src/data_retrieval.py

```python
"""
data_retrieval.py

Retrieval of raw OSM data for a city.
Includes loading of administrative boundaries, street networks,
railways, water features and POIs.
"""


from pyrosm import OSM, get_data
from shapely.geometry import Polygon, MultiPolygon

# Mapping from city name to (OSM boundary name, admin_level)
# https://wiki.openstreetmap.org/wiki/Template:Admin_level
CITY_CONFIG = {
    "Copenhagen": {"boundary_name": "Københavns Kommune", "admin_level": 7},
    "Gdansk": {"boundary_name": "Gdańsk", "admin_level": 8},
    # Add more cities here
}

# Remove interior holes from polygons
def drop_holes(geom):
    if geom.geom_type == "Polygon":
        return Polygon(geom.exterior)
    elif geom.geom_type == "MultiPolygon":
        return MultiPolygon([Polygon(p.exterior) for p in geom.geoms])
    else:
        return geom


# Initialize OSM object for a city, clipped to its administrative boundary
# Update set to false (for the thesis), but for more up-to-date results change to true
def init_osm(city_name, directory="data"):

    # Download OSM data (.pbf file) for the given city
    fp = get_data(city_name, directory=directory, update=False)

    # Initialize OSM object
    osm = OSM(fp)

    # Look up boundary name and admin level in CITY_CONFIG
    cfg = CITY_CONFIG[city_name]
    boundary_name = cfg["boundary_name"]
    admin_level = cfg["admin_level"]

    # Load only boundaries with that name
    boundary = osm.get_boundaries(name=boundary_name)
    boundary = boundary[boundary["name"] == boundary_name]

    # If admin_level is specified, filter further
    if admin_level is not None:
```

```
        boundary = boundary[boundary["admin_level"] == str(admin_level)]

    if boundary.empty:
        raise ValueError(f"No boundary found for {city_name} with admin_level={admin_level}")

    # Clean geometry (keep only outer shells, without any holes)
    boundary["geometry"] = boundary["geometry"].apply(drop_holes)

    # Keep only the largest polygon
    boundary["geometry"] = boundary["geometry"].apply(
        lambda g: max(g.geoms, key=lambda p: p.area) if g.geom_type == "MultiPolygon" else g
    )

    # Extract geometry of the boundary
    bbox_geom = boundary['geometry'].values[0]

    # Re-initialize OSM with bounding box
    osm = OSM(fp, bounding_box=bbox_geom)

    return osm, boundary
```

src/preprocessing.py

```python
"""
preprocessing.py

Preprocessing of raw OSM data. Includes boundary filtering,
geometry cleaning and preparation of layers for further analysis.
"""


import geopandas as gpd
import pandas as pd

# Method for exploding multilines into lines, but with keeping attributes
# Used so that geometries are in LineStrings (easier to work with)
def explode_multilines_with_attrs(gdf):

    # Create an empty list to store the new rows
    exploded_rows = []

    # Iterate over all rows
    for idx, row in gdf.iterrows():
        geom = row.geometry

        # Skip rows with missing geometry
        if geom is None:
            continue

        # If geometry is a MultiLineString, split it into its individual parts
        if geom.geom_type == "MultiLineString":
            for part in geom.geoms:
                new_row = row.copy()
                new_row.geometry = part
                exploded_rows.append(new_row)

        # If geometry is already a LineString, just keep it as it is
        elif geom.geom_type == "LineString":
            exploded_rows.append(row)

    return gpd.GeoDataFrame(exploded_rows, crs=gdf.crs)

### Roads
# Extract and filter roads to be used for block construction
def load_roads(osm):

    # Extract all driving roads
    driving_roads = osm.get_network(network_type="driving")

    # Explode MultiLineStrings into LineStrings
    driving_roads = explode_multilines_with_attrs(driving_roads)

    # Define road types that form urban blocks
    selected_types = [
```

```
        "motorway", "motorway_link", "trunk", "trunk_link",
        "primary", "primary_link", "secondary", "secondary_link",
        "tertiary", "tertiary_link", "unclassified", "residential",
        "living_street", "service", "pedestrian", "cycleway", "path"
    ]

    # Use .loc[] to filter
    selected_roads = driving_roads.loc[
        driving_roads["highway"].isin(selected_types)
    ].copy()

    # Extract all ways tagged as bridge
    bridges = osm.get_data_by_custom_criteria(
        custom_filter={
            "bridge": ["yes"],
            "man_made": ["bridge"]
        },
        filter_type="keep",
        keep_nodes=False,
        keep_relations=True
    )

    # --- CLEANUP STEP ---
    if bridges is not None and not bridges.empty:
        # Drop anything that's not a line geometry
        bridges = bridges[bridges.geometry.type.isin(["LineString",
"MultiLineString"])].copy()

        # Drop invalid geometries
        bridges = bridges[bridges.is_valid]

        # Explode MultiLineStrings into LineStrings
        bridges = bridges.explode(index_parts=False).reset_index(drop=True)

        # Ensure same CRS
        bridges = bridges.to_crs(selected_roads.crs)

        # Merge safely
        roads = pd.concat([selected_roads, bridges], ignore_index=True)
    else:
        roads = selected_roads

    return roads

### Railways
# Extract and filter railways to be used for block construction
def load_railways(osm):
   # Extract railway features (only main rail lines)
    railways = osm.get_data_by_custom_criteria(
        custom_filter={"railway": ["rail"]},
        filter_type="keep"
    )

    # Explode MultiLineStrings into LineStrings
```

```
105     railways = explode_multilines_with_attrs(railways)
106 
107     return railways
108 
109 
110 ### Water
111 # Load water-related polygons from OSM
112 def load_water_polygons(osm):
113     water_polygons = osm.get_data_by_custom_criteria(
114         custom_filter={
115             "natural": ["water", "coastline", "bay", "wetland"],
116             "landuse": ["reservoir", "basin"],
117             "waterway": ["riverbank", "canal"],
118         },
119         filter_type="keep",
120     )
121     return water_polygons
122 
123 # Prepare water geometries
124 def prepare_water_geometries(water_polygons):
125 
126     # Choose geometry types - polygons and multipolygons
127     polygons = water_polygons[water_polygons.geometry.type.isin(["Polygon",
    "MultiPolygon"])].copy()
128 
129     water_polygons = polygons
130 
131     return water_polygons
132 
133 # Convert water polygons to edge lines
134 def build_water_edges(water_polygons):
135 
136     # Build edges for block construction
137     def to_edges(geom):
138         if geom is None or geom.is_empty:
139             return None
140         if geom.geom_type in ["Polygon", "MultiPolygon"]:
141             return geom.boundary
142         elif geom.geom_type in ["LineString", "MultiLineString"]:
143             return geom
144         # Skip points, multipoints, etc.
145         else:
146             return None
147 
148     water_edges = water_polygons.copy()
149     water_edges["geometry"] = water_edges["geometry"].apply(to_edges)
150 
151     # Drop invalid/empty
152     water_edges = water_edges[~water_edges.geometry.is_empty]
153 
154     # Explode multilines into single lines
155     water_edges = explode_multilines_with_attrs(water_edges)
156 
157     return water_edges
```

```python


### Handling crs
# Find the best UTM projection for a GeoDataFrame
def get_local_crs(gdf):
    lon, lat = gdf.total_bounds[0::2].mean(), gdf.total_bounds[1::2].mean()
    zone = int((lon + 180) / 6) + 1
    epsg = 32600 + zone if lat >= 0 else 32700 + zone
    return epsg

# Reproject all GeoDataFrames in a dict to the same EPSG
def reproject_all(layers: dict, epsg: int):
    for name, gdf in layers.items():
        if gdf is not None and hasattr(gdf, "crs"):
            layers[name] = gdf.to_crs(epsg=epsg)
    return layers

# Change data crs
def change_crs(boundary, roads, railways, water_polygons, pois):
    local_epsg = get_local_crs(boundary)
    print(f"Using local CRS: EPSG:{local_epsg}")

    # Reproject all at once
    layers = {
        "boundary": boundary,
        "roads": roads,
        "railways": railways,
        "water_polygons": water_polygons,
        "pois": pois
    }
    layers = reproject_all(layers, local_epsg)

    # Unpack
    boundary = layers["boundary"]
    roads = layers["roads"]
    railways = layers["railways"]
    water_polygons = layers["water_polygons"]
    pois = layers["pois"]

    return boundary, roads, railways, water_polygons, pois

```

src/process_blocks.py

```python
"""
process_blocks.py

Constructs and filters urban blocks from spatial boundaries.
Includes polygonization of roads, railways and water features,
as well as removal or merging of water, small and irregular blocks.
"""


import geopandas as gpd
from matplotlib import pyplot as plt
import numpy as np
import shapely
from shapely.ops import polygonize, linemerge
import pandas as pd

# Create initial city blocks (polygons) from roads, railways and water edges
def construct_blocks(roads, railways, water_edges):
    # Merge boundaries into one GeoDataFrame
    boundaries = gpd.GeoDataFrame(
        pd.concat([roads, railways, water_edges], ignore_index=True),
        crs=roads.crs
    )

    # Create a single MultiLineString network
    merged_boundaries = linemerge(shapely.union_all(boundaries.geometry))

    # Polygonize into blocks
    blocks = list(polygonize(merged_boundaries))

    # Convert to GeoDataFrame
    blocks = gpd.GeoDataFrame(geometry=blocks, crs=boundaries.crs)

    return blocks

# Remove blocks dominated by water
def filter_water_blocks(blocks, water_polygons):
    # Add blocks IDs, to keep track of them
    blocks = blocks.reset_index(drop=True)
    blocks["block_id"] = blocks.index

    # Spatial join to find blocks that intersect with water. Add it as a flag column
    joined = gpd.sjoin(blocks, water_polygons, how="left", predicate="within")
    joined["is_water"] = ~joined["index_right"].isna()

    # Split blocks into water and non-water
    water_blocks = joined[joined["is_water"]].copy()
    cleaned_blocks = joined[~joined["is_water"]].copy()

    # Drop helper columns
    cleaned_blocks = cleaned_blocks.drop(columns=["index_right",
"is_water"]).reset_index(drop=True)
```

```
52
53      # Remove empty/broken geometries
54      cleaned_blocks = cleaned_blocks[~cleaned_blocks.geometry.is_empty &
    cleaned_blocks.geometry.notnull()].reset_index(drop=True)
55
56      return cleaned_blocks
57
58  # Filter out small blocks
59  def filter_small_blocks(blocks, lower_percentile=25, max_iterations=5):
60
61      # Compute block areas ---
62      blocks = blocks.reset_index(drop=True).copy()
63      blocks["block_id"] = blocks.index
64
65      # Compute current area and threshold
66      blocks["area_m2"] = blocks.geometry.area
67      min_area = blocks["area_m2"].quantile(lower_percentile / 100)
68
69      for iteration in range(1, max_iterations + 1):
70
71          # Compute current areas based on merged geometries
72          blocks["area_m2"] = blocks.geometry.area
73          small_blocks = blocks[blocks["area_m2"] < min_area].copy()
74          valid_blocks = blocks[blocks["area_m2"] >= min_area].copy()
75
76          # Stop if there's no small blocks
77          if small_blocks.empty:
78              print("No small blocks left")
79              break
80
81          # Compute centroids for distance calculations
82          valid_blocks["centroid"] = valid_blocks.geometry.centroid
83          merged_ids = []
84
85          # Merge small blocks into nearest valid blocks
86          for idx, small_row in small_blocks.iterrows():
87              small_centroid = small_row.geometry.centroid
88              if valid_blocks.empty:
89                  continue
90
91              distances = valid_blocks["centroid"].distance(small_centroid)
92              nearest_idx = distances.idxmin()
93
94              merged_geom = small_row.geometry.union(valid_blocks.loc[nearest_idx,
    "geometry"])
95              valid_blocks.at[nearest_idx, "geometry"] = merged_geom
96              merged_ids.append(small_row["block_id"])
97
98          # Combine both sets before removing merged ones
99          merged_blocks = pd.concat([valid_blocks, small_blocks], ignore_index=True)
100
101         # Drop centroid helper column
102         merged_blocks = merged_blocks.drop(columns=["centroid"], errors="ignore")
103
```

```
        # Remove the small blocks that were merged into others
        blocks = merged_blocks[~merged_blocks["block_id"].isin(merged_ids)].reset_index(drop=True)

        # Clean geometries and reassign IDs for next iteration
        blocks = blocks[~blocks.geometry.is_empty & blocks.geometry.notnull()].reset_index(drop=True)
        blocks["block_id"] = blocks.index
        blocks["area_m2"] = blocks.geometry.area

        print(f" Merged {len(merged_ids)} small blocks.")
        print(f" Remaining blocks = {len(blocks)}")

    # Final cleanup
    blocks = blocks[~blocks.geometry.is_empty & blocks.geometry.notnull()].reset_index(drop=True)

    return blocks


# Filtering of irregular blocks
def filter_irregular_blocks(blocks, compactness_threshold=0.05, max_iterations=5):
    # Perimeter is length
    blocks["perimeter"] = blocks.geometry.length

    # Compactness: Polsby-Popper (4π * Area / Perimeter²)
    # Closer to 1 means more compact
    blocks["compactness"] = 4 * np.pi * blocks["area_m2"] / blocks["perimeter"]**2

    # Determine threshold
    threshold_value = blocks["compactness"].quantile(compactness_threshold)

    # Iterative merging
    for iteration in range (max_iterations):
        blocks["perimeter"] = blocks.geometry.length
        blocks["compactness"] = 4 * np.pi * blocks["area_m2"] / blocks["perimeter"] ** 2

        # Find irregular and compact blocks
        irregular = blocks[blocks["compactness"] <= threshold_value]
        compact = blocks[blocks["compactness"] > threshold_value]

        # Merge irregular blocks
        compact["centroid"] = compact.geometry.centroid
        merge_count = 0
        for idx, row in irregular.iterrows():
            centroid = row.geometry.centroid
            distances = compact["centroid"].distance(centroid)
            nearest_idx = distances.idxmin()
            merged_geom = row.geometry.union(compact.loc[nearest_idx, "geometry"])
            compact.at[nearest_idx, "geometry"] = merged_geom
            merge_count += 1

        compact = compact.drop(columns=["centroid"]).reset_index(drop=True)
        compact["area_m2"] = compact.geometry.area
```

```
        blocks = compact

    # Drop helper columns
    blocks = blocks.drop(columns=["perimeter", "compactness"], errors="ignore")

    return blocks

# Combine filtering steps
def filter_blocks(initial_blocks, water):
    # Filtering out water blocks
    blocks_no_water = filter_water_blocks(initial_blocks, water)

    # Filtering out small blocks
    blocks_no_small = filter_small_blocks(blocks_no_water)

    # Filtering out irregular blocks
    blocks_no_irregular = filter_irregular_blocks(blocks_no_small)

    # Blocks after filtering
    blocks = blocks_no_irregular

    return blocks


# Detect and remove large irregular blocks (e.g. false 'water' areas caused by boundary errors).
def remove_false_water_blocks(blocks, area_quantile=0.99, compactness_quantile=0.1, plot=True):
    blocks = blocks.copy()

    # Compute compactness (Polsby-Popper)
    blocks["compactness"] = 4 * np.pi * blocks.area / (blocks.length ** 2)

    # Define thresholds
    area_thr = blocks.area.quantile(area_quantile)
    comp_thr = blocks["compactness"].quantile(compactness_quantile)

    # Identify suspicious blocks (large + irregular)
    suspicios = blocks[(blocks.area > area_thr) & (blocks["compactness"] < comp_thr)]

    # Remove suspicious blocks
    cleaned = blocks.drop(suspicios.index)

    print(f"Removed {len(suspicios)} suspected false-water blocks "
          f"(area>{area_quantile*100:.0f}%, compactness<{compactness_quantile*100:.0f}%).")

    return cleaned, suspicios

```

src/prepare_pois.py

```python
"""
prepare_pois.py

Cleans, filters and categorizes Points of Interest (POIs).
Converts geometries to point representations and assigns
categories used in the analysis.
"""


import pandas as pd
import geopandas as gpd

# Defining mapping from "amenity" column to categories
amenity_to_category = {
    # Food
    "bar": "food",
    "restaurant": "food",
    "cafe": "food",
    "ice_cream": "food",
    "fast_food": "food",
    "pub": "food",
    "hookah_lounge": "food",
    "food_court": "food",
    "internet_cafe": "food",
    "food_sharing": "food",
    "pastry": "food",
    "community_centre;cafe": "food",
    "canteen": "food",
    "biergarten": "food",

    # Infrastructure & transport
    "parking": "infrastructure_transport",
    "parking_space": "infrastructure_transport",
    "bicycle_parking": "infrastructure_transport",
    "motorcycle_parking": "infrastructure_transport",
    "charging_station": "infrastructure_transport",
    "taxi": "infrastructure_transport",
    "ferry_terminal": "infrastructure_transport",
    "car_rental": "infrastructure_transport",
    "car_wash": "infrastructure_transport",
    "bicycle_rental": "infrastructure_transport",
    "bus_station": "infrastructure_transport",
    "car_sharing": "infrastructure_transport",
    "scooter_parking": "infrastructure_transport",
    "traffic_park": "infrastructure_transport",
    "motorcycle_rental": "infrastructure_transport",
    "kick-scooter_parking": "infrastructure_transport",

    # Education
    "school": "education",
    "kindergarten": "education",
```

```
    "childcare": "education",
    "university": "education",
    "college": "education",
    "language_school": "education",
    "research_institute": "education",
    "music_school": "education",
    "prep_school": "education",

    # Culture & leisure
    "social_facility": "culture_leisure",
    "events_venue": "culture_leisure",
    "theatre": "culture_leisure",
    "library": "culture_leisure",
    "cinema": "culture_leisure",
    "gambling": "culture_leisure",
    "music_venue": "culture_leisure",
    "arts_centre": "culture_leisure",
    "casino": "culture_leisure",
    "nightclub": "culture_leisure",
    "stripclub": "culture_leisure",
    "brothel": "culture_leisure",
    "gallery": "culture_leisure",
    "swingerclub": "culture_leisure",
    "monastery": "culture_leisure",
    "dojo": "culture_leisure",
    "dive_centre": "culture_leisure",
    "exhibition_centre": "culture_leisure",
    "planetarium": "culture_leisure",
    "public_bath": "culture_leisure",
    "festival_grounds": "culture_leisure",
    "climbing_wall": "culture_leisure",
    "dancing_school": "culture_leisure",
    "surf_school": "culture_leisure",
    "convent": "culture_leisure",

    # Public services
    "place_of_worship": "public_services",
    "community_centre": "public_services",
    "bank": "public_services",
    "post_office": "public_services",
    "police": "public_services",
    "courthouse": "public_services",
    "fire_station": "public_services",
    "social_centre": "public_services",
    "conference_centre": "public_services",
    "funeral_hall": "public_services",
    "crematorium": "public_services",
    "townhall": "public_services",
    "parliament": "public_services",
    "lost_property_office": "public_services",
    "animal_shelter": "public_services",
    'local government unit': "public_services",
    "ranger_station": "public_services",

```

```
    # Healthcare
    "pharmacy": "healthcare",
    "clinic": "healthcare",
    "dentist": "healthcare",
    "doctors": "healthcare",
    "veterinary": "healthcare",
    "hospital": "healthcare",
    "nursing_home": "healthcare",
    "fysioterapi": "healthcare",
    "healthcare": "healthcare",

    # Retail
    "marketplace": "retail",

    # Green spaces
    "playground": "green_spaces",

    # Other daily utilities
    "recycling": "other_daily_utilities",
    "toilets": "other_daily_utilities",
    "drinking_water": "other_daily_utilities",
    "atm": "other_daily_utilities",
    "fuel": "other_daily_utilities",
    "parcel_locker": "other_daily_utilities",
    "bicycle_repair_station": "other_daily_utilities",
    "coworking_space": "other_daily_utilities",
    "bureau_de_change": "other_daily_utilities",
    "luggage_locker": "other_daily_utilities",
    "locker": "other_daily_utilities",
    "left_luggage": "other_daily_utilities",
    "self_storage": "other_daily_utilities"
}

# Define mappings for other useful columns - for rows that didn't have "amenity" value
other_columns_to_category = {
    # Public services
    "office": "public_services",
    "post_office": "public_services",
    "charity": "public_services",
    "police": "public_services",

    # Culture & leisure
    "attraction": "culture_leisure",
    "camp_site": "culture_leisure",
    "information": "culture_leisure",
    "museum": "culture_leisure",
    "tourism": "culture_leisure",
    "caravan_site": "culture_leisure",
    "zoo": "culture_leisure",
    "swimming_pool ": "culture_leisure",

    # Food
    "bar": "food",
    "tea": "food",
```

```
    "pastry": "food",
    "restaurant ": "food",

    # Retail
    "books": "retail",
    "butcher": "retail",
    "clothes": "retail",
    "confectionery": "retail",
    "craft": "retail",
    "furniture": "retail",
    "gift": "retail",
    "massage": "retail",
    "model": "retail",
    "music": "retail",
    "outdoor": "retail",
    "pet": "retail",
    "second_hand": "retail",
    "wholesale": "retail",
    "shop": "retail",
    "shoes": "retail",
    "medical_supply ": "retail",

    # Infrastructure & transport
    "bicycle_rental": "infrastructure_transport",

    # Green spaces
    "green_spaces": "green_spaces"
}

# Extract raw POIs from OSM
def load_pois(osm):
    pois = osm.get_pois()
    return pois

# Process POIs: clean geometries and basic filtering
def process_pois(pois):

    # Drop empty or null geometries
    pois = pois[~pois.geometry.is_empty & pois.geometry.notna()].copy()

    # Drop invalid geometry types (multipolygons and multilinestrings)
    invalid_types = ['MultiPolygon', 'MultiLineString']
    pois = pois[~pois.geometry.type.isin(invalid_types)].copy()

    # Convert Polygon and LineString to centroid points
    pois.loc[pois.geometry.type.isin(['Polygon', 'LineString']), 'geometry'] = \
        pois.loc[pois.geometry.type.isin(['Polygon', 'LineString']), 'geometry'].centroid

    return pois


# Assign categories based on OSM tags
def assign_categories(pois):

```

```
    # Normalize - treat NaN, None and no as missing
    def normalize(val):
        if val is None:
            return None
        v = str(val).strip().lower()
        return None if v in {"nan", "none", "no", ""} else v

    # Check columns and assign category when a match is found
    def assign_category(row, ordered_mapping):
        for col, cat in ordered_mapping.items():
            if col in row and normalize(row[col]) is not None:
                return cat
        return None

    # Map directly from amenity_to_category
    pois["category"] = pois["amenity"].map(amenity_to_category)

    # Handle green spaces
    if "green_spaces" in pois.columns:
        pois.loc[pois["green_spaces"] == "yes", "category"] = "green_spaces"

    # For remaining, check other columns
    missing_mask = pois["category"].isna()
    pois_no_cat = pois.loc[missing_mask].copy()

    if not pois_no_cat.empty:
        pois_no_cat["category"] = pois_no_cat.apply(lambda r: assign_category(r, other_columns_to_category), axis=1)
        pois.loc[pois_no_cat.index, "category"] = pois_no_cat["category"]

    # Check
    print(f"Assigned categories for {pois['category'].notna().sum()} POIs "
          f"({pois['category'].notna().mean():.1%} coverage)")

    return pois

# Method doing all
def prepare_pois(pois):
    pois = process_pois(pois)
    print("Number of POIs after cleaning: " + str(len(pois)))

    pois = assign_categories(pois)
    print("Number of POIs after categorization: " + str(len(pois)))

    return pois


# Green spaces POIS from the "leisure" tag
def load_pois_with_green(osm):

    # Load all regular POIs
    pois = load_pois(osm)
    print(f"Loaded {len(pois)} regular POIs")

```

```
    # Load green areas (parks, gardens, etc.)
    green_spaces = osm.get_data_by_custom_criteria(
        custom_filter={
            "leisure": ["park"],
            "landuse": ["recreation_ground"]
        },
        filter_type="keep"
    )

    if not green_spaces.empty:
        print(f"Loaded {len(green_spaces)} green-space features")

        # Filter out invalid geometries
        green_spaces = green_spaces[~green_spaces.geometry.is_empty &
green_spaces.geometry.notna()].copy()

        # Add category
        green_spaces["green_spaces"] = "yes"

        # Merge
        pois = pd.concat([pois, green_spaces], ignore_index=True)

    print(f"Total combined POIs: {len(pois)}")
    return pois

# Assigning POIs to blocks
def assign_pois_to_blocks (pois, blocks):

    blocks = blocks.copy()
    pois = pois.copy()

    # Rename 'id' to 'poi_id'
    pois = pois.rename(columns={'id': 'poi_id'})

    # Make sure that there ar no rows with all none values
    pois = pois.dropna(how='all')

    # Keep only categorized POIs
    pois = pois[pois["category"].notna()].copy()

    # Spatial join (assign POIs to blocks)
    pois_with_blocks = gpd.sjoin(pois, blocks, how="left", predicate="within")

    # Count POIs by category per block
    category_counts = (
        pois_with_blocks.groupby(["block_id", "category"])
        .size()
        .reset_index(name="count")
    )

    # Group POI IDs per block
    pois_grouped = (
        pois_with_blocks.groupby("block_id")
        .agg({"poi_id": list})
```

```
320         .reset_index()
321     )
322 
323     # Merge POIs info into final_blocks
324     blocks_with_pois  = blocks.merge(pois_grouped, on="block_id", how="left")
325 
326     # Create a new column that counts how many POIs each block has
327     blocks_with_pois["poi_count"] = blocks_with_pois["poi_id"].apply(
328         lambda x: len(x) if isinstance(x, list) else 0
329     )
330 
331     # Store category_counts
332     blocks_with_pois.attrs["category_counts"] = category_counts
333 
334     print(f"POIs assigned to {blocks_with_pois['poi_count'].gt(0).sum()} blocks "
335           f"(out of {len(blocks_with_pois)})")
336 
337     return blocks_with_pois
```

src/construct_graph.py

```python
"""
construct_graph.py

Builds the block adjacency graph.
Each node represents an urban block and edges represent
spatial adjacency between blocks.
"""


import networkx as nx

# Build an undirected graph where each block is a node and edges connect touching blocks
def build_block_graph(blocks):
    blocks = blocks.copy()

    # Initialize graph
    G = nx.Graph()

    # Add each polygon as a node, using its block id as identifier
    for idx, row in blocks.iterrows():
        G.add_node(row["block_id"], geometry=row.geometry)

    # Use spatial index for efficient neighbor finding
    sindex = blocks.sindex

    # For each block, find and connect neighboring blocks
    for idx, row in blocks.iterrows():
        geom = row.geometry
        block_id = row["block_id"]

        # Find possible neighbors based on bounding box
        possible_neighbors_index = list(sindex.intersection(geom.bounds))

        for neighbor_idx in possible_neighbors_index:
            neighbor_row = blocks.iloc[neighbor_idx]
            neighbor_geom = neighbor_row.geometry
            neighbor_id = neighbor_row["block_id"]

            # Avoid self-comparison
            if block_id == neighbor_id:
                continue

            # If geometries touch (share a boundary), add an edge
            if geom.touches(neighbor_geom):
                G.add_edge(block_id, neighbor_id)

    # Add POIs counts as node attributes
    poi_counts = blocks.set_index("block_id")["poi_count"].fillna(0).to_dict()
    nx.set_node_attributes(G, poi_counts, "poi_count")


```

```
    print(f"Graph constructed: {G.number_of_nodes()} nodes, {G.number_of_edges()} edges")
    return G
```

src/network_distance.py

```
"""
network_distance.py

(Code from my supervisor)
Implements network-based distance measures.
"""


import copy, os, subprocess, itertools
import numpy as np
import pandas as pd
import networkx as nx
from pyemd import emd as _emd
from scipy import spatial
from scipy.sparse import csgraph
from collections import defaultdict
from multiprocessing import Pool, Manager

def _ge_Q(network):
   A = nx.adjacency_matrix(network).todense().astype(float)
   return np.linalg.pinv(csgraph.laplacian(np.matrix(A), normed = False))

def _ge_ml_collapse_vectors(vectors, network, couplings = set([]), coupling_style =
"clique", layer_jump_weight = defaultdict(lambda : defaultdict(lambda : 1))):
   for n in network.nodes:
      network.nodes[n]["layer"] = list(dict(network[n]).values())[0]["layer"]
   for i in range(len(vectors)):
      vectors[i] = defaultdict(lambda: 0, vectors[i])
   if coupling_style == "clique":
      for coupling in couplings:
         for pair in itertools.combinations(coupling, 2):
            if layer_jump_weight[network.nodes[pair[0]]["layer"]][network.nodes[pair[1]]
["layer"]] == np.inf:
               for i in range(len(vectors)):
                  vectors[i][pair[0]] += vectors[i][pair[1]]
   elif coupling_style == "chain":
      for coupling in couplings:
         for i in range(len(coupling) - 1):
            if layer_jump_weight[network.nodes[coupling[i]]["layer"]]
[network.nodes[coupling[i + 1]]["layer"]] == np.inf:
               for j in range(len(vectors)):
                  vectors[j][coupling[i]] += vectors[j][coupling[i + 1]]
   return [{k: v for k, v in d.items() if v} for d in vectors]

def _ge_ml_preprocess(network, couplings = set([]), coupling_style = "clique",
layer_jump_weight = defaultdict(lambda : defaultdict(lambda : 1))):
   # Phase 1: Infer in which layer a node is
   for n in network.nodes:
      network.nodes[n]["layer"] = list(dict(network[n]).values())[0]["layer"]
   # Phase 2: Add the coupling edges as a special "coupling" layer
   # At the same time, if some layers have instantaneaous spread, contract their coupled
nodes
```

```
48    network = nx.MultiGraph(network)
49    if coupling_style == "clique":
50       for coupling in couplings:
51          for pair in itertools.combinations(coupling, 2):
52             if pair[0] in network.nodes and pair[1] in network.nodes: # if I can't find
the node, it was already collapsed in the correct one
53                if layer_jump_weight[network.nodes[pair[0]]["layer"]]
[network.nodes[pair[1]]["layer"]] == np.inf:
54                   network = nx.contracted_nodes(network, pair[0], pair[1])
55                else:
56                   network.add_edge(pair[0], pair[1], layer = "coupling")
57    elif coupling_style == "chain":
58       for coupling in couplings:
59          for i in range(len(coupling) - 1):
60             if layer_jump_weight[network.nodes[coupling[i]]["layer"]]
[network.nodes[coupling[i + 1]]["layer"]] == np.inf:
61                nx.contracted_nodes(network, coupling[i], coupling[i + 1], copy = False)
62             else:
63                network.add_edge(coupling[i], coupling[i + 1], layer = "coupling")
64    if nx.number_connected_components(network) > 1:
65       raise ValueError("Node vector distance is only valid if calculated on a network with
a single connected component. The network passed has more than one.")
66    # Phase 2b: As a result of the previous phase, network might be a multigraph. So we
need to collapse the parallel edges according to the chosen collapse style
67    _network = nx.Graph()
68    for e in network.edges(data = True):
69       if _network.has_edge(e[0], e[1]):
70          _network[e[0]][e[1]]["layer"].add(e[2]["layer"])
71       else:
72          _network.add_edge(e[0], e[1], layer = set([e[2]["layer"],]))
73    nx.set_node_attributes(_network, {n: {"layer": network.nodes[n]["layer"]} for n in
_network.nodes})
74    network = _network
75    network.graph["coupled"] = True
76    return network
77
78 def _ge_ml_Q(network, collapse_style = sum, layer_weight = defaultdict(lambda : 1),
layer_jump_weight = defaultdict(lambda : defaultdict(lambda : 1))):
79    # Phase 3: Build B, the oriented incidence matrix of the input network
80    B = nx.incidence_matrix(network, oriented = True)
81    # Phase 4: Build W, the edge weight diagonal matrix
82    edge_weights = []
83    for e in network.edges(data = True):
84       if "coupling" in e[2]["layer"]:
85          edge_weights.append(layer_jump_weight[network.nodes[e[0]]["layer"]]
[network.nodes[e[1]]["layer"]])
86       else:
87          edge_weights.append(collapse_style(layer_weight[l] for l in e[2]["layer"]))
88    W = np.diag(edge_weights)
89    # Phase 5: Create the weighted Laplacian
90    L = B * W * B.T
91    # Phase 6: Generate Q and pass it to normal GE
92    return np.linalg.pinv(L)
93
```

```
# GE interprets edge weights as capacities: doubling all edge weights lowers GE by a factor of sqrt(2) in a chain graph
def ge(src, trg, network, Q = None, normed = True, multilayer = False, couplings = set([]), coupling_style = "clique", collapse_style = sum, layer_weight = defaultdict(lambda : 1), layer_jump_weight = defaultdict(lambda : defaultdict(lambda : 1))):
   if multilayer:
      if not "coupled" in network.graph:
         network = _ge_ml_preprocess(network, couplings, coupling_style, layer_jump_weight)
      if Q is None:
         Q = _ge_ml_Q(network, collapse_style, layer_weight, layer_jump_weight)
   if nx.number_connected_components(network) > 1:
      raise ValueError("Node vector distance is only valid if calculated on a network with a single connected component. The network passed has more than one.")
   src = np.array([src[n] if n in src else 0. for n in network.nodes()])
   trg = np.array([trg[n] if n in trg else 0. for n in network.nodes()])
   if normed:
      src = src / src.sum() if src.sum() > 0 else src
      trg = trg / trg.sum() if trg.sum() > 0 else trg
   diff = src - trg
   if Q is None:
      Q = _ge_Q(network)
   return np.sqrt(diff.T.dot(np.array(Q).dot(diff)))

def _gft_ml_v(network, collapse_style = max, layer_weight = defaultdict(lambda : 1), layer_jump_weight = defaultdict(lambda : defaultdict(lambda : 1))):
   # Phase 3: Build B, the oriented incidence matrix of the input network
   B = nx.incidence_matrix(network, oriented = True)
   # Phase 4: Build W, the edge weight diagonal matrix
   edge_weights = []
   for e in network.edges(data = True):
      if "coupling" in e[2]["layer"]:
         edge_weights.append(1 / layer_jump_weight[network.nodes[e[0]]["layer"]][network.nodes[e[1]]["layer"]])
      else:
         edge_weights.append(1 / collapse_style(layer_weight[l] for l in e[2]["layer"]))
   W = np.diag(edge_weights)
   # Phase 5: Create the weighted Laplacian
   L = B * W * B.T
   l, v = np.linalg.eig(L)
   idx = l.argsort()
   l = l[idx]
   v = np.diag(l).dot(v[:,idx].T)
   return v

def _gft_v(network):
   A = nx.adjacency_matrix(network).todense().astype(float)
   laplacian = csgraph.laplacian(np.matrix(A), normed = False)
   l, v = np.linalg.eig(laplacian)
   idx = l.argsort()
   l = l[idx]
   v = np.diag(l).dot(v[:,idx].T)
   return v

```

```
141 # GFT interprets edge weights as costs: doubling all edge weights increases GFT by a
    factor of 2 in a chain graph
142 def gft(src, trg, network, v = None, normed = True, linkage = "euclidean", multilayer =
    False, couplings = set([]), coupling_style = "clique", collapse_style = max, layer_weight
    = defaultdict(lambda : 1), layer_jump_weight = defaultdict(lambda : defaultdict(lambda :
    1))):
143     if multilayer:
144         if not "coupled" in network.graph:
145             network = _ge_ml_preprocess(network, couplings = couplings, coupling_style =
    coupling_style, layer_jump_weight = layer_jump_weight)
146         if v is None:
147             v = _gft_ml_v(network, collapse_style = collapse_style, layer_weight =
    layer_weight, layer_jump_weight = layer_jump_weight)
148     if nx.number_connected_components(network) > 1:
149         raise ValueError("Node vector distance is only valid if calculated on a network with
    a single connected component. The network passed has more than one.")
150     src = np.array([src[n] if n in src else 0. for n in network.nodes()])
151     trg = np.array([trg[n] if n in trg else 0. for n in network.nodes()])
152     if normed:
153         src = src / src.sum() if src.sum() > 0 else src
154         trg = trg / trg.sum() if trg.sum() > 0 else trg
155     if v is None:
156         v = _gft_v(network)
157     src = src.dot(v)
158     trg = trg.dot(v)
159     if linkage == "euclidean":
160         return spatial.distance.euclidean(src, trg)
161     elif linkage == "cosine":
162         return spatial.distance.cosine(src, trg)
163     elif linkage == "pearson":
164         return spatial.distance.correlation(src, trg)
165     else:
166         raise ValueError("No valid linkage strategy. Possible values: euclidean (default),
    cosine, pearson.")
167
168 def _spl_ml_preprocess(network, couplings = set([]), coupling_style = "clique",
    collapse_style = max, layer_weight = defaultdict(lambda : 1), layer_jump_weight =
    defaultdict(lambda : defaultdict(lambda : 1))):
169     network = _ge_ml_preprocess(network, couplings = couplings, coupling_style =
    coupling_style, layer_jump_weight = layer_jump_weight)
170     for e in network.edges(data = True):
171         if "coupling" in e[2]["layer"]:
172             network[e[0]][e[1]]["weight"] = 1 / layer_jump_weight[network.nodes[e[0]]
    ["layer"]][network.nodes[e[1]]["layer"]]
173         else:
174             network[e[0]][e[1]]["weight"] = 1 / collapse_style(layer_weight[l] for l in e[2]
    ["layer"])
175     return network
176
177 def _spl(x):
178     x[2][x[1]] = dict(nx.shortest_path_length(x[0], source = x[1], weight = "weight"))
179
180 def calculate_spl(network, nonzero_nodes, n_proc, return_as_dict = False):
181     manager = Manager()
182     shortest_path_lengths = manager.dict()
```

```
    pool = Pool(processes = n_proc)
    _ = pool.map(_spl, [(network, n, shortest_path_lengths) for n in nonzero_nodes])
    pool.close()
    pool.join()
    shortest_path_lengths = dict(shortest_path_lengths)
    if return_as_dict:
        return {j: {i: shortest_path_lengths[j][i] for i in nonzero_nodes} for j in nonzero_nodes}
    else:
        return np.array([[shortest_path_lengths[i][j] if j in shortest_path_lengths[i] else np.inf for i in nonzero_nodes] for j in nonzero_nodes], dtype = "float64")

# The current pathfinding implementation ignores edge weights, thus they do not affect EMD nor SPL
def emd(src, trg, network, shortest_path_lengths = None, n_proc = 1, normed = True, multilayer = False, couplings = set([]), coupling_style = "clique", collapse_style = max, layer_weight = defaultdict(lambda : 1), layer_jump_weight = defaultdict(lambda : defaultdict(lambda : 1))):
    if multilayer and not "coupled" in network.graph:
        network = _spl_ml_preprocess(network, couplings = couplings, coupling_style = coupling_style, collapse_style = collapse_style, layer_weight = layer_weight, layer_jump_weight = layer_jump_weight)
    if nx.number_connected_components(network) > 1:
        raise ValueError("Node vector distance is only valid if calculated on a network with a single connected component. The network passed has more than one.")
    nonzero_nodes = list((set(src) | set(trg)) & set(network.nodes))
    src = np.array([src[n] if n in src else 0. for n in nonzero_nodes], dtype = float)
    trg = np.array([trg[n] if n in trg else 0. for n in nonzero_nodes], dtype = float)
    if normed:
        src = src / src.sum() if src.sum() > 0 else src
        trg = trg / trg.sum() if trg.sum() > 0 else trg
    if shortest_path_lengths is None:
        shortest_path_lengths = calculate_spl(network, nonzero_nodes, n_proc)
    else:
        shortest_path_lengths = np.array([[shortest_path_lengths[i][j] for i in nonzero_nodes] for j in nonzero_nodes], dtype = "float64")
    return _emd(src, trg, shortest_path_lengths)

def _standardize_node_vectors(src, trg, normed):
    if normed:
        src_std = {x: src[x] / sum(src.values()) for x in src}
        trg_std = {x: trg[x] / sum(trg.values()) for x in trg}
    else:
        src_sum = sum(src.values())
        trg_sum = sum(trg.values())
        if src_sum < trg_sum:
            src_std = {x: src[x] * (trg_sum / src_sum) for x in src}
            trg_std = copy.deepcopy(trg)
        else:
            src_std = copy.deepcopy(src)
            trg_std = {x: trg[x] * (src_sum / trg_sum) for x in trg}
    return src_std, trg_std

def _spl_mean(src, trg, shortest_path_lengths, normed):
```

```
227     total_path_length = sum(src[origin] * trg[dest] * shortest_path_lengths[dest][origin]
    for origin in src for dest in trg)
228     if normed:
229         return total_path_length
230     else:
231         return total_path_length / sum(src.values())
232 
233 def _make_pathlengths(spl, src, trg, delete_s = None, delete_t = None):
234     pathlengths = defaultdict(set)
235     if delete_s is not None:
236         for length in spl:
237             for path in spl[length]:
238                 if path[0] != delete_s:
239                     pathlengths[length].add(path)
240     elif delete_t is not None:
241         for length in spl:
242             for path in spl[length]:
243                 if path[1] != delete_t:
244                     pathlengths[length].add(path)
245     else:
246         for s in src:
247             for t in trg:
248                 pathlengths[spl[s][t]].add((s,t))
249     return pathlengths
250 
251 def _spl_greedy(src, trg, ascending, shortest_path_lengths):
252     pathlengths = _make_pathlengths(shortest_path_lengths, src, trg)
253     total_path_length = 0.0
254     while len(pathlengths) > 0:
255         if ascending:
256             pathlength = min(pathlengths)
257         else:
258             pathlength = max(pathlengths)
259         weights = {pair: min(src[pair[0]], trg[pair[1]]) for pair in
    pathlengths[pathlength]}
260         row = max(weights, key = weights.get)
261         total_path_length += (weights[row] * pathlength)
262         src[row[0]] -= weights[row]
263         trg[row[1]] -= weights[row]
264         if src[row[0]] == 0:
265             del src[row[0]]
266             pathlengths = _make_pathlengths(pathlengths, None, None, delete_s = row[0])
267         if trg[row[1]] == 0:
268             del trg[row[1]]
269             pathlengths = _make_pathlengths(pathlengths, None, None, delete_t = row[1])
270     return total_path_length
271 
272 def spl(src, trg, network, linkage = "avg", shortest_path_lengths = None, n_proc = 1,
    normed = True):
273     if nx.number_connected_components(network) > 1:
274         raise ValueError("Node vector distance is only valid if calculated on a network with
    a single connected component. The network passed has more than one.")
275     nonzero_nodes = list((set(src) | set(trg)) & set(network.nodes))
276     src_std, trg_std = _standardize_node_vectors(src, trg, normed)
```

```
277     if shortest_path_lengths is None:
278         shortest_path_lengths = calculate_spl(network, nonzero_nodes, n_proc, return_as_dict
    = True)
279     if linkage == "single":
280         return _spl_greedy(src_std, trg_std, True, shortest_path_lengths)
281     elif linkage == "complete":
282         return _spl_greedy(src_std, trg_std, False, shortest_path_lengths)
283     elif linkage == "avg":
284         return _spl_mean(src_std, trg_std, shortest_path_lengths, normed)
285     else:
286         raise ValueError("No valid linkage strategy. Possible values: avg (default), single,
    complete.")
287 
288 def _make_mapp_runs(src, trg, shortest_path_lengths):
289     runs = []
290     while sum(src.values()) > 1e-12:
291         spl_df = pd.DataFrame(shortest_path_lengths).unstack().reset_index()
292         spl_df.columns = ("trg", "src", "spl")
293         spl_df = spl_df[spl_df["src"].isin(src) & spl_df["trg"].isin(trg)]
294         spl_df["allocable_weight"] = spl_df.apply(lambda x : min(src[x["src"]],
    trg[x["trg"]]), axis = 1)
295         spl_df = spl_df[spl_df["allocable_weight"] > 0].sort_values(by = ["spl",
    "allocable_weight"], ascending = [True, False])
296         robot_starts = {} # vertex, robotid
297         robot_ends = {} # vertex, robotid
298         robot_weights = {} # robotid, weight
299         cur_robot_id = 1
300         while spl_df.shape[0] > 0:
301             robot_starts[int(spl_df.iloc[0]["src"])] = cur_robot_id
302             robot_ends[int(spl_df.iloc[0]["trg"])] = cur_robot_id
303             robot_weights[cur_robot_id] = spl_df.iloc[0]["allocable_weight"]
304             cur_robot_id += 1
305             src[spl_df.iloc[0]["src"]] -= spl_df.iloc[0]["allocable_weight"]
306             trg[spl_df.iloc[0]["trg"]] -= spl_df.iloc[0]["allocable_weight"]
307             spl_df = spl_df[(spl_df["src"] != spl_df.iloc[0]["src"]) & (spl_df["trg"] !=
    spl_df.iloc[0]["trg"])]
308         runs.append({"starts": robot_starts, "ends": robot_ends, "weights": robot_weights})
309     return runs
310 
311 def _mapp_cost(G, runs):
312     total_path_length = 0
313     for cur_run in range(len(runs)):
314         with open("temp.cpf", 'w') as f:
315             f.write("V =\n")
316             for n in G.nodes:
317                 start_code = runs[cur_run]["starts"][n] if n in runs[cur_run]["starts"] else 0
318                 end_code = runs[cur_run]["ends"][n] if n in runs[cur_run]["ends"] else 0
319                 f.write("(%d:-1)[%s:%s:%s]\n" % (n, start_code, end_code, end_code))
320             f.write("E =\n")
321             for e in G.edges:
322                 f.write("{%d,%d} (-1)\n" % (e[0], e[1]))
323         bash_command = "./insolver_reLOC --input-file=temp.cpf --output-file=temp.out"
324         process = subprocess.Popen(bash_command.split(), stdout = subprocess.PIPE)
325         output, error = process.communicate()
```

```
326         if error is None and b"Total cost = 0" in output:
327             continue
328         with open("temp.out", 'r') as f:
329             for line in f:
330                 try:
331                     fields = line.strip().split()
332                     robot = int(fields[0])
333                     total_path_length += runs[cur_run]["weights"][robot]
334                 except:
335                     continue
336         os.remove("temp.cpf")
337         os.remove("temp.out")
338     return total_path_length
339 
340 #TODO: Slow, but probably I can't do much about it...
341 def mapp(src, trg, network, shortest_path_lengths = None, n_proc = 1, normed = True):
342     if nx.number_connected_components(network) > 1:
343         raise ValueError("Node vector distance is only valid if calculated on a network with
    a single connected component. The network passed has more than one.")
344     nodemap = {list(network.nodes)[i]: i for i in range(len(network.nodes))}
345     G = nx.relabel_nodes(network, nodemap)
346     src = {nodemap[x]: src[x] for x in src}
347     trg = {nodemap[x]: trg[x] for x in trg}
348     nonzero_nodes = list(set(src) | set(trg))
349     src, trg = _standardize_node_vectors(src, trg, normed)
350     if shortest_path_lengths is None:
351         shortest_path_lengths = calculate_spl(G, nonzero_nodes, n_proc, return_as_dict =
    True)
352     runs = _make_mapp_runs(src, trg, shortest_path_lengths)
353     return _mapp_cost(G, runs)
354 
355 def _mmc_P(network, time_steps):
356     A = nx.to_numpy_array(network)
357     np.fill_diagonal(A, 1)
358     P = A / A.sum(axis = 0)
359     return np.linalg.matrix_power(P, time_steps)
360 
361 def _mmc_matrix(v, network, P, time_steps):
362     if P is None:
363         P = _mmc_P(network, time_steps)
364     sigma_sum = v.dot(P.dot(1.0 - P)) ** 2
365     return sigma_sum
366 
367 # MMC interprets edge weights as costs: increasing all edge weights increases MMC with
    diminishing returns in a chain graph
368 def mmc(src, trg, network, time_steps = None, P = None, normed = True):
369     if nx.number_connected_components(network) > 1:
370         raise ValueError("Node vector distance is only valid if calculated on a network with
    a single connected component. The network passed has more than one.")
371     src = np.array([src[n] if n in src else 0. for n in network.nodes()])
372     trg = np.array([trg[n] if n in trg else 0. for n in network.nodes()])
373     if normed:
374         src = src / src.sum()
375         trg = trg / trg.sum()
```

```
    diff = src - trg
    if time_steps == None:
        time_steps = nx.diameter(network)
    sigma_src = _mmc_matrix(src, network, P, time_steps)
    sigma_trg = _mmc_matrix(trg, network, P, time_steps)
    sigma = (sigma_src + sigma_trg) / 2
    Q = np.diag(1.0 / sigma)
    return np.sqrt(diff.dot(Q.dot(diff)))

def _annihilation_Q(A):
    P = A / A.sum(axis = 0)
    l, v = np.linalg.eig(P)
    idx = l.argsort()
    l = l[idx]
    v = v[:,idx]
    stationary = v[:,-1] / np.sum(v[:,-1])
    P_inf = np.tile(stationary, (1, P.shape[0]))
    return np.array((np.identity(P.shape[0]) - (P - P_inf)) ** -1)

# The current implementation of annihilation ignores edge weights, thus they do not affect it
def annihilation(src, trg, network, Q = None, normed = True):
    if nx.number_connected_components(network) > 1:
        raise ValueError("Node vector distance is only valid if calculated on a network with a single connected component. The network passed has more than one.")
    src = np.array([src[n] if n in src else 0. for n in network.nodes()])
    trg = np.array([trg[n] if n in trg else 0. for n in network.nodes()])
    if normed:
        src = src / src.sum()
        trg = trg / trg.sum()
    if Q is None:
        A = nx.adjacency_matrix(network).todense().astype(float)
        Q = _annihilation_Q(A)
    return _annihilation(src, trg, Q)

def _annihilation(src, trg, Q):
    diff = src - trg
    return np.real(np.sqrt(diff.T.dot(Q).dot(diff)))

def correlation(src, trg, network, shortest_path_lengths = None, n_proc = 1):
    src = np.array([src[n] if n in src else 0. for n in network.nodes], dtype = float)
    trg = np.array([trg[n] if n in trg else 0. for n in network.nodes], dtype = float)
    src_ = src - np.mean(src)
    trg_ = trg - np.mean(trg)
    if shortest_path_lengths is None:
        shortest_path_lengths = calculate_spl(network, network.nodes, n_proc)
    W = 1 / np.exp(shortest_path_lengths)
    numerator = (W * np.outer(src_, trg_)).sum()
    denominator_src = np.sqrt((W * np.outer(src_, src_)).sum())
    denominator_trg = np.sqrt((W * np.outer(trg_, trg_)).sum())
    return numerator / (denominator_src * denominator_trg)

def moran(v, network, shortest_path_lengths = None, n_proc = 1, kernel = "neighbors"):
    v = np.array([v[n] if n in v else 0. for n in network.nodes], dtype = float)
```

```
428     return _moran(v, network, shortest_path_lengths = shortest_path_lengths, n_proc = n_proc, kernel = kernel)
429
430 def _moran(v, network, shortest_path_lengths = None, n_proc = 1, kernel = "neighbors"):
431     v_ = v - np.mean(v)
432     if kernel == "neighbors":
433         W = np.array(nx.adjacency_matrix(network).todense())
434     else:
435         if shortest_path_lengths is None:
436             shortest_path_lengths = calculate_spl(network, network.nodes, n_proc)
437         if kernel == "linear":
438             W = 1 / shortest_path_lengths
439         elif kernel == "exponential":
440             W = 1 / np.exp(shortest_path_lengths - 1)
441         np.fill_diagonal(W, 0)
442     W = W / W.sum(axis = 1)
443     return (W * np.outer(v_, v_)).sum() / (v_ ** 2).sum()
444
445 def _resistance(network):
446     L = csgraph.laplacian(np.matrix(nx.adjacency_matrix(network).todense().astype(float)), normed = False)
447     Q = np.linalg.pinv(L)
448     zeta = np.diag(Q)
449     u = np.ones(zeta.shape[0])
450     return np.array((np.matrix(u).T * zeta) + (np.matrix(zeta).T * u) - (2 * Q))
451
452 def variance(v, network, shortest_path_lengths = None, n_proc = 1, kernel = "geodesic"):
453     v = np.array([v[n] if n in v else 0. for n in network.nodes], dtype = float)
454     v = v / v.sum()
455     if kernel == "geodesic":
456         if shortest_path_lengths is None:
457             shortest_path_lengths = calculate_spl(network, network.nodes, n_proc)
458     elif kernel == "resistance":
459         if shortest_path_lengths is None:
460             shortest_path_lengths = _resistance(network)
461     return (np.outer(v, v) * (shortest_path_lengths ** 2)).sum() / 2
462
```

src/network_analysis.py

```python
"""
network_analysis.py

Computes network-level statistics and variance-based measures.
Includes randomized null models, network variance estimation,
and z-score computation for amenity distributions.
"""


import os
import pickle
import random
import numpy as np
import networkx as nx
import pandas as pd

from src.network_distance import ge, _ge_Q, variance, _resistance, calculate_spl

# Get the largest component from the graph
def get_largest_component(G):
    n_components = nx.number_connected_components(G)
    print(f"Graph has {n_components} components.")

    largest_cc = max(nx.connected_components(G), key=len)
    G_largest = G.subgraph(largest_cc).copy()

    print(f"Largest component: {len(G_largest)} nodes, {G_largest.number_of_edges()} edges.")
    return G_largest

# Method for returning dictionary bloc_id:count for a catagory
def make_category_dict(category_counts, category_name, all_blocks):
    d = (category_counts[category_counts["category"] == category_name]
         .set_index("block_id")["count"]
         .to_dict())

    # Ensure every block in the graph has a value (fill missing with 0)
    return {block: d.get(block, 0) for block in all_blocks}

# Create a dictionary of {category: {block_id: count}} for nodes in graph
def prepare_category_dicts(category_counts, G):
    all_blocks = list(G.nodes())
    categories = category_counts["category"].unique().tolist()

    return {
        cat: make_category_dict(category_counts, cat, all_blocks)
        for cat in categories
    }

# Compute and store Laplacian pseudoinverse (Q matrix) and graph for a city
def compute_and_store_Q(G, city_name, output_dir):
    Q = _ge_Q(G)
```

```
52
53      # Save Q matrix
54      q_path = os.path.join(output_dir, f"{city_name.lower()}_Q_matrix.npy")
55      np.save(q_path, Q)
56
57      # Save graph
58      g_path = os.path.join(output_dir, f"{city_name.lower()}_G_largest.pkl")
59      with open(g_path, "wb") as f:
60          pickle.dump(G, f)
61
62      return Q
63
64
65  # Load precomputed Q
66  def load_Q(city_name, input_dir):
67      q_path = os.path.join(input_dir, f"{city_name.lower()}_Q_matrix.npy")
68      g_path = os.path.join(input_dir, f"{city_name.lower()}_G_largest.pkl")
69
70      if not os.path.exists(q_path) or not os.path.exists(g_path):
71          raise FileNotFoundError(
72              f"Missing Q or graph for {city_name}"
73          )
74
75      Q = np.load(q_path)
76      with open(g_path, "rb") as f:
77          G_largest = pickle.load(f)
78
79      print(f"Graph nodes: {len(G_largest)}, Q shape: {Q.shape}")
80
81      return G_largest, Q
82
83  # Compute a matrix of Generalized Euclidean distances between categories
84  def compute_generalized_euclidean_matrix(G, category_dicts, ge, Q_func):
85
86      categories = list(category_dicts.keys())
87      n = len(categories)
88      ge_matrix = np.zeros((n, n))
89      Q = Q_func(G)
90
91      for i, c1 in enumerate(categories):
92          for j, c2 in enumerate(categories):
93              if j >= i:
94                  d = ge(category_dicts[c1], category_dicts[c2], G, Q=Q)
95                  ge_matrix[i, j] = ge_matrix[j, i] = d
96
97      df_ge = pd.DataFrame(ge_matrix, index=categories, columns=categories)
98      return df_ge
99
100 # Compute variance for each category
101 def compute_variance(category_dicts_largest, G_largest, resistance_matrix):
102     variances = {}
103
104     for category, v_dict in category_dicts_largest.items():
105         var = variance(
```

```
            v_dict,
            G_largest,
            shortest_path_lengths=resistance_matrix,
            kernel="resistance"
        )
        variances[category] = var

    df_var = pd.DataFrame.from_dict(variances, orient="index", columns=["variance"])
    print(df_var)

    return df_var

# Generate distribution of variances from shuffled pois
def shuffled_variances(v_dict, G, resistance_matrix, n_iter=1000):

    # Store variance values from each shuffle
    results = []

    #Extract node IDs and their associated values separately
    nodes = list(v_dict.keys()) # stay fix
    values = list(v_dict.values()) # to shuffle

    # Perform random shuffling multiple times
    for _ in range(n_iter):
        # Shuffle POI values
        random.shuffle(values)

        # Reassign shuffled values to nodes
        shuffled_v = dict(zip(nodes, values))

        # Compute variance for this shuffled configuration
        var_random = variance(shuffled_v, G, shortest_path_lengths=resistance_matrix,
kernel="resistance")

        # Store
        results.append(var_random)

    return np.array(results)


# For each POI category:
# 1. Compute real variance
# 2. Generate random variance distribution
# 3. Compute z-score
def compute_z_scores(category_dicts, G, resistance_matrix, n_iter=1000):
    # Store results for all categories
    random_stats = {}

    # Iterate over each POI category
    for category, v_dict in category_dicts.items():
        # Compute real variance
        real_var = variance(v_dict, G,
                            shortest_path_lengths=resistance_matrix,
                            kernel="resistance")
```

```
159
160         # Generate dstribution from shuffled data
161         rand_vars = shuffled_variances(v_dict, G, resistance_matrix, n_iter)
162
163         # Compute mean and standard deviation
164         mean_rand = np.mean(rand_vars)
165         std_rand = np.std(rand_vars)
166
167         # Compute z-score
168         # Positive z => more spread out than random
169         # Negative z => more clustered than random
170         z = (real_var - mean_rand) / std_rand if std_rand > 0 else np.nan
171
172         # Store all results for this category: variance, average of random variances,
    standard deviation if random variances, normalized difference
173         random_stats[category] = {
174             "real_var": real_var,
175             "mean_rand": mean_rand,
176             "std_rand": std_rand,
177             "z_score": z,
178             "rand_vars": rand_vars
179         }
180
181     # Return as dataframe
182     return pd.DataFrame.from_dict(random_stats, orient="index")
183
184
185 # Precompute resistance once
186 def precompute_resistance(G_largest):
187     resistance_matrix = _resistance(G_largest)
188
189     return resistance_matrix
```

src/livability_scores.py

```python
"""
livability_scores.py

Computes all versions of the final livability score,
both unnormalized and all normalization variants.
Combines generalized network distances and variance-based
z-scores.
"""


import numpy as np
import pandas as pd


# First livability score - unnormalized
def compute_livability_from_matrix(df_ge, z_scores):
   # Extract GE matrix
    GE = df_ge.values

    # Convert z-scores into array and take absolute values
    z = np.abs(np.array(z_scores, dtype=float))

    # Change the shape of z to column vector and muptiply by ge
    scaled = z[:, None] * GE

    # Sum values
    total = scaled.sum()

    # Invert the total so that smaller distances correspond to higher livability
    # Avoid division by zero
    return 1 / total if total != 0 else np.nan


# Normalization variant 1: average instead of sum
def compute_livability_normalized_1(df_ge, z_scores):
    # Extract GE matrix
    GE = df_ge.values

    # Convert z-scores into array and take absolute values
    z = np.abs(np.array(z_scores, dtype=float))

    # Change the shape of z to column vector and muptiply by ge
    scaled = z[:, None] * GE

    # Number of amenity categories
    n=len(z)

    # Sum values
    sum = scaled.sum()

    # Compute average over all category pairs
```

```
    # (n * (n - 1)) corresponds to number of unique directed pairs
    average = sum/(n*(n-1))

    # Invert the total so that smaller distances correspond to higher livability
    # Avoid division by zero
    return 1 / average if average != 0 else np.nan


# Method for adding weigths to edges (favor well-connected areas)
def add_inv_degree_weights(G):
    # Compute node degrees
    deg = dict(G.degree())

    # Assign weight to each edge based on inverse degree of endpoints
    for u, v in G.edges():
        w = 0.5 * (1.0 / max(deg[u], 1) + 1.0 / max(deg[v], 1))
        G[u][v]["weight"] = float(w)

    return G


# Normalization variant 3a: global GE normalization
# Normalize GE matrix so that all values sum to 1 - global normalization
def normalize_ge_sum_to_one(df_ge):
    # Copy GE matrix as float array
    GE = df_ge.values.astype(float).copy()

    # Ensure diagonal is zero (distance of category to itself)
    np.fill_diagonal(GE, 0.0)

    # Total sum of all distances
    s = GE.sum()

    # Normalize if sum is positive
    if s > 0:
        GE = GE / s

    return pd.DataFrame(GE, index=df_ge.index, columns=df_ge.columns)

# Normalization variant 3b: row-wise GE normalization
# Normalize GE matrix row-wise so that each row sums to 1
def normalize_ge_rowwise(df_ge):
    # Copy GE matrix as float array
    GE = df_ge.values.astype(float).copy()

    # Set diagonal to zero
    np.fill_diagonal(GE, 0.0)

    # Compute sum of each row
    row_sums = GE.sum(axis=1, keepdims=True)

    # Divide each row by its sum, avoiding division by zero
    GE = np.divide(GE, row_sums, where=row_sums!=0)

```

```
    return pd.DataFrame(GE, index=df_ge.index, columns=df_ge.columns)


```

src/plotting.py

```python
"""
plotting.py

Contains all plots.
"""


import os
import re
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib import cm, colors
from matplotlib.lines import Line2D
from matplotlib.ticker import FuncFormatter
import numpy as np
import seaborn as sns
import networkx as nx


# Plot city boundary, roads, railways, and water features
def plot_base_map(boundary, roads, railways, water_polygons, water_edges, city_name = None, title=None, save_path=None):

    # Set up figure
    fig, ax = plt.subplots(figsize=(10, 10))

    # Water polygons
    if water_polygons is not None and not water_polygons.empty:
        water_polygons.plot(
            ax=ax,
            color="skyblue",
            edgecolor="none",
            alpha=0.4,
        )

    # Water edges (water polygons outline)
    if water_edges is not None and not water_edges.empty:
        water_edges.plot(
            ax=ax,
            color="navy",
            linewidth=0.4,
            alpha=0.6,
        )

    # Boundary
    if boundary is not None and not boundary.empty:
        boundary.plot(
            ax=ax,
            facecolor="none",
            edgecolor="black",
            linewidth=1,
            linestyle="--",
```

```
        )

    # Roads
    if roads is not None and not roads.empty:
        roads.plot(
            ax=ax,
            color="dimgray",
            linewidth=0.2)

    # Railways
    if railways is not None and not railways.empty:
        railways.plot(
            ax=ax,
            color="firebrick",
            linewidth=0.4)

    # Title
    if title is None:
        title = f"{city_name.capitalize()} base map"
    ax.set_title(title, fontsize=18)

    # Legend
    legend_handles = [
        mpatches.Patch(facecolor="skyblue", edgecolor="navy", alpha=0.5, label="Water"),
        Line2D([0], [0], color="dimgray", linewidth=0.8, label="Roads"),
        Line2D([0], [0], color="firebrick", linewidth=1, label="Railways"),
        mpatches.Patch(facecolor="none", edgecolor="black", linestyle="--", linewidth=1,
label="City boundary")
    ]
    ax.legend(handles=legend_handles, loc="upper right", frameon=True, fontsize=16)

    # Formatting
    ax.axis("off")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        filename = f"{city_name.lower()}_base.pdf"
        save_path = os.path.join(save_path, filename)
        plt.savefig(save_path)
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax



# Plot water polygons and their edges, with optional city boundary.
def plot_water_map(water_polygons, water_edges, boundary=None, city_name=None, title=None,
save_path=None):
    fig, ax = plt.subplots(figsize=(10, 10))

    # Water polygons (filled)
```

```
    if water_polygons is not None and not water_polygons.empty:
        water_polygons.plot(
            ax=ax,
            color="skyblue",
            edgecolor="none",
            alpha=0.4
        )

    # Water edges (outlines)
    if water_edges is not None and not water_edges.empty:
        water_edges.plot(
            ax=ax,
            color="navy",
            linewidth=0.4,
            alpha=0.6
        )

    # Boundary
    if boundary is not None and not boundary.empty:
        boundary.plot(
            ax=ax,
            facecolor="none",
            edgecolor="black",
            linewidth=1,
            linestyle="--"
        )

    # Title
    if title is None:
        title = f"{city_name.capitalize()} water features"
    ax.set_title(title, fontsize=18)

    # Legend
    legend_handles = [
        mpatches.Patch(color="skyblue", alpha=0.4, label="Water polygons"),
        mpatches.Patch(facecolor="none", edgecolor="navy", linewidth=1, alpha=0.6,
label="Water edges")
    ]
    if boundary is not None:
        legend_handles.append(
            mpatches.Patch(facecolor="none", edgecolor="black", linestyle="--",
linewidth=1, label="City boundary")
        )

    ax.legend(handles=legend_handles, loc="upper right", frameon=True, fontsize=16)

    # Formatting
    ax.axis("off")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        filename = f"{city_name.lower()}_water.pdf"
        save_path = os.path.join(save_path, filename)
```

```
156         plt.savefig(save_path)
157         plt.close(fig)
158     else:
159         return fig, ax
160 
161     return fig, ax
162 
163 
164 # Plot blocks
165 def plot_blocks(blocks, city_name=None, title=None, save_path=None):
166     fig, ax = plt.subplots(figsize=(10, 10))
167 
168     # Plot generated blocks
169     if blocks is not None and not blocks.empty:
170         blocks.plot(
171             ax=ax,
172             facecolor='orange',
173             edgecolor='black',
174             linewidth=0.2,
175             alpha=0.6
176         )
177 
178     # Title
179     if title is None:
180         title = f"Blocks"
181     full_title = f"{city_name.capitalize()} {title}" if city_name else title
182     ax.set_title(full_title, fontsize=18)
183 
184     # Legend
185     legend_handles = [
186         mpatches.Patch(facecolor='orange', edgecolor='black', alpha=0.4, label='Blocks'),
187     ]
188     ax.legend(handles=legend_handles, loc='upper right', frameon=True, fontsize=16)
189 
190     # Formatting
191     ax.axis("off")
192     plt.tight_layout()
193 
194     # Save (if requested)
195     if save_path:
196         # Create a safe filename
197         parts = []
198         if city_name:
199             parts.append(city_name.lower())
200         # Sanitize title: lowercase, remove special chars, replace spaces with underscores
201         safe_title = re.sub(r"[^a-zA-Z0-9]+", "_", title.strip().lower())
202         parts.append(safe_title)
203         parts.append(".pdf")
204 
205         filename = "_".join(parts)
206         filepath = os.path.join(save_path, filename)
207         plt.savefig(filepath, dpi=300, bbox_inches="tight")
208         plt.close(fig)
209     else:
```

```
        return fig, ax

    return fig, ax


# Plot all blocks, highlighting suspicious ones in red
def plot_blocks_with_suspicious(blocks, suspicious=None, city_name=None, title=None,
save_path=None):

    fig, ax = plt.subplots(figsize=(10, 10))

    if blocks is None or blocks.empty:
        # Nothing to plot
        ax.set_axis_off()
        return fig, ax

    # Determine remaining vs suspicious
    if (suspicious is not None) and (not suspicious.empty):
        remaining = blocks.drop(suspicious.index, errors="ignore")
    else:
        remaining = blocks
        suspicious = None

    # Plot remaining blocks
    if not remaining.empty:
        remaining.plot(
            ax=ax,
            facecolor='lightgray',
            edgecolor='black',
            linewidth=0.2,
            alpha=0.4
        )

    # Plot suspicious blocks (in red)
    if suspicious is not None and not suspicious.empty:
        suspicious.plot(
            ax=ax,
            facecolor='red',
            edgecolor='black',
            linewidth=0.3,
            alpha=0.8
        )

    # Title
    if title is None:
        title = "Blocks after removal (red = removed)"
    full_title = f"{city_name.capitalize()} {title}" if city_name else title
    ax.set_title(full_title, fontsize=18)

        # Legend
    legend_handles = [
        mpatches.Patch(facecolor='lightgray', edgecolor='black',
                       alpha=0.6, label='Blocks')
    ]
```

```
    if suspicious is not None and not suspicious.empty:
        legend_handles.append(
            mpatches.Patch(facecolor='red', edgecolor='black',
                           alpha=0.8, label='False water blocks')
        )

    ax.legend(handles=legend_handles, loc='upper right',
              frameon=True, fontsize=16)

    # Formatting
    ax.axis("off")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        parts = []
        if city_name:
            parts.append(city_name.lower())
        safe_title = re.sub(r"[^a-zA-Z0-9]+", "_", title.strip().lower())
        parts.append(safe_title)
        parts.append(".pdf")

        filename = "_".join(parts)
        filepath = os.path.join(save_path, filename)
        plt.savefig(filepath, dpi=300, bbox_inches="tight")
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax


# Plot blocks colored by POI count (log scale)
def plot_blocks_with_pois(blocks, city_name=None, title=None, save_path=None):
    fig, ax = plt.subplots(figsize=(10, 8))

    # Handle empty data
    if blocks is not None and not blocks.empty:
        # Compute vmin/vmax for LogNorm (avoid log(0))
        data = blocks["poi_count"]
        vmin = max(1, data.min())   # at least 1
        vmax = data.max()

        blocks.plot(
            ax=ax,
            column="poi_count",
            cmap="OrRd",
            norm=colors.LogNorm(vmin=vmin, vmax=vmax),
            legend=True,
            legend_kwds={
                "label": "Number of POIs",
                "shrink": 0.6
            },
            edgecolor="white",
```

```
            linewidth=0.1
        )

        # Formatting and changing font sizes
        cbar = ax.get_figure().axes[-1]
        cbar.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f"{int(x)}"))
        cbar.tick_params(labelsize=12)
        cbar.set_ylabel("Number of POIs", fontsize=14)

    # Title
    if title is None:
        title = "Blocks with POIs"
    full_title = f"{city_name.capitalize()} {title}" if city_name else title
    fig.suptitle(full_title, fontsize=16, y=0.95)

    # Formatting
    ax.axis("off")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        parts = []
        if city_name:
            parts.append(city_name.lower())
        # Sanitize title: lowercase, remove special chars, replace spaces with underscores
        safe_title = re.sub(r"[^a-zA-Z0-9]+", "_", title.strip().lower())
        parts.append(safe_title)
        parts.append(".pdf")

        filename = "_".join(parts)
        filepath = os.path.join(save_path, filename)
        plt.savefig(filepath, dpi=300, bbox_inches="tight")
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax


# Plot block adjacency graph colored/scaled by a chosen node attribute
def plot_block_graph(G, blocks, attr="poi_count", city_name=None, title=None,
save_path=None):

    # Collect node values
    nodes = list(G.nodes())
    attr_values = np.array([G.nodes[n].get(attr, 0) for n in nodes], dtype=float)

    # Keep non-negative
    attr_values[attr_values < 0] = 0

    # Log scale setup
    if np.any(attr_values > 0):
        vmin = max(1, attr_values[attr_values > 0].min())
        vmax = attr_values.max()
```

```
    else:
        # all zeros - avoid log problems
        vmin, vmax = 1, 1

    # Values actually sent to LogNorm (no zeros)
    attr_for_color = np.where(attr_values <= 0, vmin, attr_values)

    norm = colors.LogNorm(vmin=vmin, vmax=vmax)
    cmap = cm.OrRd
    node_colors = [cmap(norm(v)) for v in attr_for_color]

    # Node sizes
    base_sizes = 3 + attr_values * 2
    node_sizes = np.clip(base_sizes, 5, 200)


    # Positions: block centroids
    pos = {
        row["block_id"]: (row.geometry.centroid.x, row.geometry.centroid.y)
        for _, row in blocks.iterrows()
        if row["block_id"] in G.nodes
    }

    # Plot
    fig, ax = plt.subplots(figsize=(10, 8))

    nx.draw_networkx_edges(
        G, pos, ax=ax,
        edge_color="gray",
        alpha=0.2,
        width=0.5
    )

    nx.draw_networkx_nodes(
        G, pos, ax=ax,
        node_color=node_colors,
        node_size=node_sizes,
        alpha=0.9,
        linewidths=0.05,
        edgecolors="lightgray",
    )

    # Colorbar
    sm = cm.ScalarMappable(norm=norm, cmap=cmap)
    sm.set_array([])

    cbar = plt.colorbar(sm, ax=ax, shrink=0.6)

    cbar.ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f"{int(x)}"))
    cbar.ax.tick_params(labelsize=12)
    cbar.ax.set_ylabel("Number of POIs", fontsize=14)


    # Title
```

```
    if title is None:
        title = "network graph"
    full_title = f"{city_name.capitalize()} {title}" if city_name else title
    fig.suptitle(full_title, fontsize=16, y=0.95)

    # Formatting
    ax.axis("off")
    ax.set_aspect("equal")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        parts = []
        if city_name:
            parts.append(city_name.lower())
        safe_title = re.sub(r"[^a-zA-Z0-9]+", "_", title.strip().lower())
        parts.append(safe_title)
        parts.append(".pdf")

        filename = "_".join(parts)
        filepath = os.path.join(save_path, filename)
        plt.savefig(filepath, dpi=300, bbox_inches="tight")
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax


# Plot largest component
def plot_largest_component(blocks, G, city_name=None, title=None, save_path=None):

    # Compute largest connected component
    largest = max(nx.connected_components(G), key=len)
    blocks = blocks.copy()
    blocks["in_largest"] = blocks["block_id"].isin(largest)

    # Figure
    fig, ax = plt.subplots(figsize=(10, 10))

    # Plot blocks
    blocks.plot(
        ax=ax,
        color=blocks["in_largest"].map({True: "darkorange", False: "lightgray"}),
        edgecolor="white",
        linewidth=0.1,
    )

    # Title
    if title is None:
        title = "Largest connected component"
    full_title = f"{city_name.capitalize()} {title}" if city_name else title
    fig.suptitle(full_title, fontsize=16, y=0.95)

```

```
    # Legend (match style)
    legend_handles = [
        mpatches.Patch(facecolor="darkorange", edgecolor="black", label="Largest connected component"),
        mpatches.Patch(facecolor="lightgray", edgecolor="black", label="Other components")
    ]

    ax.legend(handles=legend_handles, loc="upper right", frameon=True, fontsize=16)

    # Formatting
    ax.axis("off")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        # Create a safe filename
        parts = []
        if city_name:
            parts.append(city_name.lower())
        # Sanitize title: lowercase, remove special chars, replace spaces with underscores
        safe_title = re.sub(r"[^a-zA-Z0-9]+", "_", title.strip().lower())
        parts.append(safe_title)
        parts.append(".pdf")

        filename = "_".join(parts)
        filepath = os.path.join(save_path, filename)
        plt.savefig(filepath, dpi=300, bbox_inches="tight")
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax

def plot_ge_heatmap(df_ge, city_name=None, title="GE distances", order=None, pretty_labels=None, vmin=None, vmax=None, cmap=cm.OrRd, cbar_shrink=0.6, save_path=None):
    # Ordering
    if order is not None:
        df_plot = df_ge.loc[order, order]
    else:
        df_plot = df_ge.copy()
        order = list(df_plot.index)

    if pretty_labels is None:
        pretty_labels = order

    # Color scaling
    if vmin is None:
        vmin = df_plot.values.min()
    if vmax is None:
        vmax = df_plot.values.max()

    # Figure
    fig, ax = plt.subplots(figsize=(10, 10))

```

```
    hm = sns.heatmap(
        df_plot,
        annot=True,
        fmt=".3f",
        annot_kws={"fontsize": 14},
        cmap=cmap,
        vmin=vmin,
        vmax=vmax,
        square=True,
        linewidths=0.4,
        linecolor="white",
        cbar=False,
        ax=ax,
    )

    # Colorbar
    sm = cm.ScalarMappable(norm=colors.Normalize(vmin=vmin, vmax=vmax), cmap=cmap)
    sm.set_array([])
    cbar = plt.colorbar(sm, ax=ax, shrink=cbar_shrink)
    cbar.ax.tick_params(labelsize=12)
    cbar.ax.set_ylabel("GE distance", fontsize=14, labelpad=10)

    # Labels & formatting
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.set_xticklabels(pretty_labels, rotation=45, ha="right", fontsize=16)
    ax.set_yticklabels(pretty_labels, rotation=0, fontsize=16)

    plt.tight_layout()

    if save_path:
        full_title = f"{city_name.capitalize()} {title}" if city_name else title
        safe_title = re.sub(r"[^a-zA-Z0-9]+", "_", full_title.lower())
        filename = f"{safe_title}.pdf"
        filepath = os.path.join(save_path, filename)
        plt.savefig(filepath, dpi=300, bbox_inches="tight")
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax


# Plot distribution of randomized variances vs observed variance for a given category
def plot_variance_distribution_from_results(category, df_z, label_map, CITY_NAME,
save_path=None):

    rand_vars = df_z.loc[category, "rand_vars"]
    real_var  = df_z.loc[category, "real_var"]
    mean_rand = df_z.loc[category, "mean_rand"]
    std_rand  = df_z.loc[category, "std_rand"]
    z         = df_z.loc[category, "z_score"]

    pretty = label_map.get(category, category)
```

```
583
584     # KDE peak calc (kept from your code)
585     plt.figure(figsize=(8, 5))
586     tmp = sns.kdeplot(rand_vars, bw_adjust=1, color="black")
587     plt.clf()
588
589     fig, ax = plt.subplots(figsize=(10, 5))
590
591     sns.histplot(rand_vars, bins=30, stat='density',
592                  color="lightgray", edgecolor=None, alpha=0.6, ax=ax)
593
594     sns.kdeplot(rand_vars, bw_adjust=1, color="dimgray",
595                 linewidth=2, alpha=0.6, label="Distribution", ax=ax)
596
597     ax.axvline(real_var, color="red", linestyle="--", linewidth=2,
598                label=f"Observed variance = {real_var:.2f}")
599
600     ax.axvline(mean_rand - std_rand, color="gray", linestyle="--", linewidth=1)
601     ax.axvline(mean_rand + std_rand, color="gray", linestyle="--", linewidth=1)
602
603     ax.set_xlabel("Variance")
604     ax.set_ylabel("Density")
605     ax.tick_params(axis="both", labelsize=12)
606     ax.grid(True, color="lightgray", alpha=0.3)
607
608     # Title with city name
609     ax.set_title(f"{CITY_NAME} - {pretty}: z-score = {z:.2f}", fontsize=16)
610
611     ax.legend()
612     fig.tight_layout()
613
614     # Save if requested
615     if save_path:
616
617         filename = f"{CITY_NAME}_{category}.pdf"
618         filepath = os.path.join(save_path, filename)
619
620         fig.savefig(filepath, dpi=300, bbox_inches="tight")
621         plt.close(fig)
622     else:
623         plt.show()
624
625     return fig, ax
626
627
628
629 # Plot POI distribution on a base
630 def plot_poi_distribution(boundary, roads, pois, city_name=None, title=None,
    save_path=None):
631
632     # Figure / axis
633     fig, ax = plt.subplots(figsize=(10, 7))
634
635     # Boundary
```

```
    if boundary is not None and not boundary.empty:
        boundary.plot(
            ax=ax,
            facecolor="none",
            edgecolor="black",
            linewidth=1,
            linestyle="--",
        )

    # Roads
    if roads is not None and not roads.empty:
        roads.plot(
            ax=ax,
            color="dimgray",
            linewidth=0.2
        )

    # POIs (red dots, on top)
    if pois is not None and not pois.empty:
        pois.plot(
            ax=ax,
            markersize=3,
            color="red",
            alpha=0.6,
            zorder=5,
        )

    # Title
    if title is None and city_name is not None:
        title = f"{city_name.capitalize()} – POI distribution"
    elif title is None:
        title = "POI distribution"

    ax.set_title(title, fontsize=18)

    # Legend
    legend_handles = []

    # City boundary
    legend_handles.append(
        mpatches.Patch(
            facecolor="none",
            edgecolor="black",
            linestyle="--",
            linewidth=1,
            label="City boundary",
        )
    )

    # Roads
    legend_handles.append(
        Line2D(
            [0], [0],
            color="dimgray",
```

```
            linewidth=0.8,
            label="Roads",
        )
    )

    # POIs
    legend_handles.append(
        Line2D(
            [0], [0],
            marker="o",
            linestyle="None",
            color="red",
            markersize=6,
            label="POIs",
            alpha=0.8,
        )
    )

    ax.legend(handles=legend_handles, loc="upper right", frameon=True, fontsize=14)

    # Formatting
    ax.axis("off")
    plt.tight_layout()

    # Save (if requested)
    if save_path:
        os.makedirs(save_path, exist_ok=True)
        if city_name is not None:
            filename = f"{city_name.lower()}_poi_distribution.pdf"
        else:
            filename = "poi_distribution.pdf"

        full_path = os.path.join(save_path, filename)
        plt.savefig(full_path)
        plt.close(fig)
    else:
        return fig, ax

    return fig, ax

```

scripts/run_pipeline.py

```python
"""
running_pipeline.py

The full analysis pipeline.
Sequentially executes data retrieval, preprocessing, block construction,
graph creation, network analysis and livability score computation.
The main code in this thesis.
"""


import os
import numpy as np
import pandas as pd

from src.data_retrieval import init_osm
from src.preprocessing import (
    load_roads, load_railways, load_water_polygons,
    get_local_crs, reproject_all,
    build_water_edges, prepare_water_geometries
)
from src.prepare_pois import load_pois_with_green, prepare_pois, assign_pois_to_blocks
from src.process_blocks import (
    construct_blocks, filter_water_blocks, filter_small_blocks,
    filter_irregular_blocks, remove_false_water_blocks
)
from src.plotting import (
    plot_base_map, plot_block_graph, plot_blocks, plot_blocks_with_pois,
plot_blocks_with_suspicious, plot_ge_heatmap,
    plot_largest_component, plot_variance_distribution_from_results,
    plot_poi_distribution, plot_water_map
)
from src.construct_graph import build_block_graph
from src.network_analysis import (
    precompute_resistance, get_largest_component,
    prepare_category_dicts, compute_generalized_euclidean_matrix,
    load_Q, compute_and_store_Q, compute_variance, compute_z_scores
)
from src.network_distance import ge
from src.livability_scores import (
    add_inv_degree_weights, compute_livability_from_matrix,
    compute_livability_normalized_1, normalize_ge_rowwise, normalize_ge_sum_to_one
)


# The whole pipeline code - with city name as parameter
def run_pipeline(CITY_NAME):

    print(f"\n=== Starting pipeline for: {CITY_NAME} ===")

    # ----- 0. Create output folders -----
    # Create main and subfolders if they don't exist
    os.makedirs(CITY_NAME, exist_ok=True)
```

```
    save_dir = CITY_NAME

    os.makedirs(os.path.join(save_dir, "figures"), exist_ok=True)
    os.makedirs(os.path.join(save_dir, "data"), exist_ok=True)

    data_save_path = os.path.join(save_dir, "data")
    figures_path = os.path.join(save_dir, "figures")

    print(f"Output folders ready: {figures_path} and {data_save_path}")

    # For plotting z-scores
    order = [
        "food",
        "retail",
        "education",
        "healthcare",
        "infrastructure_transport",
        "culture_leisure",
        "green_spaces",
        "public_services",
        "other_daily_utilities",
    ]
    pretty_labels = [
        "Food",
        "Retail",
        "Education",
        "Healthcare",
        "Infrastructure & transport",
        "Culture & leisure",
        "Green spaces",
        "Public services",
        "Other daily utilities",
    ]
    label_map = dict(zip(order, pretty_labels))

    # ----- 1. Data retrieval (osm init + boundary) -----
    print("Initializing OSM...")
    osm, boundary = init_osm(CITY_NAME, data_save_path)

    # ----- 2. Preprocessing -----
    # Load data
    print("Loading raw layers from OSM (roads, railways, water, POIs)...")
    roads = load_roads(osm)
    railways = load_railways(osm)
    water_polygons = load_water_polygons(osm)
    pois = load_pois_with_green(osm)

    # Check
    print(f"Loaded {len(roads):,} roads")
    print(f"Loaded {len(railways):,} railways")
    print(f"Loaded {len(water_polygons):,} water polygons")

    plot_poi_distribution(boundary, roads, pois, CITY_NAME, "POI distribution",
save_path=figures_path)
```

```
105
106     # Change crs. Compute city-local UTM projection
107     print("Computing city-local CRS and reprojecting all layers...")
108     local_epsg = get_local_crs(boundary)
109     print(f"Using local CRS: EPSG:{local_epsg}")
110
111     # Reproject all layers at once
112     layers = {
113         "boundary": boundary,
114         "roads": roads,
115         "railways": railways,
116         "water_polygons": water_polygons,
117         "pois": pois
118     }
119     layers = reproject_all(layers, local_epsg)
120
121     # Unpack the reprojected layers
122     boundary = layers["boundary"]
123     roads = layers["roads"]
124     railways = layers["railways"]
125     water_polygons = layers["water_polygons"]
126     pois = layers["pois"]
127
128     # Build water edges after reprojection (they are used for constructing blocks)
129     print("Building water edges from water polygons...")
130     water_edges = build_water_edges(layers["water_polygons"])
131
132     # Change water geometries
133     print("Preparing water geometries...")
134     water_polygons = prepare_water_geometries(water_polygons)
135
136     # Prepare pois
137     print("Preparing POIs...")
138     pois = prepare_pois(pois=pois)
139
140     plot_base_map(boundary, roads, railways, water_polygons, water_edges,
    city_name=CITY_NAME, save_path=figures_path)
141     plot_water_map(water_polygons, water_edges, boundary = None, city_name=CITY_NAME,
    save_path=figures_path)
142
143     # ----- 3. Block construction and filtering -----
144
145     # Create initial blocks
146     print("Constructing initial blocks...")
147     initial_blocks = construct_blocks(roads, railways, water_edges)
148
149     # Process blocks
150     print("Filtering blocks...")
151     blocks_no_water = filter_water_blocks(initial_blocks, water_polygons)
152     blocks_no_small = filter_small_blocks(blocks_no_water)
153     blocks_cleaned, suspicious = remove_false_water_blocks(blocks_no_small,
    area_quantile=0.999, compactness_quantile=0.03)
154     blocks_no_irregular = filter_irregular_blocks(blocks_cleaned)
155
```

```
156     blocks = blocks_no_irregular
157 
158     plot_blocks(blocks=initial_blocks, city_name=CITY_NAME, title="initial blocks",
    save_path=figures_path)
159     plot_blocks(blocks=blocks_no_water, city_name=CITY_NAME, title="blocks after water
    filtering", save_path=figures_path)
160     plot_blocks(blocks=blocks_no_small, city_name=CITY_NAME, title="blocks after small
    filtering", save_path=figures_path)
161     plot_blocks_with_suspicious(blocks=blocks_cleaned, suspicious=suspicious,
    city_name=CITY_NAME, title="false water blocks", save_path=figures_path)
162     plot_blocks(blocks=blocks_no_irregular, city_name=CITY_NAME, title="blocks after
    irregular filtering", save_path=figures_path)
163     plot_blocks(blocks=blocks_no_irregular, city_name=CITY_NAME, title="final blocks",
    save_path=figures_path)
164 
165     # Check block counts
166     print(f"Initial blocks constructed: {len(initial_blocks):,}")
167     print(f"Blocks after removing water blocks: {len(blocks_no_water):,}")
168     print(f"Blocks after merging small blocks: {len(blocks_no_small):,}")
169     print(f"Blocks after removing false water blocks: {len(blocks_cleaned):,}")
170     print(f"Final number of blocks: {len(blocks):,}")
171 
172     print("Final block layer prepared")
173 
174 
175     # ----- 4. Assign POIs to blocks -----
176     print("Assigning POIs to blocks...")
177     blocks_with_pois = assign_pois_to_blocks(pois, blocks)
178     blocks = blocks_with_pois
179 
180     plot_blocks_with_pois(blocks=blocks_with_pois, city_name=CITY_NAME, title="block-level
    POI density", save_path=figures_path)
181 
182 
183     # ----- 5. Graph construction -----
184     # Build graph from blocks with POIs
185     print("Building block adjacency graph...")
186     G = build_block_graph(blocks)
187 
188     plot_block_graph(G, blocks, city_name=CITY_NAME, save_path=figures_path)
189 
190     # Select largest connected component
191     print("Extracting largest connected component...")
192     G_largest = get_largest_component(G)
193 
194     plot_largest_component(blocks, G, CITY_NAME, title="largest connected component",
    save_path=figures_path)
195 
196 
197     # ----- 6. Network analysis (Q, GE, resistance, variance, z-scores) -----
198 
199     # Compute Q
200     print("Computing and storing Q matrix for largest component...")
201     Q = compute_and_store_Q(G_largest, CITY_NAME, data_save_path)
202 
```

```
    # Load
    G_largest, Q = load_Q(CITY_NAME, data_save_path)

    # Prepare category dictionaries
    print("Preparing category dictionaries for nodes in largest component...")
    category_counts = blocks.attrs["category_counts"]
    category_dicts_largest = prepare_category_dicts(category_counts, G_largest)

    # Compute GE matrix
    print("Computing generalized Euclidean (GE) matrix...")
    df_ge = compute_generalized_euclidean_matrix(G_largest, category_dicts_largest, ge=ge, Q_func=lambda G: Q)
    df_ge.to_csv(os.path.join(data_save_path, f"{CITY_NAME}_results_GE.csv"), index=False)

    print(f"GE matrix size: {df_ge.shape[0]} × {df_ge.shape[1]}")
    print(f"Number of categories: {len(category_dicts_largest)}")

    # Plot heatmap
    plot_ge_heatmap(df_ge, city_name=CITY_NAME, order=order, pretty_labels=pretty_labels, save_path=figures_path)

    # Precompute resistance once
    print("Precomputing resistance matrix...")
    resistance_matrix = precompute_resistance(G_largest)

    # Compute variance for each category (using precomputed resistance matrix)
    print("Computing variance per category...")
    variance_categories = compute_variance(category_dicts_largest, G_largest, resistance_matrix)

    # Compute z-scores
    print("Computing z-scores...")
    df_z = compute_z_scores(category_dicts_largest, G_largest, resistance_matrix, n_iter=1000)

    # Store after computing df_z
    df_z.to_csv(os.path.join(data_save_path, f"{CITY_NAME}_results_z_scores.csv"), index=True)

    # Plotting z-scores for each category
    for category in df_z.index:
        plot_variance_distribution_from_results(category, df_z, label_map, CITY_NAME, save_path=figures_path)


    # ----- 7. Livability score -----
    # Align df_z to the same order as df_ge. So they correspond exactly
    df_z_aligned = df_z.reindex(df_ge.index)

    # Save z-scores
    z_scores = df_z_aligned["z_score"].values

    # Livabiity score for the city
    print("Computing livability score (unnormalized)...")
```

```
    livability_unnormalized = compute_livability_from_matrix(df_ge, z_scores)
    print("Livability score:", livability_unnormalized)


    # ----- 8. Normalization variants -----

    # Normalization 1 - use the average instead of the sum
    print("Computing livability score variant: average instead of sum...")
    livability_normalized1 = compute_livability_normalized_1(df_ge, z_scores)
    print("Livability score (with using average instead of sum):", livability_normalized1)

    # Normalization 2 - add weights to the edges in graph
    print("Computing livability score variant: weighted graph...")

    # Make a weighted copy of the graph
    G_weighted = G_largest.copy()

    # Add degree-based weights
    G_weighted = add_inv_degree_weights(G_weighted)

    # Recompute Q and GE matrix using the weighted graph
    Q_weighted = compute_and_store_Q(G_weighted, CITY_NAME, output_dir=data_save_path)

    # Prepare category dictionaries
    category_dicts_largest_weighted = prepare_category_dicts(category_counts, G_weighted)

    # Compute GE matrix
    df_ge_weighted = compute_generalized_euclidean_matrix(
        G_weighted, category_dicts_largest_weighted, ge=ge, Q_func=lambda G: Q_weighted
    )

    resistance_matrix_weighted = precompute_resistance(G_weighted)

    # Compute variance for each category (using precomputed resistance matrix)
    variance_categories_weighted = compute_variance(category_dicts_largest_weighted,
G_weighted, resistance_matrix_weighted)

    # Compute z-scores
    df_z_weighted = compute_z_scores(category_dicts_largest_weighted, G_weighted,
resistance_matrix_weighted, n_iter=1000)

    # Align df_z to the same order as df_ge. So they correspond exactly
    df_z_aligned_weighted = df_z_weighted.reindex(df_ge_weighted.index)

    # Save z-scores
    z_scores_weighted = df_z_aligned_weighted["z_score"].values

    # Compute livability score
    livability_normalized2 = compute_livability_from_matrix(df_ge_weighted,
z_scores_weighted)

    print("Livability with weighted edges:", livability_normalized2)


```

```
    # Normalization 3 - normalizing generalized euclidean

    # Normalize GE matrix so that all values sum to 1 - global
    print("Computing livability score variant: GE normalized to sum=1...")
    df_ge_sum1 = normalize_ge_sum_to_one(df_ge)

    # Compute livability again using normalized GE
    livability_normalized3 = compute_livability_from_matrix(df_ge_sum1, z_scores)
    print("Livability score (with normalized GE):", livability_normalized3)

    # Row normalization; per category. Every row sum up to 1
    print("Computing livability score variant: GE row-normalized (each row sums to 1)...")
    df_ge_row = normalize_ge_rowwise(df_ge)

    # Compute livability again using row-normalized GE
    livability_rowwise = compute_livability_from_matrix(df_ge_row, z_scores)
    print("Livability score (row-normalized GE):", livability_rowwise)


    # Normalization 4 - Logarithm of ge
    print("Computing livability score variant: log(GE)...")

    # Copy GE matrix
    GE = df_ge.values.astype(float)

    # Replace zeros to avoid log(0)
    GE[GE <= 0] = 1e-9

    # Apply logarithm
    GE_log = np.log(GE)

    # Create DataFrame again
    df_ge_log = pd.DataFrame(GE_log, index=df_ge.index, columns=df_ge.columns)

    # Compute livability again
    livability_log = compute_livability_from_matrix(df_ge_log, z_scores)
    print("Livability (log-transformed GE):", livability_log)


    # ----- 9. Final score -----
    final_score = livability_unnormalized

    print(f"\nFinal livability score for {CITY_NAME} is {final_score:.4f}.")
    print(f"=== Pipeline finished for: {CITY_NAME} ===\n")

    return final_score
```

try.py

```
"""
try.py

Additional script used for pipeline execution.
"""


from scripts.run_pipeline import run_pipeline

copenhagen_score = run_pipeline("Copenhagen")

gdansk_score = run_pipeline("Gdansk")
```