

Analysis and Ranking of Milan's Neighborhoods Operational Guide

Edoardo Olivieri, Federica Romano, Francesca Verna

January 29, 2025

Version: 1.1

Last updated: 29-01-2024

Operational Guide regarding the Report:

Olivieri, E., Romano, F., & Verna, F. (2025). *Analysis and Ranking of Milan's
Neighborhoods*.

Contents

1	Introduction	2
2	Environment Setup	2
2.1	Software and Libraries	2
3	Data Acquisition	2
3.1	Data from OpenStreetMap	3
3.2	Data from Yelp API	3
3.3	Data from Comune di Milano	6
4	Data Preparation	6
4.1	Spatial Join	7
4.2	Data Cleaning	7
4.3	Home Prices	7
5	Data Storage	8
5.1	Connecting to MongoDB	8
5.2	Reading the Source Files	8
5.3	Creating the Base Neighborhood Documents	8
5.4	Appending Points of Interest	9
5.5	Appending Home Prices	10
5.6	Storing into MongoDB	10
6	Queries	11
6.1	Query 1: Most Diverse Neighborhoods	13
6.2	Query 2: Demographic-Based Neighborhood Scoring	14
6.2.1	Function to Compute Score	14
6.2.2	Weights	16
6.2.3	Scores	17
6.3	Query 3: Neighborhood Matching Based on User Preferences	19

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 Comune di Milano Open Data 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`:

```
1 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:

```
1 [out:json][timeout:25];
2 area["name"="Milano"]["boundary"="administrative"]["admin_level"="8"]
3 ->.searchArea;
4 way["place"="quarter"](area.searchArea);
5 out body;
6 >;
7 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.

```
1 [out:json][timeout:25];
2 area["name"="Milano"]["boundary"="administrative"]->.searchArea;
3 way["place"="quarter"](area.searchArea);
4 relation["place"="quarter"](area.searchArea);
5 out body;
6 >;
7 out skel qt;
```

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

```
1 [out:json];
2 area["name"="Milano"]["boundary"="administrative"]->.searchArea;
3 (
4   node["shop"="supermarket"](area.searchArea);
5   way["shop"="supermarket"](area.searchArea);
6   relation["shop"="supermarket"](area.searchArea);
7 );
8 out body;
9 >;
10 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. Here below is an example of how data points about restaurants were retrieved:

```
1 HEADERS = {"Authorization": f"Bearer {API_KEY}"}
2 businesses_per_request = 50 # Maximum allowed by Yelp per request
3
4 def make_request(url, params=None):
5     # Makes a request to the Yelp API.
6     response = requests.get(url, headers=HEADERS, params=params)
7     response.raise_for_status()
8     return response
9
10 def search_businesses(location, term="restaurant", limit=businesses_per_request,
11    ↪ offset=0):
12     # Searches for businesses in a given location with pagination.
13     url = "https://api.yelp.com/v3/businesses/search"
14     params = {
15         "location": location,
16         "term": term,
17         "limit": limit,
18         "offset": offset
19     }
20     response = make_request(url, params=params)
21     return response.json().get("businesses", [])
22 # Initialize data storage
23 data = []
24
25 try:
26     # Loop through each neighborhood
27     for neighborhood in neighborhoods:
28         offset = 0
29         while True: # fetching data until no more results
30             print(f"Fetching businesses in {neighborhood} with offset: {offset}...")
31             try:
```

```
31         # fetch businesses using the current offset and location
32         businesses = search_businesses(location=neighborhood,
33         ↪ term="restaurant", limit=businesses_per_request, offset=offset)
34     if not businesses:
35         # no more businesses to fetch
36         print(f"No more businesses returned for {neighborhood}.")
37         break
38     for biz in businesses:
39         name = biz.get("name", None)
40         location_info = biz.get("location", {})
41         address = location_info.get("address1", None)
42         categories = biz.get("categories", [])
43         category_list = [cat.get("title", "") for cat in categories if
44         ↪ cat.get("title")]
45         category_str = ", ".join(category_list) if category_list else
46         ↪ None
47         rating = biz.get("rating", None)
48         review_count = biz.get("review_count", None)
49         price = biz.get("price", None)
50         coordinates = biz.get("coordinates", {})
51         latitude = coordinates.get("latitude", None)
52         longitude = coordinates.get("longitude", None)
53         # append to data
54         data.append({
55             "Fetch Location": neighborhood,
56             "Business Name": name,
57             "Business Address": address,
58             "Categories": category_str,
59             "Average Star Rating": rating,
60             "Review Count": review_count,
61             "Price": price,
62             "Latitude": latitude,
63             "Longitude": longitude
64         })
65         # increment offset for the next batch
66         offset += len(businesses)
67         # optional: sleep to respect API rate limits
68         time.sleep(0.5)
69         # break if the offset exceeds Yelp's maximum results per query
70         if offset >= 240: # Maximum 240 results per query
71             print(f"Reached maximum results for {neighborhood}.")
72             break
73     except requests.HTTPError as he:
74         # log the error and skip this neighborhood
75         print(f"HTTP error occurred for {neighborhood}: {he}")
76         break
```

```
74     # convert the collected data into a DataFrame
75     df = pd.DataFrame(data)
76     # convert DataFrame to GeoDataFrame
77     df["geometry"] = df.apply(
78         lambda row: Point(row["Longitude"], row["Latitude"]) if row["Longitude"] and
79         ↪ row["Latitude"] else None,
80         axis=1
81     )
82     Restaurants = gpd.GeoDataFrame(df, geometry="geometry", crs="EPSG:4326")
83     # print summary
84     print(Restaurants.head())
85     print(f"Total businesses collected: {len(data)}")
86 except Exception as e:
87     print(f"An unexpected error occurred: {e}")
```

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.

```
1  geojson_urls = [
2      # Insert here the urls for the geojson files
3  ]
4  output_directory = "geojson_files"
5  # Download and save each geojson file
6  for url in geojson_urls:
7      try:
8          response = requests.get(url)
9          response.raise_for_status()
10         file_name = os.path.basename(url)
11         output_path = os.path.join(output_directory, file_name)
12         with open(output_path, "wb") as file:
13             file.write(response.content)
14     except requests.exceptions.RequestException as e:
15         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 Preparation

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.

```
1 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 detailed 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:

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

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

```
1 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:

```
1 final = joined.groupby("Neighborhood").agg({
2     "Compr_min": "min",           # Minimum Compr_min for each Neighborhood
3     "Compr_max": "max",           # Maximum Compr_max for each Neighborhood
4     "Compr_mean": "mean",         # Average Compr_mean for each Neighborhood
```



```
5     "geometry": "first"  
6 }).reset_index()
```

5 Data 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:

```
1 from pymongo import MongoClient  
2  
3 client = MongoClient("mongodb://admin:DataMan2023!@localhost:27017/")  
4 db = client["my_database"]  
5 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:

```
1 import geopandas as gpd  
2  
3 gdf_combined = gpd.read_file("C:/path of the polygons file.geojson")  
4 PolyRestaurants = gpd.read_file("C:/path of the restaurants file.geojson")  
5 PolyMuseums = gpd.read_file("C:/path of the museums file.geojson")  
6 ...  
7 # other files  
8 ...
```

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:

```
1 neighborhood_docs = {}
2
3 for idx, row in gdf_combined.iterrows():
4     nb_name = row["Neighborhood"]
5     geo_json = mapping(row["geometry"])
6
7     neighborhood_docs[nb_name] = {
8         "_id": nb_name,
9         "neighborhood_name": nb_name,
10        "geometry": geo_json,
11        "locations": {
12            "restaurants": [],
13            "museums": [],
14            "nightlife": [],
15            ... # other POIs here ...
16        }
17        "home_prices": {
18            ... # home prices metrics ...
19        }
20    }
```

5.4 Appending Points of Interest

We define a function called `append_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`. It then converts the relevant columns in each row into a dictionary (`poi_data`) using `field_mappings` to standardize field names and appends each POI dictionary to the relevant list inside `neighborhood_docs[nb_name][field][poi_key]`.

```
1 def append_to_neighborhoods(field, gdf, poi_key, field_mappings=None):
2     if field_mappings is None:
3         field_mappings = {
4             col: col
5             for col in gdf.columns
6             if col not in ("Neighborhood", "geometry")
7         }
8     neighborhood
9     grouped = gdf.groupby("Neighborhood")
10    for nb_name, group_df in grouped:
11        if nb_name not in neighborhood_docs:
12            continue
13        needed_fields
14        for _, row in group_df.iterrows():
```

```
15     poi_data = {}
16     for src_col, dest_col in field_mappings.items():
17         if src_col in row:
18             poi_data[dest_col] = row[src_col]
19     correct field
20     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 and `field_mappings`. For instance, for restaurants:

```
1 append_to_neighborhoods(
2     field="locations",
3     gdf=PolyRestaurants,
4     poi_key="restaurants",
5     field_mappings={
6         "Business Name": "name",
7         "Business Address": "address",
8         "Categories": "category",
9         "Average Star Rating": "avg_star_rating",
10        "Review Count": "tot_ratings",
11        "Price": "price"
12    }
13 )
```

We repeat 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:

```
1 for idx, row in PolyHomePrices.iterrows():
2     nb_name = row["Neighborhood"]
3     if nb_name in neighborhood_docs:
4         neighborhood_docs[nb_name]["home_prices"]["min_price"] = row["Compr_min"]
5         neighborhood_docs[nb_name]["home_prices"]["max_price"] = row["Compr_max"]
6         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:

```
1 documents_to_insert = list(neighborhood_docs.values())
2 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 queries that can be done using PyMongo to interact with the database. Example 1 through 5 are very basic queries, while 6 through 8 are the ones we personally used for ranking and scoring the Neighborhoods. For results of such queries, either visit our in depth code in the [MongoDB Integration and Queries](#), or the summary in our report [Analysis and Ranking of Milan's Neighborhoods](#).

Before querying the database, ensure that you have established a connection as shown earlier:

```
1 from pymongo import MongoClient
2
3 client = MongoClient("mongodb://admin:DataMan2023!@localhost:27017/")
4 db = client["my_database"]
5 collection = db["neighborhoods"]
```

Example

1. Retrieve All Neighborhood Names

This query fetches the names of all neighborhoods in the database:

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

Example

2. Find a specific Neighborhood by name

This query retrieves a document for a specific neighborhood, such as "Duomo":

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

Example**3. Count the number of POIs in each Neighborhood**

This query calculates the total number of restaurants in each neighborhood:

```
1 cursor = collection.aggregate([
2     {"$project": {
3         "neighborhood_name": 1,
4         "restaurant_count": {"$size": "$locations.restaurants"}
5     }}
6 ])
7 for doc in cursor:
8     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:

```
1 cursor = collection.find(
2     {"$expr": {"$gte": [{"$size": "$locations.restaurants"}, 50]
3     }
4     }, {"_id": 0, "neighborhood_name": 1}
5 )
6 for doc in cursor:
7     print(doc["neighborhood_name"])
```

Example**5. Find the Most Expensive Neighborhoods**

This query sorts neighborhoods by their average home prices in descending order:

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

6.1 Query 1: Most Diverse Neighborhoods

```
1 pipeline = [  
2     {  
3         "$addFields": {  
4             "diversity_score": {  
5                 "$size": {  
6                     "$filter": {  
7                         "input": {"$objectToArray": "$locations"}, # convert  
7                         ↪ locations sub-document to array  
8                         "as": "amenity",  
9                         "cond": {"$gt": [{"$size": "$$amenity.v"}, 0]} # count  
9                         ↪ non-empty categories  
10                    }  
11                }  
12            }  
13        }  
14    },  
15    {"$sort": {"diversity_score": -1}}, # sort neighborhoods by diversity score  
15    ↪ (highest first)  
16    {"$limit": 5}, # show only the top 5 neighborhoods  
17    {"$project": { # project the fields to include in the output  
18        "neighborhood_name": 1,  
19        "diversity_score": 1  
20    }}  
21 ]  
22  
23 # query  
24 results = list(collection.aggregate(pipeline))  
25  
26 # results  
27 print("\n=== Top 5 Neighborhoods with the Most Diverse Amenities ===")  
28 for i, result in enumerate(results, start=1):  
29     print(f"{i}. {result['neighborhood_name']} (Diversity Score:  
29     ↪ {result['diversity_score']})")
```

6.2 Query 2: Demographic-Based Neighborhood Scoring

6.2.1 Function to Compute Score

```

1  def compute_score(neighborhood_doc, weights, price_weight, collection):
2      # Compute a score for a neighborhood, considering distinct POIs for certain
      ↪ categories.
3
4      # takes in input
5      #   neighborhood_doc: A MongoDB document with neighborhood data.
6      #   weights: A dictionary of weights for each POI category.
7      #   price_weight: Weight to apply to the normalized avg_price.
8      #   collection: The MongoDB collection to query for global min and max
      ↪ avg_price.
9
10     # Returns:
11     #   The total score for the neighborhood.
12
13     total_score = 0.0
14
15     # Categories requiring distinct filtering
16     distinct_categories = {"universities", "sportvenues", "schools"}
17
18     # Retrieve global min and max for each POI category to normalize the count
19     poi_stats = collection.aggregate([
20         {"$project": {
21             "poi_counts": {
22                 "$map": {
23                     "input": {"$objectToArray": "$locations"},
24                     "as": "poi",
25                     "in": {"k": "$$poi.k", "v": {"$size": {"$ifNull": ["$$poi.v",
      ↪ []]]}}}
26             }
27         }},
28         {"$unwind": "$poi_counts"},
29         {"$group": {
30             "_id": "$poi_counts.k",
31             "min_count": {"$min": "$poi_counts.v"},
32             "max_count": {"$max": "$poi_counts.v"}
33         }}
34     ])
35
36
37     # Convert the results into a dictionary
38     global_poi_min_max = {stat["_id"]: {"min": stat["min_count"], "max":
      ↪ stat["max_count"]} for stat in poi_stats}

```

```
39
40     # Normalize the POI counts and compute the scores
41     for category, weight in weights.items():
42         pois = neighborhood_doc.get("locations", {}).get(category, [])
43
44         if category in distinct_categories:
45             # For distinct categories, filter unique entries by address
46             unique_pois = {poi.get("address") for poi in pois if "address" in poi}
47             count = len(unique_pois)
48         else:
49             # Regular count for other categories
50             count = len(pois)
51
52         global_min = global_poi_min_max.get(category, {}).get("min", 0)
53         global_max = global_poi_min_max.get(category, {}).get("max", 1) # to avoid
54         ↪ division by zero
55
56         if global_max > global_min:
57             normalized_count = (count - global_min) / (global_max - global_min)
58         else:
59             normalized_count = 0.0
60
61         total_score += normalized_count * weight
62
63     # Price influence
64     avg_price = neighborhood_doc.get("home_prices", {}).get("avg_price")
65
66     # Retrieve global min and max prices from the database for normalization of the
67     ↪ avg price
68     price_stats = collection.aggregate([
69         {"$group": {
70             "_id": None,
71             "min_avg_price": {"$min": "$home_prices.avg_price"},
72             "max_avg_price": {"$max": "$home_prices.avg_price"}
73         }}
74     ])
75     price_stats = next(price_stats, None)
76     min_avg_price = price_stats["min_avg_price"]
77     max_avg_price = price_stats["max_avg_price"]
78
79     normalized_price = (avg_price - min_avg_price) / (max_avg_price - min_avg_price)
80     total_score -= normalized_price * price_weight
81
82     return total_score
```


6.2.2 Weights

```
1  # Example weighting dictionaries (tweak as you wish)
2
3  students_weights = {
4      "restaurants": 2.0,
5      "museums": 5.0,
6      "nightlife": 8.0,
7      "dogparks": 1.0,
8      "pharmacies": 6.0,
9      "playgrounds": 6.0,
10     "sportvenues": 8.0,
11     "schools": 1.0,
12     "universities": 10.0,
13     "coworking": 7.0,
14     "libraries": 9.0,
15     "supermarkets": 9.0,
16     "transport": 10.0
17 }
18 # price weight
19 price_weight_students = 10.0
20
21
22 single_couples_weights = {
23     "restaurants": 7.0,
24     "museums": 5.0,
25     "nightlife": 8.0,
26     "dogparks": 5.0,
27     "pharmacies": 6.0,
28     "playgrounds": 1.0,
29     "sportvenues": 7.0,
30     "schools": 1.0,
31     "universities": 1.0,
32     "coworking": 8.0,
33     "libraries": 5.0,
34     "supermarkets": 10.0,
35     "transport": 10.0
36 }
37 # price weight
38 price_weight_single_couples = 6.0
39
40
41 families_weights = {
42     "restaurants": 1.0,
43     "museums": 6.0,
44     "nightlife": 1.0,
```

```
45     "dogparks": 10.0,
46     "pharmacies": 7.0,
47     "playgrounds": 10.0,
48     "sportvenues": 3.0,
49     "schools": 10.0,
50     "universities": 1.0,
51     "coworking": 1.0,
52     "libraries": 8.0,
53     "supermarkets": 8.0,
54     "transport": 4.0
55 }
56 # price weight
57 price_weight_families = 7.5
```

6.2.3 Scores

```
1  # Read all neighborhoods
2  all_neighborhoods = list(collection.find({}))
3
4  # --- Ranking for Students ---
5  print("=== Ranking for Students ===")
6  students_scores = []
7  for nb in all_neighborhoods:
8      score = compute_score(nb, students_weights, price_weight= price_weight_students,
9                             ↪ collection=collection)
9      students_scores.append({
10         "neighborhood_name": nb["neighborhood_name"],
11         "score": score
12     })
13
14  # sort by score descending
15  students_scores.sort(key=lambda x: x["score"], reverse=True)
16
17  # transform scores into percentages
18  if students_scores:
19      max_score_students = students_scores[0]["score"]
20      for item in students_scores: # sort of scaling
21         item["percentage"] = (item["score"] / max_score_students) * 100 if
22         ↪ max_score_students > 0 else 0
23
24  # showing the top 5 neighborhoods
25  for rank, item in enumerate(students_scores[:5], start=1):
26      print(f"{rank}. {item['neighborhood_name']} => {item['percentage']:.2f}%")
```

```
26
27
28 # --- Ranking for Singles/Couples ---
29 print("\n=== Ranking for Singles/Couples ===")
30 single_couples_scores = []
31 for nb in all_neighborhoods:
32     score = compute_score(nb, single_couples_weights, price_weight=
33         ↪ price_weight_single_couples, collection=collection)
34     single_couples_scores.append({
35         "neighborhood_name": nb["neighborhood_name"],
36         "score": score
37     })
38 single_couples_scores.sort(key=lambda x: x["score"], reverse=True)
39
40 # transform scores into percentages
41 if single_couples_scores:
42     max_score_single_couples = single_couples_scores[0]["score"]
43     for item in single_couples_scores: # sort of scaling
44         item["percentage"] = (item["score"] / max_score_single_couples) * 100 if
45         ↪ max_score_single_couples > 0 else 0
46
47 # showing the top 5 neighborhoods
48 for rank, item in enumerate(single_couples_scores[:5], start=1):
49     print(f"{rank}. {item['neighborhood_name']} => {item['percentage']:.2f}%")
50
51 # --- Ranking for Families ---
52 print("\n=== Ranking for Families ===")
53 families_scores = []
54 for nb in all_neighborhoods:
55     score = compute_score(nb, families_weights, price_weight= price_weight_families,
56         ↪ collection=collection)
57     families_scores.append({
58         "neighborhood_name": nb["neighborhood_name"],
59         "score": score
60     })
61 families_scores.sort(key=lambda x: x["score"], reverse=True)
62
63 # transform scores into percentages
64 if families_scores:
65     max_score_families = families_scores[0]["score"]
66     for item in families_scores: # sort of scaling
67         item["percentage"] = (item["score"] / max_score_families) * 100 if
68         ↪ max_score_families > 0 else 0
```

```
68
69 # showing the top 5 neighborhoods
70 for rank, item in enumerate(families_scores[:5], start=1):
71     print(f"{rank}. {item['neighborhood_name']} => {item['percentage']:.2f}%")
```

6.3 Query 3: Neighborhood Matching Based on User Preferences

```
1 # Interactive user inputs
2 print("\nPlease specify your preferences:")
3
4 faculty = input("Enter the faculty you're looking for (e.g., Economia):
   ↳ ").strip().upper() # convert to uppercase
5 parks_required = input("Do you need a dog park? (yes/no): ").strip().lower() == "yes"
6 library_required = input("Do you need a library? (yes/no): ").strip().lower() ==
   ↳ "yes"
7
8 restaurant_category = input("Enter the type of restaurant you prefer (e.g., Italian):
   ↳ ").strip()
9 min_restaurants = int(input("Enter the minimum number of restaurants you want:
   ↳ ").strip() or 0)
10
11 coworking_required = input("Do you need a coworking space? (yes/no):
   ↳ ").strip().lower() == "yes"
12 sport_venue_required = input("Do you need a sport venue? (yes/no): ").strip().lower()
   ↳ == "yes"
13 sport_venue_category = input("Enter the sport venue category (e.g., Piscina,
   ↳ Atletica) [optional]: ").strip().upper() if sport_venue_required else None
14 supermarket_required = input("Do you need a supermarket? (yes/no): ").strip().lower()
   ↳ == "yes"
15 museum_required = input("Do you need a museum? (yes/no): ").strip().lower() == "yes"
16 pharmacy_required = input("Do you need a pharmacy? (yes/no): ").strip().lower() ==
   ↳ "yes"
17 playground_required = input("Do you need a playground? (yes/no): ").strip().lower()
   ↳ == "yes"
18
19 transport_required = input("Do you need public transport? (yes/no):
   ↳ ").strip().lower() == "yes"
20 metro_required = train_required = bus_required = False
21 if transport_required:
22     metro_required = input("Do you need metro service? (yes/no): ").strip().lower()
   ↳ == "yes"
23     train_required = input("Do you need train service? (yes/no): ").strip().lower()
   ↳ == "yes"
```

```
24     bus_required = input("Do you need bus service? (yes/no): ").strip().lower() ==  
    ↪     "yes"  
25  
26 budget = float(input("Enter your budget for home prices (price in euros per square  
    ↪ meter): ").strip() or 0)  
27  
28 # query  
29 query = {  
30     "$addFields": {  
31         "match_score": {  
32             "$add": [  
33                 # check for faculty match in university  
34                 {"$cond": [  
35                     {"$in": [faculty, "$locations.universities.faculty"]}, 1, 0  
36                 ]} if faculty else 0,  
37                 # check for restaurant category match  
38                 {"$cond": [  
39                     {"$in": [restaurant_category,  
    ↪                     "$locations.restaurants.category"]}, 1, 0  
40                 ]} if restaurant_category else 0,  
41                 # check for minimum number of restaurants  
42                 {"$cond": [  
43                     {"$gte": [{"$size": "$locations.restaurants"}, min_restaurants]},  
    ↪                     1, 0  
44                 ]} if min_restaurants > 0 else 0,  
45                 # check for parks presence  
46                 {"$cond": [  
47                     {"$gt": [{"$size": "$locations.dogparks"}, 0]}, 1, 0  
48                 ]} if parks_required else 0,  
49                 # check for libraries presence  
50                 {"$cond": [  
51                     {"$gt": [{"$size": "$locations.libraries"}, 0]}, 1, 0  
52                 ]} if library_required else 0,  
53                 # check for coworking presence  
54                 {"$cond": [  
55                     {"$gt": [{"$size": "$locations.coworking"}, 0]}, 1, 0  
56                 ]} if coworking_required else 0,  
57                 # check for sport venue presence  
58                 {"$cond": [  
59                     {"$gt": [{"$size": "$locations.sportvenues"}, 0]}, 1, 0  
60                 ]} if sport_venue_required else 0,  
61                 # check for specific sport venue category match  
62                 {"$cond": [  
63                     {"$in": [sport_venue_category,  
    ↪                     "$locations.sportvenues.category"]}, 1, 0  
64                 ]} if sport_venue_category else 0,
```

```

65         # check for supermarket presence
66         {"$cond": [
67             {"$gt": [{"$size": "$locations.supermarkets"}, 0]}, 1, 0
68         ]} if supermarket_required else 0,
69         # check for museum presence
70         {"$cond": [
71             {"$gt": [{"$size": "$locations.museums"}, 0]}, 1, 0
72         ]} if museum_required else 0,
73         # check for pharmacy presence
74         {"$cond": [
75             {"$gt": [{"$size": "$locations.pharmacies"}, 0]}, 1, 0
76         ]} if pharmacy_required else 0,
77         # check for playground presence
78         {"$cond": [
79             {"$gt": [{"$size": "$locations.playgrounds"}, 0]}, 1, 0
80         ]} if playground_required else 0,
81         # check for public transport
82         {"$cond": [
83             {"$or": [
84                 {"$in": ["Metro", "$locations.transport.transport_type"]} if
85                 ↪ metro_required else False,
86                 {"$in": ["Treno", "$locations.transport.transport_type"]} if
87                 ↪ train_required else False,
88                 {"$in": ["Bus", "$locations.transport.transport_type"]} if
89                 ↪ bus_required else False
90             ]}, 1, 0
91         ]} if transport_required else 0,
92         # check for budget in home prices
93         {"$cond": [
94             {"$lte": ["$home_prices.avg_price", budget]}, 1, 0
95         ]} if budget > 0 else 0
96     ]
97 }
98
99 # running the aggregation pipeline
100 pipeline = [
101     {"$match": {
102         "home_prices.avg_price": {"$lte": 2 * budget} # keep neighborhoods within
103         ↪ twice the budget
104     }},
105     query, # add the match_score field
106     {"$sort": {
107         "match_score": -1, # higher match scores first

```

```
107         "home_prices.avg_price": 1 # lower avg prices first
108     }
109 },
110 {"$limit": 3}, # limit to the top 3 neighborhoods
111 {"$project": { # project only relevant fields for output
112     "neighborhood_name": 1,
113     "match_score": 1,
114     "locations": 1,
115     "home_prices.avg_price": 1
116 }}
117 ]
118
119 results = list(collection.aggregate(pipeline))
120 # Output the top neighborhoods
121 print("\n=== Top 3 Suitable Neighborhoods ===")
122 if results:
123     for i, neighborhood in enumerate(results, start=1):
124         print(f"\n{i}. {neighborhood['neighborhood_name']} (Score:
125             ↪ {neighborhood['match_score']})")
126         print(f"    - Average Price: €{neighborhood.get('home_prices',
127             ↪ {}).get('avg_price', 'N/A')}")
128
129     fulfilled = []
130     not_fulfilled = []
131
132     # Faculty check
133     if faculty:
134         if any(faculty in uni.get("faculty", []) for uni in
135             ↪ neighborhood["locations"].get("universities", [])):
136             fulfilled.append("Faculty")
137         else:
138             not_fulfilled.append("Faculty")
139
140     # Restaurant category check
141     if restaurant_category:
142         if any(restaurant_category in rest.get("category", []) for rest in
143             ↪ neighborhood["locations"].get("restaurants", [])):
144             fulfilled.append("Restaurant Category")
145         else:
146             not_fulfilled.append("Restaurant Category")
147
148     # Minimum number of restaurants
149     if min_restaurants > 0:
150         if len(neighborhood["locations"].get("restaurants", [])) >=
151             ↪ min_restaurants:
152             fulfilled.append("Minimum Number of Restaurants")
```

```
148         else:
149             not_fulfilled.append("Minimum Number of Restaurants")
150
151     # Parks check
152     if parks_required:
153         if len(neighborhood["locations"].get("dogparks", [])) > 0:
154             fulfilled.append("Dog Park")
155         else:
156             not_fulfilled.append("Dog Park")
157
158     # Library check
159     if library_required:
160         if len(neighborhood["locations"].get("libraries", [])) > 0:
161             fulfilled.append("Library")
162         else:
163             not_fulfilled.append("Library")
164
165     # Coworking check
166     if coworking_required:
167         if len(neighborhood["locations"].get("coworking", [])) > 0:
168             fulfilled.append("Coworking Space")
169         else:
170             not_fulfilled.append("Coworking Space")
171
172     # Sport venue check
173     if sport_venue_required:
174         if len(neighborhood["locations"].get("sportvenues", [])) > 0:
175             fulfilled.append("Sport Venue")
176         else:
177             not_fulfilled.append("Sport Venue")
178
179     # Sport venue category check
180     if sport_venue_category:
181         if any(sport_venue_category in venue.get("category", []) for venue in
182             ↪ neighborhood["locations"].get("sportvenues", [])):
183             fulfilled.append("Specific Sport Venue Category")
184         else:
185             not_fulfilled.append("Specific Sport Venue Category")
186
187     # Supermarket check
188     if supermarket_required:
189         if len(neighborhood["locations"].get("supermarkets", [])) > 0:
190             fulfilled.append("Supermarket")
191         else:
192             not_fulfilled.append("Supermarket")
```



```
193     # Museum check
194     if museum_required:
195         if len(neighborhood["locations"].get("museums", [])) > 0:
196             fulfilled.append("Museum")
197         else:
198             not_fulfilled.append("Museum")
199
200     # Pharmacy check
201     if pharmacy_required:
202         if len(neighborhood["locations"].get("pharmacies", [])) > 0:
203             fulfilled.append("Pharmacy")
204         else:
205             not_fulfilled.append("Pharmacy")
206
207     # Playground check
208     if playground_required:
209         if len(neighborhood["locations"].get("playgrounds", [])) > 0:
210             fulfilled.append("Playground")
211         else:
212             not_fulfilled.append("Playground")
213
214     # Transport check
215     if transport_required:
216         transport_fulfilled = []
217         transport_list = neighborhood["locations"].get("transport", [])
218
219         # Iterate through transport_list to find the required transport types
220         if metro_required and any("Metro" in transport.get("transport_type", []))
221         ↪ for transport in transport_list:
222             transport_fulfilled.append("Metro")
223         if train_required and any("Treno" in transport.get("transport_type", []))
224         ↪ for transport in transport_list:
225             transport_fulfilled.append("Train")
226         if bus_required and any("Bus" in transport.get("transport_type", [])) for
227         ↪ transport in transport_list:
228             transport_fulfilled.append("Bus")
229
230         # If any transport types are fulfilled, add them to fulfilled; otherwise,
231         ↪ add to not_fulfilled
232         if transport_fulfilled:
233             fulfilled.append(f"Transport ({', '.join(transport_fulfilled)})")
234         else:
235             not_fulfilled.append("Transport")
236
237     # Budget check
238     if budget > 0:
```

```
235     avg_price = neighborhood.get("home_prices", {}).get("avg_price",  
    ↪     float("inf"))  
236     if avg_price <= budget:  
237         fulfilled.append("Budget")  
238     else:  
239         not_fulfilled.append("Budget")  
240  
241     # Print fulfilled and not fulfilled  
242     print(" - Fulfilled:", ", ".join(fulfilled) if fulfilled else "None")  
243     print(" - Not Fulfilled:", ", ".join(not_fulfilled) if not_fulfilled else  
    ↪     "None")  
244 else:  
245     print("No neighborhoods match your criteria.")
```