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

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. Here below is an example of how data points about restaurants were retrieved:

```python
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", [])
# Initialize data storage
data = []

try:
    # Loop through each neighborhood
    for neighborhood in neighborhoods:
        offset = 0
        while True: # fetching data until no more results
            print(f"Fetching businesses in {neighborhood} with offset: {offset}...")
            try:
                # fetch businesses using the current offset and location
                businesses = search_businesses(location=neighborhood,
                ↪   term="restaurant", limit=businesses_per_request, offset=offset)
                if not businesses:
                    # no more businesses to fetch
                    print(f"No more businesses returned for {neighborhood}.")
                    break
                for biz in businesses:
                    name = biz.get("name", None)
                    location_info = biz.get("location", {})
                    address = location_info.get("address1", None)
                    categories = biz.get("categories", [])
                    category_list = [cat.get("title", "") for cat in categories if
                    ↪   cat.get("title")]
                    category_str = ", ".join(category_list) if category_list else
                    ↪   None
                    rating = biz.get("rating", None)
                    review_count = biz.get("review_count", None)
                    price = biz.get("price", None)
                    coordinates = biz.get("coordinates", {})
                    latitude = coordinates.get("latitude", None)
                    longitude = coordinates.get("longitude", None)
                    # append to data
                    data.append({
                        "Fetch Location": neighborhood,
                        "Business Name": name,
                        "Business Address": address,
                        "Categories": category_str,
                        "Average Star Rating": rating,
                        "Review Count": review_count,
                        "Price": price,
                        "Latitude": latitude,
                        "Longitude": longitude
                    })
                # increment offset for the next batch
                offset += len(businesses)
                # optional: sleep to respect API rate limits
                time.sleep(0.5)
                # break if the offset exceeds Yelp's maximum results per query
                if offset >= 240:  # Maximum 240 results per query
                    print(f"Reached maximum results for {neighborhood}.")
                    break
            except requests.HTTPError as he:
                # log the error and skip this neighborhood
                print(f"HTTP error occurred for {neighborhood}: {he}")
                break
```

```python
    # convert the collected data into a DataFrame
    df = pd.DataFrame(data)
    # convert DataFrame to GeoDataFrame
    df["geometry"] = df.apply(
        lambda row: Point(row["Longitude"], row["Latitude"]) if row["Longitude"] and row["Latitude"] else None,
        axis=1
    )
    Restaurants = gpd.GeoDataFrame(df, geometry="geometry", crs="EPSG:4326")
    # print summary
    print(Restaurants.head())
    print(f"Total businesses collected: {len(data)}")
except Exception as e:
    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.

```python
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 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.

```
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 enriched 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 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. 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]["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 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:

```
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 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:

```
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")
```

6.1 Query 1: Most Diverse Neighborhoods

```python
pipeline = [
    {
        "$addFields": {
            "diversity_score": {
                "$size": {
                    "$filter": {
                        "input": {"$objectToArray": "$locations"}, # convert locations sub-document to array
                        "as": "amenity",
                        "cond": {"$gt": [{"$size": "$$amenity.v"}, 0]} # count non-empty categories
                    }
                }
            }
        }
    },
    {"$sort": {"diversity_score": -1}}, # sort neighborhoods by diversity score (highest first)
    {"$limit": 5}, # show only the top 5 neighborhoods
    {"$project": { # project the fields to include in the output
        "neighborhood_name": 1,
        "diversity_score": 1
    }}
]

# query
results = list(collection.aggregate(pipeline))

# results
print("\n=== Top 5 Neighborhoods with the Most Diverse Amenities ===")
for i, result in enumerate(results, start=1):
    print(f"{i}. {result['neighborhood_name']} (Diversity Score: {result['diversity_score']})")
```

6.2 Query 2: Demographic-Based Neighborhood Scoring

6.2.1 Function to Compute Score

```python
def compute_score(neighborhood_doc, weights, price_weight, collection):
    # Compute a score for a neighborhood, considering distinct POIs for certain categories.

    # takes in input
    #    neighborhood_doc: A MongoDB document with neighborhood data.
    #    weights: A dictionary of weights for each POI category.
    #    price_weight: Weight to apply to the normalized avg_price.
    #    collection: The MongoDB collection to query for global min and max avg_price.

    # Returns:
    #    The total score for the neighborhood.

    total_score = 0.0

    # Categories requiring distinct filtering
    distinct_categories = {"universities", "sportvenues", "schools"}

    # Retrieve global min and max for each POI category to normalize the count
    poi_stats = collection.aggregate([
        {"$project": {
            "poi_counts": {
                "$map": {
                    "input": {"$objectToArray": "$locations"},
                    "as": "poi",
                    "in": {"k": "$$poi.k", "v": {"$size": {"$ifNull": ["$$poi.v", []]}}}
                }
            }
        }},
        {"$unwind": "$poi_counts"},
        {"$group": {
            "_id": "$poi_counts.k",
            "min_count": {"$min": "$poi_counts.v"},
            "max_count": {"$max": "$poi_counts.v"}
        }}
    ])

    # Convert the results into a dictionary
    global_poi_min_max = {stat["_id"]: {"min": stat["min_count"], "max": stat["max_count"]} for stat in poi_stats}

    # Normalize the POI counts and compute the scores
    for category, weight in weights.items():
        pois = neighborhood_doc.get("locations", {}).get(category, [])

        if category in distinct_categories:
            # For distinct categories, filter unique entries by address
            unique_pois = {poi.get("address") for poi in pois if "address" in poi}
            count = len(unique_pois)
        else:
            # Regular count for other categories
            count = len(pois)

        global_min = global_poi_min_max.get(category, {}).get("min", 0)
        global_max = global_poi_min_max.get(category, {}).get("max", 1) # to avoid division by zero

        if global_max > global_min:
            normalized_count = (count - global_min) / (global_max - global_min)
        else:
            normalized_count = 0.0

        total_score += normalized_count * weight

    # Price influence
    avg_price = neighborhood_doc.get("home_prices", {}).get("avg_price")

    # Retrieve global min and max prices from the database for normalization of the avg price
    price_stats = collection.aggregate([
        {"$group": {
            "_id": None,
            "min_avg_price": {"$min": "$home_prices.avg_price"},
            "max_avg_price": {"$max": "$home_prices.avg_price"}
        }}
    ])
    price_stats = next(price_stats, None)
    min_avg_price = price_stats["min_avg_price"]
    max_avg_price = price_stats["max_avg_price"]

    normalized_price = (avg_price - min_avg_price) / (max_avg_price - min_avg_price)
    total_score -= normalized_price * price_weight

    return total_score
```

6.2.2 Weights

```python
# Example weighting dictionaries (tweak as you wish)

students_weights = {
    "restaurants": 2.0,
    "museums": 5.0,
    "nightlife": 8.0,
    "dogparks": 1.0,
    "pharmacies": 6.0,
    "playgrounds": 6.0,
    "sportvenues": 8.0,
    "schools": 1.0,
    "universities": 10.0,
    "coworking": 7.0,
    "libraries": 9.0,
    "supermarkets": 9.0,
    "transport": 10.0
}
# price weight
price_weight_students = 10.0


single_couples_weights = {
    "restaurants": 7.0,
    "museums": 5.0,
    "nightlife": 8.0,
    "dogparks": 5.0,
    "pharmacies": 6.0,
    "playgrounds": 1.0,
    "sportvenues": 7.0,
    "schools": 1.0,
    "universities": 1.0,
    "coworking": 8.0,
    "libraries": 5.0,
    "supermarkets": 10.0,
    "transport": 10.0
}
# price weight
price_weight_single_couples = 6.0


families_weights = {
    "restaurants": 1.0,
    "museums": 6.0,
    "nightlife": 1.0,
    "dogparks": 10.0,
    "pharmacies": 7.0,
    "playgrounds": 10.0,
    "sportvenues": 3.0,
    "schools": 10.0,
    "universities": 1.0,
    "coworking": 1.0,
    "libraries": 8.0,
    "supermarkets": 8.0,
    "transport": 4.0
}
# price weight
price_weight_families = 7.5
```

6.2.3 Scores

```
# Read all neighborhoods
all_neighborhoods = list(collection.find({}))

# --- Ranking for Students ---
print("=== Ranking for Students ===")
students_scores = []
for nb in all_neighborhoods:
    score = compute_score(nb, students_weights, price_weight= price_weight_students, collection=collection)
    students_scores.append({
        "neighborhood_name": nb["neighborhood_name"],
        "score": score
    })

# sort by score descending
students_scores.sort(key=lambda x: x["score"], reverse=True)

# transform scores into percentages
if students_scores:
    max_score_students = students_scores[0]["score"]
    for item in students_scores: # sort of scaling
        item["percentage"] = (item["score"] / max_score_students) * 100 if max_score_students > 0 else 0

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

```python


# --- Ranking for Singles/Couples ---
print("\n=== Ranking for Singles/Couples ===")
single_couples_scores = []
for nb in all_neighborhoods:
    score = compute_score(nb, single_couples_weights, price_weight=
    ↪ price_weight_single_couples, collection=collection)
    single_couples_scores.append({
        "neighborhood_name": nb["neighborhood_name"],
        "score": score
    })

single_couples_scores.sort(key=lambda x: x["score"], reverse=True)

# transform scores into percentages
if single_couples_scores:
    max_score_single_couples = single_couples_scores[0]["score"]
    for item in single_couples_scores: # sort of scaling
        item["percentage"] = (item["score"] / max_score_single_couples) * 100 if
        ↪ max_score_single_couples > 0 else 0

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


# --- Ranking for Families ---
print("\n=== Ranking for Families ===")
families_scores = []
for nb in all_neighborhoods:
    score = compute_score(nb, families_weights, price_weight= price_weight_families,
    ↪ collection=collection)
    families_scores.append({
        "neighborhood_name": nb["neighborhood_name"],
        "score": score
    })

families_scores.sort(key=lambda x: x["score"], reverse=True)

# transform scores into percentages
if families_scores:
    max_score_families = families_scores[0]["score"]
    for item in families_scores: # sort of scaling
        item["percentage"] = (item["score"] / max_score_families) * 100 if
        ↪ max_score_families > 0 else 0
```

```

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

6.3 Query 3: Neighborhood Matching Based on User Preferences

```
# Interactive user inputs
print("\nPlease specify your preferences:")

faculty = input("Enter the faculty you're looking for (e.g., Economia):
↪  ").strip().upper() # convert to uppercase
parks_required = input("Do you need a dog park? (yes/no): ").strip().lower() == "yes"
library_required = input("Do you need a library? (yes/no): ").strip().lower() ==
↪  "yes"

restaurant_category = input("Enter the type of restaurant you prefer (e.g., Italian):
↪  ").strip()
min_restaurants = int(input("Enter the minimum number of restaurants you want:
↪  ").strip() or 0)

coworking_required = input("Do you need a coworking space? (yes/no):
↪  ").strip().lower() == "yes"
sport_venue_required = input("Do you need a sport venue? (yes/no): ").strip().lower()
↪  == "yes"
sport_venue_category = input("Enter the sport venue category (e.g., Piscina,
↪  Atletica) [optional]: ").strip().upper() if sport_venue_required else None
supermarket_required = input("Do you need a supermarket? (yes/no): ").strip().lower()
↪  == "yes"
museum_required = input("Do you need a museum? (yes/no): ").strip().lower() == "yes"
pharmacy_required = input("Do you need a pharmacy? (yes/no): ").strip().lower() ==
↪  "yes"
playground_required = input("Do you need a playground? (yes/no): ").strip().lower()
↪  == "yes"

transport_required = input("Do you need public transport? (yes/no):
↪  ").strip().lower() == "yes"
metro_required = train_required = bus_required = False
if transport_required:
    metro_required = input("Do you need metro service? (yes/no): ").strip().lower()
    ↪  == "yes"
    train_required = input("Do you need train service? (yes/no): ").strip().lower()
    ↪  == "yes"
    bus_required = input("Do you need bus service? (yes/no): ").strip().lower() ==
    ↪   "yes"

budget = float(input("Enter your budget for home prices (price in euros per square
↪   meter): ").strip() or 0)

# query
query = {
    "$addFields": {
        "match_score": {
            "$add": [
                # check for faculty match in university
                {"$cond": [
                    {"$in": [faculty, "$locations.universities.faculty"]}, 1, 0
                ]} if faculty else 0,
                # check for restaurant category match
                {"$cond": [
                    {"$in": [restaurant_category,
                    ↪   "$locations.restaurants.category"]}, 1, 0
                ]} if restaurant_category else 0,
                # check for minimum number of restaurants
                {"$cond": [
                    {"$gte": [{"$size": "$locations.restaurants"}, min_restaurants]},
                    ↪   1, 0
                ]} if min_restaurants > 0 else 0,
                # check for parks presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.dogparks"}, 0]}, 1, 0
                ]} if parks_required else 0,
                # check for libraries presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.libraries"}, 0]}, 1, 0
                ]} if library_required else 0,
                # check for coworking presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.coworking"}, 0]}, 1, 0
                ]} if coworking_required else 0,
                # check for sport venue presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.sportvenues"}, 0]}, 1, 0
                ]} if sport_venue_required else 0,
                # check for specific sport venue category match
                {"$cond": [
                    {"$in": [sport_venue_category,
                    ↪   "$locations.sportvenues.category"]}, 1, 0
                ]} if sport_venue_category else 0,
                # check for supermarket presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.supermarkets"}, 0]}, 1, 0
                ]} if supermarket_required else 0,
                # check for museum presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.museums"}, 0]}, 1, 0
                ]} if museum_required else 0,
                # check for pharmacy presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.pharmacies"}, 0]}, 1, 0
                ]} if pharmacy_required else 0,
                # check for playground presence
                {"$cond": [
                    {"$gt": [{"$size": "$locations.playgrounds"}, 0]}, 1, 0
                ]} if playground_required else 0,
                # check for public transport
                {"$cond": [
                    {"$or": [
                        {"$in": ["Metro", "$locations.transport.transport_type"]} if metro_required else False,
                        {"$in": ["Treno", "$locations.transport.transport_type"]} if train_required else False,
                        {"$in": ["Bus", "$locations.transport.transport_type"]} if bus_required else False
                    ]}, 1, 0
                ]} if transport_required else 0,
                # check for budget in home prices
                {"$cond": [
                    {"$lte": ["$home_prices.avg_price", budget]}, 1, 0
                ]} if budget > 0 else 0
            ]
        }
    }
}

# running the aggregation pipeline
pipeline = [
    {"$match": {
            "home_prices.avg_price": {"$lte": 2 * budget} # keep neighborhoods within twice the budget
        }
    },
    query, # add the match_score field
    {"$sort": {
            "match_score": -1,  # higher match scores first
            "home_prices.avg_price": 1  # lower avg prices first
        }
    },
    {"$limit": 3},  # limit to the top 3 neighborhoods
    {"$project": {  # project only relevant fields for output
        "neighborhood_name": 1,
        "match_score": 1,
        "locations": 1,
        "home_prices.avg_price": 1
    }}
]

results = list(collection.aggregate(pipeline))
# Output the top neighborhoods
print("\n=== Top 3 Suitable Neighborhoods ===")
if results:
    for i, neighborhood in enumerate(results, start=1):
        print(f"\n{i}. {neighborhood['neighborhood_name']} (Score:
        ↪   {neighborhood['match_score']})")
        print(f"  - Average Price: €{neighborhood.get('home_prices',
        ↪   {}).get('avg_price', 'N/A')}")

        fulfilled = []
        not_fulfilled = []

        # Faculty check
        if faculty:
            if any(faculty in uni.get("faculty", []) for uni in
            ↪   neighborhood["locations"].get("universities", [])):
                fulfilled.append("Faculty")
            else:
                not_fulfilled.append("Faculty")

        # Restaurant category check
        if restaurant_category:
            if any(restaurant_category in rest.get("category", []) for rest in
            ↪   neighborhood["locations"].get("restaurants", [])):
                fulfilled.append("Restaurant Category")
            else:
                not_fulfilled.append("Restaurant Category")

        # Minimum number of restaurants
        if min_restaurants > 0:
            if len(neighborhood["locations"].get("restaurants", [])) >=
            ↪   min_restaurants:
                fulfilled.append("Minimum Number of Restaurants")
            else:
                not_fulfilled.append("Minimum Number of Restaurants")

        # Parks check
        if parks_required:
            if len(neighborhood["locations"].get("dogparks", [])) > 0:
                fulfilled.append("Dog Park")
            else:
                not_fulfilled.append("Dog Park")

        # Library check
        if library_required:
            if len(neighborhood["locations"].get("libraries", [])) > 0:
                fulfilled.append("Library")
            else:
                not_fulfilled.append("Library")

        # Coworking check
        if coworking_required:
            if len(neighborhood["locations"].get("coworking", [])) > 0:
                fulfilled.append("Coworking Space")
            else:
                not_fulfilled.append("Coworking Space")

        # Sport venue check
        if sport_venue_required:
            if len(neighborhood["locations"].get("sportvenues", [])) > 0:
                fulfilled.append("Sport Venue")
            else:
                not_fulfilled.append("Sport Venue")

        # Sport venue category check
        if sport_venue_category:
            if any(sport_venue_category in venue.get("category", []) for venue in
             ↪  neighborhood["locations"].get("sportvenues", [])):
                fulfilled.append("Specific Sport Venue Category")
            else:
                not_fulfilled.append("Specific Sport Venue Category")

        # Supermarket check
        if supermarket_required:
            if len(neighborhood["locations"].get("supermarkets", [])) > 0:
                fulfilled.append("Supermarket")
            else:
                not_fulfilled.append("Supermarket")

        # Museum check
        if museum_required:
            if len(neighborhood["locations"].get("museums", [])) > 0:
                fulfilled.append("Museum")
            else:
                not_fulfilled.append("Museum")

        # Pharmacy check
        if pharmacy_required:
            if len(neighborhood["locations"].get("pharmacies", [])) > 0:
                fulfilled.append("Pharmacy")
            else:
                not_fulfilled.append("Pharmacy")

        # Playground check
        if playground_required:
            if len(neighborhood["locations"].get("playgrounds", [])) > 0:
                fulfilled.append("Playground")
            else:
                not_fulfilled.append("Playground")

        # Transport check
        if transport_required:
            transport_fulfilled = []
            transport_list = neighborhood["locations"].get("transport", [])

            # Iterate through transport_list to find the required transport types
            if metro_required and any("Metro" in transport.get("transport_type", [])
            ↪   for transport in transport_list):
                transport_fulfilled.append("Metro")
            if train_required and any("Treno" in transport.get("transport_type", [])
            ↪   for transport in transport_list):
                transport_fulfilled.append("Train")
            if bus_required and any("Bus" in transport.get("transport_type", []) for
            ↪   transport in transport_list):
                transport_fulfilled.append("Bus")

            # If any transport types are fulfilled, add them to fulfilled; otherwise,
            ↪   add to not_fulfilled
            if transport_fulfilled:
                fulfilled.append(f"Transport ({', '.join(transport_fulfilled)})")
            else:
                not_fulfilled.append("Transport")

        # Budget check
        if budget > 0:
            avg_price = neighborhood.get("home_prices", {}).get("avg_price", float("inf"))
            if avg_price <= budget:
                fulfilled.append("Budget")
            else:
                not_fulfilled.append("Budget")

        # Print fulfilled and not fulfilled
        print("  - Fulfilled:", ", ".join(fulfilled) if fulfilled else "None")
        print("  - Not Fulfilled:", ", ".join(not_fulfilled) if not_fulfilled else "None")
else:
    print("No neighborhoods match your criteria.")
```