datetime
import re
import time
```

1.3 Utility Functions

Functions that we use all around our assignment.

[3]:
```python
# get seconds from hh:mm:ss format
def get_sec(time_str):
    """Get Seconds from time."""
    h, m, s = time_str.split(':')
    return int(h) * 3600 + int(m) * 60 + int(s)
```

[4]:
```python
# get hh:mm:ss format from seconds
def get_hhmmss(seconds: int):
  return time.strftime('%H:%M:%S', time.gmtime(seconds))
```

[5]:
```python
# get hour from hh:mm:ss
def get_hour(hour_str: str):
  return int(hour_str.split(':')[0])
```

1.3.1 Data input and processing

In this part of the assignment we are loading the static data and the vehicle snapshots to our notebook because we will need to calculate the delays for the vehicles that or on a stop. We need to see the structure of the data, analyze it and decide how we are going to manipulate it in order to use it for our assignment.

We are using the gtfs files from September 3rd which we will reffere to as **static** files and the vehicle positions as **vehicle positions or dynamic data**.

Let's start by loading the static GTFS data

[38]:
```python
calendar = pd.read_csv('../data/raw/gtfs/calendar.txt')
calendar_dates = pd.read_csv('../data/raw/gtfs/calendar_dates.txt')
routes = pd.read_csv('../data/raw/gtfs/routes.txt')
stops = pd.read_csv('../data/raw/gtfs/stops.txt')
stop_times = pd.read_csv('../data/raw/gtfs/stop_times.txt')
trips = pd.read_csv('../data/raw/gtfs/trips.txt')
```

1.3.2 Stops

We noticed that the stop_id's from stops.csv are not clean. Most of the stops were duplicated or had different records. Most of them were in a format with zeros up front and some letters at the end while still referencing to the same longitude and latitude for that stop.

This is the part where we clean the stops and later use them as cleaned in the rest of the notebook.

[39]:
```python
stops.head(1)
```

[39]:
```
  stop_id  stop_code   stop_name  …  stop_url  location_type  parent_station
0    0089        NaN  MONTGOMERY  …       NaN              0            37.0
```

[1 rows x 10 columns]

[40]:
```
stops['stop_id'] = stops['stop_id'].map(lambda x: int(re.sub('[^0-9]', '', x)))␣
↪# clean the letters and zeros
stops = stops.drop_duplicates() # remove the duplicates['stop_id'] =␣
↪stops['stop_id'].map(lambda x: int(re.sub('[^0-9]', '', x))) # clean the␣
↪letters and zeros
```

[41]:
```
stops.head(1)
```

[41]:
```
   stop_id  stop_code   stop_name  …  stop_url  location_type  parent_station
0       89        NaN  MONTGOMERY  …       NaN              0            37.0

[1 rows x 10 columns]
```

We are doing the same thing for stop_times as we also use a stops column.

[42]:
```
stop_times['stop_id'] = stop_times['stop_id'].map(lambda x: int(re.
↪sub('[^0-9]', '', str(x)))) # clean the letters and zeros
stop_times = stop_times.drop_duplicates() # remove the duplicates
```

1.3.3 Calendar

We can reduce the day columns and create one column holding this information. We also noticed that we only have the following type of patterns in the calendar.csv file:

- 1111100
- 0000010
- 0000001

Therefore, we can conclude that we can use the following notation: 0 - weekday, 1 - saturday, 2 - sunday, 3 - other

[43]:
```
calendar.head(1)
```

[43]:
```
   service_id  monday  tuesday  …  sunday  start_date  end_date
0   234578052       1        1  …       0    20210823  20210831

[1 rows x 10 columns]
```

[44]:
```
# transform calendar_dates
calendar['day'] = calendar.apply(lambda x: 0 if x.monday==1 else 1 if x.
↪saturday==1 else 2 if x.sunday==1 else 3 , axis=1)
```

```
calendar.
↪drop(['monday','tuesday','wednesday','thursday','friday','saturday','sunday'],␣
↪axis=1, inplace=True)
```

[45]:
```
calendar.head(1)
```

[45]:
```
   service_id  start_date  end_date  day
0   234578052    20210823  20210831    0
```

1.4 Joining Static Files

To make things easier ot manage we can deal with one big table where we store all the needed data from the static GTFS files.

We call this table trips_merged as we have information for every single STIB's trip from the GTFS data. From this part on we refer to this table as **trips**.

We can see from which date is a trip is running, which calendar it follows, at what time it is supposed to stop, what the name of the stop is, the line id and other useful information which we can be used to find the delay.

[ ]:
```
# make the big join, we have all the needed information
# del trips_merged
routes = routes[['route_id','route_short_name']]
stops = stops[['stop_id','stop_name']]
trips =␣
↪trips[['route_id','service_id','trip_id','trip_headsign','direction_id']]
stop_times = stop_times[['trip_id','arrival_time','stop_id','stop_sequence']]
trips_merged = trips.merge(stop_times, on='trip_id', how='inner')\
                    .merge(calendar, on='service_id', how='inner')\
                    .merge(stops, on='stop_id', how='inner')\
                    .merge(routes, on='route_id', how='inner')
```

[ ]:
```
trips_merged.head(1)
```

[ ]:
```
   route_id  service_id             trip_id  … day  stop_name route_short_name
0        24   235954071  112387248235954071  …   0   BEEKKANT               87

[1 rows x 13 columns]
```

1.5 Vehicle Positions

Start by loading the vehicle positions. *Note: We have previously merged all .json vehicle files into a .csv one called* `vehiclePositions.csv`

[ ]:
```
vehicle_positions = pd.read_csv('../data/processed/assignment2/vehiclePositions.
↪csv')
```

[ ]:
```
vehicle_positions.head(1)
```

[ ]:

	Timestamp	LineId	DirectionId	DistanceFromPoint	PointId
0	1630914886924	1	8161	1	8012

Because our vehicle_positions have a timestamp, we have to get useful information from that value. Therefor we are adding 3 new columns from the timestamp column.

timestamp -> date, hour, day

- The date is in YYYYMMDD format.
- The hour is extracted in HH:MM:SS.
- The day is either 0 if it is working day, 1 if Saturday, 2 if Sunday. We are deriving these categories from the Timestamp

It is important to notice that we are saving a new file called `vehiclePositionsTime.csv` which we can use later in our notebook.

[ ]:
```
# timestamp -> date, hour, day
# USE ONLY ONCE IF YOU NEED TO GENERATE THE COLUMNS FROM THE BASE FILE␣
↪VEHICLEPOSITOISN

weekdays = ['Monday','Tuesday','Wednesday', 'Thursday','Friday']

vehicle_positions['date'] = pd.to_datetime(vehicle_positions['Timestamp'],␣
↪unit='ms')
vehicle_positions['hour'] = pd.to_datetime(vehicle_positions['Timestamp'],␣
↪unit='ms')
vehicle_positions['day'] = pd.to_datetime(vehicle_positions['Timestamp'],␣
↪unit='ms')

vehicle_positions['date'] = vehicle_positions['date'].dt.strftime("%Y%m%d").
↪astype(int)
vehicle_positions['hour'] = vehicle_positions['hour'].dt.strftime("%H:%M:%S")
vehicle_positions['day'] = vehicle_positions['day'].dt.strftime("%A") \
                    .apply(lambda x: 0 if x in weekdays else 1 if x=='Saturday'␣
↪else 2 if x=='Sunday' else 3)
```

Note: Because this operation is costly, we can't always run this code. Therefore once we run it we save it and we can use the vehiclePositionsTime.csv file.

[ ]:
```
vp = vehicle_positions.copy()
```

```
[ ]: vehicle_positions.to_csv('../data/processed/assignment2/vehiclePositionsTime.
     ↪csv',mode='w', index=False)
```

```
[ ]: vehicle_positions.head(1)
```

```
[ ]:        Timestamp  LineId  DirectionId  …      date      hour  day
     0  1630914886924       1         8161  …  20210906  07:54:46    0

     [1 rows x 8 columns]
```

2 Calculating the Delays

Our goal now is to calculate the delays for the vehicles from the vehicle positions file that are on a certain stop.

The way we will do this is that we will pick the vehicles that are on the stop. As we don't have the vehicle ID in the static files we will have to make an **assumption** for which tripid matches the vehicle the most.

```
[ ]: vehicle_positions = pd.read_csv('../data/processed/assignment2/
     ↪vehiclePositionsTime.csv')
```

```
[ ]: # for a chosen line number get the coresponding subtable from the trips and␣
     ↪vehicle positions
     def getLineAndVehicle(line_number: int):
         line = trips_merged[trips_merged['route_short_name'] == str(line_number)]
         vehicle = vehicle_positions[vehicle_positions['LineId'] == line_number]
         return line, vehicle
```

```
[ ]: # get the stop name from a given stop number
     def getStopName(stop_number: str):
         for k,v in stops_dictionary.items():
             if(stop_number in v):
                 return k
```

Using this function we are removing the vehicle positions that are between the first arrival time in the trips and last expected time or midnight for that exact vehicle.

We are doing this because there are a lot of vehicles who are running outside of their schedule. Meaning that we have timestamps which are at 2 AM, but there are no scheduled busses between 11PM and 5AM.

We are also avoiding all the busses after midnight because they are represented as 00:00:00 in the vehicle positions file, while in the static files 1AM is represented as 25:00:00. We are not able to compare them as when we convert them to seconds, 01:00:00 AM will give much smaller value than 25:00:00 and will match with the earliest scheduled stop, therefore the delay will be very large and without meaning.

As an initial analysis of the STIB network performance this drop data for vehicle movement after midnight made the dataset much easier to manage. For a next revision, this is a location where we would focus to link up late night movement after midnight to the previous day's data.

[ ]:
```
def filterVehicles(line,vehicles):
  first_vehicle = line.arrival_time.min()
  last_vehicle = line.arrival_time.max() # could be > '24:00:00'

  # dropping by earliest vehicles (also removes the after midnight busses)
  # we give a window of 1 minute if it leaves earlier
  vehicles.drop(vehicles.loc[(vehicles['hour'].values <=␣
↪get_hhmmss(get_sec(first_vehicle) - 60))].index, inplace=True)

  # dropping busses after the last vehicle
  # It is optional because if it's past midnight it is dropped by the previous␣
↪drop
  if(last_vehicle < '24:00:00'):
    vehicles.drop(vehicles.loc[(vehicles['hour'].values >= last_vehicle)].
↪index, inplace=True)

  # drop the rows that are not on the bus stop
  vehicles.drop(vehicles.loc[(vehicles['DistanceFromPoint'].values != 0)].
↪index, inplace=True)
```

Here we get the line corresponding to the vehicle, and we filter all the possible trips that might be on that stop between the dates in which our bus is riding and the date as well.

This function will return the list of the possible trips to match .

[ ]:
```
# get the possible trips that match our vehicle snapshot
def matchTimeWithStops(line, vehicle_timestamp):
    return line.loc[(line['stop_id'].values == vehicle_timestamp.PointId) & \
                (line['start_date'].values <= vehicle_timestamp.date) & \
                (line['end_date'].values >= vehicle_timestamp.date) & \
                (line['day'].values == vehicle_timestamp.day)]
```

Once we get the possible trips from `matchTimeWithStops`, we are trying to find the one that is closest to our vehicle's snapshot.

A naive approach is being used, as we are matching vehicles that have a delay closest to zero. We are first getting the difference between the vehicle snapshot and the delays, and then we are finding the minimum value in the array of delays.

Once we have the two closest values to zero we first check it is inside the [min_bound, 0] interval. If it is, then we this is the value that will be returned (i.e. the vehicle is assumed to be early) as a negative delay, represents an early vehicles. Otherwise, we deem the vehicle to be late if that are between [0, max_bound]

This logic is being done in the `getBestDelay` function.

We will numpy vectorization as it is the fastest way to iterate through dataframes.

```
[ ]: def getBestDelay(val1, val2, min_bound, max_bound):
         # [min_bound,0] priority, then [0, max_bound]
         # check if it is between min and 0
         if(min_bound <= val1 <= 0):
             if(min_bound <= val2 <= 0):
                 return max(val1,val2)
             else:
                 return val1
         elif(min_bound <= val2 <=0):
             return val2

         # check if it is between 0 and min
         if(0 <= val1 <= max_bound):
             if(0 <= val2 <= max_bound):
                 return min(val1,val2)
             else:
                 return val1
         elif(0 <= val2 <= max_bound):
             return val2

         return None
```

```
[ ]: def calculateTheDelay(delays, vehicle_row, interval):
         def f(x, vehicle_row):
           return get_sec(vehicle_row) - get_sec(x)

         fvec = np.vectorize(f)
         result_array = fvec(delays.arrival_time, vehicle_row.hour)
         result_array_abs = np.abs(result_array)
         argsort = np.argsort(result_array_abs)

         # there are cases when we only have < 2 values which are very rare.
         try:
             arg1, arg2 = np.argsort(result_array_abs)[:2]
         except ValueError:
             return None

         min_bound = interval[0]
         max_bound = interval[1]

         val1 = result_array[arg1]
         val2 = result_array[arg2]
         best_delay = getBestDelay(val1, val2, min_bound, max_bound)
```

```
    # print(f"{np.unique(delays.stop_id)} - {np.unique(delays.stop_name)} -␣
↪{val1},{val2} -> {best_delay}")
    return best_delay
```

In the following function we are iterating through every line and vehicle snapshots, and we calculate the delay for the vehicles that are on the stop. We are using the three key functions that we mentioned before this one which are the essential logic when we are searching for the delay.

At the end we are creating a new column called 'Delay' in the vehicle position suitable for that line where we add the delay that we have counted. For the snapshots that are not on the stop we give them Null value.

We are also specifying the `INTERVAL` which is the tolerance threshold for a vehicle being early or late. The interval is currently set to be no more than **200 seconds early** or **1400 seconds late**

[ ]:
```
line_numbers = vehicle_positions.LineId.unique() # get all the lines that are␣
↪on the network
INTERVAL = [-200, 1400]

for number in line_numbers:
    line, vehicles = getLineAndVehicle(number) # ex for line 1, we get the␣
↪static files of line 1 and vp of line 1
    filterVehicles(line,vehicles)
    print(f"LINE{number}")
    total_delays = []

    for index, vehicle_row in vehicles.iterrows(): # go through each bus that␣
↪is on the stop TBD: iterrows is slow
        delays = matchTimeWithStops(line, vehicle_row) # get the delays from␣
↪the matched possible stoptimes for that vehicle in that line

        if delays.empty != True: # avoid when it doesn't find anything
            final_delay = calculateTheDelay(delays, vehicle_row, INTERVAL)
            total_delays.append(final_delay)
        else:
            total_delays.append(None) # it is possible that we don't find␣
↪anything from calculateTheDelay (ex. missing stops)

    vehicles['delay'] = total_delays
    vehicles.to_csv(f'../data/processed/assignment2/vehicleDelaysLine{number}.
↪csv',mode='w', index=False)
```

[ ]:
```

```

Because we are setting the interval manually, we would like to see the average difference between every bus that comes to the stop for every line. This gives us an idea of the correct interval value to set.

As a future improvement, this limit value is something that can be improved by having upper and lower bounds unique to each bus and each bus stop. For example, some buses run **once every hour**. These buses should not follow the same rules for being early or late as a bus that runs every **10 minutes**. The vehicle frequency also changes throuhgout the day as additional buses are added to or removed from the line.

[ ]:
```
line, _ = getLineAndVehicle(71)

ld = line[line['direction_id'] == 0]

avg_diff1 = {}

for number in vehicle_positions.LineId.unique():

    line, _ = getLineAndVehicle(number)
    rush_hour = line[(line['arrival_time'] > '15:00:00') &␣
 ↪(line['arrival_time'] < '16:00:00')]
    service_id = rush_hour.iloc[-1,1]
    stop_id = rush_hour.iloc[-1,6]

    line.arrival_time = line.arrival_time.apply(lambda x: get_sec(x))

    avg_diff1[number] = np.mean(np.diff(np.unique(line[(line['stop_id'] ==␣
 ↪stop_id) & (line['service_id'] == service_id)].arrival_time)))
```

[ ]:
```
avg_diff1
```

[ ]:
```
avg_diff
```

[ ]:
```
{1: 424.6125,
 2: 458.40277777777777,
 3: 484.61538461538464,
 4: 481.2676056338028,
 5: 412.5297619047619,
 6: 439.9415584415584,
 7: 590.4201680672269,
 8: 605.1282051282051,
 9: 640.3636363636364,
 12: 439.6363636363636,
 13: 886.578947368421,
 14: 779.3406593406594,
 17: 894.6153846153846,
 19: 549.4488188976378,
 20: 719.3548387096774,
 21: 860.0,
 25: 603.5897435897435,
 27: 1464.878048780488,
 28: 447.4172185430464,
```

```
29: 888.0,
33: 1202.5,
34: 1568.5714285714287,
36: 435.97402597402595,
37: 1070.0,
38: 676.5833333333334,
39: 697.7319587628866,
41: 905.5211267605633,
42: 888.3333333333334,
43: 1064.27868852459,
44: 718.125,
45: 740.4545454545455,
46: 1444.0,
47: 286.80497925311204,
48: 468.6510067114094,
49: 8115.0,
50: 1310.9433962264152,
51: 592.1311475409836,
53: 739.7802197802198,
54: 789.3975903614457,
55: 966.6666666666666,
56: 880.0,
57: 917.8947368421053,
58: 2585.4545454545455,
59: 748.6666666666666,
60: 850.9090909090909,
61: 862.1917808219179,
62: 13960.0,
63: 678.6,
64: 719.3478260869565,
65: 677.4,
66: 694.639175257732,
69: 1023.0769230769231,
70: 1219.4117647058824,
71: 484.6478873239437,
72: 3135.0,
74: 954.0,
75: 964.6875,
76: 500.74766355140184,
77: 1016.9230769230769,
78: 1000.3636363636364,
79: 803.855421686747,
80: 3469.4117647058824,
81: 603.0508474576271,
82: 540.4545454545455,
83: 892.8,
86: 787.7386363636364,
```

```
87: 579.1525423728814,
88: 912.6760563380282,
89: 385.17241379310343,
92: 583.3613445378152,
93: 624.0,
95: 1076.0,
97: 593.2758620689655,
98: 1329.7674418604652}
```

Note: All the subtables are saved as a new file vehicleDelaysLine{number}.csv, these files are then merged in `vehicleDelays.csv`

3 Cleaning the delays

In this section we will use the calculated delays and remove the null values and do some data exploration with our delays.

vehicleDelays.csv is generated by a shell comand from the vehcileDelaysLine{number}.csv files:

```
head -1 vehicleDelaysLine1.csv > vehicleDelays.csv
tail -n -q +2  vehicleDelaysLine* >> vehicleDelays.csv
```

[22]:
```
# We are loading the vehicle delays here from a new file or we can use the same
delays_uncleaned = pd.read_csv('../data/processed/assignment2/vehicleDelays.
↪csv')
# removing the None values from delay.
delays = delays_uncleaned.dropna(subset=['delay'])
```

[23]:
```
delays = delays.drop_duplicates() # we could have multiple snapshots at the␣
↪same time for the same vehicle
```

As there have been null values when calculating the delays in vehicle positions, mostly due to not having the static gtfs timetables for some dates, we are dropping almost 25% of the data.

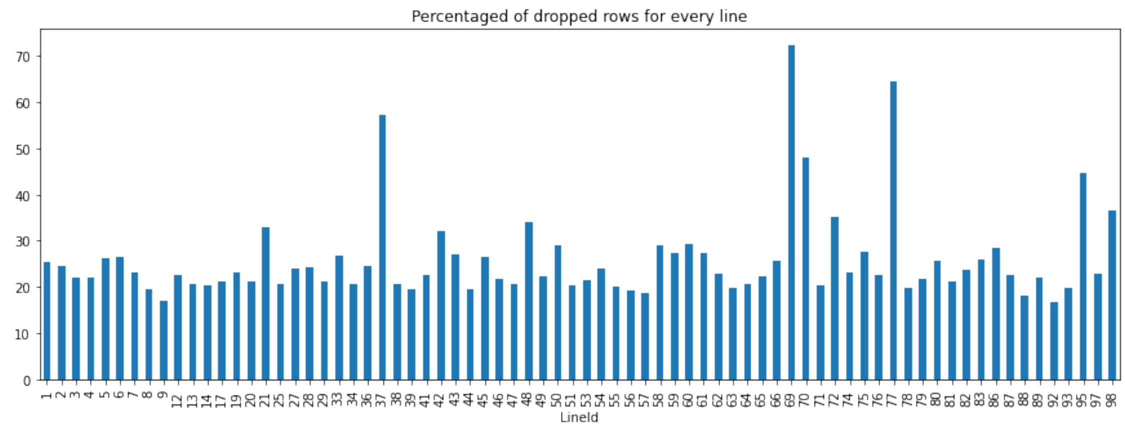
[24]:
```
print(f"Delays before cleaning:\t{len(delays_uncleaned.index) }")
print(f"Delays after dropping the null values:\t{len(delays.index)}")
print(f"Percentage of null values dropped:\t{(1 - (len(delays.index) /␣
↪len(delays_uncleaned.index))) * 100 }")
```

```
Delays before cleaning: 6416423
Delays after dropping the null values:  4274610
Percentage of null values dropped:      33.38017147560252
```

We also see that by line we get peaks of dropped data for lines like 37, 69, 70, 95. One of the reasons is because some stops in the vehicle position files do not exist in the gtfs file. An example for that would be the terminus Grand-Place which ID is not in the GTFS files.

[25]:
```
dropped_rows_by_line = delays_uncleaned.groupby(['LineId'])['delay'].agg(lambda␣
↪x: x.isnull().sum() / len(x) *100)

dropped_rows_by_line.plot(kind='bar', figsize=(15,5), title="Percentaged of␣
↪dropped rows for every line")
```

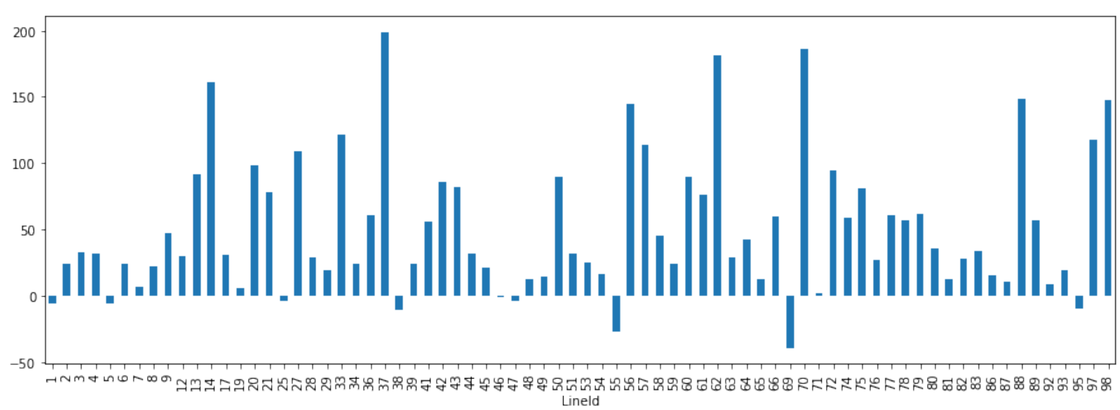
[25]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fed75cea3d0>
```



We should now check the outliers and see how our data is being distributed

[26]:
```
delays.groupby('LineId')['delay'].median().plot(kind='bar', figsize=(15,5))
```

[26]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fed75520550>
```

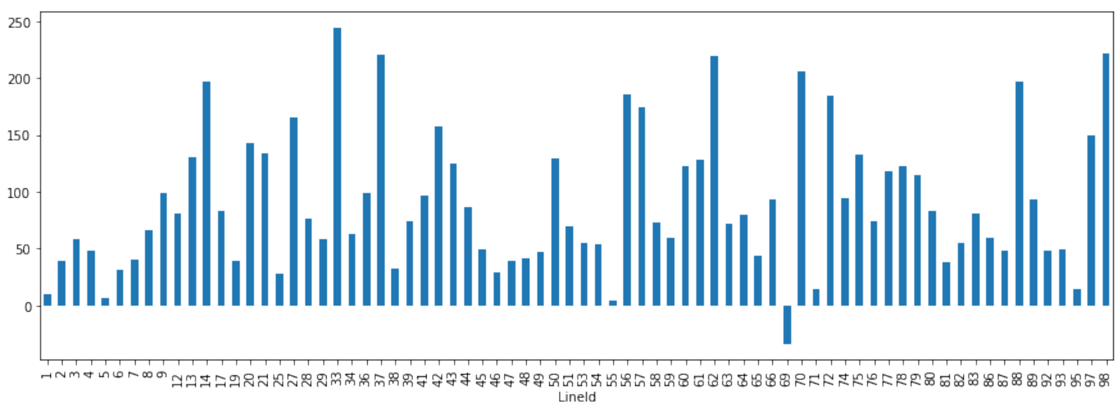


We see that the median values are mostly positive, which means that the for each line we have mostly delays. This visualization also shows us which lines are "bad players". Although we are looking at STIB network performance as a whole right now, we could dive into a certain line's data to ensure that the cleaning and assumption make sense to the particular scenario. This would be an important next step before making any business decisions on a line by line basis.

Now let's see the mean delays for each of our lines.

[27]:
```python
delays.groupby('LineId')['delay'].mean().plot(kind='bar', figsize=(15,5))
```

[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fed770be250>
```



From both mean and median we see that the line number 69 has negative values which means that it is early.

We can also notice how metros and trams are having very low delays at around 50 seconds because they are going either underground where there is no other traffic or they have their own signalization when they are on the ground.

[28]:
```python
delays.to_csv('../data/processed/assignment2/delaysCleaned.csv', index=False)
```

4 Export data

We are preparing the data for the visualization part

[29]:
```python
delays = pd.read_csv('../data/processed/assignment2/delaysCleaned.csv')
delays = delays.dropna()
```

[30]:
```
delays = delays.drop(columns=['DistanceFromPoint'], axis=1)
delays.columns = ['Timestamp', 'LineId', 'TerminusStop', 'CurrentStop',␣
↪'Date','Hour','Day','Delay']
delays['Delay'] = delays['Delay'].astype(int)
delays['ExpectedArrivalTime'] = [ get_hhmmss(get_sec(x) - y) for x, y in␣
↪zip(delays['Hour'], delays['Delay'])]
delays = delays.drop_duplicates()
```

[31]:
```
delays.head()
```

[31]:
```
       Timestamp  LineId  TerminusStop  …  Day  Delay ExpectedArrivalTime
0  1630914886924      12          9600  …    0    286            07:50:00
1  1630914886924      12          1780  …    0    286            07:50:00
2  1630914886924      12          9600  …    0    -14            07:55:00
3  1630914886924      12          1780  …    0    -74            07:56:00
4  1630914917746      12          9600  …    0    317            07:50:00

[5 rows x 9 columns]
```

While visualizing the data in rapidminer software, we noticed that the vehicles spend a lot of time on the stops. For one stop, we can have 5-6 snapshots of that vehicle staying on the stop.

We also noticed a lot of vehicle timestamp snapshots on the very first stop of the trip. We **assume** that the drivers have started the device to run before leaving the stop and officially starting the route.

We are demonstrating this case by looking at the time a bus stays on a certain stop.

[32]:
```
delays_line5 = delays[delays['LineId'] == 5]
```

[34]:
```
start_stops = np.unique(delays_line5['TerminusStop'].values)
intermediate_delays = delays.loc[(delays['LineId'] == 5) &␣
↪~(delays['CurrentStop'].isin(start_stops))]
intermediate_delays.
↪groupby(['LineId','Date','Day','TerminusStop','CurrentStop',␣
↪'ExpectedArrivalTime'])['Delay'].agg(list)
```

[34]:
```
LineId  Date      Day  TerminusStop  CurrentStop  ExpectedArrivalTime
5       20210906  0    8161          8652         19:20:34
[-10, 22]
                                     8662         19:21:56
[-28, 5]
                                     8672         19:23:19
[-15, 17]
                                     8692         19:25:47
[-4]
                                     8702         19:27:12
[3]
```

```
…
        20210918  1    8731          8261         19:32:28                [165,
198, 230, 262, 294]
                       8733          8271         09:19:59
[137, 168]
                                     8281         09:21:22
[181]
                                     8291         09:22:36
[139, 169]
                                     8741         09:24:10
[139, 170, 202]
Name: Delay, Length: 79358, dtype: object
```

Now, we decide we will only consider the first snapshot when a vehicle arrives at an "intermediate" stop, and we discard the other timestamps on that stop for that vehicle. This means that we consider the time of first arrival as the "true" time for which we will compare against the scheduled time.

Note: An intermediate stop is one that is not a terminus stop or first stop of the line.

In order to avoid these problems we are dropping some rows: - We only take the last snapshot when our vehicle is on the first stop. - We take the first stop when the vehicle reaches an intermediate stop and on the last/terminus stop.

In the following code we are treating all the cases.

[35]:
```
intermediate_delays = intermediate_delays.
 ↪drop_duplicates(subset=['LineId','Date','Day','TerminusStop','CurrentStop',␣
 ↪'ExpectedArrivalTime'], keep='first')
intermediate_delays.
 ↪groupby(['LineId','Date','Day','TerminusStop','CurrentStop',␣
 ↪'ExpectedArrivalTime'])['Delay'].agg(list)
```

[35]:
```
LineId  Date      Day  TerminusStop  CurrentStop  ExpectedArrivalTime
5       20210906  0    8161          8652         19:20:34               [-10]
                                     8662         19:21:56               [-28]
                                     8672         19:23:19               [-15]
                                     8692         19:25:47                [-4]
                                     8702         19:27:12                 [3]
                                                                        …
        20210918  1    8731          8261         19:32:28               [165]
                       8733          8271         09:19:59               [137]
                                     8281         09:21:22               [181]
                                     8291         09:22:36               [139]
                                     8741         09:24:10               [139]
Name: Delay, Length: 79358, dtype: object
```

[ ]:
```
line_numbers = delays.LineId.unique()
drop_condition =␣
↪['LineId','Date','Day','TerminusStop','CurrentStop','ExpectedArrivalTime']

all_delays = []

for line in line_numbers:
  # dropping the start stops
  start_stops = np.unique(delays[delays['LineId'] == line]['TerminusStop'].
↪values)

  start_delays = delays.loc[(delays["LineId"] == line) & \
                            (delays['TerminusStop'] != delays['CurrentStop']) &␣
↪ # it is not the terminus stop
                            (delays['CurrentStop'].isin(start_stops))] # the␣
↪current stop is a start or terminus stop
  start_delays = start_delays.drop_duplicates(subset=drop_condition,␣
↪keep='last')

  # dropping the intermediate stops
  intermediate_delays = delays.loc[(delays['LineId'] == line) & \
                              ~(delays['CurrentStop'].isin(start_stops))]

  intermediate_delays = intermediate_delays.
↪drop_duplicates(subset=drop_condition, keep='first')

  # dropping the terminus stops
  terminus_delays = delays.loc[(delays['LineId'] == line) & \
                              (delays['TerminusStop'] ==␣
↪delays['CurrentStop'])]

  terminus_delays = terminus_delays.drop_duplicates(subset=drop_condition,␣
↪keep='first')

  # merging them all in one
  df_concat = pd.concat([start_delays.reset_index(drop=True),\
                         intermediate_delays.reset_index(drop=True),\
                         terminus_delays.reset_index(drop=True)])

  all_delays.append(df_concat.reset_index(drop=True))

# for each line we merge the delays
final_filtered_delays = pd.concat(all_delays)
```

[ ]:
```
final_filtered_delays
```

```
[ ]:            Timestamp  LineId  TerminusStop  …  Day  Delay ExpectedArrivalTime
0       1630914886924      12          9600  …    0    -14            07:55:00
1       1630914886924      12          1780  …    0    -74            07:56:00
2       1630915011138      12          1780  …    0    411            07:50:00
3       1630915074628      12          9600  …    0     -6            07:58:00
4       1630915202427      12          1780  …    0    -58            08:01:00
…                 …       …              …    …   …      …                 …
11656   1632002984861       9          1687  …    1     44            22:09:00
11657   1632004253330       9           473  …    1   -127            22:33:00
11658   1632004724486       9          1687  …    1    -16            22:39:00
11659   1632005097438       9           473  …    1   -123            22:47:00
11660   1632006001041       9           473  …    1   -119            23:02:00

[1626465 rows x 9 columns]
```

```
[ ]: final_filtered_delays.to_csv('../data/processed/assignment2/
     ↪final_filtered_delays.csv', index=False)
```

5 Final Output

These are the final delays for each line, for each stop. This will form the input used in Kepler.gl to visualize STIB vehicle performance.

```
[ ]: final_filtered_delays.head()
```

```
[ ]:        Timestamp  LineId  TerminusStop  …  Day  Delay ExpectedArrivalTime
0   1630914886924      12          9600  …    0    -14            07:55:00
1   1630914886924      12          1780  …    0    -74            07:56:00
2   1630915011138      12          1780  …    0    411            07:50:00
3   1630915074628      12          9600  …    0     -6            07:58:00
4   1630915202427      12          1780  …    0    -58            08:01:00

[5 rows x 9 columns]
```

6 Statistics

In this part we will look at some interesting statistics from the data to see what kind of insight we can develop. We chose line 5 as a case study.

From the visualizations below we can say the following: - Line 5 typically arrives at a stop up to 50s early during morning and afternoon rush-hour. It would be interesting to see what time a metro vehicle leaves a stop (e.g. is it LEAVING the stop early as well, or late?) - Line 5 begins to see delays in service towards the evening well after afternoon rush-hour. Is this due to fewer vehicles running? In the future we could look to see how a line performs with a changing number of vehicles running. - The delays in Line 5 get progressively longer as the vehicle moves down the line. This indicates it is arriving less and less early at each stop. The large delay in arrival time at Erasme stop (the terminus) could be impacted by how the data is being treated before analysis.

[36]:
```
final_delays = pd.read_csv('../data/processed/assignment2/final_filtered_delays.
 ↪csv')
```

[46]:
```
final_delays = final_delays.merge(stops[['stop_name', 'stop_id']],␣
 ↪left_on='CurrentStop', right_on='stop_id')
final_delays = final_delays.drop(columns=['stop_id'])
```

[47]:
```
final_delays.head()
```

[47]:
```
       Timestamp  LineId  TerminusStop  …  Delay  ExpectedArrivalTime
stop_name
0  1630914886924      12          9600  …    -14             07:55:00
TRONE
1  1630915074628      12          9600  …     -6             07:58:00
TRONE
2  1630915268784      12          9600  …      8             08:01:00
TRONE
3  1630915876546      12          9600  …    376             08:05:00
TRONE
4  1630916005472      12          9600  …    -95             08:15:00
TRONE

[5 rows x 10 columns]
```

[48]:
```
# creating the hour_interval column to know in which hour is the delay
final_delays['hour_interval'] = final_delays['Hour'].apply(lambda x: str(x.
 ↪split(':')[0] + ':00'))
```

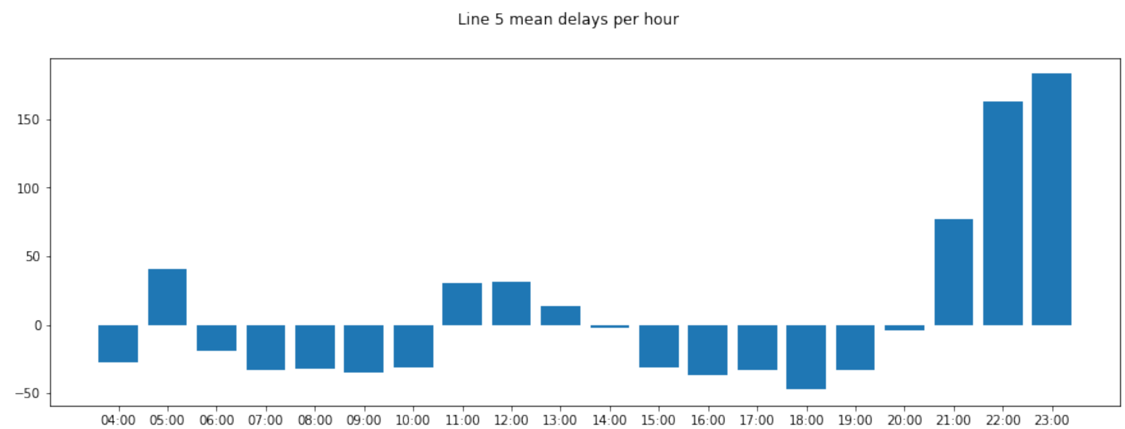
[49]:
```
line5 = final_delays[final_delays['LineId'] == 5]
```

[51]:
```
line5.head()
```

[51]:
```
            Timestamp  LineId  …  stop_name  hour_interval
146969  1630915011138       5  …     MERODE          07:00
146970  1630915334425       5  …     MERODE          08:00
146971  1630915653306       5  …     MERODE          08:00
146972  1630915971167       5  …     MERODE          08:00
146973  1630916356000       5  …     MERODE          08:00

[5 rows x 11 columns]
```

[ ]:
```python
# What point of the day do we have the biggest delays for line 5?

line5_delay = line5.groupby('hour_interval', as_index=False)['Delay'].mean()
line5_delay['hour_interval'].values
line5_delay['Delay'].values

fig, ax = plt.subplots(figsize=(15, 5))
ax.bar(line5_delay['hour_interval'].values,line5_delay['Delay'].values)
fig.suptitle('Line 5 mean delays per hour')
```

[ ]:
```
Text(0.5, 0.98, 'Line 5 mean delays per hour')
```



[ ]:
```python
# stops that have the biggest delay on line 5
delay_by_stop = line5.groupby('stop_name')['Delay'].mean()
delay_by_stop = delay_by_stop.sort_values()
delay_by_stop.tail()
```

[ ]:
```
stop_name
SAINTE-CATHERINE    -4.354205
SCHUMAN              1.979281
BIZET                4.085119
DELTA                8.794253
ERASME              67.821355
Name: Delay, dtype: float64
```

[ ]:
```python
# stop that are very early on line 5
delay_by_stop.head()
```

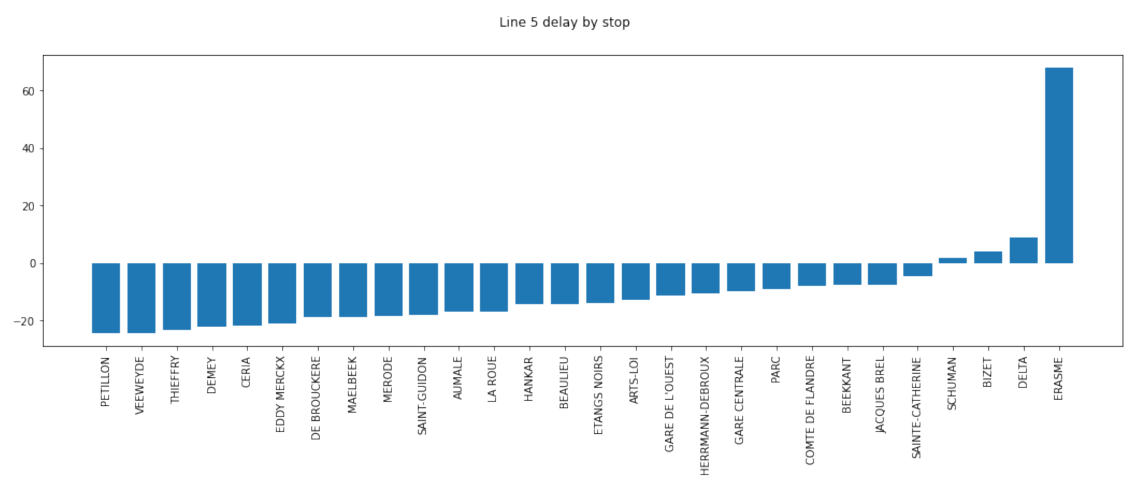
[ ]:
```
stop_name
PETILLON   -24.316170
```

```
VEEWEYDE   -24.226810
THIEFFRY   -23.251196
DEMEY      -22.033838
CERIA      -21.859994
Name: Delay, dtype: float64
```

[ ]:
```
# stops that are the closest to be on time on line 5
delay_by_stop[delay_by_stop.abs().sort_values().head(10).index]
```

[ ]:
```
stop_name
SCHUMAN              1.979281
BIZET                4.085119
SAINTE-CATHERINE    -4.354205
JACQUES BREL        -7.488394
BEEKKANT            -7.512859
COMTE DE FLANDRE    -7.803862
DELTA                8.794253
PARC                -8.818603
GARE CENTRALE       -9.607653
HERRMANN-DEBROUX   -10.549991
Name: Delay, dtype: float64
```

[ ]:
```
line5_stop_delay = pd.DataFrame(delay_by_stop).reset_index()

fig, ax = plt.subplots(figsize=(18, 5))
plt.bar(line5_stop_delay['stop_name'].values,line5_stop_delay['Delay'].values)
fig.suptitle('Line 5 delay by stop')
plt.xticks(rotation=90)
plt.show()
```



[ ]:
```
# line that has the biggest delay
delay_by_line = final_delays.groupby('LineId')['Delay'].mean()
delay_by_line = delay_by_line.sort_values()
delay_by_line.tail()
```

[ ]:
```
LineId
55     8.430649
2     14.928415
7     20.766036
87    50.857861
38    60.944335
Name: Delay, dtype: float64
```

[ ]:
```
# line that is early
delay_by_line.head()
```

[ ]:
```
LineId
5    -10.071684
55     8.430649
2     14.928415
7     20.766036
87    50.857861
Name: Delay, dtype: float64
```

[ ]:
```
# line that is most on time
delay_by_line[delay_by_line.abs().sort_values().head(5).index]
```

[ ]:
```
LineId
55     8.430649
5    -10.071684
2     14.928415
7     20.766036
87    50.857861
Name: Delay, dtype: float64
```

[ ]:
```
# average delay per stop on line5
line5.groupby(['CurrentStop', 'stop_name'])['Delay'].mean()
```

[ ]:
```
CurrentStop  stop_name
8011         DE BROUCKERE        -31.279582
8012         DE BROUCKERE         -4.520025
8021         GARE CENTRALE        -8.368298
8022         GARE CENTRALE       -10.956246
8031         PARC                 -9.613838
8032         PARC                 -7.948220
8041         ARTS-LOI            -14.071846
8042         ARTS-LOI            -11.083925
```

8051	MAELBEEK	-12.621399
8052	MAELBEEK	-24.824190
8061	SCHUMAN	13.246686
8062	SCHUMAN	-10.657401
8071	MERODE	-21.985673
8072	MERODE	-14.106796
8201	THIEFFRY	-33.692564
8202	THIEFFRY	-11.049287
8211	PETILLON	-17.025852
8212	PETILLON	-31.980235
8221	HANKAR	-12.743097
8222	HANKAR	-16.095641
8231	DELTA	3.451786
8232	DELTA	14.435575
8241	BEAULIEU	-8.927445
8242	BEAULIEU	-20.164722
8251	DEMEY	-22.803150
8252	DEMEY	-21.159091
8261	HERRMANN-DEBROUX	-74.698406
8262	HERRMANN-DEBROUX	34.336953
8271	SAINTE-CATHERINE	-6.228859
8272	SAINTE-CATHERINE	-2.331206
8281	COMTE DE FLANDRE	-2.332554
8282	COMTE DE FLANDRE	-13.643793
8291	ETANGS NOIRS	-20.156896
8292	ETANGS NOIRS	-6.583440
8641	ERASME	64.988146
8642	ERASME	69.342273
8651	EDDY MERCKX	-16.831543
8652	EDDY MERCKX	-25.986257
8661	CERIA	-3.359375
8662	CERIA	-41.185185
8671	LA ROUE	-13.148571
8672	LA ROUE	-20.656005
8681	BIZET	-12.327616
8682	BIZET	22.439258
8691	VEEWEYDE	-21.171541
8692	VEEWEYDE	-27.700128
8701	SAINT-GUIDON	-16.300562
8702	SAINT-GUIDON	-19.673548
8711	AUMALE	1.897605
8712	AUMALE	-36.308550
8721	JACQUES BREL	1.295033
8722	JACQUES BREL	-16.896795
8731	GARE DE L'OUEST	15.070885
8732	GARE DE L'OUEST	-42.177940
8733	GARE DE L'OUEST	1062.200000

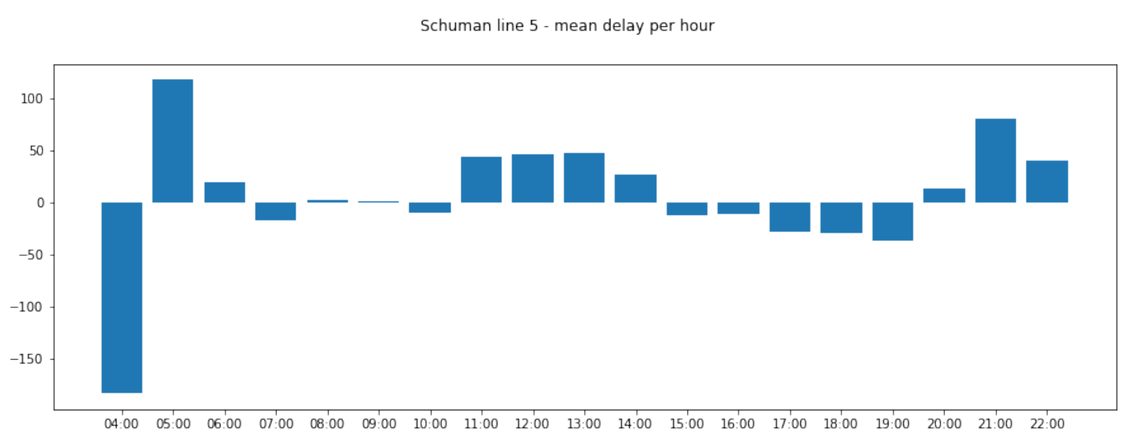
```
8741          BEEKKANT                  -5.907440
8742          BEEKKANT                  -9.227785
Name: Delay, dtype: float64
```

[ ]:
```
# Get which line is slow at what point of the day
final_delays.groupby(['LineId', 'hour_interval'])['Delay'].median()
```

[ ]:
```
LineId  hour_interval
2       05:00             66.0
        06:00             -1.0
        07:00             30.0
        08:00              6.0
        09:00            -12.0
                           …
87      18:00            -24.0
        19:00             46.0
        20:00            217.0
        21:00            177.0
        22:00             78.5
Name: Delay, Length: 116, dtype: float64
```

[ ]:
```
# Get the delays for Schuman stop at different time of day on line 5

stop_by_hour = line5.groupby(['stop_name', 'CurrentStop', 'hour_interval'],␣
 ↪as_index=False)['Delay'].mean()

schuman_stop = stop_by_hour[stop_by_hour['CurrentStop'] == 8061]
x,y = schuman_stop['hour_interval'].values, schuman_stop['Delay'].values
fig, ax = plt.subplots(figsize=(15, 5))
ax.bar(x,y)
fig.suptitle('Schuman line 5 - mean delay per hour')
```

[ ]:
```
Text(0.5, 0.98, 'Schuman line 5 - mean delay per hour')
```



[ ]:
```
# Foreach interval get the stop that is the busiest on the line
final_delays.groupby(['hour_interval','stop_name'])['Delay'].mean()
```

[ ]:
```
hour_interval  stop_name
04:00          ARAUCARIA         -75.250000
               ARSENAL           118.800000
               BASCULE           184.285714
               BOILEAU            54.800000
               BORDET STATION    310.333333
                                    …
23:00          THIEFFRY          265.133333
               THOMAS             84.000000
               VAN PRAET         412.000000
               VERBOEKHOVEN       41.600000
               VUB               133.416667
Name: Delay, Length: 2597, dtype: float64
```

assignment_3

December 20, 2021

0.1 What are we looking to accomplish?

What we want to do is forecast the arrival time of a line's vehicle across all it's stops given a start time at the departure stop. We will also test the accuracy of our forecasting by splitting our dataset into training and testing subsets.

From the first assignment we have a dataset that details the average time it takes for a vehicle to go from stop to stop at every hour of the day. We have 16 tables from 06/09/2021 till 21/09/2021. The travel time in this dataset includes the time a vehicle is waiting at each stop, so we will not need to account for any additional time if we use this dataset to build our prediction model.

To accomplish this, a SARIMA (Seasonal Auto Regressive Integrate Moving Average) time-series model was employed. This allows us to account for the seasonality of the data and predict the next data points behaviour based on the previous data point(s) as well as the data points from the previous seasonal cycle.

With this approach, an individual SARIMA will be created for each pair of stops, on each line, at each hour. This will result in approximately [170 lines * 4700 stops * 24 lines] 19M SARIMA functions. The function will predict the future time between stops. We can precalculate each of these predictions and save them in a lookup table for fast retrieval when needed. This way, we do not need to recompute the model every time it is called.

Technically, this approach gives us an accurate prediction for the *day following* the training data, however given the consistency of the STIB network, the model can be used for any day. An improvement to this approach would be to generate SARIMA predictions for different days of the week, and different months of the year. With access to 365 days worth of data, the model can be employed to predict today's time based on yesterday, last week, and last year, which would account for seasonality on a year-to-year basis.

So… let's get started. We will (as usual) import our needed libraries and files and complete some data manipulation to get our data into the desired format.

[1]:
```
import pandas as pd
import json
import csv
import os
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

```python
import math
import numpy as np
```

[6]:
```python
days = os.listdir('./processed_data/daily_average_time_between_stops_filtered/')
days = [x[:-4] for x in days]
average_times_per_day = []
for path in days:
    average_times_per_day.append(pd.read_csv(r'./processed_data/
 ↪daily_average_time_between_stops_filtered/'+path+'.csv'))
```

Let's visualize one of those tables:

[4]:
```python
average_times_per_day[1]
```

[4]:
```
      LineId  FromStop  ToStop     0    1  2  3    4      5      6  …  \
0          1      8733    8742   0.0  0.0  0  0  0.0    0.0   64.0  …
1          1      8742    8292  90.0  0.0  0  0  0.0   93.0  101.6  …
2          1      8292    8282  90.0  0.0  0  0  0.0   60.0   69.6  …
3          1      8282    8272  91.0  0.0  0  0  0.0   95.0  118.4  …
4          1      8272    8012  61.0  0.0  0  0  0.0   64.0   69.0  …
…        …       …       …     …   … .. ..  …     …     …   …
3326      98      2610    9686   0.0  0.0  0  0  0.0   94.0  110.0  …
3327      98      9686    5964   0.0  0.0  0  0  0.0  128.0   94.5  …
3328      98      5964    3815   0.0  0.0  0  0  0.0   61.0   61.0  …
3329      98      3815    3851   0.0  0.0  0  0  0.0    0.0   62.0  …
3330      98      3851    2382   0.0  0.0  0  0  0.0    0.0  419.5  …

          14      15      16      17      18      19      20      21     22  \
0     416.00  383.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00
1      96.00   95.83   96.60  105.67  103.50   89.67   99.60   99.00  93.67
2      62.50   67.17   63.50  219.17  218.33   78.83   57.00   62.17  67.50
3     105.67  108.40  101.00   94.80  108.80   99.33  103.67   93.50  88.17
4      79.83   63.40   85.50   63.40   69.40   62.20   58.17   73.50  71.67
…        …      …      …      …      …      …      …      …      …      …
3326  127.00  134.00   30.50   85.00   78.50   97.00   63.00   46.50   0.00
3327  129.00  125.75  168.50  120.00  113.00  109.00  127.00   92.50   0.00
3328   83.33   64.00   64.33   52.00   95.00   79.00   30.00   64.00   0.00
3329   62.67   48.50   52.67   52.00   30.00   30.00   31.00   94.00   0.00
3330  371.33  327.33  308.00  231.00  258.00  174.00  272.67  246.00   0.00

          23
0       0.00
1     124.00
2      78.50
3      83.33
4      72.33
…        …
```

```
3326    0.00
3327    0.00
3328    0.00
3329    0.00
3330    0.00

[3331 rows x 27 columns]
```

Let's take one segment to analyze:

[7]:
```python
# segment departure stop
from_stop = 8282

# segment arrival stop
to_stop = 8272
```

Create the time series for that segment joining all days' hours:

[8]:
```python
# Time series list:
time_series_of_segment = []

# Populate the time series list_
for day in average_times_per_day:
    for hour in range(24):
        time_series_of_segment.append(day[(day['FromStop']==from_stop) &␣
 ↪(day['ToStop']==to_stop)][str(hour)].values[0])

# Create the labels for every element of time series:
date_and_time = []
for day in days:
    for hour in range(24):
        date_and_time.append(day + '_' + str(hour))

# Create a table with showing the dates and hours and the corresponding average␣
 ↪time between stops:
time_series_of_segment_df = pd.DataFrame(data={'date & hour': date_and_time,␣
 ↪'time': time_series_of_segment })
time_series_of_segment_df
```

[8]:

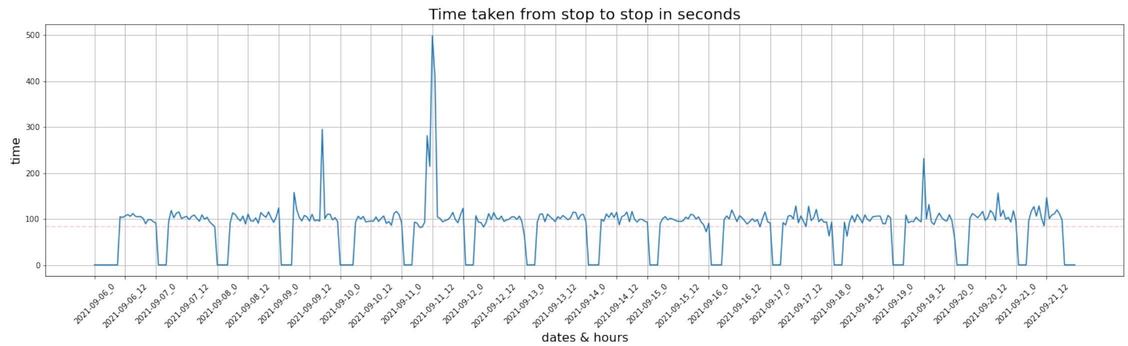
	date & hour	time
0	2021-09-06_0	0.0
1	2021-09-06_1	0.0
2	2021-09-06_2	0.0
3	2021-09-06_3	0.0
4	2021-09-06_4	0.0
..	…	…
379	2021-09-21_19	0.0

```
380  2021-09-21_20   0.0
381  2021-09-21_21   0.0
382  2021-09-21_22   0.0
383  2021-09-21_23   0.0

[384 rows x 2 columns]
```

The goal is to create a model that, based on this table, predicts new values. Let's analyze how the time series is behaving.

[9]:
```
#-----------------------------------------------------------------
# GRAPH
#-----------------------------------------------------------------

# size:
plt.figure(figsize=(25,6))

# title:
plt.title('Time taken from stop to stop in seconds', fontsize=20)

# x axis:
# x values
x = []
for index in range(time_series_of_segment_df.index.stop):
    x.append(index)
# x ticks
my_xticks = time_series_of_segment_df['date & hour']
plt.xticks(x[::12], my_xticks[::12], rotation='45')
# # x label
plt.xlabel("dates & hours", fontsize=16)

# y axis:
# y values
y = time_series_of_segment_df['time']
# y label
plt.ylabel("time", fontsize=16)

# mean
plt.axhline(time_series_of_segment_df['time'].mean(), color='r', alpha=0.2,␣
 ↪linestyle='--')

# create plot
plt.plot(x, y)
# add grids to plot
plt.grid()
# show plot
plt.show()
```

```
#-----------------------------------------------------------------
```



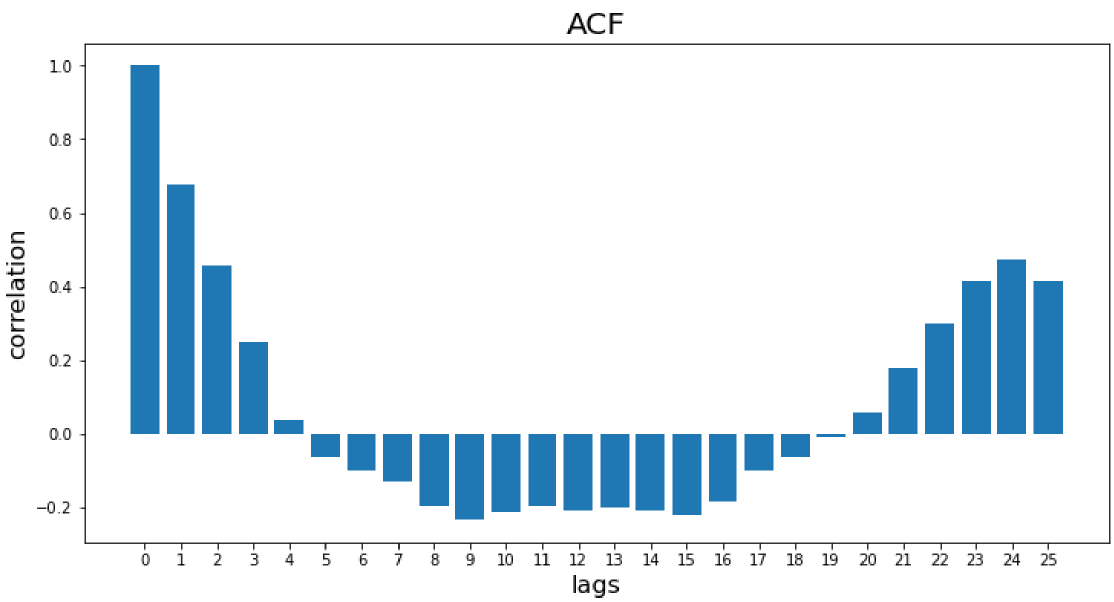
From the above graph we can determine that the time series does not have a trend but there is a seasonality every 24 hours as expected. We will plot the Auto Correlation Function (ACF) and the Partional Auto Correlation Function to help determine appropriate values to use in our SARIMA model.

We will also define the size of our training and testing sets. We will take the first 90% of data to train and remaining 10% to test. We will then fit a SARIMA model and test the accuracy of the predictions with the testing data.

0.1.1 Auto Correlation Function (ACF)

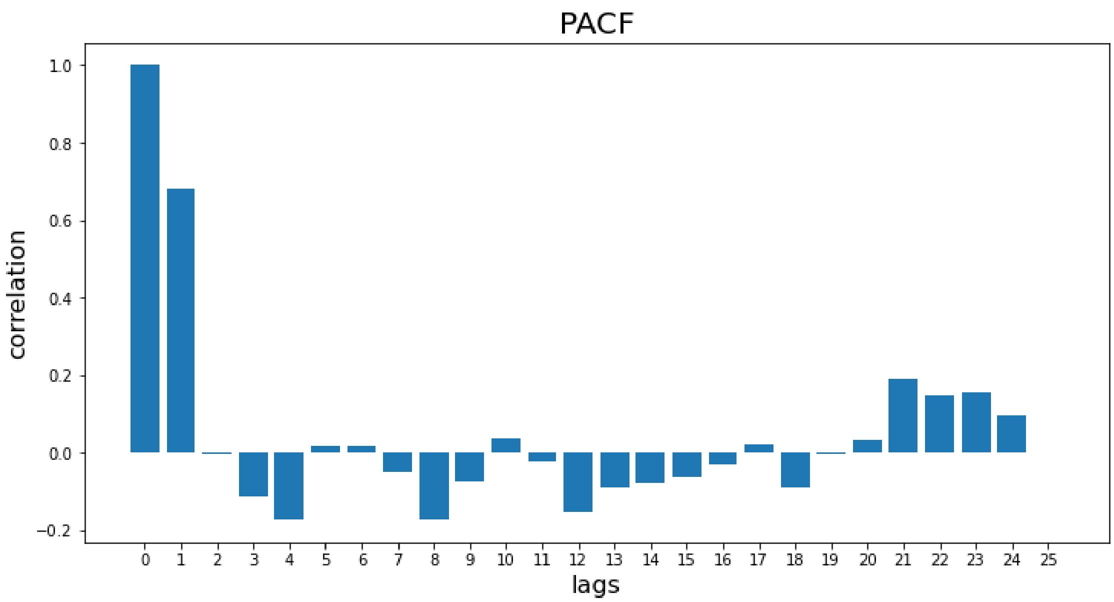
[21]:
```
acf_vals = acf(time_series_of_segment_df['time'])

#-----------------------------------------------------------------
# GRAPH
#-----------------------------------------------------------------

# size:
plt.figure(figsize=(12,6))

# title:
plt.title('ACF', fontsize=20)

# x axis:
# x values
x = []
for index in range(len(acf_vals)):
    x.append(index)
# x ticks
```

```
my_xticks = x
plt.xticks(x, x, rotation='horizontal')
# # x label
plt.xlabel("lags", fontsize=16)

# y axis:
# y values
y = acf_vals
# y label
plt.ylabel("correlation", fontsize=16)

# create plot
plt.bar(x, y)

# show plot
plt.show()

#-----------------------------------------------------------------
```



0.1.2 Partial Auto Correlation Function (PACF)

[22]:
```
pacf_vals = pacf(time_series_of_segment_df['time'])

#-----------------------------------------------------------------
# GRAPH
#-----------------------------------------------------------------

# size:
plt.figure(figsize=(12,6))

# title:
plt.title('PACF', fontsize=20)

# x axis:
# x values
x = []
for index in range(len(pacf_vals)):
    x.append(index)
# x ticks
my_xticks = x
plt.xticks(x, x, rotation='horizontal')
# # x label
plt.xlabel("lags", fontsize=16)

# y axis:
# y values
y = pacf_vals
# y label
plt.ylabel("correlation", fontsize=16)

# create plot
plt.bar(x, y)

# show plot
plt.show()

#-----------------------------------------------------------------
```



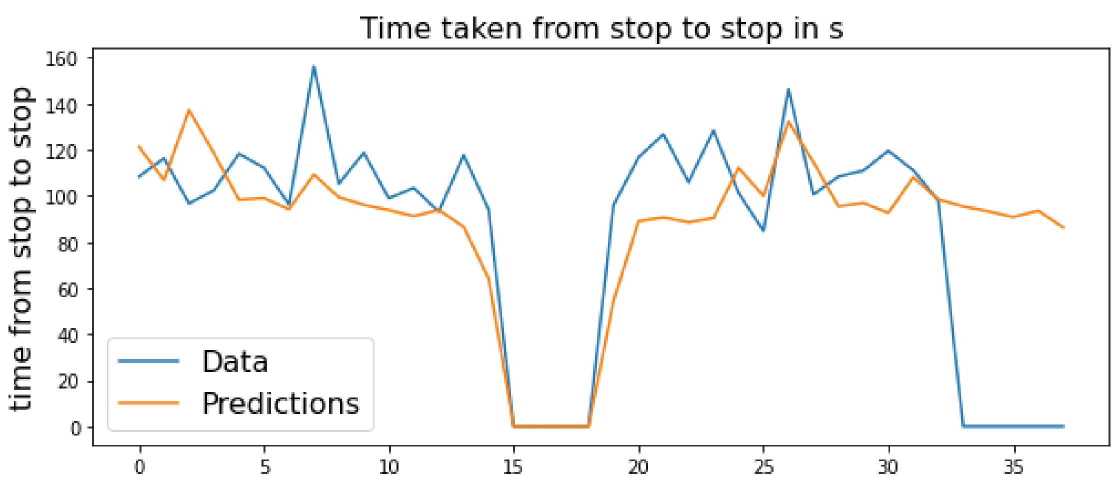
[ ]:
```
# Percentage of data to train with:
training_percentage = 0.9

# Total number of elements:
total_number_of_elements = len(time_series_of_segment_df['time'])

# Total number of elements to train with:
number_of_elements_to_train =␣
 ↪round(training_percentage*(total_number_of_elements))

# Training data
train_data = list(time_series_of_segment_df['time'])[:
 ↪-(total_number_of_elements-number_of_elements_to_train)]

# Testing data
test_data = list(time_series_of_segment_df['time'])[number_of_elements_to_train:
 ↪]
```

[59]:
```
non_seasonal_order = (1, 0, 1)
seasonal_order = (1, 0, 1, 24)

model_fit = SARIMAX(train_data, order=non_seasonal_order,␣
 ↪seasonal_order=seasonal_order).fit()

pred_start_date = len(train_data)
pred_end_date = len(test_data) + len(train_data)-1
```

```
predictions = model_fit.predict(start=pred_start_date, end=pred_end_date)
```

[62]:
```
plt.figure(figsize=(10,4))

plt.plot(test_data)
plt.plot(predictions)

plt.legend(('Data', 'Predictions'), fontsize=16)

plt.title('Time taken from stop to stop in s', fontsize=16)
plt.ylabel('time from stop to stop', fontsize=16)
```

[62]:
```
Text(0, 0.5, 'time from stop to stop')
```



Now, let's apply this aproach to make predictions for all the segments of a line for a specific numbers of days:

[47]:
```
# INPUT 1: line number
line = 1
```

Check if the available tables the same number of segments for the given line:

[48]:
```
def all_days_have_same_segments(line):

    count_segments_per_day = []

    for day_data in average_times_per_day:

        day_data = day_data[day_data['LineId']==line]

```

```
        from_stop_list = day_data['FromStop'].tolist()
        to_stop_list = day_data['ToStop'].tolist()

        segments = []

        for i in range(len(from_stop_list)):
            segment = []
            segment.append(from_stop_list[i])
            segment.append(to_stop_list[i])

            segments.append(segment)

        count_segments_per_day.append(len(segments))

    if len(set(count_segments_per_day)) == 1:
        print('The data is complete, all the tables have the same segments for␣
 ↪line: '+str(line))
    else:
        print('The data is incomplete,some tables have different quantity of␣
 ↪segments for line: '+str(line))

    # return count_segments_per_day

all_days_have_same_segments(line)
```

```
The data is complete, all the tables have the same segments for line: 1
```

Now get a list with all the segments:

[49]:
```
def list_of_segments(line):
    day_data =␣
 ↪average_times_per_day[0][average_times_per_day[0]['LineId']==line]

    from_stop_list = day_data['FromStop'].tolist()
    to_stop_list = day_data['ToStop'].tolist()

    segments = []

    for i in range(len(from_stop_list)):
        segment = []
        segment.append(from_stop_list[i])
        segment.append(to_stop_list[i])

        segments.append(segment)

    return segments
```

```
line_segments = list_of_segments(line)
```

Now make a dictionary whose key is the segments and value is the corresponding time series for the given line:

[50]:
```
# Dictionary with all the time series:
all_time_series_of_line = dict()

# Populate the dictionary with a time series list for each line's segment:
for segment in line_segments:
    segment_time_series = []
    for day in average_times_per_day:
        for hour in range(24):
            segment_time_series.append(day[(day['FromStop']==segment[0]) &␣
 ↪(day['ToStop']==segment[1])][str(hour)].values[0])
    all_time_series_of_line[str(segment[0])+'_'+str(segment[1])] =␣
 ↪segment_time_series

# Create the labels for every element of the time series:
date_and_time = []
for day in days:
    for hour in range(24):
        date_and_time.append(day + '_' + str(hour))
```

Let's now create an SARIMA model for every segment of the dictionary using the above defined orders:

[74]:
```
# Number of available days:
total_days = 16

# Number of days train with:
training_days = 10

# Dictionary with all the predictions:
all_predictions_of_line = dict()

# Populate the predictions table:
for segment in all_time_series_of_line:
    # Get one time series
    current_time_series = all_time_series_of_line[segment]
    # Get the total number of elements:
    total_number_of_elements = len(current_time_series)
    # Get the total number of elements to train with:
    number_of_elements_to_train = round(training_days*len(current_time_series)/
 ↪total_days)
    # Training data:
```

```
    train_data = current_time_series[:
↪-(len(current_time_series)-number_of_elements_to_train)]
    # Testing data:
    test_data = current_time_series[number_of_elements_to_train:]
    # Fit a model for this time series:
    model_fit = SARIMAX(train_data, order=non_seasonal_order,␣
↪seasonal_order=seasonal_order).fit()
    # Define the start of prediction
    pred_start_date = len(train_data)
    # Define the end of prediction
    pred_end_date = len(test_data) + len(train_data)-1
    # Get the predictions
    predictions = model_fit.predict(start=pred_start_date, end=pred_end_date)
    # Add the predictions to the dictionary
    all_predictions_of_line[segment] = predictions
```

```
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\base\model.py:604: ConvergenceWarning: Maximum Likelihood
optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\base\model.py:604: ConvergenceWarning: Maximum Likelihood
optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\base\model.py:604: ConvergenceWarning: Maximum Likelihood
optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
```

```
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:997: UserWarning: Non-stationary
starting seasonal autoregressive Using zeros as starting parameters.
  warn('Non-stationary starting seasonal autoregressive'
C:\Users\PREDATOR\.conda\envs\data_mining\lib\site-
packages\statsmodels\tsa\statespace\sarimax.py:966: UserWarning: Non-stationary
starting autoregressive parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
```

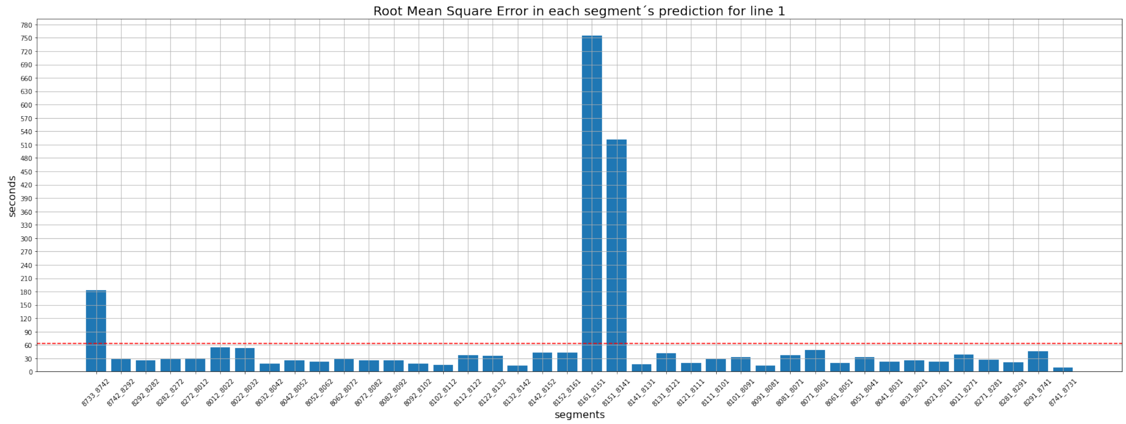
Let's measure the accuracy of the predictions compared to the actual values:

[52]:
```
# Create a dictionary with the testing data for all segments of line:
all_testing_data_of_line = dict()

# Populate the dictionary of testing data:
for segment in all_time_series_of_line:
    # Get one time series
    current_time_series = all_time_series_of_line[segment]
    # Get the portion of data for testing
    all_testing_data_of_line[segment] = current_time_series[round(training_days*len(current_time_series)/total_days):]

# List of Root Mean Square Error for every segment:
all_rmse = []
for segment in all_time_series_of_line:
    all_rmse.append(math.sqrt(np.square(np.subtract(all_testing_data_of_line[segment], all_predictions_of_line[segment])).mean()))

# Average Root Mean Square Error:
average_rmse = sum(all_rmse) / len(all_rmse)
average_rmse
```

[52]: 63.54419158749219

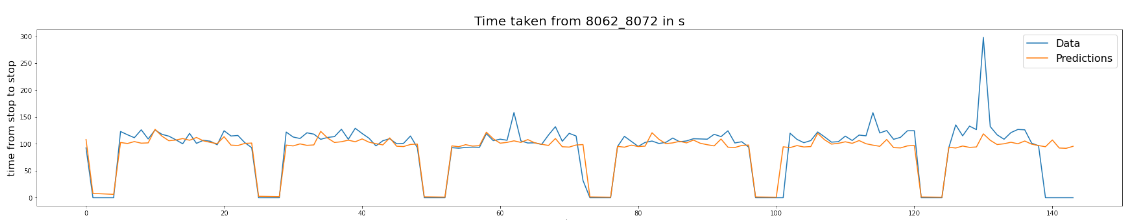
Create a graph with the results to check the Root Mean Square Error for each segment:

```
[71]: #-----------------------------------------------------------------
      # GRAPH
      #-----------------------------------------------------------------

      # size:
      plt.figure(figsize=(30,10))

      # title:
      plt.title('Root Mean Square Error in each segment´s prediction for line ' +␣
       ↪str(line), fontsize=20)

      # x axis:
      # x values
      x=[]
      for index in range(len(all_testing_data_of_line)):
          x.append(index)
      # x ticks
      my_xticks = []
      for segment in all_testing_data_of_line:
          my_xticks.append(segment)
      plt.xticks(x, my_xticks, rotation='45')
      # # x label
      plt.xlabel("segments", fontsize=16)

      # y axis:
      # y values
      y = all_rmse
      # y ticks
      plt.yticks(np.arange(0, max(all_rmse)+30, 30))
      # y label
      plt.ylabel("seconds", fontsize=16)

      # mean
      plt.axhline(sum(all_rmse)/len(all_rmse), color='r', alpha=1, linestyle='--')

      # create plot
      plt.bar(x, y)
      # add grids to plot
      plt.grid()
      # show plot
      plt.show()

      #-----------------------------------------------------------------
```




```
[85]: # Select an index to select a segment in the dictionaries
dict_index = 10

plt.figure(figsize=(30,5))

plt.plot(list(all_testing_data_of_line.values())[dict_index])
plt.plot(list(all_predictions_of_line.values())[dict_index])

plt.legend(('Data', 'Predictions'), fontsize=16)

plt.title('Time taken from '+list(all_testing_data_of_line.
 ↪keys())[dict_index]+' in s', fontsize=20)
plt.ylabel('time from stop to stop', fontsize=16)
plt.xlabel('hours', fontsize=16)
```

```
[85]: Text(0.5, 0, 'hours')
```



Now, let's create the daily tables with the predictions:

```
[464]: # Prediction tables:
prediction_tables = []
for i in range(total_days-training_days):
```

```
    prediction_tables.append(pd.DataFrame(columns = ['LineId', 'FromStop',␣
↪'ToStop'] + [str(i) for i in range(24)]))

# Populate the tables:
for table in range(len(prediction_tables)):
    for segment in all_predictions_of_line:
        row = pd.DataFrame([[line, segment.split("_", 1)[0], segment.split("_",␣
↪1)[1]]], columns=['LineId', 'FromStop', 'ToStop'])
        prediction_tables[table] = prediction_tables[table].append(row,␣
↪ignore_index=True)
        for time in range(24):
            prediction_tables[table].
↪loc[(prediction_tables[table]['FromStop']==segment.split("_", 1)[0]) &␣
↪(prediction_tables[table]['ToStop']==segment.split("_", 1)[1]) &␣
↪(prediction_tables[table]['LineId']==line), str(time)]=␣
↪round(all_predictions_of_line[segment][time+24*table],2)
```

0.2 Implementing the Model

Now that we have used SARIMA to creat a lookup table, we can use the data to start predicting arrival times given some starting point.

Let's try it out by getting the arrival time of a vehicle from line 1 to every stop given some start time. We will need to build a couple extra functions to make this process repeatable for any combination of line, date and time.

[465]:
```
# INPUT 1: line number
line = 1

# INPUT 2: start time
start_time = '05:59:23'
start_time = pd.to_datetime(start_time, format='%H:%M:%S')

# INPUT 3: date (2021-09-16 to 21-09-21)
date = '2021-09-16'
```

So, given the line we defined before, let's create a function to get this information:

get_line_segment_times Given a line number, it gives the average time to get from stop to stop every hour.

Input: line number Output: average time from stop to stop

[419]:
```
#get_line_segment_times

def get_line_segment_times(line, date, prediction_tables):
```

```
    # Get the predicted average times for the selected date
    if (date =='2021-09-16' ):
        all_lines_segment_times = prediction_tables[0]
    elif (date =='2021-09-17' ):
        all_lines_segment_times = prediction_tables[1]
    elif (date =='2021-09-18' ):
        all_lines_segment_times = prediction_tables[2]
    elif (date =='2021-09-19' ):
        all_lines_segment_times = prediction_tables[3]
    elif (date =='2021-09-20' ):
        all_lines_segment_times = prediction_tables[4]
    elif (date =='2021-09-20' ):
        all_lines_segment_times = prediction_tables[5]

    # Convert the times in seconds to format HH:MM:SS
    for i in range(24):
        all_lines_segment_times[str(i)]=pd.
↪to_datetime(all_lines_segment_times[str(i)], unit='s').dt.strftime("%H:%M:
↪%S")

    # Rename the headers for the time ranges
    all_lines_segment_times.rename(columns={
                                                "LineId": "lineId",
                                                "0": "00_01",
                                                "1": "01_02",
                                                "2": "02_03",
                                                "3": "03_04",
                                                "4": "04_05",
                                                "5": "05_06",
                                                "6": "06_07",
                                                "7": "07_08",
                                                "8": "08_09",
                                                "9": "09_10",
                                                "10": "10_11",
                                                "11": "11_12",
                                                "12": "12_13",
                                                "13": "13_14",
                                                "14": "14_15",
                                                "15": "15_16",
                                                "16": "16_17",
                                                "17": "17_18",
                                                "18": "18_19",
                                                "19": "19_20",
                                                "20": "20_21",
                                                "21": "21_22",
                                                "22": "22_23",
                                                "23": "23_24"
```

```
                                                    }, inplace = True)

    # Take the segment times for the selected line
    line_segment_times =␣
↪all_lines_segment_times[all_lines_segment_times['lineId']==line]

    return line_segment_times
```

[420]:
```
line_segment_times = get_line_segment_times(line, date, prediction_tables)
line_segment_times
```

[420]:

	lineId	FromStop	ToStop	00_01	01_02	02_03	03_04	04_05	\
0	1	8733	8742	00:00:52	00:00:00	00:00:00	00:00:00	00:00:00	
1	1	8742	8292	00:00:57	00:00:00	00:00:00	00:00:00	00:00:00	
2	1	8292	8282	00:00:45	00:00:00	00:00:00	00:00:00	00:00:00	
3	1	8282	8272	00:00:58	00:00:00	00:00:00	00:00:00	00:00:00	
4	1	8272	8012	00:01:03	00:00:00	00:00:00	00:00:00	00:00:00	
5	1	8012	8022	00:00:58	00:00:00	00:00:00	00:00:00	00:00:00	
6	1	8022	8032	00:00:53	00:00:00	00:00:00	00:00:00	00:00:00	
7	1	8032	8042	00:00:54	00:00:00	00:00:00	00:00:00	00:00:00	
8	1	8042	8052	00:00:49	00:00:00	00:00:00	00:00:00	00:00:00	
9	1	8052	8062	00:01:10	00:00:00	00:00:00	00:00:00	00:00:00	
10	1	8062	8072	00:01:46	00:00:00	00:00:00	00:00:00	00:00:00	
11	1	8072	8082	00:01:08	00:00:00	00:00:00	00:00:00	00:00:00	
12	1	8082	8092	00:01:03	00:00:00	00:00:00	00:00:00	00:00:00	
13	1	8092	8102	00:00:59	00:00:00	00:00:00	00:00:00	00:00:00	
14	1	8102	8112	00:01:13	00:00:00	00:00:00	00:00:00	00:00:00	
15	1	8112	8122	00:02:00	00:00:00	00:00:00	00:00:00	00:00:00	
16	1	8122	8132	00:01:44	00:00:00	00:00:00	00:00:00	00:00:00	
17	1	8132	8142	00:01:02	00:00:00	00:00:00	00:00:00	00:00:00	
18	1	8142	8152	00:01:29	00:00:00	00:00:00	00:00:00	00:00:00	
19	1	8152	8161	00:00:49	00:00:00	00:00:00	00:00:00	00:00:00	
20	1	8161	8151	00:09:37	00:00:00	00:00:00	00:00:00	00:11:35	
21	1	8151	8141	00:00:24	00:00:00	00:00:00	00:05:19	00:06:38	
22	1	8141	8131	00:00:38	00:00:00	00:00:00	00:00:00	00:00:00	
23	1	8131	8121	00:01:37	00:00:00	00:00:00	00:00:00	00:00:00	
24	1	8121	8111	00:00:31	00:00:00	00:00:00	00:00:00	00:00:00	
25	1	8111	8101	00:01:35	00:00:00	00:00:00	00:00:00	00:00:00	
26	1	8101	8091	00:01:00	00:00:00	00:00:00	00:00:00	00:00:00	
27	1	8091	8081	00:00:33	00:00:00	00:00:00	00:00:00	00:00:00	
28	1	8081	8071	00:02:02	00:00:00	00:00:00	00:00:00	00:00:00	
29	1	8071	8061	00:01:48	00:00:00	00:00:00	00:00:00	00:01:25	
30	1	8061	8051	00:01:11	00:00:00	00:00:00	00:00:00	00:00:00	
31	1	8051	8041	00:00:52	00:00:00	00:00:00	00:00:00	00:00:00	
32	1	8041	8031	00:01:06	00:00:00	00:00:00	00:00:00	00:00:00	
33	1	8031	8021	00:00:56	00:00:00	00:00:00	00:00:00	00:00:00	
34	1	8021	8011	00:00:40	00:00:00	00:00:00	00:00:00	00:00:00	

35	1	8011	8271	00:02:43	00:00:00	00:00:00	00:00:00	00:00:00
36	1	8271	8281	00:01:05	00:00:00	00:00:00	00:00:00	00:00:00
37	1	8281	8291	00:00:22	00:00:00	00:00:00	00:00:00	00:00:00
38	1	8291	8741	00:01:13	00:00:00	00:00:00	00:00:00	00:00:00
39	1	8741	8731	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

	05_06	06_07	…	14_15	15_16	16_17	17_18	18_19	\
0	00:01:15	00:01:17	…	00:01:21	00:01:00	00:00:27	00:01:20	00:01:01	
1	00:01:03	00:01:29	…	00:01:44	00:01:40	00:01:40	00:02:06	00:01:49	
2	00:00:37	00:00:55	…	00:01:04	00:01:07	00:01:06	00:01:10	00:01:05	
3	00:00:55	00:01:34	…	00:01:37	00:01:40	00:01:36	00:01:55	00:01:39	
4	00:00:41	00:01:10	…	00:01:04	00:01:06	00:01:11	00:01:09	00:01:10	
5	00:00:50	00:01:07	…	00:01:17	00:01:18	00:01:19	00:01:17	00:01:14	
6	00:01:02	00:01:06	…	00:01:05	00:01:04	00:01:05	00:01:09	00:01:05	
7	00:00:48	00:01:02	…	00:01:17	00:01:11	00:01:05	00:01:07	00:01:12	
8	00:00:39	00:00:47	…	00:00:50	00:00:55	00:00:56	00:00:57	00:00:55	
9	00:00:52	00:01:13	…	00:01:35	00:01:19	00:01:24	00:01:21	00:01:39	
10	00:01:44	00:01:41	…	00:01:51	00:01:47	00:01:55	00:01:47	00:01:45	
11	00:02:27	00:01:36	…	00:01:41	00:01:16	00:01:16	00:01:20	00:01:27	
12	00:01:36	00:01:08	…	00:01:17	00:01:20	00:01:19	00:01:13	00:01:14	
13	00:00:59	00:00:58	…	00:01:02	00:01:04	00:00:58	00:01:00	00:01:05	
14	00:01:13	00:01:13	…	00:01:13	00:01:14	00:01:11	00:01:17	00:01:09	
15	00:00:38	00:00:39	…	00:01:02	00:00:44	00:00:46	00:00:46	00:00:44	
16	00:02:03	00:02:55	…	00:01:54	00:01:58	00:01:59	00:01:55	00:01:55	
17	00:01:04	00:01:01	…	00:01:04	00:00:58	00:01:07	00:01:02	00:01:04	
18	00:01:29	00:01:47	…	00:01:47	00:01:50	00:01:44	00:02:08	00:01:48	
19	00:00:42	00:00:52	…	00:01:04	00:00:54	00:01:20	00:00:42	00:00:46	
20	00:04:07	00:01:49	…	00:02:22	00:01:29	00:00:53	00:01:19	00:01:28	
21	00:04:56	00:00:34	…	00:00:39	00:00:37	00:00:37	00:00:36	00:00:35	
22	00:00:43	00:00:42	…	00:00:48	00:00:47	00:00:50	00:00:47	00:00:49	
23	00:01:43	00:02:10	…	00:02:03	00:01:50	00:01:50	00:01:47	00:01:45	
24	00:00:43	00:00:47	…	00:00:54	00:00:53	00:00:55	00:01:02	00:00:50	
25	00:01:50	00:01:47	…	00:02:09	00:01:50	00:01:59	00:01:49	00:01:53	
26	00:01:00	00:00:57	…	00:01:10	00:01:03	00:01:03	00:01:01	00:01:01	
27	00:00:45	00:00:47	…	00:00:50	00:00:51	00:00:53	00:00:50	00:00:50	
28	00:01:29	00:01:46	…	00:01:44	00:01:52	00:01:53	00:02:06	00:02:02	
29	00:01:39	00:01:46	…	00:02:22	00:02:01	00:02:11	00:01:57	00:01:57	
30	00:01:05	00:01:07	…	00:01:07	00:01:14	00:01:12	00:01:13	00:01:12	
31	00:01:15	00:01:06	…	00:01:05	00:01:08	00:01:08	00:01:13	00:01:19	
32	00:01:06	00:01:15	…	00:01:19	00:01:21	00:01:35	00:01:18	00:01:20	
33	00:00:52	00:00:51	…	00:00:59	00:00:57	00:00:58	00:01:00	00:00:59	
34	00:00:46	00:00:58	…	00:00:50	00:00:58	00:01:02	00:01:01	00:00:55	
35	00:01:34	00:01:32	…	00:01:58	00:01:46	00:01:48	00:02:02	00:02:03	
36	00:01:17	00:01:11	…	00:01:25	00:01:32	00:01:41	00:01:29	00:01:32	
37	00:00:48	00:00:51	…	00:00:49	00:00:56	00:00:54	00:00:52	00:00:55	
38	00:02:05	00:01:56	…	00:02:01	00:02:14	00:02:11	00:02:18	00:02:04	
39	00:00:00	00:00:00	…	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	

	19_20	20_21	21_22	22_23	23_24
0	00:02:04	00:02:55	00:00:00	00:02:03	00:07:59
1	00:01:38	00:01:37	00:01:40	00:01:34	00:01:24
2	00:01:05	00:01:01	00:00:59	00:01:02	00:00:56
3	00:01:39	00:01:39	00:01:35	00:01:35	00:01:29
4	00:01:04	00:01:04	00:01:06	00:01:06	00:01:08
5	00:01:10	00:01:14	00:01:14	00:01:13	00:01:08
6	00:01:11	00:01:07	00:01:01	00:01:04	00:01:08
7	00:01:08	00:01:04	00:01:04	00:01:03	00:01:00
8	00:00:52	00:00:56	00:00:52	00:00:49	00:00:56
9	00:01:19	00:01:38	00:01:18	00:01:17	00:01:15
10	00:01:43	00:01:59	00:01:40	00:01:41	00:01:44
11	00:01:20	00:01:20	00:01:18	00:01:29	00:01:12
12	00:01:12	00:01:28	00:01:28	00:01:12	00:01:10
13	00:00:59	00:01:03	00:01:01	00:01:02	00:01:05
14	00:01:16	00:01:10	00:01:14	00:01:11	00:01:12
15	00:00:44	00:00:47	00:00:40	00:00:43	00:00:46
16	00:01:55	00:01:56	00:02:02	00:01:57	00:01:47
17	00:01:01	00:00:58	00:01:04	00:01:02	00:01:11
18	00:01:49	00:01:52	00:01:46	00:01:47	00:01:38
19	00:00:58	00:01:41	00:02:01	00:01:04	00:00:50
20	00:00:59	00:01:06	00:00:09	00:00:27	23:59:42
21	00:00:37	00:00:38	00:00:35	00:00:36	00:00:33
22	00:00:47	00:00:46	00:00:52	00:00:44	00:00:38
23	00:01:43	00:01:42	00:01:50	00:01:44	00:01:43
24	00:01:04	00:01:15	00:00:49	00:00:51	00:00:48
25	00:01:50	00:01:47	00:01:49	00:01:46	00:01:51
26	00:01:02	00:01:01	00:01:02	00:01:04	00:00:54
27	00:00:49	00:00:49	00:00:48	00:00:46	00:00:44
28	00:01:45	00:01:33	00:01:34	00:01:34	00:01:46
29	00:01:56	00:01:52	00:01:51	00:01:52	00:01:52
30	00:01:09	00:01:12	00:01:11	00:01:08	00:01:09
31	00:01:07	00:01:08	00:01:05	00:01:04	00:01:00
32	00:01:17	00:01:11	00:01:11	00:01:15	00:01:15
33	00:00:52	00:01:00	00:01:05	00:00:56	00:00:57
34	00:00:52	00:00:46	00:00:47	00:00:47	00:00:42
35	00:01:45	00:01:43	00:01:41	00:01:41	00:01:44
36	00:01:36	00:02:07	00:01:24	00:01:26	00:00:51
37	00:00:50	00:00:50	00:00:46	00:00:47	00:00:48
38	00:01:57	00:01:54	00:01:57	00:01:55	00:01:44
39	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

[40 rows x 27 columns]

We must consider here that this table includes the average time from stop to stop in both directions. So, the output of our predicting model should include two tables, each one containing the arrival times for one direction. The structure of the output tables should be:

[429]:
```
output_structure= pd.DataFrame(columns=['lineId', 'departure_stop_id',␣
↪'departure_time', 'arrival_stop_Id', 'arrival_time'])
output_structure
```

[429]:
```
Empty DataFrame
Columns: [lineId, departure_stop_id, departure_time, arrival_stop_Id,
arrival_time]
Index: []
```

Let's start making our model:

From the inputs, we have *start time*. This will be useful to know with which time range(s) from **line_segment_times** should we work with. To do this, we will create a function that takes the *start time* and outputs the corresponding time range's name from **line_segment_times*:

get_time_range Function to get the time range (two hours ranges) in which a given time is in. *Input: time Output: time range code*

[422]:
```
# get_time_range: get

def get_time_range(start_time):
    if (pd.to_datetime('00:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('01:00:00', format='%H:%M:%S')):
        return '00_01'
    elif (pd.to_datetime('01:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('02:00:00', format='%H:%M:%S')):
        return '01_02'
    elif (pd.to_datetime('02:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('03:00:00', format='%H:%M:%S')):
        return '02_03'
    elif (pd.to_datetime('03:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('04:00:00', format='%H:%M:%S')):
        return '03_04'
    elif (pd.to_datetime('04:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('05:00:00', format='%H:%M:%S')):
        return '04_05'
    elif (pd.to_datetime('05:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('06:00:00', format='%H:%M:%S')):
        return '05_06'
    elif (pd.to_datetime('06:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('07:00:00', format='%H:%M:%S')):
        return '06_07'
    elif (pd.to_datetime('07:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('08:00:00', format='%H:%M:%S')):
        return '07_08'
```

```
    elif (pd.to_datetime('08:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('09:00:00', format='%H:%M:%S')):
        return '08_09'
    elif (pd.to_datetime('09:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('10:00:00', format='%H:%M:%S')):
        return '09_10'
    elif (pd.to_datetime('10:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('11:00:00', format='%H:%M:%S')):
        return '10_11'
    elif (pd.to_datetime('11:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('12:00:00', format='%H:%M:%S')):
        return '11_12'
    elif (pd.to_datetime('12:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('13:00:00', format='%H:%M:%S')):
        return '12_13'
    elif (pd.to_datetime('13:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('14:00:00', format='%H:%M:%S')):
        return '13_14'
    elif (pd.to_datetime('14:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('15:00:00', format='%H:%M:%S')):
        return '14_15'
    elif (pd.to_datetime('15:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('16:00:00', format='%H:%M:%S')):
        return '15_16'
    elif (pd.to_datetime('16:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('17:00:00', format='%H:%M:%S')):
        return '16_17'
    elif (pd.to_datetime('17:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('18:00:00', format='%H:%M:%S')):
        return '17_18'
    elif (pd.to_datetime('18:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('19:00:00', format='%H:%M:%S')):
        return '18_19'
    elif (pd.to_datetime('19:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('20:00:00', format='%H:%M:%S')):
        return '19_20'
    elif (pd.to_datetime('20:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('21:00:00', format='%H:%M:%S')):
        return '20_21'
    elif (pd.to_datetime('21:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('22:00:00', format='%H:%M:%S')):
        return '21_22'
    elif (pd.to_datetime('22:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('23:00:00', format='%H:%M:%S')):
        return '22_23'
```

```
    elif (pd.to_datetime('23:00:00', format='%H:%M:%S')<= start_time < pd.
↪to_datetime('24:00:00', format='%H:%M:%S')):
        return '23_24'
```

Let´s get the time range in which the starting time is in:

[444]:
```
time_range = get_time_range(start_time)
time_range
```

[444]: '05_06'

Now that we got the time range of the vehicle's starting time, let´s calculate the arrival time to the very next stop. For this we have to consider the two possible *departure_stop_id*'s. From assignment 1, we can get the series of stops that a line's sequentially vehicle goes trough. So, let's make a function that gives the sequence of stops given a line:

get_sequence_stops_by_line Given a line number, it gives the two sequences of stops, each for one direction

Input: line number Output: list with the two sequences of stops (direction 1 and direction 2)

[445]:
```
# get_sequence_stops_by_line

def get_sequence_stops_by_line(line):

    # Get a table indicating all the stops sequences for all the lines
    stops_sequences_by_lines = pd.read_csv(r'./processed_data/line_stops.csv')

    # Convert lineId to integer
    stops_sequences_by_lines['lineId'] = stops_sequences_by_lines['lineId'].
↪map(lambda x: int(str(x)[:-1]))

    # Get the number of stops for the selected line travelling in direction 1
    number_of_stops_by_line_dir1 =␣
↪stops_sequences_by_lines[(stops_sequences_by_lines['lineId']==line ) &␣
↪(stops_sequences_by_lines['direction'] == 1)]['order'].max()

    # Get the number of stops for the selected line travelling in direction 2
    number_of_stops_by_line_dir2 =␣
↪stops_sequences_by_lines[(stops_sequences_by_lines['lineId']==line ) &␣
↪(stops_sequences_by_lines['direction'] == 2)]['order'].max()


    # Make two list with the stops sequence in each direction
    stops_sequences_by_line = []
    stops_sequence_by_line_dir1 = []
```

```
    stops_sequence_by_line_dir2 = []

    # Direction 1
    for i in range(number_of_stops_by_line_dir1):
        stops_sequence_by_line_dir1.
↪append(stops_sequences_by_lines[(stops_sequences_by_lines['lineId']==line) &␣
↪(stops_sequences_by_lines['direction'] == 1) &␣
↪(stops_sequences_by_lines['order'] ==( i+1))]['stop_id_int'].values[0])
    stops_sequences_by_line.append(stops_sequence_by_line_dir1)

    # Direction 2
    for i in range(number_of_stops_by_line_dir2):
        stops_sequence_by_line_dir2.
↪append(stops_sequences_by_lines[(stops_sequences_by_lines['lineId']==line) &␣
↪(stops_sequences_by_lines['direction'] == 2) &␣
↪(stops_sequences_by_lines['order'] ==( i+1))]['stop_id_int'].values[0])
    stops_sequences_by_line.append(stops_sequence_by_line_dir2)

    return stops_sequences_by_line
```

Let's get the two sequences:

[447]:
```
stops_sequence_by_line = get_sequence_stops_by_line(line)
```

We can say that our possible *departure_stop_id*'s are:

[448]:
```
departure_stop_id_dir1 = stops_sequence_by_line[0][0]
departure_stop_id_dir2 = stops_sequence_by_line[1][0]

print('The departure stop in direction 1 is: '+str(departure_stop_id_dir1))
print('The departure stop in direction 2 is: '+str(departure_stop_id_dir2))
```

```
The departure stop in direction 1 is: 8733
The departure stop in direction 2 is: 8161
```

In order to get what is the next stop given a departure stop and a current stop, we will make a function:

get_next_stop Get the next stop given a departure stop (to know the direction) and the current stop

Input: departure stop id, current stop id, stop ids sequence Output: next stop id

[449]:
```
def get_next_stop(departure_stop_id, current_stop_id, stop_ids_sequence):
    if (departure_stop_id == stop_ids_sequence[0][0]):
        next_stop = stop_ids_sequence[0][stop_ids_sequence[0].
↪index(current_stop_id)+1]
    elif (departure_stop_id == stop_ids_sequence[1][0]):
```

```
        next_stop = stop_ids_sequence[1][stop_ids_sequence[1].
 ↪index(current_stop_id)+1]
    return next_stop
```

[454]:
```
line_segment_times.head()
```

[454]:
```
   lineId FromStop ToStop     00_01     01_02     02_03     03_04     04_05  \
0       1     8733   8742  00:00:52  00:00:00  00:00:00  00:00:00  00:00:00
1       1     8742   8292  00:00:57  00:00:00  00:00:00  00:00:00  00:00:00
2       1     8292   8282  00:00:45  00:00:00  00:00:00  00:00:00  00:00:00
3       1     8282   8272  00:00:58  00:00:00  00:00:00  00:00:00  00:00:00
4       1     8272   8012  00:01:03  00:00:00  00:00:00  00:00:00  00:00:00

      05_06     06_07  …     14_15     15_16     16_17     17_18     18_19  \
0  00:01:15  00:01:17  …  00:01:21  00:01:00  00:00:27  00:01:20  00:01:01
1  00:01:03  00:01:29  …  00:01:44  00:01:40  00:01:40  00:02:06  00:01:49
2  00:00:37  00:00:55  …  00:01:04  00:01:07  00:01:06  00:01:10  00:01:05
3  00:00:55  00:01:34  …  00:01:37  00:01:40  00:01:36  00:01:55  00:01:39
4  00:00:41  00:01:10  …  00:01:04  00:01:06  00:01:11  00:01:09  00:01:10

      19_20     20_21     21_22     22_23     23_24
0  00:02:04  00:02:55  00:00:00  00:02:03  00:07:59
1  00:01:38  00:01:37  00:01:40  00:01:34  00:01:24
2  00:01:05  00:01:01  00:00:59  00:01:02  00:00:56
3  00:01:39  00:01:39  00:01:35  00:01:35  00:01:29
4  00:01:04  00:01:04  00:01:06  00:01:06  00:01:08

[5 rows x 27 columns]
```

Let's calculate the arrival time to the next stop assuming that the vehicle is leaving from *departure_stop_id_0*

[458]:
```
# Current stop_id
current_stop = departure_stop_id_dir1

# Next stop_id
next_stop = get_next_stop(departure_stop_id_dir1, current_stop,␣
 ↪stops_sequence_by_line )

# Average time that takes to go to the next stop at a given specific time range
avg_time = line_segment_times[(line_segment_times['FromStop'] ==␣
 ↪str(current_stop)) & (line_segment_times['ToStop'] == str(next_stop))␣
 ↪][time_range].values[0]

avg_time
```

[458]:
```
'00:01:15'
```

Let´s now get the time of arrival to that next stop:

[459]:
```
avg_time = pd.to_datetime(avg_time, format='%H:%M:%S')
arrival_time = (start_time - pd.to_datetime('00:00:00', format='%H:%M:%S') +
 ↪avg_time).time().strftime('%H:%M:%S')
arrival_time
```

[459]: '06:00:38'

This process has to be done for every segment (stop to stop) of the vehicle trip for both directions. So, let's make a function that does this.

forecasting_arrival_times Function to get the arrival times of a line's vehicle accross all it's stops for both directions.

Input: line and start time Output: a list with two tables with the arrival (one table for every direction)

[466]:
```
def forecasting_arrival_times_by_line(line, start_time, date,
 ↪prediction_tables):

    # Get the average times per segments (stop to stop)
    line_segment_times = get_line_segment_times(line, date, prediction_tables)

    # Get the stops sequence for the line
    stops_sequence_by_line = get_sequence_stops_by_line(line)

    # Get the stops sequence in direction 1
    stops_sequence_by_line_dir1 = stops_sequence_by_line[0]

    # Get the stops sequence in direction 2
    stops_sequence_by_line_dir2 = stops_sequence_by_line[1]

    # Get the two possible departure stops for the line:
    departure_stop_id_dir1 = stops_sequence_by_line[0][0]
    departure_stop_id_dir2 = stops_sequence_by_line[1][0]

    # Departure time
    departure_time = start_time

    # Populate the table:

    # assuming departure_stop_id_dir1:
    # Data to output
    output_data = []
    for i in range(len(stops_sequence_by_line_dir1)-1):

        # Stops sequence
        stops_sequence = stops_sequence_by_line
```

```
        # Current stop
        current_stop =stops_sequence[0][i]

        # Next stop
        next_stop = get_next_stop(departure_stop_id_dir1, current_stop,␣
↪stops_sequence_by_line )

        # Get the time range of the departure time at current stop
        time_range = get_time_range(departure_time)

        # Average time to get to next stop
        avg_time = pd.
↪to_datetime(line_segment_times[(line_segment_times['FromStop'] ==␣
↪str(current_stop)) & (line_segment_times['ToStop'] ==␣
↪str(next_stop))][time_range].values[0], format='%H:%M:%S')

        # Arrival time
        arrival_time = (departure_time - pd.to_datetime('00:00:00', format='%H:
↪%M:%S') + avg_time).time().strftime('%H:%M:%S')

        # Add the data to the data list
        output_data.append({'lineId': line, 'departure_stop_id' : current_stop,␣
↪'departure_time' : departure_time.strftime('%H:%M:%S'), 'arrival_stop_Id' :␣
↪next_stop, 'arrival_time' :  arrival_time })

        # Update next departure time
        departure_time = pd.to_datetime(arrival_time, format='%H:%M:%S')

        output_table_1 = pd.DataFrame(output_data)

    # Set again departure time to analyze the other direction
    departure_time = start_time

    # assuming departure_stop_id_dir2:
    # Data to output
    output_data = []

    for i in range(len(stops_sequence_by_line_dir2)-1):

        # Stops sequence
        stops_sequence = stops_sequence_by_line

        # Current stop
        current_stop = stops_sequence[1][i]

        # Next stop
```

```
        next_stop = get_next_stop(departure_stop_id_dir2, current_stop,␣
↪stops_sequence_by_line )

        # Get the time range of the departure time at current stop
        time_range = get_time_range(departure_time)

        # Average time to get to next stop
        avg_time = pd.
↪to_datetime(line_segment_times[(line_segment_times['FromStop'] ==␣
↪str(current_stop)) & (line_segment_times['ToStop'] == str(next_stop))␣
↪][time_range].values[0], format='%H:%M:%S')

        # Arrival time
        arrival_time = (departure_time - pd.to_datetime('00:00:00', format='%H:
↪%M:%S') + avg_time).time().strftime('%H:%M:%S')

        # Add the data to the data list
        output_data.append({'lineId': line, 'departure_stop_id' : current_stop,␣
↪'departure_time' : departure_time.strftime('%H:%M:%S'), 'arrival_stop_Id' :␣
↪next_stop, 'arrival_time' :  arrival_time })

        # Update next departure time (here we may add an average delay that␣
↪buses have at stations)
        departure_time = pd.to_datetime(arrival_time, format='%H:%M:%S')

        output_table_2 = pd.DataFrame(output_data)

    output = [output_table_1, output_table_2 ]


    return output
```

Let's use the function with the two inputs (line and start time) we defined at the beginning:

[467]:
```
arrival_times = forecasting_arrival_times_by_line(line, start_time, date,␣
↪prediction_tables)
```

The arrival times for the line travelling in direction 1 would be:

[468]:
```
arrival_times[0]
```

[468]:

	lineId	departure_stop_id	departure_time	arrival_stop_Id	arrival_time
0	1	8733	05:59:23	8742	06:00:38
1	1	8742	06:00:38	8292	06:02:07
2	1	8292	06:02:07	8282	06:03:02
3	1	8282	06:03:02	8272	06:04:36
4	1	8272	06:04:36	8012	06:05:46

5	1	8012	06:05:46	8022	06:06:53
6	1	8022	06:06:53	8032	06:07:59
7	1	8032	06:07:59	8042	06:09:01
8	1	8042	06:09:01	8052	06:09:48
9	1	8052	06:09:48	8062	06:11:01
10	1	8062	06:11:01	8072	06:12:42
11	1	8072	06:12:42	8082	06:14:18
12	1	8082	06:14:18	8092	06:15:26
13	1	8092	06:15:26	8102	06:16:24
14	1	8102	06:16:24	8112	06:17:37
15	1	8112	06:17:37	8122	06:18:16
16	1	8122	06:18:16	8132	06:21:11
17	1	8132	06:21:11	8142	06:22:12
18	1	8142	06:22:12	8152	06:23:59
19	1	8152	06:23:59	8161	06:24:51

The arrival times for the line travelling in direction 2 would be:

[469]:
```
arrival_times[1]
```

[469]:

	lineId	departure_stop_id	departure_time	arrival_stop_Id	arrival_time
0	1	8161	05:59:23	8151	06:03:30
1	1	8151	06:03:30	8141	06:04:04
2	1	8141	06:04:04	8131	06:04:46
3	1	8131	06:04:46	8121	06:06:56
4	1	8121	06:06:56	8111	06:07:43
5	1	8111	06:07:43	8101	06:09:30
6	1	8101	06:09:30	8091	06:10:27
7	1	8091	06:10:27	8081	06:11:14
8	1	8081	06:11:14	8071	06:13:00
9	1	8071	06:13:00	8061	06:14:46
10	1	8061	06:14:46	8051	06:15:53
11	1	8051	06:15:53	8041	06:16:59
12	1	8041	06:16:59	8031	06:18:14
13	1	8031	06:18:14	8021	06:19:05
14	1	8021	06:19:05	8011	06:20:03
15	1	8011	06:20:03	8271	06:21:35
16	1	8271	06:21:35	8281	06:22:46
17	1	8281	06:22:46	8291	06:23:37
18	1	8291	06:23:37	8741	06:25:33
19	1	8741	06:25:33	8731	06:25:33

An additional function can be created to get as a result only one arrival's table (one direction). For that, we would have to specify the departure_stop_id:

forecasting_arrival_times_by_line_and_departure Gives a table with the line's arrival times at every stop given a departure time and departure stop

Input: line, start time, departure stop id Output: Table with arrival times at every stop

[ ]:
```python
def forecasting_arrival_times_by_line_and_departure(line, start_time,␣
↪departure_stop_id):
    arrival_times_both_directions = forecasting_arrival_times_by_line(line,␣
↪start_time)
    # If the departure stop corresponds to the direction 1
    if (arrival_times_both_directions[0]['departure_stop_id'].
↪values[0]==departure_stop_id):
        output =  arrival_times_both_directions[0]

    # If the departure stop corresponds to the direction 2
    elif (arrival_times_both_directions[1]['departure_stop_id'].
↪values[0]==departure_stop_id):
        output =  arrival_times_both_directions[1]

    else:
    # If the departure stops does not correspond to any departure stop from the␣
↪line
        output = print('The departure stop provided does not correspond to any␣
↪of the two possible departures stops from line '+str(line))
    return output
```

[ ]:
```python
forecasting_arrival_times_by_line_and_departure(line, start_time, 8161)
```

[ ]:

	lineId	departure_stop_id	departure_time	arrival_stop_Id	arrival_time
0	1	8161	05:59:23	8151	06:22:01
1	1	8151	06:22:01	8141	06:23:09
2	1	8141	06:23:09	8131	06:24:02
3	1	8131	06:24:02	8121	06:26:12
4	1	8121	06:26:12	8111	06:27:06
5	1	8111	06:27:06	8101	06:28:58
6	1	8101	06:28:58	8091	06:29:57
7	1	8091	06:29:57	8081	06:30:51
8	1	8081	06:30:51	8071	06:32:47
9	1	8071	06:32:47	8061	06:34:43
10	1	8061	06:34:43	8051	06:35:52
11	1	8051	06:35:52	8041	06:37:11
12	1	8041	06:37:11	8031	06:38:31
13	1	8031	06:38:31	8021	06:39:34
14	1	8021	06:39:34	8011	06:40:31
15	1	8011	06:40:31	8271	06:42:21
16	1	8271	06:42:21	8281	06:43:46
17	1	8281	06:43:46	8291	06:44:36
18	1	8291	06:44:36	8741	06:46:51
19	1	8741	06:46:51	8731	06:48:04

0.3 How did we do?

We were able to use our time between stops to predict the arrival time at every station on a line. A SARIMA time series was used to create a lookup table that can be applied for any stop and any line. This allows for fast processing as the computationally expensive operations can all be completed once and reused. This approach would allow this model to easily be applied to a live data stream and updated constantly given some input.

assignment_4

December 20, 2021

1 Assignment 4

1.1 Predicting Passenger Movement

1.2 What are we looking to accomplish?

Here we will predict the mode of transport a client made use of given some GPS data and timestamps. We will do this by comparing the users polyline to the polyline of each vehicle on the STIB system. Anywhere there is a perfect match, we will be able to guess the client's system usage. Wherever there are multiple possibilities we can compare the client's speed of movement to the calculated average vehicle speed for that particular time of day on the candidate lines.

1.3 Convert Line shapes to EPSG:4326

[ ]:
```
import json

import shapefile
from numpy import mean, median
from pyproj import Proj, transform
from tqdm.notebook import tqdm
```

[2]:
```
# already run if you have shapes_lat_long.json
lines = shapefile.Reader('../data/raw/shapefiles/ACTU_LINES')
lambert = Proj('EPSG:31370')
latlong = Proj('EPSG:4326')
shapes = {f'{line.record["LIGNE"]}-{line.record["VARIANTE"]}':␣
↪[transform(lambert, latlong, x, y) for x, y in
                                                                line.shape.
↪points] for line in
          tqdm(lines.shapeRecords())}
with open('../data/raw/shapes_lat_long.json', 'w', encoding='utf8') as␣
↪json_file:
    json.dump(shapes, json_file)
```

```
  0%|          | 0/174 [00:00<?, ?it/s]
```

```
/var/folders/wh/6llm7dq16rvbwjb_yc0kfvvr0000gn/T/ipykernel_25836/102539854.py:5:
DeprecationWarning: This function is deprecated. See:
https://pyproj4.github.io/pyproj/stable/gotchas.html#upgrading-to-pyproj-2-from-
pyproj-1
  shapes = {f'{line.record["LIGNE"]}-{line.record["VARIANTE"]}':
[transform(lambert, latlong, x, y) for x, y in
```

1.4 Create plots for each track and line

[3]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import json
import os
from tqdm.notebook import tqdm
```

[6]:
```python
tracks = pd.read_csv('../data/raw/GPSTracksAssignment4/GPStracks.csv')
with open('../data/raw/shapes_lat_long.json', 'r', encoding='utf8') as␣
↪json_file:
    shapes = json.load(json_file)
for track_id in tqdm(tracks['TrackId'].unique()):
    track = tracks[tracks['TrackId'] == track_id].sort_values(by='time')
    if not os.path.exists(f'../data/line_plots/Track{track_id}'):
        os.mkdir(f'../data/line_plots/Track{track_id}')
    for index, (line_id, line) in tqdm(enumerate(shapes.items())):
        plt.scatter(x=track['lon'], y=track['lat'], label=f'Track {track_id}')
        plt.scatter(x=[lon for _, lon in line], y=[lat for lat, _ in line],␣
↪label=line_id)
        plt.title(line_id)
        plt.savefig(f'../data/line_plots/Track{track_id}/{line_id}.png')
        plt.close()
```

```
  0%|          | 0/9 [00:00<?, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

1.4.1 Human Validation

Having plotted each actual line against the provided GPS tracks, we can make a guess as to what the actual method of transport was.

Track Line:

- Track 1: Other
- Track 3: Bus 50 or Tram 82 or Bus 49
- Track 4: Bus 50
- Track 5: Tram 82 or Tram 97
- Track 6: Tram 8
- Track 7: Tram 7
- Track 8: Other
- Track 10: Tram 8 or Tram 93
- Track 11: Tram 25

1.5 Create CSV of labeled points

[7]:
```
import json
from scripts.helpers import write_csv
```

[8]:
```
# creating the dataset.
```

```
[10]: with open('../data/raw/shapes_lat_long.json', 'r', encoding='utf8') as␣
      ↪json_file:
          shapes = json.load(json_file)
      with write_csv('../data/processed/assignment4/labeled_line_points.csv') as␣
      ↪csv_out:
          csv_out.writerow(['Lat', 'Long', 'LineId'])
          for line_id, points in shapes.items():
              if line_id[:-2] in ['032t']:  # Remove lines that are not active anymore
                  continue
              for lat, long in points:
                  csv_out.writerow([lat, long, line_id[:-2]])
```

1.6 Create Classification Model

Now we will create a classification model to determine which line the GPS tracks belong to.

```
[12]: from sklearn.neighbors import KNeighborsClassifier
      from scripts.helpers import read_csv_list
      import pandas as pd
      from typing import Tuple, List
```

```
[15]: def get_datasets() -> Tuple[List[List[float]], List[str], pd.DataFrame]:
          csv_lines = read_csv_list('../data/processed/assignment4/
      ↪labeled_line_points.csv')
          data_set = [[float(point[0]), float(point[1])] for point in csv_lines[1:]]
          data_labels = [point[-1] for point in csv_lines[1:]]
          tracks = pd.read_csv('../data/raw/GPSTracksAssignment4/GPStracks.csv')
          return data_set, data_labels, tracks
```

1.6.1 Create "Ensemble" with KNN using average probability

```
[16]: data_set, data_labels, tracks = get_datasets()
      print('Training model')
      model = KNeighborsClassifier(n_neighbors=10)
      model.fit(data_set, data_labels)
      print('Model trained')
      for track_id in tracks['TrackId'].unique():
          track_points = tracks[tracks['TrackId'] == track_id][['lat', 'lon']]
          predictions = model.predict_proba(track_points)
          final_predictions = [(sum(predictions[i][x]
                                    for i in range(len(predictions))) * 100 /␣
      ↪len(predictions), model.classes_[x])
                               for x in range(len(predictions[0]))]
```

```
    sorted_predictions = sorted(final_predictions, key=lambda x: x[0],␣
↪reverse=True)

    print(track_id, '->', sorted_predictions[:4])
```

```
Training model
Model trained
1 -> [(28.314606741573037, '046b'), (17.07865168539325, '218b'),
(11.96629213483146, '051t'), (9.438202247191013, '088b')]
3 -> [(33.304347826086946, '082t'), (23.826086956521745, '049b'),
(23.043478260869566, '050b'), (10.782608695652176, '048b')]
4 -> [(32.365145228215795, '050b'), (13.81742738589212, '074b'),
(12.987551867219914, '212b'), (10.248962655601662, '097t')]
5 -> [(33.669724770642205, '212b'), (24.67889908256885, '097t'),
(24.12844036697251, '082t'), (11.559633027522937, '050b')]
6 -> [(40.72202166064981, '008t'), (26.498194945848372, '093t'),
(7.725631768953068, '007t'), (4.007220216606497, '209b')]
7 -> [(41.487179487179475, '007t'), (38.56410256410256, '025t'),
(12.256410256410266, '209b'), (4.153846153846154, '008t')]
8 -> [(14.840764331210197, '007t'), (12.261146496815282, '008t'),
(11.910828025477711, '038b'), (8.980891719745237, '004t')]
10 -> [(42.258064516129025, '008t'), (42.258064516129025, '093t'),
(4.838709677419354, '038b'), (4.354838709677419, '060b')]
11 -> [(38.51674641148325, '025t'), (34.736842105263136, '007t'),
(16.028708133971293, '209b'), (4.019138755980861, '071b')]
```

```
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
```

```
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
```

1.6.2 Create "Ensemble" with KNN using "presence"

[17]:
```
data_set, data_labels, tracks = get_datasets()
print('Training model')
model = KNeighborsClassifier(n_neighbors=15)
model.fit(data_set, data_labels)

track_predictions = {}

print('Model trained')
for track_id in tracks['TrackId'].unique():
    track_points = tracks[tracks['TrackId'] == track_id][['lat', 'lon']]
    predictions = model.predict_proba(track_points)
    converted_predictions = [
        [1 if probability > 0 else 0 for probability in prediction]
        for prediction in predictions
    ]
    final_predictions = [(sum(converted_predictions[i][x]
                              for i in range(len(predictions))) * 100 /␣
 ↪len(predictions), model.classes_[x])
                         for x in range(len(predictions[0]))]
    filtered_predictions = [
        (percentage, line) for percentage, line in final_predictions if␣
 ↪percentage > 75
    ]

    filtered_predictions = sorted(filtered_predictions, key=lambda x: x[0],␣
 ↪reverse=True)
```

```
    if len(filtered_predictions):
        filtered_predictions = [
            (prob, line) for prob, line in filtered_predictions if␣
↪filtered_predictions[0][0] - prob < 5
        ]
    track_predictions[track_id] = filtered_predictions[:4]
    print(track_id, '->', filtered_predictions[:4])
```

```
Training model
Model trained
1 -> []
3 -> [(97.3913043478261, '049b'), (96.52173913043478, '050b'),
(95.65217391304348, '082t')]
4 -> [(100.0, '050b')]
5 -> [(98.1651376146789, '097t'), (97.70642201834862, '082t'),
(96.3302752293578, '212b')]
6 -> [(96.75090252707581, '008t')]
7 -> [(100.0, '007t')]
8 -> []
10 -> [(93.54838709677419, '008t'), (93.54838709677419, '093t')]
11 -> [(98.08612440191388, '025t')]
```

```
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
```

```
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/sklearn/base.py:438: UserWarning: X has feature names, but
KNeighborsClassifier was fitted without feature names
  warnings.warn(
```

[18]:
```python
import datetime
from scripts.helpers import distance
import pandas as pd
from numpy import mean, median
```

[19]:
```python
def calculate_average_speed_of_track(track: pd.DataFrame) -> float:
    total_time = 0.0
    total_distance = 0.0
    previous_time = datetime.datetime.fromisoformat(track.iloc[0]['time'][:-1])
    previous_position = (track.iloc[0]['lat'], track.iloc[0]['lon'])
    for _, _, lat, long, time in track.sort_values(by='time').itertuples():
        timestamp = datetime.datetime.fromisoformat(time[:-1])
        seconds = (timestamp - previous_time).total_seconds()
        if seconds > 0:
            dist = abs(distance(*previous_position, lat, long)) * 1000
            total_distance += dist
            total_time += seconds
        previous_time = timestamp
        previous_position = (lat, long)
    return total_distance / total_time
```

[21]:
```python
tracks = pd.read_csv('../data/raw/GPSTracksAssignment4/GPStracks.csv')
tracks_speed = {}
for track_id in tracks['TrackId'].unique():
    track = tracks[tracks['TrackId'] == track_id]
    speed = calculate_average_speed_of_track(track)
    tracks_speed[track_id] = speed * 3.6
    print(track_id, '->', speed * 3.6, 'km/h')
```

```
1 -> 7.116623587308064 km/h
3 -> 23.126448391942652 km/h
```

```
4 -> 13.695407924105305 km/h
5 -> 14.871617291497595 km/h
6 -> 14.471340208418587 km/h
7 -> 16.346108370386162 km/h
8 -> 22.46110292777329 km/h
10 -> 51.63218769316401 km/h
11 -> 25.019532962404657 km/h
```

[22]:
```python
stops = pd.read_csv('../data/processed/assignment1/line_stops.csv')


def get_closest_stop_in_direction(position, line_direction_stops):
    distances = line_direction_stops.apply(
        lambda row: distance(row['stop_lat'], row['stop_lon'], position['lat'],␣
↪position['lon']), axis=1)
    min_distance = distances.min()
    return line_direction_stops[distances == min_distance].iloc[0]


def get_closest_stops_in_line(position, line_stops):
    return (get_closest_stop_in_direction(position,␣
↪line_stops[line_stops['direction'] == 1]),
            get_closest_stop_in_direction(position,␣
↪line_stops[line_stops['direction'] == 2]))
```

[23]:
```python
def select_line_by_speed(track, track_speed, possible_lines, speeds):
    selected_line = None
    min_dif = 99999
    for _, line_id in possible_lines:
        first_stop, last_stop = get_first_and_last_stops(line_id, track)
        median_speed = get_speed_of_line_between_stops(first_stop, last_stop,␣
↪line_id, speeds)
        if abs(median_speed - track_speed) < min_dif:
            min_dif = abs(median_speed - track_speed)
            selected_line = line_id
    return selected_line


def get_speed_of_line_between_stops(first_stop, last_stop, line_id, speeds):
    line_speeds = speeds[speeds['LineId'] == int(line_id[:-1])]
    total_speed = []
    current_stop = int(first_stop['stop_id'].
↪strip('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'))
    while current_stop != int(last_stop['stop_id'].
↪strip('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')):
        row = line_speeds[line_speeds['FromStop'] == current_stop].iloc[0]
        total_speed.append(row['speed9'])
```

```
        current_stop = row['ToStop']
    median_speed = median(total_speed)
    return median_speed


def get_first_and_last_stops(line_id, track):
    line_stops = stops[stops['lineId'] == line_id]
    first_stops = get_closest_stops_in_line(track.iloc[0], line_stops)
    last_stops = get_closest_stops_in_line(track.iloc[-1], line_stops)
    if first_stops[0]['order'] < last_stops[0]['order']:
        first_stop = first_stops[0]
        last_stop = last_stops[0]
    else:
        first_stop = first_stops[1]
        last_stop = last_stops[1]
    return first_stop, last_stop
```

[ ]:
```
def get_type_of_line(line):
    return {'m': 'Metro', 't': 'Tram', 'b': 'Bus'}[line[-1]]


speeds = pd.read_csv('../data/processed/assignment1/vehicleSpeed.csv')

for track_id in tracks['TrackId'].unique():
    track = tracks[tracks['TrackId'] == track_id]
    possible_lines = track_predictions[track_id]
    if len(possible_lines) == 0:
        print(f'{track_id} ==> Other')
    elif len(possible_lines) == 1:
        print(f'{track_id} ==> {get_type_of_line(possible_lines[0][1])}␣
↪({possible_lines[0][1]})')
    elif len(possible_lines) > 1 and len({l[1][-1] for l in possible_lines}) ==␣
↪1:
        selected_line = select_line_by_speed(track, tracks_speed[track_id],␣
↪possible_lines, speeds)
        print(
            f'{track_id} ==> {get_type_of_line(possible_lines[0][1])} ({" || ".
↪join([l[1] for l in possible_lines])}) [ ==>␣
↪{get_type_of_line(selected_line)} ({selected_line})]')
    else:
        possible_types = {get_type_of_line(l[1]) for l in possible_lines}
        selected_line = select_line_by_speed(track, tracks_speed[track_id],␣
↪possible_lines, speeds)
        print(f'{track_id} ==> {" or ".join(possible_types)} ({" || ".
↪join([l[1] for l in possible_lines])}) ==> {get_type_of_line(selected_line)}␣
↪({selected_line})')
```

1.7 How did we do?

Using this approach, we are able to predict actual movement of the client for each route, and comparing against human validation, each prediction is correct.

Predicted

- Track 1: Other
- Track 3: Tram (082t)
- Track 4: Bus (050b)
- Track 5: Tram (082t)
- Track 6: Tram (008t)
- Track 7: Tram (007t)
- Track 8: Other

- Track 10: Tram (008t)
- Track 11: Tram (025t)

Validation

- Track 1: Other
- Track 3: Bus 50 or Tram 82 or Bus 49
- Track 4: Bus 50
- Track 5: Tram 82 or Tram 97
- Track 6: Tram 8
- Track 7: Tram 7
- Track 8: Other
- Track 10: Tram 8 or Tram 93
- Track 11: Tram 25

[ ]:
```

```

visualization

December 20, 2021

1 Visualizations

1.1 Preparing Data for Kepler.gl

2 What are we looking to accomplish?

Here we will transform our speed and delays data into a format that can be consumed by the **kepler.gl** tool. Data that is deemed important to include for visualization or filtering is all included and merged.

[1]:
```python
import pandas as pd
import re
import numpy as np
from keplergl import KeplerGl
```

[2]:
```python
# Cleaning the stops from letters and zeros
stops = pd.read_csv('../data/raw/gtfs/stops.txt')
stops = stops.drop(columns=['stop_code', 'stop_desc','zone_id','stop_url',␣
↪'location_type', 'parent_station'])
stops = stops # remove the .copy() if you want the change to be made in stops,␣
↪or do it directly there
stops['stop_id'] = stops['stop_id'].map(lambda x: int(re.sub('[^0-9]', '', x)))␣
↪# clean the letters and zeros
stops = stops.drop_duplicates(subset=['stop_id'], keep='first') # remove the␣
↪duplicates
len(stops)
```

[2]: 2842

[3]:
```python
line_stops = pd.read_csv('../data/processed/assignment1/line_stops.csv')
# Extracting the information we need from line_stops
line_stops = line_stops[['lineId','direction', 'name_ascii',␣
↪'stop_id_int','order', 'lat', 'long']]
line_stops['type'] = line_stops['lineId'].apply(lambda x: str(x[-1])) # getting␣
↪the type m/b/t
line_stops['lineId'] = line_stops['lineId'].apply(lambda x: int(x[:-1])) #␣
↪lineId number
```

```
line_stops = line_stops.drop(line_stops[line_stops.lineId >= 100].index) #␣
↪dropping the night buses
line_stops = line_stops.drop_duplicates(subset=['lineId', 'direction',␣
↪'stop_id_int','order'])
len(line_stops)
```

[3]: 3596

3 Speed - Assignment 1

[24]:
```
vehicle_speed_path = '../data/processed/assignment1/vehicleSpeedReformatted.csv'
vehicle_speed = pd.read_csv(vehicle_speed_path)
vehicle_speed.head(1)
```

[24]:
```
   LineId  FromStop  ToStop  Day LineIdFormatted LineId_GeoMerge Type  \
0      37      2957    5810  NaN          line37          037b-1    b

   Direction  fromIndex  toIndex  FromStop_lat  FromStop_lon  ToStop_lat  \
0        1.0       49.0     57.0     50.821413      4.341859   50.818572

   ToStop_lon                                            geojson   hour  \
0    4.340952  {'type': 'FeatureCollection', 'features': [{'t…  00:00

       speed
0  14.936818
```

[25]:
```
# vehicle_speed = vehicle_speed[['LineId', 'Type', 'FromStop', 'ToStop',␣
↪'Direction', 'hour', 'speed', 'geojson']]
vehicle_speed = vehicle_speed[['LineId', 'FromStop', 'ToStop', 'Direction',␣
↪'hour', 'speed',␣
↪'geojson','FromStop_lat','FromStop_lon','ToStop_lat','ToStop_lon']]

# Remove the inf values - we drop like 20k values
vehicle_speed = vehicle_speed.replace([np.inf], np.nan)
vehicle_speed = vehicle_speed.dropna()

vehicle_speed = vehicle_speed.astype({'Direction': int})
vehicle_speed.head(2)
```

[25]:
```
   LineId  FromStop  ToStop  Direction   hour      speed  \
0      37      2957    5810          1  00:00  14.936818
5      37      2957    5810          1  05:00   2.299439

                                             geojson  FromStop_lat  \
0  {'type': 'FeatureCollection', 'features': [{'t…     50.821413
5  {'type': 'FeatureCollection', 'features': [{'t…     50.821413
```

```
   FromStop_lon  ToStop_lat  ToStop_lon
0      4.341859   50.818572    4.340952
5      4.341859   50.818572    4.340952
```

4 Delays - Assignment 2

[27]:
```
# Using the delays and creating the Time coulmn from the Hour column: 7 -> '07:
↪00:00'
delays_path = '../data/processed/assignment2/final_filtered_delays.csv'

vehicle_delays = pd.read_csv(delays_path)
vehicle_delays['Time'] = vehicle_delays['Hour'].apply(lambda x: str(x.split(':
↪')[0]+':00'))

# vehicle_delays = vehicle_delays[['LineId', 'CurrentStop', 'Date', 'Time',␣
↪'Delay', 'Day', 'Hour']]
vehicle_delays.head(3)
```

[27]:
```
       Timestamp  LineId  TerminusStop  CurrentStop      Date      Hour  Day  \
0  1630915041551       2          8472         8382  20210906  07:57:21    0
1  1630915074628       2          8472         8763  20210906  07:57:54    0
2  1630915106373       2          8472         8372  20210906  07:58:26    0

   Delay ExpectedArrivalTime   Time
0    129            07:55:12  07:00
1     49            07:57:05  07:00
2    114            07:56:32  07:00
```

[28]:
```
# joining on lineId and stopid because we can have one station on different␣
↪lines
vehicle_delays = vehicle_delays.merge(line_stops, how='inner',␣
↪left_on=['LineId','CurrentStop'], right_on=['lineId','stop_id_int'] )
```

[29]:
```
vehicle_delays = vehicle_delays[['LineId', 'CurrentStop', 'Date','Hour',␣
↪'Time', 'Delay','direction', 'Day', 'type', 'name_ascii']]
```

[30]:
```
# cleaninug up the visual_data
vehicle_delays = vehicle_delays.drop_duplicates(subset=['LineId',␣
↪'CurrentStop', 'direction', 'Time','Day', 'type'])
# visual_data.rename(columns={'direction': 'Direction', 'order': 'Order',␣
↪'type': 'Type'}, inplace=True)
# visual_data = visual_data.astype({'LineId': str, 'Delay': 'int', 'Direction':␣
↪int, 'Order': int})
```

[31]:
```
#TODO Should we add order column?
#grouping by to get the result
grouped_delays = vehicle_delays.groupby(['LineId', 'type', 'CurrentStop',␣
↪'direction',  'Day', 'Time'], as_index=False)['Delay'].mean()
grouped_delays.head()
```

[31]:
```
   LineId type  CurrentStop  direction  Day   Time  Delay
0       2    m         8301          2    0  05:00  145.0
1       2    m         8301          2    0  06:00  113.0
2       2    m         8301          2    0  07:00   47.0
3       2    m         8301          2    0  08:00   19.0
4       2    m         8301          2    0  09:00   80.0
```

5 The merge

Once we transformed the data for the delays and vehicle speed we can merge them in one file having LineId', 'CurrentStop', 'direction', 'Time' as mutual columns on which we will join

[34]:
```
# Merging on FromStop
```

[16]:
```
visualization_data = grouped_delays.merge(vehicle_speed, how='inner',␣
↪left_on=['LineId', 'CurrentStop', 'direction', 'Time'],␣
↪right_on=['LineId','FromStop','Direction','hour'])
visualization_data = visualization_data.drop(columns=['direction'])
```

[17]:
```
visualization_data.head(1)
```

[17]:
```
   LineId type  CurrentStop  Day   Time  Delay  FromStop  ToStop  Direction  \
0       1    m         8011    0  05:00   58.0      8011    8271          2

    hour      speed                                            geojson  \
0  05:00  11.219475  {'type': 'FeatureCollection', 'features': [{'t…

   FromStop_lat  FromStop_lon  ToStop_lat  ToStop_lon
0     50.850095      4.352165   50.852347    4.347749
```

[18]:
```
# adding the 'line'+id to the LineId column so that we use it as string in the␣
↪visualization
visualization_data['LineId'] = visualization_data['LineId'].apply(lambda x:␣
↪'line'+str(x))
```

[19]:
```
visualization_data_stops = visualization_data.
↪merge(line_stops[['name_ascii','stop_id_int']], how='inner',␣
↪left_on='CurrentStop', right_on='stop_id_int').drop(columns=['stop_id_int'])
```

[20]:
```
visualization_data = visualization_data_stops.drop_duplicates()
```

We can drop the geojson column or we can use it in our data. Note that with the geojson we will generate a table of 2.5GB which kepler has difficulties runing.

[21]:
```
visualization_data_no_geojson = visualization_data.drop(columns=['geojson'])
```

[22]:
```
visualization_data_no_geojson.to_csv('../data/processed/Visualization/
↪visualization_data.csv')
```

[23]:
```
visualization_data_no_geojson
```

[23]:
```
        LineId type  CurrentStop  Day   Time  Delay  FromStop  ToStop  \
0        line1    m         8011    0  05:00   58.0      8011    8271
2        line1    m         8011    1  05:00 -198.0      8011    8271
4        line1    m         8011    0  06:00   66.0      8011    8271
6        line1    m         8011    1  06:00   71.0      8011    8271
8        line1    m         8011    0  07:00  -47.0      8011    8271
…          …   …            …    …     …      …        …       …
224958  line98    b         9686    0  17:00  -55.0      9686    5964
224959  line98    b         9686    1  17:00  110.0      9686    5964
224960  line98    b         9686    0  18:00  109.0      9686    5964
224961  line98    b         9686    1  18:00  314.0      9686    5964
224962  line98    b         9686    0  19:00  -92.0      9686    5964

        Direction   hour      speed  FromStop_lat  FromStop_lon  ToStop_lat  \
0               2  05:00  11.219475     50.850095      4.352165   50.852347
2               2  05:00  11.219475     50.850095      4.352165   50.852347
4               2  06:00  11.554020     50.850095      4.352165   50.852347
6               2  06:00  11.554020     50.850095      4.352165   50.852347
8               2  07:00  10.736122     50.850095      4.352165   50.852347
…               …      …          …             …             …           …
224958          2  17:00  30.144700     50.794407      4.302507   50.803967
224959          2  17:00  30.144700     50.794407      4.302507   50.803967
224960          2  18:00  30.133882     50.794407      4.302507   50.803967
224961          2  18:00  30.133882     50.794407      4.302507   50.803967
224962          2  19:00  33.475951     50.794407      4.302507   50.803967

        ToStop_lon    name_ascii
0         4.347749  DE BROUCKERE
2         4.347749  DE BROUCKERE
4         4.347749  DE BROUCKERE
6         4.347749  DE BROUCKERE
8         4.347749  DE BROUCKERE
…              …             …
224958    4.302302        DORENT
224959    4.302302        DORENT
224960    4.302302        DORENT
224961    4.302302        DORENT
```

```
224962    4.302302        DORENT

[112772 rows x 16 columns]
```

6 Visuals

[43]:
```python
visualization_data_no_geojson = pd.read_csv('../data/processed/Visualization/
↪visualization_data.csv')
```

[38]:
```python
map_1 = KeplerGl(height=600, data={"data": visualization_data_no_geojson,␣
↪"config": config})
```

```
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

[39]:
```python
map_1
```

```
KeplerGl(data={'data':         LineId type  CurrentStop  Day   Time  Delay ␣
↪FromStop  ToStop  \
0        line1…
```