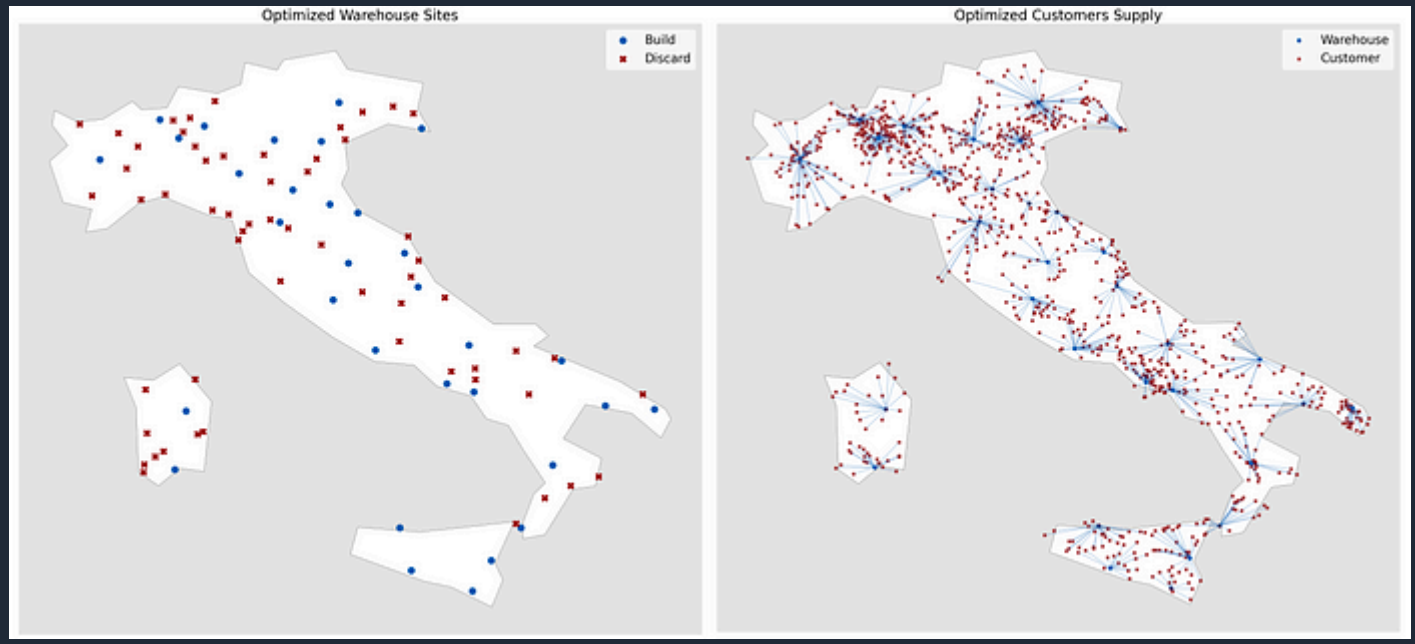

Optimization: Capacitated Facility Location Problem in Python

Find the optimal number and location of warehouses to reduce costs and meet demand

Nicolo Cosimo Albanese

Follow

Table of contents

1. Introduction

Facility Location Problems (FLPs) are classical optimization tasks. They aim at determining the best among potential sites for warehouses or factories.

Warehouses may or may not have a limited capacity. This distinguishes the *capacitated (CFLP)* from the *uncapacitated (UFLP)* variants of the problem.

The business goal to find the set of warehouse locations that minimize the costs. The original problem definition by Balinski (1965) minimizes the sum of two (annual) cost voices:

Transportation costs account for the expenses generated by reaching customers from the warehouse location. The warehouse fixed cost is location-specific. It may include, for instance, voices such as rent, taxes, electricity and maintenance.

Facility location is a well known subject and has a fairly rich literature. As such, many variants of the problem exist, as well as approaches. This post introduces the classical CFLP formulation and shares a practical Python example with PuLP.

2. Problem statement

The goal of CFLP is to determine the number and location of warehouses that will meet the customers demand while reducing fixed and transportation costs. Therefore, we can pose the problem as the minimization of the following objective function:

$$\sum_{j=1}^{M} f_j \cdot y_j + \sum_{i=1}^{N} \sum_{j=1}^{M} t_{ij} \cdot x_{ij}$$

- M is a set of candidate warehouse locations.
- f_j represent the annual fixed cost for warehouse j.
- t_{ij} represents the cost of transportation from warehouse j to customer i.
- x_{ij} is the number of units delivered from warehouse j to customer i.
- y_j is a binary variable $y_j \in \{0,1\}$, indicating whether the warehouse should be built in the location j ($y_j = 1$) or not ($y_j = 0$).

Let us now consider the addition of **constraints** to the objective function.

Since we are modeling a *capacitated* problem, each facility j can supply an annual maximum capacity C_j. Therefore, the number of units delivered to a customer x_{ij} cannot be greater than this value:

$$\sum_{i=1}^{N} x_{ij} \leq C_j \cdot y_j$$

The yearly units delivered from warehouse j to customer i must range between zero and d_i, the annual demand from customer i:

And last but not least, we must meet customers' demand. In this example, we impose that each warehouse serving a customer location must fully meet its demand:

$$\sum_{j=1}^{M} x_{ij} = d_i$$

In conclusion, we can define the problem as follows:

$$\sum_{j=1} \qquad \sum_{i=1}\sum_{j=1}$$

subject to:

$$\sum_{j=1}^{M} x_{ij} = d_i$$

$$\sum_{i=1}^{N} x_{ij} \leq C_j \cdot y_j$$

$$0 \leq x_{ij} \leq d_i \cdot y_j$$

$$y_j \in \{0,1\}$$

Image by author.

3. Implementation

Let us import the needed libraries:

- `NumPy`, `Pandas` for data manipulation.
- `math` for specific mathematical functions.
- `GeoPandas` for geospatial representations.

Copy

```
import numpy as np
import pandas as pd
import geopandas
from math import sin, cos, asin, acos, radians

from pulp import LpProblem, LpMinimize, LpVariable, LpBinary, lpSum, LpStatus, val

import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

3.1. The dataset

We settle our optimization problem in Italy.

The starting dataset is available on simplemaps.com. We may download the input csv file here and use it freely for personal and commercial use under the MIT license.

Copy

```
        './it.csv',
        usecols = ['city', 'lat', 'lng', 'population', 'capital', 'admin_name'])
```

We are interested in the following columns:

- `city`: name of the town;
- `lat`: latitude;
- `lng`: longitude;
- `population`: number of inhabitants;
- `capital`: indicates whether the city is a chief town or administrative center;
- `admin_name`: name of the highest level administration region.

3.2. Customers, warehouses and demand

When creating customers, facility and demand, we assume that:

- Customers are a fraction (30%) of the input cities.
- Facilities can be established only in administrative centers. As a starting condition, we assume we could build warehouses in 80% of the Italian chief towns.

Copy

```
RANDOM_STATE = 2          # For reproducibility
FRACTION_CUSTOMERS = 0.3  # Fraction of cities we want to keep as customers
FRACTION_WAREHOUSES = 0.8 # Fraction of cities we want to keep as warehouse locati
FRACTION_DEMAND = 0.02    # Fraction of citizens of a city that may order a produc

# List of the 20 regions of Italy
REGION_LIST = [
    'Lombardy', 'Veneto', 'Emilia-Romagna', 'Sicilia', 'Campania', 'Piedmont', 'Pu
    'Lazio', 'Calabria', 'Tuscany', 'Sardegna', 'Marche', 'Friuli-Venezia Giulia',
    'Umbria', 'Trentino-Alto Adige', 'Liguria', 'Basilicata', 'Molise', 'Valle d'A

# Demand is composed of:
#   1. A fraction of the population
#   2. An error term of uniform distribution
# Note: demand is approximated to the closest int
# as its physical meaning denies decimals
df['demand'] = np.floor(
    FRACTION_DEMAND * df.population + np.random.uniform(-10, 10, size=(df.shape[0]

# Create the warehouses dataframe:
#   1. Filter the 20 regions of Italy
#   2. Filter capitals as candidate warehouse locations
#   3. Sample a fraction of the original cities
facility_df = df.\
                loc[df.admin_name.isin(REGION_LIST)].\
```

```
# Create the customers dataframe:
#   1. Filter the 20 regions of Italy
#   2. Sample a fraction of the original cities
customer_df = df.\
                loc[df.admin_name.isin(REGION_LIST)].\
                sample(frac=FRACTION_CUSTOMERS, random_state=RANDOM_STATE, ignore_

# Customers IDs list
customer_df['customer_id'] = range(1, 1 + customer_df.shape[0])
```

Note: in the online dataset, the region name `Valle d'Aosta` contains a typographic (curved) apostrophe (U+2019) instead of the typewriter (straight) apostrophe (U+0027). Please consider it if reproducing this code.

Although it is not necessary to the optimization task, we may want to observe our locations on a map. `geopandas` simplifies this task. One may easily create a `GeoDataFrame` enriched with geospatial information using the `points_from_xy` method:

Copy

```
    Add column "geometry" with <shapely.geometry.point.Point> objects
        built from latitude and longitude values in the input dataframe

    Args:
        - df: input dataframe
        - lat: name of the column containing the latitude (default: lat)
        - lng: name of the column containing the longitude (default: lng)
    Out:
        - df: same dataframe enriched with a geo-coordinate column
    '''
    assert pd.Series([lat, lng]).isin(df.columns).all(),\
        f'Cannot find columns "{lat}" and/or "{lng}" in the input dataframe.'
    return geopandas.GeoDataFrame(
        df, geometry=geopandas.points_from_xy(df.lng, df.lat))


customer_df = add_geocoordinates(customer_df)
facility_df = add_geocoordinates(facility_df)
```

We can access a map of Italy through `geopandas` and plot customers and potential warehouse locations:

```
Copy
```

```
# Extract and plot the shape of Italy
italy = world[world.name == 'Italy']
ax = italy.plot(color='white', edgecolor='black', figsize=(10, 10))

# Plot customers as points
customer_df.\
    plot(ax=ax, marker='X', color='red', markersize=30, alpha=0.5, label='Customer

# Plot potential facility locations as points
facility_df.\
    plot(ax=ax, marker='D', color='blue', markersize=30, alpha=0.5, label='Potenti

# Add legend
plt.legend(facecolor='white', title='Location')

# Add title
plt.title('Customer and potential warehouses')

# Remove ticks from axis
plt.xticks([])
plt.yticks([])

# Show plot
plt.show()
```

Image by author.

Similarly, we can observe the average demand for each of the 20 Italian regions:

Copy

```
# Prepare region dataframe:
#   1. Filter the 20 regions of Italy
#   2. Group by region
#   3. Calculate:
#      - Mean regional latitude
#      - Mean regional longitude
#      - Sum of regional demand
region_df = df.\
            loc[df.admin_name.isin(REGION_LIST)].\
            groupby(['admin_name']).\
            agg({'lat': 'mean', 'lng': 'mean', 'demand': 'sum'}).\
            reset_index()

# Add geo-coordinates
region_df = add_geocoordinates(region_df)

# Plot the shape of Italy
ax = italy.plot(color='white', edgecolor='black', figsize=(10, 10))

# Plot region area colored based on demand
region_df.\
```

```
region_df.\
    plot(ax=ax, marker='o', c='red', markersize=25, alpha=0.8, label='Customer loc

# Add region name above the center
for i, row in region_df.iterrows():
    plt.annotate(
        row.admin_name, xy=(row.lng, row.lat+0.2), horizontalalignment='center')

# Add color bar with demand scale
plt.colorbar(ax.get_children()[1], ax=ax, label='Annual Demand', fraction=0.04, pa

# Add title
plt.title('Annual demand by region')

# Remove ticks from axis
plt.xticks([])
plt.yticks([])

# Show plot
plt.show()
```

Image by author.

Copy

```
# Dictionary of cutomer id (id) and demand (value)
demand_dict = { customer : customer_df['demand'][i] for i, customer in enumerate(c
```

3.3. Supply and fixed costs

To model supply and fixed costs, we assume that:

- Each warehouse can meet a maximum yearly supply equal to 3 times the average regional demand.
- Each warehouse has a constant annual fixed cost of 100.000,00 €, independently from its location.

As we did for the demand, we store supply and fixes costs in dictionaries:

Copy

```
# Assumptions:
#    1. Each warehouse has an annual cost of 100.000,00 euros: rent, electricity,
```

```
SUPPLY_PER_WAREHOUSE = region_df.demand.mean() * SUPPLY_FACTOR_PER_WAREHOUSE

# Warehouses list
facility_df['warehouse_id'] = ['Warehouse ' + str(i) for i in range(1, 1 + facilit

# Dictionary of warehouse id (id) and max supply (value)
annual_supply_dict = { warehouse : SUPPLY_PER_WAREHOUSE for warehouse in facility_

# Dictionary of warehouse id (id) and fixed costs (value)
annual_cost_dict = { warehouse : COST_PER_WAREHOUSE for warehouse in facility_df['
```

3.4. Transportation costs

The estimate of transportation costs requires:

- the distance between the different locations, and
- a cost function per unit of distance.

We can approximate the distance between two locations on a spherical surface using the Haversine formula:

Copy

```
    Calculate distance between two locations given latitude and longitude.

    Args:
       - lat1: latitude of the first location
       - lon1: longitude of the first location
       - lat2: latitude of the second location
       - lon2: longitude of the second location
    Out:
       - Distance in Km

    Ref:
       - https://en.wikipedia.org/wiki/Haversine_formula
    '''
    return 6371.01 *\
            acos(sin(radians(lat1))*sin(radians(lat2)) +\
            cos(radians(lat1))*cos(radians(lat2))*cos(radians(lon1)-radians(lon2))
```

Let us try it on two cities:

- Milano (latitude: 45.4654219, longitude: 9.18854)
- Bergamo (latitude: 45.695000, longitude: 9.670000)

Copy

We obtain a distance of 45.5 Km. Unfortunately, this measure does not correspond to the one we would see, for instance, on a car navigation system, as we do not take routes into account:

Haversine and route distance between two cities, Milano and Bergamo. Image by author.

Nevertheless, we can use our estimate as a reasonable approximation for our task.

Finally, we need to convert distances in a measure of cost. At the moment of this writing, the average price of gasoline in Italy is 1.87 €/L (source). The average consumption of an EURO VI truck is around 0.38 L/Km (source). With a simple, yet reasonable, approximation, we can estimate an average cost of 0.71 € per Km traveled on the Italian soil:

Copy

```
    Return traveling cost in euros given a distance in Km.

    Args:
      - distance_in_km: travel distance in Km
    Out:
      - cost of the trip in euros
    '''
    return 0.71 * distance_in_km
```

We can now calculate the traveling costs for each warehouse-customer pair and store them in a dictionary:

Copy

```
# Dict to store the distances between all warehouses and customers
transport_costs_dict = {}

# For each warehouse location
for i in range(0, facility_df.shape[0]):

    # Dict to store the distances between the i-th warehouse and all customers
    warehouse_transport_costs_dict = {}

    # For each customer location
    for j in range(0, customer_df.shape[0]):
```

```
            facility_df.lat[i], facility_df.lng[i], customer_df.lat[j], customer_d

        # Update costs for warehouse i
        warehouse_transport_costs_dict.update({customer_df.customer_id[j]: traveli

    # Final dictionary with all costs for all warehouses
    transport_costs_dict.update({facility_df.warehouse_id[i]: warehouse_transport_
```

3.5. Optimization

Let us remind the optimization problem:

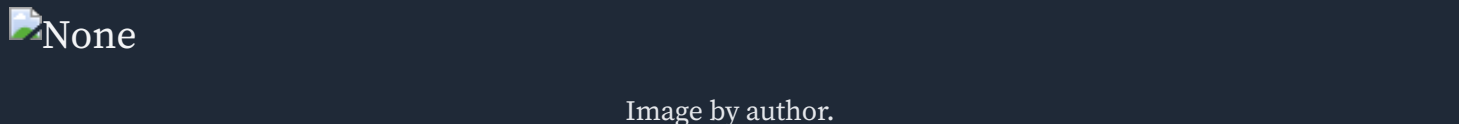

Image by author.

We can define the two decision variables x_{ij} and y_j, the objective function and constraints as follows:

Copy

```
# Define linear problem
lp_problem = LpProblem('CFLP', LpMinimize)
```

```
created_facility = LpVariable.dicts(
    'Create_facility', facility_df['warehouse_id'], 0, 1, LpBinary)

# Variable: x_ij
served_customer = LpVariable.dicts(
    'Link', [(i,j) for i in customer_df['customer_id'] for j in facility_df['wareh


# Objective function
objective = lpSum(annual_cost_dict[j]*created_facility[j] for j in facility_df['wa
            lpSum(transport_costs_dict[j][i]*served_customer[(i,j)] \
                  for j in facility_df['warehouse_id'] for i in customer_df['custo

lp_problem += objective

# Costraint: the demand must be met
for i in customer_df['customer_id']:
    lp_problem += lpSum(served_customer[(i,j)] for j in facility_df['warehouse_id'

# Constraint: a warehouse cannot deliver more than its capacity limit
for j in facility_df['warehouse_id']:
    lp_problem += lpSum(served_customer[(i,j)] for i in customer_df['customer_id']

# Constraint: a warehouse cannot give a customer more than its demand
for i in customer_df['customer_id']:
    for j in facility_df['warehouse_id']:
        lp_problem += served_customer[(i,j)] <= demand_dict[i] * created_facility[
```

```
Copy

lp_problem.solve()
```

We can check the outcome as follows:

```
Copy

print('Solution: ', LpStatus[lp_problem.status])
Solution: Optimal
```

We are now interested in exploring the decision variables: how many warehouses do we need? In what locations?

4. Explore the results

At first, let us consider the business goal: minimize costs. We can check the value assumed by the objective function:

```
value(lp_problem.objective)
8964323.323646087
```

This is the minimum possible cost we can achieve under the given constraints. Any other choice in the number or location of the warehouses would lead to a higher value of the objective function.

We can access the decision variables through the `varValue` property. For example, we can see the value assumed by y_j for `j = Warehouse 1`:

Copy

```
created_facility['Warehouse 1'].varValue
1.0
```

As $y_j = 1$, we should establish a warehouse in that location. We can easily manipulate the variable and count the number of needed facilities:

Copy

```
# Count of each distinct value of the list
[[i, facility_values.count(i)] for i in set(facility_values)]
[[0.0, 59], [1.0, 32]]
```

It is sufficient to **build just 32 of the initially budgeted 91 sites.** The 35.1% (32 / 91) of all potential warehouses is enough to meet the demand under the given constraints.

We can save the decision variable in the initial data frame and observe the chosen locations:

Copy

```
# Create dataframe column to store whether to build the warehouse or not
facility_df['build_warehouse'] = ''

# Assign Yes/No to the dataframe column based on the optimization binary variable
for i in facility_df['warehouse_id']:
    if created_facility[i].varValue == 1:
        print('Build site at: ', i)
        facility_df.loc[facility_df['warehouse_id'] == i, 'build_warehouse'] = 'Ye
    else:
        facility_df.loc[facility_df['warehouse_id'] == i, 'build_warehouse'] = 'No
```

```
Build site at:  Warehouse 4
Build site at:  Warehouse 7
Build site at:  Warehouse 8
Build site at:  Warehouse 16
Build site at:  Warehouse 18
Build site at:  Warehouse 20
Build site at:  Warehouse 21
Build site at:  Warehouse 22
Build site at:  Warehouse 23
Build site at:  Warehouse 25
Build site at:  Warehouse 26
Build site at:  Warehouse 27
Build site at:  Warehouse 29
Build site at:  Warehouse 33
Build site at:  Warehouse 35
Build site at:  Warehouse 38
Build site at:  Warehouse 48
Build site at:  Warehouse 49
Build site at:  Warehouse 55
Build site at:  Warehouse 56
Build site at:  Warehouse 57
Build site at:  Warehouse 58
Build site at:  Warehouse 63
Build site at:  Warehouse 66
Build site at:  Warehouse 70
Build site at:  Warehouse 74
Build site at:  Warehouse 82
Build site at:  Warehouse 83
```

```
facility_df.build_warehouse.value_counts().plot.barh(
  title='Warehouse sites to be established', xlabel='Number of sites', color=color

for i, v in enumerate(facility_df.build_warehouse.value_counts()):
    plt.text(v, i, ' '+str(round(v,3)), color=colors[i], va='center', fontweight='
```

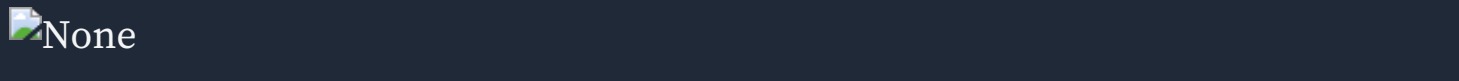

Image by author.

Copy

```
# Plot the shape of Italy
ax = italy.plot(color='white', edgecolor='black', figsize=(10, 10))

# Plot sites to establish
facility_df.\
    loc[facility_df.build_warehouse =='Yes'].\
    plot(ax=ax, marker='o', c='#0059b3', markersize=50, label='Build')

# Plot sites to discard
facility_df.\
    loc[facility_df.build_warehouse =='No'].\
    plot(ax=ax, marker='X', c='#990000', markersize=40, label='Discard')
```

```
# Add legend
plt.legend(title='Warehouse Site', facecolor='white')

# Remove ticks from axis
plt.xticks([])
plt.yticks([])

# Show plot
plt.show()
```

Image by author.

Similarly, we can iterate over the decision variable x_{ij} and find the customers served by each warehouse in the optimized solution:

Copy

```
def get_linked_customers(input_warehouse):
    '''
    Find customer ids that are served by the input warehouse.
```

```
        - List of customers ids connected to the warehouse
    '''
    # Initialize empty list
    linked_customers = []

    # Iterate through the xij decision variable
    for (k, v) in served_customer.items():

            # Filter the input warehouse and positive variable values
            if k[1]==input_warehouse and v.varValue>0:

                # Customer is served by the input warehouse
                linked_customers.append(k[0])

    return linked_customers


# Warehouses to establish
establish = facility_df.loc[facility_df.build_warehouse =='Yes']

# Plot the shape of Italy
ax = italy.plot(color='white', edgecolor='black', figsize=(30, 30))

# Plot sites to establish
establish.\
    plot(ax=ax, marker='o', c='#0059b3', markersize=100, label='Warehouse')

# Plot customers
```

```
# For each warehouse to build
for w in establish.warehouse_id:

    # Extract list of customers served by the warehouse
    linked_customers = get_linked_customers(w)

    # For each served customer
    for c in linked_customers:

        # Plot connection between warehouse and the served customer
        ax.plot(
         [establish.loc[establish.warehouse_id==w].lng, customer_df.loc[customer_d
         [establish.loc[establish.warehouse_id==w].lat, customer_df.loc[customer_d
         linewidth=0.8, linestyle='--', color='#0059b3')

# Add title
plt.title('Optimized Customers Supply', fontsize = 35)

# Add legend
plt.legend(facecolor='white', fontsize=30)

# Remove ticks from axis
plt.xticks([])
plt.yticks([])

# Show plot
plt.show()
```

Image by author.

5. Conclusions

In this post, we introduced a classical optimization challenge: the Capacitated Facility Location Problem (CFLP). We described its derivation and shared a practical Python example. In particular, since we started with a raw dataset of geographical locations, we covered all the necessary passages and assumptions needed to frame and solve the problem.

#optimization #data-science #machine-learning #artificial-intelligence