Home Zone Analyzer

1st Riccardo Benedetti
Matricola: 0001131825
riccardo.benedetti7@studio.unibo.it

***Abstract*—This paper presents the Home Zone Analyzer, an app developed using Flask and PostGIS that facilitates users in finding the optimal residential zones in Bologna using spatial data analysis. The app incorporates users' preferences via an interactive quiz to evaluate Points of Interest (POIs) and their relative importance. Other features also include the dynamic visualization of POIs on an interactive map, a ranking algorithm and spatial autocorrelation analysis using Moran's I index to property clustering patterns evaluation. PostGIS's spatial capabilities are used to answer geospatial queries, while the frontend uses Leaflet's interactive mapping.**

I. Introduction

This report describes the development of a software platform for the management and processing of spatial data, specifically designed to provide recommendations for purchasing real estate in the city of Bologna. The platform uses a backend responsible for data management and a frontend that offers an interactive interface for end users. The primary objective of the system is to analyze user preferences regarding the proximity and density of various Points of Interest (POIs) and to provide targeted recommendations on areas and properties that best meet their needs.

II. Project's Architecture

A. General Structure

The application is organized in a frontend-backend architecture, where the frontend manages the interface, while the backend serves data and takes care of storing data in the database.

```
|-- backend
|   |-- data/
|   |   |-- db.json
|   |-- app.py
|   |-- config.py
|   |-- models.py
|   |-- utils.py
|   |-- Dockerfilebe
|   |-- requirements.txt
|-- frontend
|   |-- src/
|   |   |-- input.css
|   |-- node_modules/
|   |-- index.html
|   |-- map.js
|   |-- script.js
|   |-- styles.css
|   |-- utils.js
|   |-- package.json
|   |-- package-lock.json
|   |-- postcss.config.js
|   |-- tailwind.config.js
|   |-- Dockerfilefe
|-- devcontainer/
|   |-- devcontainer.json
|   |-- Dockerfiledev
|-- k8s
|   |-- backend-deployment.yml
|   |-- db-deployment.yml
|   |-- frontend-deployment.yml
|   |-- postgres-pvc.yml
|-- tests
|   |-- filter_tests.json
|   |-- thunder-collection_Moran Index Test.json
|   |-- thunder-collection_Moran Strong Clustering
|-- init-postgis.sql
|-- docker-compose.yml
|-- README.md
```

Listing 1. Project's directory structure organization

B. Frontend

The frontend component is designed to provide a simple experience for interacting with the system's functionalities.

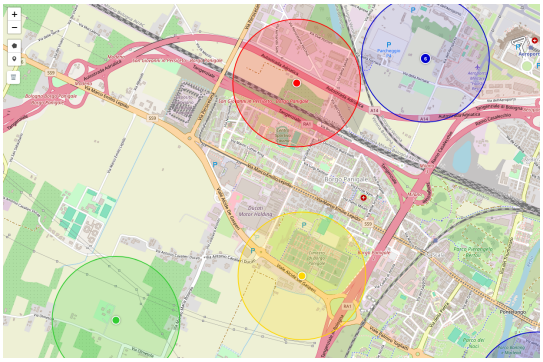
1) index.html and script.js: These files handle the user interaction with an interactive questionnaire designed to get preferences across various categories of Points of Interest (PoI). Users provide ratings on a scale from 0 to 5, choosing the importance of different types of PoIs.

2) map.js and utils.js: These modules provide various mapping functionalities:

- **Interactive Map Visualization:** For viewing and interacting with PoIs.
- **Marker Interaction:** For placing and evaluating potential properties.
- **Customizable Neighborhood Radius Parameter:** For defining analysis areas.
- **Optimal Position Finding:** For automated location recommendations.

C. Backend

The backend component provides the logic of the application and the connection to the database.



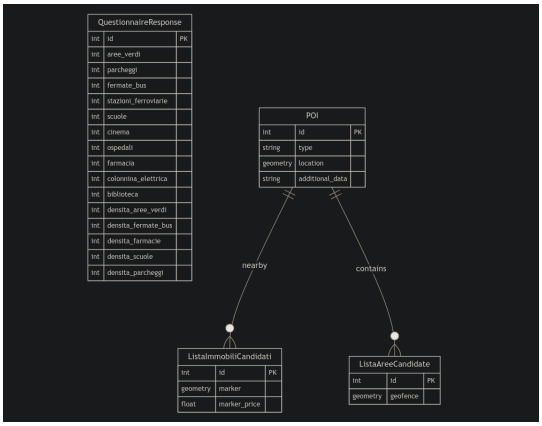


Fig. 1. The Architecture of the DB

*1) **app.py**:* It initializes the application, configures the database connection, sets up CORS, and registers the blueprints for the various API routes. It also includes the Flask Admin configuration for database management and defines the main route for serving the frontend. Here developers can choose to load the POIs directly from Bologna's OpenData portal (which requires a brief time after running the container) or from a local json Database, stored in the /data folder.

*2) **config.py**:* Contains the configuration settings for the application:

- Database connection URL and settings.
- Frontend path configuration.
- Secret key configuration.

*3) **models.py**:* Defines the database models using SQLAlchemy:

- ListaImmobiliCandidati: For storing markers.
- ListaAreeCandidate: For storing candidate areas.
- QuestionnaireResponse: For storing user preferences.
- POI: For managing Points of Interest with spatial data.

*4) **utils.py**:* Contains the core logic and utility functions:

- POI management and updates from external APIs.
- Spatial calculations for neighborhoods.
- Location ranking algorithms.
- Moran's I index calculations.
- Travel time calculations.
- Route handling for various API endpoints.

III. Implementation

This section details the implementation aspects of the system and their location inside the code.

A. *POI Data Management*

1) Data Collection: The app collects data from Bologna's Open Data Portal or a local db for ten distinct POI categories:

- Transportation Infrastructure: parking lots, bus stops, railway stations, and electric charging points.
- Healthcare Facilities: hospitals and pharmacies.
- Educational Facilities: schools.
- Leisure and Culture: parks, cinemas, and libraries.

The data collection process includes handling pagination automatically, with error handling in case of API failures. Special attention is given to bus stop data, where the system implements a clustering algorithm to consolidate multiple stops at the same physical location. It then normalizes coordinates from the different source formats. Those heterogeneous formats are normalized into a standard (latitude, longitude) format and then transformed into PostGIS geometric points for storage.

2) Database Organization: The spatial database structure, defined in models.py, uses PostGIS extensions with the following optimizations:

- Spatial indexing using GiST for efficient proximity queries
- Additional index for coordinate projection operations
- Type-based indexing for category filtering
- Composite indexes for optimizing filtered spatial searches

B. *Optimal Location Finding*

The system implements a mechanism for identifying optimal locations, implemented in utils.py.

1) Grid-based Search: The algorithm employs a grid-based approach to analyze the Bologna area:

- Divides Bologna into a 50x50 grid
- Bounds: latitude 44.4-44.6, longitude 11.2-11.4
- Computes ranking score for each grid point
- Initially filters points with rank above 30

2) Location Diversification: The system implements a diversification algorithm to make sure that all the suggest areas are not in the center, due to the high concentration of POIs it has:

- Maintains minimum distance of 0.008 degrees between locations
- Sorts locations by descending rank
- Selects highest-ranked location as initial point
- Evaluates rank differences between nearby locations

3) Result Classification: Results are classified into score bands for statistical analysis:

- Distribution 0-20: not recommended areas
- Distribution 21-40: acceptable areas
- Distribution 41-60: good areas

- Distribution 61-80: very good areas
- Distribution 81-100: excellent areas

C. Spatial Autocorrelation Analysis

The system implements a comprehensive spatial autocorrelation analysis using Moran's I index, primarily implemented in utils.py through specialized endpoints and calculation functions.

1) Spatial Pattern Detection: The system calculates two distinct Moran's I indices: For prices:

$$I_{prices} = \frac{n}{\sum_i \sum_j w_{ij}} \cdot \frac{\sum_i \sum_j w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_i (x_i - \bar{x})^2}$$

where:

- n is the number of properties
- w_{ij} is the spatial weight between properties i and j
- x_i is the price of property i
- $\bar{x}$ is the mean price

For POI density:

$$I_{density} = \frac{n}{\sum_i \sum_j w_{ij}} \cdot \frac{\sum_i \sum_j w_{ij}(y_i - \bar{y})(y_j - \bar{y})}{\sum_i (y_i - \bar{y})^2}$$

where:

- n is the number of properties
- w_{ij} is the spatial weight between properties i and j
- y_i is the POI count near property i
- $\bar{y}$ is the mean POI count

The spatial weights w_{ij} are calculated using an exponential distance decay function:

$$w_{ij} = e^{-2d_{ij}/h}$$

where d_{ij} is the distance between locations i and j, and h is the threshold distance (1000m).

D. Location Ranking Algorithm

The system implements a location ranking system primarily in utils.py through the `calculate_location_rank()` and `calculate_rank()` functions.

1) Scoring Mechanism: The ranking process evaluates locations through multiple stages:

- Creates circular buffers around evaluation points
- Applies distance decay to POIs near buffer boundaries
- Normalizes densities against city-wide averages
- Integrates user preferences with spatial metrics

2) Density Computation: The system implements a weighted density calculation:

$$D_{local} = \frac{N_{poi}}{\pi r^2}$$

where:

- N_{poi} is the count of POIs within radius r
- r is the user-defined neighborhood radius (default 500m)

The normalized density score is then computed as:

$$D_{score} = min(\frac{D_{local}}{D_{city}} \cdot 100)$$

where D_{city} is the average city-wide POI density.

3) Weight Distribution: The final rank combines multiple components:

- 70% weight for normalized POI density scores
- 30% weight for user preference scores
- Additional weights based on POI type importance
- Distance-based score degradation for peripheral POIs

4) Score Normalization: The system implements several normalization steps:

- Density scores capped at 100 points
- User preferences scaled from 0-5 to percentages
- Distance decay applied through exponential function
- Final scores normalized to 0-100 range

E. Travel Time Analysis

The system implements a travel time-based POI filtering mechanism through OpenRouteService integration, with functionalities implemented in utils.py.

1) Routing Modes: The system supports multiple transportation modes through OpenRouteService API:

- Walking routes (`foot-walking`)
- Cycling routes (`cycling-regular`)
- Driving routes (`driving-car`)

2) Travel Time Calculation: The `calculate_travel_time` function implements:

- OpenRouteService API calls.
- Source point centered in Bologna (44.4949, 11.3426)
- Time threshold filtering based on user preferences
- Coordinate transformation for API compatibility

3) Result Processing: The filtered results are then passed to the front-end through the function's route.

F. Frontend Visualization

The system implements a comprehensive visualization layer primarily through Leaflet.js integration, with functionalities distributed across map.js and utils.js.

- Color-coded markers are based on ranking score.
- Clustering system for POI visualization:
  - Dynamic cluster resizing based on zoom level
  - Custom cluster icons with POI count
- Marker placement and evaluation
- Polygon drawing for area analysis
- Buffer zone radius adjustment
- POI filtering by category and travel time
- Real-time rank updates
- Price input for property markers

IV. Technologies Used

A. Backend Technologies

- **Python 3.9**: Core programming language for the backend implementation
- **Flask Framework**: web framework used for API development and server-side logic
- **Flask Extensions**:
  - Flask-SQLAlchemy for ORM support

  - Flask-Admin for administrative interface
  - Flask-CORS for cross-origin resource sharing
  - Flask-Caching for performance optimization

B. Frontend Technologies

- **Node.Js**: used to install the packages required in the front-end
- **Leaflet.js**: Open-source JavaScript library for interactive maps
- **Tailwind CSS**: Utility-first CSS framework for styling
- **DaisyUI**: Component library for Tailwind CSS
- **Vanilla JavaScript**: Used for DOM manipulation and AJAX requests

C. Database and Spatial Analysis

- **PostgreSQL 15**: Primary database system
- **PostGIS 3.3**: Spatial database extender for PostgreSQL
- **GeoAlchemy2**: Spatial extensions for SQLAlchemy

D. External Services

- **OpenRouteService API**: Used for calculating travel times and routing
- **Bologna Open Data Portal**: Source for Points of Interest data

E. Deployment and Infrastructure

- **Docker**: Container platform for application packaging
- **Docker Compose**: Multi-container development environment
- **Kubernetes**: Container orchestration for production deployment
- **Nginx**: Web server for static content delivery

V. Results

The goal of this project was to develop a simple and user friendly platform to help its users in finding housing in the city of Bologna. The first of key achievements of this project is surely the looks and the user experience of the app, both of which are simple but effective, while looking good. The backend is solid and manages different useful features like POI ranking and optimal location finding, and also complex tasks like the calculation of spatial and price correlation between user inserted markers.

A. Future Improvements

Although the app manages its current task in an efficient way, there is for sure space for improvement:

- Regional Support: in future the application may support other cities or zones, despite this needing to switch to a global POI furnisher instead of the one of Bologna's administration.
- More categories of POIs: adding more categories of POIs to vote in the questionaire would allow differente users with other needs to use the app.
- Real Time Price Tracking: instead of manually setting the price of an housing, the application may automatically fetch for them from an external source, for example with an API

By integrating this features this application may become useful to a broad amount of users, helping them to find accomodations in various regions, with nearby points of interest that suit their needs.

References