In [ ]:
```
import pandas as pd
import numpy as np

from scipy.stats import norm

import pyomo.environ as pyo

import gurobipy as gp
from gurobipy import GRB

import openpyxl

from icecream import ic
```

In [ ]:
```
###
### RETRIEVE AND CLEAN DATA
###
file_path = "Railway services-2024.xlsx"

try:
    df = pd.read_excel(io=file_path, engine="openpyxl")
    ic(df.head())

except FileNotFoundError:
    print(f"File '{file_path}' not found. Please check the file path.")

# Data cleaning/ enhancing

# Removing unneccesary spaces from 'From' and 'To' columns.
df['From'] = df['From'].str.strip()
df['To'] = df['To'].str.strip()

# Adding an indicator when the trip is located on line 400
indicator_line_400 = np.where(df['Line'] == 400.0, 1, 0)
df["line_400"] = indicator_line_400
```

In [ ]:
```
###
### CREATING THE MODEL
###

##
## Initial values:
##
train_type = [
    "OC",
    "OH"
    ]
cost_train_type = {
    "OC": 260,
    "OH": 210
}
length_train_type = {
    "OC": 100,
    "OH": 70
    }
capacity_train_type = {
    "OC": 620,
    "OH": 420
```

```
    }

#
# Data manipulated initial values:
#
cross_section = set(
    df['From'].values + df['To'].values
    ) #NOTE: Not used

trips_using_line400 = ((df.loc[df['line_400'] == 1]).index + 1).to_list()


###
### Creation of the Model:
###
def create_general_model():
    m = pyo.ConcreteModel()

    ##
    ## CREATION OF SETS
    ##
    m.trips = pyo.Set(
        initialize = df['Trip']
        )

    m.train_type = pyo.Set(
        initialize = train_type
        )

    ##
    ## CREATION OF PARAMETERS
    ##
    m.cost_train_type = pyo.Param(
        m.train_type,
        initialize = cost_train_type
    )

    m.length_train_type = pyo.Param(
        m.train_type,
        initialize = length_train_type
        )

    m.capacity_train_type = pyo.Param(
        m.train_type,
        initialize = capacity_train_type
        )

    #
    # Parameters created using functions:
    #
    m.trips_using_line400 = pyo.Param(
        m.trips,  # Assuming model.cross_section contains all possible cr
        initialize={trip: 1 if trip in trips_using_line400 else 0 for tri
    )

    m.passengers_per_trip = pyo.Param(
        m.trips,
        initialize = {trip: df['Demand(μ)'].loc[trip - 1] for trip in m.t
    )
```

```
    return m


m = create_general_model()
```

In [ ]:
```
###
### ADDING SPECIFICS FOR MODEL1
###

##
## Variable specific for Model1
##
def specific_variables_Model1(m):
    #
    # Index set for allocation variable model1
    #
    m.index_set_allocation = pyo.Set(
        initialize = m.trips * m.train_type
        )

    #
    # Variable for allocation in Model1
    #
    # the variable represents the number of trains per type allocated to
    #
    m.allocation_train_numbers = pyo.Var(
        m.index_set_allocation,
        domain = pyo.NonNegativeIntegers,
        name = 'train_allocation',
        doc = 'The number of trains of a certain type allocated to a cros
    )
    return m

##
## Constraints specific for Model1
##
def specific_constraints_Model1(m):
    #
    # Rule for restricting the combined train length.
    #
    def rule_maximum_length(
            m,
            trip
            ):
        combined_train_length = sum(
            m.length_train_type[(train_type)] * m.allocation_train_number
            for train_type in m.train_type
        )
        return combined_train_length <= 300 - 100 * m.trips_using_line400
    m.constr_maximum_length = pyo.Constraint(
        m.trips,
        rule = rule_maximum_length
        )

    #
    # Rule for enforcing the ability to accomodate all passengers.
    #
    def rule_passenger_limit(
            m,
            trip
```

```
            ):
        available_capacity = sum(
            m.capacity_train_type[(train_type)] * m.allocation_train_numb
            for train_type in m.train_type
        )
        return available_capacity >= int(m.passengers_per_trip[(trip)])
    m.constr_passenger_limit = pyo.Constraint(
        m.trips,
        rule = rule_passenger_limit
        )

    # FIXME: Should this be with [0.75, 1] or with [1, 1.25] Same for the
    #
    # Rules for limiting the 'favorate' train type to be max 0.25% higher
    #
    def rule_difference_between_number_of_train_types1(m):
        return 1 * sum(
                m.allocation_train_numbers[(trip, train_type)]
                for trip in m.trips
                for train_type in m.train_type if train_type == "OC"
            ) <= 1.25 * sum(
                m.allocation_train_numbers[(trip, train_type)]
                for trip in m.trips
                for train_type in m.train_type if train_type == "OH"
            )
    m.constr_difference_between_number_of_train_types1 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types1
    )
    def rule_difference_between_number_of_train_types2(m):
        return 1 * sum(
                m.allocation_train_numbers[(trip, train_type)]
                for trip in m.trips
                for train_type in m.train_type if train_type == "OH"
            ) <= 1.25 * sum(
                m.allocation_train_numbers[(trip, train_type)]
                for trip in m.trips
                for train_type in m.train_type if train_type == "OC"
            )
    m.constr_difference_between_number_of_train_types2 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types2
    )

    return m


##
## Objective rule specific for Model1
##
def specific_objective_Model1(m):
    #
    # Objective rule to minimize the total cost of all trains.
    #
    def obj_minimize_total_cost(m):
        total_cost = sum(
            m.cost_train_type[(train_type)] * sum(
                m.allocation_train_numbers[(trip, train_type)]
                for trip in m.trips
            )
            for train_type in m.train_type
        )
```

```
        return total_cost
    m.objective = pyo.Objective(
        rule=obj_minimize_total_cost,
        sense=pyo.minimize
    )
    return m


##
## Add all variables, constraints and objectives to Model1
##
new_model = create_general_model()
model1 = specific_variables_Model1(m=new_model)
model1 = specific_constraints_Model1(m=model1)
model1 = specific_objective_Model1(m=model1)
```

In [ ]:

```
###
### ADDING SPECIFICS FOR MODEL2
###



#
# Obtain all compositions
#
def get_compositions(bool_all_combinations:bool = True):
    # Define the lengths of each train type
    OH_length = 100
    OC_length = 70

    OH_cap = 420
    OC_cap = 620

    # Define the maximum length of a train
    max_train_length = 300

    trips = df['Trip']

    # Generate all possible combinations of OH and OC trains
    all_combinations = {}
    allowed_combinations_per_trip = {}


    for trip in trips.to_list():
        combination_trip = []
        for num_OH in range(max_train_length // OH_length + 1):
            for num_OC in range(max_train_length // OC_length + 1):

                length = num_OH * OH_length + num_OC * OC_length
                if length <= max_train_length - 100 * df['line_400'].loc[
                    combination = []
                    if num_OH > 0:  # Append 'OH' only if num_OH is great
                        combination.extend(['OH'] * num_OH)
                    if num_OC > 0:  # Append 'OC' only if num_OC is great
                        combination.extend(['OC'] * num_OC)

                    cap = num_OH * OH_cap + num_OC * OC_cap

                    if length != 0 and length <= max_train_length - 100 *
                        if OC_cap * num_OC + OH_cap * num_OH >= df['Deman
                            all_combinations[str(combination)] = [length,
```

```
                                combination_trip.append(str(combination))

            allowed_combinations_per_trip[trip] = combination_trip


        if bool_all_combinations:
            return all_combinations
        else:
            return allowed_combinations_per_trip


    def get_allowed_comp_per_trip_df():

        allowed_comp = pd.DataFrame(index=df['Trip'].tolist(), columns=get_co

        compositions_per_trip = get_compositions(False)
        for trip in get_compositions(False).keys():

            compos_current_trip = compositions_per_trip.get(trip)

            for comp in compos_current_trip:
                allowed_comp.loc[trip, comp] = 1

        return allowed_comp

    ##
    ## Parameter specific for Model2
    ##
    def specific_parameter_Model2(m):
        #
        # Set containing all compositions
        #
        compositions = get_compositions()
        #ic(compositions)

        m.compositions = pyo.Set(initialize = compositions.keys())

        #
        # Set containing the index for the quantification of the number of tr
        #
        m.index_quantify_compositions = pyo.Set(
            initialize= m.compositions * m.train_type
        )

        #
        # Parameter indicating number of 'OC' and 'OH' within a composition
        #
        m.quantify_compositions = pyo.Param(
            m.index_quantify_compositions,
            initialize = {
                (composition, train_type): compositions[composition][2] if tr
                for composition in m.compositions for train_type in m.train_t
            }
        )

        return m


    ##
    ## Variable specific for Model2
```

```
##
def specific_variables_Model2(m):
    #
    # Index set for allocation variable model1
    #
    m.index_set_allocation = pyo.Set(initialize = m.trips * m.composition


    #
    # Variable for allocation in Model1
    #
    # the binary variable indicates the composition used on a trip.
    #
    m.allocation_compositions = pyo.Var(
        m.index_set_allocation,
        domain = pyo.Binary,
        name = "Specific composition on trip",
        doc = "Indicates which composition is used on a trip"
    )

    return m


##
## Constraints specific for Model2
##
def specific_constraints_Model2(m):
    # NOTE: CORRECT
    # Rule that ensures a trip can only have a single composition
    #
    def rule_one_composition_per_trip(m, trip):
        return sum(
            m.allocation_compositions[(trip, composition)]
            for composition in m.compositions
        ) == 1
    m.constr_one_composition_per_trip = pyo.Constraint(
        m.trips,
        rule=rule_one_composition_per_trip
    )

    #
    # Rules for limiting the 'favorate' train type to be max 0.25% higher
    #
    def rule_difference_between_number_of_train_types1(m):
        return 1 * sum(
            m.quantify_compositions[(composition, "OH")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        ) <= 1.25 * sum(
            m.quantify_compositions[(composition, "OC")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        )
    m.constr_difference_between_number_of_train_types1 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types1
    )
    def rule_difference_between_number_of_train_types2(m):
        return 1 * sum(
            m.quantify_compositions[(composition, "OC")] * m.allocation_c
            for composition in m.compositions
```

```
            for trip in m.trips
        ) <= 1.25 * sum(
            m.quantify_compositions[(composition, "OH")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        )
    m.constr_difference_between_number_of_train_types2 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types2
    )

    #
    # Rule for allowed compositions
    #
    allowed_df = get_allowed_comp_per_trip_df()
    def rule_allowed_compos(m, trip, composition):
        return m.allocation_compositions[(trip, composition)] <= allowed_
    m.constr_allowed_compos = pyo.Constraint(
        m.trips,
        m.compositions,
        rule=rule_allowed_compos
    )

    return m



def specific_objective_Model2(m):
    m.cost_train_type.pprint()
    def obj_minimize_total_cost(m):
        return m.cost_train_type["OH"] * sum(
                m.quantify_compositions[(composition, "OH")] * m.allocati
                for composition in m.compositions
                for trip in m.trips
            ) + (m.cost_train_type["OC"]) * sum(
                m.quantify_compositions[(composition, "OC")] * m.allocati
                for composition in m.compositions
                for trip in m.trips
            )
    m.objective = pyo.Objective(rule=obj_minimize_total_cost, sense=pyo.m
    m.objective.pprint()
    return m

new_model = create_general_model()
model2 = specific_parameter_Model2(m=new_model)
model2 = specific_variables_Model2(m=model2)
model2 = specific_constraints_Model2(m=model2)
model2 = specific_objective_Model2(m=model2)
```

In [ ]:

```
###
### ADDING ADDITIONAL REQUIREMENTS FOR MODEL4
###

def get_probs():
    def calc_prob(cap, mu, sigma):
        z_score = (cap - mu) / sigma
        prob = norm.cdf(z_score)
        return prob

    compos = get_compositions()
    probs = pd.DataFrame(index=df['Trip'].tolist(), columns=compos.keys()
```

```
    for trip in df['Trip'].tolist():
        for comp in compos.keys():
            probs.loc[trip, comp] = calc_prob(
                    cap=compos.get(comp)[1],
                    mu=df['Demand(μ)'].loc[trip - 1],
                    sigma=df['Demand(σ)'].loc[trip - 1]
                )

    return probs
probs = get_probs()

##
## Parameter specific for Model4
##
def specific_parameter_Model4(m):
    #
    # Set containing all compositions
    #
    compositions = get_compositions()
    #ic(compositions)

    m.compositions = pyo.Set(initialize = compositions.keys())

    #m.days = pyo.Set(initialize = list(range(1, 251)))

    #
    # Set containing the index for the quantification of the number of tr
    #
    """m.index_quantify_compositions = pyo.Set(
        initialize= m.compositions * m.train_type * m.days
    )"""

    #
    # Parameter indicating number of 'OC' and 'OH' within a composition
    #
    m.quantify_compositions = pyo.Param(
        m.compositions * m.train_type,
        initialize = {
            (composition, train_type): compositions[composition][2] if tr
            for composition in m.compositions for train_type in m.train_t
        }
    )
    m.demand_upperbound = 5000

    return m


##
## Variable specific for Model4
##
def specific_variables_Model4(m):
    #
    # Index set for allocation variable model1
    #
    m.index_set_allocation = pyo.Set(initialize = m.trips * m.composition
    #m.index_set_satisfaction = pyo.Set(initialize = m.trips * m.composit


    #
```

```
    # Variable for allocation in Model3
    #
    # the binary variable indicates the composition used on a trip.
    #
    m.allocation_compositions = pyo.Var(
        m.index_set_allocation,
        domain = pyo.Binary,
        name = "Specific composition on trip on a day",
        doc = "Indicates which composition is used on a trip on a specifi
    )

    return m



##
## Constraints specific for Model4
##
def specific_constraints_Model4(m, random_demand: bool = False):
    # NOTE: CORRECT
    # Rule that ensures a trip can only have a single composition
    #
    def rule_one_composition_per_trip(m, trip):
        return sum(
            m.allocation_compositions[(trip, composition)]
            for composition in m.compositions
        ) == 1
    m.constr_one_composition_per_trip = pyo.Constraint(
        m.trips,
        rule=rule_one_composition_per_trip
    )

    #
    # Rules for limiting the 'favorate' train type to be max 0.25% higher
    #
    def rule_difference_between_number_of_train_types1_4(m):
        return 1 * sum(
            m.quantify_compositions[(composition, "OH")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        ) <= 1.25 * sum(
            m.quantify_compositions[(composition, "OC")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        )
    m.constr_difference_between_number_of_train_types1 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types1_4
    )
    def rule_difference_between_number_of_train_types2_4(m):
        return 1 * sum(
            m.quantify_compositions[(composition, "OC")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        ) <= 1.25 * sum(
            m.quantify_compositions[(composition, "OH")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        )
    m.constr_difference_between_number_of_train_types2 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types2_4
    )
```

```
    #
    # Rule for allowed compositions
    #
    if random_demand:
        allowed_df = get_allowed_comp_per_trip_df(random=True)
    else:
        allowed_df = get_allowed_comp_per_trip_df()

    def rule_allowed_compos(m, trip, composition):
        return m.allocation_compositions[(trip, composition)] <= allowed_
    m.constr_allowed_compos = pyo.Constraint(
        m.trips,
        m.compositions,
        rule=rule_allowed_compos
    )

    #
    # Capacity constraints
    #
    def rule_125days_satisfied(m, trip):
        return sum(
                probs.loc[trip, composition] * m.allocation_compositions[
                for composition in m.compositions
                ) >= 125/250 #FIXME
    m.constr_125days = pyo.Constraint(
        m.trips,
        rule=rule_125days_satisfied
    )
    #m.constr_125days.pprint()
    def rule_81_percent_satisfied(m):
        return sum(
                probs.loc[trip, composition] * m.allocation_compositions[
                for trip in m.trips
                for composition in m.compositions
                ) >= 0.81 * 50000/250
    m.constr_81_persent = pyo.Constraint(
        rule=rule_81_percent_satisfied
    )
    m.constr_81_persent.pprint()

    return m


#
# Objective function for minimizing total costs.
#
def specific_objective_Model4(m):
    #m.cost_train_type.pprint()
    def obj_minimize_total_cost(m):
        return m.cost_train_type["OH"] * sum(
                m.quantify_compositions[(composition, "OH")] * m.allocati
                for composition in m.compositions
                for trip in m.trips
            ) + (m.cost_train_type["OC"]) * sum(
                m.quantify_compositions[(composition, "OC")] * m.allocati
                for composition in m.compositions
                for trip in m.trips
            )
    m.objective = pyo.Objective(rule=obj_minimize_total_cost, sense=pyo.m
```

```
    #m.objective.pprint()

    return m

new_model = create_general_model()
model4 = specific_parameter_Model4(m=new_model)
model4 = specific_variables_Model4(m=model4)
model4 = specific_constraints_Model4(m=model4)
model4 = specific_objective_Model4(m=model4)
```

In [ ]:

```
#
# Model Q5
#


def get_probs(delta: 1):
    def calc_prob(cap, mu, sigma):
        z_score = (cap - mu) / sigma
        prob = norm.cdf(z_score)
        return prob

    compos = get_compositions()
    probs = pd.DataFrame(index=df['Trip'].tolist(), columns=compos.keys()

    for trip in df['Trip'].tolist():
        for comp in compos.keys():
            probs.loc[trip, comp] = calc_prob(
                        cap=compos.get(comp)[1],
                        mu=delta * df['Demand(μ)'].loc[trip - 1],
                        sigma=df['Demand(σ)'].loc[trip - 1]
                    )

    return probs
decreased_probs = get_probs(delta=0.9)
increased_probs = get_probs(delta=1.15)

ic(increased_probs)
ic(decreased_probs)

##
## Variable specific for Model3
##
def specific_variables_Model5(m):
    #
    # Index set for allocation variable model1
    #
    m.index_set_allocation = pyo.Set(initialize = m.trips * m.composition
    #m.index_set_satisfaction = pyo.Set(initialize = m.trips * m.composit

    #
    # Variable for allocation in Model3
    #
    # the binary variable indicates the composition used on a trip.
    #
    m.allocation_compositions = pyo.Var(
        m.index_set_allocation,
        domain = pyo.Binary,
        name = "Specific composition on trip on a day",
        doc = "Indicates which composition is used on a trip on a specifi
    )
```

```
    return m

def specific_parameter_Model5(m):
    #
    # Set containing all compositions
    #
    compositions = get_compositions()
    #ic(compositions)

    m.compositions = pyo.Set(initialize = compositions.keys())

    return m


#
# Constraint
#
def specific_constraints_Model5(m, sol: pd.DataFrame):
    sol = sol.sum(axis=0).transpose()
    #ic(sol)
    compos = get_compositions()

    #
    # Constraints restricting the number of traintypes.
    #
    def rule_number_per_type_OC(m):
        return sum(
            compos.get(compo)[2] * m.allocation_compositions[trip, compo]
                for trip in m.trips
                for compo in m.compositions
            ) <= sum(
                sol.loc[compo] * compos.get(compo)[2]
                for compo in m.compositions
                )
    m.constr_number_per_type_OC = pyo.Constraint(
        rule=rule_number_per_type_OC
    )
    def rule_number_per_type_OH(m):
        return sum(
            compos.get(compo)[3] * m.allocation_compositions[trip, compo]
                for trip in m.trips
                for compo in m.compositions
            ) <= sum(
                sol.loc[compo] * compos.get(compo)[3]
                for compo in m.compositions
                )
    m.constr_number_per_type_OH = pyo.Constraint(
        rule=rule_number_per_type_OH
    )

    def rule_one_composition_per_trip(m, trip):
        return sum(
            m.allocation_compositions[(trip, composition)]
            for composition in m.compositions
        ) == 1
    m.constr_one_composition_per_trip = pyo.Constraint(
        m.trips,
        rule=rule_one_composition_per_trip
    )
```

```
    compos = get_compositions()
    def rule_length(m, trip, composition):
        OH_length = 100
        OC_length = 70

        length = compos.get(composition)[3] * OH_length  + compos.get(com
        return length * m.allocation_compositions[(trip, composition)] <=
    m.constr_length = pyo.Constraint(
        m.trips,
        m.compositions,
        rule=rule_length
    )
    return m

#
# Objective function for minimizing total costs.
#
def specific_objective_Model5(m):
    days = range(1, 6)

    def obj_maximize_prob_satisfy_capacity(m):
        return 1/(200*5) * sum(
            ((day % 5 == 2 or day % 5 == 4) * increased_probs.loc[trip, c
             + (day % 5 != 2 and day % 5 != 4) * decreased_probs.loc[trip
            ) * m.allocation_compositions[trip, composition]
            for trip in m.trips
            for composition in m.compositions
            for day in days
        )
    m.objective = pyo.Objective(rule=obj_maximize_prob_satisfy_capacity,
    m.objective.pprint()

    return m


####
#### Adding days as dimension
####

##
## Variable specific for Model3
##
def specific_variables_Model5_2(m):
    #
    # Index set for allocation variable model1
    #
    m.days = pyo.Set(initialize = range(1, 6))
    m.index_set_allocation = pyo.Set(initialize = m.trips * m.composition
    #m.index_set_satisfaction = pyo.Set(initialize = m.trips * m.composit

    #
    # Variable for allocation in Model3
    #
    # the binary variable indicates the composition used on a trip.
    #
    m.allocation_compositions = pyo.Var(
        m.index_set_allocation,
        domain = pyo.Binary,
        name = "Specific composition on trip on a day",
```

```
        doc = "Indicates which composition is used on a trip on a specifi
    )

    return m

def specific_parameter_Model5_2(m):
    #
    # Set containing all compositions
    #
    compositions = get_compositions()
    #ic(compositions)

    m.compositions = pyo.Set(initialize = compositions.keys())

    return m


#
# Constraint
#
def specific_constraints_Model5_2(m, sol: pd.DataFrame):
    sol = sol.sum(axis=0).transpose()
    #ic(sol)
    compos = get_compositions()

    #
    # Constraints restricting the number of traintypes.
    #
    def rule_number_per_type_OC(m, day):
        return sum(
            compos.get(compo)[2] * m.allocation_compositions[trip, compo,
                for trip in m.trips
                for compo in m.compositions
            ) <= sum(
                sol.loc[compo] * compos.get(compo)[2]
                for compo in m.compositions
                )
    m.constr_number_per_type_OC = pyo.Constraint(
        m.days,
        rule=rule_number_per_type_OC
    )
    def rule_number_per_type_OH(m, day):
        return sum(
            compos.get(compo)[3] * m.allocation_compositions[trip, compo,
                for trip in m.trips
                for compo in m.compositions
            ) <= sum(
                sol.loc[compo] * compos.get(compo)[3]
                for compo in m.compositions
                )
    m.constr_number_per_type_OH = pyo.Constraint(
        m.days,
        rule=rule_number_per_type_OH
    )

    def rule_one_composition_per_trip(m, trip, day):
        return sum(
            m.allocation_compositions[(trip, composition, day)]
            for composition in m.compositions
        ) == 1
```

```
    m.constr_one_composition_per_trip = pyo.Constraint(
        m.trips,
        m.days,
        rule=rule_one_composition_per_trip
    )

    compos = get_compositions()
    def rule_length(m, trip, composition, day):
        OH_length = 100
        OC_length = 70

        length = compos.get(composition)[3] * OH_length  + compos.get(com
        return length * m.allocation_compositions[(trip, composition, day
    m.constr_length = pyo.Constraint(
        m.trips,
        m.compositions,
        m.days,
        rule=rule_length
    )
    return m

#
# Objective function for minimizing total costs.
#
def specific_objective_Model5_2(m):

    def obj_maximize_prob_satisfy_capacity(m):
        return 1/(200*5) * sum(
            ((day % 5 == 2 or day % 5 == 4) * increased_probs.loc[trip, c
             + (day % 5 != 2 and day % 5 != 4) * decreased_probs.loc[trip
            ) * m.allocation_compositions[trip, composition, day]
            for trip in m.trips
            for composition in m.compositions
            for day in m.days
        )
    m.objective = pyo.Objective(rule=obj_maximize_prob_satisfy_capacity,
    m.objective.pprint()

    return m

# initialized later because it needs the solution of 4
```

In [ ]:

```
#
# Model extension
#

#
# Obtain all compositions
#
def get_compositions(bool_all_combinations:bool = True):
    # Define the lengths of each train type
    OH_length = 100
    OC_length = 70

    OH_cap = 420
    OC_cap = 620

    # Define the maximum length of a train
    max_train_length = 300
```

```
    trips = df['Trip']

    # Generate all possible combinations of OH and OC trains
    all_combinations = {}
    allowed_combinations_per_trip = {}


    for trip in trips.to_list():
        combination_trip = []
        for num_OH in range(max_train_length // OH_length + 1):
            for num_OC in range(max_train_length // OC_length + 1):

                length = num_OH * OH_length + num_OC * OC_length
                if length <= max_train_length - 100 * df['line_400'].loc[
                    combination = []
                    if num_OH > 0:  # Append 'OH' only if num_OH is great
                        combination.extend(['OH'] * num_OH)
                    if num_OC > 0:  # Append 'OC' only if num_OC is great
                        combination.extend(['OC'] * num_OC)

                    cap = num_OH * OH_cap + num_OC * OC_cap

                    if length != 0 and length <= max_train_length - 100 *
                        if OC_cap * num_OC + OH_cap * num_OH >= df['Deman
                            all_combinations[str(combination)] = [length,
                            combination_trip.append(str(combination))

        allowed_combinations_per_trip[trip] = combination_trip


    if bool_all_combinations:
        return all_combinations
    else:
        return allowed_combinations_per_trip


def get_allowed_comp_per_trip_df():

    allowed_comp = pd.DataFrame(index=df['Trip'].tolist(), columns=get_co

    compositions_per_trip = get_compositions(False)
    for trip in get_compositions(False).keys():

        compos_current_trip = compositions_per_trip.get(trip)

        for comp in compos_current_trip:
            allowed_comp.loc[trip, comp] = 1

    return allowed_comp

##
## Parameter specific for ModelEX
##
def specific_parameter_Model_ex(m):
    #
    # Set containing all compositions
    #
    compositions = get_compositions()
    #ic(compositions)
```

```
    m.compositions = pyo.Set(initialize = compositions.keys())

    #
    # Set containing the index for the quantification of the number of tr
    #
    m.index_quantify_compositions = pyo.Set(
        initialize= m.compositions * m.train_type
    )

    #
    # Parameter indicating number of 'OC' and 'OH' within a composition
    #
    m.quantify_compositions = pyo.Param(
        m.index_quantify_compositions,
        initialize = {
            (composition, train_type): compositions[composition][2] if tr
            for composition in m.compositions for train_type in m.train_t
        }
    )

    return m


##
## Variable specific for ModelEX
##
def specific_variables_Model_ex(m):
    #
    # Index set for allocation variable model1
    #
    m.index_set_allocation = pyo.Set(initialize = m.trips * m.composition


    #
    # Variable for allocation in Model1
    #
    # the binary variable indicates the composition used on a trip.
    #
    m.allocation_compositions = pyo.Var(
        m.index_set_allocation,
        domain = pyo.Binary,
        name = "Specific composition on trip",
        doc = "Indicates which composition is used on a trip"
    )

    m.allocation_driver = pyo.Var(
        m.trips,
        domain=pyo.NonNegativeIntegers,
        name = "Specific number of drivers on a trip",
        doc = 'Indicates how many drives are used in a trip'
    )

    return m


##
## Constraints specific for ModelEX
##
def specific_constraints_Model_ex(m):
    # NOTE: CORRECT
```

```
    # Rule that ensures a trip can only have a single composition
    #
    def rule_one_composition_per_trip(m, trip):
        return sum(
            m.allocation_compositions[(trip, composition)]
            for composition in m.compositions
        ) == 1
    m.constr_one_composition_per_trip = pyo.Constraint(
        m.trips,
        rule=rule_one_composition_per_trip
    )

    #
    # Rules for limiting the 'favorate' train type to be max 0.25% higher
    #
    def rule_difference_between_number_of_train_types1(m):
        return 1 * sum(
            m.quantify_compositions[(composition, "OH")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        ) <= 1.25 * sum(
            m.quantify_compositions[(composition, "OC")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        )
    m.constr_difference_between_number_of_train_types1 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types1
    )
    def rule_difference_between_number_of_train_types2(m):
        return 1 * sum(
            m.quantify_compositions[(composition, "OC")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        ) <= 1.25 * sum(
            m.quantify_compositions[(composition, "OH")] * m.allocation_c
            for composition in m.compositions
            for trip in m.trips
        )
    m.constr_difference_between_number_of_train_types2 = pyo.Constraint(
        rule=rule_difference_between_number_of_train_types2
    )

    #
    # Rule for allowed compositions
    #
    allowed_df = get_allowed_comp_per_trip_df()
    def rule_allowed_compos(m, trip, composition):
        return m.allocation_compositions[(trip, composition)] <= allowed_
    m.constr_allowed_compos = pyo.Constraint(
        m.trips,
        m.compositions,
        rule=rule_allowed_compos
    )

    #
    # Rule for total available number of drivers
    #
    def rule_limit_on_available_drivers(m):
        return sum(
            m.allocation_driver[trip]
```

```
            for trip in m.trips
            ) <= 285
    #m.constr_limit_on_available_drivers = pyo.Constraint(
    #    rule=rule_limit_on_available_drivers
    #)

    def rule_number_of_drivers_per_train(m, trip):
        return 3 * sum(
            m.quantify_compositions[(compo, "OH")] * m.allocation_composi
            for compo in m.compositions
            ) + 4 * sum(
            m.quantify_compositions[(compo, "OC")] * m.allocation_composi
            for compo in m.compositions
            ) <= 7 * m.allocation_driver[trip]
    m.constr_number_of_drivers_per_train = pyo.Constraint(
        m.trips,
        rule=rule_number_of_drivers_per_train
    )

    return m


def specific_objective_Model_ex(m):
    m.cost_train_type.pprint()
    def obj_minimize_total_cost(m):
        return m.cost_train_type["OH"] * sum(
                m.quantify_compositions[(composition, "OH")] * m.allocati
                for composition in m.compositions
                for trip in m.trips
            ) + (m.cost_train_type["OC"]) * sum(
                m.quantify_compositions[(composition, "OC")] * m.allocati
                for composition in m.compositions
                for trip in m.trips
            ) + 36 * sum(m.allocation_driver[trip] for trip in m.trips)
    m.objective = pyo.Objective(rule=obj_minimize_total_cost, sense=pyo.m
    m.objective.pprint()
    return m

new_model = create_general_model()
model_ex = specific_parameter_Model_ex(m=new_model)
model_ex = specific_variables_Model_ex(m=model_ex)
model_ex = specific_constraints_Model_ex(m=model_ex)
model_ex = specific_objective_Model_ex(m=model_ex)
```

In [ ]:

```
#
# Additional for sensitivity analysis Model3
#
def simulation_sensitivity_model4():
    new_model = create_general_model()
    model_sim = specific_parameter_Model4(m=new_model)
    model_sim = specific_variables_Model4(m=model_sim)
    model_sim = specific_constraints_Model4(m=model_sim, random_demand=Tr
    model_sim = specific_objective_Model4(m=model_sim)

    return model_sim

def simulation_sensitivity_model1():
    new_model = create_general_model()
    model_sim = specific_variables_Model1(m=new_model)
    model_sim = specific_constraints_Model1(m=model_sim, random_demand=Tr
```

```
    model_sim = specific_objective_Model1(m=model_sim)

    return model_sim
```

In [ ]:
```
###
### CREATING THE SOLVING STRUCTURE
###

##
## Solve a specific model
##
def solve_model(model_to_solve, time_limit: int = 90, print_info: bool =
    ##
    ## Selecting the solver
    ##
    solver = pyo.SolverFactory('cbc') #FIXME: Either 'cbc' or 'glpk': glp

    #
    # Solve the model
    #
    results = solver.solve(
        model_to_solve,
        options={'seconds': time_limit})#,
        #tee=True)

    #
    # Check the solver status
    #
    if (results.solver.status == pyo.SolverStatus.ok) and (results.solver
        print("Solver terminated successfully. Model is feasible.")
    elif results.solver.termination_condition == pyo.TerminationCondition
        print("Solver terminated: Model is infeasible.")
    else:
        print("Solver terminated with non-optimal solution.")

    #
    # Output the information from the solver if necessary.
    #
    if print_info:
        ic(results)
        ic(results.solver.time)

    return results

##
## Obtain solution from Model1
##
def get_solution_df_Model1(model, display_solution_df: bool = False):
    #
    # Use a dictionary to store all information from the decision variabl
    #
    solution_dict = {}

    for trip in model.trips:
        for train_type in model.train_type:
            solution_dict[(trip, train_type)] = model.allocation_train_nu

    #
    # Create and fill a pandas dataframe to store the data from the dicti
    #
```

```
    solution_DF = pd.DataFrame(index=model.trips, columns=model.train_typ

    for trip in model.trips:
        for train_type in model.train_type:
            solution_DF.loc[trip, train_type] = solution_dict.get((trip,

    #
    # Display the solution dataframe
    #
    if display_solution_df:
        ic(solution_DF)


    return solution_DF


##
## Obtain solution from Model2
##
def get_solution_df_Model2(model, display_solution_df: bool = False):
    #
    # Use a dictionary to store all information from the decision variabl
    #
    solution_dict = {}

    for trip in model.trips:
        for composition in model.compositions:
            solution_dict[(trip, composition)] = model.allocation_composi

    #
    # Create and fill a pandas dataframe to store the data from the dicti
    #
    solution_DF = pd.DataFrame(index=model.trips, columns=model.compositi

    for trip in model.trips:
        for composition in model.compositions:
            solution_DF.loc[trip, composition] = solution_dict.get((trip,

    #
    # Display the solution dataframe
    #
    if display_solution_df:
        ic(solution_DF)

    return solution_DF


##
## Obtain solution from Model4
##
def get_solution_df_Model4(model, display_solution_df: bool = False):
    #
    # Use a dictionary to store all information from the decision variabl
    #
    solution_dict = {}

    for trip in model.trips:
        for composition in model.compositions:
            solution_dict[(trip, composition)] = model.allocation_composi
```

```
    #
    # Create and fill a pandas dataframe to store the data from the dicti
    #
    solution_DF = pd.DataFrame(index=model.trips, columns= model.composit

    for trip in model.trips:
        for composition in model.compositions:
            solution_DF.loc[trip, composition] = solution_dict.get((trip,

    #
    # Display the solution dataframe
    #
    if display_solution_df:
        ic(solution_DF)

    return solution_DF

##
## Obtain solution from Model4
##
def get_solution_df_Model5_2(model, display_solution_df: bool = False):
    #
    # Use a dictionary to store all information from the decision variabl
    #
    solution_dict = {}

    for trip in model.trips:
        for composition in model.compositions:
            for day in model.days:
                solution_dict[(trip, composition, day)] = model.allocatio

    #
    # Create and fill a pandas dataframe to store the data from the dicti
    #
    index = pd.MultiIndex.from_product([model.trips, model.days], names=[
    solution_DF = pd.DataFrame(index=index, columns= model.compositions)

    for trip in model.trips:
        for composition in model.compositions:
            for day in model.days:
                solution_DF.loc[(trip, day), composition] = solution_dict

    #
    # Display the solution dataframe
    #
    if display_solution_df:
        ic(solution_DF)

    return solution_DF

##
## Obtain solution from ModelEX
##
def get_solution_df_Model_ex(model, display_solution_df: bool = False):
    #
    # Use a dictionary to store all information from the decision variabl
    #
    solution_dict = {}
    solution_driver_dict = {}
```

```
    for trip in model.trips:
        for composition in model.compositions:
            solution_dict[(trip, composition)] = model.allocation_composi

        solution_driver_dict[(trip)] = model.allocation_driver[trip].valu

    #
    # Create and fill a pandas dataframe to store the data from the dicti
    #
    solution_DF = pd.DataFrame(index=model.trips, columns= model.composit
    solution_driver_DF = pd.Series(index=model.trips)

    for trip in model.trips:
        for composition in model.compositions:
            solution_DF.loc[trip, composition] = solution_dict.get((trip,

        solution_driver_DF.loc[trip] = solution_driver_dict.get(trip, np.

    #
    # Display the solution dataframe
    #
    if display_solution_df:
        ic(solution_DF)
        ic(solution_driver_DF)

    return solution_DF, solution_driver_DF

def get_solution_df_Model_simulation():
    solutions_time = {}
    solutions_objective = {}
    solutions_dfs = {}

    for i in range(0, 100):
        model_sim = simulation_sensitivity_model4()
        results = solve_model(model_to_solve=model_sim, time_limit=3)
        solution4 = get_solution_df_Model4(model=model_sim, display_solut

        if (results.solver.status == pyo.SolverStatus.ok) and (results.so

            solutions_time[i] = results.solver.time
            solutions_objective[i] = pyo.value(model_sim.objective)
            solutions_dfs[i] = solution4
            ic(pyo.value(model_sim.objective))

    ic(sum(solutions_time.values())/len(solutions_time.values()))
    ic(sum(solutions_objective.values())/len(solutions_objective.values()
```

In [ ]:

```
###
### Solving the actual models
###


#
# Solving Model1
#
solve_model(model_to_solve=model1, time_limit=1)
solution1 = get_solution_df_Model1(model=model1, display_solution_df=True



#
# Solving Model2
```

```
#
solve_model(model_to_solve=model2, time_limit=1)
solution2 = get_solution_df_Model2(model=model2, display_solution_df=True

#
# Solving Model3
#
solve_model(model_to_solve=model4, time_limit=300)
solution4 = get_solution_df_Model4(model=model4, display_solution_df=True

#
# Solving Model5
#
new_model = create_general_model()
model5 = specific_parameter_Model5(m=new_model)
model5 = specific_variables_Model5(m=model5)
model5 = specific_constraints_Model5(m=model5, sol=solution4)
model5 = specific_objective_Model5(m=model5)

solve_model(model_to_solve=model5, time_limit=10)
solution5 = get_solution_df_Model4(model=model5, display_solution_df=True

#
# Solving Model5_2
#
new_model = create_general_model()
model5_2 = specific_parameter_Model5_2(m=new_model)
model5_2 = specific_variables_Model5_2(m=model5_2)
model5_2 = specific_constraints_Model5_2(m=model5_2, sol=solution4)
model5_2 = specific_objective_Model5_2(m=model5_2)

solve_model(model_to_solve=model5_2, time_limit=10)
solution5_2 = get_solution_df_Model5_2(model=model5_2, display_solution_d



#
# Solve model Extension
#
solve_model(model_to_solve=model_ex, time_limit=10)
solution_ex, solution_ex_driver = get_solution_df_Model_ex(model=model_ex


#
# Solving Model Sim // Sensitivity Analysis
#
#get_solution_df_Model_simulation()
```

In [ ]:

```
#
# CHECK FEASABILITY MODELS
#

# Get number of trains per thing
compos = get_compositions()
number_of_trains = {}
model_sols = [solution1, solution2, solution4, solution5, solution_ex]
for i in range(0, len(model_sols)):
    sol = model_sols[i]
    sol = sol.sum(axis=0).transpose()
    #ic(sol)
```

```
    num_OC = 0
    num_OH = 0
    for compo in compos:
        try:
            if i == 0:
                compo_stripped = compo.strip("[]'")
                num_OC = num_OC + sol.loc[compo_stripped] * compos.get(co
                num_OH = num_OH + sol.loc[compo_stripped] * compos.get(co
            else:
                num_OC = num_OC + sol.loc[compo] * compos.get(compo)[2]
                num_OH = num_OH + sol.loc[compo] * compos.get(compo)[3]


            #ic(compo)
            #ic(compos.get(compo)[2])

            #ic(num_OC)
            #ic(num_OH)
        except:
            pass

    number_of_trains[(i + 1, "OC")] = num_OC
    number_of_trains[(i + 1, "OH")] = num_OH

ic(number_of_trains)

ic((solution2 == solution_ex).sum())

ic(solution_ex_driver[solution_ex_driver != 1])
ic(solution_ex_driver.sum())
#
# CHECK FEASABILITY MODEL2
#

"""# Check allocation to specific train types.
total_composition_sol2 = solution2.sum(axis=0)
total_composition_sol2 = total_composition_sol2[total_composition_sol2 !=
ic(total_composition_sol2)

total_alloc_per_trip = solution2.sum(axis=1)
total_alloc_per_trip = total_alloc_per_trip[total_alloc_per_trip != 1.0]
ic(total_alloc_per_trip)

ic((111/89 - 1) <= 0.25)


dataframe = pd.DataFrame(index=model4.trips, columns=model4.days)

for trip in model4.trips:
    for day in model4.days:
        dataframe.loc[trip, day] = model4.passengers_per_trip[(trip, day)


dataframe = dataframe.max(axis=0)

ic(dataframe[(dataframe > 2000)])
ic(df[df['line_400'] == 1])
ic(get_compositions())
"""
```

```
#model3.binairy_capacity_satisfied.pprint()
```