geoanonymizer Documentation

Release 0.1.0

Stephan Jorek

Oct 05, 2017

Contents

Contents:

CHAPTER 1

geoanonymizer package

Subpackages

geoanonymizer.spatial package

Submodules

geoanonymizer.spatial.filter module

Functions to filter spatial coordinates.

> "One variation on geographic masking is the use of additional spatial filters to ensure masked locations fall within predefined areas of interest. For example, displacement could be limited to a physical land base by excluding surface water bodies (e.g., oceans, bays, rivers, and lakes) to ensure that no masked locations appear in areas which are obviously uninhabited."
>
> —from chapter 7 of Ensuring Confidentiality of Geocoded Health Data: Assessing Geographic Masking Strategies for Individual-Level Data

geoanonymizer.spatial.mask module

Functions to mask spatial coordinates.

> "Geographic masking is the process of altering the coordinates of point location data to limit the risk of reidentification upon release of the data. In effect, the purpose of geographic masking is to make it much more difficult to accurately reverse geocode the released data."
>
> —from chapter 6 of Ensuring Confidentiality of Geocoded Health Data: Assessing Geographic Masking Strategies for Individual-Level Data

Some implementations are inspired by chapter 7 of Ensuring Confidentiality of Geocoded Health Data: Assessing Geographic Masking Strategies for Individual- Level Data.

geoanonymizer.spatial.mask.**add_vector**(*point*, *vector=(None, None, None)*)

Masked points are displaced by a fixed vector, hence we move the point.

If we define a coordinate like ...

```
>>> coordinate = Point(12.3456, 12.3456, 12.3456)
```

... and call this function like ...

```
>>> add_vector(coordinate)
Point(12.3456, 12.3456, 12.3456)
```

```
>>> add_vector(coordinate, (0, 0, 0))
Point(12.3456, 12.3456, 12.3456)
```

```
>>> add_vector(coordinate, (None, None, None))
Point(12.3456, 12.3456, 12.3456)
```

... the given point returns unchanged.

In all other cases the given point will be moved by the given values:

```
>>> add_vector(coordinate, (1.0, 1.0, 1.0))
Point(13.3456, 13.3456, 13.3456)
```

Mind that geodesic points with latitude, longitude, and altitude are used. This results in points with limited value range, hence we rotate the points around the globe:

```
>>> add_vector(coordinate, (100.0, 100.0, 100.0))
Point(-67.65440000000001, 112.3456, 112.3456)
```

```
>>> add_vector(coordinate, (-100.0, -100.0, -100.0))
Point(-87.6544, -87.6544, -87.6544)
```

geoanonymizer.spatial.mask.**circular_bimodal_gaussian_displacement**(*point*, *inner_mu=1.0*, *inner_sigma=1.0*, *outer_mu=2.0*, *outer_sigma=1.0*)

This is a variation on the Gaussian masking technique, employing a bimodal Gaussian distribution for the random distance function. In effect, this approximates donut masking, but with a less uniform probability of placement.

With a given radius, a randomly circular displaced point will return. That implies the altitude always remains untouched. The given coodinates are the circle's center:

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

```
>>> random.seed(1)
>>> circular_bimodal_gaussian_displacement(coordinate)
Point(3.1735160345853632, -0.10110949606778825, 0.0)
```

```
>>> random.seed(1)
>>> circular_bimodal_gaussian_displacement(coordinate,
...                                        1.0, 1.0, 2.0, 1.0)
Point(3.1735160345853632, -0.10110949606778825, 0.0)
```

geoanonymizer.spatial.mask.**circular_gaussian_displacement**(*point*, *mu=1.0*, *sigma=1.0*)

The direction of displacement is random, but the distance follows a Gaussian distribution, where *mu* is the mean and *sigma* is the standard deviation. The dispersion of the distribution can be varied based on other parameters of interest, such as local population density.

With a given radius, a randomly circular displaced point will return. That implies the altitude always remains untouched. The given coodinates are the circle's center:

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

```
>>> random.seed(1)
>>> circular_gaussian_displacement(coordinate)
Point(-2.279620117247094, 0.19779177337662887, 0.0)
```

```
>>> random.seed(1)
>>> circular_gaussian_displacement(coordinate, 1.0, 1.0)
Point(-2.279620117247094, 0.19779177337662887, 0.0)
```

geoanonymizer.spatial.mask.**displace_on_a_circle**(*point*, *radius=0.0*)

Masked points are placed on a random location on a circle around the original location. Masked points are not placed inside the circle itself.

If we define a coordinate like ...

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

... and call this function without any radius ...

```
>>> displace_on_a_circle(coordinate)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_on_a_circle(coordinate, 0)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_on_a_circle(coordinate, None)
Point(0.0, 0.0, 0.0)
```

... the given point returns unchanged.

With a given radius, a randomly circular displaced point will return. That implies the altitude always remains untouched. The given coodinates are the circle's center, the given radius is the distance between given and resulting coodinate:

```
>>> displace_on_a_circle(coordinate, 1)
Point(..., 0.0)
```

```
>>> displace_on_a_circle(coordinate, -1)
Point(..., 0.0)
```

```
>>> random.seed(1)
>>> displace_on_a_circle(coordinate, 1)
Point(0.7474634341555553, 0.6643029539301958, 0.0)
```

geoanonymizer.spatial.mask.**displace_on_a_sphere**(*point*, *radius=0.0*)

Masked points are placed on a random location on a sphere around the original location. Masked points are not placed inside the sphere itself.

If we define a coordinate like ...

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

... and call this function without any radius ...

```
>>> displace_on_a_sphere(coordinate)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_on_a_sphere(coordinate, 0)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_on_a_sphere(coordinate, None)
Point(0.0, 0.0, 0.0)
```

... the given point returns unchanged.

With a given radius, a randomly spherical displaced point will return. The given coodinates are the sphere's center, the given radius is the distance between given and resulting coodinate:

```
>>> displace_on_a_sphere(coordinate, 1)
Point(...)
```

```
>>> displace_on_a_sphere(coordinate, -1)
Point(...)
```

```
>>> random.seed(1)
>>> displace_on_a_sphere(coordinate, 1)
Point(-0.6117158867827159, -0.5436582607079324, 0.5746645712253897)
```

geoanonymizer.spatial.mask.**displace_within_a_circle**(*point*, *radius=0.0*)

Masked locations are placed anywhere within a circular area around the original location. Since every location within the circle is equally likely, masked locations are more likely to be placed at larger distances compared to small distances. A variation on this technique is the use of random direction and random radius. In this technique, masked points are displaced using a vector with random direction and random radius. The radius is constrained by a maximum value. This effectively results in a circular area where masked locations can be placed, but the masked locations are as likely to be at large distances compared to small distances. These two techniques therefore only differ slightly in the probability of how close masked locations are placed to the original locations.

If we define a coordinate like ...

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

... and call this function without any radius ...

```
>>> displace_within_a_circle(coordinate)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_within_a_circle(coordinate, 0)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_within_a_circle(coordinate, None)
Point(0.0, 0.0, 0.0)
```

... the given point returns unchanged.

With a given radius, a randomly circular displaced point will return. That implies the altitude always remains untouched. The given coodinates are the circle's center, the given radius is the maximum distance between given and resulting coodinate:

```
>>> displace_within_a_circle(coordinate, 1)
Point(..., 0.0)
```

```
>>> displace_within_a_circle(coordinate, -1)
Point(..., 0.0)
```

```
>>> random.seed(1)
>>> displace_within_a_circle(coordinate, 1)
Point(-0.10996222555283103, 0.07721437073087664, 0.0)
```

geoanonymizer.spatial.mask.**displace_within_a_circular_donut**(*point*, *radius_inner=0.5*, *radius_outer=1.0*)

This technique is similar to random displacement within a circle, but a smaller internal circle is utilized within which displacement is not allowed. In effect, this sets a minimum and maximum level for the displacement. Masked locations are placed anywhere within the allowable area. A slightly different approach to donut masking is the use of a random direction and two random radii: one for maximum and one for minimum displacement. These two techniques only differ slightly in the probability of how close masked locations are placed to the original locations. Both approaches enforce a minimum amount of displacement.

With a given radius, a randomly circular displaced point will return. That implies the altitude always remains untouched. The given coodinates are the circle's center, the given radii are the minimum and maximum distance between given and resulting coodinate:

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

```
>>> random.seed(1)
>>> displace_within_a_circular_donut(coordinate)
Point(-0.4641756357658914, 0.32593947097813314, 0.0)
```

```
>>> random.seed(1)
>>> displace_within_a_circular_donut(coordinate, 0.5, 1.0)
Point(-0.4641756357658914, 0.32593947097813314, 0.0)
```

geoanonymizer.spatial.mask.**displace_within_a_sphere**(*point*, *radius=0.0*)

Masked locations are placed anywhere within a spherical space around the original location. Since every location within the sphere is equally likely, masked locations are more likely to be placed at larger distances compared to small distances. A variation on this technique is the use of random direction and random radius. In this technique, masked points are displaced using a vector with random direction and random radius. The radius is constrained by a maximum value. This effectively results in a spherical space where masked locations can be placed, but the masked locations are as likely to be at large distances compared to small distances. These two techniques therefore only differ slightly in the probability of how close masked locations are placed to the original locations.

If we define a coordinate like ...

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

... and call this function with any radius ...

```
>>> displace_within_a_sphere(coordinate)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_within_a_sphere(coordinate, 0)
Point(0.0, 0.0, 0.0)
```

```
>>> displace_within_a_sphere(coordinate, None)
Point(0.0, 0.0, 0.0)
```

... the given point returns unchanged.

With a given radius, a randomly spherical displaced point will return. The given coodinates are the sphere's center, the given radius is the maximum distance between given and resulting coodinate:

```
>>> displace_within_a_sphere(coordinate, 1)
Point(...)
```

```
>>> displace_within_a_sphere(coordinate, -1)
Point(...)
```

```
>>> random.seed(1)
>>> displace_within_a_sphere(coordinate, 1)
Point(0.10955063884671598, -0.07692535867829137, 0.011614508874230087)
```

geoanonymizer.spatial.mask.**displace_within_a_spherical_donut**(*point*, *radius_inner=0.5*, *radius_outer=1.0*)

This technique is similar to random displacement within a sphere, but a smaller internal sphere is utilized within which displacement is not allowed. In effect, this sets a minimum and maximum level for the displacement. Masked locations are placed anywhere within the allowable space. A slightly different approach to donut masking is the use of a random direction and two random radii: one for maximum and one for minimum displacement. These two techniques only differ slightly in the probability of how close masked locations are placed to the original locations. Both approaches enforce a minimum amount of displacement.

With a given radius, a randomly spherical displaced point will return. The given coodinates are the sphere's center, the given radii are the minimum and maximum distance between given and resulting coodinate:

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

```
>>> random.seed(1)
>>> displace_within_a_spherical_donut(coordinate)
Point(0.4624382343989833, -0.32471948518229893, 0.049027491156171304)
```

```
>>> random.seed(1)
>>> displace_within_a_spherical_donut(coordinate, 0.5, 1.0)
Point(0.4624382343989833, -0.32471948518229893, 0.049027491156171304)
```

geoanonymizer.spatial.mask.**limit_precision**(*point*, *precisions=(None, None, None)*)

Masked points have a limited precision, hence we cut decimal places.

In the given *precisions* tupel positive integers simply cut decimal places after the comma. Negative integers define the amount decimals to keep before the comma, while cutting all decimal places after the comma. In both cases standard mathematical rounding is applied to the first digit which won't be cut, before cutting the other digits. The first value is the longitude's precision, the second value is the latitude's precision and the third value is the altitude's precision.

If we define a coordinate like ...

```
>>> coordinate = Point(12.3456, 12.3456, 123.456)
```

... and call this function like ...

```
>>> limit_precision(coordinate)
Point(12.3456, 12.3456, 123.456)
```

```
>>> limit_precision(coordinate, (0, 0, 0))
Point(12.3456, 12.3456, 123.456)
```

```
>>> limit_precision(coordinate, (None, None, None))
Point(12.3456, 12.3456, 123.456)
```

... the given point returns unchanged.

If we assign to the second parameter tuple some positive integers the given point will return with altered decimal places after the comma and rounding applied:

```
>>> limit_precision(coordinate, (1, 1, 1))
Point(12.3, 12.3, 123.5)
```

```
>>> limit_precision(coordinate, (2, 2, 2))
Point(12.35, 12.35, 123.46)
```

```
>>> limit_precision(coordinate, (3, 3, 3))
Point(12.346, 12.346, 123.456)
```

Assigning negative integers to the second parameter tuple will return a point with altered decimal places before the comma and all decimal places cut after the comma:

```
>>> limit_precision(coordinate, (-1, -1, -1))
Point(10.0, 10.0, 120.0)
```

```
>>> limit_precision(coordinate, (-2, -2, -2))
Point(10.0, 10.0, 100.0)
```

Mind that decimals with less digits than the given absolute precision, keep the last remaining digit intact.

```
>>> limit_precision(coordinate, (-3, -3, -3))
Point(10.0, 10.0, 100.0)
```

The mathematical rounding can lead to unexpected results, if applied before the comma:

```
>>> limit_precision(Point(65.4321, 65.4321, 654.321), (-1, -1, -1))
Point(70.0, 70.0, 650.0)
```

```
>>> limit_precision(Point(65.4321, 65.4321, 654.321), (-2, -2, -2))
Point(70.0, 70.0, 700.0)
```

geoanonymizer.spatial.mask.**spherical_bimodal_gaussian_displacement**(*point*, *inner_mu=1.0*, *inner_sigma=1.0*, *outer_mu=2.0*, *outer_sigma=1.0*)

This is a variation on the Gaussian masking technique, employing a bimodal Gaussian distribution for the

random distance function. In effect, this approximates donut masking, but with a less uniform probability of placement.

With a given radius, a randomly spherical displaced point will return. The given coodinates are the sphere's center:

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

```
>>> random.seed(1)
>>> spherical_bimodal_gaussian_displacement(coordinate)
Point(0.09101092182341257, -0.002899644539981792, -3.173820372380246)
```

```
>>> random.seed(1)
>>> spherical_bimodal_gaussian_displacement(coordinate,
...                                         1.0, 1.0, 2.0, 1.0)
Point(0.09101092182341257, -0.002899644539981792, -3.173820372380246)
```

geoanonymizer.spatial.mask.**spherical_gaussian_displacement**(*point*, *mu=1.0*, *sigma=1.0*)

The direction of displacement is random, but the distance follows a Gaussian distribution, where *mu* is the mean and *sigma* is the standard deviation. The dispersion of the distribution can be varied based on other parameters of interest, such as local population density.

With a given radius, a randomly spherical displaced point will return. The given coodinates are the sphere's center:

```
>>> coordinate = Point(0.0, 0.0, 0.0)
```

```
>>> random.seed(1)
>>> spherical_gaussian_displacement(coordinate)
Point(-2.278463993019554, 0.19769146198725365, -0.07286551271936394)
```

```
>>> random.seed(1)
>>> spherical_gaussian_displacement(coordinate, 1.0, 1.0)
Point(-2.278463993019554, 0.19769146198725365, -0.07286551271936394)
```

geoanonymizer.spatial.projection module

Functions dealing with geodesic projection systems.

WGS84 (EPSG 4326) projection system

> "OpenStreetMap uses the WGS84 spatial reference system used by the Global Positioning System (GPS). It uses geographic coordinates between -180° and 180° longitude and -90° and 90° latitude. So this is the "native" OSM format.
>
> This is the right choice for you if you need geographical coordinates or want to transform the coordinates into some other spatial reference system or projection."
>
> —from Projections/Spatial reference systems: WGS84 (EPSG 4326)

Mercator (EPSG 3857) projection system

> "Most tiled web maps (such as the standard OSM maps and Google Maps) use this Mercator projection.
>
> The map area of such maps is a square with x and y coordiates both between -20,037,508.34 and 20,037,508.34 meters. As a result data north of about 85.1° and south of about -85.1° latitude can not be shown and has been cut off. . . .

> This is the right choice for you if you are creating tiled web maps."
>
> —from Projections/Spatial reference systems: Mercator (EPSG 3857)

Hint: Apple™ iOS or Google™ Android tracked coordinates use WGS84 (EPSG 4326) projection and nearly all geomap-services, like google-maps, return this too, although they're utilizing Mercator (EPSG 3857) projection internally.

`geoanonymizer.spatial.projection.`**`convert_gps_to_map_coordinates`**(*latitude*, *longitude*)

Convert WGS84 (EPSG 4326) to Mercator (EPSG 3857) projection.

`geoanonymizer.spatial.projection.`**`convert_map_to_gps_coordinates`**(*x*, *y*)

Convert Mercator (EPSG 3857) to WGS84 (EPSG 4326) projection.

geoanonymizer.spatial.shape module

Functions dealing with shapes, especially points and polygons.

`geoanonymizer.spatial.shape.`**`is_on_polygon`**(*x*, *y*, *polygon*, *bounds=None*)

Check if the given *x/y* coordinate is on the given *polygon*. The *bounds* can given be given as *(minx, miny, maxx, maxy)* to speed up the algorithm.

```
>>> polygon = ((0.0, 0.0), (1.0, 0.0), (1.0, 1.0), (0.0, 1.0))
```

```
>>> is_on_polygon(0.0, 0.0, polygon)
True
```

```
>>> is_on_polygon(0.0, 0.5, polygon)
True
```

```
>>> is_on_polygon(0.5, 0.0, polygon)
True
```

```
>>> is_on_polygon(0.5, 0.5, polygon)
True
```

```
>>> is_on_polygon(1.0, 0.5, polygon)
True
```

```
>>> is_on_polygon(0.5, 1.0, polygon)
True
```

```
>>> is_on_polygon(1.0, 1.0, polygon)
True
```

```
>>> is_on_polygon(2.0, 0.0, polygon)
False
```

```
>>> is_on_polygon(0.0, 2.0, polygon)
False
```

```
>>> is_on_polygon(-1.0, -1.0, polygon)
False
```

Beware:

- latitude is *y* and longitude is *x*
- coordinate and *polygon* must use the same geodesic projection system

Module contents

geoanonymizer.trajectory package

Submodules

geoanonymizer.trajectory.TrajectoryPoint module

`TrajectoryPoint` represents points in time.

class `geoanonymizer.trajectory.TrajectoryPoint.`**`TrajectoryPoint`**(*timestamp=None*, *point=None*)

Bases: `object`

Contains a trajectory point, ie. a point in time. Can be iterated over as (timestamp<float>, (latitude<float>, longitude<float>, altitude<float>)). Or one can access the properties *timestamp*, *latitude*, *longitude* or *altitude*.

`altitude`
Location's altitude.

Return type float or None

`latitude`
Location's latitude.

Return type float or None

`longitude`
Location's longitude.

Return type float or None

`point`
`geopy.point.Point` instance representing the location's latitude, longitude, and altitude.

Return type `geopy.point.Point` or None

`timestamp`
Timestamp as a float.

Return type float or None

geoanonymizer.trajectory.permutation module

Implement the two permutation-based methods *SwapLocations* and *ReachLocations* described in Anonymization of trajectory data

`geoanonymizer.trajectory.permutation.`**`permutate_reachable_locations`**`()`
This method takes reachability constraints into account: from a given location, only those locations at a distance below a threshold following a path in an underlying graph (e.g., urban pattern or road network) are considered to be directly reachable. Enforcing such reachability constraints while requiring full trajectory k-anonymity would result in a lot of original locations being discarded. To avoid this, trajectory k-anonymity is changed by another useful privacy definition: location k-diversity.

Computationally, this means that trajectories are not microaggregated into clusters of size k. Instead, each location is k-anonymized independently using the entire set of locations of all trajectories. To do so, a cluster Cλ of "unswapped" locations is created around a given location λ, i.e. λ Cλ. The cluster Cλ is constrained as follows:

1) it must have the lowest intra-cluster distance among those clusters of k "unswapped" locations that contain the location λ;

2.it must have locations belonging to k different trajectories; and

3) it must contain only locations at a path from λ at most Rs long and with time-stamps differing from tλ at most Rt.

Then, the spatial coordinates (xλ,yλ) are swapped with the spatial coordinates of some random location in Cλ and both locations are marked as "swapped". If no cluster Cλ can be found, the location λ is removed from the data set and will not be considered anymore in the subsequent anonymization. This process continues until no more "unswapped" locations appear in the data set.

geoanonymizer.trajectory.permutation.**permutate_swap_locations**(*cardinality=1.0*, **cluster*)

This method needs sets of trajectories as clusters, partitioned using microaggregation. Limit yourself to clustering algorithms which try to minimize the sum of the intra-cluster distances.

The cardinality of each cluster must be approximately k, with k an input parameter, here: *cardinality*; if the number of trajectories in the *cluster* is not a multiple of k, one or more clusters must absorb up to k - 1 remaining trajectories, hence those clusters will have cardinalities between k + 1 and 2k 1. The purpose of setting k as the cluster size is to fulfill trajectory k-anonymity.

The SwapLocations function begins with a random trajectory T in C. The function attempts to cluster each unswapped triple λ in T with another k 1 unswapped triples belonging to different trajectories such that:

1) the timestamps of these triples differ by no more than a time threshold Rt from the timestamp of λ; and

2.the spatial coordinates differ by no more than a space threshold Rs.

If no k 1 suitable triples can be found that can be clustered with λ, then λ is removed; otherwise, random swaps of triples are performed within the formed cluster. As a result, at least one of the trajectories returned by this function has all its triples swapped.

Module contents

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index