

MINISTRY OF HIGHER EDUCATION, RESEARCH AND INNOVATION  
PARIS-SACLAY UNIVERSITY



**Project's Subject:**

# Exploring UNESCO World Heritage Sites and Their Proximity to Major Cities: A Geospatial Knowledge Graph Approach

**As part of the Web of Data Course**

**Done by:** TOUATI Yanis

**Supervisor:** Dr. SAÏS Fatiha and Mr. RAAD Joe

**Class:** M2 Data Sciences

**Academic Year:** 2024 - 2025

# Contents

<b>1</b>	<b>Introduction and Objectives</b>	<b>2</b>
<b>2</b>	<b>Find Geospatial Data and the Datasets Used</b>	<b>2</b>
2.1	UNESCO World Heritage Sites . . . . .	2
2.2	Major Cities Dataset . . . . .	3
<b>3</b>	<b>Cleaning and Converting to a Knowledge Graph</b>	<b>3</b>
3.1	Cleaning the Data in Python . . . . .	3
3.1.1	Cleaning the UNESCO World Heritage Sites Data . . . . .	4
3.1.2	Cleaning the Cities Data . . . . .	4
3.2	Converting to RDF Triples Using OntoRefine . . . . .	4
3.2.1	Loading the Data into OntoRefine . . . . .	4
3.2.2	Mapping Cities Data to RDF . . . . .	4
3.2.3	Mapping UNESCO World Heritage Sites Data to RDF . . . . .	5
3.3	Store the RDF in GraphDB . . . . .	6
<b>4</b>	<b>Link to Wikidata</b>	<b>6</b>
<b>5</b>	<b>Answering the Question</b>	<b>7</b>
<b>6</b>	<b>Visualization of the Results on a Map</b>	<b>8</b>
6.1	Attempt with YASGUI . . . . .	8
6.2	Solution: Using Python and Folium . . . . .	8
<b>7</b>	<b>Limitations and Challenges</b>	<b>10</b>
7.1	Data Quality Issues . . . . .	10
7.2	Inconsistencies in Wikidata . . . . .	10
7.3	AI Assistance in Handling Data Challenges . . . . .	10
7.4	Conclusion . . . . .	10

# 1 Introduction and Objectives

**Notice:** You can view the video presentation of this project by following the link below:  
Video Presentation

This project leverages Linked Data to integrate and analyze information from various sources, demonstrating the power of semantic technologies. The question : *which UNESCO World Heritage Sites are located within 50 km of major cities with populations over 2.5 million, and what are the primary languages spoken in these cities?* addresses how interconnected datasets can answer complex queries, and the motivation behind it is :

- **Semantic Integration:** Combines city and UNESCO data to create meaningful connections between datasets.
- **SPARQL and RDF:** Uses SPARQL to query linked RDF datasets, illustrating how structured queries can combine information.
- **Geospatial Data:** Geographical proximity as a linking factor enables spatial analysis in a linked data context.
- **Real-World Utility:** Provides insights for urban planning, cultural preservation, and tourism using interconnected, reusable data.

This project highlights how Linked Data principles enable the creation of interoperable, extensible data solutions. The full code and documentation can be accessed on the [project's GitHub repository].

## 2 Find Geospatial Data and the Datasets Used

### 2.1 UNESCO World Heritage Sites

To gather information about UNESCO World Heritage Sites, I queried Wikidata using SPARQL. In this step, I specifically focused on retrieving:

- The name of the UNESCO World Heritage Site
- The unique link to the site on Wikidata
- The geographical location of the site (latitude and longitude)

I utilized Python and the 'SPARQLWrapper' library to send a SPARQL query to the Wikidata endpoint and collect the required information. Below is the code used for querying the Wikidata SPARQL endpoint and processing the results:

```
endpoint_url = "https://query.wikidata.org/sparql"
sparql = SPARQLWrapper(endpoint_url)
query = """
SELECT ?site ?siteLabel ?location WHERE {
    ?site wdt:P31 wd:Q9259;    # Heritage site instance
        wdt:P625 ?location.    # Geographical coordinates
    SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
"""
sparql.setQuery(query)
sparql.setReturnFormat(JSON)
results = sparql.query().convert()
data = []
for result in results["results"]["bindings"]:
```

```

site = result["site"]["value"]
site_label = result["siteLabel"]["value"]
location = result["location"]["value"]
data.append({"Site": site, "SiteLabel": site_label, "Location": location})

heritage_df = pd.DataFrame(data)
heritage_df

```

The results were then stored in a pandas dataframe, which allowed me to easily view and manipulate the data for further analysis. The dataframe includes columns for the site name, its unique Wikidata link, and the geographical coordinates in the form of latitude and longitude. For more information on retrieving data from Wikidata, you can refer to their SPARQL endpoint (<https://query.wikidata.org/>).

## 2.2 Major Cities Dataset

In addition to UNESCO World Heritage Sites, I also needed data about major cities to answer the research question. The cities data is crucial for identifying cities with populations over 2.5 million, along with their geographical coordinates. For this, we used the GeoNames dataset, specifically the file `cities1000.txt`.

The `cities1000.txt` file contains data for cities with populations over 1,000 inhabitants, including the following information:

- City name
- Population
- Latitude and Longitude

This dataset was downloaded from the GeoNames Data Portal. It can be accessed via the following link: [GeoNames Data Portal](<http://www.geonames.org/>).

Once the dataset was downloaded, it was processed and cleaned to extract relevant information, such as cities with populations greater than 2.5 million and their respective latitude and longitude values. The dataset was then loaded into a pandas dataframe.

After retrieving and processing both datasets, we now have two key data sources:

- **UNESCO World Heritage Sites Dataset:** Contains the name, Wikidata link, and geographical coordinates of UNESCO World Heritage Sites.
- **Major Cities Dataset:** Contains the name, population, and geographical coordinates of cities with populations greater than 1000 inhabitants.

## 3 Cleaning and Converting to a Knowledge Graph

In this step, I clean the data and convert it into a knowledge graph format. The process involves two main stages: cleaning the data using Python and then transforming the cleaned datasets into RDF triples, which can be queried and analyzed as a knowledge graph.

### 3.1 Cleaning the Data in Python

I begin by cleaning the datasets for both UNESCO World Heritage Sites and cities. The cleaning process involves selecting relevant columns and ensuring the data is formatted appropriately for further use.

### 3.1.1 Cleaning the UNESCO World Heritage Sites Data

For the UNESCO World Heritage Sites dataset, I selected the necessary columns for the analysis. Specifically, I chose the site label and location (latitude and longitude) as the key data points.

The cleaning process was implemented in Python as follows:

```
heritage_df_cleaned = heritage_df[['SiteLabel', 'Location']]
# Assuming 'Site' is the name of the heritage site
```

Here, I retained only the `SiteLabel` (the name of the heritage site) and the `Location` (which contains the geographical coordinates) columns.

### 3.1.2 Cleaning the Cities Data

Similarly, the cities dataset was cleaned by selecting the relevant columns, such as the city name, latitude, longitude, and population. The following Python code was used for this:

```
cities_df_cleaned = cities_df[[1, 2, 3, 4, 5, 14]]
cities_df_cleaned.columns = ['CityID', 'CityName', 'OtherInfo', 'Longitude',
                             'Latitude', 'Population']
```

Next, I formatted the latitude and longitude data into the Well-Known Text (WKT) format, **which is used to represent geographical data in a standardized way for GeoSPARQL**:

```
cities_df_cleaned['Location'] = 'POINT(' + cities_df_cleaned['Longitude'].astype(str) + ' ' +
cities_df_cleaned['Latitude'].astype(str) + ')'
```

Finally, I selected only the necessary columns for the RDF conversion:

```
cities_df_final = cities_df_cleaned[['CityName', 'Population', 'Location']]
```

The cleaned cities dataset now contains the following columns: City Name, Population, and Location (in WKT format).

## 3.2 Converting to RDF Triples Using OntoRefine

Once the datasets were cleaned, we proceeded with converting them into RDF triples using OntoRefine. This step allows us to structure the data in a way that is compatible with semantic technologies, such as SPARQL queries.

### 3.2.1 Loading the Data into OntoRefine

The cleaned CSV datasets were loaded into OntoRefine, a tool that allows the conversion of tabular data into RDF format. After importing the datasets, we used the Visual RDF Mapper and SPARQL Query Editor to define the mappings between the CSV columns and RDF properties.

### 3.2.2 Mapping Cities Data to RDF

We mapped the cities data to RDF triples using the GeoSPARQL vocabulary to represent geographical data. Below is the mapping for the cities data:

```
BASE <http://example.com/base/>
SELECT *
WHERE {
  BIND( ?c_CityName AS ?CityName ) .
  BIND( ?c_Population AS ?Population ) .
  BIND( ?c_Location AS ?Location ) .
}
```

In the RDF format, each city is represented by a subject, with the **CityName** and **Population** as literal values. The **Location** is represented as a geometry (WKT).

Visual RDF Mapper

SPARQL Query Editor

Configuration Preview Both

All mapping changes saved Save Download JSON Upload JSON RDF Open in GraphDB New Mapping

CityName Population Location

Base IRI  
http://example.com/base/

Use the current repository prefixes or add new using the Turtle or SPARQL syntax. i.e. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ex geo rdfs

CityName	<IRI>	a	<IRI>	ex: City	<IRI>
	geo: hasGeometry	<IRI>		Location	"Literal"
	ex: population	<IRI>		Population	"Literal"
+ Subject		+ Predicate		+ Object	

Figure 1: Data Linking 1.0

### 3.2.3 Mapping UNESCO World Heritage Sites Data to RDF

Similarly, the UNESCO World Heritage Sites data was mapped to RDF. Each heritage site is represented as a subject, with the site label and geographical location mapped to RDF properties:

```
BASE <http://example.com/base/>
SELECT *
WHERE {
  BIND( ?site_SiteLabel AS ?SiteLabel ) .
  BIND( ?site_Location AS ?Location ) .
}
```

In the RDF triples for the UNESCO sites, the **SiteLabel** is a literal, while the **Location** is represented as a geo-spatial property.

Visual RDF Mapper

SPARQL Query Editor

Configuration Preview Both

All mapping changes saved Save Download JSON Upload JSON RDF Open in GraphDB New Mapping

SiteLabel Location

Base IRI  
http://example.com/base/

Use the current repository prefixes or add new using the Turtle or SPARQL syntax. i.e. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ex geo rdfs

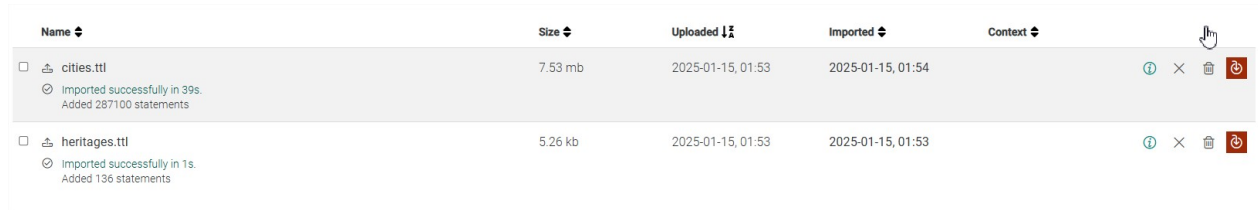
ex: Site	<IRI>	rdfs: label	<IRI>	SiteLabel	"Literal"
ex: Site	<IRI>	geo: hasGeometry	<IRI>	Location	<IRI>
+ Subject		+ Predicate		+ Object	

Figure 2: Data Linking 2.0

The results of this part are stored in the two knowledge graphs, available in the files [cities.ttl](#) and [heritages.ttl](#).

### 3.3 Store the RDF in GraphDB

Once the datasets are converted into RDF format using OntoRefine, the next step is to store the RDF in GraphDB. This involves exporting the RDF data from OntoRefine and uploading it to a new repository in GraphDB, where it can be queried and further analyzed.



Name	Size	Uploaded	Imported	Context
<input type="checkbox"/> cities.ttl Imported successfully in 39s. Added 287100 statements	7.53 mb	2025-01-15, 01:53	2025-01-15, 01:54	
<input type="checkbox"/> heritages.ttl Imported successfully in 1s. Added 136 statements	5.26 kb	2025-01-15, 01:53	2025-01-15, 01:53	

Figure 3: KGs imported in GraphDB

## 4 Link to Wikidata

Here I aimed to enrich our knowledge graph by linking the cities in our dataset to relevant information on Wikidata, specifically their population and primary language. To achieve this, I wrote a ***federated*** SPARQL query that combined data from our local knowledge graph (stored in GraphDB) with external data from Wikidata.

The query was constructed using multiple SPARQL services:

- The first service was connected to the local repository in GraphDB, where we had stored our cities' data, including the city name, population, and geographical location. The query selects these details for each city.
- The second service queried Wikidata to retrieve the official language (`wdt:P37`) and population (`wdt:P1082`) (it wasn't necessary because we already have it, but did it to check the values) of the cities. This was achieved by matching city names to their corresponding Wikidata entries and extracting the necessary properties.

Here is the SPARQL query that was used:

```
PREFIX base: <http://example.com/base/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX schema: <http://schema.org/>

CONSTRUCT {
  ?city base:CityName ?CityName .
  ?city base:Population ?Population .
  ?city base:Location ?Location .
  ?city base:Language ?Language .
}

WHERE {
  SERVICE <http://localhost:7333/repositories/ontorefine:1758731059719> {
    BIND( ?c_CityName AS ?CityName ) .
    BIND( ?c_Population AS ?Population ) .
    BIND( ?c_Location AS ?Location ) .
  }

  SERVICE <https://query.wikidata.org/sparql> {
    ?city_wd rdfs:label ?CityName .
  }
}
```

```

    ?city_wd wdt:P37 ?language_wd .
    ?language_wd rdfs:label ?Language .
    FILTER(LANG(?Language) = "en")
}

BIND(URI(CONCAT("http://example.com/base/", ENCODE_FOR_URI(?CityName))) AS ?city)
}

```

By running this federated query in GraphDB's SPARQL editor, I was able to enrich our local dataset with the population and official language of each city. This process effectively links the cities in our dataset to the corresponding entries in Wikidata, thereby enhancing the knowledge graph with external data from a reliable source. The results of the analysis are stored in the knowledge graph, available in the file [cleaned-enriched-cities.ttl](#)

## 5 Answering the Question

At this level, I had to identify cities and heritage sites that are located within 50 kilometers of each other, while also retrieving additional information such as the population and primary language for each city. To achieve this, I wrote a SPARQL query utilizing the GeoSPARQL `geof:distance` function, which calculates the distance between two geographical points (in this case, the cities and heritage sites).

The query retrieves the following data for each city and heritage site within the 50 km radius:

- **City name** and **population**, along with the **primary language**.
- **Heritage site name** and its **location**.
- The **geographical coordinates** (latitude and longitude) of both the city and heritage site.

I used the following SPARQL query to perform this task:

```

PREFIX base: <http://example.com/base/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?cityName ?cityPopulation ?heritageSiteName ?heritageSiteLocation ?
language ?cityLat ?cityLon ?siteLat ?siteLon WHERE {
    ?city base:CityName ?cityName ;
          base:Population ?cityPopulation ;
          base:Location ?cityLocation ;
          base:Language ?language .

    BIND(STRBEFORE(STRAFTER(STR(?cityLocation), "("), " ") AS ?cityLon)
    BIND(STRAFTER(STRAFTER(STR(?cityLocation), "("), " ") AS ?cityLat)
    BIND(REPLACE(?cityLat, "\", "") AS ?cityLat)

    ?heritageSite base:HeritageSiteName ?heritageSiteName ;
                  base:Location ?heritageSiteLocation .

    BIND(STRBEFORE(STRAFTER(STR(?heritageSiteLocation), "("), " ") AS ?siteLon)
    BIND(STRAFTER(STRAFTER(STR(?heritageSiteLocation), "("), " ") AS ?siteLat)
    BIND(REPLACE(?siteLat, "\", "") AS ?siteLat)

    BIND(geof:distance(?cityLocation, ?heritageSiteLocation) AS ?distance)
    FILTER(?distance <= 50) # Filter for sites within 50 km
}

```



- Explanation of the Query: This query works in the following manner:

- **City Data:** The first part of the query retrieves the city name, population, location, and primary language for each city from our enriched graph.
- **Heritage Site Data:** The second part queries the heritage site data, retrieving the name and location of each UNESCO World Heritage Site.
- **Geographical Coordinates:** The query uses the `BIND` function to extract the latitude and longitude of both the cities and heritage sites from their geographical location in WKT format.
- **Distance Calculation:** The `geof:distance` function calculates the distance between the city's location and the heritage site's location. The result is filtered to include only those pairs where the distance is less than or equal to 50 kilometers.

- Outcome: This query was used to answer the question of which cities are within 50 kilometers of UNESCO World Heritage Sites. The result of the query can be seen in the file [SPARQL\\_GraphDB\\_RESULTS.json](#)

## 6 Visualization of the Results on a Map

### 6.1 Attempt with YASGUI

YASGUI provides an intuitive interface for executing SPARQL queries and visualizing results directly on a map. The process involves loading the local GraphDB endpoint into YASGUI, running the SPARQL query, and selecting a map-based visualization to display the locations of cities and heritage sites. However, I faced an issue with the YASGUI interface, where the geospatial visualization feature was not working as expected. Despite multiple attempts to configure the map-based visualization, I was unable to render the cities and heritage sites on the map due to a compatibility problem between the geospatial data format returned by GraphDB and YASGUI's mapping feature.

### 6.2 Solution: Using Python and Folium

Given the limitations with YASGUI, I decided to use an alternative approach by leveraging Python and the `Folium` library for geospatial visualization. `Folium` allows for the creation of interactive maps and is well-suited for visualizing geographical data such as city locations and heritage sites.

The following steps were taken:

- Exported the SPARQL query results to a CSV file, containing the city names, heritage site names, and their respective geographical coordinates (latitude and longitude).
- Used Python to read the CSV file and process the data.
- Created a map centered around the cities and plotted both the cities and heritage sites as markers on the map using `Folium`.
- Added interactive popups for each marker, displaying additional information such as the population of the city and the name of the heritage site.

This solution provided a fully interactive map that could be easily shared and visualized in a web browser. although YASGUI offered a potential solution, technical challenges prevented its use. However, by using Python and `Folium`, I was able to successfully visualize the data on a map and gain valuable insights into the geographical distribution of cities and heritage sites. The whole process can be seen in the jupyter notebook file [WOD-Project.ipynb](#)

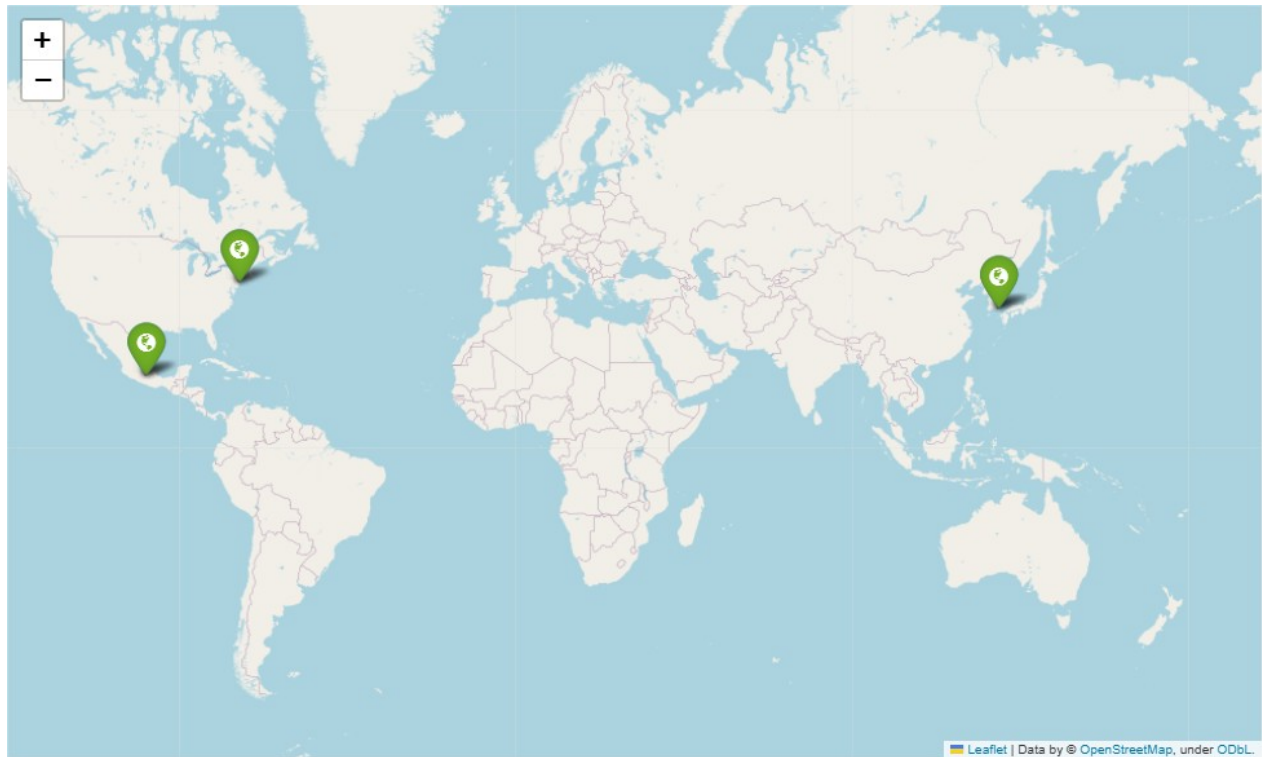


Figure 4: Map Visualization using Python

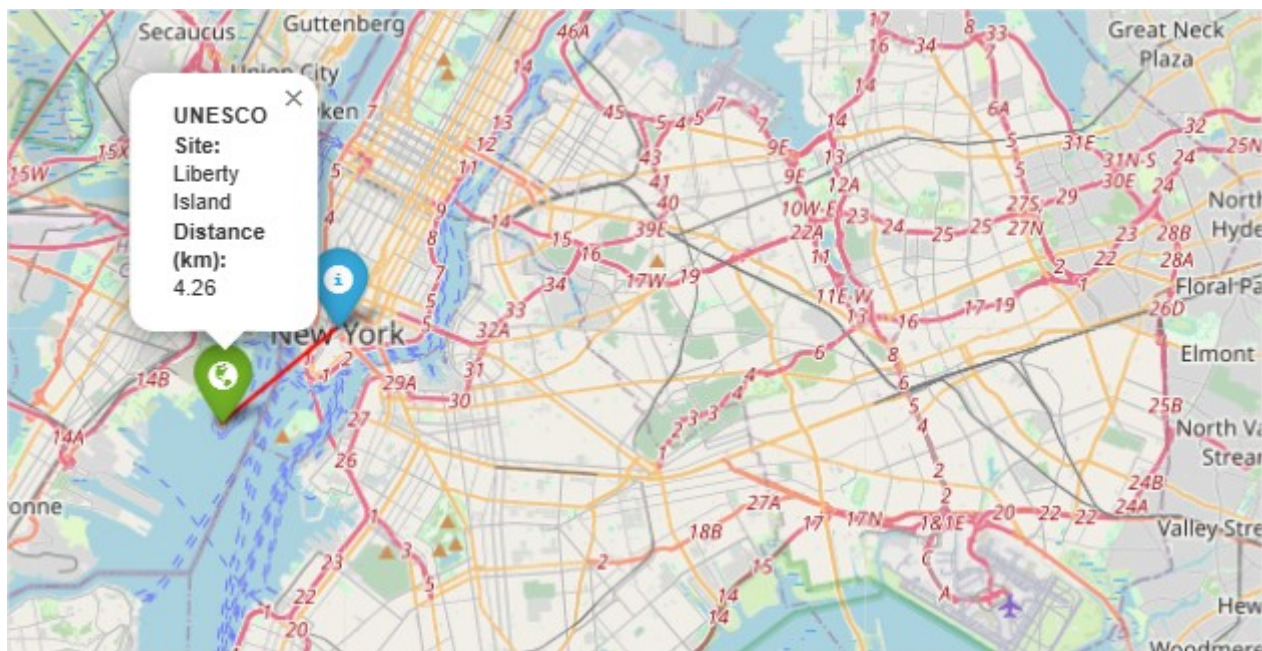


Figure 5: Result Example using Map Visualization with Python

## 7 Limitations and Challenges

While the project provided valuable insights, several challenges arose due to the quality and completeness of the datasets used.

### 7.1 Data Quality Issues

A major limitation was the quality of the data, as some crucial information was missing or poorly defined. For instance, the population as well as certain main Unesco sites missing from Wikidata and primary language for certain cities were missing, creating gaps in the analysis. and also the irrelevant and old information that the cities dataset had. These issues were more pronounced when trying to enrich the dataset using external sources like Wikidata. All these issues did lead to having limited results not fully complete or accurate.

### 7.2 Inconsistencies in Wikidata

Wikidata was a helpful resource for enriching the local knowledge graph, but inconsistencies across city pages posed a challenge. Some cities lacked the property P37 (official language), leading to missing language data. This lack of uniformity meant that certain cities couldn't be fully enriched, affecting the overall consistency of the dataset.

### 7.3 AI Assistance in Handling Data Challenges

AI was instrumental in addressing these issues. It helped with data cleaning, enrichment, and processing, especially when matching cities and heritage sites to their Wikidata entries. AI techniques also assisted in filling missing values and refining geospatial data as well as the structuration of this report. Although some gaps remained, AI significantly improved the dataset's quality.

### 7.4 Conclusion

In conclusion, while the analysis provided valuable insights, it faced challenges due to incomplete and inconsistent data. However, AI played a key role in overcoming these challenges, enhancing data quality and helping produce meaningful results despite the limitations.

**Notice:** You can view the video presentation of this project by following the link below:  
Video Presentation