rijksuniversiteit groningen

Individual Assignment

Data Analysis and Programming for Operations Management
Dr. Michael Dienstknecht

Date: 13-03-2024

Guillermo Gil De Avalle Bellido (S5787084)

Table of Contents

Introduction

In this analysis, we delve into the operational challenges faced by a carsharing provider in Groningen, marked by poor profitability and customer satisfaction. Employing detailed request data collected over a year, the goal is to unveil demand patterns, optimize fleet allocation, and enhance service efficiency. Through a combination of data cleaning, exploratory analysis, and mathematical modeling, including the strategic assignment of vehicles based on customer demand and location data, we aim to propose actionable insights

Dataset Observations

The car_locations.json and request_data.json datasets provide critical insights for a carsharing service operating in Groningen, albeit with notable issues affecting their utility. The car_locations.json file, with 500 entries and 31KB, inaccurately positions the first 200 cars' locations due to reversed latitude and longitude coordinates, affecting its logistical use. Similarly, the request_data.json dataset, documenting a year's worth of customer requests with 615,046 rows and 130MB, suffers from the same lat-long inversion in all entries, compromising its effectiveness for demand analysis and service improvement. These errors underscore the urgent need for thorough data verification and correction to avoid misleading operational strategies and to boost the service's efficiency and customer satisfaction in Groningen.

Ingestion and Cleaning

The initial step involved ingesting the relevant data. Given the substantial size of the request_data.json dataset and potential computational delays with dataframes, a NoSQL database approach was preferred. Elasticsearch was selected, and a mapping including essential properties for the assignment was created: origin_datetime, destination_datetime, origin_location, destination_location (all formatted appropriately), request_nr, and weekend (formatted as integers). The binary format of the weekend field suggested a Boolean structure, yet Elasticsearch documentation indicated that 'boolean' values require explicit 'true'/'false' labels, not numerical representations.

For data ingestion, the bulk method was employed via the ingest_json_file_into_elastic_index function from the loaders.py script. Successful ingestion led to the data cleaning phase, involving four specific functions: three as outlined by the case and an additional one for correcting the latitude and longitude inversions. A for loop iterated over the records, applying cleaning functions, including the coordinate swap, distance calculations (using the geodesic library), time difference calculations, and speed computations. Subsequently, an if statement assessed whether the calculated speeds fell within the reasonable urban speed range of 5 to 50 km/h for Groningen, considering the city's varied speed limits. This process resulted in a cleaned dataset of 600,065 entries, excluding 14,981 records that fell outside the specified criteria.

Demand Pattern Analysis

Now that the file was cleaned, exploratory descriptive analysis began to uncover patterns like seasonality. An Elasticsearch aggregation query gathered dates and demand from the year-long carsharing request dataset, aggregating data by origin_datetime at daily intervals via date histogram aggregation. This facilitated document grouping by specific time frames. Subsequently, a for loop parsed the query responses into two lists—dates and demand—later transformed into pandas datetime objects for enhanced slicing and matplotlib integration. The analysis culminated in a matplotlib line graph visualizing daily demand throughout the year, with improvements like label rotation enhancing visibility.

The results depicted in Figure 1 demonstrate variability in demand across almost all months, with fluctuations of approximately ±1000 request demands occurring in very short intervals—often from one day to the next—likely influenced by weekends, where car-sharing demand tends to spike. Despite this, the most significant deviation is observed when analyzing monthly demand trends. A notable decrease in demand is consistently seen throughout August and another discernible drop between the end of December and the beginning of January. These patterns align closely with the demographics of Groningen, where almost 25% of the population is connected to the university. The timing of these demand dips correlates with periods of reduced activity in the academic calendar, namely the summer and Christmas vacations, offering a plausible explanation for the observed trends in car-sharing usage.

Demand Forecasting Analysis

After examining seasonality, attention shifts toward forecasting future car demand to ensure availability for all potential customers. To this end, another Elasticsearch query was conducted to ascertain origin_datetime at 'hour' calendar intervals using the date histogram, this time incorporating a "terms" filter to exclusively include results from typical working days (Monday to Friday), identified by a weekend value of 0. Utilizing a process similar to the one previously described, involving lists, a for loop, and a dataframe for simplified manipulation, we proceeded to calculate relevant statistical measurements. These included aggregation by hour and demand, with statistical measures encompassing mean (for both per data point and overall), standard deviation, minimum, and maximum. We believe these parameters are essential for discerning whether fluctuations in demand necessitate adjustments in car availability.

The findings, illustrated in Figure 2, reveal two distinct demand peaks on working days: one from 7:00 to 8:00h and another from 15:00 to 18:00h, both surpassing the overall mean demand. Additionally, the hours from 9:00h to 14:00h also exhibit relatively higher demand compared to nighttime, though not exceeding the mean. This observation aligns with our hypothesis that students (and possibly commuters) constitute a significant portion of the user base, with the peak hours corresponding to typical university or job start and end times. The observed wider variance in standard deviation, minimum, and maximum during these peak periods suggests less predictability in demand. If ensuring higher availability is paramount—as the case indicates—then it is crucial to guarantee adequate car coverage during these peaks. Enhancements were also applied to the visualization to facilitate a clearer interpretation of the data, employing a bar chart, scatter plot, and horizontal bars to illustrate individual deviations.

Next, our goal is to determine the expected number of requests on a typical working day. To achieve this, we initiate another Elasticsearch query, closely mirroring the approach used for aggregating hourly data, with the primary distinction being that this query aggregates data by day. Once the data is extracted and efficiently allocated to a dataframe, we proceed to calculate the average daily demand using the numpy mean function. Subsequently, in accordance with the case's guidance, the demand is augmented by 30%,

resulting in an adjusted average daily demand of 2521. This figure is consistent with the observations from Figure 1, where daily demand was also analyzed, albeit without the 30% increase.
Finally, our last objective was to draw a sample from an appropriate subset of the raw data, reflecting the adjusted average daily demand (as our sample size), and to visualize the corrected demand pattern observed on a typical working day by the hour. This was to determine if the new sample was significant enough to yield a similar visualization. We began by selecting a sample size of 2521, using Python's random options to select a sample from the dataframe utilized for the hourly visualization. Employing a random_state of 42 was crucial to ensure the random sample remained consistent upon multiple executions of the code. Subsequently, the demand for the random sample was also increased by 30%, and the previously defined statistical measures (individual mean, overall mean, standard deviation, maximum, and minimum) were recalculated. The random sample was then visualized using the same Pyplot settings. The results, depicted in Figure 3, showcased an almost identical representation to the original, albeit with increased demand, indicating that the selected sample is significant enough to draw conclusions about the overall dataset.

Optimization Problem

After exploring forecasting possibilities and identifying a significant random sample, we propose an optimization model to enable providers to assign cars more efficiently. This model, mathematically detailed in Figure 7, seeks to maximize profits for the carsharing company by accommodating as many requests as possible within certain constraints (e.g., maximum walking distance, w) and parameters (e.g., standard profit per minute per served request). The decision variable is binary, indicating whether a request is served.

To implement this model using the Gurobipy library, we first prepared the data from the two datasets. Given the smaller size of car_locations.json, we used json.loads(line) for data integration into a dataframe, including an on-the-fly correction of latitudes and longitudes for the first 200 rows. For request_data.json, an Elasticsearch query fetches all fields for a sample size matching the adjusted average daily demand. The data was formatted, and mathematical parameters, along with two lists (profits and matched assignments), were defined for subsequent use.

The optimization model required pre-model calculations to preserve linearity, such as computing distances between request origins and car locations and assessing all request/car compatibilities. An initial iteration of our analysis created a standalone code only for this model. However, since the optimization model is also necessary for the next part of analysis, a decision is made to transform these two parts (check compatibility/distance, and optimize model), into two different functions. The first function calculates distances and checks compatibility across different w values using nested for loops through car and request data. Following this, the second function determines the necessary parameters (length of cars and requests) and integrates various model components into Gurobipy, including variable addition, objective setting, and constraint definition, ensuring a streamlined analysis process.

Following the execution of these functions for w = 0.4, an if statement verifies the optimization's success. Upon confirmation, profits and matches are added to their respective lists. For matches, a double for loop (spanning both cars and requests dataframes) facilitates assignment into a dataframe, which is then segmented into longitudes and latitudes for visualization purposes. Utilizing the Smopy library (which plots maps based on the minimum and maximum longitudes and latitudes as bounding box) two visual representations are created: one showcasing matched requests—including origins, linked cars, and destinations—as illustrated in Figure 4, and another depicting the origins of unmatched cars in the sample, as seen in Figure 5. The results indicate that out of a sample of 2521, only 330 requests are matched to cars for $w = 0.4$, constituting 13% of the requests. This scenario only yields a maximized profit of 531.05, which is far from ideal and does not align with the company's aims for a high availability.

Walking distance (w) impact on Profits

Given the low coverage percentage of the optimization model at $w = 0.4$, exploring the impact of walking distance on profits becomes the next logical step to assess the viability of this constraint. Fortunately, functions for checking compatibility and distance, as well as optimizing the model, are already established, necessitating only the reset of lists for recalculations at different walking distances. Nineteen points were chosen for a chart to effectively show the correlation, initially raising concerns about rendering time. Solutions like storing pre-matched data and enhancing compatibility checks in Elasticsearch were considered. However, the rendering time for the code, excluding ingestion and cleaning, was found to be around 10 minutes, deemed optimal for a one-time data study.

Following the model's execution across varied walking distances, a simple line chart with a pre-calculated standard deviation and a marker for the current max walking distance was plotted. These enhancements aid in understanding data sensitivity and envisioning outcomes beyond the current maximum distance. Results in Figure 6 illustrate profit increases that taper off as w nears 0.4, with a slight acceleration up to 0.7 before slowing again. At a 1km maximum, matching requests peak at 2521, or 19.6% of requests, indicating a modest rise. While initially suggesting an increase in the maximum distance might be feasible, other factors, such as potential customer aversion to longer distance matches, must be considered. Despite a significant young and active user base, the city's accessible and affordable bike and bus transport options might lead this group to prefer these modes of transport over extended walking distances.

Conclusions

Overall, our analysis reveals seasonality in car sharing demand during August and December, with peaks at typical commuting hours, suggesting a user base likely linked to universities or office environments. The optimization model, with w set to 400m, doesn't significantly maximize profits, covering only 13% of total requests. Even increasing w to 1 km barely improves coverage, matching only 19.6% of requests.

Given these findings, we propose a different model. Instead of positioning cars in strategic areas awaiting customers, part of the fleet could be mobilized during peak times at congested nodes frequented by our primary user base, such as universities and major offices. We suggest leveraging our optimization algorithm to dynamically match these mobile cars with incoming requests from this user base. This approach aims to enhance the utility of Python's dynamic capabilities, as well as applying the insights from this one-time analysis to refine targeting based on user location and timing, potentially improving service efficiency and customer satisfaction.

ANNEX A: Figures

Note: The figures cannot be enlarged with A4 paging settings. Please zoom in for better readability.

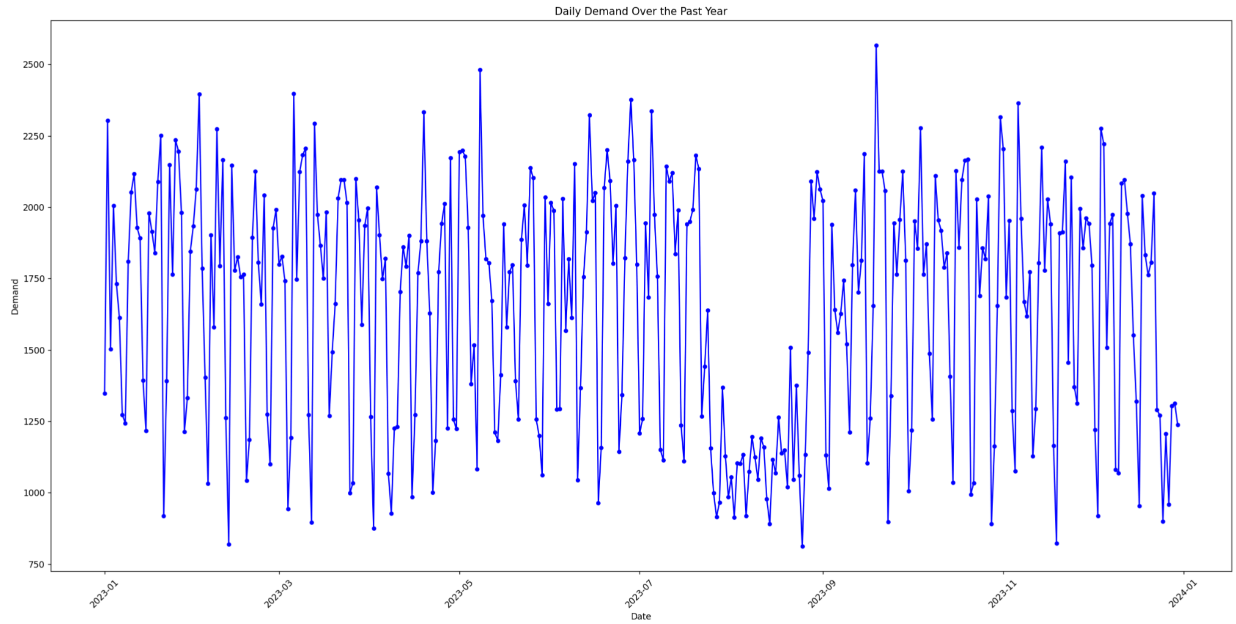

Figure 1: Daily Demand over the part year.

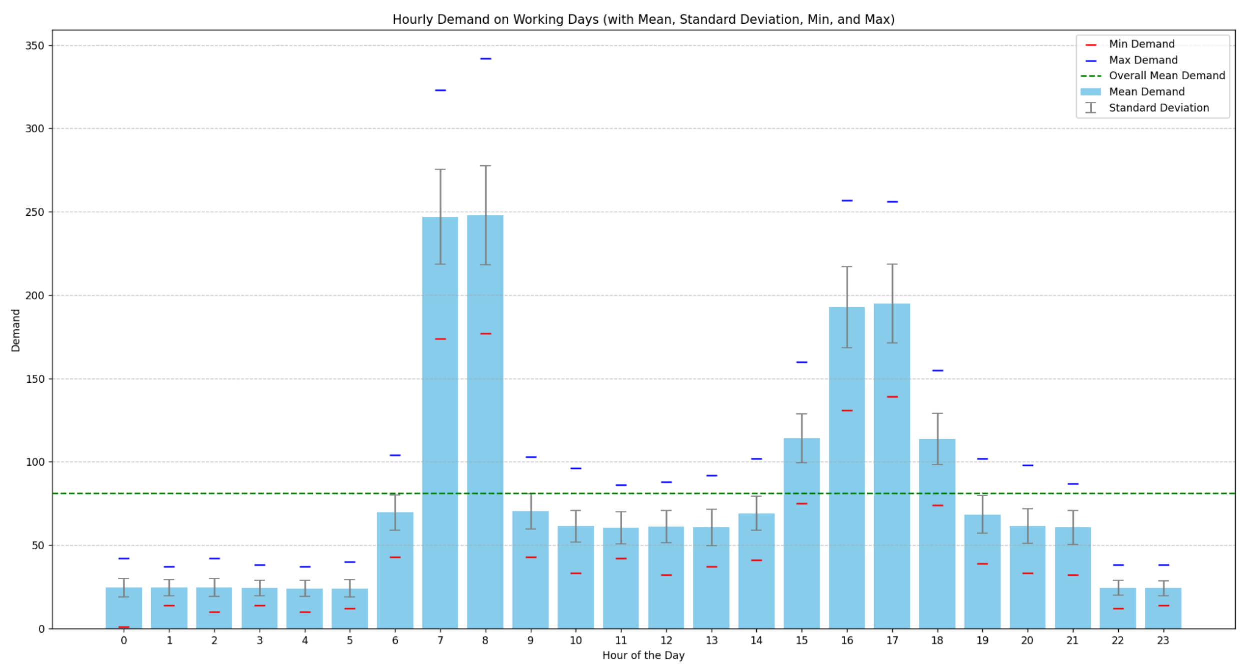

Figure 2: Aggregated hourly demand on working days over all requests

Including Min, Max, Means and Standard Deviation

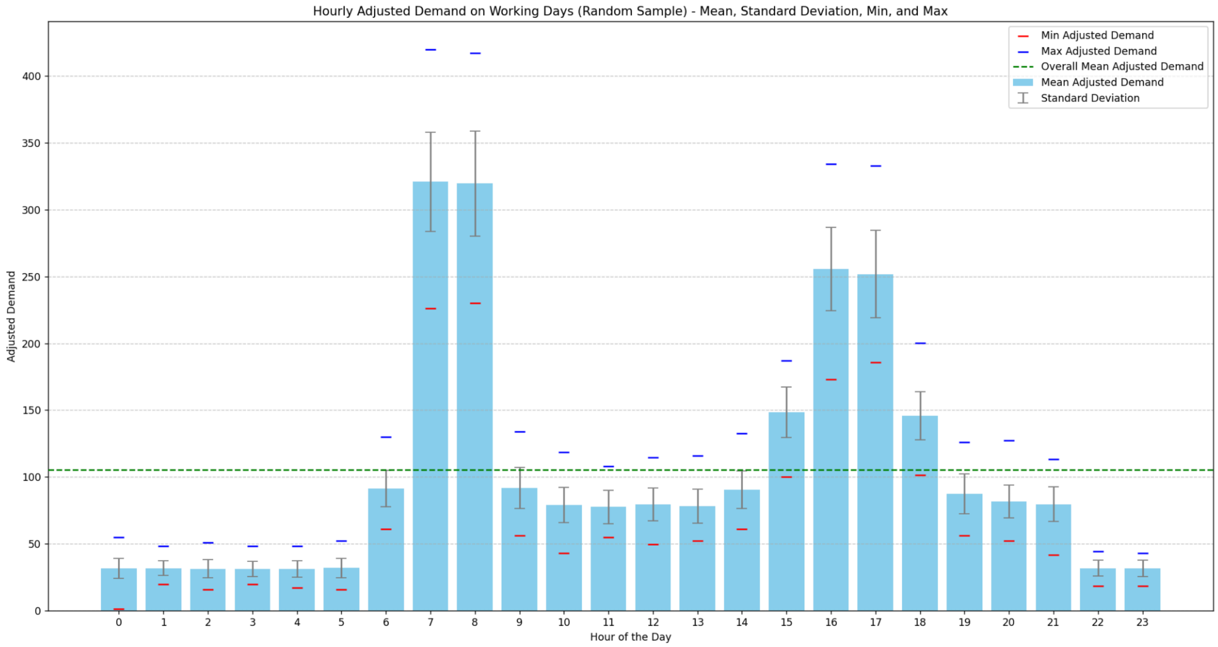

Figure 3: Aggregated hourly demand on working days over random sample (with 30% increase in demand)
Including Min, Max, Means and Standard Deviation

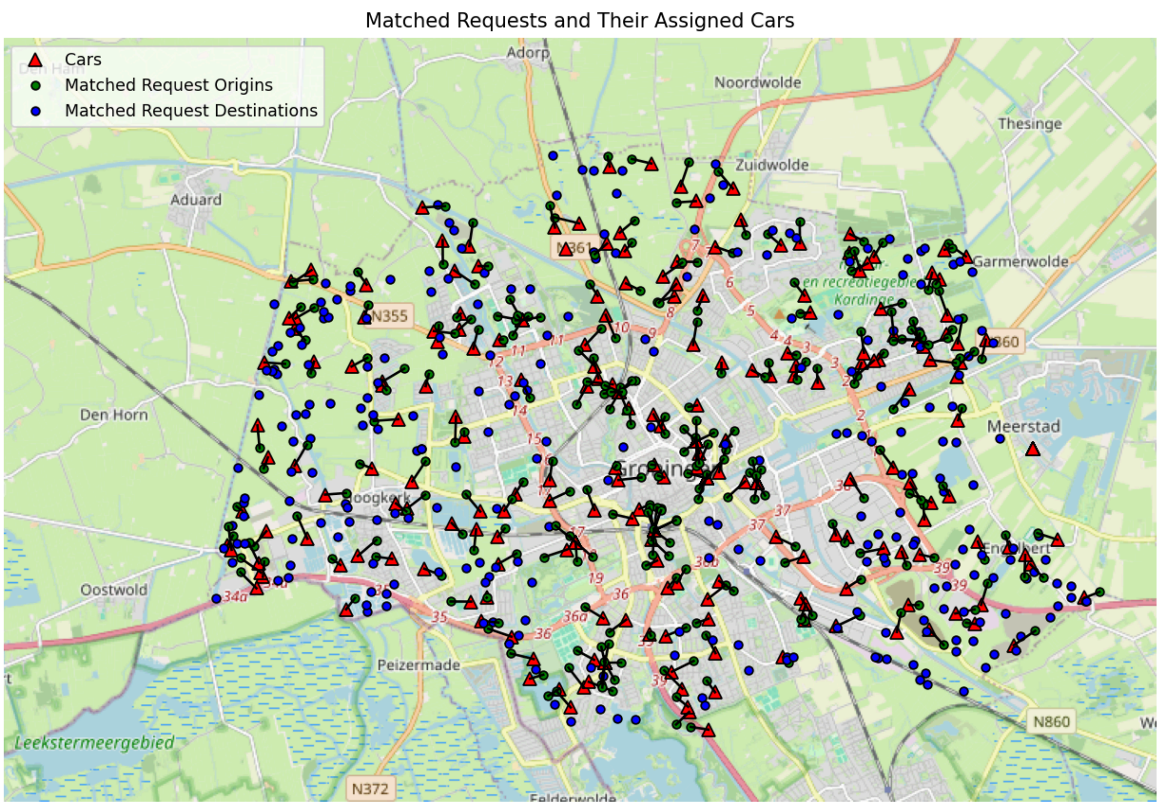

Figure 4: Gurobi Optimization Matches between request and cars for w = 0.4km
Including car location, matched requests origins, and destinations

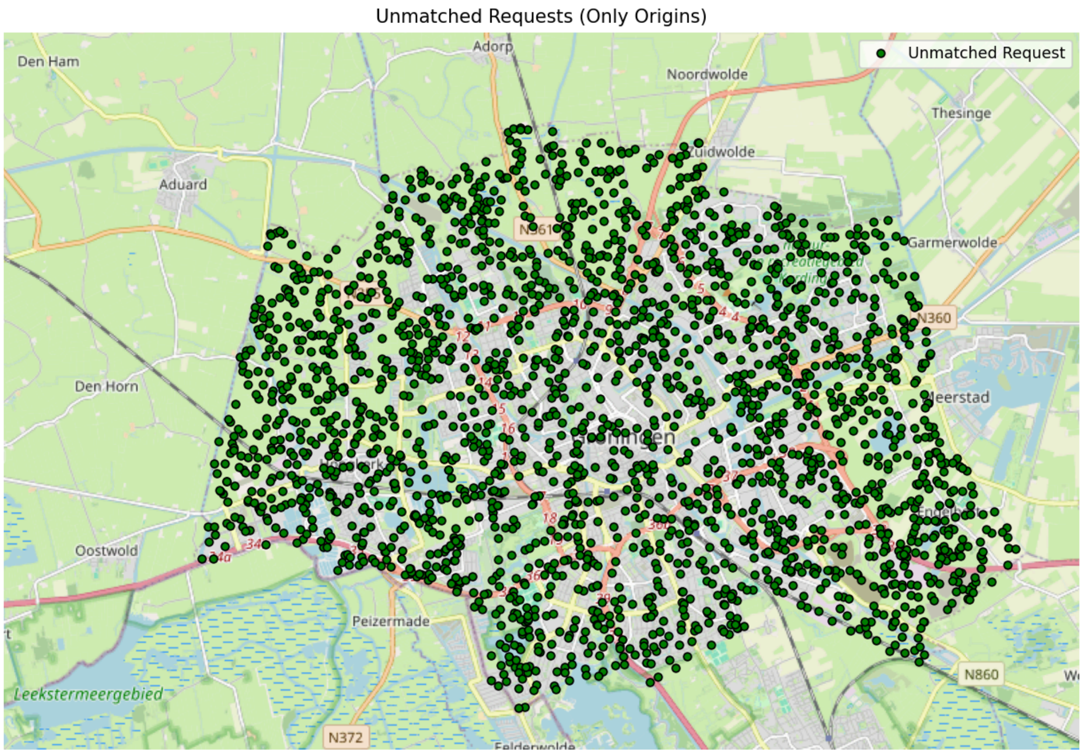

Figure 5: Unmatched results for optimization at w = 0.4km from random sample

Including unmatched request origins

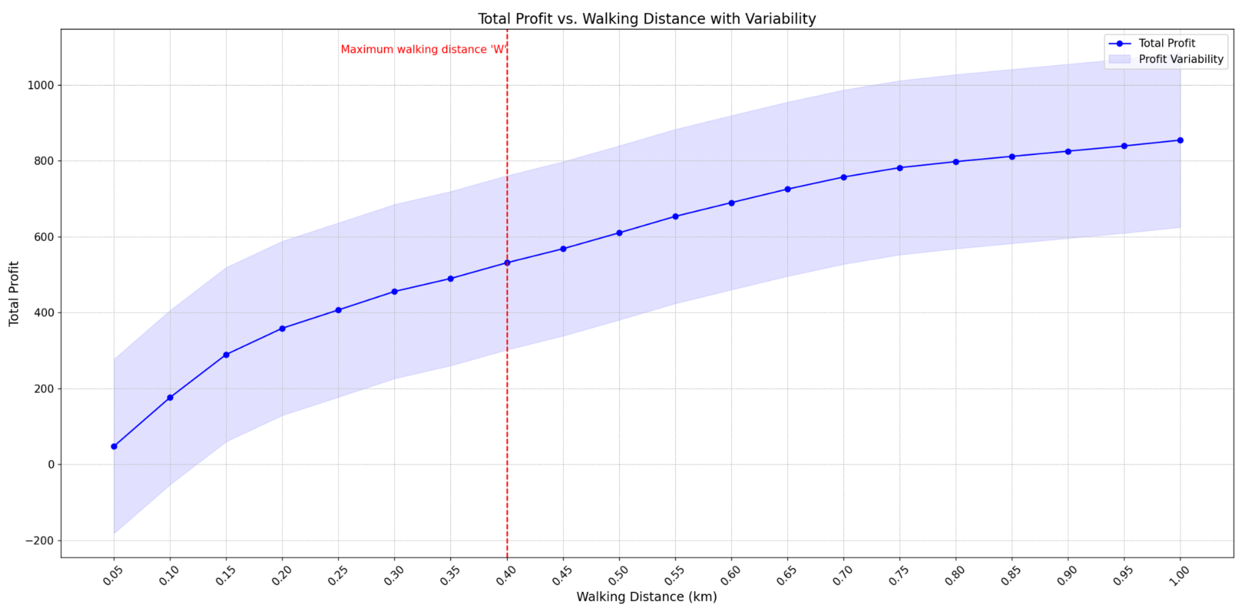

Figure 6: Correlation between Total Profits and Walking Distance

Including Standard Variation and Current Max w markdown

For a **set** of:

- J: Set of carsharing requests
- C: Set of cars available in the fleet

With **parameters**:

- O_j : Origin location of customer request $j \in J$
- d_j: Destination location of request $j \in J$
- w: Maximum walking distance a customer is willing to walk to pick up a car (400 meters)
- π_j: Profit if request j is served, calculated based on rental duration (€0.19 per started minute)
- p_c: Current parking position of car $c \in C$, immediately available for pick-up

And **decision variable**:

- x_{cj}: Binary variable, 1 if request j is assigned to car c, and 0 otherwise

Objective: Maximize total profit

$$\text{Maximize } Z = \sum_{j \in J} \sum_{c \in C} \pi_j \cdot x_{cj}$$

Constraint 1: Each request j is assigned to at most one car

$$\sum_{c \in C} x_{cj} \leq 1 \quad \forall j \in J$$

Constraint 2: Each car c has at most one request assigned

$$\sum_{j \in J} x_{cj} \leq 1 \quad \forall c \in C$$

Constraint 3: A request j is only assigned to car c if the car's current position p_c is within the maximum walking distance w from the customer's current position o_j

$$x_{cj} = 0 \quad \text{if distance}(p_c, o_j) > w \quad \forall j \in J, c \in C$$

Figure 7: Optimization Mathematical Model

ANNEX B: Code

```
"""
Assignment DAPOM - Car Sharing Analysis case
Guillermo Gil de Avalle Bellido S5787084
"""

# Importing libraries. Some may appear unused due to the relevant code
commented out.
from elasticsearch import Elasticsearch, helpers  # Helpers imported twice to
ensure only one doc
from elasticsearch.helpers import scan, bulk
from loaders import ingest_json_file_into_elastic_index
from datetime import datetime
from gurobipy import Model, GRB
from geopy.distance import geodesic
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import smopy
import json
```

```
# Connecting to Elasticsearch and defining initial variables
es    =    Elasticsearch([{"host":    "localhost",    "port":    9200,    "scheme":
"http"}]).options(ignore_status=[400, 405])
index_name = "car_sharing"
data_path                                                                          =
"C:/Users/guill/Desktop/DAPOM/Assignment/Assignment/request_data.json"


# PART 1: READ AND INGESTION
# INGESTING JSON FILE
# Define mapping for the ingestion
settings = {
   'settings': {
       "number_of_shards": 3
   },
   'mappings': {
       'properties': {
                    'origin_datetime':  {'type':  'date',  "format":  "yyyy-MM-dd
HH:mm:ss"},
                 'destination_datetime':  {'type':  'date',  "format":  "yyyy-MM-dd
HH:mm:ss"},
           'origin_location': {'type': 'geo_point'},
           'destination_location': {'type': 'geo_point'},
           'request_nr': {'type': 'integer'},
           'weekend': {'type': 'integer'}
       }
   }
}

# Create index and ingest data using loaders.py function
"""
es.indices.create(index=index_name, body=settings)
ingest_json_file_into_elastic_index(data_path,          es,          index_name,
buffer_size=5000)
print("Index created and ingested")
"""


# Check first five lines to check whether ingestion is correct
"""
print("Sample    documents:",    es.search(index=index_name,    body={"query":
{"match_all": {}}, "size": 5}))
"""
# Commented because check was successful
```

```
# Determine the initial number of entries to compare with the number of entries
after deleting the incorrect ones
es.indices.refresh(index=index_name)
print(f"Number of entries = {es.count(index=index_name)['count']}")

# CLEANING DATA
batch_size = 10000
batch = []

all_query = {
   "query": {
       "bool": {
           "must": {
               "match_all": {}
           }
       }
   }
}


# Function to calculate distance using origin lon/lat and destination lon/lat
def calculate_distance(origin, destination):
   # Using Geopy library
   result = geodesic(origin, destination)
   return result


# Function to swipe longs and lats
def swap_lon_lat(wrong_location):
   right_location = [wrong_location[1], wrong_location[0]]
   return right_location


# Function to calculate time difference in hours from origin & destination
datetime
def calculate_time_difference_in_hours(origin, destination):
   time_difference = destination - origin
    time_difference_in_hours = time_difference.total_seconds() / 3600  # The
number of seconds in an hour
   return time_difference_in_hours


# Function to calculate speed in kms using formula speed = distance / time.
```

```
def calculate_speed(distances, time):
   if time > 0:
       speed = distances / time
   else:
       speed = 0
   return speed


# Alter timeout for bulk operations (according to new practices of
Elasticsearch)
es_with_options = es.options(request_timeout=30)

# Application - Commented out, due to successful outcome
"""
# Iterating over each instance and loading the main fields in variables
for hit in scan(client=es, index=index_name, query=all_query):
   origin_location = hit['_source']['origin_location']
   destination_location = hit['_source']['destination_location']
       origin_datetime = datetime.strptime(hit['_source']['origin_datetime'],
"%Y-%m-%d %H:%M:%S")
                                             destination_datetime =
datetime.strptime(hit['_source']['destination_datetime'], "%Y-%m-%d %H:%M:%S")

    # Check whether location and datetime are different. If so, set delete = 0,
otherwise set delete = 1
       if origin_location != destination_location and origin_datetime !=
destination_datetime:

       # Swipe longs and lats
       origin_location = swap_lon_lat(origin_location)
       destination_location = swap_lon_lat(destination_location)

       # Calculate distance
                          distance_km = calculate_distance(origin_location,
destination_location).km

       # Calculate time difference in hours
              time_hours = calculate_time_difference_in_hours(origin_datetime,
destination_datetime)

       # Calculate speed
       speed_kmh = calculate_speed(distance_km, time_hours)

       # Check whether speed is reasonable. If not, set delete = 1
```

```
        if 5 <= speed_kmh <= 50:  # For urban areas, speed typically ranges an
avg of 5km/h to 50km/h
            # Update the document with distance, time, and speed, and set delete
= 0
            update_request = {
                "_op_type": "update",
                "_index": index_name,
                "_id": hit['_id'],
                "doc": {
                    "distance_km": distance_km,
                    "time_hours": time_hours,
                    "speed_kmh": speed_kmh,
                    "demand": 1,
                    "delete": 0
                }
            }
            # Update request to batch
            batch.append(update_request)
        else:
            # Update the document with distance, time, and speed, and set delete
= 1
            update_request = {
                "_op_type": "update",
                "_index": index_name,
                "_id": hit['_id'],
                "doc": {
                    "distance_km": distance_km,
                    "time_hours": time_hours,
                    "speed_kmh": speed_kmh,
                    "demand": 1,
                    "delete": 1
                }
            }
            # Add update request to batch
            batch.append(update_request)
            if len(batch) >= batch_size:
                bulk(client=es_with_options, actions=batch)
                batch = []
                print("Sent to elasticsearch")

if batch:
   bulk(client=es_with_options, actions=batch)

print("Distance, time, and speed calculations and updates completed.")
```

```
# Query to delete all element in which delete = 1
delete_query = {
 "query": {
   "term": {
     "delete": {
       "value": 1
     }
   }
 }
}

es.delete_by_query(index=index_name,                         body=delete_query,
wait_for_completion=True)

# Determine the number of entries after deleting the incorrect ones
print(f"Number of entries = {es.count(index=index_name)['count']}")
"""

# PART 2: PLOTTING TIME SERIES OF DAILY DEMAND
# Define query for aggregation in order to plot time series of daily demand
aggregation_query = {
   "size": 0,
   "aggs": {
       "daily_demand": {
           "date_histogram": {
               "field": "origin_datetime",
               "calendar_interval": "day"
           }
       }
   }
}

# Execute the query for daily demand
aggregation_response = es.search(index=index_name, body=aggregation_query)

# Parse the response to get the data for plotting
dates = []
demand = []
for bucket in aggregation_response['aggregations']['daily_demand']['buckets']:
   dates.append(bucket['key_as_string'])
   demand.append(bucket['doc_count'])

# Convert dates to pandas datetime, which slices better and works better with
matplotlib
dates = pd.to_datetime(dates)
```

```
# Plot the time series
plt.figure(figsize=(10, 6))
plt.plot(dates, demand, marker='o', markersize=4, linestyle='-', color='b')
plt.xticks(rotation=45)  # Rotates x labels 45 degrees for better readability
plt.tight_layout()  # Adjust plots to give optimal spacing between each other
plt.title('Daily Demand Over the Past Year')
plt.xlabel('Date')
plt.ylabel('Demand')
plt.show()

# PART 3: DEMAND FORECASTING
# VISUALIZE DEMAND IN HOURLY BASIS
# Query hourly data for weekdays
hourly_query = {
    "query": {
        "bool": {
            "must": [
                {"term": {"weekend": 0}}  # Only consider working days
            ]
        }
    },
    "aggs": {
        "hourly_demand": {
            "date_histogram": {
                "field": "origin_datetime",
                "calendar_interval": "hour",
                "min_doc_count": 1  # Making sure the instance has data to show
            },
            "aggs": {
                "demand": {"value_count": {"field": "request_nr"}}
            }
        }
    },
    "size": 0
}

# Execute the query
hourly_response = es.search(index=index_name, body=hourly_query)

# Process the results & add into new lists
hours = []
demands = []
for bucket in hourly_response['aggregations']['hourly_demand']['buckets']:
    hour = datetime.utcfromtimestamp(bucket['key'] / 1000).hour
```

```
    # Convert to UTC from UNIX time. Divide by 1000 to convert to seconds.
Select only hours.
   demand = bucket['demand']['value']
   hours.append(hour)
   demands.append(demand)

# Create a DataFrame (long format) for easier manipulation
df_long_format = pd.DataFrame({'Hour': hours, 'Demand': demands})

# Group by hour and calculate mean, std and overall mean demand
stats  =  df_long_format.groupby('Hour')['Demand'].agg(['mean',  'std',  'min',
'max']).reset_index()
overall_mean_demand = stats['mean'].mean()

plt.figure(figsize=(14, 8))

# Plotting mean demand as bars
plt.bar(stats['Hour'], stats['mean'], color='skyblue', label='Mean Demand')

# Adding error bars for standard deviation directly on the bars
plt.errorbar(stats['Hour'],   stats['mean'],   yerr=stats['std'],   fmt='none',
ecolor='gray',
            capsize=5, label='Standard Deviation')

# Min and Max values as points on top of each bar
plt.scatter(stats['Hour'],   stats['min'],   color='red',   marker='_',   s=100,
label='Min Demand', zorder=5)
plt.scatter(stats['Hour'],   stats['max'],   color='blue',   marker='_',   s=100,
label='Max Demand', zorder=5)

# Overall mean demand as a horizontal line
plt.axhline(y=overall_mean_demand,        color='green',        linestyle='--',
label='Overall Mean Demand', zorder=4)

plt.title('Hourly Demand on Working Days (with Mean, Standard Deviation, Min,
and Max)')
plt.xlabel('Hour of the Day')
plt.ylabel('Demand')
plt.xticks(np.arange(stats['Hour'].min(), stats['Hour'].max() + 1, 1.0))
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Generate legend
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys())
```

```
plt.show()

# EXPECTED NUMBER OF REQUESTS ON A WORKING DAY
workday_query = {
    "size": 0,
    "query": {
        "bool": {
            "must": [
                {"term": {"weekend": 0}}  # Only consider working days
            ]
        }
    },
    "aggs": {
        "daily_demand": {
            "date_histogram": {
                "field": "origin_datetime",
                "calendar_interval": "day",
                  "min_doc_count": 1   # Ensure we only count days with at least
one request
            }
        }
    }
}

# Execute the query
workday_response = es.search(index=index_name, body=workday_query)

# Extract the daily demand counts
daily_demands = [bucket['doc_count'] for bucket in
workday_response['aggregations']['daily_demand']['buckets']]

# Calculate the average daily demand
average_daily_demand = np.mean(daily_demands)

# Increase by 30% to account for unsatisfied demand
adjusted_average_daily_demand = round(average_daily_demand * 1.3, 0)

print(f"Average daily demand (adjusted for unsatisfied demand):
{adjusted_average_daily_demand}")


# VISUALIZE A SAMPLE FROM 3(a) AND 3(b)
# Using average daily demand as the size of our sample, as calculated in 3b
sample_size = int(adjusted_average_daily_demand)
```

```
# Selecting a random sample of the data
random_sample_df = df_long_format.sample(n=sample_size, random_state=42)
# random_state=42 to ensure that the same sample is selected every time

# Increase the 'Demand' column values by 30%
random_sample_df['Adjusted_Demand'] = random_sample_df['Demand'] * 1.30

# Group by hour and calculate statistics on Adjusted_Demand
stats_random_df                                                                  =
random_sample_df.groupby('Hour')['Adjusted_Demand'].agg(['mean',  'std',  'min',
'max']).reset_index()
overall_mean_demand_adjusted_random_df = stats_random_df['mean'].mean()

plt.figure(figsize=(14, 8))

# Plotting mean adjusted demand as bars
plt.bar(stats_random_df['Hour'],    stats_random_df['mean'],    color='skyblue',
label='Mean Adjusted Demand')

# Adding error bars for standard deviation of the adjusted demand
plt.errorbar(stats_random_df['Hour'],                     stats_random_df['mean'],
yerr=stats_random_df['std'], fmt='none', ecolor='gray',
            capsize=5, label='Standard Deviation')

# Min and Max values as points on top of each bar for adjusted demand
plt.scatter(stats_random_df['Hour'],     stats_random_df['min'],     color='red',
marker='_',
           s=100, label='Min Adjusted Demand', zorder=5)
plt.scatter(stats_random_df['Hour'],     stats_random_df['max'],    color='blue',
marker='_',
           s=100, label='Max Adjusted Demand', zorder=5)

# Overall mean adjusted demand as a horizontal line
plt.axhline(y=overall_mean_demand_adjusted_random_df, color='green',
           linestyle='--', label='Overall Mean Adjusted Demand', zorder=4)

plt.title('Hourly  Adjusted  Demand  on  Working  Days  (Random  Sample)  -  Mean,
Standard Deviation, Min, and Max')
plt.xlabel('Hour of the Day')
plt.ylabel('Adjusted Demand')
plt.xticks(np.arange(stats_random_df['Hour'].min(),
stats_random_df['Hour'].max() + 1, 1.0))
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
# Generate legend
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys())

plt.show()

# Print the number of instances in random_sample_df
print("Number of instances in random_sample_df:", len(random_sample_df))


# PART 4: OPTIMIZATION MODEL
# Load car data
cars_location_df = []
with
open("C:/Users/guill/Desktop/DAPOM/Assignment/Assignment/car_locations.json",
'r') as file:
    for line in file:
        car = json.loads(line)
        # The geo-data for the first 200 lines is swapped. Cleaning it...
        if len(cars_location_df) < 200:
                                        car['start_location']  =  [car['start_location'][1],
car['start_location'][0]]
        cars_location_df.append(car)

# Load again required request_data.json from Elasticsearch
query = {
    "size": sample_size,
    "query": {
        "match_all": {}
    }
}

response = es.search(index=index_name, body=query)
requests_data_df = []

# Format and clean it
for hit in response['hits']['hits']:
    request = hit['_source']
    origin_location = tuple(request['origin_location'])
    destination_location = tuple(request['destination_location'])
    origin_location = swap_lon_lat(origin_location)
    destination_location = swap_lon_lat(destination_location)

    requests_data_df.append({
```

```
        "request_id": hit['_id'],
        "origin_location": origin_location,
        "destination_location": destination_location,
        "walking_distance": 0.4,  # Assuming 400 meters for all requests
        "profit": 0
    })

print(f"The sample size for optimization is: {len(requests_data_df)}")

# Define parameters
average_speed_kmh = 50
revenue_rate = 0.19
maximum_distance = 0.4

# Define as a list as it will receive more values later
profits = []
matched_assignments = []


# Pre-emptively populate profit to avoid errors (based on formula distance /
speed)
for request in requests_data_df:
            distance_km    =    calculate_distance(request['origin_location'],
request['destination_location']).kilometers
    travel_time_hours = distance_km / average_speed_kmh
    travel_time_minutes = np.ceil(travel_time_hours * 60)
    request['profit'] = travel_time_minutes * revenue_rate


# Function to check the pre-process compatibility and distance for any given w
def          calculate_compatibility_and_distance(car_data,         request_data,
walking_distances):
    compatibility_for_walking_distances = {w: set() for w in walking_distances}
      distances  =  {}   #  This  will  store  the  calculated  distances  to  avoid
recalculating

    for car in car_data:
        car_id = car['car_id']
        car_start_location = tuple(car['start_location'])

        for request in request_data:
            request_id = request['request_id']
            request_origin_location = tuple(request['origin_location'])

            # Key for distances dictionary
```

```
            distance_key = (car_id, request_id)

            # Calculate distance if not already done
            if distance_key not in distances:
                 distances[distance_key] = calculate_distance(car_start_location,
request_origin_location).km

            # Check compatibility for each walking distance
            for w in walking_distances:
                if distances[distance_key] <= w:
                             compatibility_for_walking_distances[w].add((car_id,
request_id))
   return compatibility_for_walking_distances, distances


# Function that calculates maximization model for any given w
def optimize_model(car_data, request_data, w,
compatibility_for_walking_distances):
   model = Model("Car_sharing")

   num_cars = len(car_data)
   num_requests = len(request_data)

   x = model.addVars(num_cars, num_requests, vtype=GRB.BINARY, name="assign")

   model.setObjective(
       sum(x[c, j] * request_data[j]['profit']
           for c, car in enumerate(car_data)
           for j, request in enumerate(request_data)
                               if (car['car_id'], request['request_id']) in
compatibility_for_walking_distances[w]),
       GRB.MAXIMIZE)

   for j, request in enumerate(request_data):
       model.addConstr(sum(x[c, j] for c, car in enumerate(car_data)
                                 if (car['car_id'], request['request_id']) in
compatibility_for_walking_distances[w]) <= 1)

   for c, car in enumerate(car_data):
       model.addConstr(
           sum(x[c, j] for j, request in enumerate(request_data)
                                 if (car['car_id'], request['request_id']) in
compatibility_for_walking_distances[w]) <= 1)

   model.optimize()
```

```
    return model


# Precompute compatibility and distances for only 0.4km
compatibility_for_walking_distances,                  distances                  =
(calculate_compatibility_and_distance(cars_location_df,        requests_data_df,
[0.4]))

# Processing the model only for 0.4km
model  =  optimize_model(cars_location_df,  requests_data_df,  maximum_distance,
compatibility_for_walking_distances)
if model.status == GRB.OPTIMAL:
    profits.append(model.getObjective().getValue())

    matched_assignments = []
    for c, car in enumerate(cars_location_df):
        for j, request in enumerate(requests_data_df):
                              if   (car['car_id'],   request['request_id'])   in
compatibility_for_walking_distances[0.4]:
               if model.getVarByName(f"assign[{c},{j}]").X > 0.5:
                        print(f"Request  {request['request_id']}  assigned  to  car
{car['car_id']} in 0.4km walking distance")
                   matched_assignments.append((c, j))

     print(f"The  number  of  matched  requests  for  0.4  km  walking  distance  is
{len(matched_assignments)}")
else:
    profits.append(0)
    print(f"Optimization failed for 0.4km")

# Create lists for matched_assignments for printing
matched_lats = []
matched_lons = []

for car_index, request_index in matched_assignments:
    request = requests_data_df[request_index]
    matched_lats.append(request['origin_location'][0])
    matched_lons.append(request['origin_location'][1])

# Map visualization for matched requests and their connections to cars
matched_map        =        smopy.Map((min(matched_lats),        min(matched_lons),
max(matched_lats), max(matched_lons)), z=12)
ax = matched_map.show_mpl(figsize=(10, 10))

# Plotting cars
```

```
for c, car in enumerate(cars_location_df):
            cx,     cy     =     matched_map.to_pixels(car['start_location'][0],
car['start_location'][1])
    ax.plot(cx, cy, 'r^', markersize=8, markeredgecolor='k', label='Cars' if c
== 0 else "")

# Plotting matched requests and drawing lines
for c, j in matched_assignments:
   request = requests_data_df[j]
          ox,    oy    =    matched_map.to_pixels(request['origin_location'][0],
request['origin_location'][1])
        dx,   dy   =   matched_map.to_pixels(request['destination_location'][0],
request['destination_location'][1])
     cx,  cy  =  matched_map.to_pixels(cars_location_df[c]['start_location'][0],
cars_location_df[c]['start_location'][1])
     ax.plot(ox,  oy,  'go',  markersize=5,  markeredgecolor='k',  label='Matched
Request Origins' if c == 0 else "")
     ax.plot(dx,  dy,  'bo',  markersize=5,  markeredgecolor='k',  label='Matched
Request Destinations' if c == 0 else "")
    ax.plot([cx,  ox],  [cy,  oy],  '-',  color='k')   #  Line  from  car  to  request
origin

# Adding a title and legend
ax.set_title("Matched Requests and Their Assigned Cars")
handles, labels = ax.get_legend_handles_labels()
by_label = dict(zip(labels, handles))
ax.legend(by_label.values(), by_label.keys(), loc='upper left')

plt.show()

# Extract request indices from matched_assignments
matched_request_indices    =    {request_index    for    _,    request_index    in
matched_assignments}

#  Calculate  unmatched_requests_indices  by  subtracting  matched_request_indices
from the set of all request indices
unmatched_requests_indices      =      set(range(len(requests_data_df)))      -
matched_request_indices

# Now you can proceed with calculating unmatched_lats and unmatched_lons
unmatched_lats    =    [requests_data_df[j]['origin_location'][0]    for    j    in
unmatched_requests_indices]
unmatched_lons    =    [requests_data_df[j]['origin_location'][1]    for    j    in
unmatched_requests_indices]
```

```
# Check if there are any unmatched requests to plot
if unmatched_lats and unmatched_lons:
        unmatched_map   =   smopy.Map((min(unmatched_lats),   min(unmatched_lons),
max(unmatched_lats), max(unmatched_lons)), z=12)
   ax = unmatched_map.show_mpl(figsize=(10, 10))

   # Plot the origins of unmatched requests
   for lat, lon in zip(unmatched_lats, unmatched_lons):
       ux, uy = unmatched_map.to_pixels(lat, lon)
       ax.plot(ux, uy, 'go', markersize=5, markeredgecolor='k',
               label='Unmatched Request' if lat == unmatched_lats[0] else "")

   handles, labels = ax.get_legend_handles_labels()
   by_label = dict(zip(labels, handles))
   ax.legend(by_label.values(), by_label.keys())
   ax.set_title("Unmatched Requests (Only Origins)")
   plt.show()


# PART 5 - CORRELATION BETWEEN WALKING DISTANCE AND PROFIT
# Define walking distances to iterate over
walking_distances  =  [0.05,  0.1,  0.15,  0.2,  0.25,  0.3,  0.35,  0.4,  0.45,  0.5,
0.55,
                    0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]

# Restart profits
profits = []

# Precompute compatibility and distances
compatibility_for_walking_distances,                 distances                 =
calculate_compatibility_and_distance(cars_location_df,         requests_data_df,
walking_distances)

# Execute modeling and calculate different profits
for w in walking_distances:
          model    =    optimize_model(cars_location_df,    requests_data_df,    w,
compatibility_for_walking_distances)
   if model.status == GRB.OPTIMAL:
       profits.append(model.getObjective().getValue())
       matched_assignments = []
       for c, car in enumerate(cars_location_df):
           for j, request in enumerate(requests_data_df):
                                if   (car['car_id'],   request['request_id'])   in
compatibility_for_walking_distances[w]:
                   if model.getVarByName(f"assign[{c},{j}]").X > 0.5:
```

```
                    print(f"Request {request['request_id']} assigned to car
{car['car_id']} in {w} km walking distance")
                    matched_assignments.append((c, j))

        print(f"The number of matched requests in {w} km walking distance is
{len(matched_assignments)}")
    else:
        profits.append(0)
        print(f"Optimization failed for walking distance {w}")

# Calculate the overall profit variability as the standard deviation of the
given profits
profit_variability = np.std(profits)

# Calculate upper and lower bounds for the shaded region representing
variability
upper_bound = [profit + profit_variability for profit in profits]
lower_bound = [profit - profit_variability for profit in profits]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(walking_distances, profits, marker='o', linestyle='-', color='blue',
label='Total Profit')
plt.fill_between(walking_distances, lower_bound, upper_bound, color='blue',
alpha=0.1, label='Profit Variability')

# Adding enhancements for visual appeal
plt.title('Total Profit vs. Walking Distance with Variability', fontsize=16)
plt.xlabel('Walking Distance (km)', fontsize=14)
plt.ylabel('Total Profit', fontsize=14)
plt.xticks(walking_distances, rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', linewidth=0.5)
plt.legend(fontsize=12)
plt.tight_layout()  # Adjust layout to make sure everything fits without
overlapping

# Add a vertical line at 0.4 km
plt.axvline(x=0.4, color='red', linestyle='--', label="Maximum walking distance
'W'")
plt.text(0.4, max(upper_bound), "Maximum walking distance 'W'",
horizontalalignment='right', color='red', fontsize=12)

plt.show()
```

ANNEX C: ChatGPT Usage

As discussed with the professors, only the prompts asked to ChatGPT are provided. However, as these prompts may be incomplete, here is the link to the actual chat.

1. Create a code in elasticsearch to ingest a big json file into an index called car_sharing.
2. I got this errors DeprecationWarning: Passing transport options in the API method is deprecated. Use 'Elasticsearch.options()' instead.
3. I tried this code in order to check whether there were docs within the created index but the result is: Sample documents: [] Total documents in the index 'car_sharing': 0- what is going on?
4. Also got this error -DeprecationWarning: Received 'size' via a specific parameter in the presence of a 'body' parameter, which is deprecated and will be removed in a future version.
5. The results still show that the index has been created with the right mapping, but no docs have been indexed: Sample documents: [] Total documents in the index 'car_sharing': 0 Settings: {'car_sharing': {'settings': {'index': {'routing': {'allocation': {'include': {'_tier_preference': 'data_content'}}}, 'number_of_shards': '3', 'provided_name': 'car_sharing', 'creation_date': '1710612559669', 'number_of_replicas': '1', 'uuid': 'V2fR3lBXSAWqIs2s0DEE6g', 'version': {'created': '8500010'}}}} What could i do next?
6. A sample instance of the request_data.json file would be: {"request_nr":0,"origin_datetime":"2023-01-01 00:34:19","weekend":1,"origin_location":[6.5176403369,53.2519585674],"destination_datetime":"2023-01-01 00:39:30","destination_location":[6.5384431178,53.216220107]}
7. Error during data ingestion: 'items' - what does this mean?
8. for hit in scan(client=es, index=index_name, query=query): what is this doing? is this similar to establishing settings like this: settings = { ... }?
9. Applying the right format for each field based for all instances using match_all query defined before, what would you input in those places?
10. Fix this if statement used to check whether the speed in kmh is realistic if speed_kmh > 0 and speed_kmh < 9999.
11. Find my code attached, and im getting this weird error I dont understand.
12. What does 'DeprecationWarning: Passing transport options in the API method is deprecated. Use 'Elasticsearch.options()' instead. bulk(client=es, actions=batch, request_timeout=30) # Adjust request timeout as needed' mean?
13. What is request timeout?
14. Check whether location and datetime are different. If so, set delete = 0, otherwise set delete = 1 ...
15. Is append the function for deletion? Can you update the previous code to delete the records where either the location or the datetime are the same?
16. Suggest a code for performing step 2 of the case, i.e., After cleaning the data set, you want to get a feeling for the information contained in it.
17. What is the benefit of converting dates to pandas datetime? can you do it without that?
18. What is plt.xticks(rotation=45) plt.tight_layout() (short answer)?
19. what is the color code for black?
20. in the uniletter system of matplotlib.

21. Is the previous data already loaded into a dataframe? Can it be used to determine the demand pattern observed on a typical working day?
22. Yes but wasn't the data already added to a dataframe called dates before? Can't that be reutilized?
23. Yes but I did not create demand_data before.
24. I want to aggregate the average demand by the hour by avg, not by mean (only provide the code).
25. Wouldn't it be easier to query elasticsearch for all instances where the 'weekend' field equals 0 (working dates) and then use date_histogram to pull the data for the hours? how could this work?
26. How to then get the hour in the right format when using the data? could I use maybe the utcfromtimestamp function?
27. Different question - How can I reduce the marker size on my matplotlib plot?
28. Can you alter this query to determine the expected number of requests on a working day? Use date_histogram as well.
29. This is the code I use to determine the demand pattern observed on a typical working day. You decide to do so by visualizing demand on an hourly basis; in addition, you also plot appropriate representations of statistical measures of hourly demand. However, at the moment, I do not print any statistical measures. Could you recommend statistical measures to print (e.g., standard deviation) and suggest changes in the code to include them
30. I have done so far till part two of the assignment - how do it do part 3 (directly generate code, no explanation)?
31. The query to select days of the week is wrong - You need to select any day where weekend parameter equals 0.
32. Do you need to do this v"range": {"origin_datetime": {"gte": "2023-01-01", "lt": "2024-01-01"}? what is this for?
33. "min_doc_count": 0 what is this?
34. Is the variables with a string within [] a tuple?
35. Hour = datetime.utcfromtimestamp(bucket['key'] / 1000).hour explain this?
36. How can I reduce the marker size on my matplotlib plot?
37. The query to select days of the week is wrong - You need to select any day where weekend parameter equals 0.
38. Do you need to do this v"range": {"origin_datetime": {"gte": "2023-01-01", "lt": "2024-01-01"}? what is this for?
39. "min_doc_count": 0 what is this?
40. Is the variables with a string within []a tuple?
41. Hour = datetime.utcfromtimestamp(bucket['key'] / 1000).hour explain this?
42. How can i reduce the marker size on my matplotlib plot?
43. How to draw a sample out of elasticsearch as defined in exercise 3c?
44. This is my code - iterate over a random sample the size of 'average_daily_demand' to show hours and demands as captured in 3a.
45. Can you use random to make sure the sample is random?
46. What is size = 10000? how is this related to 3(a)? how is this using random?
47. Can you give me the corrected code?

48. Generate the mathematical formula for the optimization problem in part 4, where the values, decision variables, objectives, and constraints are clear. Add a short explanation to why you have selected each.
49. Import this file to my python and separate both car id and start location.
50. Read a file called car_locations.json into a df and separate the different elements. an example of how the file looks inside: {"car_id":0,"start_location":[6.6160272067,53.234156008]}.
51. Generate python code for reading a file called car_locations.json into a df and separate the different elements. an example of how the file looks inside: {"car_id":0,"start_location":[6.6160272067,53.234156008]}.
52. The file path is C:\Users\guill\Desktop\DAPOM\Assignment\Assignment. Use /.
53. Remember the mathematical formulation that you did before?
54. Write the Gurobi optimization for this aforementioned model having in mind the car_locations.json file and the request_data.json (this file can't be updated due to the size, but a usual line looks like this: {"request_nr":1398,"origin_datetime":"2023-01-02 04:21:35","weekend":0,"origin_location":[6.5065144044,53.246767634],"destination_datetime":"2023-01-02 04:38:12","destination_location":[6.6358961634,53.197183205]}) as our two main sources, as well as the previous code that I've written (main3.py).
55. The car df is called car_locations_df, and it has three columns (car_id, longitude, latitude). The request data is loaded in Elasticsearch and this is the mapping settings = { 'settings': { "number_of_shards": 3 }, 'mappings': { 'properties': { 'origin_datetime': {'type': 'date', "format": "yyyy-MM-dd HH:mm:ss"}, 'destination_datetime': {'type': 'date
56. 'destination_datetime': {'type': 'date', "format": "yyyy-MM-dd HH:mm:ss"}, 'origin_location': {'type': 'geo_point'}, 'destination_location': {'type': 'geo_point'}, 'request_nr': {'type': 'integer'}, 'weekend': {'type': 'integer'} } }. Feel free to query Elasticsearch to obtain the data. Change the previous code to account for these names.
57. Can you query directly the data from Elasticsearch you need instead of doing this?
58. All_query = { "query": { "bool": { "must": { "match_all": {} } } } } can I not use this query?
59. I already have a dataframe sample called random_sample_df. However, this dataframe only includes hours, demand, and adjusted demand (as you can see in my code). How could I add the relevant parameters for this optimization model?
60. I don't want to query Elasticsearch, because I already did. Please look at my code, and suggest a way of expanding random_sample_df with the relevant parameters.
61. I need to import the parameters request_nr, origin_datetime, and origin_location into the random_sample_df.
62. Give me an Elasticsearch query to request request_nr, origin_datetime, origin_location.
63. Merge request_df with the data in random_sample_df. Make sure it has all columns on random_sample_df, and only the matches on request_df.
64. Modify this query to also request the request_nr, origin_datetime, origin_location (all unaltered).
65. How can you add the request_nr when you are creating the dataframe?
66. Import smopy. From elasticsearch import Elasticsearch. From gurobipy import Model, GRB. From geopy.distance import geodesic. Import matplotlib.pyplot as plt. Import numpy as np. Import json. Initialize Elasticsearch client. Define the optimization model. Load car data. Average speed in km/h and revenue rate in € per started minute. Elasticsearch query to fetch sample_size number of requests. Fetching data from Elasticsearch. Decision variables. Objective function: Maximize total profit. Constraints. Each request is assigned to at most one car. Each car is assigned to at

most one request. Compatibility constraint: A request is only assigned to a car if the car’s position is within the walking distance from the request's origin. Solve the model. Extract and print the solution. Let's say the bounding box is the min and max of the car and request coordinates. Create a map using the bounding box. Plot the cars on the map. Plot the requests on the map. Extract and print the solution. Draw lines for assignments. Add legend.

67. Why is this code giving this error?
68. I'm now getting this. Set parameter Username. Academic license - for non-commercial use only - expires 2025-03-07.
69. How can I zoom in on the map plot created? At the moment, I see the whole of Europe, when the requests are all within the same city.
70. At the bottom of the result, it says 'Lowered zoom level to keep map size reasonable. (z = 4)' - how can I avoid that?
71. I want to use smopy - is there any way for avoiding their automatic reductions?
72. Specify the map for only the area of Groningen - provide code, no comments.
73. Perfect coordinates! Now make the cars, request origins points, and request destinations points much smaller, and draw lines from the request origins to their respective request destinations.
74. Change the line to grey discontinuous and semitransparent.
75. Remove the transparency - I want full opaque (only the code).
76. What does this error mean "Name 'matched_assignments' can be undefined"?
77. Check my code for error. After that, check its compatibility with the assignment (i.e., have I missed anything?).
78. Remember the mathematical model we discussed for this case? (part 4) Please show again.
79. Generate a list with all the questions I have asked in this chat ever since the start. Don't add any comments. If you run out of space. please ask me to re-prompt you to keep going.