Spatial Algorithms – Clustering of Flickr Photos in German Cities and Parks

1. Motivation

Social media platforms have increasingly become a crucial source of geographic data. Millions of people share photos online every day, many of which contain location information in the form of GPS coordinates. This *Volunteered Geographic Information* (VGI) has attracted growing attention in geographic research, since it reveals how people move through, and interact with space (Goodchild, 2007). Flickr, one of the oldest photo-sharing platforms, is particularly well studied in this context, because its data has been openly available to researchers for many years through the Yahoo Flickr Creative Commons 100 Million dataset (YFCC100M) and similar archives (Thomee et al., 2016).

Existing research has used Flickr data to understand tourism patterns (Kádár, 2014), map urban activity (Croitoru et al., 2013), identify points of interest (Crandall et al., 2009), and even track seasonal behavior across landscapes (Sinclair et al., 2020). Many of these studies focus on photo clustering and its underlying reasons. Often pictures are not randomly taken in space but are rather situated at certain points of interest, such as specific captivating landscapes or locations of people meeting. This makes Flickr data serve as a proxy for human spatial behavior in a wide range of research domains, though with important caveats.

We selected Germany as our case study region, since it combines dense urban environments with large protected natural areas. Germany's major cities, including Berlin, Hamburg, Munich, and Cologne, are among Europe's most visited urban tourist destinations, and geotagged photo density has been shown to correlate strongly with registered overnight stays across European cities (Kádár, 2014). At the same time, Germany has an extensive system of national parks, nature parks, and biosphere reserves that attract hikers, nature enthusiasts, and tourists. Comparing how Flickr photos are distributed across these two, very different types of space – dense, event-rich cities versus large, quieter natural parks – allows us to ask a meaningful spatial question:

How does the spatial distribution of Flickr photo data differ between cities and parks in Germany in terms of clustering, measured with the Nearest Neighbor Index (NNI)?

We hypothesize that photos in cities are more clustered than in parks. The NNI is a well-established measure in spatial statistics that compares the average observed distance between nearest neighbors to the expected average distance under complete spatial randomness (Clark & Evans, 1954). A value below one indicates clustering; a value above 1 indicates dispersion, and a value of exactly one suggests randomness. By applying this measure to Flickr points within city boundaries on the one hand and within nature park boundaries on the other, we can compare the degree of spatial clustering between these two contexts.

Similar research has been conducted in other settings. Straumann et al. (2014) analyzed the spatial structure of Flickr activity across Switzerland and found strong clustering around tourist hotspots and transportation nodes. Kisilevich et al. (2010) examined the clustering behavior of geotagged photos across multiple cities and found systematic concentration around landmarks and city centers. Hollenstein and Purves (2010) compared Flickr-based place descriptions with official geographic boundaries, while Li et al. (2013) used geotagged photos to analyze tourism behavior in national parks. Our project builds on this body of work by focusing specifically on the contrast between urban and natural park environments in Germany and by applying the NNI as the primary clustering metric.

2. Methods

The project follows a structured pipeline consisting of data acquisition and cleaning, spatial assignment of photo points to city and park polygons, NNI computation, and validation and visualization. In this section,

the main steps are described. A specific theoretical description of each algorithm can be found in the detailed description of the 3 key-algorithms, implemented in this project.

1.1 Algorithm 1: Spatial Index

The raw Flickr dataset of approximately 13.7 million points is too large for direct spatial queries. Thus, a two-step filtering approach is applied (Fig. 1). First, the full dataset is scanned in chunks of 5'000,000 rows and any point falling outside the combined bounding boxes of all city and park polygons is discarded using vectorized NumPy comparisons. The remaining candidates are then inserted into a grid-based spatial index (Point Index) that divides the study area into regular cells of 0.05° (approximately 5.5 km). Each candidate is assigned to a cell based on its longitude and latitude (Fig. 2). When processing a polygon, only points stored in cells that overlap the polygon's bounding box are retrieved, reducing the candidate set from millions to a few hundred per polygon.

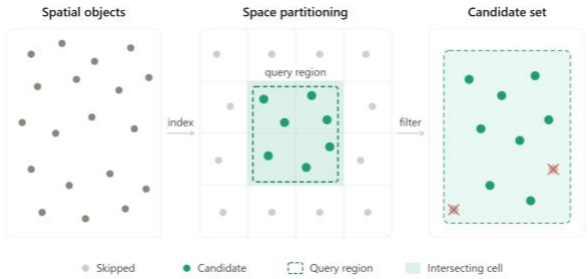


Figure 1. Concept visualization of spatial indexing (Image generated by ClaudeAI)

1.2 Algorithm 2: Point in Polygon (PIP)

For each polygon, the spatial index returns the candidate points from all overlapping grid cells. A vectorized ray-casting algorithm then tests all candidates simultaneously: for each polygon edge, a single NumPy operation determines which of the N candidate points have a horizontal ray crossing that edge, and the parity of crossing counts determines inside versus outside classification via the even-odd rule (Fig. 2). A bounding-box pre-filter rejects points outside the polygon extent before ray-casting begins. Park polygons often contain interior holes, such as lakes or other non-park areas. These are handled with a per-hole bounding-box pre-filter followed by a second ray-casting pass on the inner ring; any point found inside a hole is excluded from the result.

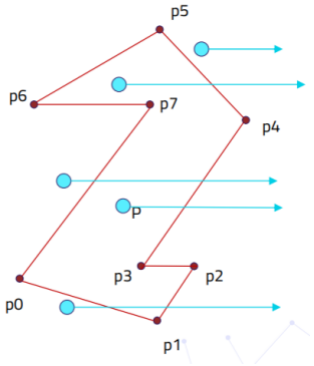


Figure 2. Concept visualization of ray-casting, used in the manually implemented point in polygon algorithm. Even-odd-rule: an even number (including 0) of ray-crossings means a point is outside a polygon; an odd number means a point is located inside (Image from lecture slides of GEO877).

1.3 Algorithm 3: Nearest Neighbor Index (NNI) using Haversine Distance

Before computing distances, coordinates are spatially deduplicated per polygon, as multiple Flickr photos frequently share identical GPS coordinates due to coordinate rounding. The NNI is computed as the ratio of the observed mean nearest-neighbor distance to the expected mean distance under complete spatial randomness. For polygons with more than 10,000 unique locations, a maximum of 10,000 query points are sampled randomly, while the search set and density calculation retain the full N, keeping the expected distance realistic. For each query point, Haversine distances (Fig. 3) to all unique locations are computed in a single vectorized NumPy operation and the minimum is recorded. Statistical significance is assessed with a Z-score using the Clark & Evans (1954) standard error.

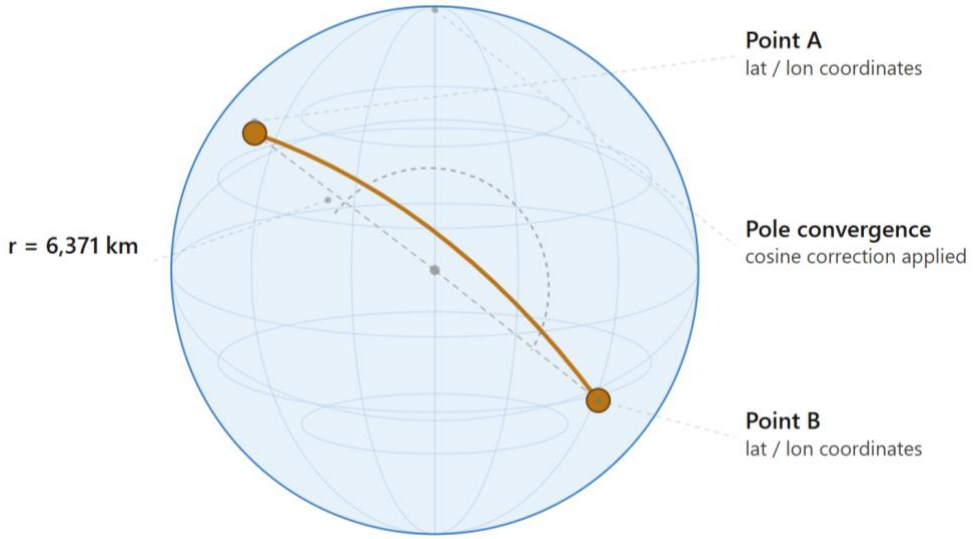


Figure 3. Illustration of haversine distance concept. The orange line represent the distance. The bigger it gets, the more the curvature of the earth has an influence (Image generated by ClaudeAI).

3. Data

The Flickr photo data used in this project consists of a pre-collected dataset of geotagged photos taken in Germany. The raw data is stored as a series of text files organized by the German federal state, each containing latitude and longitude coordinates alongside metadata such as the photo ID and timestamp. During preprocessing, all files were merged into a single dataset. Coordinate columns were parsed as numeric values, and any rows with missing or invalid coordinates were dropped. The final dataset contains all geotagged Flickr photos taken within the geographic extent of Germany.

For the area boundaries, two GeoJSON files were used. City polygons were sourced for the ten largest German cities by population, and national and nature park polygons were sourced for ten selected parks across Germany (Fig. 4). Both files were projected in WGS84 (EPSG:4326) for consistency.

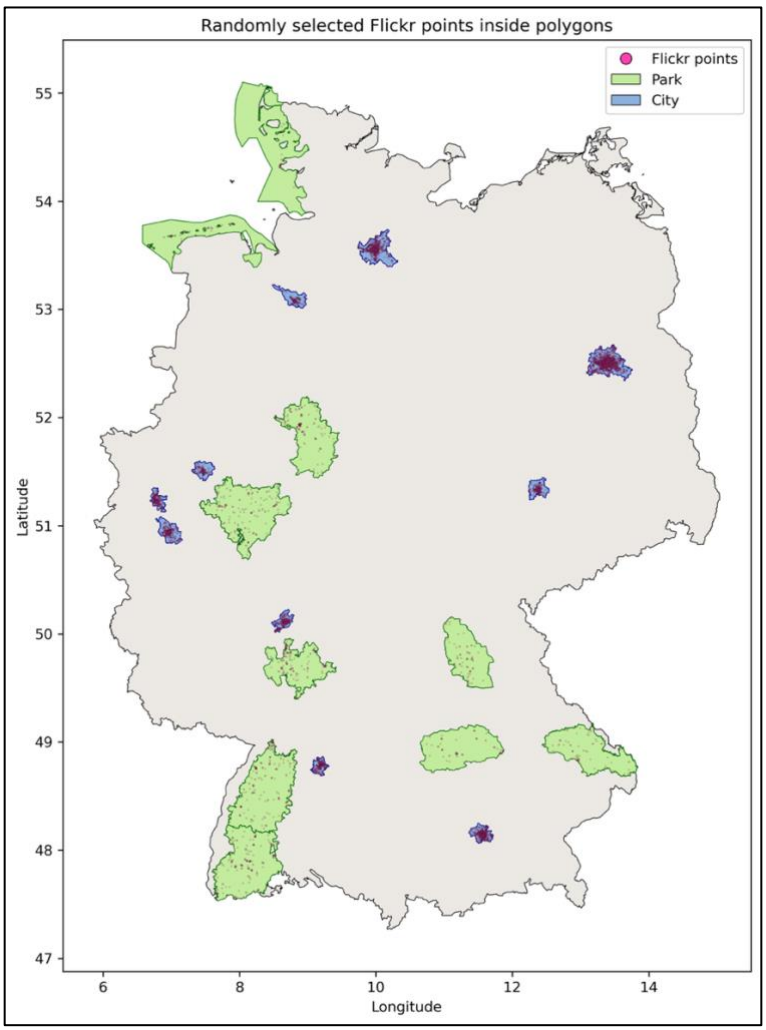


Figure 4. Map, showing Germany as a background and all ten polygons for cities, as well as parks, examined in this project. In addition, a limited number of randomly sampled points inside the specific polygons are shown for adding context (own depiction).

4. Results

The analysis examined the spatial distribution of Flickr photo data across ten German cities and nature parks respectively, using the Nearest Neighbor Index (NNI), measuring how clustered or dispersed a set of points is (Fig. 5).

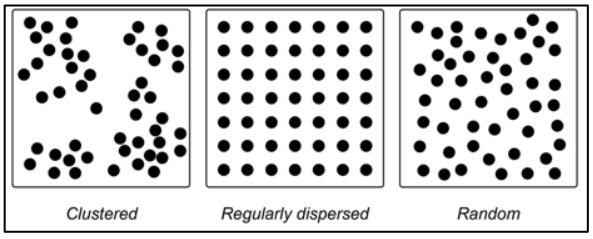


Figure 5. Illustration of different levels of point distribution in space: NNI < 1 is clustered, NNI = 1 is random, NNI > 1 is dispersed (Google pictures).

In total, the dataset contained 13,699,216 compiled Flickr points across Germany. After the bounding-box pre-filter, 3,509,122 candidate points remained (Fig. 6). The point-in-polygon algorithm then identified 2,829,125 points that fell inside the 20 polygons (Fig. 7).

```
Chunks: 10 | Rows reviewed: 5,000,000 | Candidates: 1,271,457
Chunks: 20 | Rows reviewed: 10,000,000 | Candidates: 2,546,997
Bounding-Box Pre-Filter completed: 13,699,216 -> 3,509,122 Candidates
Candidate points for the Algorithms: 3,509,122
PointIndex(res=0.05, nCols=146, nRows=152, points=3509122)
Spatial-Index-Test passed: 0 == 0
```

Figure 6. Output of the Spatial indexing and bounding-box pre-filter.

```
PIP finished. Saved results: 2,829,125
```

Figure 7. Number of points found inside polygons after PIP-Algorithm.

All 20 polygons, meaning both cities and parks, showed strongly clustered Flickr photo distributions. In the cities, the NNI values ranged from 0.280 (Köln) to 0.417 (Dortmund), with an average of 0.339 (Fig. 8). Berlin had the highest total number of photos and unique locations (1,086,095 photos, 463,217 unique locations) and an NNI of 0.352 (Fig. 9). Parks showed even stronger clustering, with NNI values ranging from 0.248 (Schleswig-Holsteinisches Wattenmeer) to 0.343 (Sauerland-Rothaargebirge), and an average of 0.302 (Fig. 8, 9). All Z-scores were far below −1.96, confirming that the clustering in every polygon is statistically significant and not a result of chance.

	polygon_type	polygons	unique_locations	mean_nni	median_nni	mean_observed_distance_m	mean_expected_distance_m
0	City	10	1047937	0.338569	0.343230	13.434340	39.518133
1	Park	10	88171	0.302217	0.305534	99.462327	338.045255

```
Average Sampled-Query NNI by City: 0.3386
Average Sampled-Query NNI by Parks:  0.3022
Conclusion: The Flickr points are stronger clustered in Parks!
```

Figure 8. Screenshot of an output in the Jupyter-Notebook, showing the Average NNI for cities and parks. With a NNI of almost 0.35, cities are less cluster than parks (0.3). Nonetheless it is worth noting that both areas show significant clustering. In addition, the observed mean distance is around 7x bigger in the parks compared to cities (own depiction).

```
City_Berlin: photos = 1,086,095 | unique locations = 463,217 | NNI = 0.3520 | Z-Score = -843.7
City_Bremen: photos = 52,322 | unique locations = 22,296 | NNI = 0.2965 | Z-Score = -201.0
City_Dortmund: photos = 48,377 | unique locations = 16,137 | NNI = 0.4166 | Z-Score = -141.8
City_Düsseldorf: photos = 114,413 | unique locations = 38,468 | NNI = 0.3582 | Z-Score = -240.8
City_Frankfurt am Main: photos = 205,459 | unique locations = 82,014 | NNI = 0.3524 | Z-Score = -354.8
City_Hamburg: photos = 358,183 | unique locations = 142,121 | NNI = 0.3360 | Z-Score = -478.9
City_Köln: photos = 213,116 | unique locations = 78,289 | NNI = 0.2804 | Z-Score = -385.2
City_Leipzig: photos = 93,681 | unique locations = 29,385 | NNI = 0.3071 | Z-Score = -227.2
City_München: photos = 338,808 | unique locations = 135,989 | NNI = 0.3425 | Z-Score = -463.8
City_Stuttgart: photos = 128,815 | unique locations = 40,021 | NNI = 0.3439 | Z-Score = -251.1
Park_Altmühltal: photos = 11,298 | unique locations = 5,159 | NNI = 0.2824 | Z-Score = -98.6
Park_Bayerischer Wald: photos = 9,356 | unique locations = 4,694 | NNI = 0.3102 | Z-Score = -90.4
Park_Bergstraße: photos = 26,189 | unique locations = 10,708 | NNI = 0.3365 | Z-Score = -131.4
Park_Fränkische Schweiz-Veldensteiner Forst: photos = 11,076 | unique locations = 5,635 | NNI = 0.3281 | Z-Score = -96.5
Park_Niedersächsisches Wattenmeer: photos = 6,318 | unique locations = 4,027 | NNI = 0.2605 | Z-Score = -89.8
Park_Sauerland-Rothaargebirge: photos = 25,739 | unique locations = 14,733 | NNI = 0.3426 | Z-Score = -152.7
Park_Schleswig-Holsteinisches Wattenmeer: photos = 4,923 | unique locations = 2,873 | NNI = 0.2480 | Z-Score = -77.1
Park_Schwarzwald Mitte/Nord: photos = 32,130 | unique locations = 15,903 | NNI = 0.3009 | Z-Score = -168.7
Park_Südschwarzwald: photos = 31,686 | unique locations = 15,013 | NNI = 0.3271 | Z-Score = -157.7
Park_Teutoburger Wald/Eggegebirge: photos = 31,141 | unique locations = 9,426 | NNI = 0.2860 | Z-Score = -132.6
```

Figure 9. Detailed summary of photo numbers, unique locations, NNI and z-score for every polygon (cities and parks), for the manually implemented algorithms.

The two NNI methods used in the notebook produced consistent results. The sampled-query NNI, which used up to 10,000 query points per polygon but searched among all unique locations, gave an average of 0.339 for cities and 0.302 for parks (Fig. 8). The agreement scatter plot showed that both methods produced nearly identical values across all polygons, which validates our manual implementation (Fig. 10). The library validation using sklearn's BallTree on all unique locations confirmed these numbers (Fig. 11). The NNI values of the validation method are further shown on a map in Figure 14.

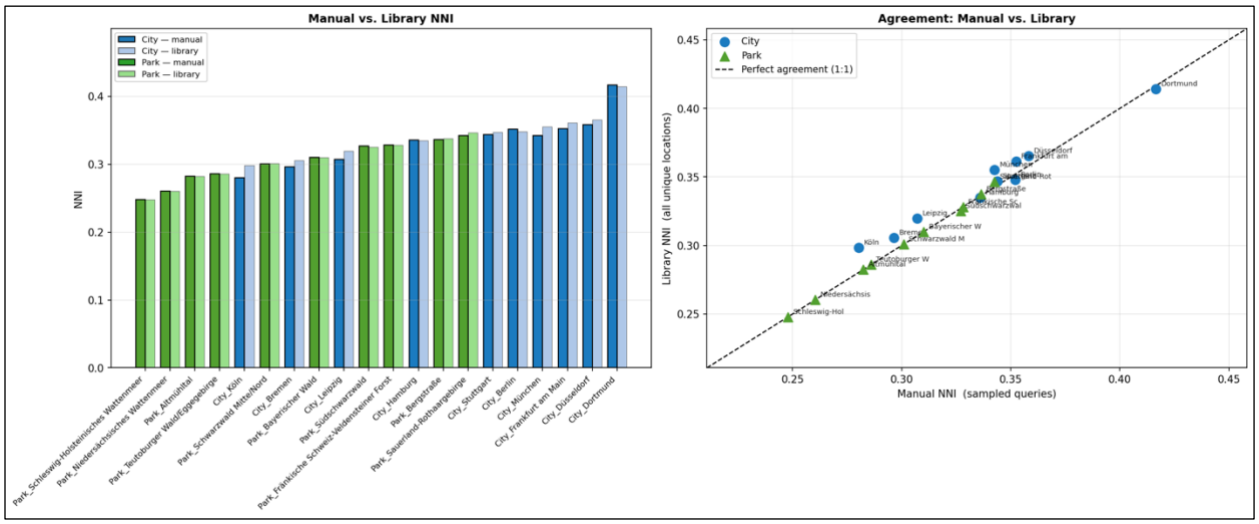


Figure 10. Bar and agreement plot, comparing the results of the manual algorithm implementation with the results of the validation, where modern geospatial libraries were used. The manual algorithm implementation is confirmed (own depiction).

```
Calculate validation NNI (BallTree, all unique coordinates)...
City_Berlin                                  unique = 463,217  NNI = 0.3479  Z =  -849.1  [clustered]
City_Bremen                                  unique =  22,296  NNI = 0.3056  Z =  -198.3  [clustered]
City_Dortmund                                unique =  16,137  NNI = 0.4142  Z =  -142.4  [clustered]
City_Düsseldorf                              unique =  38,468  NNI = 0.3653  Z =  -238.2  [clustered]
City_Frankfurt am Main                       unique =  82,014  NNI = 0.3614  Z =  -349.9  [clustered]
City_Hamburg                                 unique = 142,121  NNI = 0.3351  Z =  -479.5  [clustered]
City_Köln                                    unique =  78,289  NNI = 0.2983  Z =  -375.6  [clustered]
City_Leipzig                                 unique =  29,385  NNI = 0.3199  Z =  -223.0  [clustered]
City_München                                 unique = 135,989  NNI = 0.3552  Z =  -454.9  [clustered]
City_Stuttgart                               unique =  40,021  NNI = 0.3471  Z =  -249.9  [clustered]
Park_Altmühltal                              unique =   5,159  NNI = 0.2824  Z =   -98.6  [clustered]
Park_Bayerischer Wald                        unique =   4,694  NNI = 0.3102  Z =   -90.4  [clustered]
Park_Bergstraße                              unique =  10,708  NNI = 0.3375  Z =  -131.1  [clustered]
Park_Fränkische Schweiz-Veldensteiner Forst  unique =   5,635  NNI = 0.3281  Z =   -96.5  [clustered]
Park_Niedersächsisches Wattenmeer            unique =   4,027  NNI = 0.2605  Z =   -89.8  [clustered]
Park_Sauerland-Rothaargebirge                unique =  14,733  NNI = 0.3465  Z =  -151.7  [clustered]
Park_Schleswig-Holsteinisches Wattenmeer     unique =   2,873  NNI = 0.2480  Z =   -77.1  [clustered]
Park_Schwarzwald Mitte/Nord                  unique =  15,903  NNI = 0.3011  Z =  -168.6  [clustered]
Park_Südschwarzwald                          unique =  15,013  NNI = 0.3252  Z =  -158.2  [clustered]
Park_Teutoburger Wald/Eggegebirge            unique =   9,426  NNI = 0.2860  Z =  -132.6  [clustered]
```

Figure 11. Detailed summary of unique locations, NNI and z-score for every polygon (cities and parks), for the validation.

Given that we identified clustering for both parks and cities, we wanted to measure in which scale the clustering was present and if parks and cities differed in this regard. Thus, we plotted the density of points using kernel density estimation as a function of nearest-neighbor distance. The peak density for cities is around 2.7 meters distance between nearest neighbors and decays rapidly with increasing distance. For parks, the peak density was around 13.0 meters distance and showed a longer tail to the right (Fig. 12). Potential reasons for these differences are further discussed in the discussion. The big difference in this trend is contrary to the different, but nonetheless quite close NNI-values between cities and parks (Fig. 13).

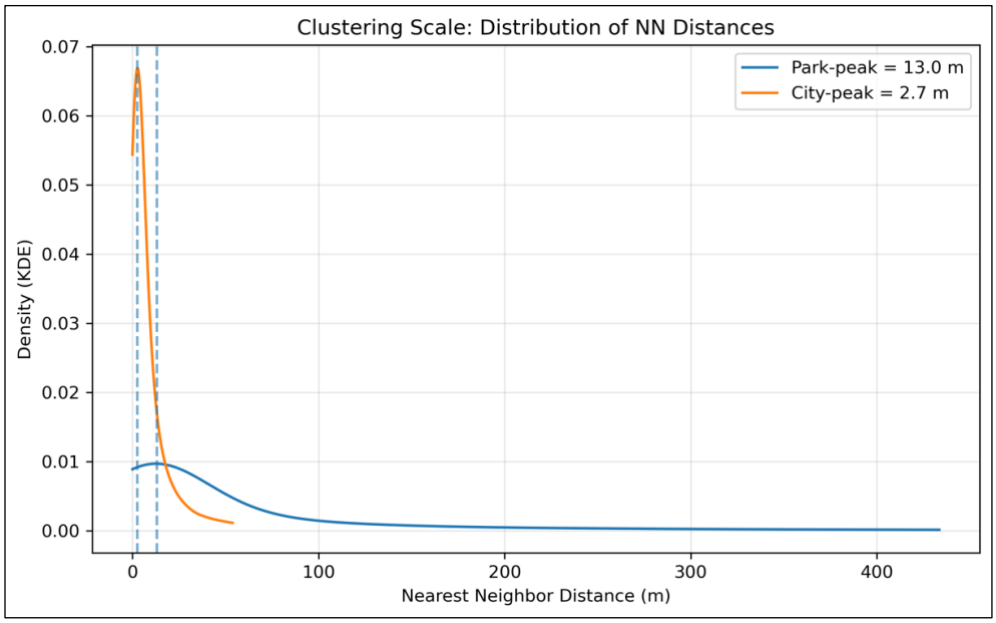


Figure 12. Plot showing the distribution of nearest-neighbor distances and their density. The peak for parks (orange) is significantly higher than for cities (blue) (own depiction).

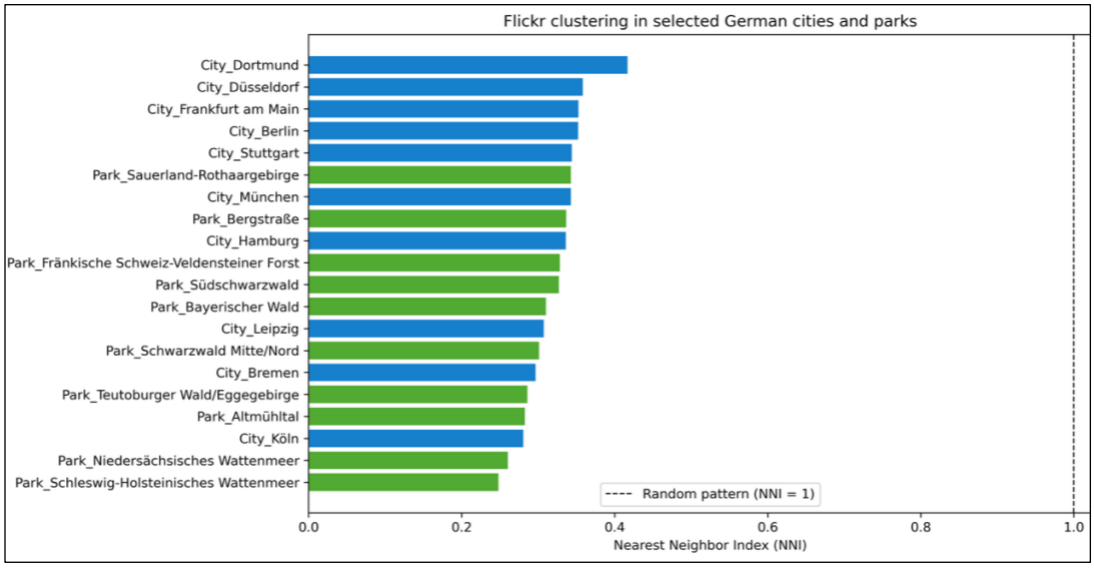


Figure 13. Vertical bar plot, showing the NNI for all polygons. Green are parks, and blue are the cities. Overall, it shows that cities have a slightly higher NNI, therefore are a little bit less clustered than parks (own depiction).

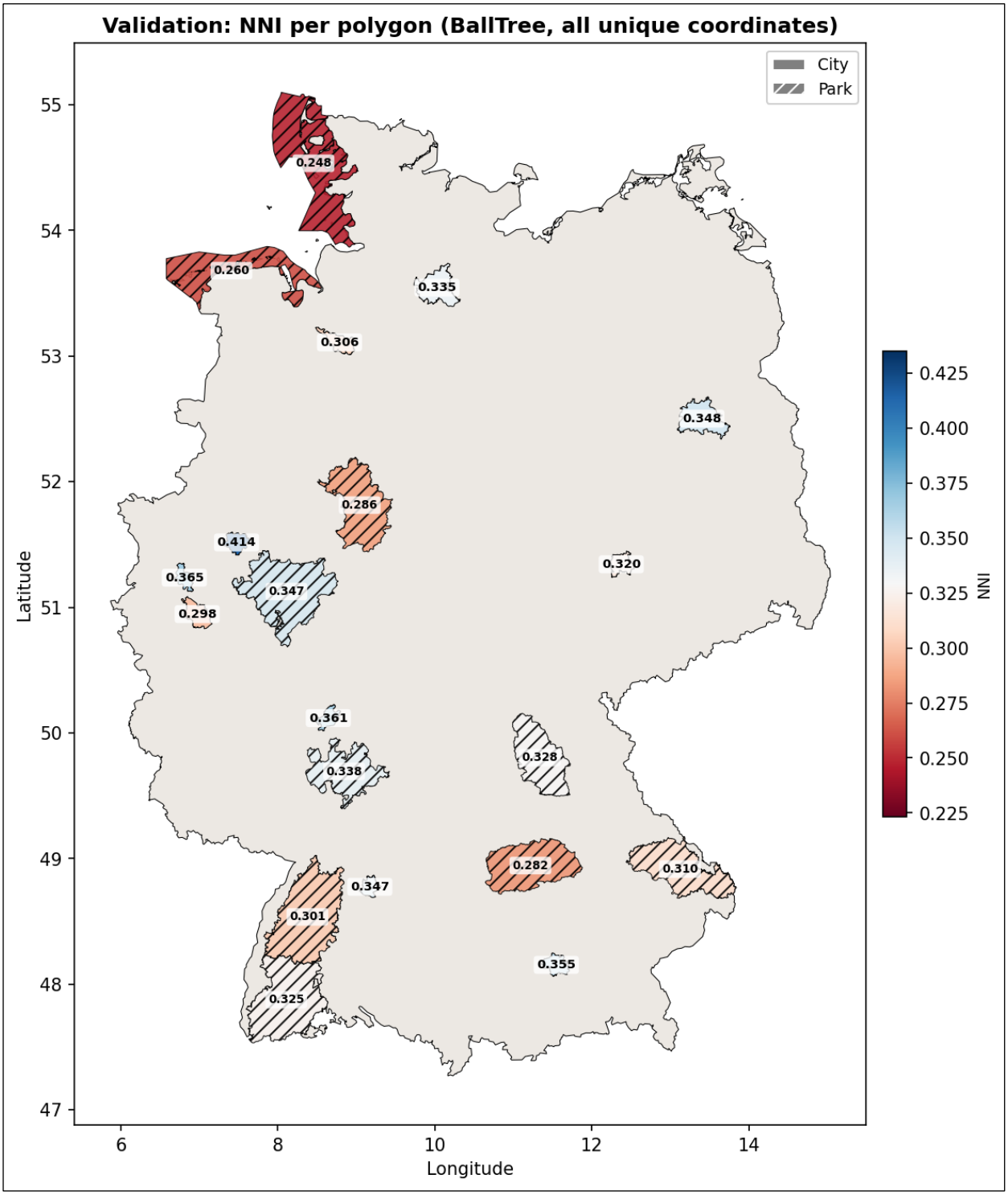


Figure 14. Map, showing the NNI for parks and cities, calculated using modern geospatial libraries for validation (own depiction).

Note: Minor numerical differences between some reported values/figures and the newest version of the notebook may occur due to the random sampling applied in the NNI computation. All substantive findings and conclusions are unaffected by these differences.

5. Discussion

The study was guided by two working hypotheses. H1 predicted that Flickr photos would be more clustered in cities than in parks, and H2 predicted that mean distances between photos would be shorter in cities than in parks.

H1 could not be confirmed. The results showed the opposite: parks had a slightly lower average NNI (0.302) than cities (0.339), meaning that clustering was actually stronger in parks. A lower NNI value indicates that observed distances between nearest neighbors are shorter relative to what a random pattern would predict, which means more concentrated hotspot behavior. While photos in cities are also strongly clustered, the relative degree of clustering was consistently higher across all ten parks. H1 is thus rejected.

H2, however, could clearly be confirmed. The mean observed nearest-neighbor distances were much shorter in cities than in parks. On average, the observed mean distance in cities was around 13.4 meters, compared to roughly 99.5 meters in parks. This makes intuitive sense: cities are densely built environments where photos are taken close together in space, while parks are large areas where even clustered photos can still be hundreds of meters apart in absolute terms. So, photos in cities are physically closer to each other, even though they are not more clustered in a relative sense.

This distinction between H1 and H2 is the most important insight of the analysis. It reveals that absolute distance and relative clustering are two different things and can point in opposite directions. Cities have shorter absolute distances between photos because the entire area is densely photographed. Parks have stronger relative clustering because photo activity is concentrated into a few specific spots within a very large area, while most of the park area remains essentially empty. The two Wattenmeer parks showed the most extreme clustering of all polygons with NNI values of 0.248 and 0.260, yet their mean observed distances of around 107 and 153 meters were far larger than any city value.

This finding has a straightforward geographic explanation. Urban areas have a high density of attractions, infrastructure, and everyday activity spread across the whole city, which produces dense but relatively more evenly distributed photo coverage. Nature parks, by contrast, funnel visitors to specific accessible and scenic locations, such as viewpoints, trailheads, lakes, or coastal access points, while leaving the rest of the park largely unvisited and unphotographed. This creates a pattern of extreme local hotspots against a very sparse background, which the NNI captures as stronger clustering.

This analysis has also limitations. It is worth noting that Flickr data reflects the behavior of a specific and non-representative user group. Flickr photographers tend to be tourism-oriented and technically engaged, which likely amplifies hotspot behavior at well-known scenic locations, particularly in parks. The spatial deduplication step in the analysis was important here, as many photos shared identical coordinates due to GPS rounding or metadata copying. Without removing these stacked duplicates, the NNI would have been artificially deflated. The close agreement between the manual NNI results and the sklearn BallTree validation across all 20 polygons confirms that the methodology was sound and the conclusions are reliable.

6. Conclusion

This project investigated the spatial clustering of Flickr photo data across ten German cities and ten nature parks respectively, using 3 manually implemented spatial algorithms. The results show that both environment types exhibit statistically significant clustering, with all Z-scores far below the significance threshold (-1.96). In contrast to our initial hypothesis (H1), photos in parks are more clustered (average NNI = 0.302) than in cities (average NNI = 0.339). H2 however, was confirmed. The absolute mean nearest-neighbor distances in cities (13.4 m) are substantially shorter than in parks (99.5 m).

The main finding is that relative clustering and absolute distance are distinct and can point in opposite directions. Cities produce dense, spatially more continuous photo coverage across a large built-up area,

while parks concentrate visitor activity, and with that also the photo-taking activity, into a small number of accessible hotspots within a vast, largely unphotographed landscape. The KDE analysis of nearest-neighbor distances reinforces this. The peak clustering scale in cities (2.7m) is dominated by street-level and landmark-level photography, while in parks the peak (13.0m) reflects concentrated but more spatially spread visitor spots.

From a technical perspective, the project demonstrates that robust spatial analysis of large, geotagged datasets is achievable through manually implemented algorithms. The grid-based spatial index vectorized ray-casting PIP, and Haversine-based sampled-query NNI together reduced processing time substantially, without sacrificing correctness. The close agreement between the manual NNI results and the scikit-learn BallTree validation across all 20 polygons confirms the reliability of the manual implementation. Despite that, it has to be said that for more complex studies and a larger extent, modern libraries are most likely irreplaceable. Also, while the use of the haversine distance might be technically the more accurate method, e.g. than using the simpler Euclidean distance, for a country like Germany it might not have been necessary.

Following our findings, future work could extend this analysis to additional federal states, incorporate temporal dimensions (e.g. seasonal clustering shifts), or apply the same methods to other VGI sources such as OpenStreetMap edits or Instagram locations. Differentiating between tourist and resident photography behavior, for example using temporal metadata, would also allow a more nuanced interpretation of the clustering patterns observed in this project.

Credits

Table 1. Distribution of project work.

Task	Distribution
Conceptualization	All
Methodology	All
Data curation and preprocessing	Javier and Max
Literature review	Pascal and Till
Formal analysis	All
Code implementation	Max and Javier
Investigation	All
Writing report	All

AI statement

LLMs were used to improve grammar as well as creating and debugging code in this project, especially for content and challenges that go beyond the scope of the course. All content, analysis, and arguments remain the author's own, and any information derived from external sources is properly cited.

Links

- GitHub: https://github.com/mlengenfelder/GEO877_Spatial_Algorithms.git
- Interactive map: https://mlengenfelder.github.io/GEO877_Spatial_Algorithms/

References

Clark, P. J. and Evans, F. C. 1954. Distance to nearest neighbor as a measure of spatial relationships in populations. Ecology, 35(4), pp. 445–453.

Crandall, D. J., Backstrom, L., Huttenlocher, D. and Kleinberg, J. 2009. Mapping the world's photos. In Proceedings of the 18th International Conference on World Wide Web. Madrid, Spain, 20–24 April 2009, pp. 761–770.

Croitoru, A., Crooks, A. T., Radzikowski, J. and Stefanidis, A. 2013. GeoSocial Gauge: A system prototype for knowledge discovery from social media. International Journal of Geographical Information Science, 27(12), pp. 2483–2508.

Goodchild, M. F. 2007. Citizens as sensors: The world of volunteered geography. GeoJournal, 69(4), pp. 211–221.

Hollenstein, L. and Purves, R. S. 2010. Exploring place through user-generated content: Using Flickr tags to describe city cores. Journal of Spatial Information Science, 1(1), pp. 21–48.

Kádár, B. 2014. Measuring tourist activities in cities using geotagged photography. Tourism Geographies, 16(1), pp. 88–104.

Kisilevich, S., Krstajić, M., Keim, D., Andrienko, N. and Andrienko, G. 2010. Event-based analysis of people's activities and behavior using Flickr and Panoramio geotagged photo collections. In 2010 14th International Conference Information Visualisation. London, 26–29 July 2010, pp. 289–296.

Li, L., Goodchild, M. F. and Xu, B. 2013. Spatial, temporal, and socioeconomic patterns in the use of Twitter and Flickr. Cartography and Geographic Information Science, 40(2), pp. 61–77.

Sinclair, M., Mayer, M., Woltering, M. and Ghermandi, A. 2020. Using social media to estimate visitor provenance and patterns of recreation in Germany's national parks. Journal of Environmental Management, 263, p. 110418.

Straumann, R. K., Çöltekin, A. and Andrienko, G. 2014. Towards reconstructing narratives from georeferenced photographs through visual analytics. The Cartographic Journal, 51(2), pp. 152–165.

Thomee, B., Shamma, D. A., Friedland, G., Elizalde, B., Ni, K., Poland, D., Borth, D. and Li, L. J. 2016. YFCC100M: The new data in multimedia research. Communications of the ACM, 59(2), pp. 64–73.

Detailed description of the 3 key-algorithms

1. Spatial Indexing

Spatial Indexing is used to partition space into smaller compartments with the idea of storing spatially close objects similarly close in the data structure. Thus, enabling the retrieval of information based on location. This allows for efficient two-step query processing of big data repositories. Assigning a spatial index to objects can be used to pre-select candidates for further, more computationally demanding, spatial algorithms, thereby reducing the number of necessary calculations to answer a query. Spatial Indexing can be utilized with different tessellations and can be static or hierarchical.

Given that the Flickr-Dataset contains millions of records it is beneficial to spatially index these records before running exact geometric queries as to increase the performance of those. The Spatial index was built using a grid-based approach, dividing the examined space into regular cells (5.5km) and assigning each record a cell position based on its longitude and latitude coordinates. The cells store references to the information of all points within their cell position and can be called later in further queries.

Implementation of this spatial indexing has been programmed along the following steps. First, the bounding boxes (BBox) of the individual park and city polygons were created. Then, utilizing their latitude and longitude columns, chunks of records within the csv file were analyzed based on their spatial relation to all polygon bounding boxes. Although similarly, this is a faster process than a point in polygon algorithm, as only 4 comparisons per point-polygon pair are necessary. All records within any of the polygon BBox were stored as candidates and subsequently converted to point objects. Based on the min-max coordinates within the candidate's dataset a BBox was created. This BBox was then partitioned into cells with a resolution of 5.5km, suitable for the scale of the parks and cities. Each candidate was assigned to a point index, stored in a one-dimensional array.

Further algorithms processing polygons now utilize the Spatial Index to query only cells intersecting the BB of the polygon. All Candidate points stored within the intersecting cells can then be queried in the point in polygon algorithm.

2. Point in Polygon

The points in polygon algorithm can be most effectively done using the ray casting algorithm. The ray casting algorithm projects a ray in one direction extending indefinitely. Ray casting, using the even-odd rule, counts the number of intersections between the ray and polygon edges. If the number of intersections is odd, the point is within the polygon, else outside.

Using the before built spatial index, processing a city or park polygon queries only the intersecting cells and their stored information about candidates. A few optimizations have been made to make the algorithm more efficient. First, a quick rejection test for all points outside the BBox of the polygon. Further, polygon edges get skipped if both their edge endpoints are to the left of the point or they are parallel to the ray. Finally, vertices are handled through boundary logic so as not to count them twice. Applying the ray casting algorithm, to the right, then classifies candidates on whether they are inside or outside of the polygon

Because the parks polygons contained holes, the point in polygon algorithm was extended for hole detection . Ray casting was applied to the outer ring of a polygon. All candidates with odd counts were then tested against inner rings separately. If a point fell within any inner ring it was rejected.

3. Nearest Neighbor Index (NNI) & Haversine Distance

With the aim to analyze the candidates based on their clustering pattern within individual polygons the distance between neighboring points had to be calculated. The Haversine Distance was chosen as the method of calculation. The haversine distance accounts for the curvature of the earth and is thus a good approximation. The Haversine Distance is calculated by converting the coordinates of 2 points to radians and calculating the latitude and longitude angular difference. The Haversine value, between 0 and 1, can then be reconverted to distance, using arcsin and the earths radius, using the following formula:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

where:

- *d is the computed geodesic distance between two points on the Earth's surface (in meters),*
- *r is the mean radius of the Earth (defined as approximately 6,371,000 meters),*
- *φ_1 (phi 1) and φ_2 (phi 2) represent the geographic latitudes of the first and second points, respectively, converted into radians,*
- *λ_1 (lambda 1) and λ_2 (lambda 2) represent the geographic longitudes of the first and second points, respectively, converted into radians.*

Before any distance computation, coordinates are spatially deduplicated per polygon. Multiple Flickr photos frequently share identical GPS coordinates due to Flickr's coordinate rounding precision (approximately 0.1 m). Retaining stacked points would artificially reduce observed nearest-neighbor distances and inflate the apparent degree of clustering. Only unique coordinate pairs are retained.

The NNI compares the observed mean nearest-neighbor distance to the expected mean distance under complete spatial randomness (Clark & Evans, 1954). The expected distance is derived from the point density

$$\rho = \frac{N}{A}$$

where:

- *ρ is the spatial point density, representing the number of unique photo observations per unit area,*
- *N is the total count of unique, spatially deduplicated photo locations within the boundary polygon,*
- *A is the total geographical area of the study polygon (expressed in square meters).*

$$\bar{d}_{exp} = \frac{1}{2\sqrt{\rho}} = 0.5\sqrt{\frac{A}{N}}$$

where:

- *$\bar{d}_{exp}$ is the expected mean distance between nearest neighbors under the assumption of Complete Spatial Randomness (CSR),*
- *ρ (rho) is the spatial point density (N/A),*
- *A is the polygon area in square meters calculated via the local UTM coordinate reference system (EPSG:25832),*
- *N is the total count of unique photo locations.*

The NNI is then

$$NNI = \frac{\bar{d}_{obs}}{\bar{d}_{exp}} = \frac{\bar{d}_{obs}}{0.5\sqrt{A/N}}$$

where:

- *NNI is the Nearest Neighbor Index value (where NNI < 1 implies a clustered pattern, NNI = 1 implies a random pattern, and NNI > 1 implies a dispersed pattern),*
- *$\bar{d}_{obs}$ is the observed mean nearest-neighbor distance derived from the dataset,*
- *$\bar{d}_{exp}$ is the expected mean nearest-neighbor distance under complete spatial randomness,*
- *A is the area of the polygon in square meters,*
- *N is the total number of unique, deduplicated photo points.*

A value below 1 indicates clustering, above 1 dispersion.

For polygons with more than 5,000 unique locations, a sampled-query approach is used: up to 5,000 query points are selected at random, but the search set and the density formula use the full N. This bounds computation time while keeping the expected distance realistic. For each query point, the Haversine distance to all N unique locations is computed in a single vectorized NumPy operation, replacing the original Python per-candidate loop. The minimum distance across all N values is recorded, and the mean of these minima gives the observed mean distance $\bar{d}_{obs}$.

Statistical significance is assessed using the Clark & Evans (1954) standard error

$$SE = \frac{0.26136}{\sqrt{N^2/A}}$$

where:

- *SE is the standard error of the expected nearest-neighbor distance, describing the statistical variance expected under complete spatial randomness,*
- 0.26136 *is a mathematical constant derived from the theoretical sampling distribution of nearest-neighbor distances in a two-dimensional Poisson process,*
- *N is the total number of unique photo locations within the polygon,*
- *A is the polygon area in square meters.*

Giving a Z-score

$$Z = \frac{\bar{d}_{obs} - \bar{d}_{exp}}{SE}$$

where:

- *Z is the standard score (or critical value) testing the deviation from complete spatial randomness,*
- *$\bar{d}_{obs}$ is the observed mean nearest-neighbor distance,*
- *$\bar{d}_{exp}$ is the expected mean nearest-neighbor distance under complete spatial randomness,*
- *SE is the Clark & Evans standard error of the expected distance baseline.*

Polygons with $Z < -1.96$ are classified as significantly clustered at the 5 % level. Results are cross-validated using scikit-learn's BallTree with the haversine metric applied to all unique locations without sampling, confirming the correctness of the manual implementation.

References

Clark, P. J. and Evans, F. C. 1954. Distance to nearest neighbor as a measure of spatial relationships in populations. Ecology, 35(4), pp. 445–453.

1. Setup

GeoPandas is used only for reading GeoJSON files, CRS transformation, area calculation, and plotting. The spatial algorithms themselves are implemented manually!

Run the following Cell once, to create the folders for the data!

In [1]:
```
import csv
import glob
import math
import os
import random
import sys
from collections import defaultdict
from pathlib import Path

import duckdb
import geopandas as gpd
import numpy as np
import pandas as pd
from numpy import arcsin, cos, radians, sin, sqrt

from scipy.stats import gaussian_kde
from sklearn.neighbors import BallTree

import branca.colormap as cm
import folium
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
from folium.plugins import MarkerCluster
from matplotlib.cm import ScalarMappable
from matplotlib.colors import Normalize
from matplotlib.lines import Line2D


ROOT_DIR = Path.cwd()

if str(ROOT_DIR) not in sys.path:
    sys.path.append(str(ROOT_DIR))

# Folder Paths
PATH_DATA = ROOT_DIR / 'data'
PATH_RAW = PATH_DATA / 'raw'
PATH_PROCESSED = PATH_DATA / 'processed'
PATH_OUTPUTS = ROOT_DIR / 'outputs'

for p in [PATH_RAW, PATH_PROCESSED, PATH_OUTPUTS]:
    p.mkdir(parents=True, exist_ok=True)

# Input File Paths
```

```
DATA_BASE_DIR = PATH_RAW / "DE"
GEOJSON_PARKS = PATH_RAW / "park_polygons.geojson"
GEOJSON_CITIES = PATH_RAW / "city_polygons.geojson"
GEOJSON_GERMANY = PATH_RAW / "germany_base.geojson"

# Output File Paths
COMPILED_CSV = PATH_OUTPUTS / "all_flickr_points.csv"
CANDIDATE_CSV = PATH_OUTPUTS / "flickr_bbox_candidates.csv"
FILTERED_CSV = PATH_OUTPUTS / "filtered_flickr_inside_polygons.csv"
NNI_RESULTS_CSV = PATH_OUTPUTS / "nni_results_sampled.csv"
NNI_QUERY_RESULTS_CSV = PATH_OUTPUTS / "nni_results_sampled_query.c

# Sample-Configuration
USE_DATA_SAMPLE = False
SAMPLE_MAX_FILES = 12

# Cache switch: Set to True if outputs are to be rebuilt
REBUILD_COMPILED_CACHE = False
REBUILD_CANDIDATE_CACHE = False
REBUILD_PIP_RESULTS = False
REBUILD_NNI_RESULTS = False

# Spatial-index & NNI Settings
GLOBAL_INDEX_RESOLUTION = 0.05
NNI_SAMPLE_SIZE = 10000
RANDOM_SEED = 42
CSV_CHUNK_SIZE = 500000

# Prevent DtypeWarnings warnings:
MIXED_TYPE_DTYPES = {"ID": "string", "PhotoID": "string", "Views": '

os.makedirs("./outputs", exist_ok=True)

print("Setup ready!")
```

```
Setup ready!
```

2. Data Preparation

The Flickr data are loaded recursively from `./data/raw/DE`. All `settings.txt` files are excluded. Coordinates are cleaned, photos are deduplicated by `PhotoID`, and the cleaned point table is cached in `./outputs/all_flickr_points.csv`.

Some descriptive fields, especially `PhotoID`, `Views`, and `MTags`, can look numeric in some rows and textual or empty in others. These fields are therefore read as strings in later cached reads. This avoids Pandas `DtypeWarning` messages and does not affect the spatial algorithms, because only `Latitude` and `Longitude` are used as coordinates.

In [2]:
```
FLICKR_COLUMNS = [
    "ID", "Latitude", "Longitude", "NAME", "URL", "PhotoID", "Owner'
```

```
    "DateTaken", "UploadDate", "Views", "Tags", "MTags"
]


def find_flickr_files(base_dir):
    all_txt_files = sorted(glob.glob(os.path.join(base_dir, "**", "
    data_files = [f for f in all_txt_files if os.path.basename(f).l
    settings_files = [f for f in all_txt_files if os.path.basename(
    return data_files, settings_files


def validate_headers(files):
    header_counts = defaultdict(int)
    for file_path in files:
        with open(file_path, "r", encoding="utf-8", errors="replace
            reader = csv.reader(f)
            header = tuple(next(reader))
            header_counts[header] += 1

    print(f"Flickr data files found: {len(files)}")
    print(f"settings.txt Files excluded: {len(settings_files)}")
    print(f"Header-Variants found: {len(header_counts)}")

    for header, count in header_counts.items():
        print(f"{count} Files with header: {header}")
        if list(header) != FLICKR_COLUMNS:
            raise ValueError("Unexpected header in Flickr-Files.")


flickr_files, settings_files = find_flickr_files(DATA_BASE_DIR)
validate_headers(flickr_files)
```

```
Flickr data files found: 403
settings.txt Files excluded: 90
Header-Variants found: 1
403 Files with header: ('ID', 'Latitude', 'Longitude', 'NAME', 'UR
L', 'PhotoID', 'Owner', 'UserID', 'DateTaken', 'UploadDate', 'View
s', 'Tags', 'MTags')
```

In [3]:

```
def compile_raw_flickr_data_duckdb(files, output_csv):
    if os.path.exists(output_csv) and not REBUILD_COMPILED_CACHE:
        print(f"Load already compiled Data from {output_csv}...")
        return output_csv

    files_to_read = files[:SAMPLE_MAX_FILES] if USE_DATA_SAMPLE else
    print(f"Starting DuckDB Engine... Processing {len(files_to_read

    con = duckdb.connect()

    keep_cols = "PhotoID, Latitude, Longitude, UserID, DateTaken"

    query = f"""
    COPY (
        SELECT DISTINCT ON (PhotoID)
            {keep_cols}
        FROM read_csv(
            ?,
```

```
            delim=',',
            header=true,
            all_varchar=true,
            ignore_errors=true
        )
        WHERE TRY_CAST(Latitude AS DOUBLE) BETWEEN -90 AND 90
          AND TRY_CAST(Longitude AS DOUBLE) BETWEEN -180 AND 180
    ) TO '{output_csv}' (HEADER, DELIMITER ',');
    """

    con.execute(query, [files_to_read])

    print(f"DuckDB compiling finished! Master dataset saved to: {ou
    return output_csv

REBUILD_COMPILED_CACHE = True

compile_raw_flickr_data_duckdb(flickr_files, COMPILED_CSV)

pd.read_csv(COMPILED_CSV, nrows=5)
```

```
Starting DuckDB Engine... Processing 403 files.
FloatProgress(value=0.0, layout=Layout(width='auto'), style=Progress
Style(bar_color='black'))
DuckDB compiling finished! Master dataset saved to: /Users/maxlengen
felder/Desktop/FS26/GEO877/Project/GEO877_Flickr_Project/outputs/all
_flickr_points.csv
```

Out[3]:

	PhotoID	Latitude	Longitude	UserID	DateTaken
0	14944465206	48.581944	7.749620	24415314@N02	8/15/2014 14:07:05
1	14731893217	48.581930	7.750650	56346723@N06	6/10/2014 14:34:33
2	14900251493	48.575400	7.754180	84070589@N04	7/14/2014 22:41:36
3	14877912904	48.575400	7.754180	84070589@N04	7/14/2014 22:35:49
4	14876572115	48.582933	7.743749	36536432@N00	7/25/2014 5:53:15

In [4]:

```
def load_polygon_data(city_path, park_path):
    polygon_records = []

    layers = [
        ("City", city_path, "Geografisc"),
        ("Park", park_path, "NAME"),
    ]

    for poly_type, path, name_field in layers:
        gdf_original = gpd.read_file(path)
        print(f"{poly_type} CRS before: {gdf_original.crs}")

        gdf_lonlat = gdf_original.to_crs(epsg=4326)
```

```
        gdf_metric = gdf_original.to_crs(epsg=25832)

        print(f"{poly_type} Bounds after: EPSG:4326: {tuple(round(v

        for idx, row in gdf_lonlat.iterrows():
            geom = row.geometry
            base_name = row[name_field] if name_field in row and pd
            poly_name = f"{poly_type}_{base_name}"
            area_m2 = float(gdf_metric.geometry.iloc[idx].area)

            if geom.geom_type == "Polygon":
                parts = [geom]
            elif geom.geom_type == "MultiPolygon":
                parts = list(geom.geoms)
            else:
                continue

            for part_id, part in enumerate(parts):
                exterior = [(float(x), float(y)) for x, y in part.e
                holes = [[(float(x), float(y)) for x, y in ring.coo
                polygon_records.append({
                    "name": poly_name,
                    "type": poly_type,
                    "part_id": part_id,
                    "exterior": exterior,
                    "holes": holes,
                    "area_m2": area_m2,
                    "geometry": part,
                })

    return polygon_records

polygon_records = load_polygon_data(GEOJSON_CITIES, GEOJSON_PARKS)
print(f"Polygon-objects prepared: {len(polygon_records)}")
print(f"Polygons with holes: {sum(1 for p in polygon_records if p['
```

```
City CRS before: EPSG:25832
City Bounds after: EPSG:4326: (np.float64(6.68982), np.float64(48.06
155), np.float64(13.76047), np.float64(53.73847))
Park CRS before: EPSG:4326
Park Bounds after: EPSG:4326: (np.float64(6.58083), np.float64(47.53
24), np.float64(13.83964), np.float64(55.09917))
Polygon-objects prepared: 20
Polygons with holes: 4
```

3. Spatial Algorithms

Using classes for the three manual algorithms. The code follows the logics like: point distances, bounding boxes, line/segment logic, point-in-polygon, and grid-based point indexing.

In [5]:
```
class Point():
    def __init__(self, x=None, y=None, pid=None, attrs=None):
        self.x = float(x)
```

```
        self.y = float(y)
        self.id = pid
        self.attributes = attrs or {}

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

    def __eq__(self, other):
        if not isinstance(other, Point):
            return NotImplemented
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y))

    def distEuclidean(self, other):
        return sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

    def distHaversine(self, other):
        r = 6371000
        phi1 = radians(self.y)
        phi2 = radians(other.y)
        lam1 = radians(self.x)
        lam2 = radians(other.x)

        d = 2 * r * arcsin(sqrt(
            sin((phi2 - phi1) / 2)**2 +
            cos(phi1) * cos(phi2) * sin((lam2 - lam1) / 2)**2
        ))
        return float(d)

    def sideLine(self, p1, p2):
        side = int((p2.x - p1.x) * (self.y - p1.y) - (self.x - p1.x
        if side != 0:
            side = side / abs(side)
        return side


class Bbox():
    def __init__(self, data):
        if isinstance(data, Segment):
            x = [data.start.x, data.end.x]
            y = [data.start.y, data.end.y]
        else:
            x = [p.x for p in data]
            y = [p.y for p in data]

        self.ll = Point(min(x), min(y))
        self.ur = Point(max(x), max(y))
        self.ctr = Point((min(x) + max(x)) / 2, (min(y) + max(y)) /
        self.area = abs(max(x) - min(x)) * abs(max(y) - min(y))

    def __repr__(self):
        return f"Bounding box with lower-left {self.ll} and upper-r

    def testOverlap(self, other):
```

```
        if (self.ur.x >= other.ll.x and other.ur.x >= self.ll.x and
            self.ur.y >= other.ll.y and other.ur.y >= self.ll.y):
            return True
        return False

    def containsPoint(self, p):
        if (self.ur.x >= p.x >= self.ll.x and self.ur.y >= p.y >= s
            return True
        return False

    def intersectsRegion(self, other):
        if not self.testOverlap(other):
            return None
        llx = max(self.ll.x, other.ll.x)
        lly = max(self.ll.y, other.ll.y)
        urx = min(self.ur.x, other.ur.x)
        ury = min(self.ur.y, other.ur.y)
        return Bbox([Point(llx, lly), Point(urx, ury)])


class Segment():
    def __init__(self, p0, p1, sid=None):
        self.start = p0
        self.end = p1
        self.sid = sid
        self.length = p0.distEuclidean(p1)

    def __repr__(self):
        return f"Segment with start {self.start} and end {self.end}

    def intersects(self, other):
        self_bbox = Bbox(self)
        other_bbox = Bbox(other)
        bbox_overlap = self_bbox.testOverlap(other_bbox)

        if bbox_overlap == False:
            return False

        apq = self.start.sideLine(other.start, other.end)
        bpq = self.end.sideLine(other.start, other.end)
        pab = other.start.sideLine(self.start, self.end)
        qab = other.end.sideLine(self.start, self.end)

        if (apq + bpq == 0 and pab + qab == 0):
            return True
        return False


class CoursePolygon():
    def __init__(self, exterior=None, holes=None, name="", poly_typ
        self.points = [Point(x, y) for x, y in exterior]
        if self.points[0] != self.points[-1]:
            self.points.append(self.points[0])

        self.holes = []
        for ring in holes or []:
```

```
            hole_points = [Point(x, y) for x, y in ring]
            if hole_points and hole_points[0] != hole_points[-1]:
                hole_points.append(hole_points[0])
            self.holes.append(hole_points)

        self.hole_bboxes = [Bbox(h) for h in self.holes]

        self.size = len(self.points)
        self.name = name
        self.poly_type = poly_type
        self.area_m2 = area_m2
        self.bbox = Bbox(self.points)

    def __repr__(self):
        return f"CoursePolygon(name={self.name}, type={self.poly_ty

    def __getitem__(self, key):
        return self.points[key]

    def isClosed(self):
        return self.points[0] == self.points[-1]

    def _ringContainsPoint(self, p, ring_points):
        count = 0
        for i in range(0, len(ring_points) - 1):
            start = ring_points[i]
            end = ring_points[i + 1]

            if (p.y > min(start.y, end.y)):
                if (p.y <= max(start.y, end.y)):
                    if (p.x <= max(start.x, end.x)):
                        if (start.y != end.y):
                            x_intersection = start.x + (p.y - start
                            if p.x <= x_intersection:
                                count += 1

        return count % 2 != 0


    def containsPoint(self, p):
        if not self.bbox.containsPoint(p):
            return False

        if not self._ringContainsPoint(p, self.points):
            return False

        for hole_pts, hole_bbox in zip(self.holes, self.hole_bboxes
            if hole_bbox.containsPoint(p) and self._ringContainsPoi
                return False

        return True
```

In [6]:
```
def build_course_polygons(records):
    polygons = []
    for record in records:
        polygons.append(CoursePolygon(
```

```
            exterior=record["exterior"],
            holes=record["holes"],
            name=record["name"],
            poly_type=record["type"],
            area_m2=record["area_m2"]
        ))
    return polygons


all_polygons = build_course_polygons(polygon_records)
print(f"Manual polygons created: {len(all_polygons)}")
print(all_polygons[0])

sample_polygon = CoursePolygon(
    exterior=[[0, 0], [10, 0], [10, 10], [0, 10], [0, 0]],
    holes=[[[3, 3], [7, 3], [7, 7], [3, 7], [3, 3]]],
    name="Sample",
    poly_type="Test",
    area_m2=100
)
assert sample_polygon.containsPoint(Point(2, 2)) == True
assert sample_polygon.containsPoint(Point(5, 5)) == False
assert sample_polygon.containsPoint(Point(12, 2)) == False
print("PIP-Test passed!")
```

```
Manual polygons created: 20
CoursePolygon(name=City_Hamburg, type=City, points=1227, holes=0)
PIP-Test passed!
```

3.1 Spatial Indexing

The Flickr dataset is too large to test every point against every polygon directly. Spatial indexing is used to further speed up the processing: the study area is divided into regular grid cells, and each `Point` is assigned to one cell based on its longitude and latitude.

For this project the grid is used in two steps. First, the compiled Flickr table is filtered with polygon bounding boxes so only points near the selected cities and parks are kept as candidates. Second, those candidates are inserted into a `PointIndex`. When a polygon is processed, the index only reads cells intersecting the polygon bounding box, instead of scanning all Flickr points again. This keeps the extensive point-in-polygon checks focused on relevant candidate points.

The grid index is an acceleration structure only: it does not decide whether a point is inside a polygon. It only reduces the search space before the exact manual point-in-polygon algorithm is applied.

In [7]:
```
class PointIndex():
    def __init__(self, data, box=None, res=0.05):
        self.res = res
```

```
        self.bBox = box if box else Bbox(data)
        w = self.bBox.ur.x - self.bBox.ll.x
        h = self.bBox.ur.y - self.bBox.ll.y
        self.nCols = int(w / self.res) + 1
        self.nRows = int(h / self.res) + 1

        ur = Point(
            self.bBox.ll.x + (self.nCols * self.res),
            self.bBox.ll.y + (self.nRows * self.res)
        )
        self.bBox = Bbox([self.bBox.ll, ur])
        self.maxIndex = (self.nCols * self.nRows) - 1
        self.points = [[0, []] for _ in range(self.maxIndex + 1)]
        self.bigArray = []

        self.addPoints(data)

    def __repr__(self):
        return f"PointIndex(res={self.res}, nCols={self.nCols}, nRo

    def addPoints(self, data):
        for p in data:
            self.addPoint(p)
            self.bigArray.append(p)

    def addPoint(self, p):
        i = self.pointIndex(p)
        if 0 <= i <= self.maxIndex:
            self.points[i][0] += 1
            self.points[i][1].append(p)

    def pointIndex(self, p):
        j = int((p.y - self.bBox.ll.y) / self.res)
        i = int((p.x - self.bBox.ll.x) / self.res)
        return (j * self.nCols) + i

    def regionQuery(self, region):
        query = self.bBox.intersectsRegion(region)
        if query is None:
            return [], 0

        c_start = max(0, int((query.ll.x - self.bBox.ll.x) / self.r
        c_end = min(self.nCols - 1, int((query.ur.x - self.bBox.ll.
        r_start = max(0, int((query.ll.y - self.bBox.ll.y) / self.r
        r_end = min(self.nRows - 1, int((query.ur.y - self.bBox.ll.

        ps = []
        for r in range(r_start, r_end + 1):
            for c in range(c_start, c_end + 1):
                index = (r * self.nCols) + c
                if 0 <= index <= self.maxIndex and self.points[inde
                    ps.extend(self.points[index][1])

        final = []
        for p in ps:
            if region.containsPoint(p):
```

```
                final.append(p)

        return final, len(final)

    def bruteRegionQuery(self, region):
        final = []
        for p in self.bigArray:
            if region.containsPoint(p):
                final.append(p)
        return final, len(final)

    def nearestPoint(self, p, method="haversine"):
        step = self.res
        max_step = max(self.bBox.ur.x - self.bBox.ll.x, self.bBox.u
        ps = []

        while len(ps) == 0 and step <= max_step:
            ll = Point(p.x - step, p.y - step)
            ur = Point(p.x + step, p.y + step)
            box = Bbox([ll, ur])
            ps, count = self.regionQuery(box)
            ps = [q for q in ps if q is not p]
            step = step + self.res

        if len(ps) == 0:
            return None, None

        old_count = len(ps)
        nearest_point, nearest_distance = self.__minDist(ps, p, met

        if method == "haversine":
            lat_buffer = nearest_distance / 111320.0
            lon_buffer = nearest_distance / (111320.0 * max(math.co
        else:
            lat_buffer = nearest_distance
            lon_buffer = nearest_distance

        ll = Point(p.x - lon_buffer, p.y - lat_buffer)
        ur = Point(p.x + lon_buffer, p.y + lat_buffer)
        ps2, count = self.regionQuery(Bbox([ll, ur]))
        ps2 = [q for q in ps2 if q is not p]

        if count > old_count and len(ps2) > 0:
            nearest_point, nearest_distance = self.__minDist(ps2, p

        return nearest_point, nearest_distance

    def __minDist(self, points, p, method="haversine"):
        best_point = None
        best_distance = float("inf")
        for q in points:
            if method == "haversine":
                d = p.distHaversine(q)
            else:
                d = p.distEuclidean(q)
            if d < best_distance:
```

```
                best_point = q
                best_distance = d
        return best_point, best_distance
```

In [8]:

```
def bbox_prefilter_flickr_points(compiled_csv, output_csv, polygons
    if os.path.exists(output_csv) and not REBUILD_CANDIDATE_CACHE:
        print(f"Load Bounding-Box candidates from {output_csv}...")
        return output_csv

    polygon_bboxes = [(p.bbox.ll.x, p.bbox.ll.y, p.bbox.ur.x, p.bbo
    total_rows = 0
    total_candidates = 0

    with open(output_csv, "w", encoding="utf-8", newline="") as out
        wrote_header = False
        for chunk_no, chunk in enumerate(pd.read_csv(compiled_csv,
            total_rows += len(chunk)
            lons = chunk["Longitude"].to_numpy(dtype=float)
            lats = chunk["Latitude"].to_numpy(dtype=float)
            mask = np.zeros(len(chunk), dtype=bool)

            for minx, miny, maxx, maxy in polygon_bboxes:
                mask |= ((lons >= minx) & (lons <= maxx) & (lats >=

            candidates = chunk.loc[mask].copy()
            total_candidates += len(candidates)
            candidates.to_csv(out_file, index=False, header=not wro
            wrote_header = True

            if chunk_no % 10 == 0:
                print(f"Chunks: {chunk_no} | Rows reviewed: {total_

    print(f"Bounding-Box Pre-Filter completed: {total_rows:,} -> {t
    return output_csv

bbox_prefilter_flickr_points(COMPILED_CSV, CANDIDATE_CSV, all_polyg
candidate_df = pd.read_csv(CANDIDATE_CSV, dtype=MIXED_TYPE_DTYPES,


def build_points_fast(df):
    spatial_points = []
    for idx, lon, lat, pid in zip(df.index,
                                  df['Longitude'],
                                  df['Latitude'],
                                  df['PhotoID']):
        spatial_points.append(Point(lon, lat, pid=pid, attrs={'df_i
    return spatial_points


spatial_points = build_points_fast(candidate_df)

print(f"Candidate points for the Algorithms: {len(spatial_points):,
if len(spatial_points) == 0:
    raise ValueError("No candidate points found. Please verify CRS

global_grid = PointIndex(spatial_points, res=GLOBAL_INDEX_RESOLUTIO
```

```
print(global_grid)

subset = spatial_points[:min(5000, len(spatial_points))]
subset_index = PointIndex(subset, res=GLOBAL_INDEX_RESOLUTION)
indexed_points, indexed_count = subset_index.regionQuery(all_polygo
brute_points, brute_count = subset_index.bruteRegionQuery(all_polyg
assert indexed_count == brute_count
print(f"Spatial-Index-Test passed: {indexed_count} == {brute_count}"
```

```
Chunks: 10 | Rows reviewed: 5,000,000 | Candidates: 1,271,457
Chunks: 20 | Rows reviewed: 10,000,000 | Candidates: 2,546,997
Bounding-Box Pre-Filter completed: 13,699,216 -> 3,509,122 Candidate
s
Candidate points for the Algorithms: 3,509,122
PointIndex(res=0.05, nCols=146, nRows=152, points=3509122)
Spatial-Index-Test passed: 0 == 0
```

3.2 Point in Polygon

The point-in-polygon algorithm follows the ray-casting logic. For a test point, a horizontal ray is imagined from the point to the right. The algorithm counts how often this ray crosses the polygon boundary. If the number of crossings is odd, the point is inside; if the number is even, it is outside.

Before ray casting, every polygon uses its bounding box as a fast pre-check. Points outside this rectangle cannot be inside the polygon, so they are skipped immediately. The candidate points come from the spatial index, which means the exact ray-casting test is only applied to a much smaller set of points.

The park polygons contain interior rings, or holes. A point must be inside the exterior ring and outside all holes to be counted as inside the park. The notebook therefore checks the exterior first and then excludes points that fall inside any interior ring. The output keeps one row per point-polygon match, which allows city and park results to be summarized independently.

In [9]:
```
def _ring_contains_batch(ring_points, xs, ys):
    """
    Vectorized ray-casting for N points simultaneously.
    Same algorithm as CoursePolygon._ringContainsPoint — just numpy
    of a Python loop over one point at a time.

    ring_points : list of Point  (polygon ring, closed)
    xs, ys      : numpy arrays of candidate coordinates (length N)
    returns     : boolean numpy array, True where point is inside t
    """
    count = np.zeros(len(xs), dtype=np.int32)

    for i in range(len(ring_points) - 1):
        x1 = ring_points[i].x;     y1 = ring_points[i].y
        x2 = ring_points[i + 1].x; y2 = ring_points[i + 1].y
```

```
        if y1 == y2:
            continue

        cond = (
            (ys >  min(y1, y2)) &
            (ys <= max(y1, y2)) &
            (xs <= max(x1, x2))
        )

        if not np.any(cond):
            continue

        x_int = x1 + (ys[cond] - y1) * (x2 - x1) / (y2 - y1)
        count[cond] += (xs[cond] <= x_int).astype(np.int32)

    return (count % 2) != 0


def batch_pip(poly, xs, ys):
    """
    Batch Point-in-Polygon for a CoursePolygon against N candidate p

    Pipeline (mirrors containsPoint):
      1. Bbox rejection     - numpy comparison, eliminates most can
      2. Exterior ring test - vectorized ray-casting (_ring_contain
      3. Hole tests         - vectorized ray-casting per hole, with

    Returns a boolean numpy array of length N.
    """
    # 1. Bbox pre-filter (vectorized)
    inside = (
        (xs >= poly.bbox.ll.x) & (xs <= poly.bbox.ur.x) &
        (ys >= poly.bbox.ll.y) & (ys <= poly.bbox.ur.y)
    )

    if not np.any(inside):
        return inside

    # 2. Exterior ring - only run on bbox survivors
    inside[inside] = _ring_contains_batch(poly.points,
                                          xs[inside], ys[inside])

    if not np.any(inside):
        return inside

    # 3. Holes - same vectorized ray-casting with per-hole bbox pre
    for hole_pts, hole_bbox in zip(poly.holes, poly.hole_bboxes):
        candidates = inside.copy()

        candidates &= (
            (xs >= hole_bbox.ll.x) & (xs <= hole_bbox.ur.x) &
            (ys >= hole_bbox.ll.y) & (ys <= hole_bbox.ur.y)
        )

        if np.any(candidates):
```

```
            in_hole = _ring_contains_batch(hole_pts,
                                           xs[candidates], ys[candi
            tmp = candidates.copy()
            tmp[tmp] = in_hole
            inside[tmp] = False

    return inside


def run_point_in_polygon(polygons, point_index, output_csv):
    if os.path.exists(output_csv) and not REBUILD_PIP_RESULTS:
        print(f"Load saved PIP-Results from {output_csv}...")
        return pd.read_csv(output_csv, dtype=MIXED_TYPE_DTYPES, low

    inside_indices = []
    poly_names     = []
    poly_types     = []
    poly_areas     = []

    for i, poly in enumerate(polygons, start=1):
        candidates, candidate_count = point_index.regionQuery(poly.

        if candidate_count == 0:
            print(f"{i:02d}/{len(polygons)} {poly.name}: Candidates
            continue

        xs = np.array([p.x for p in candidates], dtype=np.float64)
        ys = np.array([p.y for p in candidates], dtype=np.float64)

        mask = batch_pip(poly, xs, ys)
        inside_count = int(mask.sum())

        for p in (c for c, m in zip(candidates, mask) if m):
            inside_indices.append(p.attributes['df_index'])
            poly_names.append(poly.name)
            poly_types.append(poly.poly_type)
            poly_areas.append(poly.area_m2)

        print(f"{i:02d}/{len(polygons)} {poly.name}: "
              f"Candidates = {candidate_count:,}, inside = {inside_

    result = candidate_df.loc[inside_indices].copy()
    result["polygon_name"]    = poly_names
    result["polygon_type"]    = poly_types
    result["polygon_area_m2"] = poly_areas

    result.to_csv(output_csv, index=False)
    print(f"PIP finished. Saved results: {len(result):,}")
    return result


inside_points_df = run_point_in_polygon(all_polygons, global_grid, 
inside_points_df.head()
```

```
01/20 City_Hamburg: Candidates = 411,248, inside = 358,183
02/20 City_Bremen: Candidates = 79,735, inside = 52,322
03/20 City_Düsseldorf: Candidates = 121,390, inside = 114,413
04/20 City_Köln: Candidates = 228,634, inside = 213,116
05/20 City_Dortmund: Candidates = 53,965, inside = 48,377
06/20 City_Frankfurt am Main: Candidates = 245,229, inside = 205,459
07/20 City_Stuttgart: Candidates = 140,113, inside = 128,815
08/20 City_München: Candidates = 351,316, inside = 338,808
09/20 City_Berlin: Candidates = 1,109,976, inside = 1,086,095
10/20 City_Leipzig: Candidates = 98,570, inside = 93,681
11/20 Park_Schleswig-Holsteinisches Wattenmeer: Candidates = 44,880,
inside = 4,923
12/20 Park_Niedersächsisches Wattenmeer: Candidates = 53,381, inside
= 6,318
13/20 Park_Bayerischer Wald: Candidates = 20,370, inside = 9,356
14/20 Park_Südschwarzwald: Candidates = 110,015, inside = 31,686
15/20 Park_Schwarzwald Mitte/Nord: Candidates = 92,554, inside = 32,
130
16/20 Park_Bergstraße: Candidates = 155,705, inside = 26,189
17/20 Park_Fränkische Schweiz-Veldensteiner Forst: Candidates = 32,3
78, inside = 11,076
18/20 Park_Teutoburger Wald/Eggegebirge: Candidates = 76,628, inside
= 31,141
19/20 Park_Sauerland-Rothaargebirge: Candidates = 59,977, inside = 2
5,739
20/20 Park_Altmühltal: Candidates = 27,974, inside = 11,298
PIP finished. Saved results: 2,829,125
```

Out[9]:

	PhotoID	Latitude	Longitude	UserID	DateTaken	pc
296382	7034969235	53.430775	9.919569	51337247@N08	3/31/2012 19:29:12	(
856237	7035036113	53.430775	9.919569	51337247@N08	3/31/2012 19:35:30	(
1403479	6888861108	53.430775	9.919569	51337247@N08	3/31/2012 19:19:02	(
2987147	6888874024	53.430775	9.919569	51337247@N08	3/31/2012 19:26:30	(
3122427	6888863686	53.430775	9.919569	51337247@N08	3/31/2012 19:21:40	(

In [10]:

```
if len(inside_points_df) == 0:
    print("No points found within the polygons.")

else:
    point_counts = (
        inside_points_df
        .groupby(["polygon_type", "polygon_name"])
        .size()
        .reset_index(name="point_count")
        .sort_values(["polygon_type", "point_count"], ascending=[Tr
    )
    display(point_counts)
```

	polygon_type	polygon_name	point_count
0	City	City_Berlin	1086095
5	City	City_Hamburg	358183
8	City	City_München	338808
6	City	City_Köln	213116
4	City	City_Frankfurt am Main	205459
9	City	City_Stuttgart	128815
3	City	City_Düsseldorf	114413
7	City	City_Leipzig	93681
1	City	City_Bremen	52322
2	City	City_Dortmund	48377
17	Park	Park_Schwarzwald Mitte/Nord	32130
18	Park	Park_Südschwarzwald	31686
19	Park	Park_Teutoburger Wald/Eggegebirge	31141
12	Park	Park_Bergstraße	26189
15	Park	Park_Sauerland-Rothaargebirge	25739
10	Park	Park_Altmühltal	11298
13	Park	Park_Fränkische Schweiz-Veldensteiner Forst	11076
11	Park	Park_Bayerischer Wald	9356
14	Park	Park_Niedersächsisches Wattenmeer	6318
16	Park	Park_Schleswig-Holsteinisches Wattenmeer	4923

3.3 Sampled-Query Nearest Neighbor Index

The sampled-query NNI samples at most `NNI_SAMPLE_SIZE` query points per polygon to keep runtime manageable, but it builds the local `PointIndex` from all points inside that polygon. Therefore, each sampled query point searches for its nearest neighbor among the full polygon point set.

The expected nearest-neighbor distance is also based on the full point density, using `total_points / polygon_area_m2`. This better reflects the actual Flickr point pattern while avoiding the cost of calculating nearest neighbors for every single point in very dense polygons such as Berlin or Hamburg.

Additional step: spatial deduplication of points that have been stored stacked at the sam ecoordinates, that would falsify the NNI.

Note: If this step takes to long to load, reduce the `NNI_SAMPLE_SIZE` in the firs Setup cell.

In [11]:
```
def compute_fast_sampled_query_nni(group, area_m2, sample_size=NNI_
    # 1. SPATIAL DEDUPLICATION: Keep only unique coordinates
    unique_group = group.drop_duplicates(subset=["Latitude", "Longi
    total_points = len(unique_group)

    if total_points < 2 or area_m2 <= 0:
        return None

    lats = unique_group["Latitude"].to_numpy(dtype=float)
    lons = unique_group["Longitude"].to_numpy(dtype=float)

    # 2. Select random query indices from the UNIQUE points
    if total_points > sample_size:
        np.random.seed(RANDOM_SEED)
        query_indices = np.random.choice(total_points, sample_size,
    else:
        query_indices = np.arange(total_points)

    query_points_used = len(query_indices)

    lats_rad = np.radians(lats)
    lons_rad = np.radians(lons)

    distances = np.empty(query_points_used)
    R = 6371000.0

    for i, q_idx in enumerate(query_indices):
        lat1_rad = lats_rad[q_idx]
        lon1_rad = lons_rad[q_idx]

        dphi = lats_rad - lat1_rad
        dlam = lons_rad - lon1_rad

        a = np.sin(dphi / 2.0)**2 + np.cos(lat1_rad) * np.cos(lats_
        c = 2 * np.arcsin(np.sqrt(a))
        dist_array = R * c

        dist_array[q_idx] = np.inf
        distances[i] = np.min(dist_array)

    observed_mean_distance = np.mean(distances)
    point_density = total_points / area_m2
    expected_mean_distance = 1.0 / (2.0 * np.sqrt(point_density))
    nni = observed_mean_distance / expected_mean_distance

    # 3. NORMALIZATION: Calculate Z-score to determine statistical
    se = 0.26136 / np.sqrt((total_points**2) / area_m2)
    z_score = (observed_mean_distance - expected_mean_distance) / s

    return observed_mean_distance, expected_mean_distance, nni, z_s
```

```
def run_fast_sampled_query_nni(inside_df, output_csv):
    result_columns = [
        "polygon_type", "polygon_name", "total_photos", "unique_loca
        "observed_mean_distance_m", "expected_mean_distance_m", "nn
    ]

    if os.path.exists(output_csv) and not REBUILD_NNI_RESULTS:
        print(f"Load saved Sampled Query NNI results from {output_cs
        return pd.read_csv(output_csv, low_memory=False)

    if len(inside_df) == 0:
        empty = pd.DataFrame(columns=result_columns)
        empty.to_csv(output_csv, index=False)
        return empty

    results = []

    for (poly_type, poly_name), group in inside_df.groupby(["polygo
        area_m2 = float(group["polygon_area_m2"].iloc[0])

        stats = compute_fast_sampled_query_nni(group, area_m2)

        if stats is None:
            observed, expected, nni, z_score = np.nan, np.nan, np.na
            query_points_used = 0
            unique_locations = 0
            nn_distances = None
            pattern = "not enough points"
        else:
            observed, expected, nni, z_score, query_points_used, un

            if nni < 1 and z_score < -1.96:
                pattern = "clustered"
            elif nni > 1 and z_score > 1.96:
                pattern = "dispersed"
            else:
                pattern = "random"

        results.append({
            "polygon_type": poly_type,
            "polygon_name": poly_name,
            "total_photos": len(group),
            "unique_locations": unique_locations,
            "query_points_used": query_points_used,
            "area_m2": area_m2,
            "observed_mean_distance_m": observed,
            "expected_mean_distance_m": expected,
            "nni": nni,
            "z_score": z_score,
            "pattern": pattern,
            "all_nn_distances": nn_distances,
        })

        print(
            f"{poly_name}: photos = {len(group):,} | unique locatio
            f"NNI = {nni:.4f} | Z-Score = {z_score:.1f}"
```

```
            if not np.isnan(nni) else f"{poly_name}: not enough poi
        )

    results_df = pd.DataFrame(results).sort_values(["polygon_type",
    results_df.drop(columns=["all_nn_distances"]).to_csv(output_csv
    return results_df


nni_query_results = run_fast_sampled_query_nni(inside_points_df, NN
display(nni_query_results.head())
```

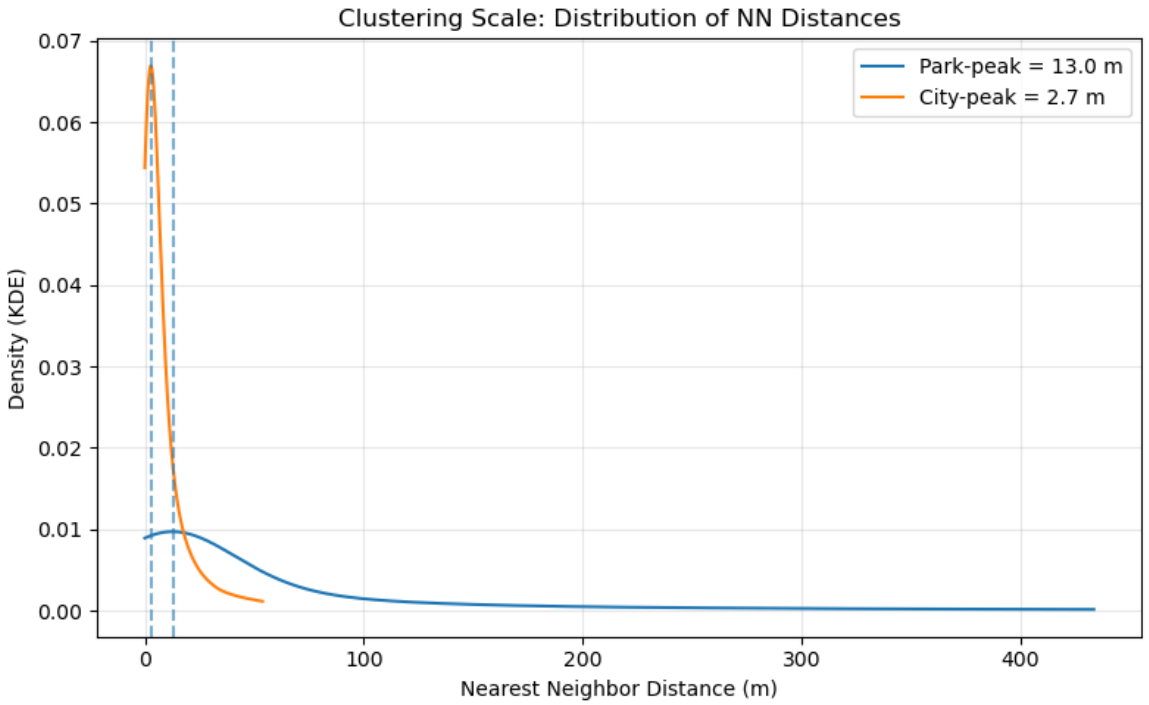
```
City_Berlin: photos = 1,086,095 | unique locations = 463,217 | NNI =
0.3520 | Z-Score = -843.7
City_Bremen: photos = 52,322 | unique locations = 22,296 | NNI = 0.2
965 | Z-Score = -201.0
City_Dortmund: photos = 48,377 | unique locations = 16,137 | NNI =
0.4166 | Z-Score = -141.8
City_Düsseldorf: photos = 114,413 | unique locations = 38,468 | NNI
= 0.3582 | Z-Score = -240.8
City_Frankfurt am Main: photos = 205,459 | unique locations = 82,014
| NNI = 0.3524 | Z-Score = -354.8
City_Hamburg: photos = 358,183 | unique locations = 142,121 | NNI =
0.3360 | Z-Score = -478.9
City_Köln: photos = 213,116 | unique locations = 78,289 | NNI = 0.28
04 | Z-Score = -385.2
City_Leipzig: photos = 93,681 | unique locations = 29,385 | NNI = 0.
3071 | Z-Score = -227.2
City_München: photos = 338,808 | unique locations = 135,989 | NNI =
0.3425 | Z-Score = -463.8
City_Stuttgart: photos = 128,815 | unique locations = 40,021 | NNI =
0.3439 | Z-Score = -251.1
Park_Altmühltal: photos = 11,298 | unique locations = 5,159 | NNI =
0.2824 | Z-Score = -98.6
Park_Bayerischer Wald: photos = 9,356 | unique locations = 4,694 | N
NI = 0.3102 | Z-Score = -90.4
Park_Bergstraße: photos = 26,189 | unique locations = 10,708 | NNI =
0.3365 | Z-Score = -131.4
Park_Fränkische Schweiz-Veldensteiner Forst: photos = 11,076 | uniqu
e locations = 5,635 | NNI = 0.3281 | Z-Score = -96.5
Park_Niedersächsisches Wattenmeer: photos = 6,318 | unique locations
= 4,027 | NNI = 0.2605 | Z-Score = -89.8
Park_Sauerland-Rothaargebirge: photos = 25,739 | unique locations =
14,733 | NNI = 0.3426 | Z-Score = -152.7
Park_Schleswig-Holsteinisches Wattenmeer: photos = 4,923 | unique lo
cations = 2,873 | NNI = 0.2480 | Z-Score = -77.1
Park_Schwarzwald Mitte/Nord: photos = 32,130 | unique locations = 1
5,903 | NNI = 0.3009 | Z-Score = -168.7
Park_Südschwarzwald: photos = 31,686 | unique locations = 15,013 | N
NI = 0.3271 | Z-Score = -157.7
Park_Teutoburger Wald/Eggegebirge: photos = 31,141 | unique location
s = 9,426 | NNI = 0.2860 | Z-Score = -132.6
```

	polygon_type	polygon_name	total_photos	unique_locations	query_points
6	City	City_Köln	213116	78289	
1	City	City_Bremen	52322	22296	
7	City	City_Leipzig	93681	29385	
5	City	City_Hamburg	358183	142121	
8	City	City_München	338808	135989	

3.4 Nearest Neighbor Distance Distribution (KDE)

In [12]:
```
park_dist = np.concatenate(
    nni_query_results[nni_query_results["polygon_type"] == "Park"]['
)
city_dist = np.concatenate(
    nni_query_results[nni_query_results["polygon_type"] == "City"]['
)


def kde_peak(dist):
    """KDE of NN distances — returns peak distance, x-values and y-
    kde = gaussian_kde(dist)
    xs  = np.linspace(0, np.percentile(dist, 95), 500)
    ys  = kde(xs)
    return xs[np.argmax(ys)], xs, ys


park_peak, park_x, park_y = kde_peak(park_dist)
city_peak, city_x, city_y = kde_peak(city_dist)

plt.figure(figsize=(8, 5))
plt.plot(park_x, park_y, label=f"Park-peak = {park_peak:.1f} m")
plt.plot(city_x, city_y, label=f"City-peak = {city_peak:.1f} m")
plt.axvline(park_peak, linestyle="--", alpha=0.6)
plt.axvline(city_peak, linestyle="--", alpha=0.6)
plt.xlabel("Nearest Neighbor Distance (m)")
plt.ylabel("Density (KDE)")
plt.title("Clustering Scale: Distribution of NN Distances")
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
```

```
plt.savefig(PATH_OUTPUTS / "NN_KDE_plot.png", dpi=300, bbox_inches='
plt.show()

print(f"Park  clustering scale (KDE peak): {park_peak:.1f} m")
print(f"City  clustering scale (KDE peak): {city_peak:.1f} m")
```




```
Park  clustering scale (KDE peak): 13.0 m
City  clustering scale (KDE peak): 2.7 m
```

4. Summary & Results

The final comparison uses the sampled-query NNI as the prefered result. This version samples query points for speed, but searches nearest neighbors among all points in each polygon and uses the full point density for the expected distance. Values below 1 indicate clustering; values above 1 indicate dispersion/randomness relative to a random point pattern.

The sampled/thinned NNI from the previous cell remains available as a fallback and comparison, but its values describe a random subset rather than the full Flickr point pattern.

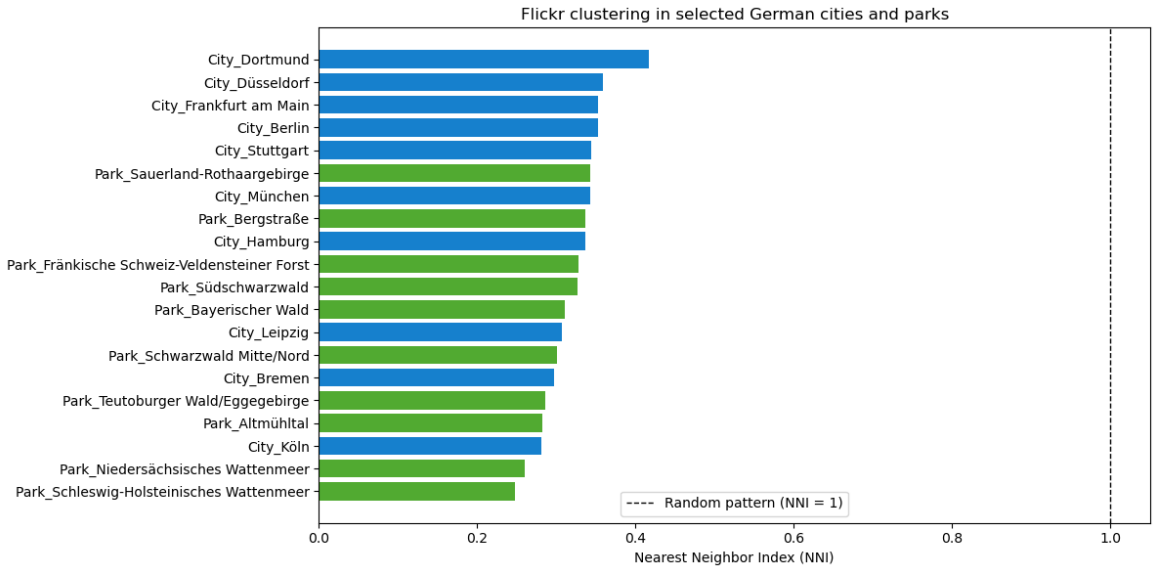
In [13]:
```
if len(nni_query_results) == 0:
    print("No Sampled Query NNI results available.")
else:
    summary = (
        nni_query_results
        .dropna(subset=["nni"])
        .groupby("polygon_type")
        .agg(
            polygons=("polygon_name", "count"),
            unique_locations=("unique_locations", "sum"),
            mean_nni=("nni", "mean"),
```

```
            median_nni=("nni", "median"),
            mean_observed_distance_m=("observed_mean_distance_m", "
            mean_expected_distance_m=("expected_mean_distance_m", "
        )
        .reset_index()
    )
    display(summary)

    if set(summary["polygon_type"]) >= {"City", "Park"}:
        avg_city = float(summary.loc[summary["polygon_type"] == "Ci
        avg_park = float(summary.loc[summary["polygon_type"] == "Pa
        stronger_cluster = "Städten" if avg_city < avg_park else "P
        print(f"Average Sampled-Query NNI by City: {avg_city:.4f}")
        print(f"Average Sampled-Query NNI by Parks:  {avg_park:.4f}"
        print(f"Conclusion: The Flickr points are stronger clustere
```

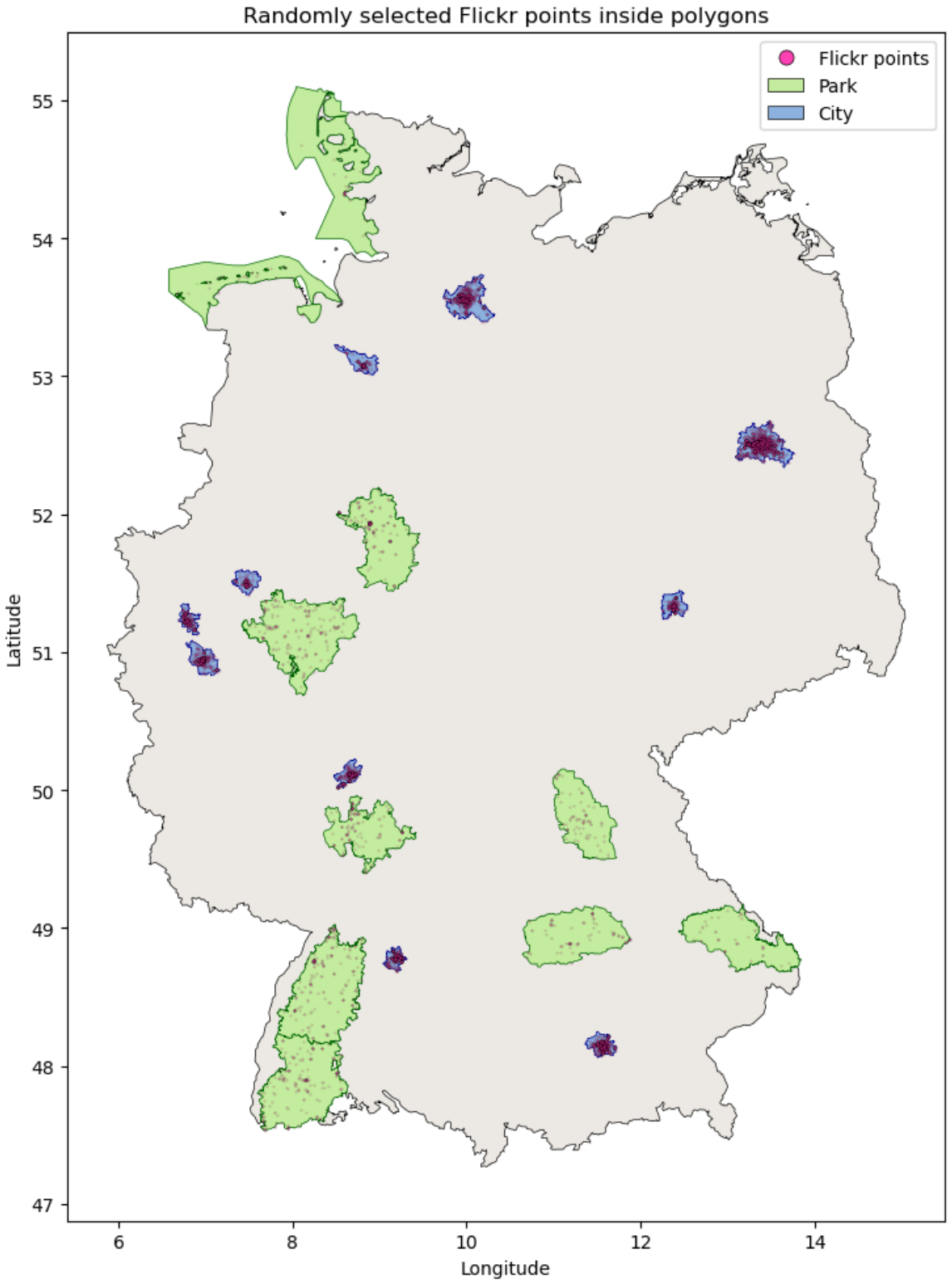
	polygon_type	polygons	unique_locations	mean_nni	median_nni	mean_ol
0	City	10	1047937	0.338569	0.343230	
1	Park	10	88171	0.302217	0.305534	

```
Average Sampled-Query NNI by City: 0.3386
Average Sampled-Query NNI by Parks:  0.3022
Conclusion: The Flickr points are stronger clustered in Parks!
```

In [14]:
```
if len(nni_query_results.dropna(subset=["nni"])) > 0:
    plot_df = nni_query_results.dropna(subset=["nni"]).sort_values("
    colors = plot_df["polygon_type"].map({"City": "#1680cd", "Park"

    fig, ax = plt.subplots(figsize=(12, 6))
    ax.barh(plot_df["polygon_name"], plot_df["nni"], color=colors)
    ax.axvline(1.0, color="black", linewidth=1, linestyle="--", lab
    ax.set_xlabel("Nearest Neighbor Index (NNI)")
    ax.set_title("Flickr clustering in selected German cities and p
    ax.legend()
    plt.tight_layout()
    plt.savefig(PATH_OUTPUTS / "NNI_barh_plot.png", dpi=300, bbox_i
    plt.show()
```



In [15]:
```
germany_basemap = gpd.read_file(GEOJSON_GERMANY).to_crs(epsg=4326)

if len(inside_points_df) > 0:
    cities_plot = gpd.read_file(GEOJSON_CITIES).to_crs(epsg=4326)
    parks_plot = gpd.read_file(GEOJSON_PARKS).to_crs(epsg=4326)

    plot_points = inside_points_df.drop_duplicates("PhotoID")

    if len(plot_points) > 20000:
        plot_points = plot_points.sample(20000, random_state=RANDOM_

    fig, ax = plt.subplots(figsize=(12, 10))
    germany_basemap.plot(ax=ax, color="#ece8e3", linewidth=0.5, edg
    parks_plot.plot(ax=ax, color="#C4EC9E", linewidth=0.5, edgecolo
    cities_plot.plot(ax=ax, color="#8DB1DF", linewidth=0.5, edgecol
    ax.scatter(plot_points["Longitude"], plot_points["Latitude"], s
    ax.set_xlabel("Longitude")
    ax.set_ylabel("Latitude")
    ax.set_title("Randomly selected Flickr points inside polygons")

    legend_elements = [
        Line2D([0], [0], marker='o', color='w', label='Flickr point
               markerfacecolor='#ff41b3', markersize=8,
               markeredgecolor='black', markeredgewidth=0.5),
        mpatches.Patch(facecolor='#C4EC9E', edgecolor='black', linew
        mpatches.Patch(facecolor='#8DB1DF', edgecolor='black', linew
    ]
    ax.legend(handles=legend_elements, loc='upper right')

    plt.tight_layout()
    plt.savefig(PATH_OUTPUTS / "Flickr_points_rand_selection.png",
    plt.show()
```



In [16]:
```
print("Output-Files:")
print(f"Compiled Flickr-Points: {COMPILED_CSV}")
print(f"Bounding-Box-Candidates: {CANDIDATE_CSV}")
print(f"PIP-Results: {FILTERED_CSV}")
print(f"Sampled/thinned NNI Results: {NNI_RESULTS_CSV}")
print(f"Sampled-query NNI Results: {NNI_QUERY_RESULTS_CSV}")
```

```
Output-Files:
Compiled Flickr-Points: /Users/maxlengenfelder/Desktop/FS26/GEO877/P
roject/GEO877_Flickr_Project/outputs/all_flickr_points.csv
Bounding-Box-Candidates: /Users/maxlengenfelder/Desktop/FS26/GEO877/
Project/GEO877_Flickr_Project/outputs/flickr_bbox_candidates.csv
PIP-Results: /Users/maxlengenfelder/Desktop/FS26/GEO877/Project/GEO8
77_Flickr_Project/outputs/filtered_flickr_inside_polygons.csv
Sampled/thinned NNI Results: /Users/maxlengenfelder/Desktop/FS26/GEO
877/Project/GEO877_Flickr_Project/outputs/nni_results_sampled.csv
Sampled-query NNI Results: /Users/maxlengenfelder/Desktop/FS26/GEO87
7/Project/GEO877_Flickr_Project/outputs/nni_results_sampled_query.cs
v
```

5. Extra: Validation with geospatial libraries

Three steps:

1. Export point data to GeoPackage files (data/processed/)
2. Compute ground-truth NNI using sklearn BallTree (haversine, ALL unique locations)
3. Visualise: bar-chart comparison (manual vs library) + NNI choropleth map

In [17]:
```
PROCESSED_ALL_GPKG      = PATH_PROCESSED / 'all_flickr_points.gpkg'
PROCESSED_FILTERED_GPKG = PATH_PROCESSED / 'filtered_flickr_inside_
VALIDATION_NNI_CSV      = PATH_OUTPUTS / 'nni_results_validation.cs

print("1/3  Export GeoPackage files...")

all_df = pd.read_csv(COMPILED_CSV, dtype=MIXED_TYPE_DTYPES, low_mem
gpd.GeoDataFrame(
    all_df,
    geometry=gpd.points_from_xy(all_df.Longitude, all_df.Latitude),
    crs="EPSG:4326"
).to_file(PROCESSED_ALL_GPKG, driver="GPKG")

print(f"     All Flickr-Points      : {len(all_df):,}  →  {PROCESSE

filtered_gdf = gpd.GeoDataFrame(
    inside_points_df,
    geometry=gpd.points_from_xy(inside_points_df.Longitude, inside_
    crs="EPSG:4326"
)
filtered_gdf.to_file(PROCESSED_FILTERED_GPKG, driver="GPKG")

print(f"     Filtered Points (PIP)  : {len(filtered_gdf):,}  →  {PR
```

```
1/3  Export GeoPackage files...
     All Flickr-Points      : 13,699,216  →  /Users/maxlengenfelder/
Desktop/FS26/GEO877/Project/GEO877_Flickr_Project/data/processed/all
_flickr_points.gpkg
     Filtered Points (PIP)  : 2,829,125  →  /Users/maxlengenfelder/D
esktop/FS26/GEO877/Project/GEO877_Flickr_Project/data/processed/filt
ered_flickr_inside_polygons.gpkg
```

In [18]:

```python
def compute_library_nni(group, area_m2):
    unique  = group.drop_duplicates(subset=["Latitude", "Longitude"
    n       = len(unique)
    if n < 2 or area_m2 <= 0:
        return None

    coords_rad        = np.radians(unique[["Latitude", "Longitude"]
    tree              = BallTree(coords_rad, metric="haversine")
    distances, _      = tree.query(coords_rad, k=2)
    nn_m              = distances[:, 1] * 6371000.0

    observed  = float(np.mean(nn_m))
    density   = n / area_m2
    expected  = 1.0 / (2.0 * np.sqrt(density))
    nni       = observed / expected
    se        = 0.26136 / np.sqrt(n**2 / area_m2)
    z_score   = (observed - expected) / se

    return observed, expected, nni, z_score, n


print("\n2/3  Calculate validation NNI (BallTree, all unique coordi
val_rows = []

for (poly_type, poly_name), group in inside_points_df.groupby(["poly
    area_m2 = float(group["polygon_area_m2"].iloc[0])
    stats   = compute_library_nni(group, area_m2)

    if stats is None:
        obs, exp, nni, z, n_uniq = np.nan, np.nan, np.nan, np.nan, (
        pattern = "not enough points"
    else:
        obs, exp, nni, z, n_uniq = stats
        if   nni < 1 and z < -1.96: pattern = "clustered"
        elif nni > 1 and z >  1.96: pattern = "dispersed"
        else:                        pattern = "random"

    val_rows.append({
        "polygon_type":             poly_type,
        "polygon_name":             poly_name,
        "total_photos":             len(group),
        "unique_locations":         n_uniq,
        "area_m2":                  area_m2,
        "observed_mean_distance_m": obs,
        "expected_mean_distance_m": exp,
        "nni":                      nni,
        "z_score":                  z,
        "pattern":                  pattern,
```

```
    })
    print(f"     {poly_name:<45}  unique = {n_uniq:>7,}  NNI = {nni

val_df = (pd.DataFrame(val_rows)
            .sort_values(["polygon_type", "nni"])
            .reset_index(drop=True))
val_df.to_csv(VALIDATION_NNI_CSV, index=False)

print(f"\n     Results saved: {VALIDATION_NNI_CSV}")

display(val_df[["polygon_type", "polygon_name", "unique_locations",
```

```
2/3  Calculate validation NNI (BallTree, all unique coordinates)...
     City_Berlin                                        unique = 463,217
NNI = 0.3479  Z =  -849.1  [clustered]
     City_Bremen                                        unique =  22,296
NNI = 0.3056  Z =  -198.3  [clustered]
     City_Dortmund                                      unique =  16,137
NNI = 0.4142  Z =  -142.4  [clustered]
     City_Düsseldorf                                    unique =  38,468
NNI = 0.3653  Z =  -238.2  [clustered]
     City_Frankfurt am Main                             unique =  82,014
NNI = 0.3614  Z =  -349.9  [clustered]
     City_Hamburg                                       unique = 142,121
NNI = 0.3351  Z =  -479.5  [clustered]
     City_Köln                                          unique =  78,289
NNI = 0.2983  Z =  -375.6  [clustered]
     City_Leipzig                                       unique =  29,385
NNI = 0.3199  Z =  -223.0  [clustered]
     City_München                                       unique = 135,989
NNI = 0.3552  Z =  -454.9  [clustered]
     City_Stuttgart                                     unique =  40,021
NNI = 0.3471  Z =  -249.9  [clustered]
     Park_Altmühltal                                    unique =   5,159
NNI = 0.2824  Z =   -98.6  [clustered]
     Park_Bayerischer Wald                              unique =   4,694
NNI = 0.3102  Z =   -90.4  [clustered]
     Park_Bergstraße                                    unique =  10,708
NNI = 0.3375  Z =  -131.1  [clustered]
     Park_Fränkische Schweiz-Veldensteiner Forst        unique =   5,635
NNI = 0.3281  Z =   -96.5  [clustered]
     Park_Niedersächsisches Wattenmeer                  unique =   4,027
NNI = 0.2605  Z =   -89.8  [clustered]
     Park_Sauerland-Rothaargebirge                      unique =  14,733
NNI = 0.3465  Z =  -151.7  [clustered]
     Park_Schleswig-Holsteinisches Wattenmeer           unique =   2,873
NNI = 0.2480  Z =   -77.1  [clustered]
     Park_Schwarzwald Mitte/Nord                        unique =  15,903
NNI = 0.3011  Z =  -168.6  [clustered]
     Park_Südschwarzwald                                unique =  15,013
NNI = 0.3252  Z =  -158.2  [clustered]
     Park_Teutoburger Wald/Eggegebirge                  unique =   9,426
NNI = 0.2860  Z =  -132.6  [clustered]

     Results saved: /Users/maxlengenfelder/Desktop/FS26/GEO877/Proje
ct/GEO877_Flickr_Project/outputs/nni_results_validation.csv
```

	polygon_type	polygon_name	unique_locations	nni	z_s
0	City	City_Köln	78289	0.298296	-375.60
1	City	City_Bremen	22296	0.305638	-198.34
2	City	City_Leipzig	29385	0.319934	-223.02
3	City	City_Hamburg	142121	0.335145	-479.49
4	City	City_Stuttgart	40021	0.347088	-249.87
5	City	City_Berlin	463217	0.347905	-849.05
6	City	City_München	135989	0.355235	-454.86
7	City	City_Frankfurt am Main	82014	0.361359	-349.89
8	City	City_Düsseldorf	38468	0.365287	-238.15
9	City	City_Dortmund	16137	0.414239	-142.35
10	Park	Park_Schleswig-Holsteinisches Wattenmeer	2873	0.247954	-77.11
11	Park	Park_Niedersächsisches Wattenmeer	4027	0.260461	-89.78
12	Park	Park_Altmühltal	5159	0.282394	-98.60
13	Park	Park_Teutoburger Wald/Eggegebirge	9426	0.285991	-132.61
14	Park	Park_Schwarzwald Mitte/Nord	15903	0.301079	-168.61
15	Park	Park_Bayerischer Wald	4694	0.310150	-90.41
16	Park	Park_Südschwarzwald	15013	0.325206	-158.17
17	Park	Park_Fränkische Schweiz-Veldensteiner Forst	5635	0.328126	-96.48
18	Park	Park_Bergstraße	10708	0.337508	-131.14
19	Park	Park_Sauerland-Rothaargebirge	14733	0.346504	-151.74

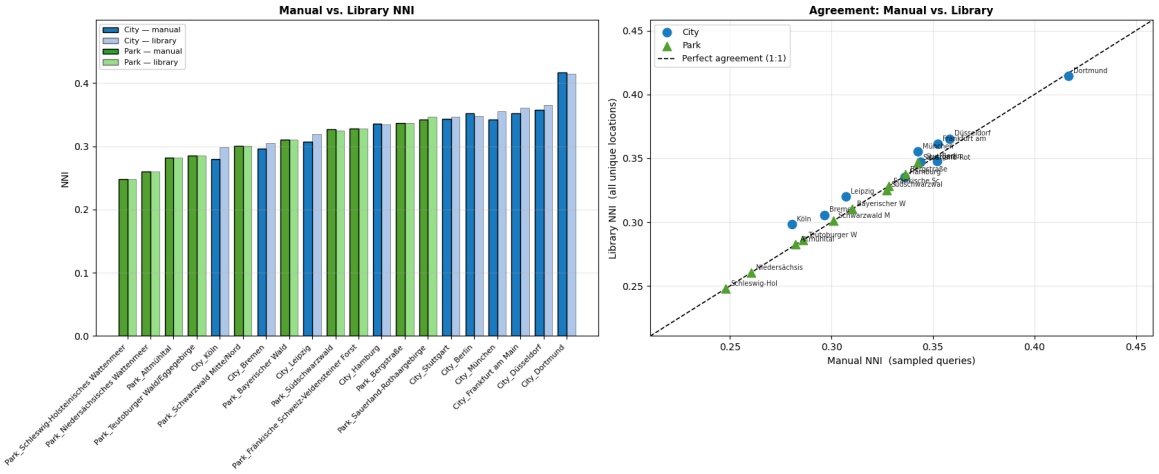
In [19]:
```
print("\n3/3  Create validation plots...")

TYPE_COLORS = {"City": "#1b7bc0", "Park": "#52a02f"}

compare = (
    nni_query_results[["polygon_name", "polygon_type", "nni"]]
    .rename(columns={"nni": "nni_manual"})
    .dropna(subset=["nni_manual"])
    .merge(
        val_df[["polygon_name", "nni"]].rename(columns={"nni": "nni_
        on="polygon_name"
```

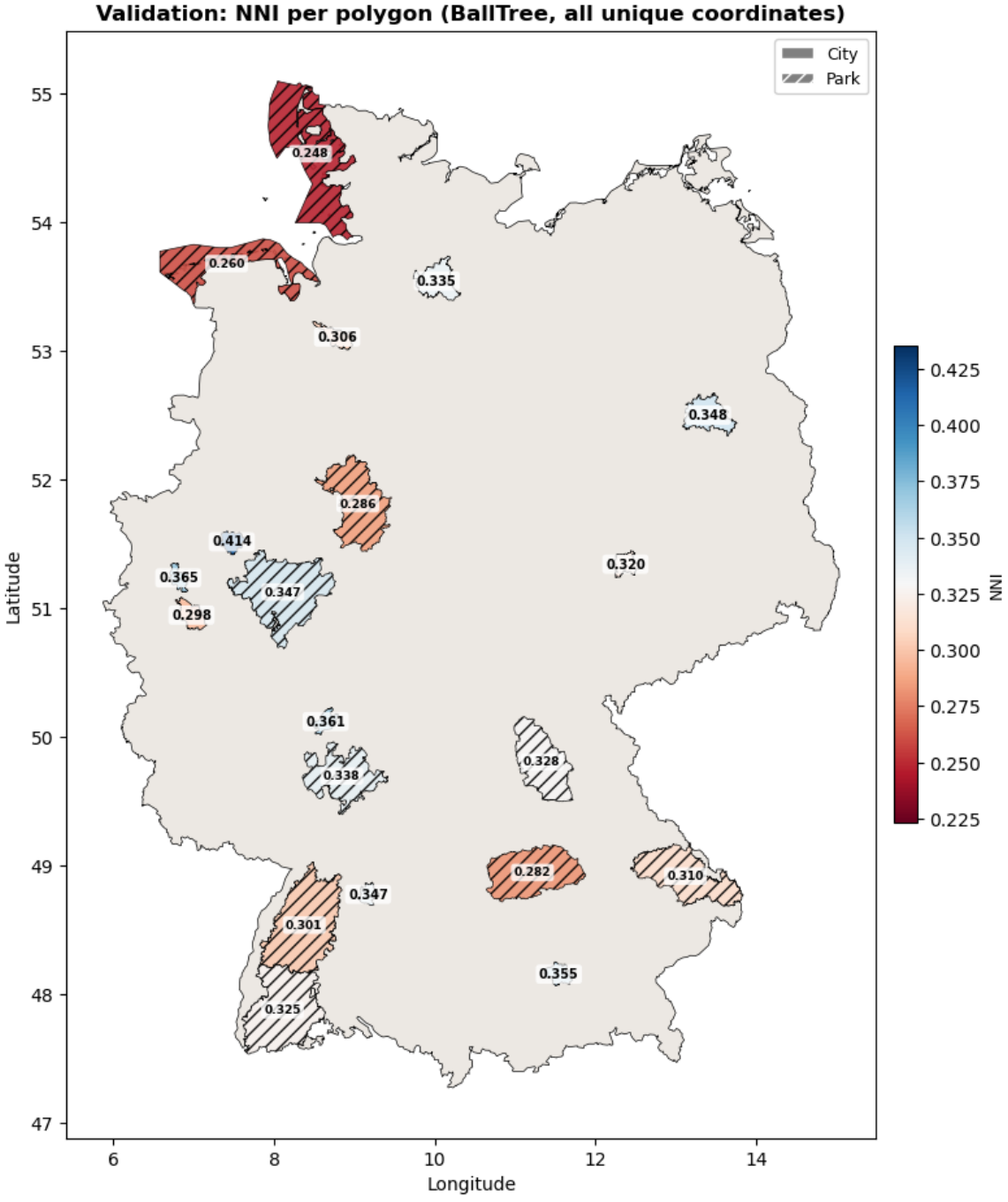
```
    )
    .sort_values("nni_library")
    .reset_index(drop=True)
)


# Plot A + B: comparison bar chart  &  scatter agreement
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(17, 7))

# Bar chart
x  = np.arange(len(compare))
w  = 0.38
c_dark  = [TYPE_COLORS[t] for t in compare["polygon_type"]]
c_light = ["#aec7e8" if t == "City" else "#98df8a" for t in compare

ax1.bar(x + w/2, compare["nni_library"], width=w, color=c_light, ed
ax1.bar(x - w/2, compare["nni_manual"],  width=w, color=c_dark, edg
ax1.axhline(1.0, color="black", linestyle="--", linewidth=1.2, labe
ax1.set_xticks(x)
ax1.set_xticklabels(compare["polygon_name"], rotation=45, ha="right'
ax1.set_ylabel("NNI")
ax1.set_title("Manual vs. Library NNI", fontsize=11, fontweight="bo
ax1.set_ylim(0, max(compare[["nni_manual", "nni_library"]].max()) *
ax1.grid(axis="y", alpha=0.3)
handles = [
    mpatches.Patch(facecolor="#1f77b4", edgecolor="black", label="C
    mpatches.Patch(facecolor="#aec7e8", edgecolor="dimgray", label='
    mpatches.Patch(facecolor="#2ca02c", edgecolor="black", label="Pa
    mpatches.Patch(facecolor="#98df8a", edgecolor="dimgray", label='
]
ax1.legend(handles=handles, loc='upper left', fontsize=8, framealpha

# Scatter plot
for ptype, marker in [("City", "o"), ("Park", "^")]:
    sub = compare[compare["polygon_type"] == ptype]
    ax2.scatter(sub["nni_manual"], sub["nni_library"],
                color=TYPE_COLORS[ptype], marker=marker, s=80, zorde
    for _, row in sub.iterrows():
        label = row["polygon_name"].replace("City_", "").replace("Pa
        ax2.annotate(label, (row["nni_manual"], row["nni_library"])
                     textcoords="offset points", xytext=(5, 3), font

all_nni = pd.concat([compare["nni_manual"], compare["nni_library"]]
lo, hi  = all_nni.min() * 0.85, all_nni.max() * 1.1
ax2.plot([lo, hi], [lo, hi], "k--", linewidth=1.2, label="Perfect ag
ax2.set_xlim(lo, hi); ax2.set_ylim(lo, hi)
ax2.set_xlabel("Manual NNI  (sampled queries)")
ax2.set_ylabel("Library NNI  (all unique locations)")
ax2.set_title("Agreement: Manual vs. Library", fontsize=11, fontweig
ax2.legend(fontsize=9); ax2.grid(alpha=0.3)

plt.tight_layout()
plt.savefig(PATH_OUTPUTS / "validation_nni_comparison.png", dpi=150
plt.show()

# NNI choropleth Map
```

```
cities_gdf = gpd.read_file(GEOJSON_CITIES).to_crs(epsg=4326)
parks_gdf  = gpd.read_file(GEOJSON_PARKS).to_crs(epsg=4326)

cities_gdf["polygon_name"] = "City_" + cities_gdf["Geografisc"].asty
parks_gdf["polygon_name"]  = "Park_" + parks_gdf["NAME"].astype(str

cities_plot = cities_gdf.merge(val_df[["polygon_name", "nni", "patt
parks_plot  = parks_gdf.merge( val_df[["polygon_name", "nni", "patt

nni_min = val_df["nni"].min()
nni_max = val_df["nni"].max()
cmap    = plt.cm.RdBu
norm    = Normalize(vmin=nni_min * 0.9, vmax=nni_max * 1.05)

fig, ax = plt.subplots(figsize=(8, 12))
germany_basemap.plot(ax=ax, color="#ece8e3", linewidth=0.5, edgecol

# Polygon fills colour-coded by NNI
cities_plot.plot(column="nni", ax=ax, cmap=cmap, norm=norm,
                 edgecolor="black", linewidth=0.5, zorder=2)
parks_plot.plot(column="nni", ax=ax, cmap=cmap, norm=norm,
                edgecolor="black", linewidth=0.5, zorder=2,
                hatch="///", alpha=0.9)

bbox_props = dict(boxstyle="round,pad=0.25", fc="white", ec="none",

for _, row in cities_plot.dropna(subset=["nni"]).iterrows():
    cx, cy = row.geometry.centroid.x, row.geometry.centroid.y
    ax.annotate(f"{row['nni']:.3f}", (cx, cy),
                ha="center", va="center", fontsize=7, fontweight="b
                color="black", zorder=5, bbox=bbox_props)

for _, row in parks_plot.dropna(subset=["nni"]).iterrows():
    cx, cy = row.geometry.centroid.x, row.geometry.centroid.y
    ax.annotate(f"{row['nni']:.3f}", (cx, cy),
                ha="center", va="center", fontsize=6.5, fontweight="
                color="black", zorder=5, bbox=bbox_props)

sm   = ScalarMappable(cmap=cmap, norm=norm)
sm.set_array([])
cbar = fig.colorbar(sm, ax=ax, fraction=0.028, pad=0.02)
cbar.set_label("NNI", fontsize=9)

city_p = mpatches.Patch(facecolor="gray", edgecolor="white", label="
park_p = mpatches.Patch(facecolor="gray", edgecolor="white", hatch="

ax.legend(handles=[city_p, park_p], loc="upper right", fontsize=9, 
ax.set_title("Validation: NNI per polygon (BallTree, all unique coo
ax.set_xlabel("Longitude"); ax.set_ylabel("Latitude")
plt.tight_layout()
plt.savefig(PATH_OUTPUTS / "validation_nni_map.png", dpi=150, bbox_
plt.show()

print("\nValidation completed.")
print(f"  Comparison plot : {PATH_OUTPUTS / 'validation_nni_compari
print(f"  Map             : {PATH_OUTPUTS / 'validation_nni_map.png
```

```
print(f"  CSV              : {VALIDATION_NNI_CSV}")
```

```
3/3  Create validation plots...
```





```
Validation completed.
  Comparison plot : /Users/maxlengenfelder/Desktop/FS26/GEO877/Proje
ct/GEO877_Flickr_Project/outputs/validation_nni_comparison.png
  Map             : /Users/maxlengenfelder/Desktop/FS26/GEO877/Proje
ct/GEO877_Flickr_Project/outputs/validation_nni_map.png
  CSV             : /Users/maxlengenfelder/Desktop/FS26/GEO877/Proje
ct/GEO877_Flickr_Project/outputs/nni_results_validation.csv
```

In [20]:

```
OUTPUT_MAP_HTML = PATH_OUTPUTS / "flickr_spatial_analysis_map.html"

cities_gdf = gpd.read_file(GEOJSON_CITIES).to_crs(epsg=4326)
parks_gdf = gpd.read_file(GEOJSON_PARKS).to_crs(epsg=4326)

cities_gdf["polygon_name"] = "City_" + cities_gdf["Geografisc"].asty
parks_gdf["polygon_name"] = "Park_" + parks_gdf["NAME"].astype(str)

combined_gdf = pd.concat([cities_gdf, parks_gdf], ignore_index=True
map_gdf = combined_gdf.merge(
    nni_query_results[['polygon_name', 'nni', 'pattern']],
    on='polygon_name',
    how='left'
)

for col in map_gdf.columns:
    if col != 'geometry':
        if pd.api.types.is_datetime64_any_dtype(map_gdf[col]):
            map_gdf[col] = map_gdf[col].astype(str)
        elif map_gdf[col].dtype == 'object':
            map_gdf[col] = map_gdf[col].apply(lambda x: str(x) if i

m = folium.Map(location=[51.1657, 10.4515], zoom_start=6, tiles=Non

folium.TileLayer('cartodbpositron', name='CartoDB Positron').add_to
folium.TileLayer(
    tiles='https://server.arcgisonline.com/ArcGIS/rest/services/Wor
    attr='Esri',
    name='Esri Satellite',
    overlay=False
).add_to(m)

min_nni = map_gdf['nni'].min()
max_nni = map_gdf['nni'].max()
colormap = cm.LinearColormap(colors=['red', 'lightblue', 'blue'], vm
colormap.caption = 'Sampled-Query Nearest Neighbor Index (NNI)'

def style_fn(feature):
    nni_val = feature['properties'].get('nni')
    color = colormap(nni_val) if pd.notna(nni_val) else 'gray'
    return {
        'fillColor': color,
        'color': 'black',
        'weight': 1,
        'fillOpacity': 0.65
    }

poly_layer = folium.FeatureGroup(name='Polygons (by NNI)')
```

```
folium.GeoJson(
    map_gdf,
    style_function=style_fn,
    tooltip=folium.GeoJsonTooltip(
        fields=['polygon_name', 'nni', 'pattern'],
        aliases=['Location:', 'NNI Score:', 'Pattern:']
    )
).add_to(poly_layer)
poly_layer.add_to(m)
m.add_child(colormap)

folium.LayerControl(position='topright').add_to(m)

m.save(str(OUTPUT_MAP_HTML))

print(f"Interactive HTML map successfully saved to: {OUTPUT_MAP_HTM

m
```

```
Interactive HTML map successfully saved to: /Users/maxlengenfelder/D
esktop/FS26/GEO877/Project/GEO877_Flickr_Project/outputs/flickr_spat
ial_analysis_map.html
```

Out[20]: