Általános információk

A diplomaterv szerkezete:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletesés pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő Diplomaterv sablon dokumentum tartalma. Ügyeljen a konzulens nevét és a beadás évét jelölő szövegdobozokra, mert azokra külön ki kell adni a frissítést. A mezők tartalma a sablonban a dokumentum adatlapja alapján automatikusan kerül kitöltésre.

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2,5 cm, baloldalon 1 cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel.

Minden oldalon – az első négy szerkezeti elem kivételével – szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2 táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le.

Az irodalomjegyzék szövegközi hivatkozása történhet a Harvard-rendszerben (a szerző és az évszám megadásával) vagy sorszámozva. A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben. Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóirat címeket csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internet hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

* a szakdolgozat készítő/diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás Karunkon, ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
* mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozat készítést, ill. diplomatervezést kívánunk!

FeladatkiÍrás

The spread of electric vehicles is one of the defining trends in the automotive industry today. But in addition to passenger cars, this is also true for commercial vehicles and buses. In addition to more environmentally friendly operation, electric vehicles can also mean cost savings if they perform a suitable transport task.

However, the efficiency of electric vehicles can be affected by many factors, so it would be a huge manual task to determine on which lines it is worthwhile to replace vehicles equipped with internal combustion engines with electric ones, even for the bus network of a city the size of Budapest. Our goal is to create a machine learning model that, based on the typical parameters of bus lines, makes a good estimate of whether it is worth serving the given line with electric vehicles.

C:\Users\szarnyasg\Downloads\bme_logo_nagy.eps

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Pogácsasütöde Tanszék

Electric buses in Budapest

Készítette

Konzulens

2024

Tartalomjegyzék

[Összefoglaló 6](#_Toc167707326)

[Abstract 7](#_Toc167707327)

[1. Creating a simulation 8](#_Toc167707328)

[1.1. Creating a simulation 8](#_Toc167707329)

[1.2. Importing GTFS 8](#_Toc167707330)

[1.3. Importing Elevation Data 10](#_Toc167707331)

[1.4. Output files 11](#_Toc167707332)

[1.5. Getting vehicle IDs 12](#_Toc167707333)

[1.6. Defining new vehicle types 12](#_Toc167707334)

[2. Automating simulation running and data collection 14](#_Toc167707335)

[2.1. Script Overview 14](#_Toc167707336)

[2.2. Running Simulations 15](#_Toc167707337)

[3. Data Collection 16](#_Toc167707338)

[3.1. Data from Simulation Outputs 16](#_Toc167707339)

[3.2. Additional Data 16](#_Toc167707340)

[3.3. From XML to CSV 17](#_Toc167707341)

[3.3.1. XML Data Transformation 17](#_Toc167707342)

[3.3.2. Data preparation 17](#_Toc167707343)

[3.3.3. Data Aggregation 18](#_Toc167707344)

[3.3.4. To CSV 18](#_Toc167707345)

[4. Data cleaning 19](#_Toc167707346)

[4.1. Loading and Filtering the Data 19](#_Toc167707347)

[4.2. Dropping Unnecessary Columns 19](#_Toc167707348)

[4.3. Creating New Metrics 19](#_Toc167707349)

[4.4. Removing Redundant Columns 19](#_Toc167707350)

[4.5. Handling Outliers 19](#_Toc167707351)

[4.6. Calculating Delta Elevation 19](#_Toc167707352)

[4.7. Verifying Data Distribution 20](#_Toc167707353)

[4.8. Normalizing the Dataset 20](#_Toc167707354)

[4.9. Saving the Cleaned Data 20](#_Toc167707355)

[5. Building a model 21](#_Toc167707356)

[5.1. Choosing the Model 21](#_Toc167707357)

[5.2. Loading the Dataset 21](#_Toc167707358)

[5.3. Splitting the Data 21](#_Toc167707359)

[5.4. Evaluating the Model 21](#_Toc167707360)

[6. Conclusion 22](#_Toc167707361)

# Összefoglaló

A szakdolgozat, vagy diplomaterv elkészítése minden egyetemi hallgató életében egy fontos mérföldkő. Lehetőséget ad arra, hogy az egyetemi évei során megtanultakat kamatoztassa és eredményeit szélesebb közönség előtt bemutassa, s mérnöki rátermettségét bizonyítsa. Fontos azonban, hogy a dolgozat elkészítésének folyamata számos csapdát is rejt magában. Rossz időgazdálkodás, hiányos szövegszerkesztési ismeretek, illetve a dolgozat készítéséhez nélkülözhetetlen „műfaji” szabályok ismeretének hiánya könnyen oda vezethetnek, hogy egy egyébként jelentős időbefektetéssel készült kiemelkedő szoftver is csak gyengébb minősítést kapjon a gyenge minőségű dolgozat miatt.

E dokumentum – amellett, hogy egy általános szerkesztési keretet ad a dolgozatodnak – összefoglalja a szakdolgozat/diplomaterv írás írott és íratlan szabályait. Összeszedjük a Word kezelésének legfontosabb részeit (címsorok, ábrák, irodalomjegyzék stb.), a dolgozat felépítésének általános tartalmi és szerkezeti irányelveit. Bár mindenkire igazítható sablon természetesen nem létezik, megadjuk azokat az általános arányokat, oldalszámokat, amelyek betartásával jó eséllyel készíthetsz egy színvonalas dolgozatot. A részletes és pontokba szedett elvárás-lista nem csupán a dolgozat írásakor, de akár más dolgozatok értékelésekor is kiváló támpontként szolgálhat.

Az itt átadott ismeretek és szemléletmód nem csupán az aktuális feladatod leküzdésében segíthet, de hosszútávon is számos praktikus fogással bővítheti a szövegszerkesztési és dokumentumkészítési eszköztáradat.

# Abstract

English translation of the abstract of the thesis work. This summarises the content of the thesis in 0.5–1 pages and is uploaded to the Thesis Work Portal as well.

# Creating a simulation

For any artificial intelligence model, abundant data is essential. However, one of the most significant challenges in this project has been the lack of publicly available data on public transportation emissions. To overcome this challenge the idea was to use a traffic simulator and configure it in a way that represents real life public transportation.

"Simulation of Urban MObility" (SUMO) is an open source, highly portable, microscopic, and continuous traffic simulation package designed to handle large networks. It allows for intermodal simulation including pedestrians and comes with a wide range of tools for scenario creation. It is mainly developed by employees of the Institute of Transportation Systems at the German Aerospace Center.

## Creating a simulation

In SUMO, the OSM Web Wizard is a highly accessible tool for getting started. It allows users to configure a randomized traffic demand and run visual simulations based on an OpenStreetMap excerpt. This approach provides a straightforward way to simulate traffic in real-world locations. For this project, I used the OSM Web Wizard to create a simulation for a smaller area in Budapest.

To achieve this, I navigated to the SUMO tools library in the Windows Command Prompt and executed the following command:

python osmWebWizard.py

In the OSM Web Wizard, I then selected the desired area for simulation. The only modification I made was setting the simulation duration to one day (86400 seconds), the rationale for which will be discussed in the next chapter.

While this method offers ease of use, it is not without its drawbacks. The simulations generated produce numerous warnings, which I chose to overlook for this project because SUMO can resolve them during runtime. Although this is not an ideal solution, it is sufficient for the purposes of maintaining symmetry in both emission and battery simulations. Additionally, this method is impractical for larger areas, as it calculates vehicle numbers based on population density. For a densely populated city like Budapest, this results in an excessive number of vehicles that most computers cannot handle efficiently.

The generated configuration file (osm.sumocfg) can be opened in text editors or in sumo-gui which is basically the same application as sumo, just extended by a graphical user interface. In the next steps, I modified this file to ensure it consistently works with the same imports each time it is run.

## Importing GTFS

GTFS (General Transit Feed Specification) is a standardized format for public transportation schedules and associated geographic information. It enables public transit agencies to publish their data in a format that can be consumed by software applications. GTFS data is typically organized into a collection of text files, each representing a different aspect of the transit system, such as routes, stops, and schedules. In SUMO there are tools, that facilitate the import of schedules (and also routes to a limited degree) into a simulation.

For this project, I utilized the publicly available GTFS schedules for both Volán and BKK vehicles. Initially, I attempted to import the Volán GTFS into SUMO by copying it into the simulation library and executing the recommended commands from the SUMO documentation.

After these attempts proved unsuccessful, I proceeded to work with the BKK GTFS. Following the same steps, I encountered an error due to the absence of a calendar.txt file in the BKK GTFS. This file is conditionally required for defining the regular service patterns for public transit routes. To resolve this, I created an empty calendar.txt file containing only the column names, enabling the import process to proceed.

In the simulation folder, I used the following command:

python "C:/Program Files (x86)/Eclipse/Sumo/tools/import/gtfs/gtfs2pt.py" --gtfs "bkk\_gtfs\_e.zip" --date "20240325" --network "osm.net.xml.gz"

This command runs the gtfs2pt.py Python script to import a single day's schedule into the network. I specified a weekday (March 25, 2024) when public transit operates as usual. Since this method only allows for importing of entire days, I created a day-long traffic simulation in the previous step to accommodate this requirement.

This process generated files that needed to be added to the configuration file, replacing those created by the OSM Web Wizard. I updated the configuration file as follows:

<additional-files value="vtypes.xml,gtfs\_pt\_stops.add.xml,gtfs\_pt\_vehicles.add.xml"/>

A képen szöveg, képernyőkép látható

Automatikusan generált leírás

After completing this step, the stops and added vehicles can be viewed by running the simulation in sumo-gui.

## Importing Elevation Data

In SUMO, netconvert is a tool used for importing and converting road networks. Elevation data can be imported from a grayscale height map using the netconvert option --heightmap.geotiff. A grayscale height map is a digital image used in computer graphics and geographic information systems (GIS) to represent the elevation of terrain or other surfaces. Each pixel in the image corresponds to a specific geographic location, with the intensity of the grayscale value indicating the height at that point: darker pixels represent lower elevations, while lighter pixels represent higher elevations.

I downloaded a .tif file of Budapest from the CGIAR Consortium for Spatial Information (CGIAR-CSI) website. To import this file, I used the following command in the simulation folder:

netconvert -s osm.net.xml.gz --heightmap.geotiff srtm\_40\_03.tif -o height.net.xml

This command generated a new file, height.net.xml, which includes z coordinates for elevation. The final step was to update the configuration file to use this new network file:

<net-file value="height.net.xml"/>

Elevation data can be visualized in sumo-gui by colouring streets by height at start.

A képen szöveg, képernyőkép, diagram, térkép látható

Automatikusan generált leírás

## Output files

SUMO allows to generate many different measurements in XML (Extensible Markup Language). Per default, all are disabled, and in SUMO, various measurements can be generated, but by default, they are all disabled and need to be activated individually. For this project, the key outputs required are emissions, battery usage, and trip information. Emission output provides data on the emissions of all vehicles at each simulation step, battery output tracks battery state for electric vehicles, and trip information aggregates details about each vehicle's journey.

To enable these outputs, I added the following lines to the configuration file:

<output>

<emission-output value="emission.out.xml"/>

<tripinfo-output value="tripinfo.xml"/>

<battery-output value="battery.out.xml"/>

</output>

However, merely adding these lines generates an output file without any data. To collect data, each vehicle must be assigned a device. This can be done explicitly by specifying the vehicle IDs or by randomly assigning a certain percentage of vehicles a device. I opted to explicitly define devices for each vehicle, ensuring that all vehicles have all necessary devices. This was achieved by adding the following lines to the configuration file:

<emissions>

<device.emissions.explicit value="<!--put vehicle ids here-->"/>

</emissions>

<tripinfo>

<device.tripinfo.explicit value="<!--put vehicle ids here-->"/>

</tripinfo>

<battery>

<device.battery.explicit value="<!--put vehicle ids here-->"/>

</battery>

In the placeholder <!--put vehicle ids here-->, the specific vehicle IDs will be listed.

## Getting vehicle IDs

To obtain the vehicle IDs, I utilized the gtfs\_pt\_vehicles.add.xml file, which was created earlier by importing the GTFS data. This file lists all added vehicles in XML (Extensible Markup Language) format along with their properties, including their IDs. I developed a Python script to read this file, extract the IDs from all vehicle tags, and compile them into a list. Once the list was complete, the script wrote the list elements, separated by commas, into an output file named vehicle\_ids.txt.

From this output file, I randomly selected IDs and pasted them into the configuration file in place of every placeholder. It is crucial for this step that every vehicle is assigned all necessary devices.

<emissions>

<device.emissions.explicit value="C775912413.0, C77776334.0, C763111293.0, C577041748.0, C77704682.0, C76412107.0, C76311425.0, C7778220.0, C731633508.0, C7612853.0"/>

</emissions>

<tripinfo>

<device.tripinfo.explicit value="C775912413.0, C77776334.0, C763111293.0, C577041748.0, C77704682.0, C76412107.0, C76311425.0, C7778220.0, C731633508.0, C7612853.0"/>

</tripinfo>

<battery>

<device.battery.explicit value="C775912413.0, C77776334.0, C763111293.0, C577041748.0, C77704682.0, C76412107.0, C76311425.0, C7778220.0, C731633508.0, C7612853.0"/>

</battery>

## Defining new vehicle types

While analyzing the output data, I noticed that the energy consumption of electric buses was significantly lower than real-life values. This discrepancy arose because the default weight for an electric vehicle using the battery device is set to 1830 kg, which corresponds to a standard car. To address this, I defined a custom vehicle type, named ElectricBus. Using a SUMO example definition of a city bus as a base, I adjusted the vehicle mass from 10,000 kg to 18,000 kg to reflect the average mass of a bus in Budapest.

Additionally, I assigned an emission class to the vehicle type, which all buses have by default. This emission class uses the HBEFA4 model, which is free to use but does not account for elevation data. I selected the class UBus\_Std\_gt15-18t\_Euro-VI\_A-C, as it represents a standard 15-to-18-ton bus and has comprehensive fleet data.

Changing the emission class affects the emission output; however, for electric vehicles, I only need this output to track the bus's position at each simulation step. Therefore, I did not define an emission class for electric buses.

After these steps, the definition of vTypes in the vtypes.xml file is as follows:

<vType id="bus" vClass="bus" emissionClass="HBEFA4/UBus\_Std\_gt15-18t\_Euro-VI\_A-C"/>

<vType id="light\_rail" vClass="rail\_urban"/>

<vType id="tram" vClass="tram"/>

<vType id="ElectricBus" accel="1.0" decel="1.0" length="12" maxSpeed="100.0" sigma="0.0" minGap="2.5" color="1,1,1">

<param key="maximumBatteryCapacity" value="3000"/>

<param key="maximumPower" value="1000"/>

<param key="vehicleMass" value="18000"/>

<param key="frontSurfaceArea" value="5"/>

<param key="airDragCoefficient" value="0.6"/>

<param key="rotatingMass" value="100"/>

<param key="radialDragCoefficient" value="0.5"/>

<param key="rollDragCoefficient" value="0.01"/>

<param key="constantPowerIntake" value="100"/>

<param key="propulsionEfficiency" value="0.9"/>

<param key="recuperationEfficiency" value="0.9"/>

<param key="stoppingThreshold" value="0.1"/>

</vType>

At this stage of the project, I created a copy of the simulation folder and established a naming convention. For simulations involving diesel buses, I appended the suffix "\_e" to the folder name to indicate that the emission output is the primary focus. For simulations involving electric buses, I appended the suffix "\_b" to the folder name to indicate that the battery output is the primary focus.

In the electric bus simulation, I modified the gtfs\_pt\_vehicles.add.xml file by replacing the vehicle type of each bus with the previously defined ElectricBus type. I accomplished this by using the find and replace feature in Notepad to change type="bus" to type="ElectricBus".

For efficiency purposes, I also removed the battery output and battery devices in the diesel bus simulation.

# Automating simulation running and data collection

In this chapter, I detail the process of automating the execution of simulations and subsequent data collection. This automation is crucial for handling multiple simulation runs efficiently, ensuring consistency, and reducing the likelihood of human error.

To achieve this automation, I created a Python script that utilizes the subprocess module to run the simulations and collect data. The script is designed to iterate over a range of seeds and traffic scales, enabling a comprehensive analysis of various scenarios.

## Script Overview

The script starts by defining the base command for running the simulation using the SUMO (Simulation of Urban MObility) tool. The command includes placeholders for the simulation folder, seed, traffic scale, and offset, which are dynamically filled during execution.

import subprocess

# Define the command to run the simulation

simulation\_command\_base = "sumo --c C:\\Users\\Admin\\Sumo\\{}\\osm.sumocfg --seed {} --scale {} --delay 0 --random-depart-offset {} --start"

Additionally, the script defines the base command for running a Python script to process the simulation results. The Python script name and log file name depend on the type of simulation (battery or emission), which is determined by the last character of the simulation folder name.

python\_command\_base = "python {} {} {} {} > {}"

# Define the range of seeds and traffic scales to iterate over

seeds = range(1, 4) # Assuming seeds range from 1 to 3

traffic\_scales = [0.3, 0.5, 0.7] # Traffic scales of 0.3, 0.5 and 0.7

offset = 100

simulation\_folder = "name\_of\_the\_folder" #change manually

# Define the mapping of simulation folder endings to their corresponding Python filenames and log filenames

simulation\_settings = {

'b': ("battery.py", "battery\_log.txt"),

'e': ("emission.py", "emission\_log.txt")

}

# Extract the last character of the simulation folder to determine the type

folder\_type = simulation\_folder[-1]

# Get the corresponding filenames from the dictionary, raising an error if the ending is incorrect

try:

python\_filename, log\_file\_name = simulation\_settings[folder\_type]

except KeyError:

raise ValueError("Incorrect simulation folder ending")

## Running Simulations

The script opens a log file for writing the output of the data processing scripts. It then iterates over all combinations of seeds and traffic scales, constructing and executing the simulation command for each combination.

log\_file = open(log\_file\_name, "w")

# Iterate over all combinations of seeds and traffic scales

for seed in seeds:

for traffic\_scale in traffic\_scales:

# Construct the full simulation command with the current seed and traffic scale

simulation\_command = simulation\_command\_base.format(simulation\_folder, seed, traffic\_scale, offset

# Run the simulation command

subprocess.run(simulation\_command.split(), shell=True)

print(f"{seed} simulation finished")

# Run the Python script to process the simulation data

subprocess.run(["python", python\_filename, str(seed), str(traffic\_scale), str(simulation\_folder)], stdout=log\_file)

print(f"{seed} data processing finished")

After each simulation run, the script executes the corresponding Python data processing script (battery.py or emission.py). The output of these scripts is redirected to the log file, which provides a record of the data processing results. Finally, the script closes the log file, ensuring all data is properly saved and the file is not left open.

# Data Collection

In this stage, we focus on collecting data from simulation outputs and other significant features of the simulation. This is the part that explains how battery.py and emission.py work form the previous chapter work.

## Data from Simulation Outputs

Trip Information Output:

* id: Represents the trip ID and the vehicle's trip\_id.
* timeloss: Time lost due to driving below the ideal speed (in seconds). This includes delays from intersections and other factors, excluding scheduled stops. The ideal speed accounts for the individual speed factor.
* routeLength: The total length of the vehicle's route (in meters).

Emission Output (for diesel buses):

* fuel: The fuel consumption by the vehicle during the actual simulation step (in mg/s).
* speed: The speed of the vehicle (in m/s).
* pos\_z: The Z coordinate of the vehicle.

Battery Output (for electric buses):

* totalEnergyConsumed: Total electrical energy consumed (in Wh/s).
* totalEnergyRegenerated: Total electrical energy regenerated (in Wh/s).

In this project, I developed a Python script to read and transform data for subsequent analysis and to generate data for building an artificial intelligence model. The script processes simulation outputs and aggregates key metrics for each vehicle group.

## Additional Data

GTFS Data:

* route\_id: Groups together trips that operate along the same general path and usually serve the same set of stops. Each route\_id can have multiple trip\_ids, but each trip\_id is associated with only one route\_id.

To find the route\_id, I read the trips.txt file into a pandas DataFrame and located the row with the vehicle’s trip\_id, then extracted the route\_id value from that row.

Vehicle Stops Data:

* stops: The number of stops imported for the vehicle in the specified area.

I read the gtfs\_pt\_vehicles.add.xml file into a pandas DataFrame by creating a dictionary representing a route with its attributes and a list of stops. Each stop is represented as a dictionary containing its attributes. I then counted the number of stops associated with the route matching the vehicle's route\_id.

Simulation Metadata:

* location: The name of the simulation folder.
* seed: The seed value used for the simulation's pseudorandom number generator. SUMO utilizes the Mersenne Twister algorithm for generating random numbers, initialized with a default seed value of 23423. This setup ensures deterministic applications as the sequence of random numbers remains fixed for a given seed.
* trafficScale: The scale of the traffic during the simulation. This value scales the frequency/probability of any loaded flows proportionally, including newly loaded vehicles.

## From XML to CSV

XML is a flexible, structured format for storing and transporting data, while CSV (Comma-Separated Values) is a simpler, widely used format for representing tabular data. Converting XML to CSV is necessary for efficient data analysis and manipulation, as CSV files are easier to process and integrate with data analysis tools like pandas.

### XML Data Transformation

To read and transform XML files, I utilized a function that converts XML data into a series of dictionaries. These dictionaries combine attributes from the root element with those of specified child elements. The base folder is the path to the simulation folder where the outputs are generated.

def transform\_xml(file\_name, tag):

xml\_doc = et.parse(os.path.join(base\_folder, file\_name)).getroot()

attr = xml\_doc.attrib

for xml in xml\_doc.iter(tag):

\_dict = attr.copy()

\_dict.update(xml.attrib)

yield \_dict

**Function Input:**

file\_name: A string representing the name of the XML file to be parsed.

tag: A string specifying the tag name to search for within the XML document.

**Function Output:**

Yields dictionaries containing a combination of attributes from the root XML element and the attributes of each found element with the specified tag.

### Data preparation

I converted these dictionaries into lists and subsequently into pandas DataFrames. Pandas is an open-source Python library that offers fast, powerful, flexible, and easy-to-use tools for data analysis and manipulation.

Upon visualizing these DataFrames, I identified and removed unnecessary XML columns by deleting the first column in some cases. Then I grouped emission data by id and put the data frames into a list.

### Data Aggregation

The script then groups emission data by vehicle ID and compiles the data into a list of DataFrames. This process involves calculating and aggregating important metrics for each vehicle group, including:

* Route Id
* Trip Id
* Average Speed: The mean speed of the vehicle during the trip
* Time Loss
* Route Length
* Number of Stops: The total number of stops on the route.
* Elevation Up: Based on the z coordinates.
* Elevation Down: Based on the z coordinates.
* Fuel Consumption (HUF)
  + For diesel buses: The total cost, excluding VAT, of fuel consumption adjusted for elevation changes.
  + For electric buses: Calculated by subtracting the energy regenerated from the total energy consumed and then determining the cost.

Additionally, data such as seed, simulation location, and traffic scale—which remain constant for every bus in the same simulation—were collected by retrieving the system arguments used to call the Python script described in the previous chapter.

These aggregated metrics are stored in a final pandas DataFrame, facilitating further analysis and reporting. This automated approach ensures consistency and accuracy in handling the large datasets typically produced in simulations. By systematically transforming and aggregating the data, this script lays a robust foundation for building and training an artificial intelligence model, enabling effective data-driven decision-making.

### To CSV

Finally, the pandas DataFrame was appended to a previously created CSV file, with columns established for the data. This CSV file consolidates data from all simulation runs, ensuring it is structured and ready for use with an AI model.

# Data cleaning

Data cleaning is a fundamental step in the data preparation process, ensuring that subsequent analysis, interpretation, and decision-making are based on reliable and high-quality information. In this chapter, we will examine the modifications made to the dataset to achieve this.

## Loading and Filtering the Data

I began by loading the emission and battery data from CSV files, turned them into Pandas data frames and filtered the datasets to include only entries with a seed value of 3 or less. This initial step ensured that our analysis was confined to a relevant subset of the data. To avoid overfitting the best practice would be to only have a few seeds per simulation, this is possible if I create more data in the future.

## Dropping Unnecessary Columns

Next, I removed columns that were not needed for further analysis. This step streamlined the datasets, focusing only on the variables essential for our analysis.

Removed columns include:

* routeId
* tripId
* seed
* location

## Creating New Metrics

I created new columns to express various metrics per meter of route length, such as stops, emissions, elevation changes, and time loss per meter. These derived metrics provided a standardized way to compare different routes.

## Removing Redundant Columns

After creating the new metrics, I dropped the original columns that had become redundant. This step helped to avoid redundancy and reduced the complexity of the dataset.

## Handling Outliers

I identified and removed outliers in the timeloss\_per\_m column by checking various diagrams, specifically the row with the maximum value, to prevent skewed analysis results due to extreme values.

## Calculating Delta Elevation

I calculated the difference between elevation gains and losses per meter and adjusted the dataset accordingly. This new metric, delta elevation per meter, offered additional insights into the route characteristics. I experimented with teaching the model using both this new column and the separate elevation values. However, results were not as favorable with delta elevation, so I decided to skip this step.

## Verifying Data Distribution

To verify the distribution of our cleaned data, I performed the Shapiro-Wilk test on each column. This statistical test helped determine whether the data columns followed a normal distribution. None of the columns did, which is consistent with findings in similar studies.

## Normalizing the Dataset

I normalized the entire dataset by dividing all values by the maximum value of each column to ensure uniformity across all columns, facilitating more accurate and meaningful comparisons. I also experimented with teaching the model both with and without normalization. The results were similar, so I am still determining whether this step is necessary.

## Saving the Cleaned Data

Finally, I saved the cleaned datasets to new CSV files. This step preserved the refined datasets for further use.

# Building a model

In this chapter, we describe the process of building an AI model on top of the cleaned dataset. The model aims to predict emissions per meter using various route and trip characteristics as features. The steps involve data preparation, model training, evaluation, and interpretation.

## Choosing the Model

## Loading the Dataset

I started by loading the cleaned battery dataset into a Pandas DataFrame. I also determined the range of the target variable, emission\_per\_m, to understand its distribution:

min\_value = df['emission\_per\_m'].min()

max\_value = df['emission\_per\_m'].max()

range = max\_value - min\_value

## Splitting the Data

The dataset was split into features (X) and the target variable (y):

X = df.drop(columns=['emission\_per\_m']) # Features

y = df['emission\_per\_m'] # Target variable

We then divided the data into training and testing sets to enable model evaluation on unseen data:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=5)

## Evaluating the Model

The trained model was used to predict emissions on the test set. We evaluated the model's performance using Mean Squared Error (MSE):

y\_pred = regression\_tree.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

print("Mean Squared Error:", mse)

# Conclusion