Analysis and Ranking of Milan's Neighborhoods Operational Guide

Edoardo Olivieri, Federica Romano, Francesca Verna

January 28, 2025

Version: 1.1
Last updated: 27-01-2024
Operational guide regarding the report:
Olivieri, E., Romano, F., & Verna, F. (2025). *Analysis and Ranking of Milan's Neighborhoods.*

Contents

1 Introduction

This operational guide outlines the steps to reproduce the project aimed at analyzing and ranking neighborhoods in Milan. The analysis combines datasets retrieved from the Yelp API and the Dati Comune di Milano portal to rank neighborhoods based on their amenities, such as restaurants, nightlife, universities, and other facilities. The project leverages MongoDB for data storage, Jupyter Notebooks for processing and making interactive visualizations, and PyMongo for querying. The ultimate goal is to provide insights into how neighborhoods are structured and which are most suitable for students, singles/couples and families.

2 Environment Setup

To ensure the reproducibility of this project, it is essential to set up the correct computational environment. The project requires Python and a set of libraries for data handling, geospatial processing, and visualization. Additionally, MongoDB is used for data storage and querying.

2.1 Software and Libraries

To reproduce this project, install the following:

1. **MongoDB**: Used to store data.
2. **Jupyter Notebook**: Necessary to gather and query the data.
3. **Python libraries**: Install the necessary packages via `pip`:

```
pip install pandas geopandas shapely folium requests pymongo
```

   - pandas for data manipulation.
   - geopandas for geospatial data handling.
   - shapely for geometric operations.
   - folium for creating interactive maps.
   - requests for accessing the Yelp API.
   - pymongo for interfacing with MongoDB.

3 Data Acquisition

The project relies on three main data sources: OpenStreetMap (in particular Overpass turbo) for the coordinates of the neighborhoods, the Yelp API for amenities and reviews, and the Comune di Milano datasets for home prices and public facilities. These datasets are then combined and stored in MongoDB.

3.1 Data from OpenStreetMap

To get the data about the coordinates of neighborhoods in Milan, we need two queries from overpass turbo, which results will be downloaded as geojson files. This first one identifies correctly most of the neighborhoods:

```
[out:json][timeout:25];
area["name"="Milano"]["boundary"="administrative"]["admin_level"="8"]
->.searchArea;
way["place"="quarter"](area.searchArea);
out body;
>;
out skel qt;
```

This second query is necessary to also get coordinates for the neighborhood *Forze Armate*, which can't be retrieved with the first one.

```
[out:json][timeout:25];
area["name"="Milano"]["boundary"="administrative"]->.searchArea;
way["place"="quarter"](area.searchArea);
relation["place"="quarter"](area.searchArea);
out body;
>;
out skel qt;
```

Additionally, from overpass turbo we also got a geojson file containing info about supermarkets, using the following query:

```
[out:json];
area["name"="Milano"]["boundary"="administrative"]->.searchArea;
(
  node["shop"="supermarket"](area.searchArea);
  way["shop"="supermarket"](area.searchArea);
  relation["shop"="supermarket"](area.searchArea);
);
out body;
>;
out skel qt;
```

3.2 Data from Yelp API

To use the Yelp API, you must register for access and obtain an API key by following these steps:

1. Create a Yelp Developer Account:

- Log in with your Yelp account or create a new one if you do not already have one.

2. Create a New App:
   - Navigate to the *Manage Apps* section in the developer portal.
   - Click *Create App* and fill in the required fields:
     - App Name: Give your app a name.
     - App Description: Provide a brief description of your app.
   - Once created, you will be provided with an API key.

Once the API key is obtained, using a combination of two custom functions, you can retrieve data about various points of interest:

```
HEADERS = {"Authorization": f"Bearer {API_KEY}"}
businesses_per_request = 50  # Maximum allowed by Yelp per request

def make_request(url, params=None):
    # Makes a request to the Yelp API.
    response = requests.get(url, headers=HEADERS, params=params)
    response.raise_for_status()
    return response

def search_businesses(location, term="restaurant", limit=businesses_per_request,
↪   offset=0):
    # Searches for businesses in a given location with pagination.
    url = "https://api.yelp.com/v3/businesses/search"
    params = {
        "location": location,
        "term": term,
        "limit": limit,
        "offset": offset
    }
    response = make_request(url, params=params)
    return response.json().get("businesses", [])
```

3.3 Data from Comune di Milano

Download datasets for public facilities, such as schools, parks, and transport, and save them in a dedicated folder.

```
geojson_urls = [
    # Insert here the urls for the geojson files
]
```

```
output_directory = "geojson_files"
# Download and save each geojson file
for url in geojson_urls:
    try:
        response = requests.get(url)
        response.raise_for_status()
        file_name = os.path.basename(url)
        output_path = os.path.join(output_directory, file_name)
        with open(output_path, "wb") as file:
            file.write(response.content)
    except requests.exceptions.RequestException as e:
        print(f"Failed to download {url}: {e}")
```

Regarding the dataset containing informations about home prices, it needs to be downloaded as a csv, as it is the only format available.

4 Data Processing

Once the raw data has been gathered from the three primary sources, the next step involves processing and integrating these datasets. The goal of this stage is to clean, standardize, and merge the data into a cohesive structure that allows for storage, analysis and visualization.

4.1 Spatial Join

The spatial join is a crucial step to associate point-based data with specific neighborhoods. This step uses the polygon boundaries of neighborhoods obtained from OpenStreetMap. To perform this join we propose the *geopandas.sjoin* function to match points to their corresponding neighborhoods.

```
joined = sjoin(point_based_data, polygon_boundaries, how="left", predicate="within")
```

4.2 Data Cleaning

For each dataset it is necessary to check for duplicates, potential errors and missing values. Since each dataset is different, there is no universal, one-size-fits-all approach to cleaning the data. For more datailed information about how the data was cleaned consult the original report Analysis and Ranking of Milan's Neighborhoods.

4.3 Home Prices

Regarding the dataset containing informations about home prices, it is only available in a csv format, and therefore needs a different treatment compared to the geojson files. Since

it is not a geojson file, and therefore has no coordinates in it, we decided to join it with a geojson file containing the polygons for these Zones. First the csv file needs to be cleaned, and then can be merged:

```
merged = zones_polygons_file.merge(cleaned_csv_file, on="Zona", how="left")
```

Once the dataset is enritched with the polygons, using the dataset from OpenStreetMap each entry is matched to their neighborhood:

```
joined = gpd.sjoin(polygon_boundaries, merged, how="inner", predicate="intersects")
```

The final step is to group by neighborhood so that one entry per neighborhood is kept. At this stage, some aggregated metrics are saved to serve as general info for each neighborhood:

```
final = joined.groupby("Neighborhood").agg({
    "Compr_min": "min",            # Minimum Compr_min for each Neighborhood
    "Compr_max": "max",            # Maximum Compr_max for each Neighborhood
    "Compr_mean": "mean",          # Average Compr_mean for each Neighborhood
    "geometry": "first"
}).reset_index()
```

5 Data Integration and Storage

This section describes how the neighborhood and point-of-interest (POI) data is integrated into a unified structure and then stored in MongoDB.

5.1 Connecting to MongoDB

We begin by creating a MongoDB client using the appropriate connection string. In our example, we connect to a locally hosted database with a username and password:

```
from pymongo import MongoClient

client = MongoClient("mongodb://admin:DataMan2023!@localhost:27017/")
db = client["my_database"]
collection = db["neighborhoods"]
```

5.2 Reading the Source Files

We have multiple GeoJSON files, one for each category of POI (restaurants, museums, pharmacies, etc.). We use GeoPandas to read these files into GeoDataFrames:

```
import geopandas as gpd

gdf_combined = gpd.read_file("C:/path of the polygons file.geojson")
PolyRestaurants = gpd.read_file("C:/path of the restaurants file.geojson")
PolyMuseums = gpd.read_file("C:/path of the museums file.geojson")
...
# other files
...
```

5.3 Creating the Base Neighborhood Documents

We create a dictionary in Python to hold our neighborhood documents. Each key in the dictionary corresponds to one neighborhood. We then iterate through the rows of gdf_combined, which contains the neighborhood geometries and names, and create a skeleton document for each neighborhood:

```
neighborhood_docs = {}

for idx, row in gdf_combined.iterrows():
    nb_name = row["Neighborhood"]
    geo_json = mapping(row["geometry"])

    neighborhood_docs[nb_name] = {
        "_id": nb_name,
        "neighborhood_name": nb_name,
        "geometry": geo_json,
        "locations": {
            "restaurants": [],
            "museums": [],
            "nightlife": [],
            ... # other POIs here ...
        }
        "home_prices": {
            ... # home prices metrics ...
        }
    }
```

5.4 Appending Points of Interest

We define a function called append_pois_to_neighborhoods that takes a GeoDataFrame, a target POI key, and an optional dictionary of field mappings. This function: Groups rows by the Neighborhood column. Iterates over each group and locates the matching neighborhood in neighborhood_docs. Converts the relevant columns in each row into a dictionary (poi_data) using field_mappings to standardize field names. Appends each POI dictionary to the relevant list inside neighborhood_docs[nb_name]["locations"][poi_key].

```
def append_to_neighborhoods(field, gdf, poi_key, field_mappings=None):
    if field_mappings is None:
        field_mappings = {
            col: col
            for col in gdf.columns
            if col not in ("Neighborhood", "geometry")
        }
    grouped = gdf.groupby("Neighborhood")
    for nb_name, group_df in grouped:
        if nb_name not in neighborhood_docs:
            continue
        for _, row in group_df.iterrows():
            poi_data = {}
            for src_col, dest_col in field_mappings.items():
                if src_col in row:
                    poi_data[dest_col] = row[src_col]
            neighborhood_docs[nb_name][field][poi_key].append(poi_data)
```

We then call the helper function for each GeoDataFrame, specifying the correct poi_key and providing the relevant field_mappings. For instance, for restaurants:

```
append_pois_to_neighborhoods(
    gdf=PolyRestaurants,
    poi_key="restaurants",
    field_mappings={
        "Business Name": "name",
        "Business Address": "address",
        "Categories": "category",
        "Average Star Rating": "avg_star_rating",
        "Review Count": "tot_ratings",
        "Price": "price"
    }
)
```

We repeated this process for museums, nightlife venues, pharmacies, etc., ensuring each POI category is correctly mapped to the corresponding dictionary fields.

5.5 Appending Home Prices

Once the datasets about locations are added, the informations about the home prices are inserted as follows:

```
for idx, row in PolyHomePrices.iterrows():
    nb_name = row["Neighborhood"]
    if nb_name in neighborhood_docs:
        neighborhood_docs[nb_name]["home_prices"]["min_price"] = row["Compr_min"]
        neighborhood_docs[nb_name]["home_prices"]["max_price"] = row["Compr_max"]
        neighborhood_docs[nb_name]["home_prices"]["avg_price"] = row["Compr_mean"]
```

5.6 Storing into MongoDB

MongoDB requires inserts to be in the form of a list (or single documents). We therefore convert the neighborhood_docs dictionary to a list, and used insert_many to add all neighborhood documents at once:

```
documents_to_insert = list(neighborhood_docs.values())
collection.insert_many(documents_to_insert)
```

6 Queries

Once the data has been stored in MongoDB, it is possible to perform queries to retrieve and analyze the information. Below are some examples of basic queries using PyMongo to interact with the database. To see more complex queries, for ranking and scoring the Neighborhood, visit the our in depth code in the MongoDB Integration and Queries.
Before querying the database, ensure that you have established a connection as shown earlier:

```
from pymongo import MongoClient

client = MongoClient("mongodb://admin:DataMan2023!@localhost:27017/")
db = client["my_database"]
collection = db["neighborhoods"]
```

Example

1. Retrieve All Neighborhood Names
This query fetches the names of all neighborhoods in the database:

```
neighborhood_names = collection.find({}, {"_id": 0, "neighborhood_name": 1})
for name in neighborhood_names:
    print(name["neighborhood_name"])
```

Example

2. Find a specific Neighborhood by name
This query retrieves a document for a specific neighborhood, such as "Duomo":

```
neighborhood = collection.find_one({"neighborhood_name": "Duomo"})
print(neighborhood)
```

Example

3. Count the number of POIs in each Neighborhood
This query calculates the total number of restaurants in each neighborhood:

```
cursor = collection.aggregate([
    {"$project": {
        "neighborhood_name": 1,
        "restaurant_count": {"$size": "$locations.restaurants"}
    }}
])
for doc in cursor:
    print(f"{doc['neighborhood_name']}: {doc['restaurant_count']} restaurants")
```

Example

4. Retrieve Neighborhoods with more than a certain number of POIs
This query finds neighborhoods with more than 50 restaurants:

```
cursor = collection.find(
    {"$expr": {"$gte": [{"$size": "$locations.restaurants"}, 50]
        }
    },{"_id": 0, "neighborhood_name": 1}
)
for doc in cursor:
    print(doc["neighborhood_name"])
```

Example

5. Find the Most Expensive Neighborhoods
This query sorts neighborhoods by their average home prices in descending order:

```
cursor = collection.find(
    {"home_prices.avg_price": {"$exists": True}},
    {"_id": 0, "neighborhood_name": 1, "home_prices.avg_price": 1}
).sort("home_prices.avg_price", -1)
for doc in cursor:
    print(f"{doc['neighborhood_name']}: {doc['home_prices']['avg_price']}
    ↪  average price")
```